O'REILLY®

RESTful Web API Patterns & Practices Cookbook

Connecting and Orchestrating Microservices and Distributed Data

Mike Amundsen
Foreword by Matt McLarty

O'REILLY®

RESTful Web API Patterns & Practices Cookbook

Many organizations today orchestrate and maintain apps that rely on other people's services. Software designers, developers, and architects in those companies often work to coordinate and maintain apps based on existing service APIs, including third-party services that run outside their ecosystem. This cookbook provides proven recipes to help you get those many disparate parts to work together in your network.

Author Mike Amundsen provides step-by-step solutions for finding, connecting, and maintaining applications designed and built by people outside the organization. Whether you're working on human-centric mobile apps or creating high-powered machine-to-machine solutions, this guide shows you the rules, routines, commands, and protocols—the glue—that integrate individual APIs so they can function together in a safe, scalable, and reliable way.

- **Design and build individual service interfaces that can successfully interact on the open web**
- **Increase interoperability by designing APIs that share a common understanding**
- **Build client applications that can adapt to evolving services without breaking**
- **Create resilient and reliable APIs that support peer-to-peer interactions on the web**
- **Use web-based API registries to support runtime "find and bind" operations that manage external dependencies in real time**
- **Implement stable workflows to accomplish complex, multiservice tasks consistently**

> "Amundsen has delivered a valuable resource that addresses common and complex API design choices in an easy-to-understand format. Along the way, he helps the reader tap into the power of the HTTP protocol and hypermedia. Anyone faced with designing evolvable web-based APIs that will stand the test of time should read this book."
>
> **—James Higginbotham**
> Author of *Principles of Web API Design*, Executive API Consultant, LaunchAny

Mike Amundsen is an internationally known author and speaker who travels the world consulting and talking about network architecture, web development, and the intersection of technology and society.

SOFTWARE ARCHITECTURE

US $59.99 CAN $74.99
ISBN: 978-1-098-10674-4

Twitter: @oreillymedia
linkedin.com/company/oreilly-media
youtube.com/oreillymedia

Praise for *RESTful Web API Patterns and Practices Cookbook*

Creating scalable and reliable web APIs and efficiently consuming them is challenging, maybe even more challenging than you think. This cookbook is full of recipes and principles to help you face those known and unknown challenges.

—Arnaud Lauret, API Handyman, OpenAPI Lead, Postman

This book does an excellent job of making good API design more accessible. Look up the challenge you're facing, and you'll find a recipe with a solution and an explanation of how it works.

—Erik Wilde, Catalyst, Axway

Mike Amundsen did it again with excellent best practices and examples of best of breed API patterns. Easy to read, understand, and apply.

—Vicki Reyzelman, Director of Platform and Governance

Amundsen has delivered a valuable resource that addresses common and complex API design choices in an easy-to-understand format. The book contains plenty of examples and diagrams to demonstrate the purpose and application of each recipe. Along the way, he helps the reader tap into the power of the HTTP protocol and hypermedia. Anyone faced with designing evolvable web-based APIs that will stand the test of time should read this book.

—James Higginbotham, Author of Principles of Web API Design, *Executive API Consultant, LaunchAny*

More Books by Mike Amundsen

API Strategy for Decision Makers (2022) with Derric Gilling

Continuous API Management, 2nd ed. (2021) with Medjaoui, Wilde, and Mitra

Design and Build Great Web APIs (2020)

What Is Serverless? (2020)

API Traffic Management 101 (2019)

Continuous API Management, 1st ed. (2018) with Medjaoui, Wilde, and Mitra

RESTful Web Clients (2017)

Microservice Architecture (2016) with Nadareishvili, Mitra, and McLarty

RESTful Web APIs (2013) with Leonard Richardson

Building Hypermedia APIs with HTML5 and Node (2011)

RESTful Web API Patterns and Practices Cookbook

Connecting and Orchestrating Microservices and Distributed Data

Mike Amundsen

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

RESTful Web API Patterns and Practices Cookbook
by Mike Amundsen

Copyright © 2023 Amundsen.com, Inc. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Melissa Duffield
Development Editor: Angela Rufino
Production Editor: Katherine Tozer
Copyeditor: Sonia Saruba
Proofreader: Piper Editorial Consulting, LLC

Indexer: WordCo Indexing Services, Inc.
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

October 2022: First Edition

Revision History for the First Edition
2022-10-14: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781098106744* for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *RESTful Web API Patterns and Practices Cookbook*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the author, and do not represent the publisher's views. While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-098-10674-4

[LSI]

This book is dedicated to the memory of architect and design theorist
Christopher Alexander (1936–2022).
Thanks for helping me see the big picture while I continue to focus on the possible.

Table of Contents

Foreword xi

Preface xiii
About This Book xiii
Conventions Used in This Book xx
Using Code Examples xx
O'Reilly Online Learning xxi
How to Contact Us xxi
Acknowledgments xxii

Part I. Understanding RESTful Hypermedia

1. Introducing RESTful Web APIs 3
What Are RESTful Web APIs? 4
Why Hypermedia? 9
Shared Principles for Scalable Services on the Web 16

2. Thinking and Designing in Hypermedia 19
Establishing a Foundation with Hypermedia Designs 21
Increasing Resilience with Hypermedia Clients 26
Promoting Stability and Modifiability with Hypermedia Services 33
Supporting Distributed Data 40
Empowering Extensibility with Hypermedia Workflow 46

Part II. Hypermedia Recipe Catalog

3. Hypermedia Design. 57
3.1 Creating Interoperability with Registered Media Types 60
3.2 Ensuring Future Compatibility with Structured Media Types 62
3.3 Sharing Domain Specifics via Published Vocabularies 64
3.4 Describing Problem Spaces with Semantic Profiles 68
3.5 Expressing Actions at Runtime with Embedded Hypermedia 73
3.6 Designing Consistent Data Writes with Idempotent Actions 76
3.7 Enabling Interoperability with Inter-Service State Transfers 79
3.8 Designing for Repeatable Actions 84
3.9 Designing for Reversible Actions 87
3.10 Designing for Extensible Messages 91
3.11 Designing for Modifiable Interfaces 95

4. Hypermedia Clients. 101
4.1 Limiting the Use of Hardcoded URLs 103
4.2 Coding Clients to Be HTTP Aware 107
4.3 Coding Resilient Clients with Message-Centric Implementations 110
4.4 Coding Effective Clients to Understand Vocabulary Profiles 113
4.5 Negotiating for Profile Support at Runtime 116
4.6 Managing Representation Formats at Runtime 119
4.7 Using Schema Documents as a Source of Message Metadata 123
4.8 Every Important Element Within a Response Needs an Identifier 128
4.9 Relying on Hypermedia Controls in the Response 132
4.10 Supporting Links and Forms for Nonhypermedia Services 135
4.11 Validating Data Properties at Runtime 139
4.12 Using Document Schemas to Validate Outgoing Messages 141
4.13 Using Document Queries to Validate Incoming Messages 147
4.14 Validating Incoming Data 150
4.15 Maintaining Your Own State 153
4.16 Having a Goal in Mind 156

5. Hypermedia Services. 163
5.1 Publishing at Least One Stable URL 165
5.2 Preventing Internal Model Leaks 167
5.3 Converting Internal Models to External Messages 171
5.4 Expressing Internal Functions as External Actions 176
5.5 Advertising Support for Client Response Preferences 181
5.6 Supporting HTTP Content Negotiation 186
5.7 Publishing Complete Vocabularies for Machine Clients 190

5.8 Supporting Shared Vocabularies in Standard Formats 196
5.9 Publishing Service Definition Documents 202
5.10 Publishing API Metadata 206
5.11 Supporting Service Health Monitoring 210
5.12 Standardizing Error Reporting 216
5.13 Improving Service Discoverability with a Runtime Service Registry 220
5.14 Increasing Throughput with Client-Supplied Identifiers 224
5.15 Improving Reliability with Idempotent Create 228
5.16 Providing Runtime Fallbacks for Dependent Services 232
5.17 Using Semantic Proxies to Access Noncompliant Services 238

6. Distributed Data. 245
6.1 Hiding Your Data Storage Internals 247
6.2 Making All Changes Idempotent 251
6.3 Hiding Data Relationships for External Actions 256
6.4 Leveraging HTTP URLs to Support "Contains" and "AND" Queries 260
6.5 Returning Metadata for Query Responses 264
6.6 Returning HTTP 200 Versus HTTP 400 for Data-Centric Queries 270
6.7 Using Media Types for Data Queries 274
6.8 Ignoring Unknown Data Fields 280
6.9 Improving Performance with Caching Directives 285
6.10 Modifying Data Models in Production 291
6.11 Extending Remote Data Stores 298
6.12 Limiting Large-Scale Responses 302
6.13 Using Pass-Through Proxies for Data Exchange 307

7. Hypermedia Workflow. 315
7.1 Designing Workflow-Compliant Services 317
7.2 Supporting Shared State for Workflows 322
7.3 Describing Workflow as Code 325
7.4 Describing Workflow as DSL 329
7.5 Describing Workflow as Documents 331
7.6 Supporting RESTful Job Control Language 334
7.7 Exposing a Progress Resource for Your Workflows 338
7.8 Returning All Related Actions 342
7.9 Returning Most Recently Used Resources 346
7.10 Supporting Stateful Work in Progress 350
7.11 Enabling Standard List Navigation 358
7.12 Supporting Partial Form Submit 363
7.13 Using State-Watch to Enable Client-Driven Workflow 366
7.14 Optimizing Queries with Stored Replays 375
7.15 Synchronous Reply for Incomplete Work with 202 Accepted 381

7.16 Short-Term Fixes with Automatic Retries 387
7.17 Supporting Local Undo or Rollback 391
7.18 Calling for Help 396
7.19 Scaling Workflow with Queues and Clusters 400
7.20 Using Workflow Proxies to Enlist Noncompliant Services 403

8. Closing Remarks. 407
Applying These Recipes 407
Transforming Existing Services 410
Additional Resources 411
Next Steps 412

A. Guiding Principles of RESTful Web APIs. 415

B. Additional Reading. 417

C. Related Standards. 421

D. Using the HyperCLI. 425

Index. 433

Foreword

Oh, what a tangled web we've woven!

I started my career in technology when the World Wide Web was first being used in the business arena. In fact, the common thread throughout my career has been applying the lessons of the web to the complex world of enterprise architecture and digital business strategy, especially those web APIs that seem to be at the center of everything.

In 2012, I had the pleasure of teaming up with Mike Amundsen. We've partnered on many API-themed endeavors since, and I've never stopped learning from him. It was Mike who taught me the fundamental principles of the web and so much more. It never ceases to amaze me how Mike is able to find insights I overlooked that seem so obvious in hindsight. *RESTful Web API Patterns and Practices Cookbook* puts many of these principles and insights into one very practical package.

When the internet first hit the mainstream back in the early '90s, I was at a small undergraduate school getting a mathematics degree. My professors were the ones most excited at first, which makes sense, I guess, given the web's academic roots. I used to have to pry them away from reading other people's movie reviews to get back to our algebra lessons. In those musty, book-laden campus offices, it wasn't obvious at all that the World Wide Web would become the most transformative human invention of my lifetime, and arguably of all time.

So how did we go from pixelated bulletin boards of personal interests to a world where 70% of business transactions take place digitally? A lot of the credit has to go to the fact that the web was cooked up using a fairly simple set of ingredients: clients, servers, networks, open protocols, and, of course, APIs. No matter how complex and ubiquitous the web gets, those ingredients remain at its core.

Hyperlinks—or *hypermedia*, as a more encompassing term—have had a particularly profound impact on technology. Originally modeled after academic citations (the professors again), not only were hyperlinks the ingredient that brought the network effect to the web, but they have also shaped user expectations for all technology since. There was no user manual for the web, just some blue underlined text that encouraged users to rely on intuition. This "follow your nose" principle is something Mike has always emphasized, and it provides an important perspective when considering APIs and the role of hypermedia.

So the world may not need a user manual for the web, but it certainly needs a developer guide. On this topic, Mike has already authored or coauthored a number of essential books: *RESTful Web APIs*, *RESTful Web Clients*, *Microservice Architecture*, and *Continuous API Management* to name a few. *RESTful Web API Patterns and Practices Cookbook* continues this work. Here, you will not only learn how the technologies of the web are foundational to current computing, but also how the way that the web works is fundamental to scalable architecture in current software ecosystems. Mike parallels the principles of the web with *de rigueur* API design, illustrating how these principles can be applied successfully. Most of all, you will learn how to architect and build systems that are resilient to change and last the test of time.

Every organization is compounding the complexity of their software landscapes on a continual basis, and increasingly connecting with partners and suppliers in a digital ecosystem. If you want the recipes to thrive in that reality, this is the cookbook for you.

—Matt McLarty, Global Field CTO of
MuleSoft at Salesforce Vancouver,
August 2022

Preface

Welcome to the world of the *RESTful Web API Patterns and Practices Cookbook*.

That's quite a moniker—one worth explaining and exploring. And that's what we'll be doing in this preface. I will tell you now that I'm going to break the rules a bit and include a substantial amount of pertinent text in the front matter of this book (front matter is all these pages with roman numerals as page numbers). I'll save the details for the next section (Part I). Let's first take care of some logistics.

About This Book

The goal of this book is to enable software designers, architects, developers, and maintainers to build service interfaces (APIs) that take advantage of the strengths of the web, while lowering the costs and risks of creating reliable high-level services that hold dependencies on other APIs and services reachable only over the network.

To do that, I've gathered a collection of more than 70 recipes and patterns that I've learned and used over the several decades I've spent helping clients design, build, and deploy successful business services on the open web. I suspect you will be familiar with at least some of the recipes you'll find here—possibly by other names or in different forms. I also hope that you will find novel approaches to similar problems.

Over the years, I've found that the challenges of software design rarely change. The *solutions* to those problems change frequently based on technology advances and fashion trends. We'll focus on the challenges in this book, and I'll leave the up-to-date technology and fashion choices to you, the reader.

Since this is a cookbook, there won't be much runnable code. There will, however, be lots of diagrams, code snippets, and network message examples along with explanations identifying the problems. The challenges and discussion will always be technology and platform agnostic. These recipes are presented in a way that will let you

translate them into code and components that will work within your target environment.

Who Should Read This Book

The primary audience for the book is the people tasked with planning, architecting, and implementing service interfaces that run over HTTP. For some, that will mean focusing on creating enterprise-wide service producers and consumers. For others, it will mean building services that can live on the open web and run in a scalable and reliable way for consumers across the globe. For all, it will mean creating usable application programming interfaces that allow programmers to solve the challenges before them.

Whether you are hosting your solutions locally on your own hardware or creating software that will run in the cloud, the recipes here will help you understand the challenges and will offer a set of techniques for anticipating problems and building in recovery to handle cases where the unanticipated occurs.

What's Covered

Since the book is meant to be useful to a wide audience, I've divided it into chapters focused on related topics. To start, Chapters 1 and 2 make up Part I of the book, where we explore the background and foundations of shared services on the web. To stretch the cookbook analogy, consider Part I as the story behind the "hypermedia cusine" we'll be exploring in Part II. Like any good cookbook, each of the main chapters in Part II contains a set of self-contained recipes that you can use to meet particular challenges as you design, build, and deploy your web API "dishes."

Online Resources

The book has a number of associated online resources, including a GitHub repository and related web pages, some examples, and the latest updates to the recipe catalog. You can reach all these resources via *http://WebAPICookbook.com*.

Here is a quick listing of the chapters and what they cover.

Part I: Understanding RESTful Hypermedia

The opening chapters (Chapters 1 and 2) describe the foundation that underpins all the recipes in the book. They are a mix of history, philosophy, and pragmatic thinking. These are the ideas and principles that reflect the lessons I've learned over my years of designing, building, and supporting network software applications running on the web.

Chapter 1, Introducing RESTful Web APIs
: This is a general overview of the rationale behind the selected recipes in this book. It includes a section answering the question "what are RESTful web APIs (RWAs)?," reasons hypermedia plays such an important role in the creation of RWAs, and some base-level shared principles that guide the selection and explanation of the recipes in this book. This chapter "sets the table" for all the material that follows.

Chapter 2, Thinking and Designing in Hypermedia
: This chapter explores the background of hypermedia-driven distributed systems that form the foundation for web applications. Each recipe collection covered in Part II (design, clients, services, data, and workflow) is explored with a mix of history, philosophy, and pragmatic thinking. Reading this chapter will help you understand some of the key design ideas and technical bases for all the patterns and practices outlined in the rest of the book.

Part II: Hypermedia Recipe Catalog

Part II holds all the recipes I've selected for this volume. You'll notice that most of the chapters start with the word "hypermedia." This should give you a clue to the overall approach we'll be taking throughout the book.

Chapter 3, Hypermedia Design
: Reliable and resilient services start with thoughtful designs. This chapter covers a set of common challenges you'll need to deal with before you even get to the level of coding and releasing your services. This chapter will be particularly helpful to architects as well as service designers, and helps set the tone for the various recipes that follow.

Chapter 4, Hypermedia Clients
: This chapter focuses on challenges you'll face when creating service/API consumer applications. I made a point of discussing client apps *before* talking about recipes for service interfaces themselves. A common approach for creating flexible and resilient service consumers is necessary for any program that plans on creating a stable and reliable platform for open services that can live on the web as well as within an enterprise.

Chapter 5, Hypermedia Services
: With a solid foundation of design principles and properly architected client applications, it can be easier to build and release stable service producers that can be safely updated over time without breaking existing API consumers. This set of recipes focuses not only on principles of solid service interface design but also on the importance of supporting runtime error recovery and reliability patterns to make sure your solutions stay up and running even when parts of your system experience failures.

Chapter 6, Distributed Data
: This chapter focuses on the challenges of supporting persisted data in an online, distributed environment. Most of the recipes here are aimed at improving the responsiveness, scalability, and reliability of your data services by ensuring data integrity—even when changing internal data models and implementations at runtime.

Chapter 7, Hypermedia Workflow
: The last set of recipes focuses on creating and managing service workflow on the web. The key challenge to face for open services workflow is to create a safe and reliable set of solutions for enlisting multiple unrelated services into a single, resilient workflow to solve a problem none of the individual services knows anything about. I saved this chapter for last since it relies on many of the recipes covered earlier in the book.

Chapter 8, Closing Remarks
: The final chapter is a short wrap-up of the material as well as a "call-forward" to help you decide on your own "next steps" as you set out to apply these recipes to your environment.

Appendices

There are a series of appendices for the book that you can use as additional support materials. These are sometimes referred to in the text but can also be treated as standalone references.

Appendix A, Guiding Principles
: This appendix is a short "motivational poster" version of the single guiding principle behind the selected recipes, as well as some secondary principles used to shape the description and, ultimately, the implementation of these patterns in general.

Appendix B, Additional Reading
: Throughout the book, I'll be recommending additional reading, quoting from books and articles, and calling out presentations and videos that are the source of much of the advice in the book. This appendix contains a self-standing list of reading and viewing materials that you can use as references and a guide when working through the recipes.

Appendix C, Related Standards
: Since the goal of this book is to create services that can successfully live "on the web," the recipes depend upon a number of important open web standards. This appendix contains a list of the related standards documents.

Appendix D, Using the HyperCLI
: In several places in the book, I reference a command-line interface tool called HyperCLI. You can use this tool to interact with hypermedia-aware services. This appendix provides a short introduction to the tool and some pointers to other online resources on how to take advantage of HyperCLI and HyperLang.

What's Not Covered

As a book of recipes, this text is not suited for teaching the reader *how* to implement the patterns and ideas listed here. If you are new to any of the pillars upon which this book is built, you'll want to look to other sources for assistance.

The following books are some that I have used in training and consulting engagements on topics not covered in detail in this book:

HTTP protocol
: Most of the recipes in this book were developed for HTTP protocol implementations. For more on the power and challenges of HTTP, I recommend the *HTTP Developer's Handbook* by Chris Shiflett (Sams). Shiflett's text has been a great help to me in learning the inside details of the HTTP protocol. Published in 2003, it is still a valuable book that I highly recommend.

API design
: For details on *designing* APIs for distributed services, I suggest readers check out my *Building Hypermedia APIs with HTML5 and Node* (O'Reilly). For those looking for a book focused on coding APIs, my more recent book, *Design and Build Great Web APIs* (Pragmatic Bookshelf), offers a detailed hands-on guide to the full API lifecycle.

API clients
: The work of coding API/service clients is a skill unto itself. For an extended look at the process of creating flexible hypermedia-driven client applications, I refer readers to my *RESTful Web Clients* (O'Reilly).

Web APIs
: For details on creating web APIs themselves, I encourage readers to check out the book *RESTful Web APIs* (O'Reilly), which I coauthored with Leonard Richardson, and my book *Design and Build Great Web APIs* (O'Reilly). Other books I keep close at hand include *Principles of Web API Design* by James Higginbotham (Addison-Wesley) and Arnaud Lauret's *The Design of Web APIs* (Manning).

Data
: For more on handling data at scale, I recommend *Data Management at Scale* by Piethein Strengholt (O'Reilly) and *Data Governance: The Definitive Guide* by Evren Eryurek et al. (O'Reilly).

Workflow
: The books *Practical Process Automation* by Bernd Ruecker (O'Reilly) and *Service Orchestration as Organization* by Malinda Kapuruge et al. (O'Reilly) are a good place to start exploring the world of workflow engineering.

There are many other sources of sage advice on designing and building distributed services, and you'll find a list of suggested reading in Appendix B.

About These Recipes

While the recipes in this cookbook are grouped by topic (design, client, server, data, registry, and workflow), each recipe within the chapters follows the same general pattern:

Problem
: This is a short description of the problem you may run into as you design and build your services.

Solution
: This section is a narrative of the suggested solution (or solutions) you can employ to solve the stated problem.

Example
: In some cases, the recipe will include an example. This might be an HTTP message exchange (request/response) or even a short snippet of pseudocode to show an internal workflow related to the solution.

Discussion
: Recipes will also contain a more lengthy discussion section where trade-offs, downsides, and advantages are covered. Often this is the most important section of the recipe, since very few of these challenges have just one possible solution.

Related Recipes
: Many of the recipes will end with a list of one or more other related recipes covered elsewhere in the book. Some recipes rely on other recipes or enable them, and this is where you'll learn how the recipes interact with each other in actual running systems.

How to Use This Book

I highly recommend reading the book from start to finish to get the full effect of the concepts and recipes contained here. However, I also recognize that time may be short and that you might not need a total immersion experience in order to get the benefits of the book. With this in mind, here are a couple of different ways you can read this book, depending on your focus, goals, and the amount of time you want to devote to the text.

I'm in a hurry

If you recently picked up this book and are looking to solve a pressing problem, just check out the Table of Contents for a recipe that sounds like it fits the bill and jump right in. Like all good recipes, each one is written to be a complete offering. There may be some references to other recipes in the book (especially check out the "Related" subsections), and you can follow up with them as needed.

Getting the "big picture" quickly

If you want to quickly get the big picture, I suggest you read all of Chapters 1 and 2 along with Chapter 8. Part I will give you the "tone" of the collection as well as the history of the recipes and the techniques behind them. From there you can decide whether you want to focus on a particular set in Part II or just roam the collection.

Topic reference for focused teams

If you're part of a team tasked with focusing on one or more of the topics covered here (design, client-side, services, data, workflow, etc.), I suggest you first get the big picture (Part I) and then dive into your particular topic chapter(s) in Part II. You can then use the focus chapters as references as you move ahead with your implementations.

Architect's deep dive

A thorough read, cover to cover, can be helpful if your primary task is architecting openly available producer and consumer services. Many of the recipes in this book can be used to implement a series of enterprise-level approved components that can be safely *stitched together* to form a resilient, reliable foundation for a custom service. In this way, the book can act as a set of recommendations for shareable libraries within a single enterprise.

Checklist for managing enterprise-wide programs

For readers tasked with leading enterprise-wide or other large-scale programs, I suggest getting the big picture first, and then using each topic chapter as a guide for creating your own internal management checklists for creating and releasing RESTful web APIs.

Finally, the book was designed to be a helpful reference as well as a narrative guide. Feel free to use the parts that are helpful to you and skim the sections that don't seem to apply to your situation right now. At some future point, you might find it valuable to go back and (re)read some sections as new challenges arise.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This element signifies a tip or suggestion.

This element signifies a general note.

This element indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at *http://www.webapicookbook.com*.

If you have a technical question or a problem using the code examples, please send email to *bookquestions@oreilly.com*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but generally do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*RESTful Web API Patterns and Practices Cookbook* by Mike Amundsen (O'Reilly). Copyright 2023 Amundsen.com, Inc., 978-1-098-10674-4."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

O'Reilly Online Learning

For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit *https://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *https://oreil.ly/restful-web-api*.

Email *bookquestions@oreilly.com* to comment or ask technical questions about this book.

For news and information about our books and courses, visit *https://oreilly.com*.

Find us on LinkedIn: *https://linkedin.com/company/oreilly-media*.

Follow us on Twitter: *https://twitter.com/oreillymedia*.

Watch us on YouTube: *https://youtube.com/oreillymedia*.

Acknowledgments

> *No one who achieves success does so without acknowledging the help of others. The wise and confident acknowledge this help with gratitude.*
>
> —*Alfred North Whitehead*

So many people have taught me, inspired me, advised me, and encouraged me, that I hesitate to start a list. But several were particularly helpful in the process of writing this book and they deserve notice.

As all of us do, I stand on the shoulders of giants. Over the years many have inspired me, and some of those I've had the pleasure to meet and learn from. Those whose thoughts and advice have shaped this book include Subbu Allamaraju, Belinda Barnet, Tim Berners-Lee, Mel Conway, Roy Fielding, James Gleick, Ted Nelson, Mark Nottingham, Holger Reinhardt, Leonard Richardson, Ian Robinson, and Jim Webber.

I especially want to thank Lorinda Brandon, Alianna Inzana, Ronnie Mitra, Sam Newman, Irakli Nadareishvili, Vicki Reyzelman, and Erik Wilde for their help in reading portions of the text and providing excellent notes and feedback.

I also need to thank all the folks at O'Reilly for their continued support and wise counsel on this project. Specifically, I am deeply indebted to Mike Loukides and Melissa Duffield, who believed in this project long before I was certain about its scope and shape. I also want to say thanks to Angela Rufino for supporting me at every step along the way. Also thanks to Katherine Tozer, Sonia Saruba, and so many others for all the behind-the-scenes work that makes a book like this possible. A special thanks to Kate Dullea and Diogo Lucas for supplying the book's illustrations.

Finally, a big shout-out to all those I've encountered over the years: conference organizers and track chairs, companies large and small that hosted me for talks and consulting, course attendees, and the myriad social media denizens that asked me questions, allowed me to peek into the workings of their organizations, and helped me explore, test, and sharpen the ideas in this book. Everything you see here is due, in large part, to the generosity of all those who came before me and those who work tirelessly each day to build systems that leverage the concepts in Appendix A.

PART I

Understanding RESTful Hypermedia

The difference between the novice and the teacher is simply that the novice has not learnt, yet, how to do things in such a way that they can afford to make small mistakes. The teacher knows that the sequence of their actions will always allow them to cover their mistakes a little further down the line. It is this simple but essential knowledge which gives the work of an experienced carpenter its wonderful, smooth, relaxed, and almost unconcerned simplicity.

—Christopher Alexander

CHAPTER 1
Introducing RESTful Web APIs

> *Leverage global reach to solve problems you haven't thought of for people you have never met.*
>
> —*The RESTful web APIs principle*

In the Preface, I called out the buzzword-y title of this book as a point of interest. Here's where we get to explore the thinking behind *RESTful web APIs* and why I think it is important to both use this kind of naming and grok the meaning behind it.

To start, I'll talk a bit about just what the phrase "RESTful web APIs" means and why I opted for what seems like a buzzword-laden term. Next, we'll spend a bit of time on what I claim is the key driving technology that can power resilient and reliable services on the open web—*hypermedia*. Finally, there's a short section exploring a set of *shared principles* for implementing and using REST-based service interfaces—something that guides the selection and description of the patterns and recipes in this book.

Hypermedia-based implementations rely on three key elements: messages, actions, and vocabularies (see Figure 1-1). In hypermedia-based solutions, messages are passed using common formats like HTML, Collection+JSON, and SIREN. These messages contain content based on a shared domain vocabulary, such as PSD2 for banking, ACORD for insurance, or FIHR for health information. And these same messages include well-defined actions such as `save`, `share`, `approve`, and so forth.

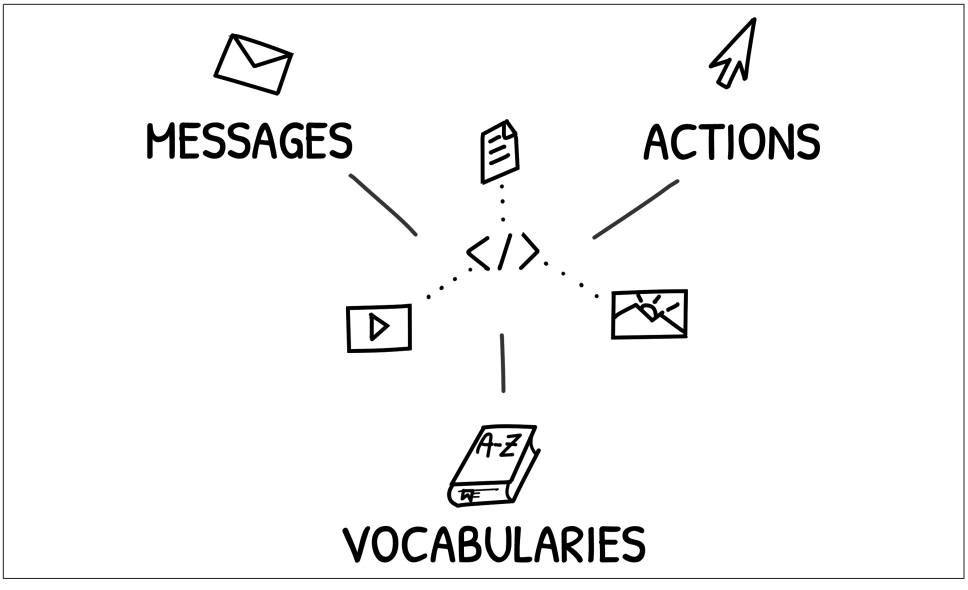

Figure 1-1. Elements of hypermedia

With these three concepts, I hope to engage you in thinking about how we build and use services over HTTP today, and how, with a slight change in perspective and approach, we can update the design and implementation of these services in a way that improves their usability, lowers the cost of creating and accessing them, and increases the ability of both service producers and consumers to build and sustain viable API-led businesses—even when some of the services we depend upon are unreliable or unavailable.

To start, we'll explore the meaning behind the title of the book.

What Are RESTful Web APIs?

I've used the phrase "RESTful web APIs" in articles, presentations, and training materials for several years. My colleague, Leonard Richardson, and I wrote a whole book on the topic in 2013. Sometimes the term generates confusion, even skepticism, but almost always it elicits curiosity. What are these three words doing together? What does the combination of these three ideas mean as a whole? To answer these questions, it can help to take a moment to clarify the meaning of each idea individually.

So, in this section, we'll visit:

Fielding's REST
: The architectural style that emphasizes scalability of component interactions, generality of interfaces, and independent deployment of components.

The web of Tim Berners-Lee
: The World Wide Web was conceived as a universal linked information system, in which generality and portability are paramount.

Alan Kay's extreme late binding
: The design aesthetic that allows you to build systems that you can safely change while they are still up and running.

Fielding's REST

As early as 1998, Roy T. Fielding made a presentation at Microsoft explaining his concept of *Representational State Transfer* (or *REST* as it is now known). In this talk, and his PhD dissertation that followed two years later ("Architectural Styles and the Design of Network-based Software Architectures"), Fielding put forth the idea that there was a unique set of software architectures for network-based implementations, and that one of the six styles he outlined—REST—was particularly suited for the World Wide Web.

Years ago I learned the phrase, "Often cited, never read." That snarky comment seems to apply quite well to Fielding's dissertation from 2000. I encourage everyone working to create or maintain web-based software to take the time to read his dissertation—and not just the infamous Chapter 5, "Representational State Transfer". His categorization of general styles over 20 years ago correctly describes styles that would later be known as gRPC, GraphQL, event-driven, containers, and others.

Fielding's method of identifying desirable system-level properties (like availability, performance, simplicity, modifiability, etc.), as well as a recommended set of constraints (client-server, statelessness, cacheability, etc.) selected to *induce* these properties, is still, more than two decades later, a valuable way to think about and design software that needs to be stable and functional over time.

A good way to sum up Fielding's REST style comes from the dissertation itself:

> REST provides a set of architectural constraints that, when applied as a whole, emphasizes scalability of component interactions, generality of interfaces, independent deployment of components, and intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems.

The recipes included in this book were selected to lead to designing and building services that exhibit many of Fielding's "architectural properties of key interest". The following is a list of Fielding's architectural properties along with a brief summary of their use and meaning:

Performance
: The performance of a network-based solution is bound by physical network limitations (throughput, bandwidth, overhead, etc.) and user-perceived performance, such as request latency and the ability to reduce completion time through parallel requests.

Scalability
: The ability of the architecture to support large numbers of components, or interactions among components, within an active configuration.

Simplicity
: The primary means of inducing simplicity in your solutions is by applying the principle of separation of concerns to the allocation of functionality within components, and the principle of generality of interfaces.

Modifiability
: The ease with which a change can be made to an application architecture via evolvability, extensibility, customizability, configurability, and reusability.

Visibility
: The ability of a component to monitor or mediate the interaction between two other components using things like caches, proxies, and other mediators.

Portability
: The ability to run the same software in different environments, including the ability to safely move code (e.g., JavaScript) as well as data between runtime systems (e.g., Linux, Windows, macOS, etc.)

Reliability
: The degree to which an implementation is susceptible to system-level failures due to the failure of a single component (machine or service) within the network.

The key reason we'll be using many of Fielding's architectural principles in these recipes: they lead to implementations that scale and can be safely modified over long distances of space and time.

The Web of Tim Berners-Lee

Fielding's work relies on the efforts of another pioneer in the the online world, Sir Tim Berners-Lee. More than a decade before Fielding wrote his dissertation, Berners-Lee authored a 16-page document titled "Information Management: A Proposal" (1989 and 1990). In it, he offered a (then) unique solution for improving information storage and retrieval for the CERN physics laboratory where he worked. Berners-Lee called this idea the World Wide Web (see Figure 1-2).

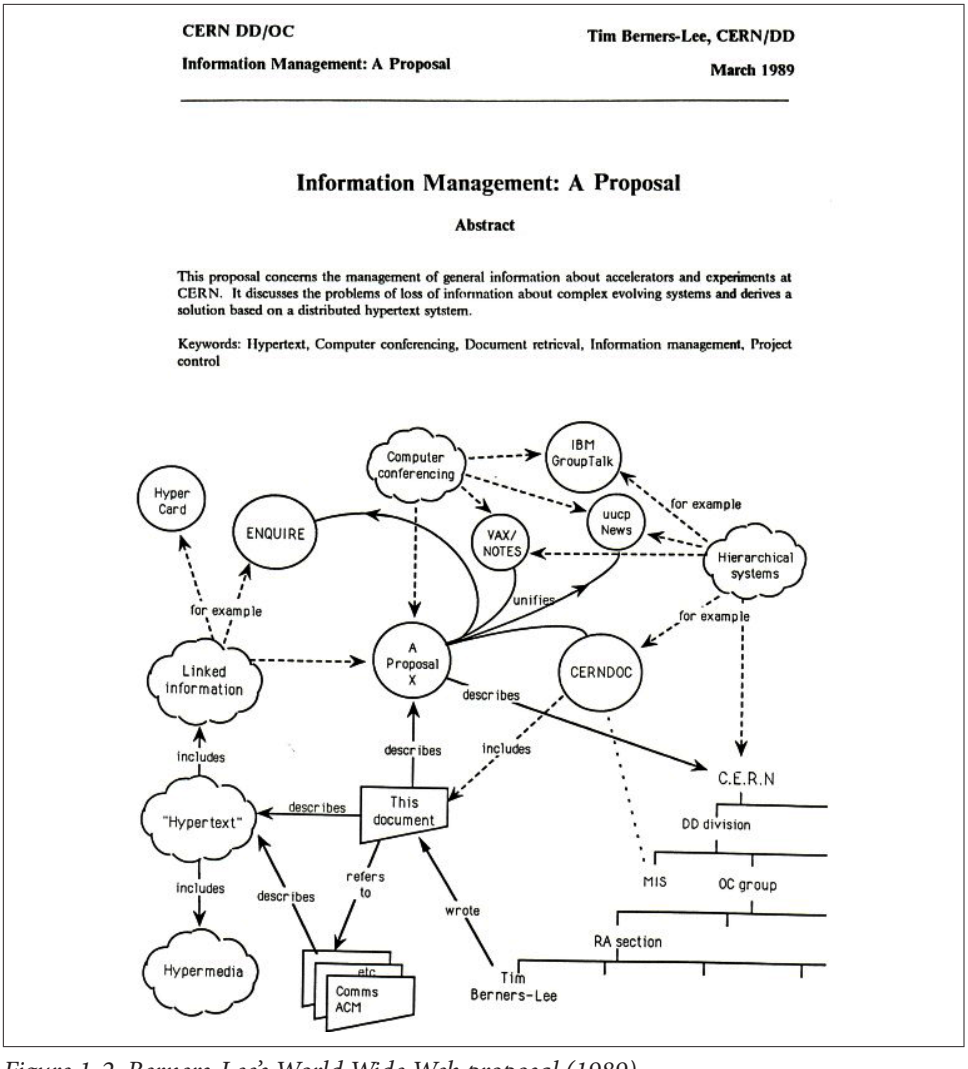

Figure 1-2. Berners-Lee's World Wide Web proposal (1989)

The World Wide Web (WWW) borrowed from the thinking of Ted Nelson, who coined the term *hypertext*, by connecting related documents via *links* and—later—*forms* that could be used to prompt users to enter data that was then sent to servers anywhere in the world. These servers could be quickly and easily set up with free software running on common desktop computers. Fittingly, the design of the WWW followed the "Rule of Least Power," which says that we should use the least powerful technology suitable for the task. In other words, keep the solution as simple as possible (and no simpler). This was later codified in a W3C document of the same name.

This set up a low barrier of entry for anyone who wished to join the WWW community, and helped fuel its explosive popularity in the 1990s and early 2000s.

The Goal of the World Wide Web

In the document that laid out what would later become "the web", Berners-Lee wrote: "We should work toward a universal linked information system, in which generality and portability are [most] important."

On the WWW, any document could be edited to link to (point to) any other document on the web. This could be done without having to make special arrangements at either end of the link. Essentially, people were free to make their own connections, collect their own favorite documents, and author their own content—without the need for permissions from anyone else. All of this content was made possible by using links and forms within pages to create unique pathways and experiences—ones that the original document authors (the ones being connected) knew nothing about.

We'll be using these two aspects of the WWW (the Rule of Least Power and being free to make your own connections) throughout the recipes in this book.

Alan Kay's Extreme Late Binding

Another important aspect of creating reliable, resilient services that can "live on the web" comes from US computer scientist Alan Kay. He is often credited with popularizing the notion of object-oriented programming in the 1990s.

Alan Kay on OOP

When explaining his view of *object-oriented programming* (OOP) on an email list in 2003, Kay stated: "OOP to me means only 1) messaging, 2) local retention and protection and hiding of state-process, and 3) extreme late-binding of all things."

In 2019, Curtis Poe wrote a blog post exploring Kay's explanation of OOP and, among other things, Poe pointed out: "Extreme late-binding is important because Kay argues that it permits you to not commit too early to the *one true way* of solving an issue (and thus makes it easier to change those decisions), but can also allow you to build systems that you can change *while they are still running*!" (emphasis Poe's).

For a more direct exploration of the connections between Roy Fielding's REST and Alan Kay's OOP, see my 2015 article, "The Vision of Kay and Fielding: Growable Systems that Last for Decades".

Just like Kay's view of programming using OOP, the web—the internet itself—is *always running*. Any services we install on a machine attached to the internet are actually changing the system while it is running. That's what we need to keep in mind when we are creating our services for the web.

It is the notion that extreme late binding supports changing systems while they are still running that we will be using as a guiding principle for the recipes in this book.

So, to sum up this section, we'll be:

- Using Fielding's notions of architecting systems to safely scale and modify over time
- Leveraging Berners-Lee's "Rule of Least Power" and the ethos of lowering the barrier of entry to make it easy for anyone to connect to anyone else easily
- Taking advantage of Kay's extreme late binding to make it easier to change parts of the system while it is still running

An important technique we can use to help achieve these goals is called *hypermedia*.

Why Hypermedia?

In my experience, the concept of hypermedia stands at the crossroads of a number of important tools and technologies that have positively shaped our information society. And it can, I think, help us improve the accessibility and usability of services on the web in general.

In this section we'll explore:

- A century of hypermedia
- The value of messages
- The power of vocabularies
- Richardson's magic strings

The history of hypermedia reaches back almost 100 years and it comes up in 20th century writing on psychology, human-computer interactions, and information theory. It powers Berners-Lee's World Wide Web (see "The Web of Tim Berners-Lee" on page 6), and it can power our "web of APIs," too. And that's why it deserves a bit of extended exploration here. First, let's define hypermedia and the notion of hypermedia-driven applications.

Hypermedia: A Definition

Ted Nelson is credited with coining the terms *hypertext* and *hypermedia* as early as the 1950s. He used these terms in his 1965 ACM paper "Complex Information Pro-

cessing: A File Structure for the Complex, the Changing and the Indeterminate". In its initial design, according to Tomas Isakowitz in 2008, a hypertext system "consists of *nodes* that contain information, and of *links*, that represent relationships between the nodes." Hypermedia systems focus on the *connections* between elements of a system.

Essentially, hypermedia provides the ability to link separate nodes, also called *resources*, such as documents, images, services, even snippets of text within a document, to each other. On the network, this connection is made using universal resource identifiers (URIs). When the connection includes the option of passing some data along, these links are expressed as *forms* that can prompt human users or scripted machines to supply inputs, too. HTML, for example, supports links and forms through tags such as `<A>`, `<IMG>`, `<FORM>`, and others. There are several formats that support hypermedia links and forms.

These hypermedia elements can also be returned as part of the request results. The ability to provide links and forms in responses gives client applications the option of selecting and activating those hypermedia elements in order to progress the application along a path. This makes it possible to create a network-based solution that is composed entirely of a series of links and forms (along with returned data) that, when followed, provide a solution to the designed problem (e.g., compute results; retrieve, update, and store data at a remote location; etc.).

Links and forms provide a generality of interfaces (use of hypermedia documents over HTTP, for example) that powers hypermedia-based applications. Hypermedia-based client applications, like the HTML browser, can take advantage of this generality to support a wide range of new applications without ever having their source code modified or updated. We simply browse from one solution to the next by following (or manually typing) links, and use the same installed client application to read the news, update our to-do list, play an online game, etc.

The recipes in this book take advantage of hypermedia-based designs in order to power not just human-driven client applications like HTML browsers, but also machine-drive applications. This is especially helpful for clients that rely on APIs to access services on the network. In Chapter 4, I'll be introducing a command-line application that allows you to quickly script hypermedia-driven client applications without changing the installed client application code base (see Appendix D).

A Century of Hypermedia

The idea of connecting people via information has been around for quite a while. In the 1930s, Belgium's Paul Otlet imagined a machine that would allow people to search and select a custom blend of audio, video, and text content, and view the results from anywhere. It took almost one hundred years, but the streaming revolution finally arrived in the 21st century.

Paul Otlet

Otlet's 1940 view (see Figure 1-3) of how his home machines could connect to various sources of news, entertainment, and information—something he called the "World Wide Network"—looks very much how Ted Nelson (introduced later in this section) and Tim Berners-Lee (see "The Web of Tim Berners-Lee" on page 6) would imagine the connect world, too.

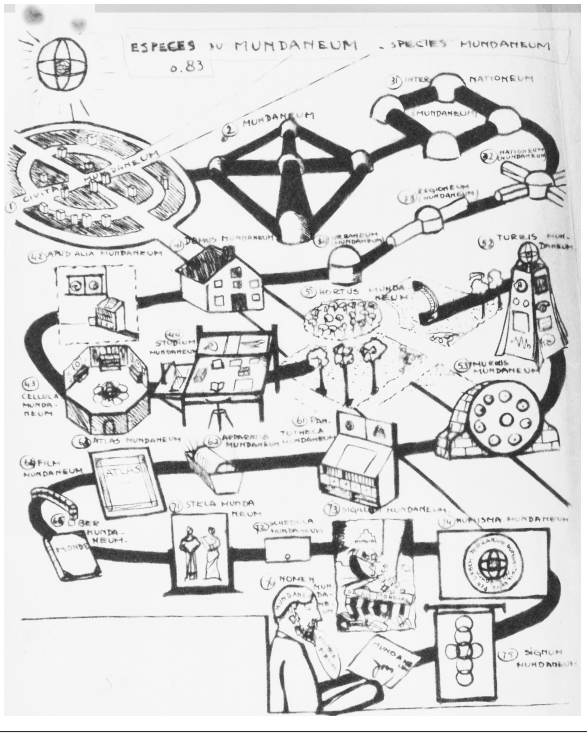

Figure 1-3. Otlet's World Wide Network (1940)

Vannevar Bush

While working as a manager for the Manhattan Project, Vannevar Bush noted that when teams of individuals got together to work out problems in a creative setting, they often bounced ideas off each other, leaping from one research idea to another and making new connections between scientific papers. He wrote up his observations in a July 1945 article, "As We May Think", and described an information workstation similar to Otlet's that relied on microfiche and a "pointing" device mounted on the reader's head.

Douglas Engelbart

Reading that article sparked a junior military officer serving in East Asia to think about how he could make Bush's workstation a reality. It took almost 20 years, but in 1968 that officer, Douglas Engelbart, led a demonstration of what he and his team had been working on in what is now known as "The Mother of All Demos". That session showed off the then unheard of "interactive computer" that allowed the operator to use a pointing device to highlight text and click to follow "a link." Engelbart had to invent the "mouse" pointer to make his demo work.

Mother of All Demos

Engelbart's "Mother of All Demos" over 50 years ago at a December 1968 mainframe convention in San Francisco set the standard for the Silicon Valley demos you see today. Engelbart was alone onstage for 90 minutes, seated in a specially designed Eames chair (the prototype for the Aeron chairs of today), working with his custom-built keyboard, mouse, and a set of "paddles," all while calmly narrating his activity via an over-the-ear microphone that looked like something out of a modern-day Madonna music video. Engelbart showed the first live interactive computer screen, illustrated features like cut-copy-paste, hyperlinking, and multicursor editing, with colleagues hundreds of miles away communicating via picture-in-picture video, version control, and a few other concepts that were still more than a decade away from common use. If you haven't watched the full video, I highly recommend it.

Ted Nelson

A contemporary of Engelbart, Ted Nelson, had been writing about the power of personal computing as early as 1965 using terms he coined, such as *hyperlinks*, *hypertext*, *hyperdata*, and *hypermedia*. By 1974, his book *Computer Lib/Dream Machines* (Tempus Books) laid out a world powered by personal electronic devices connected to each other via the internet. At this same time, Alan Kay (see "Alan Kay's Extreme Late Binding" on page 8) had described the *Dynabook* device that looked very much like the small laptops and tablets of today.

All these early explorations of how information could be linked and shared had a central idea: the connections between things would enable people and power creativity and innovation. By the late 1980s, Tim Berners-Lee had put together a successful system that embodied all the ideas of those who came before him. Berners-Lee's WWW made linking pages of documents safe, easy, and scalable.

This is what using service APIs is all about—defining the connections between things to enable new solutions.

James J. Gibson

Around the same time Ted Nelson was introducing the term *hypertext* to the world, another person was creating terms, too. Psychologist James J. Gibson, writing in his 1966 book *The Senses Considered as Perceptual Systems* (Houghton-Mifflin), on how humans and other animals perceive and interact with the world around them, created the term *affordance*. From Gibson:

> [T]he affordances of the environment are what it offers the animal, what it provides or furnishes.

Gibson's affordances support interaction between animals and the environment in the same way Nelson's hyperlinks allow people to interact with documents on the network. A contemporary of Gibson, Donald Norman, popularized the term *affordance* in his 1988 book *The Design of Everyday Things* (Doubleday). Norman, considered the grandfather of the Human-Computer Interaction (HCI) movement, used the term to identify ways in which software designers can understand and encourage human-computer interaction. Most of what we know about usability of software comes from the work of Norman and others in the field.

Hypermedia depends on affordances. Hypermedia elements (links and forms) are the things within a web response that *afford* additional actions such as searching for existing documents, submitting data to a server for storage, and so forth. Gibson and Norman represent the psychological and social aspects of computer interaction we'll be relying upon in our recipes. For that reason, you'll find many recipes involve using links and forms to enable the modification of application state across multiple services.

The Value of Messages

As we saw earlier in this chapter, Alan Kay saw object-oriented programming as a concept rooted in *passing messages* (see "Alan Kay's Extreme Late Binding" on page 8). Tim Berners-Lee adopted this same point of view when he outlined the message-centric Hypertext Transfer Protocol (HTTP) in 1992 and helped define the message format of Hypertext Markup Language (HTML) the following year.

By creating a protocol and format for passing generalized messages (rather than for passing localized *objects* or *functions*), the future of the web was established. This message-centric approach is easier to constrain, easier to modify over time, and offers a more reliable platform for future enhancements, such as entirely new formats

(XML, JSON, etc.) and modified usage of the protocol (documents, websites, web apps, etc.).

Some Not-So-Successful Examples

HTTP's encapsulated message approach also allowed for "not-so-successful" innovations, like Java Applets, Flash, and XHTML. Even though the HTTP protocol was designed to support things like these "failed" alternatives to message-centric HTML, these alternative formats had only a limited lifetime, and removing them from the ecosystem did not cause any long-term damage to the HTTP protocol. This is a testament to the resilience and flexibility of the HTTP approach to application-level communication.

Message-centric solutions online have parallels in the physical world, too. Insect colonies such as termites and ants, famous for not having any hierarchy or leadership, communicate using a pheromone-based message system. Around the same time that Nelson was talking about hypermedia and Gibson was talking about affordances, American biologist and naturalist E. O. Wilson (along with William Bossert) was writing about ant colonies and their use of pheromones as a way of managing large, complex communities.

With all this in mind, you probably won't be surprised to discover that the recipes in this book all rely on a message-centric approach to passing information between machines.

The Power of Vocabularies

A message-based approach is fine as a platform. But even generic message formats like HTML need to carry meaningful information in an understandable way. In 1998, about the same time that Roy Fielding was crafting his REST approach for network applications (see "Fielding's REST" on page 5), Peter Morville and his colleague Louis Rosenfeld published the book *Information Architecture for the World Wide Web* (O'Reilly). This book is credited with launching the *information architecture* movement. University of Michigan professor Dan Klyn explains information architecture using three key elements: *ontology* (particular meaning), *taxonomy* (arrangement of the parts), and *choreography* (rules for interaction among the parts).

These three things are all part of the vocabulary of network applications. Notably, Tim Berners-Lee, not long after the success of the World Wide Web, turned his attention to the challenge of vocabularies on the web with his Resource Description Framework (RDF) initiatives. RDF and related technologies such as JSON-LD are examples of focusing on meaning within the messages, and we'll be doing that in our recipes, too.

For the purposes of our work, Klyn's choreography is powered by hypermedia links and forms. The data passed between machines via these hypermedia elements is the ontology. Taxonomy is the connections between services on the network that, taken as a whole, create the distributed applications we're trying to create.

Richardson's Magic Strings

One more element worth mentioning here is the use and power of ontologies when you're creating and interacting with services on the web. While it makes sense that all applications need their own coherent, consistent terms (e.g., `givenName`, `familyName`, `voicePhone`, etc.), it is also important to keep in mind that these terms are essentially what Leonard Richardson called "magic strings" in the book *RESTful Web APIs* from 2015.

Closing the Semantic Gap with Magic Strings

Richardson explains the importance of using shared terms across applications in order to close the "semantic gap" of meaning between components. He also points out that, even in cases where you're focused on creating machine-to-machine services, humans are still involved—even if that is only at the programming level. In *RESTful Web APIs*, he says, "Names matter quite a bit to humans. Although computers will be your API's consumers, they'll be working on behalf of human beings, who need to understand what the *magic strings* mean. That's how we bridge the semantic gap" (emphasis mine).

The power of the identifiers used for property names has been recognized for quite some time. The whole RDF movement (see "The Power of Vocabularies" on page 14) was based on creating network-wide understanding of well-defined terms. At the application level, Eric Evans's 2014 book *Domain-Driven Design* (Addison-Wesley) spends a great deal of time explaining the concepts of "ubiquitous language" (used by all team members to connect all the activities within the application) and "bounded context" (a way to break up large application models into coherent subsections where the terms are well understood).

Evans was writing his book around the same time Fielding was completing his dissertation. Both were focusing on how to get and keep stable understanding across large applications. While Evans focused on coherence *within* a single codebase, Fielding was working to achieve the same goals *across* independent codebases.

It is this shared context across separately built and maintained services that is a key factor in the recipes within this book. We're trying to close Richardson's "semantic gap" through the design and implementation of services on the web.

In this section we've explored the hundred-plus years of thought and effort (see "A Century of Hypermedia" on page 10) devoted to using machines to better communicate ideas across a network of services. We saw how social engineering and psychology recognized the power of affordances (see "James J. Gibson" on page 13) as a way of supporting a choice of action within hypermedia messages (see "The Value of Messages" on page 13). Finally, we covered the importance, and power, of well-defined and maintained vocabularies (see "The Power of Vocabularies" on page 14) to enable and support semantic understanding across the network.

These concepts make up a kind of toolkit or set of guidelines for identifying helpful recipes throughout the book. Before diving into the details of each of the patterns, there's one more side trip worth taking. One that provides an overarching, guiding set of principles for all the content here.

Shared Principles for Scalable Services on the Web

To wrap up this introductory chapter, I want to call out some base-level shared principles that acted as a guide when selecting and defining the recipes I included in this book. For this collection, I'll call out a single, umbrella principle:

> Leverage global reach to solve problems you haven't thought of for people you have never met.

We can break this principle down a bit further into its three constituent parts.

Leverage Global Reach…

There are lots of creative people in the world, and millions of them have access to the internet. When we're working to build a service, define a problem space, or implement a solution, there is a wealth of intelligence and creativity within reach through the web. However, too often our service models and implementation tooling limit our reach. It can be very difficult to find what we're looking for and, even in cases where we *do* find a creative solution to our problem by someone else, it can be far too costly and complicated to incorporate that invention into our own work.

For the recipes in this book, I tried to select and describe them in ways that increase the likelihood that others can find your solution, and lower the barrier of entry for using your solution in other projects. That means the design and implementation details emphasize the notions of context-specific vocabularies applied to standardized messages and protocols that are relatively easy to access and implement.

Good recipes increase our global reach: the ability to share our solutions and to find and use the solutions of others.

. . . to Solve Problems You Haven't Thought of . . .

Another important part of our guideline is the idea that we're trying to create services that can be used to build solutions to problems that we haven't yet thought about. That doesn't mean we're trying to create some kind of "generic service" that others can use (e.g., data storage as a service or access control engines). Yes, these are needed, too, but that's not what I'm thinking about here.

To quote Donald Norman (from his 1994 video):

> The value of a well-designed object is when it has such as rich set of affordances that the people who use it can do things with it that the designer never imagined.

I see these recipes as tools in a craftperson's workshop. Whatever work you are doing, it often goes better when you have just the right tool for the job. For this book, I tried to select recipes that can add depth and a bit of satisfaction to your toolkit.

Good recipes make well-designed services available for others to use in ways we hadn't thought of yet.

. . . for People You Have Never Met

Finally, since we're aiming for services that work on the web—a place with global reach—we need to acknowledge that it is possible that we'll never get to meet the people who will be using our services. For this reason, it is important to carefully and explicitly define our service interfaces with coherent and consistent vocabularies. We need to apply Eric Evans's ubiquitous language across services. We need to make it easy for people to understand the intent of the service without having to hear us explain it. Our implementations need to be—to borrow Fielding's phrase—"stateless"; they need to carry with them all the context needed to understand and successfully use the service.

Good recipes make it possible for "strangers" (services and/or people) to safely and successfully interact with each other in order to solve a problem.

Dealing with Timescales

Another consideration we need to keep in mind is that systems have a life of their own and they operate on their own timescales. The internet has been around since the early 1970s. While its essential underlying features have not changed, the internet itself has evolved over time in ways few could have predicted. This is a great illustration of Norman's "well-designed object" notion.

Large-scale systems not only evolve slowly—even the features that are rarely used persist for quite a long time. There are features of the HTML language (e.g., `<marquee>`, `<center>`, `<xmp>`, etc.) that have been deprecated, yet you can still find instances of these language elements online today. It turns out it is hard to get rid of some-

thing once it gets out onto the internet. Things we do today may have long-term effects for years to come.

Design on the Scale of Decades

We can take advantage of long-term timescales in our designs and implementations. Fielding, for example, has said that "REST is software design on the scale of decades: every detail is intended to promote software longevity and independent evolution."

Of course, not all solutions may need to be designed to last for a long time. You may find yourself in a hurry to solve a short-term problem that you assume will not last for long (e.g., a short service to perform a mass update to your product catalog). And that's fine, too. My experience has been, don't assume your creations will always be short-lived.

Good recipes promote longevity and independent evolution on a scale of decades.

This Will All Change

Finally, it is worth saying that, no matter what we do, no matter how much we plot and plan, this will all change. The internet evolved over the decades in unexpected ways. So did the role of the HTTP protocol and the original HTML message format. Software that we might have thought would be around forever is no longer available, and applications that were once thought disposable are still in use today.

Whatever we build—if we build it well—is likely to be used in unexpected ways, by unknown people, to solve as yet unheard-of problems. For those committed to creating network-level software, this is our lot in life: to be surprised (pleasantly or not) by the fate of our efforts.

I've worked on projects that have taken more than 10 years to become noticed and useful. And I've thrown together short-term fixes that have now been running for more than two decades. For me, this is one of the joys of my work. I am constantly surprised, always amazed, and rarely disappointed. Even when things don't go as planned, I can take heart that eventually, all this will change.

Good recipes recognize that nothing is permanent, and things will always change over time.

With all this as a backdrop, let's take some time to more deeply explore the technology and design thinking behind the selected recipes in this book. Let's explore the art of "thinking in hypermedia."

CHAPTER 2
Thinking and Designing in Hypermedia

> *There are no separate systems. The world is a continuum. Where to draw a boundary around a system depends on the purpose of the discussion.*
>
> —*Donella H. Meadows*

Before diving into the recipes in Part II of this book, it's worth spending some time digging into the roots of the web and the ideas behind its design. To that end, this chapter highlights some of the important concepts and technical milestones of network computing. These achievements helped shape the way we interact on the web today and, in some ways, the way we think about computing in general.

When thinking about programming the network, often the focus is on what it takes to program a *machine*. Things like the programming language, use of memory, data storage, and passing properties back and forth through functions are seen as the primary tools. However, when it comes to programming the *network*, new challenges appear, and that means we need new thinking and new tooling, too.

In the following sections, you'll find some historical materials as well as commentary on their application to today's attempts to move beyond stateful, local programming models. In "Establishing a Foundation with Hypermedia Designs" on page 21, you'll find the ideas behind the *design* recipes in this book, including:

- How to establish common communication between machines first discussed in the early 1960s
- The notion of information architecture from the 1990s
- The application of hypermedia as a runtime programming model for independent machines on the network

As shown in Figure 2-1, thinking in Nelson's hypermedia means adopting Roy Fielding's generality of interfaces in order to support Alan Kay's late binding. All the while, we need to think about the importance of scalability and independent deployability in order to build resilient solutions.

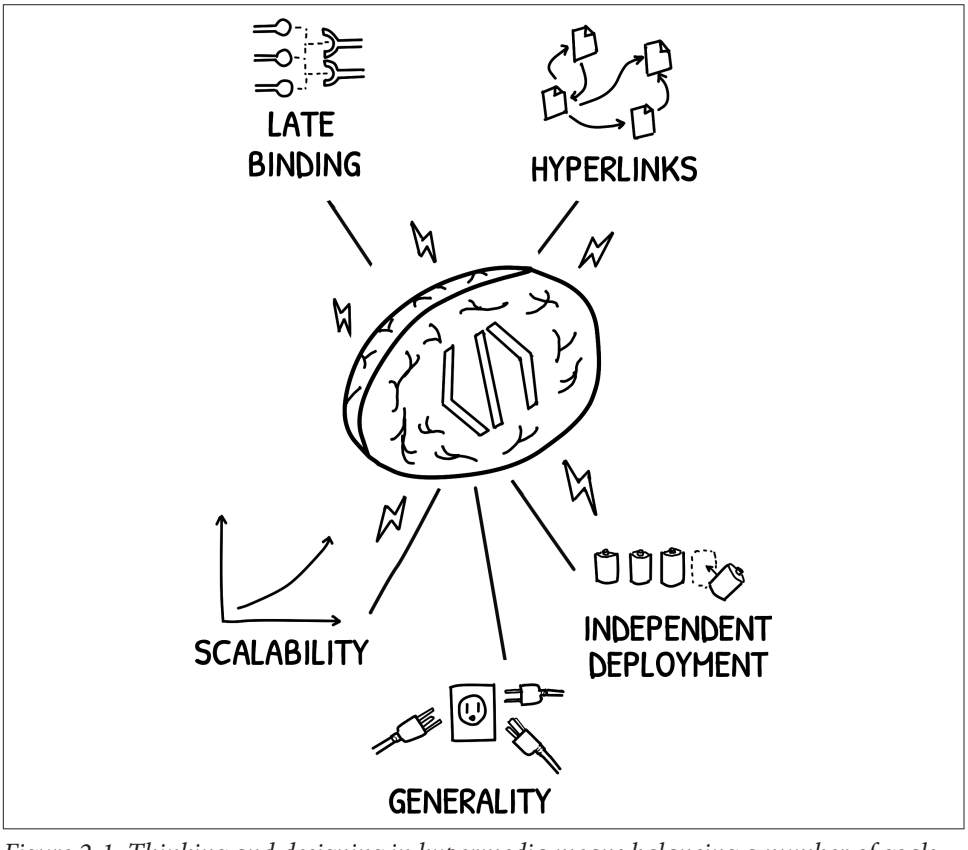

Figure 2-1. Thinking and designing in hypermedia means balancing a number of goals simultaneously

The material in "Increasing Resilience with Hypermedia Clients" on page 26 covers the background behind creating robust *client applications* that can function in a network of services. That means focusing on some important features of API consumers that improve resilience and adaptability; for example, a focus on protocols and formats as the strong typing for network clients, the ability to recognize and react to interaction details in the message at runtime (links and forms), and relying on semantic vocabularies as the understanding shared between clients and services. These three elements make up a set of practices that lead to stable API consumers that do not "break" when service elements, like protocol details, resource URLs, message schema, and operation workflow, change over time.

The key challenge to designing successful *service APIs* is to balance stability with evolvability—the ability to keep your promises to API consumers and support advancement of the capabilities of the services behind your interface. The concepts covered in "Promoting Stability and Modifiability with Hypermedia Services" on page 33 are the keys to meeting this challenge. These include the modifiability problem (the reality of handling change over time) and the need for a machine-driven "self-service" approach to finding and consuming other services in the network. Along the way you'll see how you can apply hypermedia to help solve these problems.

"Supporting Distributed Data" on page 40 introduces the notion that data is *evidence*: evidence of some action as well as the leftover effects of that action. Many attempts to program the network are mistakenly started by thinking that data is at the center of the design. In this section, you'll see that data for the most part is best thought of as *outside* the design—important, but not at the center (I'll be playing the role of Galileo here). We'll also spend time talking about the role of information retrieval query languages (IRQLs) versus database query languages (DQLs) and why it is so important to lean heavily on IRQLs when programming the network.

Finally, in "Empowering Extensibility with Hypermedia Workflow" on page 46, we'll explore the challenges of designing and implementing multiservice *workflows*. In the machine-centric programming world, this is modeled as *orchestration* when a single supervisor is in charge of service integration, or as *choreography* when the services are working closely together directly—without the aid of an intermediary. While both of these approaches make sense in a machine-centric system, enlisting independent services on a network is better served by a general protocol and vocabulary approach—one that relies on using links and forms (hypermedia) to drive a coordinated set of independent actions running at various locations to solve a defined problem. It is in this definition of workflow that the programming of the network is fulfilled.

We'll also explore the challenges for implementing web-based workflows (covered in detail in Chapter 7): sharing state between services, constraining the scope of a single workflow definition (aka a job), supporting workflow observability, and dealing with workflow errors at runtime. That's a lot to cover, so let's get started.

Establishing a Foundation with Hypermedia Designs

The first set of recipes in this book (Chapter 3) focuses on design challenges. There are three general ideas behind the design recipes:

- An agreed communication format to handle connections between networked machines
- A model for interpreting data as information
- A technique for telling machines, at runtime, just what actions are valid

All the recipes in this book are devoted to the idea of making useful connections between application services running on networked machines. The most common way to do that today is through TCP/IP at the packet level and HTTP at the message level.

There's an interesting bit of history behind the way the US Department of Defense initially designed and funded the first machine-to-machine networks (Advanced Research Projects Agency Network or ARPANET), which eventually became the internet we use today. It involves space aliens. In the 1960s, as the US was developing computer communications, the possibility of encountering aliens from outer space drove some of the design choices for communicating between machines.

Along with agreed-on protocols for intermachine communications, the work of organizing and sharing data between machines is another design theme. To do this, we'll dig a bit into information architecture (IA) and learn the value of ontologies, taxonomies, and choreography. The history of IA starts at about the same time that Roy Fielding was developing his REST software architecture style and was heavily influenced by the rise of Berners-Lee's World Wide Web of HTTP and HTML. Also, Chapter 3 uses IA as an organizing factor to guide how we describe service capabilities using a shared vocabulary pattern.

Finally, we'll go directly to the heart of how machines built by different people who have never met each other can successfully interact in real time on an open network —using "hypermedia as the engine of application state." Reliable connections via HTTP and consistent modeling using vocabularies are the prerequisites for interaction, and hypermedia is the technique that enables that interaction. The recipes in Chapter 3 will identify ways to craft hypermedia interactions, while the subsequent chapters will contain specifics on how to make those designs function consistently.

So, let's see how the possibility of aliens from outer space, information architecture, and hypermedia converge to shape the design of RESTful web APIs.

Licklider's Aliens

In 1963, J.C.R. "Lick" Licklider, a little-known civilian working in the US Deparment of Defense, penned an interoffice memo to his colleagues working in what was then called the Advanced Research Projects Agency (ARPA). Within a few years, this group would be responsible for creating the ARPANET—the forerunner of today's internet. However, at this early stage, Licklider addressed his audience as the "Members and Affiliates of the Intergalactic Network". His memo focused on how computing machines could be connected—how they could communicate successfully with one another.

In the memo, Licklider calls out two general ways to ensure computers can work together. One option was to make sure all computers on the planet used the same

languages and programming tools, which would make it easy for machines to connect, but difficult for them to specialize. The second option was to establish a separate, shared network-level control language that allowed machines to use their own preferred local tooling and languages, and then use another shared language to communicate on the network. This second option would allow computer designers to focus on optimizing local functionality, but it would add complexity to the work of programming machines to connect with each other.

In the end (lucky for us!), Licklider and his team decided on the second approach, favoring preferred local machine languages and a separate, shared network-level language. This may seem obvious to us today, but it was not clear at the time. It wasn't just Licklider's decision, but his unique *reasoning* for it that stands out today: the possibility of encountering aliens from outer space. You see, while ARPA was working to bring the age of computing to life, another US agency, NASA, was in a race with the Soviet Union to conquer outer space.

Here's the part of Licklider's memo that brings the 1960s space race and the computing revolution together:

> The problem is essentially the one discussed by science fiction writers: "how do you get communications started among totally uncorrelated 'sapient' beings?"

Licklider was speculating on how our satellites (or our ground-based transmitters) might approach the problem of communicating with other intelligent beings from outer space. He reasoned that we'd accomplish it through a process of negotiated communications—passing control messages or "metamessages" (messages about how we send messages) back and forth until both parties understood the rules of the game. Ten years later, the TCP and IP protocols of the 1970s would mirror Licklider's ideas and form the backbone of the internet we enjoy today.

The Licklider Protocol

Forty years after Licklider speculated about communicating with machines in outer space, members of the Internet Engineering Task Force (IETF) completed work on a transmission protocol for interplanetary communications. This protocol was named the Licklider Transmission Protocol or LTP and is described in IETF documents RFC 5325, RFC 5326, and RFC 5327.

Today, here on Earth, Licklider's thought experiment on how to communicate with aliens is at the heart of making RESTful web APIs (RWAs) a reality. As we work to design and implement services that communicate with each other on the web, we, too, need to adopt a metamessage approach. This is especially important when we consider that one of the aims of our work is to "get communications started among totally uncorrelated" services. In the spirit of our guiding principle (see "Shared Prin-

ciples for Scalable Services on the Web" on page 16), people should be able to confidently design and build services that will be able to talk to other services built by other people they have never met, whether the services were built yesterday, today, or in the future.

Morville's Information Architecture

The 1990s was a heady time for proponents of the internet. Tim Berners-Lee's World Wide Web and HTTP/HTML (see "The Web of Tim Berners-Lee" on page 6), was up and running, Roy Fielding was defining his REST architecture style (see "Fielding's REST" on page 5), and Richard Saul Wurman was coining a new term: *information architect*. In his 1997 book *Information Architects* (Graphis), Wurman offers this definition:

> Information Architect: 1) the individual who organizes the patterns inherent in data, making the complex clear; 2) a person who creates the structure or map of information which allows others to find their personal paths to knowledge; 3) the emerging 21st century professional occupation addressing the needs of the age focused upon clarity, human understanding and the science of the organization of information.

A physical architect by training, Wurman founded the Technology, Entertainment, and Design (TED) conferences in 1984. A prolific writer, he has penned almost a hundred books on all sorts of topics, including art, travel, and (important for our focus) information design. One of the people who picked up on Wurman's notion of architecting information was library scientist Peter Morville. Considered one of the founding fathers of the information architecture movement, Morville has authored several books on the subject. His best known, first released in 1998, is titled simply *Information Architecture for the World Wide Web* (O'Reilly) and is currently in its fourth edition.

Morville's book focuses on how humans interact with information and how to design and build large-scale information systems to best support continued growth, management, and ease of use. He points out that a system with a good information architecture (IA) helps users of that system to understand where they are, what they've found, what else is around them, and what to expect. These are all properties we need for our RWA systems, too. We'll be using recipes that accomplish these same goals for machine-to-machine interactions.

One of the ways we'll organize the IA of RWA implementations is through the use of a three-part modeling approach: ontology, taxonomy, and choreography (see "The Power of Vocabularies" on page 14). Several recipes are devoted to information architecture, including Recipes 3.3, 3.4, and 3.5.

Explaining Information Architecture

Dan Klyn, founder of The Understanding Group (TUG), has a very nice, short video titled "Explaining Information Architecture" that shows how ontology, taxonomy, and choreography all work together to form an information architecture model.

Hypermedia and "A Priori Design"

One of the reasons I started this collection of recipes with the topic of "design" is that the act of designing your information system establishes some rules from the very start. Just as the guiding principles (see "Shared Principles for Scalable Services on the Web" on page 16) we discussed in Chapter 1 establish a foundation for making decisions about information systems, design recipes make that foundation a reality. It is this first set of recipes in Chapter 3 that affect, and in many ways govern, all the recipes in the rest of the book.

In this way, setting out these first recipes is a kind of an "*a priori* design" approach. One of the definitions of *a priori* from the Merriam-Webster dictionary is "formed or conceived beforehand," and that is what we are doing here. We are setting out elements of our systems beforehand. There is an advantage to adopting this *a priori* design approach. It allows us to define stable elements of the system upon which we can build the services and implement their interaction.

Creating a design approach means we need a model that works for more than a single solution. For example, an approach that only works for content management systems (CMSs) but not for customer relationship management systems (CRMs) is not a very useful design approach. We intuitively know that these two very different solutions share quite a bit in common (both at the design and the technical solution level), but it often takes some work to tease out those similarities into a coherent set—a set of design principles.

This can be especially challenging when we want to create solutions that can change over time. Solutions that remain stable while new features are added, new technology solutions are implemented, and additional resources like servers and client apps are created to interact with the system over time. What we need is a foundational design element that provides stability while supporting change.

In this set of designs, that foundational element is the use of hypermedia, or links and forms, (see "Why Hypermedia?" on page 9) as the device for enabling communications between services. Fielding called hypermedia "the engine of application state". Hypermedia provides that metamessaging Licklider identified (see "Licklider's Aliens" on page 22). It is the use of hypermedia that enables Kay's "extreme late binding" (see "Alan Kay's Extreme Late Binding" on page 8).

Increasing Resilience with Hypermedia Clients

Since computers, as Ted Nelson tells us, "do what you tell them to do," we have a responsibility to pay close attention to what we tell them. In Chapter 4, we'll focus on what we tell API consumers (client applications). There is a tendency to be very explicit when telling computers what to do, and that's generally a good thing. This is especially true when creating API-driven services (see Chapter 5). The more accurate our instructions, the more likely it is that the service will act in ways we expect. But client applications operate in a different way. And that's the focus of these client recipes.

While API-based services need to be stable and predictable, API client applications need to excel at being adaptable and resilient. Client applications exist to accomplish a task; they have a purpose. As we'll discuss, it is important to be clear about just what that purpose is—and how explicit we want to be when creating the API consumer.

A highly detailed set of instructions for an API client will make it quite effective for its stated job. But it will also render the client API unusable for almost any other task. And, if the target service for which it was designed changes in any meaningful way, that same client application will be "broken." It turns out that the more detailed the solution, the less reusable it becomes. Conversely, if you want to be able to reuse API consumers, you need to change the way you implement them.

Balancing usability (the ease of use for an API) and reusability (the ease of using the same API for another task) is tricky. Abstraction improves reuse. The HTTP protocol is rather abstract (a URL, a set of methods, collection of name-value pairs, and a possible message body in one of many possible formats), and that makes it very reusable. But the HTTP protocol itself is not very usable without lots of supporting technologies, standards, and existing tooling (e.g., web servers and web browsers).

The recipes in Chapter 4 are aimed at increasing the resilience of client applications. That means focusing on some important features of API consumers that improve resilience and adaptability. These are:

- A focus on protocols and formats
- Resolving interaction details at runtime
- Designing differently for machine-to-machine (M2M) interactions
- Relying on a semantic vocabulary shared between client and server

These four elements make up a set of practices that lead to stable API consumers that do not "break" when service elements like protocol details, resource URLs, message

schema, and operation workflow change over time. All the client recipes focus on these four elements and the resulting stability and resilience they bring to your client applications.

Let's cover each of them in turn.

Binding to Protocols and Formats

An important element to building successful hypermedia-enabled client applications is the work of *binding* the client to responses. Although programmers might not think about it, whenever we write an API consumer app, we're creating a binding between producers (services) and consumers (clients). Whatever we use as our "binding agent" is the thing that both clients and servers share. The most effective bindings are the ones that rarely, if ever, change over time. The binding we're talking about here is the actual expression of the "shared understanding" between clients and services.

Common binding targets are things like URLs (e.g., */persons/123*) or objects (e.g., `{id:"123", {person:{...}}}`). There are, for example, lots of frameworks and generators that use these two binding agents (URLs and objects) to automatically generate static code for a client application that will work with a target service. This turns out to be a great way to quickly deploy a working client application. It also turns out to be an application that is hard to reuse and easy to break. For example, any changes in the API's storage objects will break the API consumer application. Also, even if there is an identical service (one that has the same interface) running at a different URL, the generated client is not likely to successfully interact since the URLs are not the same. URLs and object schema are not good binding agents for long-term use/reuse.

A much better binding target for web APIs is the protocol (e.g., HTTP, MQTT, etc.) and the message format (e.g., HTML, Collection+JSON, etc.). These are much more stable than URLs and objects. They are, in fact, the higher abstract of each. That is to say, protocol is the higher abstraction of URLs, and message formats (or media types on the web) are the higher abstraction of object schema. Because they are more universal and less likely to change, protocol and format make for good binding agents. Check out Recipes 4.3 and 4.6 for details.

For example, if a client application is bound to a format (like Collection+JSON), then that client application can be successfully used with any service (at any URL) that supports Collection+JSON bindings. This is what HTML web browsers have been doing for more than 30 years.

But protocol and format are just the start of a solid foundation of shared understanding. The other keys to stable, reliable API consumer applications include runtime support metadata, semantic profiles, and client-centric workflows.

Runtime Resolution with Metadata

One of the challenges of creating flexible API clients is dealing with all the details of each HTTP request: which HTTP method to use, what parameters to pass on the URL, what data is sent in request bodies, and how to deal with additional metadata like HTTP headers. That's quite a bit for information to keep track of, and it is especially tedious when you need to handle this metadata for each and every HTTP request.

The typical way to deal with request metadata is to "bake" it into the service interface. Documentation usually instructs programmers how to approach a single action for the API, like adding a new record, using instructions like this:

- Use `POST` with the */persons/* URL.
- Pass at least four (`givenName`, `familyName`, `telephone`, and `email`) parameters in the request body.
- Use the `application/x-www-form-urlencoded` serialization format for request bodies.
- Expect an HTTP status code of `201` upon successful completion, and a `Location` header indicating the URL of the new record.

The example supplied here is actually a summary of a much more detailed entry in most documentation I encounter. The good news is that most web programmers have internalized this kind of information and don't find it too off-putting. The not-so-good news is that writing all this out in code is falling into the trap of the wrong "binding agent" mentioned. Any changes to the URL or the object/parameters will render the client application "broken" and in need of an update. And, especially early in the lifecycle of an API/service, changes will happen with annoying frequency.

The way to avoid this is to program the client application to recognize and honor these request details in the metadata sent to the client at runtime. We'll cover this in Recipe 4.9. Again, HTML has been doing this kind of work for decades. The following is an example of the same information as runtime metadata:

```
<form action="/persons/" method="post",
  enc-type="application/x-www-form-urlencoded">
  <input type="text" name="givenName" value="" required />
  <input type="text" name="familyName" value="" required />
  <input type="tel" name="telephone" value="" required />
  <input type="email" name="email" value="" required />
  <input type="submit" />
</form>
```

As you have already surmised, the HTML web browser has been programmed to recognize and honor the metadata sent to the client. Yes, there is programming

involved—the one-time work of supporting `FORMS` in messages—but the good news is you only need to write that code once.

I've used HTML as the runtime metadata example here, but there are a handful of JSON-based formats that have rich support for runtime metadata. The ones I commonly encounter are Collection +JSON, SIREN, and UBER.

Support for runtime metadata can make writing human-to-machine applications pretty easy. There are libraries that support parsing hypermedia formats into human-readable user interfaces, very similar to the browsers that do this for HTML. But supporting runtime metadata for M2M interaction is more complicated. That's because the human brain is missing from the equation.

To support stable, reliable M2M interactions, we need to make up for the missing human in the interaction. That's where semantic profiles come in.

Machine-to-Machine Challenges

Our brains are amazing. So amazing that we often don't even notice how much "magic" they are doing for us. A fine example of this can be seen in the act of filling in a simple HTML form (like the one we just covered). Once our eyes scan the page (magical enough!), our brain needs to handle quite a few things:

- Recognize there is a `FORM` that can be filled in and submitted
- Work out that there are four inputs to supply
- Recognize the meaning of `givenName` and the other values
- Scour memory or some other source to find values to fill in all the inputs
- Know that one needs to press the Submit button in order to send the data to the server for processing

We also need to be able to deal with things like error messages if we don't fill in all the inputs (or do that work incorrectly), and any response from the server such as "unable to save data" or some other strings that might require the human to take further action.

When we write human-to-machine API clients, we can just assume that all that mental power is available to the client application "for free"—we don't need to program it in at all. However, for M2M applications, that "free" intelligence is missing. Instead we need to either build the power of a human into our app, or come up with a way to make the interactions work without the need for a human mind running within the application.

If you want to spend your time programming machine learning and artificial intelligence, you can take option one. In this book, we'll be working on option two instead. The client recipes will be geared toward supporting "limited intelligence" in the client application. We can do this by leaning on media types to handle things like recognizing a `FORM` and its metadata. We'll also be leveraging semantic profiles as a way of dealing with the parameters that might appear within a `FORM` and how to associate these parameters with locally (client-side) available data to fill in the parameter values. We'll also talk about how to modify the service workflow support to make it easier for M2M clients to safely interact with services (see Chapter 5 for more on this).

Relying on Semantic Vocabularies

To date, the most successful M2M interactions on the web have been those that require only *reading* data—not writing it. Web spiders, search bots, and similar solutions are good examples of this. Some of this has to do with the challenge of idempotence and safety (see Recipe 3.6 for answers to this challenge).

Another big part of the M2M challenge has to do with the data properties for individual requests. Humans have a wealth of data at their beck and call that machines usually do not. Adding a new field to a `FORM` is usually not a big challenge for a human tasked with filling it out. But it can be a "breaking change" for an M2M client application. Getting past this hurdle takes some effort on both ends of the interaction (client and server).

An effective way to meet this challenge is to rely upon semantic profiles (see Recipe 3.4)—documents that detail all the possible property names and action details (links and forms) used for a set of problems (e.g., account management, payment services, etc.) to set boundaries on the vocabulary that a client application is expected to understand. In other words, the client and server can agree ahead of time on which data properties will be needed to successfully interact with the application. You'll see this in Recipe 4.4.

By using semantic profiles to establish the boundaries of a service ahead of time—and by promising to keep that boundary stable—we get another important "binding agent" that works especially well for M2M interactions. Now we can use protocol, format, and profile as three stable, yet flexible, binding elements for client-server interactions.

There is one more important element to building successful RWA clients—the ability for clients to author and follow their *own* multiservice workflows instead of being tied to a single service or bound by a static, prebuilt interactive script.

Supporting Client-Centric Workflows

Most API client applications are statically tied to a single API service. These clients are essentially one-off and custom-built. One of the fundamental ways these apps are linked to a service is expressed in the *workflow* implementation. Just as there are applications that use URLs and object schema as binding agents, there are also applications that use sequential workflow as a binding agent. The client application knows only one set of possible workflows, and the details of that set do not change.

This statically bound workflow is often expressed in client code directly. For example, a service named `customerOnboarding` might offer the following URLs (with object schema to match):

- `/onboarding/customer` with schema `{ customer: {…}}`
- `/onboarding/contact` with schema `{ contact: {…}}`
- `/onboarding/agreement` with schema `{ agreement: {…}}`
- `/onboarding/review` with schema `{ review: {…}}`

For the service, there are four defined steps that are executed in sequence. That sequence is often outlined, not in the service code, but in the service *documentation*. That means it is up to the client application to convert the human-speak in the documentation into machine-speak in the code. It usually looks something like this:

```
function onboardCustomer(customer, contact, agreement, review) {
  http.send("/onboarding/customer","POST", customer);
  http.send("/onboarding/contact", "POST", contact);
  http.send("/onboarding/agreement", "POST", agreement);
  http.send("/onboarding/review", "POST", review);
  return "200 OK";
}
```

The first challenge in this example is that the static binding means any change to service workflow (e.g., adding a `creditCheck` step) will mean the client app is "broken." A better approach is to tell the client what work needs to be done, and provide the client application the ability chose and execute steps as they appear. We can use hypermedia in responses to solve that problem:

```
function onboardCustomer() {
  results = http.read("/onboarding/work-in-progress", "GET");
  while(results.actions) {
    var action = results.actions.pop();
    http.send(action.url, action.method, map(action.parameters,local.data));
  }
  return "200 OK";
}
```

In this simplified example, the client "asks" the service for a list of actions to take and then performs those actions using the runtime metadata (see "Runtime Resolution with Metadata" on page 28) in the message. The client continues this until there are no more actions to complete. There are other variations on this pattern that we'll explore (see Recipe 4.9). In the second example, when the order changes or even the number of steps changes, it is less likely that this client implementation will break.

There is one more version of this client-centric approach worth noting—where the *client* sets the workflow, not the service. This is a common case when the client needs to enlist several separate services to complete a task. Consider the previous example, but this time assume that the workflow relies on an external contact management cloud service and an independent credit-checking service. These two external services need to be mixed with your own company's standalone `customer`, `agreement`, and `review` services. In this world, services don't know about each other. They don't control the workflow; the client application does. In fact, the true "application" exists only as a client implementation.

A pseudocode for this scenario looks similar to the previous one but includes a client-side version of the set of steps to execute, written out in a `choreography` configuration document that contains a set of starting URLs and possible arguments for each step:

```
function onboardCustomer(choreography) {
  while(choreography) {
    var step = choreography.pop();
    var action = http.get(step.URL);
    http.send(action.url, action.method, map(action.parameters,local.data));
  }
  return "200 OK";
}
```

In this latest example, the *source* of the choreography comes from the client application. It is also up to the client application to decide when it has completed its work. This is a kind of client-centric goal setting (see Recipe 4.16) that is not very hard to support when you already have a foundation of stable, flexible binding agents relying on protocols, formats, and profiles.

There is another approach to client-centric workflow that relies on a particular status value (or set of values) on one or more servers. See Recipe 7.13 for an example of this option.

Promoting Stability and Modifiability with Hypermedia Services

The key challenge to designing successful service APIs is to balance stability with evolvability—the ability to keep your promises to API consumers and support advancement of the capabilities of the services behind your interface. All the recipes for the services chapter (Chapter 5) were selected to help with this challenge.

At the root of the problem is the reality of change over time. The longer a service stays in production, the more likely that the service will change. On the opposite end of the spectrum, short-lived, one-off services rarely "live" long enough to grow or change. They pop up, do their job, and disappear. The architectural element of *time* causes us to face the realities of change.

Taking a hypermedia approach to designing service interfaces gives us some handy tools for supporting change over time while still providing stable, predictable interfaces. By establishing hypermedia as part of your message design, your interface can set up "safe zones" of modifiability. The hypermedia controls (links and forms) provide a vector for supporting change. For example, by coding an API consumer to parse hypermedia forms, services are free to change interaction details such as URLs and HTTP methods, and even message formats and parameters. In this way, hypermedia does what Carnegie Mellon University professor and author Paul Clements advises us good software architecture does: it "*knows* what changes often and makes that easy."

But using hypermedia in your messages is not enough. Services often need to talk to *other services*. After all, we're participating in a network. When one service depends on another, we run the risk of creating a fatal dependency chain—one where changes in one service in the stack bubble up to all the other services that rely upon it. And when that happens, unexpected things can happen. Sometimes changes in one of the *called* services result in changes to the interface. Sometimes those changes might be incompatible with the task of the *calling* service—a breaking change is exposed. We need a way to account for, and resolve, the dependency problem, too.

The most direct way to survive problems with service dependencies is to eliminate them. But that's usually an unrealistic goal. We rely on other services for a reason—they provide something we need and don't have ourselves. Most often today, the next best response to broken services dependencies is for a human to find *another* service that provides the same functionality. Usually this means that 1) services break and notifications are sent, 2) humans react to the notification and work up a solution (typically finding a replacement for the broken services), and 3) the updated component is placed into production.

A better approach (that is implemented within closed systems like Kubernetes) automates that process. When services are released, their metadata is entered into a registry where other authorized services can find them and instantly start a "real-time integration" process that results in limited (if any) downtime.

Some of the recipes in Chapter 5 address this self-driven (re)location and integration activity by using standardization and automation. We'll explore this in greater detail in Chapter 7 when we talk about multiservice workflows.

The Modifiability Problem

Frankly, it's not very difficult to create a service interface—a web API. Proof of this fact is the long list of tools that read database schema and/or scan existing codebases and output HTTP APIs, ready to deploy into production. The actual production of the interface is easy. But the work of designing a well-crafted, long-lasting API is not so easy. A key element in all this is *time*. Services that live a long time experience updates—modifications—that threaten the simplicity and reliability of the API that connects that service to everything else. Time is at the heart of the modifiability problem.

As long as your service never changes, your API design and implementation needs can be minimal. This is true, for example, with APIs that expose mainframe or minicomputer based services that have been running for decades. It is unlikely they will change, and therefore it's pretty easy to create a stable interface in front of those services. That's where schema-based and object-based API tooling shines.

But there are fewer and fewer of these kinds of long-running services in the world today. More often, APIs need to connect to services that were recently created—services that are just a few years old. Many times, these services were designed to meet immediate needs and, as time goes on, those needs evolve. The easy route is to create a "new" interface, slap an identifier on it (e.g., "/v2/"), and republish. There are many companies that do that today.

The good ones keep the old versions around for a long time, too. That makes it possible for client applications to manage their own upgrade path on their own timeline. But many companies retire the old version too soon. That puts pressure on API consumers to update their own code more quickly to keep up with the pace of change on the service end. When you consider that some client apps are consuming more than one API, each on its own update schedule, it is possible that an API consumer app is always dealing with some dependency update challenge. That means that API-driven applications could be in a constant state of disruption.

A better approach is to adopt a pattern that allows services to evolve without causing client applications that use them to experience breakages or disruption. What is

needed is an API design approach that supports both service evolvability and interface stability. Here is a set of principles that can lead us to stable, evolvable interfaces:

- Adopting the "Hippocratic Oath of APIs"
- Committing to the "Don't Change It, Add It" rule
- Taking the "APIs Are Forever" point of view

The Hippocratic Oath of APIs

One way to address the modifiability problem is to pledge to never "break" the interface—the promise to maintain compatibility as the interface changes. This is a kind of Hippocratic Oath of APIs. Written between the fifth and third centuries BCE, the Hippocratic Oath is an ethical promise Greek physicians were expected to follow. The most well-known portion of the oath is the line "I will abstain from all intentional wrong-doing and harm". This is often rephrased to "First, do no harm."

When creating interfaces, it is important to commit—from the start—to "do no harm." That means the only kinds of modifications you can make are ones that don't break promises. Here are three simple rules you can use to keep this "no breaking changes promise":

Take nothing away.
: Once you document and publish an endpoint URL, support for a protocol or media type, a response message, an input parameter or output value, you cannot remove them in a subsequent interface update. You may be able to set the value to `null` or ignore the value in future releases, but you cannot take it away.

Don't redefine things.
: You cannot change the meaning or use of an existing element of the interface. For example, if the response message was published as a user object, you cannot change it to a user collection. If the output property `count` was documented as containing the number of records in the collection, you cannot change its meaning to the number of records on a single page response.

Make additions optional.
: After the initial publication of the API, any additional inputs or outputs must be documented as *optional*—you cannot add new *required* properties for existing interfaces. That means, for example, that you cannot add a new input property (`backup_email_address`) to the interface action that creates a new user record.

Don't change it, add it

By following these three rules, you can avoid the most common breaking changes for APIs. However, there are a few other details. First, you can always create *new* endpoints (URLs) where you can set the rules anew on what inputs and outputs are

expected. The ability to just "add more options" is a valuable way to support service evolvability without changes to existing elements.

For example, you may have the opportunity to create a new, improved way to return filtered content. The initial API call did not support the ability to modify the number of records returned in a response; it was always fixed at a max of one hundred. But the service now supports changing the `page_size` of a response. That means you can offer two interface options that might look like this in the SIREN media type:

```
"actions": [
    {
      "name": "filter-list",
      "title": "Filter User List",
      "method": "GET",
      "href": "http://api.example.org/users/filter"
      "type": "application/x-www-form-urlencoded",
      "fields": [
        { "name": "region", "type": "text", "value": ""},
        { "name": "last-name", "type": "text", "value": "" }
      ]
    },
    {
      "name": "paged-filter-list",
      "title": "Filter User List",
      "method": "GET",
      "href": "http://api.example.org/users/paged-filter"
      "type": "application/x-www-form-urlencoded",
      "fields": [
        { "name": "page-size", "type": "number", "value": "100"},
        { "name": "region", "type": "text", "value": ""},
        { "name": "last-name", "type": "text", "value": "" }
      ]
    }
  ]
```

Note that you might think that instead of offering two forms, it would be safe to offer a single, updated form that includes the `page_size` property with the value set to `"100"`. This is not a good idea. An existing client application might have been coded to depend upon the return list containing more than one hundred elements. By changing the default behavior to now return only one hundred rows, you might be "breaking" an existing client application.

Adding new elements is almost always safer than changing existing elements.

APIs are forever

The interface—the service API—is the contract that service producers make with API consumers. Like most contracts, they are meant to be kept. Breaking the contract is seen as breaking a promise. For this reason, it is wise for service API designers to treat the API as "unbreakable" and to assume they will last "forever." Werner Vogels,

Amazon's CTO, puts it like this: "We knew that designing APIs was a very important task as we'd only have one chance to get it right." While this seems a daunting task, it can be made much easier when you build into the API design the ability to safely change elements over time.

Of course, things change over time. Services evolve. They add features, modify inputs, and update outputs. That means the contract we make with API consumers needs to reflect the possibility of future changes. Essentially, change needs to be "written into the agreement."

While it isn't reasonable for interface designers to be able to accurately predict what the actual interface changes will be ("In two years, we'll add a third output property called `better-widget`"), it is reasonable for designers to create an interface that takes into account the possibility of future changes ("API consumers SHOULD ignore any additional properties that they do not understand"). In other words, the API design needs to not only accurately describe the current service interface, it also needs to account for possible future changes in a way that helps API consumers "survive" any unexpected evolution of the interface.

Lucky for us, there is a well-established approach we can use to account for variability in the interface over time. That method is called *hypermedia*.

How Hypermedia Can Help

At this point it's worthwhile to take a moment to ask, "Why is it important to design interfaces that offer this ability to support both stability and evolvability?" It is certainly harder than just emitting static interfaces tied directly to internal data and/or object models. And most developers don't need to worry about "change over time," either. Roy Fielding has been quoted as saying, "Software developers have always struggled with temporal thinking." They just need to get something out the door that actually solves the problem in front of them. The short answer is that long-lasting architecture in any form (software, network, structural, etc.) is a useful goal. We should all be aiming to create interfaces that are useful and remain so over a long period of time.

The longer answer touches on the role that well-designed interfaces play in the growth and egalitarian nature of computing. The man credited with bootstrapping the design of both HTTP and HTML, Tim Berners-Lee, says, "The web is for everyone." Creating well-designed, long-lasting interfaces makes it possible for more people to interact with the services behind the APIs. More interactions can lead to better, more creative uses. And that can lead to the possibility of fostering positive change—not just on the internet, but in the world in general. The HTTP protocol, and its sidekick HTML, have had a fantastic impact on how the world works today. Much of the HTTP/HTML success has to do with the fact that, after more than three decades,

these two stable, evolvable designs continue to foster innovation and creativity in all corners of the world.

Now, I don't think any of my work in designing and implementing APIs will ever have the impact that HTTP and HTML have had, but I like to think that my contributions to the internet might live up to the same ethos: "The web is for everyone." For that reason, I encourage designers and implementers of service APIs to do their utmost to create stable and evolvable interfaces.

So, how do we do that? We can provide stability for API consumers by relying on registered structured media types like HTML, HAL, Collection+JSON, SIREN, and others to exchange messages. We can support evolvability by including key elements that change often within those messages. These elements are the addresses (URLs), actions, (links and forms), and data objects (message properties).

Providing stability with message formats

API client applications need to be able to rely upon a stable of response and request messages in order to operate effectively. One way to make that possible is to commit to using one of the many registered structured message formats. This is what the web browser client does, too. It relies on a stable format (HTML) for passing most text-based messages from service to client. On the web, HTML is used as the default response format whether the service (website) is returning accounting information, social network data, game renderings, etc. Committing to using a well-known, well-designed format is essential for creating stable interfaces.

For more on programming with structured media type formats, check out my book *RESTful Web Clients* (O'Reilly).

There are a handful of worthy formats to pick from when you implement your service APIs. The IANA (Internet Assigned Numbers Authority) Media Types Registry is the source I use to find message formats that have the longevity and support that make them a good candidate for APIs. The most common formats currently in use include HAL, SIREN, and Collection+JSON, and there are several others.

Several recipes in Chapter 5 address the process of selecting and supporting structured media types.

Supporting evolvability with hypermedia controls

If the first step is to offer API consumers stability via structured media types, the second step (supporting evolvability) can be accomplished by selecting structured media

types rich in hypermedia controls. The more hypermedia features you include in your messages, the more evolvability you can support over time.

When you emit hypermedia formats, you have the ability to include new actions (added forms) as well as safe updates to existing actions (added properties, updated URLs, and changed HTTP methods). These are the most common things that will change in a service interface over time.

There are still limitations to this approach. Using hypermedia formats allows you to make it easy to change the addresses (URLs), objects (data properties), and actions (forms) safely and consistently. However, you may still need to modify the domain properties (names of things) over time, too. You might, for example, need to add a new property to the list of possible inputs (`middleName`), or a new output value (`alternateEmail`), or a new request parameter (`/filter/?newRegionFilter=south`). In this case, you'll need to create a new vocabulary document and allow clients to discover and select these additional vocabularies at runtime (see Chapter 3 for details). There are some recipes in Chapter 5 that address this ability to select vocabularies, too.

From Self-Servicing to Find and Bind

Another important element of services on the web is the ability for API consumers to "self-service" their onboarding experience. The power to simply select the desired service and instantly use it is a worthy goal. That's essentially the way people interact with websites today. Someone finds content that interests them and they can "grab a link" to that content and either share the link with others or copy that link onto their own web page where others will be able to see—and follow—the link to that same content. This is such a common, fundamental experience on the web that we don't really think of it as an "integration" or an onboarding experience. We just think, "That's the way the web works."

That's the way services should work, too. But most of the time, they do not. Instead, developers are left with a manual process of locating, selecting, and integrating existing services into their own solution. This can be a frustrating process with lots of fits and starts. It can be hard to find what you are looking for. Once you find an API that might work, it can be hard to understand the documentation. Also, it can be challenging to successfully integrate external APIs into your own service. Often, developers need to go through this process several times (once for each API on which the service will depend) before the job is done. Even after the service is completed, tested, and released, the entire process can start again when any one of the dependent APIs changes its interface.

What's needed is a way to automate most of this activity into a process that happens at runtime. A process I call "find and bind." By standardizing the way services are named and described (metadata), we can automate the way services are discovered

(find) and the way they are integrated (bind). This ability to find and interact with remote services you've never "met" before is the foundation of the current Domain Name System (DNS) for connecting machines with each other over the internet.

Supporting Distributed Data

Chapter 6 contains recipes that focus on storing and retrieving data within networked solutions. For much of the lifetime of data storage for computing, people have been focused on the concepts of *systems of record* (SOR) and *single source of truth* (SSOT). In general, these concepts focus on ensuring the accuracy of information by identifying a single location for each piece of data. This is also where the notion of *master data management* (MDM) comes into play. Controlling who can edit (what used to be called "master") the data is a key element in this story. Essentially, SSOT/MDM systems are built to assume there is just one authentic source for each data element and that everyone should agree on what (and where) those sources are.

However, on the open web, it is impossible to control where data is stored and how many copies are created of any piece or collection of data. In fact, it is wise to assume there will *always* be more than one copy of any data you are working with. It may be possible to explicitly ask for the most recent value for a data point or to try to keep track of which location is to be treated as the definitive source of a particular piece of data. But you should assume that the copy of the data you have is not always the copy others have.

All Data Is Remote

This notion of data on the web having multiple copies is an important point; one not to gloss over. Author and software architect Irakli Nadareishvili once expressed a similar "rule" to me when he told me, "Treat all data as if it was remote." When you treat data as remote, you make a few other important assumptions: 1) you can't change the storage medium or schema, 2) you can't control who can read or write information at that source, and 3) you're on your own when that data becomes unavailable (either temporarily or permanently).

For services that need to handle their own data, this "data on the web" rule should change the way you store, access, and validate the data you are charged with managing. For example, when possible you should keep track of who has requested data from you (via log records). You should also keep track of who has sent you updates to the data you are managing. Data services are also responsible for *local data integrity*. It should be impossible for anyone to write an invalid data record. Whenever possible, you should also support *reversing* data updates (within a limited time window). Finally, even when instructed to delete data, it is a good idea to keep a copy of it

around for a while in case it needs to be restored at some future point (again, subject to an established time window).

Nadareishvili's "data on the web" rule is important for services that depend on *remote* data, too. In situations where your service does not manage the data it works with (but depends on other services for that data), it is a good idea to always ask your remote sources for the most recent copies of the data you are working with to make sure you keep up with any changes that have occurred since you last read (or received) that data. You should also be prepared to have your data updates rejected by the dependent data services with which you are interacting. The service that *stores* the data is in charge of maintaining the integrity of that data.

You should be ready to support reversing a data update when needed, including the possibility of reversing a delete operation. This can be especially tricky when you are working with services that interact with more than one data source (e.g., a service that reads/writes to both `customerData` and `billingData` services). For example, when the `customerData` update is accepted and the `billingData` update is rejected, your service needs to know if it is important to *reverse* the `customerData` change.

Of course, there are cases where a service manages its own data *and* depends on other data sources. And, in my experience, this scenario happens often. Another common case is that you will want to keep a local cached *copy* of data you received from another source. This can speed performance and ease traffic for widely used resources. But copies are always just that—copies. When someone asks you for "the most recent copy" of the data, you may need to reach out and fetch an updated version of the record you currently have stored locally.

Data Is Evidence of Action

An important principle for those writing data-centric interfaces is the "Data Is Evidence of Action" mantra. Data is the *by-product* of some action that was performed. For example, when creating or updating a resource, data properties are created or modified. The data is evidence left behind by the create and update actions. Do this lots of times (e.g., many changes from many client applications), and you get a collection of actions. With this collection of actions in place, it is possible to ask questions about the evidence—to send queries. This is the key value of data stores: they provide the ability to ask questions.

When viewed as "evidence of action," data becomes a kind of witness to things happening on the web. The most accurate witness is one that reflects the best possible evidence. On the web, the best possible evidence is the HTTP exchanges themselves. The best storage format is the HTTP messages (both metadata and body) themselves. The ability to inspect and possibly replay HTTP exchanges makes for a powerful

storage platform. There is even a document format designed to properly capture HTTP messages—the HTTP Archive (HAR) format.

The */http Media Types

There is a long-standing (but rarely used) media type designed to capture and store HTTP messages directly. It is the `message/http` media type. There is also a media type defined to capture a series of HTTP requests and responses as a group: the `application/http` media type.

These media types, along with the HTTP Archive (HAR) format, offer great options for accurately capturing and storing HTTP exchanges.

The challenge is that direct HTTP messages (HAR files and `*/http` messages) are difficult to query efficiently. For that reason, you might want to store the evidence from HTTP exchanges in a more query-engine friendly format. There are many possible store-and-retrieve data systems to choose from. Some focus on making it easy to write data; most focus on making it easy to query data. The key point here is to keep in mind that all these storage systems are ways to *optimize* the management of the evidence. They are a new representation of the actions on the web.

To sum it up, for web-based services, you should be sure to capture (and make queryable) *all* the evidence of action. That means complete HTTP messages, including metadata and data. The medium you choose to do this (files, documents, rows, etc.) is not as important as maintaining the quality of the information over time.

Outside Versus Inside

No matter the data storage or query formats you use in your APIs, it is important that your internal data model not "leak out" into the service interface. In 2016, I shared an axiom on Twitter to help people remember the importance of capsulation in general:

> Your data model is not your object model is not your resource model is not your representation model.

It is worth reemphasizing this axiom when talking about services that are directly responsible for storing and managing data. The data model you are using on the *inside*—that is, the names and locations of data properties, their grouping into tables, collections, etc.—should remain independent of the model you are using for document-style messages on the *outside*. This helps maintain a loose coupling between storage and interface details, which can lead to a more reliable and stable experience for both API consumers and producers.

For example, a service interface might offer three types of resources: `user`, `job-type`, and `job-status`. Without knowing anything about internal storage, it would be easy

for API consumers to assume there are three collections of storage: ones that match the three resources types mentioned previously. But, it might be that `job-status` is just a property of `job-type`. Or that both `job-type` and `job-status` are both stored in a `name-value` collection that also stores other, similar information, like `user-status`, `shipping-status`, and more. The point is we don't know for sure what the storage model is like. And that should not matter to any API consumer.

We also don't know what storage *technology* is used to manage the example data just mentioned. It might be a simple file system, a complicated object database, or a collection of raw HTTP exchanges. And, more importantly, it does not matter (again) to the API consumer. What matters is the data properties that are available and the manner in which interface consumers can access (and possibly update) that data. The read and write details should not change often—that's an interface promise. But the storage technology can change as frequently as service implementers wish, as long as they continue to honor the interface promises.

This all adds up to another key principle for those creating data-centric services: spend a good deal of time thinking about the outside (interface) promises. These are promises that need to be kept for a long time. Conversely, don't worry too much about the inside (internal) technology details. They can (and probably will) change frequently based on technology trends, service availability, costs, and so forth.

Read Versus Write

When it comes to distributed data, the rules for reading data and writing data are fundamentally different. For example, most of the time data writes can be safely delayed—either by design or just through the normal course of the way messages are exchanged on the web. But delayed reads are often noticed more readily by API consumers and can directly affect the perceived speed and reliability of the data source. This is especially true when multiple data sources are used to complete a read query.

The greater the distance (either in space or time) the data needs to travel, the more likely it will be that API consumers will perceive the delays. The good news is that most API consumers can tolerate delays of close to one second without any major adverse effects. The truth is that *all* queries take time, and no response is "instantaneous." The reality is that most responses are short enough that we don't notice (or don't care).

In fact, the best way to implement responses for data reads is to serve up the solution without involving the network at all. That means relying on local cached copies of data to fulfill the query. This reduces the reliance on the network (no worries about the inability to reach a data source) and limits the response time to whatever it takes to assemble the local data that matches the query. See Recipe 6.9 for more on this.

It is also worth pointing out that *humans* typically have a higher tolerance for delays than *machines*. That means response delays for M2M interactions can be more troublesome than those for machine-to-human interactions. For that reason, it is wise to implement M2M interactions with the possibility of response delays in mind; plan ahead. Recipes 6.9 and 6.12 are possible ways to mitigate the effects of delayed responses for data-centric services.

Since read delays are more noticeable, you need to account for them in your interface design. For example, if some queries are likely to result in delayed responses (e.g., a large collation of data with a resulting multigigabyte report), you should design an interface that makes this delay explicit. For HTTP-based interfaces, that means making the `202 Accepted` response along with follow-up status reports part of the design (see Recipe 6.5 for details).

When it comes to writing data, while delays are undesirable, the more important element is to maintain data integrity in the process. It is possible that a single data write message needs to be processed by multiple services (either inline or parallel) and each interaction brings with it the possibility of a write error. In some cases, failure at just one of the storage sources means that all the other storage sources need to reject (or undo) the write, too. For this reason, it is best to limit the number of targets for each write action. One is best; any more than that are a problem. See Recipes 6.5, 6.8, and 6.13 for more on how to improve the write-ability of your service interfaces.

Robust Data Languages

The history of data storage and querying has provided an excellent list of data technology platforms. However, at least until recently, most data languages and platforms have ignored the unique requirements of the web environment. A few data languages are well-designed for widely distributed data, and some of them are very effective on the web where the sources are not only many but may also exist far from each other.

For the most part, a set of languages known as IRQLs are the most effective to use for reading data on the web. These languages are optimized for matching search criteria and returning a set of documents (or pointers to documents). They are also optimized for searching a large set of data, since most IRQLs are actually collections of indexes to external storage. For these reasons, it is a good idea for any service that supports data queries over HTTP to implement some type of IRQL internally. This is especially true if the API only needs to support reads and not writes. But even in cases where the interface supports both reading and writing data, a solid IRQL implementation can pay off since most data requests are reads anyway.

Database Queries Versus Information Retrieval Queries

There is an important difference between data query languages like SQL (Structured Query Language) and information retrieval query languages like Apache Lucene. Database languages are designed to return definitive results about some set of "facts" stored in the database. Information retrieval query languages (IRQLs) are designed to return a set of documents that match some supplied criteria. For the set of recipes in Chapter 6, we'll be focused primarily on the second type of query languages—information retrieval. Of course, we can (and often do) use database query languages to search for "documents" (usually rows) that match supplied criteria, too.

Information retrieval engines

There are a handful of candidates for an IRQL for your interfaces. One of the best known is the Apache Lucene project. The Lucene engine is pretty low-level and a bit of a challenge to implement on its own. But there are quite a few other implementations built on top of Lucene that are easier to deploy and maintain. As of this writing, the Solr engine is the one I see most often in use.

Other IRQL options

Along with a solid IRQL for reading data, most services will also need to support writing data. While a few IRQL-like engines support both reading and writing—e.g., GraphQL, SPARQL, OData, and JSON:API—it is a good idea to consider using a different technology when supporting data writes.

SQL-like data engines

Usually, simple data storage can be handled by implementing a file-based storage system (e.g., each record is a file in the system). However, if the service needs to be able to scale up past a handful of users or a few hundred documents, it is better to implement some kind of data storage optimized for writes. The common uses rely on some SQL-based data technology like MongoDB, SQLite, PostgreSQL, and others.

Streaming data engines

If you need to handle a large number of data writes per second (e.g., thousands), you probably want to implement a streaming data engine such as Apache Kafka, Apache Pulsar, or other alternatives.

Above all, it is important to see all these options as "behind-the-scenes" details. A well-designed interface for data-centric services will *not* expose the internal data technology in use. That makes it possible to update the internal technology as needed without adversely affecting the external API (see "Read Versus Write" on page 43). For example, you might start with a file-based data management system, add an IRQL

to support additional queries, later move to SQL-based, and eventually go to a streaming data engine. Throughout that internal technological journey, the external interface should not change. See Recipes 6.1, 6.4, and 6.7 for more on this topic.

Empowering Extensibility with Hypermedia Workflow

The web has always been a place where content servers could be easily connected to each other (via links) to form a larger offering. This ability to simply link to another website without asking for permissions or having to make special coding, layout, or content changes was one of the goals of those credited with creating the World Wide Web (WWW). For this reason, Tim Berners-Lee's proposal for his WWW information system concludes, "We should work toward a universal linked information system, in which generality and portability are more important than fancy graphics techniques and complex extra facilities." Chapter 7 has recipes aimed at helping to do just that.

While Berners-Lee (and Ted Nelson before him) focused on ways to link *documents* together in a seamless way, this same notion of easy connections with few barriers can be applied to *services*, too. By linking multiple services together, we can solve a particular need. For example, the steps needed to perform an online store checkout might include actions like `computeTotals`, `applyTax`, `arrangePayment`, `scheduleShipping`, and `sendConfirmation`.

When working in local source code, these actions are usually expressed as operations within the codebase and as a series of sequential steps to be executed in order. However, some languages also support the use of parallel processing to execute a collection of actions at the same time. This can greatly improve the scalability of the workflow process but requires that both the workflow designers and the service designers have agreements in place that clearly separate parallel from sequential tasks. Even when all tasks might be eligible for parallel execution, the workflow itself may require a special order of events. It is difficult, for example, to compute the taxes on a shopping order before you have finalized the contents of the shopping basket.

In the case of web-based services, each function needs to be treated as if it is run in a separate process at a different location. This distributed nature of the web means services can't be counted on to share data models, storage systems, or programming languages. The only thing they share is an agreed *interface*. The common interface on the web is defined by the HTTP specification. But this is a very low-level agreement. For efficient work to be done, services need to share more than the transfer protocol. They need to share a kind of language for coordinating services.

Choreography, Orchestration, and Hypermedia Workflow

When talking about coordinating services, it is common to encounter the notions of *choreography* and *orchestration*. Each represents architectural points of view on how to design and implement service coordination. In my experience, the web offers a third, more direct (and, I think, more powerful) point of view on coordination: that of describing workflows as declarative documents that, like HTTP, provide a consistent interface that services can use to interact with each other. I'll get to this *hypermedia workflow* shortly. First, it is worthwhile to review just what choreography and orchestration offer and how they differ from a hypermedia approach.

Centralized orchestration

The service orchestration model is a centralized approach to defining workflows in a single context and then submitting that document to a workflow "engine." This has the effect of creating an integration programming model (e.g., BPEL) that solution designers can use to connect services in a logical execution flow. The orchestra metaphor is a common way to talk about this approach since there is a single "conductor" (the engine) that leads a group of people all working from essentially the same agreed-upon "music" (the workflow document).

Orchestration History

Service orchestration has a long and varied history. Probably the best known "origin story" for service orchestration comes from XLang, Web Services Flow Language (WSFL), and its standardized offshoot Business Process Execution Language or WS-BPEL. These are designed to support describing a series of workflow steps and decisions and then controlling the execution of that workflow by submitting the document to a workflow "engine," such as Microsoft's BizTalk, IBM's WebSphere, Apache ODE, and others.

The advantage of the orchestration approach is that the solution is clearly spelled out in script or code. This makes it easy to reason about, validate, and test the workflow ahead of time. The use of a workflow engine also makes it easier to monitor and manage the workflow process. There are quite a few platforms and programming suites designed to support BPEL and similar languages.

There are also challenges to the centralized approach. Chief among them is the dependence on a central execution engine. If that engine is down or unreachable, the entire workflow system can grind to a halt. This can be mitigated by running the workflow engine within a cluster (to handle failing machines) and at multiple locations (to handle network-related problems). Another challenge for reliance on the

orchestration approach is that it often assumes synchronous processing and may encourage a more tightly coupled implementation.

Lastly, the orchestration model also assumes a kind of single point of contact. Usually, services don't talk to each other directly, they talk to the orchestration engine. More precisely, the engine makes calls to the services, and the engine composes its own resulting solution.

Orchestrating the Cloud

Most cloud services today offer some flavor of service orchestration. Amazon Web Services provides AWS Step, Google has Workflows, and Microsoft relies upon Azure Cloud Automation.

The drawbacks of failing central engines and tightly coupled workflow definitions can be handled by creating a central repository for the *workflow* documents and distributing the processing of those documents. This distributed execution approach is very similar to the other common way to design workflows: service *choreography*.

Stateless choreography

Typically, choreography is defined as a kind of "dance" of services. Each service knows its own "moves," and each acts as an independent entity that interacts with others. With choreography, the workflow is a by-product of the interactions between existing services. Workflow is an "emergent" behavior.

Let's Dance

The dance metaphor appears often in descriptions of the choreography approach to workflow. In fact, this dance theme is so strong with the choreography-style workflow crowd that an entire programming language designed for integration work was released in 2017 called Ballerina.

There are definite advantages to the emergent choreography model. First, individual services don't need to know about the big picture; they just need to know how to accomplish their own tasks. Composing workflow results in a more loosely coupled solution, and loosely coupled solutions are usually more resilient and easier to successfully modify over time. That can make it easier to replace parts of the solution over time without disrupting others in the process. This can result in a flexible, resilient workflow environment.

Of course, there are drawbacks, too. One challenge to working in a choreographed world is that it can be difficult to monitor individual workflow progress. Since the

solution emerges from a set of individual, stateless tasks, there is no single point of control where you can be alerted of problems in the execution of a job.

It can also be hard to assess the overall health of your workflow ecosystem. Are there certain tasks (e.g., sending email) that fail for all jobs? Or are there problems with all jobs that send emails to a single, problematic address? Are some machines performing poorly no matter what services they are hosting? Are some services unreachable due to network conditions?

Dancing in the Clouds

The major cloud vendors all offer their own version of service choreography. Google encourages the use of their Pub/Sub and Eventarc services. Microsoft has Event Grid, and AWS encourages the use of its Simple Notification Service (SNS) and Simple Queue Service (SQS) tools to build choreographed workflows.

The monitoring challenge can be handled by creating a progress resource for each job that is easy to locate and view on demand. You can then instruct services to emit progress reports to the associated resource whenever they complete their task.

Both centralized orchestration and point-to-point choreography have their advantages and drawbacks. In general, workflows without many steps and few branching options are well-serviced with an orchestration approach. Define the work in a single place (a document) and use that document as the execution script for the solution. Often, this can be written up in the native source code (see Recipe 7.3).

However, if the workflows are quite involved, it can be more beneficial to use the choreography approach. As the number of steps goes up, the sequential approach of most orchestration engines becomes a liability. Also, extensive branching points in workflows can make it hard to reason about outcomes and difficult to "roll back" problematic workflows. The choreography approach makes it easier to handle asynchronous workflows, support individual rollbacks, and can remove the need for `if .. then .. else`–style branching. This assumes that each service works independently and that there is little state information shared between them. Choreographed solutions are sometimes expressed as source code but can also be expressed in a domain-specific language (DSL) tuned for the task at hand. In this book, I'll show you a DSL designed just for this purpose (see Recipe 7.4).

Hypermedia workflow

There is, however, another approach that I have found quite reliable; one that depends on the use of hypermedia controls (forms and links) that describe the details of service interactions at runtime. I have always called this *hypermedia workflow*. This approach works well both for cases with just a few steps and ones with several

involved processes. It takes advantage of the independent, parallel processing of choreography as well as the centralized approach of orchestration.

Jazz as a Workflow Analogy

Just as it has become popular to use orchestra and dance analogies to talk about service coordination, I have been using jazz music as a way to talk about hypermedia workflow. With orchestration, there is "someone in charge." With choreography, it is important that everyone knows their part and works closely with each other to reach the final outcome. With jazz music, everyone has a basic idea of what the song is and they all get to contribute in their own way to create a final performance.

The key to designing and implementing hypermedia workflows is to make sure each service that is enlisted in the flow has a *composable* service interface (see Recipe 7.1) that supports the following actions:

`Execute`
: Do the work (e.g., `computeTax`, `createCustomerResource`, etc.).

`Repeat`
: Do the same work again in case something didn't work the first time.

`Revert`
: Undo the work in case there was an error somewhere else.

`Cancel`
: Stop processing the task and undo any previous work.

When each service (called a *task*) supports these actions, it is possible to collect them together in a *job*. Workflow jobs run all tasks in parallel and offer two other important actions:

`Continue`
: Pick up the work where it last left off.

`Restart`
: Start from the very beginning again.

`Cancel`
: Stop processing the job and tell all tasks to undo any previous work.

There are some other details for this workflow language regarding shared state and time-outs that are covered in several recipes in Chapter 7. The important thing to keep in mind is that the workflow interface described here is very simple and usually straightforward to implement for an individual service. There are definitely challenges. For example, supporting the `Revert` action has implications if the service calls

other services under the covers. But that set of details can be sorted out within source code and does not need to be exposed as part of the service interface.

Another advantage of relying on a hypermedia workflow language is that all the details of a workflow can be expressed as one or more *documents* instead of source code or script. Most programming languages are *imperative*. They describe *how* something is done. Hypermedia documents are, however, *declarative*. Declarative languages describe *what* needs to be done without explicitly specifying a step-by-step process.

This ability to describe actions as tasks, and flows as jobs, makes it possible to write composable services that can be successfully enlisted into a workflow without worrying about how all the parts will interact. Just like HTTP, hypermedia workflow standardizes the external interface and allows services to use any internal programming languages and paradigms they wish.

Workflow Challenges

The challenges for implementing web-based workflows are many. But they break down into a relatively small set of tractable problems. The one's I'll review here are:

- Sharing state, not data models
- Constraining workflows
- Observing workflows
- Time as a workflow element
- Dealing with workflow errors

Sharing state, not data models

A key stumbling block for building robust, scalable workflow systems for the web has to do with dealing with the data that needs to pass between independent services. Most solutions I've seen assume a shared data model (e.g., passing parameters as strongly typed schema) or even shared data storage (e.g., all services read the same SQL database tables). Neither of these work well.

A better approach is to share properties using a simple document model instead (see Recipe 7.2). In these cases, data values are passed as strongly typed *documents* (HTML, Collection+JSON, SIREN, HAL, etc.) and may contain all types of data elements. Just what data properties appear in these documents is governed by the semantic profiles (Recipe 3.4) and vocabularies (Recipe 3.3) established ahead of time.

Sharing these documents can be accomplished using the FORM affordance (Recipe 3.7) or a shared addressable resource itself.

Successful workflow implementations support piping data between services without constraining the types of data to be shared.

Constrained workflow

When you think of workflow as a set of constraints (execute, repeat, reverse, continue, and restart), you can then design and implement services in consistent ways as long as they agree to the rules ahead of time. Just as HTML has a set of rules for hypermedia affordances (HTML.A, HTML.FORM, HTML.IMG, and HTML.LINK) with clearly defined semantics and outcomes, workflow can be designed as a set of affordances (execute, etc.) with clearly defined semantics and outcomes.

A reliable workflow system defines the shared interface between services as a general set of affordances.

Observable workflow

Another important element of a successful workflow system is the ability to see what is going on at runtime (see Recipe 7.7). This observability is very common in shared flow systems like DevOps pipelines and can be just as valuable for general runtime workflow, too. You also need a way to intervene in a workflow to fix problems and, if appropriate, `continue` or `restart` a job or—if all else fails—`cancel` the job completely.

A well-developed workflow system provides a set of dashboards that show the process of running workflows.

Time as a workflow element

Any system that attempts to coordinate services needs to keep track of the *time* it takes to accomplish a task or complete a job. Of course, some processes take more time to complete. Computing net present value or working through a set of complex computations may take minutes or maybe hours. Workflow systems need to allow for this (see Recipe 7.15).

Healthy systems set limits (`maxTTL`) to the amount of time alloted for work. Pass that limit and you need to `cancel` the process.

Dealing with workflow errors

Finally, workflow processes need to account for runtime errors caused by network conditions (see Recipe 7.16) or other local problems (see Recipe 7.17). There are ways to automate some of these responses, but sometimes machines can't work out a solution. When that happens, you need a way to "call for help" (see Recipe 7.18) to some human somewhere who will make the right call on whether to `continue` or `cancel` the process.

Quick Summary

Now that we've explored the background and history underpinning the design, client, service, data, and workflow recipes in this book, it's time to jump directly into the details of each to help us create resilient and robust network-based solutions.

PART II

Hypermedia Recipe Catalog

No pattern is an isolated entity. Each pattern can exist in the world, only to the extent that is supported by other patterns.

—Christopher Alexander

CHAPTER 3
Hypermedia Design

> *The problem is essentially the one discussed by science fiction writers: "how do you get communications started among totally uncorrelated 'sapient' beings?"*
>
> *—J.C.R. Licklider, 1966*

A foundational element of any system-level design is a set of shared principles or understandings about how parts of the system interact. That is the overarching problem we'll address in this chapter—how to design systems where machines built by different people who have never met can successfully interact with each other. The recipes in this chapter focus on the design aspects of hypermedia systems. To do that, you'll find recipes that explore the relationship between media types, hypermedia controls, data properties, and semantic profiles that help bring them together (see Figure 3-1).

If you're interested in the background concepts and technologies behind designing hypermedia-driven solutions, see "Establishing a Foundation with Hypermedia Designs" on page 21.

For communications within the web, HTTP is at the heart of this shared understanding. HTTP sets the rules—and the *expectations*—for sending data between services. And, despite the fact that HTTP's history dates back to the mid 1980s, it is still ubiquitous and relevant after more than 30 years.

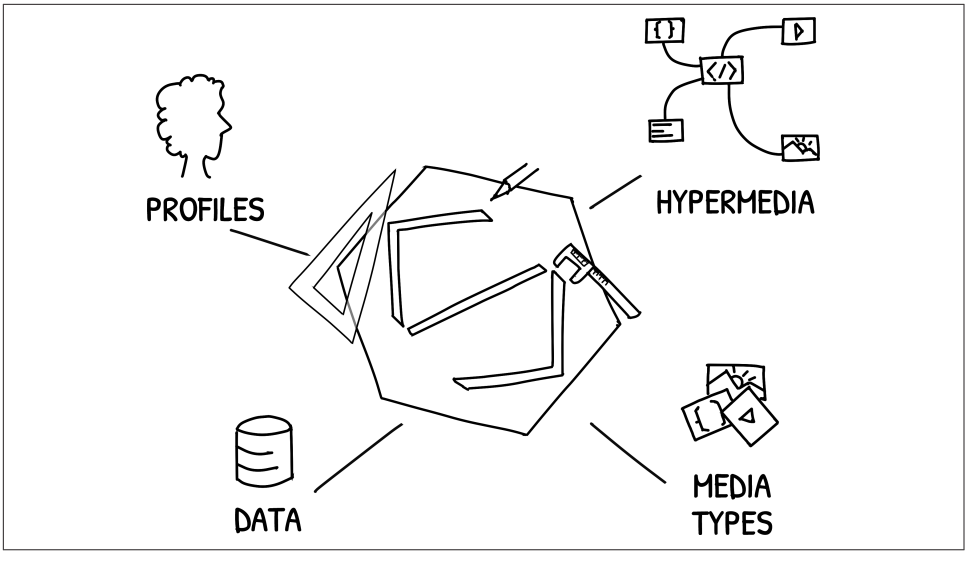

Figure 3-1. Hypermedia design recipes

It is important to remember that HTTP is the higher-level protocol agreement between machines on the network. Lower-level protocols, like TCP, IP, and UDP, provide the backbone for moving bits around. One of the things I find most interesting about this low-level communications system is that it works *because* the communication components do not understand the meaning of the data they ship around. Meaning is separated from the message.

The notion that information can be dealt with independent of the messages used to share that information is a key understanding that makes M2M communication possible "at scale." Dealing with information within its own level is also important. HTTP offers a great example of this using media types. I can send you company sales data in an HTML message. I can send you the same information in a CSV message, or as a plain text message, and so forth. The *data* is independent of the media type.

You'll find this notion of separation throughout the recipes in this chapter. While protocol and media type are well-established forms of separation, the recipes in this chapter also rely on an additional form—that of separating *vocabulary* from the message format. Vocabulary is the set of rules we use to communicate something important. Communication works best when we both share the same vocabulary.

For example, you can imagine two animated machines talking to each other:

> "Hey, Machine-1, let's talk about health care systems using the HL7 FHIR Release 4 vocabulary."
>
> "OK, Machine-2. But can we please use RDF to send messages instead of XML?"
>
> "Sure, Machine-1. As long as we can use HTTP and not MQTT."
>
> "Agreed!"

Notably, the FHIR platform recognizes that FHIR information can be shared using XML, JSON, and RDF message formats—a clear nod to the importance of separating domain-specific information from the formats used to share that information.

Another important shared understanding for scalable systems is agreement on "how we get things done." In this case, the way things get done is through hypermedia. Hypermedia controls, such as links and forms, are used to express the runtime metadata needed to complete a task. The controls themselves are structured objects that both services and clients understand ahead of time. However, the contents of that control—the values of the metadata properties—are supplied at runtime. These include the data properties needed for a particular action, the protocol method to use to execute the action, and the target URL for that action. All are decided at runtime. The client doesn't need to memorize them—it receives them in the message. Basically, the hypermedia controls are another level of separation. In fact, the JSON-LD message format relies on a separate vocabulary (Hydra) to express the hypermedia information within a JSON-LD message. You'll find a few recipes here that acknowledge the importance of hypermedia for RWAs. You'll also find several specific hypermedia recipes in the following chapters covering specific implementation patterns.

Finally, you'll find some recipes that are specific to successful communication over HTTP. These recipes cover things like network promise (safety and idempotence) and the realities of supporting M2M communications over unreliable connections at long distances. Lucky for us, HTTP is well suited for resolving broken connections—it was designed and built when most HTTP conversations were conducted over relatively slow voice telephone lines. However, for HTTP to really work well, both clients and servers need to agree on the proper use of message metadata (e.g., HTTP headers), along with some added support for how to react when network-level errors occur.

So, this design chapter focuses on recipes that acknowledge the power of separate layers of communication, the role of hypermedia in communicating *how things get done*, and some shared understanding on how to ensure reliable communications when errors occur.

3.1 Creating Interoperability with Registered Media Types

A key to establishing a stable, reliable system is to ensure long-term interoperability between services. That means services created years in the past are able to successfully exchange messages with services created years in the future.

Problem

How do you design services that have a high degree of interoperability well into the future?

Solution

The best way to ensure a high level of long-term interoperability between services is to establish stable rules for exchanging information. On the web, the best way to do that is to select and document support for one or more open source media type formats for data exchange. For example, the HTML media type offers a strongly typed format that HTML browsers can "bind" to without needing to understand the *contents* of the HTML document. In fact, the contents of a document can change over time (adding paragraphs, moving text, links, forms, etc.), but the HTML browser does not need to be updated. Using HTML ensures future compatibility.

A good source for long-term, stable media types is the Internet Assigned Numbers Authority (IANA). Viable candidate media types for long-term support of RWAs are unstructured media types like XML and JSON, as well as structured media types such as HTML, Collection+JSON, UBER, HAL, and SIREN. See Appendix C for a list of viable media types at the time of this book's release.

See Recipe 3.2 for more on the difference between structured and unstructured media types.

Discussion

When you create services, you should document which registered media types (RMTs) your service can support. It is recommended that your service support more than one RMT, and that you allow service consumers to both discover which RMTs your service supports and how they can indicate their preference when exchanging messages (see Recipe 4.6).

When I release a service, I almost always add HTML as one of the designated message formats. It has been around for 30+ years, you can use any common browser as a client for the service (great for testing), and there are enough parsers and other HTML tooling to make it relatively easy for consumers of your service to be able to exchange data with you.

Additional recommendations for candidate media types are: 1) they should support hypermedia within the message (see Recipe 3.5) and 2) they should support custom extensions (e.g., your ability to safely add new features to the format without breaking any existing applications). This last point will come in handy when you are stuck supporting a format that has fallen out of favor and you need to make some modifications to extend the life of your service.

It is also possible to create your own media type format for your services. That can work if: 1) your universe of service API consumers is relatively limited (e.g., within your own company), 2) your universe of service API consumers is crazy large (e.g., Google, Facebook, Amazon, etc.), or 3) your services are the leader in a vertical (e.g., document management, financial services, health care, etc.). If your efforts do not fall into one of these categories, I recommend you *do not* author your own media type.

So You Want to Create Your Own Media Type, Eh?

If you decide to author your own custom media type, you should treat it as if it will become a popular, public format. That means documenting it and registering it properly. You also need to be ready to support any community that grows up around your media type format—including example apps and other tooling. Finally, you need to be committed to doing this work for a long time. Also, keep in mind that some people might build their own services using your format, and they will deserve the proper support well into the future.

Over the years, I've authored close to a dozen original media types. Only a few have ever garnered much attention. But I still try to keep up my responsibilities to all of them.

See Also

- Recipe 3.2, "Ensuring Future Compatibility with Structured Media Types"
- Recipe 3.5, "Expressing Actions at Runtime with Embedded Hypermedia"
- Recipe 4.6, "Managing Representation Formats at Runtime"

3.2 Ensuring Future Compatibility with Structured Media Types

An important foundational element in the design of RWAs is supporting future compatibility. This means making it possible for services written in the past to continue to interact with services well into the future. It also means designing interactions so that it is unlikely that future changes to already-deployed services will *break* other existing services or clients.

Problem

How do you design M2M interactions that can support modifications to in-production services that do not break existing service consumers?

Solution

To support nonbreaking changes to existing services, you should use *structured media types* (SMTs) to pass information back and forth. SMTs make it possible to emit a stable, nonbreaking message even when the *contents* of that message (e.g., the properties of a record, list of actions, etc.) have changed. The key is to design interactions with the message shared between machines maintaining the same structure even when the data conveyed by that message changes. See the example for details.

A structured media type provides a strongly typed format that does not change based on the data being expressed. This is in opposition to unstructured message formats like XML and JSON. See the example for details.

It is a good idea to use well-known media types in your interactions. Media types registered through the IANA are good candidates. For example, HTML is a good example of an SMT. Other viable general-use SMT formats are Collection+JSON and UBER. See "Viable Registered Media Types for RESTful Web APIs" on page 421 for a longer list of media types to consider.

Example

An SMT provides a strongly typed format that does not change based on the data being expressed. Here's an example of a simple message expressed as HTML:

```
<ul name="Person">
  <li name="givenName">Marti</li>
  <li name="familyName">Contardi</li>
</ul>
 ...
<ul name="Person">
  <li name="givenName">Marti</li>
  <li name="familyName">Contardi</li>
```

```
  <li name="emailAddress">mcontardi@example.org</li>
</ul>
```

The *structure* of this message can be expressed (in an overly simplified way) as: "one or more `li` elements enclosed by a single `ul` element." Note that adding more content (for example, an email address) does not change the *structure* of the message (you could use the same message validator), just the *contents*.

Contrast the preceding example with the following JSON objects:

```
{"Person" : {
  "givenName": "Marti",
  "familyName": "Contardi"
  }
}
 ...
{"Person" : {
  "givenName": "Marti",
  "familyName": "Contardi",
  "emailAddress": "mcontardi@example.org",
  }
}
```

The JSON structure can be expressed as: "a JSON object with two keys (`givenName` and `familyName`)." Adding an email address to the message would result in a change in the *structure* of the JSON message (e.g., requires a new JSON schema document) as well as a change in *content*.

Discussion

The important element in this recipe is to keep the *content* of the message loosely coupled to the *structure*. That allows message consumer applications to more easily and consistently validate incoming messages (to "bind" to the message type)—even when the contents of those messages change over time.

Another way to think about this solution is to assume that the first task of message consumer applications is to make sure the message is *well-formed*—that it complies with the basic rules of how a message must be constructed. It is, however, a different task to make sure the message is *valid*—that the message *content* follows established rules for what information needs to appear within the messages (e.g., "all `Person` messages MUST include a `givenName`, `familyName`, and `emailAddress` property").

To ensure future compatibility, the first step is to make sure negotiation messages can remain well-formed even when the rules for what constitutes a valid message change over time.

See Also

- Recipe 3.3, "Sharing Domain Specifics via Published Vocabularies"
- Recipe 3.4, "Describing Problem Spaces with Semantic Profiles"
- Recipe 3.5, "Expressing Actions at Runtime with Embedded Hypermedia"
- Recipe 4.6, "Managing Representation Formats at Runtime"

3.3 Sharing Domain Specifics via Published Vocabularies

Services with a well-managed vocabulary are more likely to be understood by others and therefore more likely to gain wider adoption. Just like you need to support registered, structured media types to ensure the long-term compatibility and reliability of your service, you also need to support stable, consistent domain specifics to ensure long-term viability for both service producers and consumers.

Problem

How can you make sure your service's data property names will be understood by other services—even services that you did not create?

Solution

To ensure that the data properties your service uses to exchange information are well understood by a wide range of other services (even ones you did not create), you should employ well-documented, widely known data property names as part of your external interface (API). These names should already be defined in published data vocabularies; if not, you should publish your data vocabulary so that others creating similar services can find and use the same terms.

A good source of general-use published vocabularies for the web is *Schema.org*. It has hundreds of well-defined terms that apply to a wide set of use cases and it continues to grow in a well-governed way. There are other well-governed vocabulary sources like *Microformats.org* and Dublin Core.

A word of caution. Some vocabularies, especially industry-specific ones, are not fully open source (e.g., you must pay for access and participation). I will also point out that some vocabulary initiatives, even some open source ones, aim for more than a simple shared vocabulary. They include architectural, platform, and even SDK recommendations and/or constraints.

Your best bet for creating long-term support for interoperability is to make sure any vocabulary terms you use are disconnected from any other software or hardware dependencies.

As of this writing there are some industry-specific public vocabularies to consider, too. Some examples are: PSD2 for payments, FHIR for health care, and ACORD for insurance.

When you release your service, you should also release a document that lists all the vocabulary terms consumed or emitted by your service along with links to proper definitions. See Recipe 3.4 for more on how to properly document and publish your vocabulary documents.

Example

When you have the option, you should use well-known terms in your service's external API even when your own internal data storage system uses local or company-specific terms.

For example, here's a `person` record that reflects the terms used within a typical US company:

```
{ "collection" : {
    "links": [
      {"rel": "self", "href": "http://api.example.org/persons"}
    ],
    "items" : [
      {
        "href": "http://api.example.org/persons/q1w2e3r4",
        "data" : [
          {"name": "fname", "value": "Dana"},
          {"name": "lname", "value": "Doe"},
          {"name": "ph", "value": "123-456-7890"}
        ]
      }
    ]
  }
}
```

And here's the same record that has been updated to reflect terms available in the Schema.org vocabulary:

```
{ "collection" : {
    "links": [
      {"rel": "self", "href": "http://api.example.org/persons"},
      {"rel": "profile", "href": "http://profiles.example.org/persons"}
    ],
    "items" : [
      {
        "href": "http://api.example.org/persons/q1w2e3r4",
        "data" : [
          {"name": "givenName", "value": "Dana"},
          {"name": "familyName", "value": "Doe"},
          {"name": "telephone", "value": "123-456-7890"}
        ]
```

```
        }
      ]
    }
  }
```

Note the use of the `profile` link relation in the second example. See Recipe 3.4 for details.

Discussion

This recipe is based on the idea of making sure the data property terms are not tightly coupled to the message format. It is, essentially, the flip side of Recipe 3.2. Committing to well-known, loosely coupled vocabularies is also an excellent way to protect your service from changes to any internal data models over time.

See Chapter 6 for more recipes on handling data for RWAs.

Constraining your external interfaces to use only well-known, well-documented property names is one of the best things you can do to ensure the interoperability of your service. This, along with publishing semantic profiles (see Recipe 3.4), make up the backbone of large-scale information system management. However, this work is not at all easy. It requires attention to detail, careful documentation, and persistent support. The team that manages and enforces a community's vocabulary is doing important and valuable work.

One of the challenges of this recipe is that the *public* vocabulary for your services is quite likely not the same as the *private* vocabulary. That private set of names is usually tied to age-old internal practices, possibly a single team's design aesthetics, or even decisions dictated by commercial off-the-shelf (COTS) software purchased a long time ago. To solve this problem, services need to implement what is called an "anti-corruption layer". There is no need to modify the existing data storage model or services.

It is important to document and share your service vocabulary. A good practice is to publish a list of all the "magic strings" (see "Richardson's Magic Strings" on page 15) in a single place along with a short description and, whenever possible, a reference (in the form of a URL) to the source of the description. I use the Application-Level Profile Semantics (ALPS) format for this (see Recipe 3.4).

Here is an example ALPS vocabulary document:

```
{ "alps" : {
   "descriptor": [
     {"id": "givenName", "def": "https://schema.org/givenName",
      "title": "Given name. In the U.S., the first name of a Person.",
      "tag": "ontology"},
     {"id": "familyName", "def": "https://schema.org/givenName"
      "title": "Family name. In the U.S., the last name of a Person.",
      "tag": "ontology"},
     {"id": "telephone", "def": "https://schema.org/telephone",
      "title": "The telephone number.",
      "tag": "ontology"
     },
     {"id": "country", "def": "http://microformats.org/wiki/hcard#country-name",
      "title": "Name of the country associated with this person.",
      "tag": "ontology"
     }
   ]
  }
}
```

When creating RWA vocabularies, it is a good practice to identify a single preferred source for term definitions, along with one or more "backup" sources. For example, your vocabulary governance document guidance could read like the following:

> "Whenever possible, use terms from Schema.org first. If no acceptable terms can be found, look next to Microformats.org and then to Dublin Core for possible terms. Finally, if you cannot find acceptable terms in any of those locations, create a new term in the company-wide vocabulary repository at [some-url]."

Some additional notes:

Mixing terms
: It is perfectly acceptable to mix vocabulary references in a single set of terms. You can see in the preceding example that I included three terms from Schema.org and one term from Microformats.org.

Limiting synonyms
: Just as in real life, there can be multiple terms that mean the same thing, for example, "tel" (from Microformats.org) and "telephone" (from Schema.org). Whenever possible, limit the use of synonyms by adopting a single term for all external uses.

Publishing vocabularies
: See Recipe 3.4 on how to publish this document and retrieve it on the web.

See Also

- Recipe 3.2, "Ensuring Future Compatibility with Structured Media Types"
- Recipe 3.4, "Describing Problem Spaces with Semantic Profiles"
- Recipe 4.4, "Coding Effective Clients to Understand Vocabulary Profiles"
- Recipe 4.5, "Negotiating for Profile Support at Runtime"

3.4 Describing Problem Spaces with Semantic Profiles

Along with decoupling your well-defined data property vocabulary from your message structures, it is important to also put effort into defining how the data properties get passed back and forth. This is the work of "describing the problem space." Problem spaces are used in games and interactive artwork as a way to place "guide rails" on a set of related activities. Problem spaces are the "rules of the game," so to speak. RWAs rely on the "rules of the game" as well.

The Problem

How can you provide a detailed description of all the possible data properties, objects, and actions supported by your services in a way that is usable both at design time and at runtime?

The Solution

To make sure developers can quickly and accurately understand the data exchanges and actions supported by your service, you can publish a semantic profile document (SPD). SPDs contain a complete list of all the data properties, objects, and actions a service supports. Semantic profiles, however, are *not* API definition documents like OpenAPI, WSDL, AsyncAPI, and others. See "API Definition Formats" on page 422 in the appendix for a longer list of API definition formats.

SPDs typically include all three elements of information architecture: ontology, taxonomy, and choreography.

See "The Power of Vocabularies" on page 14 for a discussion of the three pillars of IA.

Two common SPD formats are Dublin Core Application Profiles (DCAP) and Application-Level Profile Semantics (ALPS). Since I am a coauthor on ALPS, all the examples, I'll show here use the ALPS format. See "Semantic Profile Document Formats" on page 423 in Appendix C for a longer list.

Example

Here is an example of a valid ALPS semantic profile document. Notice that the three pillars of Morville's information architecture (ontology, taxonomy, and choreography) are represented in this example (see "The Power of Vocabularies" on page 14 for details):

```
{ "$schema": "https://alps-io.github.io/schemas/alps.json",
  "alps" : {
  "title": "Person Semantic Profile Document",
  "doc": {"value":
    "Simple SPD example for http://webapicookbook.com[Web API Cookbook]."},
  "descriptor": [
    {"id": "href", "def": "https://schema.org/url",
     "tag": "ontology"},
    {"id": "identifier", "def": "https://schema.org/identifier",
     "tag": "ontology"},
    {"id": "givenName", "def": "https://schema.org/givenName",
     "tag": "ontology"},
    {"id": "familyName", "def": "https://schema.org/familyName",
     "tag": "ontology"},
    {"id": "telephone", "def": "https://schema.org/telephone",
     "tag": "ontology"},

    {"id": "Person", "tag": "taxonomy",
      "descriptor": [
        {"href": "#href"},
        {"href": "#identifier"},
        {"href": "#givenName"},
        {"href": "#familyName"},
        {"href": "#telephone"}
      ]
    },
    {"id": "Home", "tag": "taxonomy",
      "descriptor": [
        {"href": "#goList"},
        {"href": "#goHome"}
      ]
    },

    {"id": "List", "tag": "taxonomy",
      "descriptor": [
        {"href": "#Person"},
        {"href": "#goFilter"},
        {"href": "#goItem"},
        {"href": "#doCreate"},
```

```
        {"href": "#goList"},
        {"href": "#goHome"}
      ]
    },
    {"id": "Item", "tag": "taxonomy",
      "descriptor": [
        {"href": "#Person"},
        {"href": "#goFilter"},
        {"href": "#goItem"},
        {"href": "#doUpdate"},
        {"href": "#doRemove"},
        {"href": "#goList"},
        {"href": "#goHome"}
      ]
    },
    {"id": "goHome", "type": "safe", "tag": "choreography", "rt": "#Home"},
    {"id": "goList", "type": "safe", "tag": "choreography", "rt": "#List"},
    {"id": "goFilter", "type": "safe", "tag": "choreography", "rt": "#List"},
    {"id": "goItem", "type": "safe", "tag": "choreography", "rt": "#Item"},
    {"id": "doCreate", "type": "unsafe", "tag": "choreography", "rt": "#Item"},
    {"id": "doUpdate", "type": "idempotent", "tag": "choreography",
      "rt": "#Item"},
    {"id": "doRemove", "type": "idempotent", "tag": "choreography",
      "rt": "#Item"}
  ]
  }
}
```

Figure 3-2 shows a workflow diagram (the *choreography*) for the *person-alps.json* file. You can find the complete ALPS rendering (ALPS file, diagram, and documentation) of this semantic profile at the book's website.

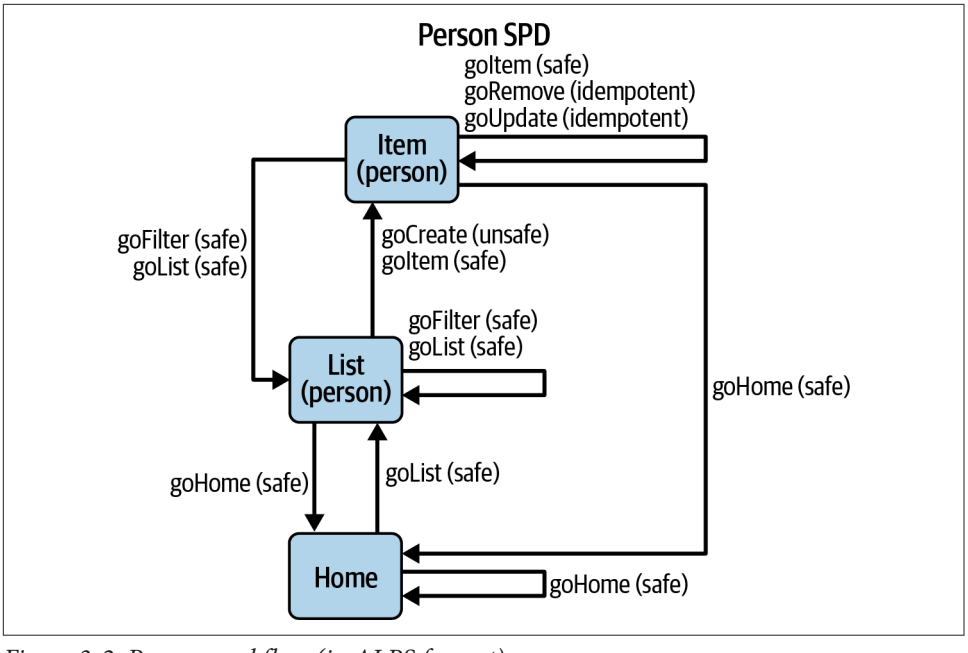

Figure 3-2. Person workflow (in ALPS format)

Discussion

Adopting SPDs is an important part of creating usable RWAs. However, this approach is relatively new and is not yet widely used. Even though ALPS and DCAP have been around for about 10 years, there aren't a lot of tools and only limited written guidance on semantic profiles. Still, my own experience with ALPS leads me to think you'll be seeing more on semantic profiles (even if it is in some future form other than ALPS and DCAP).

An SPD is a kind of machine-readable interface documentation. SPDs are designed to communicate general elements of the interface (base-level properties, aggregate objects, and actions). SPDs do not contain implementation-level details, such as MQTT topics, HTTP resources, protocol methods, return codes, etc. These details are left to those tasked with writing the actual code that runs *behind* the interface described by semantic profiles.

Some other considerations when using semantic profiles:

Aim for wide (re)use.
: The value of a single semantic profile increases with the number of services using that profile. That means you should create profiles that are easily used by others (see Appendix A). It's a good practice to keep semantic profiles more general

than specific. The more specific you make your profile, the smaller the audience for that profile.

Don't use semantic profiles as API definitions.
: Semantic profiles are *descriptions*, not *definitions*. Many of the details you need to include in an interface definition do not belong in semantic profiles. For example, don't include URLs or URL patterns in semantic profiles. Instead put them in the API definition file (e.g., OpenAPI, etc.).

Use semantic profiles to describe your information architecture.
: A good practice is to *tag* your semantic profile elements to indicate which ones are describing the ontology, which are taxonomy, and which are choreography (see "Example" on page 69).

Share semantic profiles in all responses.
: It's a good idea to return the URI of your semantic profile with each protocol response. See Chapters 4 and 5 for details on how to do this.

Limit changes to published semantic profiles.
: Since clients and/or services may create a logical dependency on your published semantic profile, it is a good practice to *not* make any breaking changes to the SPD once it is released. If you need to make changes, it is better to create a new profile document at a new URI (e.g., *api.example.org/profiles/personV2*) and to leave the existing profile (the one without the changes) online, too.

Make your semantic profiles accessible from a central location.
: To make it easy for people (and machines) to find your semantic profiles, create a central online location where they can be easily found. This might be a set of actual profile documents or a page that has pointers (URLs) to other places where each profile document is kept. This second option is good for cases where you are not the profile author and you still want to have some control over which profiles are commonly used.

See Also

- Recipe 3.1, "Creating Interoperability with Registered Media Types"
- Recipe 3.5, "Expressing Actions at Runtime with Embedded Hypermedia"
- Recipe 4.4, "Coding Effective Clients to Understand Vocabulary Profiles"
- Recipe 4.6, "Managing Representation Formats at Runtime"
- Recipe 4.9, "Relying on Hypermedia Controls in the Response"

3.5 Expressing Actions at Runtime with Embedded Hypermedia

Using embedded hypermedia within responses in order to inform API consumer applications what actions are currently possible is a foundational element in RWAs. As mentioned in "Alan Kay's Extreme Late Binding" on page 8, hypermedia controls make it possible to implement *super loose coupling*, and that means you can more easily modify services in the future, too.

Problem

How can you extend the lifetime of services by building systems that support safe, nonbreaking workflow changes as well as short-term modifiability of existing services by customizing data exchanges at runtime?

Solution

The best way to support both short- and long-term modifiability for product services is to rely on inline hypermedia affordances to express the details of context-dependent data exchanges at runtime. That means you need to adopt message exchange formats that support embedded hypermedia (see Recipes 3.1 and 3.2).

Example

HTML offers a great example of embedded hypermedia controls to express data exchanges at runtime. Here's an example:

```
<html>
  <head>
    <title>Create Person</title>
    <link rel="profile" href="http://api.example.org/profiles/person" />
    <style>
      input {display:block;}
    </style>
  </head>
  <body>
    <h1>Create Person</h1>
    <form name="doCreate" action="http://api.example.org/person/"
      method="post" enctype="application/x-www-form-urlencoded">
      <fieldset>
        <hidden name="identifier" value="q1w2e3r4" />
        <input name="givenName" placeholder="givenName" required/>
        <input name="familyName" placeholder="familyName" required/>
        <input name="telephone" placeholder="telephone" pattern="[0-9]{10}"/>
        <input type="submit" />
        <input type="reset" />
        <input type="button" value="Cancel" />
      </fieldset>
```

```
    </form>
  </body>
</html>
```

This form indicates one default input (`identifier`) and three additional inputs, two of which are required (`givenName`, `familyName`) and one which *must* pass the `pattern` validator (`telephone`). The form also indicates three possible actions (`submit`, `reset`, and `cancel`) along with details on the URL, HTTP method, and body encoding metadata for the `submit` action. Even better, any HTML-compliant client (e.g., a web browser) can support all these actions without the need for any custom programming code.

Here's a simple example in Collection+JSON:

```
{ "collection" :
  {
    "version" : "1.0",
    "href" : "http://api.example.org/person/",

    "links": [
      {"rel": "self", "href": "http://api.xample.org/person/doCreate"},
      {"rel": "reset", "href":"http://api.example.org/person/doCreate?reset"},
      {"rel": "cancel", "href":"http://api.example.org./person"}
    ],
    "template" : {
      "data" : [
        {"name" : "identifer", "value": "q1w2e3r4"},
        {"name" : "givenName", "value" : "", "required":true},
        {"name" : "familyName", "value" : "", "required":true},
        {"name" : "telephone", "value" : "", "regex":"[0-9]{10}"}
      ]
    }
  }
}
```

Again, a Collection+JSON-compliant client application would be able to enforce all the input rules described in the hypermedia response without the need for any added programming.

In both cases, the *meaning* of the values of the input elements and metadata properties do not need to be understood by the API consumer—they just need to be enforced. So the number of inputs, the destination URL of the HTTP POST, and even some of the rules can all change over time, and the API consumer application can still reliably enforce the constraints. For example, the validation rule for the `telephone` value can change (e.g., it might be context dependent based on where the application is running).

Discussion

Adopting embedded hypermedia messages is probably the most important step toward creating RESTful web APIs. There are a number of acceptable structured media types (see "Viable Registered Media Types for RESTful Web APIs" on page 421) to choose from, and supporting them with a parsing library is a one-type expense that pays off every time you use the library.

For an in-depth look at hypermedia client applications, see my book *RESTful Web Clients*.

While embedded hypermedia is valuable, using it does come at some cost. First, it is a *design-time* constraint. Both API consumers and producers need to agree to use it. This is "the price of entry" when creating RWAs. While this is not different than "you must use HTML, CSS, and JavaScript in responses to web browsers," I still find many architects and developers who chafe at the idea of using hypermedia-rich responses. When you decide to build RWAs, you may need to help some people past this hurdle. The material in Part I can be helpful for this.

Selecting the "right" structured hypermedia media type can easily turn into a battle. There are multiple types to pick from, and some people fall into the trap of "you can only pick one." In truth, you can use features designed into HTTP (see Recipe 5.6) to help support multiple response formats and select the best option at runtime. This is 30-year-old tech, so there are lots of examples and supporting code bits to help you handle this. You can even decide on a single format to start (usually HTML) and then add more types over time without breaking any existing applications.

Expressing domain state using hypermedia types can also be a challenge at times. Developers need to be able to convert internal object and model rules into valid hypermedia controls (`forms`, `inputs`, `links`, etc.). It is a kind of translation skill that must be acquired.

The work of emitting and consuming hypermedia formats at runtime is a common task, too. See Recipe 5.6 for details on how to use HTTP to help with that. Also, writing API consumers that can navigate hypermedia responses takes some skill. Many of the recipes in Chapter 4 are devoted to this work.

Other things to consider when using embedded hypermedia formats:

Supporting context changes
: You can modify the hypermedia details in a response based on the user context and/or server state. For example, a form might have five fields when an authenticated administrator logs in, but only have three input fields for anonymous users.

Enabling service location changes
: When you use hypermedia controls to describe potential actions, you can engage in Alan Kay's "extreme late binding." Hypermedia clients are designed to look for identifiers in the message ("where is the `update-customer` control?"), and to understand and act upon the metadata found in that hypermedia control. All of this can happen at runtime, not design or build time. That means the metadata details of an action can be changed while the service is in production. For example, you can change the target URL of an action from a local endpoint within the current service to an external endpoint on another machine—all without breaking the client application.

Adapting to workflow changes
: Changes in multistep processes or workflows can also be enabled with hypermedia. Service responses can return one or more steps to complete and expect the client application to act on each one. There might be three steps in the initial release (create account, add associated company, and add a new contact). Later, these three steps might be consolidated into two (create account, add associated company and contact). As long the client service consumer is following along with the supplied hypermedia instructions (instead of hardwiring the steps into code), changes like this will not break the consumer application. See Chapter 7 for recipes on implementing hypermedia workflows.

See Also

- Recipe 3.1, "Creating Interoperability with Registered Media Types"
- Recipe 3.2, "Ensuring Future Compatibility with Structured Media Types"
- Recipe 4.9, "Relying on Hypermedia Controls in the Response"
- Recipe 4.11, "Validating Data Properties at Runtime"
- Recipe 5.6, "Supporting HTTP Content Negotiation"

3.6 Designing Consistent Data Writes with Idempotent Actions

A particularly gnarly problem when working with external services in a machine-to-machine situation over HTTP is the "failed POST" challenge. Consider the case where a client application issues an HTTP POST to an account service to deduct 50 credits from an account and *never gets a reply* to that request. Did the request never arrive at the server? Did it arrive, get processed properly, and the client never received the `200 OK` response? The real question is: should the client issue the request again?

There is a simple solution to the problem, and it requires using HTTP `PUT`, not `POST`.

Problem

How do you design write actions over HTTP that remove the possibility of "double-posting" the data? How can you know whether it is safe to resend a data-write for the HTTP request if the client app never gets an HTTP response the first time?

Solution

All data-write actions should be sent using HTTP `PUT`, not HTTP `POST`. HTTP `PUT` actions can be easily engineered to prevent "double-posting" *and* ensure it is safe to retry the action when the server response never reaches the client application.

Example

Influenced by the database pattern of CRUD (create, read, update, delete), for many years the "create" action has been mapped to HTTP `POST` and the "update" action has been mapped to HTTP `PUT`. However, writing data to a server with HTTP `POST` is a challenge since the action is not consistently repeatable. To say it another way, the POST option is not *idempotent*. That is, POST does not return the same result when repeated for the same resource. However, by design, the HTTP `PUT` operation *is* idempotent—it assures the same results even when you repeat the action multiple times.

To avoid the possibility of inadvertently creating two resources when only one was needed, the best solution is to always use HTTP `PUT` to write data to another machine on the network:

```
**** REQUEST
PUT /person/q1w2e3 HTTP/2.0
Host: api.example.org
Content-Type: application/x-www-form-urlencoded
If-None-Match: *

givenName=Mace&familyName=Morris

**** RESPONSE
HTTP/2.0 201 CREATED
Content-Type: application/vnd.collection+json
ETag: "p0o9i8u7y6t5r4e3w2q1"
...
```

Note that, while not common, it is proper for HTTP to have a `PUT` request result in a `201 CREATED` response—if there is no resource at that address. But how do you tell the service that you are expecting to create a new resource instead of update an existing one? For that, you need to use `If-None-Match`. In the preceding example, the `If-None-Match: *` header says "create a new resource at the supplied URL if there is no existing resource at this URL."

If I wanted the service to treat the HTTP PUT as a replacement of an existing resource, I would use the following HTTP interaction:

```
**** REQUEST
PUT /person/q1w2e3 HTTP/2.0
Host: api.example.org
Content-Type: application/x-www-form-urlencoded
If-Match: "p0o9i8u7y6t5r4e3w2q1"

givenName=Mace&familyName=Morris

**** RESPONSE
HTTP/2.0 200 OK
Content-Type: application/vnd.collection+json
ETag: "o9i8u7y6t5r4e3w2q1p0"
...
```

Here, the HTTP request is marked with the `If-Match:` header that contains the entity tag (or ETag) that identifies the exact version of the resource at `/person/q1w2e3` you wish to update. If the ETag at that URL doesn't match the one in the request (e.g., if the resource doesn't exist or has been updated by someone else lately), then the HTTP `PUT` will be rejected with an `HTTP 412 Precondition Failed` response instead.

Discussion

This use of the PUT-Create pattern is a way to simplify the challenge of knowing when it is OK to retry an unsuccessful write operation over HTTP. It works because by design, `PUT` can be retried without creating unwanted side effects. `POST`—by design—doesn't make that promise. There have been a handful of attempts to make POST retry-able. Two examples are Bill de hÓra's HTTPLR and Mark Nottingham's POE. Neither gained wide adoption.

Using HTTP `PUT` to create new resources takes a bit more work up front—both for clients and servers—primarily because it depends on proper use of the HTTP headers `If-None-Match`, `If-Match`, and `ETag`. In my experience, it is best to "bake" this recipe into code for both the server and the client applications. I typically have "cover methods" in my local code called `createResource(…)` and `updateResource(…)` (or something similar) that know how to craft a proper HTTP `PUT` request (including the right headers) and how to respond when things don't go as planned. The convenience of these cover methods lowers the perceived added cost of using PUT-Create and ensures it is implemented consistently across client and server.

See related recipes on creating and updating resources in Chapters 4 and 5.

Another recipe that is wrapped up in this one is the ability for client applications to supply resource identifiers. When using POST, the service is usually expected to create a new resource identifier—typically a monotonic ID like `/person/1`, `/person/2`, and so forth. When using PUT, clients are expected to supply the resource identifier (think "Please upload the file *my-todo-list.doc* to my document storage"). See Chapters 4 and 5 for more on that.

Designing in this support for idempotent writes with the PUT-Create pattern means both your client and server applications will be more reliable and consistent—even if the network connections you are using are not.

See Also

- Recipe 3.8, "Designing for Repeatable Actions"
- Recipe 4.2, "Coding Clients to Be HTTP Aware"
- Recipe 5.15, "Improving Reliability with Idempotent Create"
- Recipe 6.2, "Making All Changes Idempotent"

3.7 Enabling Interoperability with Inter-Service State Transfers

An important part of creating a collection of interoperable services on the web is making sure each service can act independently and be enlisted in a larger goal or solution. We need to make sure our service APIs do not assume that the only valid use case is when clients are "captives" of that service. Services need to be seen as "part" of a whole, not just a "whole."

Problem

How can we make sure our service interfaces work even when that service is enlisted as one part of a larger solution? A solution that has been designed by someone else (the API consumer application) and that integrates with other APIs. What does it take to ensure our services can be easily integrated with other services that I haven't seen before and don't control, while still maintaining system safety and data integrity?

Solution

Establishing the principle that your service should be able to interoperate with other, unknown, services can be a challenge. The easiest and safest way to do this is to implement the API as a set of standalone, stateless operations. These can accept inputs, perform an action, return results, and forget the whole experience.

Computing values is a good example of a simple, stateless transfer between services. Consider a postal code validator service that might support passing in a body with a full address (and possibly some identity metadata) and returning an associated postal code. All the needed state data is transferred in the request/response pair.

However, there will be times when you need to transfer a block of data into a service, allow that service to operate on that data for a while, and then export the results. In these cases, more explicit import/export operations may be a good solution.

Using FORMS

The simplest way to transfer state between services is by passing data properties using a form supplied by the service *receiving* the state information. Fundamentally, this is how HTML and the other hypermedia formats work. However, some services may require additional metadata be transferred in order to process the incoming data, or may not support native hypermedia controls. In those cases, you need to design additional support in your interface to explicitly transfer state data.

Using import/export operations

The next easiest way to create a service that is easy for someone else to use is to support a step that allows transferring state from one service to another. Usually this means offering an operation that imports (`importState`) state from somewhere else, and an operation that exports (`exportState`) your service's state to somewhere else.

Instead of adding explicit import/export actions to your interface, you can also build that functionality into selected existing actions of the API. For example, you can arrange to accept inputs (via a FORM element) when creating a new user account.

The preceding use cases are examples of transferring state "by value"—sending the exact state values from one service to the next using runtime FORMS descriptions. This is the easiest way to interoperate with other services you don't control.

Shared references

You can also arrange to transfer state "by reference"—by sharing a URL that points to the collection of data you wish to transfer. This takes a bit more coordination between services since both of them need to know, ahead of time, about the idea of shared data via URLs, and the format in which the shared data is stored. See Recipe 7.2 for more details.

Example

There are three ways you can enable state transfer between services:

- Pass by value using inline existing FORMS
- Pass by value using dedicated operations (e.g., `importState` and `exportState`)
- Pass by reference (via a shared URL) using dedicated operations or a FORM

Pass by value using existing FORMS

Transferring state data between services via hypermedia forms is the simplest and most direct way to solve the problem:

```
<form action="http://api.example.org/shopping/cart"
  method="post" name="cartCreate">
  <input name="cartId" value="q1w2e3r4t5y6u7i8" />
  <input name="cartName" value="Mike's Cart" />
</form>
```

This works well when the state data collection is small and the receiving service doesn't require that an existing stateful session exist beforehand.

Pass by value using dedicated operations

You can also support state transfer by adding import and export operations to your service interface. For example, your service might offer a shopping cart management experience. Along with the usual steps (`create`, `addItem`, `removeItem`, and `checkout`), you could add a step that imports an existing shopping cart collection and one that exports that collection.

Here is a snippet of ALPS that shows some of the actions of a simple shopping API that supports import and export operations:

```
{
  $schema: "https://alps-io.github.io/schemas/alps.json",
  alps: {
    version: "1.0",
    title: "Simple Shopping Cart",
    doc: {value: "A simple shopping cart service"},

    "descriptor" : [
      {
        "id" : "doCartImport",
        "type" : "idempotent",
        "rt" : "#cartCollection",
        "tag" : "choreography",
        "descriptor" : [
          "href" : "#cartCollection"
        ]
      },
```

```
        {
          "id" : "doCartExport",
          "type" : "idempotent",
          "rt" : "cartCollection",
          "tag" : "choreography"
        }
      ]
    }
  }
```

This works well when the amount of state data to transfer is relatively large and/or complex (e.g., a collection of 15 items plus metadata). The upside is you get to transfer a lot of data in a single step. The downside is you need to coordinate this kind of state transfer ahead of time since both parties must agree on the data property names, contents, and the general shape of the collection to be transferred.

Pass by reference

You can also arrange to pass state data between services by sharing a "pointer" to a state collection stored somewhere else that is reachable by both parties. The easiest way to do this is to share a URL that points to a resource that will respond with the media type and vocabulary format expected (or explicitly requested) by the calling party.

The interaction would look like this (in HTML):

```
<form action="http://api.example.org/users/q1w2e3r4"
  method="post" enctype="multipart/form-data">
  <input type="file" name="userData" accept="application/vnd.collection+json"/>
</form>
```

This example describes an "upload" operation that the target service can implement. Again, the key to success here is that both parties have already agreed to the expected data format and vocabulary contents ahead of time. You can document these transfer operations when you publish your API to let others know how your service interface expects to send information to or receive information from other services.

Discussion

If at all possible, supply hypermedia forms in your API responses that handle state transfers in a single step. This will allow other services to act as API clients and execute the transfer directly. This requires no additional coordination between clients and servers.

Also, keep the orchestration between services to a minimum. For example, it's best to handle state transfers in a single step instead of requiring API consumers to first start a session or log in before doing other actions. If identity is needed, make that interaction the result of a redirection from the target state transfer:

```
**** REQUEST ****
POST /shopping/cartCreate HTTP/1.1
Host: api.example.org
Content-Type: application/x-www-form-urlencoded

cartId=q1w2e3r4&cartName=Mike's%20Cart

**** RESPONSE ****
HTTP/1.1 Unauthorized
WWW-Authenticate: Basic

**** REQUEST ****
POST /shopping/cartCreate HTTP/1.1
Host: api.example.org
Content-Type: application/x-www-form-urlencoded
Authorization: Basic q1i8w2o9e3p0r4u7t5...

cartId=q1w2e3r4&cartName=Mike's%20Cart

**** RESPONSE ****
HTTP/1.1 200 OK
Content-Type: application/vnd.collection+json
Authorization: Basic: q1i8w2o9e3p0r4u7t5...
Link: <http://docs.alps.io/shopping.json>; rel="profile"

{
  "alps" : {
    ...
  }
}
```

If you decide to support "pass by reference" state transfers, it is best to use structured media types like HAL, SIREN, Collection+JSON, etc. as the message format, and to supply vocabulary references (e.g., ALPS URIs) when passing the data. This will improve the chances that the uploaded data is only accepted when the receiving service confirms the format and the vocabularies are "understood" by the receiving party.

Dealing with Cross-Origin Resource Sharing Limitations

Beware that HTML browsers will not support cross-origin POSTing of messages or uploading of documents from "external" domains. This has to do with the Cross-Origin Resource Sharing (CORS) limitations implemented by HTML browsers. If you are implementing an API that will support a "by reference" state transfer (e.g., a file upload), you may also need to emit headers (e.g., `Access-Control-Allow-Origin`, etc.) to allow API consumers to successfully upload their documents.

See Also

- Recipe 4.15, "Maintaining Your Own State"
- Recipe 6.8, "Ignoring Unknown Data Fields"
- Recipe 7.2, "Supporting Shared State for Workflows"

3.8 Designing for Repeatable Actions

On the web, there are lots of chances that a single request might not get through, might be invalid, or might just "disappear" with no response. This inherent unreliability of the network means we need to design in ways to survive and—whenever possible—overcome these problems in order to create service interfaces that are dependable over time. This recipe relies on the concept of *idempotence* to make HTTP requests repeatable when necessary.

Problem

Sometimes an HTTP request fails due to network problems or some other transient problem. What can we do to make sure our failed HTTP requests can be reliably repeated without causing additional problems? What is idempotence, how does it work, and how can we design in support for repeatability for our service interfaces?

Solution

HTTP was initially designed to take into account the possibility that some requests might fail. Usually, these failures are transient network problems, and repeating the request will resolve the problem. Repeating *safe* HTTP requests—ones that are read-only—is not a hard problem. But repeating HTTP requests that modify resources can be much trickier. See Recipe 7.16 for some common solutions to this problem.

Network idempotence

There are two things you can do to improve the possibility of repeating HTTP requests. First, you can use HTTP's *idempotent* methods (`GET`, `PUT`, and `DELETE`) in order to read, write, and remove resources. That means *not* relying on HTTP's `POST` (since it was not built to support idempotent updates). Whenever possible, I rely on `PUT` to create new resources (see Recipe 5.15). It takes a bit more work to set up `PUT` for create, but it greatly improves the reliability of your service.

Operation idempotence

The second thing you can do is design write operations themselves to support idempotences. This often means *not* using patterns like increments ("add one to each

total") or computing percentages ("increase the sale price by 5%") and instead using replacement values and validators ("if the current price is 100, change it to 105").

By relying on HTTP's `PUT` to update a resource and by designing the body of updates to use replacement rather than increment, you can increase the repeatability of the write operations for your service interface.

Example

Both network- and message-based repeatability are important. This section offers examples of each.

Network repeatability

Designing resource creation and update to use HTTP's PUT method uses the idempotence of PUT to support repeatability at the network level:

```
**** REQUEST ****
PUT /onboarding/q1w2e3r4
Host: api.example.org
Content-Type: application/x-www-form-urlencoded
Content-Length: NN
...

id=q1w2e3r4&givenName=Mork&familyName=Morkelson&...

**** RESPONSE ****
503 Service Unavailable
Content-Type: application/problem+json
Content-Length: XX
...
{
"type": "https://example.com/probs/unavailable",
"title": "Server Unavailable",
"detail": "Can't reach target server",
"instance": "/onboarding/q1w2e3r4",
"status": 503
}
```

In this example, the PUT failed due to a problem with another server along the way. Since PUT is idempotent, it is safe to repeat this request again. In fact, you could repeat this request even if you never got an HTTP response at all (a rare, but possible outcome).

Message repeatability

Even when the network-level interaction is idempotent (e.g., you are using POST instead of PUT), you can improve repeatability of an HTTP request by making sure the body of the message supports functional idempotence.

For example, when updating all the prices in your product catalog, you can provide a nonidempotent update, like this:

```
**** REQUEST ****
PUT /catalog/priceUpdate
Host: api.example.org
Content-Type:application/x-www-form-urlencoded
Accept: application/vnd.siren+json
....

updatePercent=.05
```

In this example, when the PUT request is received, the value of `updatePercent` is used to update each of the 55 products in the company catalog.

But what happens if the update fails halfway through the list of products? Some products were updated, some were not. Or what if we never get a response message for the request? Was is successfully applied? To all products? It is safe to rerun this action?

We can greatly improve the repeatability of this example by redesigning it as idempotent at the message-level update:

```
**** REQUEST ****
PUT /catalog/priceUpdate
Host: api.example.org
Content-Type:text/csv
Accept: application/vnd.siren+json
....

productId, currentPrice,newPrice
q1w2e3, 100,105
t5y6u7, 200,210
i8o9p0, 250,265
i8y6r4, 50,55
...
```

Notice that this update message includes the `productId`, `currentPrice`, and `newPrice` properties. Now this action can safely be repeated. The service can check to see if the catalog price for a particular `productId` is set to `currentPrice` before applying the `newPrice` value. Otherwise, the operation can skip the price update for that record and move on to the next.

Another possible solution for this example is to simply include the `productId` and `newPrice` values. Running the process multiple times will still result in the proper pricing every time.

It's important to not just rely on the HTTP method to ensure a message's repeatability.

Discussion

Designing repeatable updates often means doing more work up front to better explain the changes you want to apply to a resource (or collection of resources). This is definitely more work, but easily pays for itself the first time you have an unexpected crash in the middle of applying a large update.

PUT or POST?

Because repeatability is so important, most of the examples you'll see in this book use `PUT` instead of `POST` for creating new resources and updating existing ones. It is possible to make `POST` idempotent by adding an idempotency key like the one proposed for the IETF. I still use `PUT` most of the time for my designs. See Recipe 3.6 for additional discussion on this topic.

See Also

- Recipe 3.6, "Designing Consistent Data Writes with Idempotent Actions"
- Recipe 5.15, "Improving Reliability with Idempotent Create"
- Recipe 6.2, "Making All Changes Idempotent"
- Recipe 7.1, "Designing Workflow-Compliant Services"

3.9 Designing for Reversible Actions

There will be cases where, after applying a change to one or more resources, you will need to roll back or undo that change. This recipe shows how you can make support for reversibility part of your service interfaces.

Problem

What happens when you want to back out or roll back an update to one or more resources? How can you design your updates to make reversibility safe and easy to implement and use?

Solution

When you need to roll back an update, there are two basic ways to solve the problem:

- Support a local undo (see Recipe 7.17)
- Send another HTTP request that reverses the previous one

The first example (support a local undo) is covered in Recipe 7.17, so I won't get into those details here. Suffice it to say that the local undo works for cases where there are no other dependent actions (e.g., you have no other services that were modified as a consequence of the update you wish to reverse). If, however, your update has affected other records in the system or your update is just one of several that might be applied to the same resource, then supporting reversibility can be more complicated.

Issue a second request

For cases where only a single record has been affected (e.g., a data-centric resource updates a system of record), you can usually issue another HTTP request that has a rollback effect on the resource. For example, if you issued an HTTP `PUT` to replace the `givenName` property of a record, you can usually issue another HTTP `PUT` to replace the same property with the previous value. This assumes you *know* the previous value and safely make the change (e.g., no one else has already modified the record).

Issue a special request

There are cases where a simple re-`PUT` will not work. You might have issued an HTTP `DELETE` for a resource and want to recover that resource. There is no HTTP `UNDELETE` method. Instead you need to restore an existing record with a special command. The safest way is to create another, special operation (`undoDelete`) that uses HTTP `PUT` and some argument (usually the previous URL along with a validation value like the ETag header). This means the service interface needs to be designed to support this feature, including saving `DELETE`d resources in case they need to be restored!

Example

You can use the two PUTs approach or a special command to add reversibility support to your designs.

Rollback with two PUTs

Here's an example where you can issue a second PUT to undo the first:

```
**** REQUEST ****
GET /users/q1w2e3r4
Host: api.example.org
Accept: application/html

**** RESPONSE ****
200 OK
Content-Type: application/html
ETag: "w/p0o9i8u7y6t5r4"
...

<html>
```

```
...
<div class="user">
  <span class="givenName">Millie</span>
  <span class="familyName">Murcheson</span>
</div>
...
</html>
```

Here's the first update:

```
**** REQUEST ****
PUT /users/q1w2e3r4
Host: api.example.org
Content-Type: application/x-www-form-urlencoded
Accept: application/html
If-Match: "w/p0o9i8u7y6t5r4"

givenName=Molly&familyName=Murcheson

**** RESPONSE ****
200 OK
Content-Type: application/html
If-Match: "w/q2w3e4r5t6y7u8"
...

<html>
...
<div class="user">
  <span class="givenName">Molly</span>
  <span class="familyName">Murcheson</span>
</div>
...
</html>
```

And, finally, we can roll back the change with another HTTP `PUT`:

```
**** REQUEST ****
PUT /users/q1w2e3r4
Host: api.example.org
Content-Type: application/x-www-form-urlencoded
Accept: application/html
If-Match: "w/q2w3e4r5t6y7u8"

givenName=Millie&familyName=Murcheson

**** RESPONSE ****
200 OK
Content-Type: application/html
ETag: "w/i9u8y7t6r5e4"
...

<html>
...
```

```
<div class="user">
  <span class="givenName">Millie</span>
  <span class="familyName">Murcheson</span>
</div>
...
</html>
```

Note the use of the `ETag` to "version" the response and `If-Match` headers to make the PUT conditional.

Special command approach

In more involved cases, you'll need to use a special rollback action on your service interface.

Here is the action that removes an existing resource from the collection:

```
**** REQUEST ****
DELETE /users/q1w2e3r4
Host: api.example.org
Accept: application/html
If-Match "w/y6t5r4e3w2q1"

**** RESPONSE ****
204 No Content
....
```

And here is a special command for the interface that supports rolling back the `DELETE` action:

```
**** REQUEST ****
PUT /users/rollback?id=q1w2e3r4
Host: api.example.org
Accept: application/html
If-Match "w/y6t5r4e3w2q1"

**** RESPONSE ****
201 Created
Location: /users/q1w2e3r4
....
```

Note the use of `201 Created` to confirm the resource has been restored.

Discussion

The advantage of issuing a special request is that client applications don't need to recall details of the original resource in order to undo an update. The downside is that service interfaces need to take responsibility for keeping a history of resource changes.

When all else fails, you may need to solve your reversibility problem the hard way: calling service support for help re-creating the previous conditions on the platform. See Recipe 7.18 for more on this approach.

The URL to use for special rollback commands is nothing special. I've seen teams use the same URL space as the original target resource (`/users/q1w2e3r4`), and I've seen teams use a special rollback resource (`/rollbacks/users/q1w2e3r4`). I've also seen a mix of both (see the preceding example).

For situations where more than one resource is modified by a single action (e.g., `removeCustomerData` affects the customer, orders, and sales data stores), you need to design a special rollback feature for your resource and, if possible, make sure all three dependent services support their own rollback or, if possible, implement your own version of a local undo (see Recipe 7.17) for each of the other services. This can be a bit tricky, and you may need to get cooperation from the owners of the other services in order to support the rollback properly.

See Also

- Recipe 7.1, "Designing Workflow-Compliant Services"
- Recipe 7.17, "Supporting Local Undo or Rollback"
- Recipe 7.18, "Calling for Help"

3.10 Designing for Extensible Messages

Usually, once a service interface experiences wide use, ideas for improving that interface start to appear. One of the ways to improve the interface is to extend the message body of the HTTP response type. This recipe shows you how to easily design in a way to extend the message body in the future without breaking any existing API consumers.

Problem

You are getting requests to modify the response body of an API call. Some want new properties added, others want a scalar property (e.g., `phoneNumber`) turned into an array (`phoneNumbers`). How can you make these kinds of extensions without breaking any API consumers already using the current release? What are the limits of extending messages?

Solution

An effective approach to modifying response bodies that are already in production is to adopt the "Don't Change It, Add It" rule (see "Don't change it, add it" on page 35).

That means you leave any existing elements in place and simply include new elements in the current response body. You can also design your response formats to support a simple name-value-pair collection where you can place additional new properties (of any type, including arrays). See the following examples for details.

Example

Consider a simple HTTP response like this message body:

```
{
  "name": "Merk Muffly",
  "region", "southwest",
  "age": 21
}
```

There are a number of possible ways to extend this object to improve its ability to accommodate requested changes in the future.

Add a properties collection

You can add new properties to the object by including a name-value pair (NVP) collection as part of the initial design:

```
{
  "name": "Merk Muffly",
  "region", "southwest",
  "age": 21,
  "nvp" : [...]
}
```

Now you can add any other kind of property (scalar, array, object) without disrupting the existing structure:

```
{
  "name": "Merk Muffly",
  "region", "southwest",
  "age": 21,
  "nvp" : [
    {"hatsize" : "3"},
    {"phoneNumbers": ["123-456-7890","980-657-3421"]},
    {"address": {"street":"...","city":"...","state":"...","zip":"..."}}
  ]
}
```

Add parallel properties

There are times when you need to modify or extend an existing property into multiple related properties. For example, you might need to improve the `name` property and extend it to use `givenName` and `familyName`. Instead of *replacing* the `name` property, you can just *add* the two new properties:

```
{
  "givenName": "Merk",
  "familyName": "Muffly",
  "name": "Merk Muffly",
  "region", "southwest",
  "age": 21,
  "nvp" : [
    {"hatsize" : "3"},
    {"phoneNumbers": ["123-456-7890","980-657-3421"]},
    {"address": {"street":"...","city":"...","state":"...","zip":"..."}}
  ]
}
```

Now, when returning the message in HTTP response, both formats are supported. API consumers that understand the `name` property will still find it, and API consumers that understand the new properties (`givenName` and `familyName`) will find what they expect.

It is important to point out that the rules of message bodies *sent* to the service (via PUT or POST) will need to be modified, too. In this case, the service can be modified to accept all three values (`name`, `givenName`, and `familyName`). When the API client sends just `name`, the service can spit up the passed value and store the results in `givenName` and `familyName`. Conversely, when the API client sends `givenName` and `familyName`, the service can combine those two values and store the results in `name`.

Host multiple output formats

In some cases, the changes to the response message are extensive enough that just modifying a few properties is not enough. Instead, you can return multiple versions of the same information in a single response. To do this, you need to design your initial message to allow for "hosting" multiple output formats. You can do this by using an additional "root" element in all your response designs.

Here's how our current example object looks with an additional "root" to host output formats:

```
{"message" : {
  "person" : {
    "givenName": "Merk",
    "familyName": "Muffly",
    "name": "Merk Muffly",
    "region", "southwest",
    "age": 21,
    "nvp" : [
      {"hatsize" : "3"},
      {"phoneNumbers": ["123-456-7890","980-657-3421"]},
      {"address": {"street":"...","city":"...","state":"...","zip":"..."}}
    ]
  }
}}
```

Now that the important properties are enclosed in the `person` element, we can easily return new releases of the `person` or even other new message elements:

```
{"message" : {
  "personv2": {...},
  "metadata": [...],
  "links": [...],
  "person" : {
    "givenName": "Merk",
    "familyName": "Muffly",
    "name": "Merk Muffly",
    "region", "southwest",
    "age": 21,
    "nvp" : [
      {"hatsize" : "3"},
      {"phoneNumbers": ["123-456-7890","980-657-3421"]},
      {"address": {"street":"...","city":"...","state":"...","zip":"..."}}
    ]
  }
}}
```

As long as the initial design includes the initial enclosing root element, adding new versions of message bodies or entirely new root elements does not disrupt existing API consumers.

Discussion

These design rules are especially handy if you decided to *not* use an existing hypermedia format and are instead creating your own.

Most of the predefined message formats used in these recipes (HTML, HAL, SIREN, and Collection+JSON) already follow these extensibility rules.

A key challenge to the success of these extensibility patterns will be that API consumers may apply strict schema validators (JSON-Schema, XML-Schema, etc.) to incoming messages. You can remind consumers to *not* make this mistake, but you cannot completely prevent it.

Usually, you can apply these same design patterns to existing message formats (e.g., HTML, HAL, etc.) without disrupting existing client applications.

See Also

- Recipe 3.10, "Designing for Extensible Messages"
- Recipe 4.7, "Using Schema Documents as a Source of Message Metadata"

- Recipe 4.12, "Using Document Schemas to Validate Outgoing Messages"
- Recipe 6.10, "Modifying Data Models in Production"
- Recipe 6.11, "Extending Remote Data Stores"
- Recipe 7.1, "Designing Workflow-Compliant Services"

3.11 Designing for Modifiable Interfaces

Just as well-used APIs are likely to generate requests for message format changes, they are also likely to experience requests for modifications to the external interface itself, including URLs, HTTP methods, and input arguments. This recipe offers three rules for updating service interfaces without breaking existing API consumers.

Problem

How can I update the service interface (including changing URLs, HTTP methods, and input arguments) without breaking existing clients? What are the three rules for creating nonbreaking changes? What other principles should I keep in mind when attempting to update an existing service interface?

Solution

Almost every API experiences interface changes. One of the advantages of working in the virtual world is that change is less costly than in the physical product world. But there are still challenges. However, you can make change a part of your design from the very beginning with some general guidelines and three solid rules.

First, do no harm

The phrase "First, do no harm" comes from the Hippocratic Oath. Dating back to the fourth century BCE, the Hippocratic Oath has been taken by many physicians as a pledge to give proper care to patients. This phrase is just one small part of the oath, but it's the best-known part. API deserve the same pledge, too.

Once your API is in production, it takes on a life of its own. When other applications are using your API, those applications become dependent on your API. You, as the author of the API, have a responsibility to all your API users. And one of the very first responsibilities is to "do no harm." If you make a change to a running API that breaks one of your API consumers, you are not following the Hippocratic Oath of APIs.

A fork in the road

Sometimes you'll run into a situation where you must make a breaking change to a production API. For example, you might release an API that allows customers to update their personal information on your servers, and a year later the company decides that it will no longer allow customers to edit that data directly. When this happens, you need to release a new API with the proper functionality and instruct all users of the existing API to update their applications to point to the new interface. While that happens, you need some time when both APIs are available in production.

This process is called "forking" the interface. You create a new API based on the existing one and run both in production until everyone has updated their client applications. Along the way, you can do a few things to make it easy to migrate from the old release to the new one.

Three rules for modifying your service interface

When you decide to update your existing interface, you need to follow these rules:

- Take nothing away
- Don't redefine things
- Make additions optional

Once you've published your API, all the URLs, associated properties (inputs and outputs), and even the HTTP methods in the documentation are *promises* that you can't break. Using hypermedia formats that allow you to include input metadata (FORMS) at runtime can help reduce dependence on protocol details like HTTP methods and input arguments, but changing existing promises its the wrong way to go. Instead, you can *add* new features that provide the needed functionality.

You also cannot redefine the existing elements of the interface. If you define the `?size` query parameter to mean the size of a page of data, you can't later redefine `?size` to mean the size of a hat or shoe. Redefining elements is the same as replacing them, and that's a breaking change.

Finally, any new things can be *added* as optional elements to the interface as long as they don't take anything else away or redefine an existing item. If you come up with more input arguments, you can add those to the FORM, but you cannot make them *required*. If you want to create a new action that has new required inputs, you can define a new API endpoint, method, and argument collection (a new FORM). But you cannot make calling that action required, either.

See Recipe 3.10 for more details on modifying the service interface response body.

Example

If you add a new argument to an input form, you need to make that argument optional, and you should supply a default value, too:

```
<!-- existing search form -->
<form action="..." method="GET" name="findUsers">
  <input name="givenName" value="" required="true" />
  <input name="familyName", value="" required="true" />
  <input type="submit" />
</form>

<!-- updated search form -->
<form action="..." method="GET" name="findUsers">
  <input name="givenName" value="" required="true" />
  <input name="familyName" value="" required="true" />
  <input name="regions" value="all" required="false" />
  <input type="submit" />
</form>
```

Note that these nonbreaking changes work both ways. API consumers should be able to send either form to the server to get results. If the `region` argument is missing, the service should *assume* the default value (`"all"`) was sent.

If you want to modify an action that has new required inputs, you can simply *add* the new action to the interface and allow API client applications to ignore that form until they are updated to handle it.

Here's a form that accepts an order for processing:

```
"actions": [
  {
    "name": "processOrder",
    "title": "Process Order",
    "method": "PUT",
    "href": "/orders/processing/q1w2e3r4",
    "type": "application/x-www-form-urlencoded",
    "fields": [
      { "name": "orderNumber", "type": "hidden", "value": "42" },
      { "name": "productCode", "type": "text", "value": "..." },
      { "name": "quantity", "type": "number", "value": "..." }
    ]
  },
  {
    "name": "ProcessSalesRepOrder",
    "title": "Process Sales Rep Order",
```

```
      "method": "PUT",
      "href": "/orders/salesrep/p0o9i8u7",
      "type": "application/x-www-form-urlencoded",
      "fields": [
        { "name": "orderNumber", "type": "hidden", "value": "42" },
        { "name": "productCode", "type": "text", "value": "..." },
        { "name": "quantity", "type": "number", "value": "..." },
        { "name": "salesRep", "type": "text", "value": "..." }
      ]
    }
  ]
```

Here, a feature has been added that allows sales representatives to get credit for new product sales. But they only get credit if the sales rep's name is sent in with the order. That's a required element of the new `processSalesRepOrder` action. A solid solution is to include both actions and allow client applications to use the one they understand.

As long as you "do no harm," you can modify the existing service interface long after it has been published to production.

Discussion

While documenting your API (and all its changes) is important, it is not enough. Especially in cases where you don't know all the people using your API (see Appendix A), you can't be sure that people will read your documentation or that, once they read it, they have the time or resources to update their API consumer apps.

Changes to the service interface vocabulary (see Recipe 3.3) require the same attention to detail. Be sure to "do no harm" when updating your semantic profiles, too.

You might think you can safely change a few of the interface details of your API without causing API consumer problems, but that's often not the case. This was pointed out by Google's Hyrum Wright when he coined what would later be known as Hyrum's Law. He said, "With a sufficient number of users of an API, it does not matter what you promise in the contract: all observable behaviors of your system will be depended on by somebody."

Internally, one way you can confirm that your interface modifications do not introduce breaking changes is to be sure to apply the existing test suite to the *new* interface. By running all the old tests against the new API, you can come close to seeing how existing client applications will react to your interface changes.

See Also

- Recipe 3.10, “Designing for Extensible Messages”
- Recipe 4.7, “Using Schema Documents as a Source of Message Metadata”
- Recipe 4.12, “Using Document Schemas to Validate Outgoing Messages”
- Recipe 6.10, “Modifying Data Models in Production”
- Recipe 6.11, “Extending Remote Data Stores”
- Recipe 7.1, “Designing Workflow-Compliant Services”

CHAPTER 4
Hypermedia Clients

The good news about computers is that they do what you tell them to do. The bad news is that they do what you tell them to do.

—Ted Nelson

Writing applications that use APIs to consume external services on the web requires a mix of specificity (*what* to do) and generality (*how* to do it) that can be challenging. The recipes in this chapter are focused on both *what* we tell client applications to do using local code and *how* we tell them via the protocol and messages we send back and forth. This combination of "what" and "how" makes up the foundation of stable, yet flexible, API consumer apps.

For more information on the art of creating API consumers, check out "Increasing Resilience with Hypermedia Clients" on page 26.

As a general rule, being very explicit about what clients can and cannot do will result in applications that are easy to break and hard to reuse. A better approach is to create API consumer apps that make only a few basic assertions about *how* they communicate (e.g., protocol, message model, and vocabulary) with servers and let the server supply all the other details (the *what*) at runtime. As shown in Figure 4-1, the recipes in this chapter focus on using hypermedia formats over HTTP protocol to support validation, state management, and the creation of goal-oriented client applications. This is what HTML browsers have done for 30+ years. We need to take a similar approach with our API consumers, too.

That doesn't mean we need to duplicate the HTML browser. That client application is focused on a high degree of usability that relies on humans to be the "brains" behind

the interface (or in front of the interface, really). The HTML message format is designed to make it possible to control the visual rendering of messages as well as the animation of content using inline scripting. This focus on UI and humans is not needed for most web service API consumers.

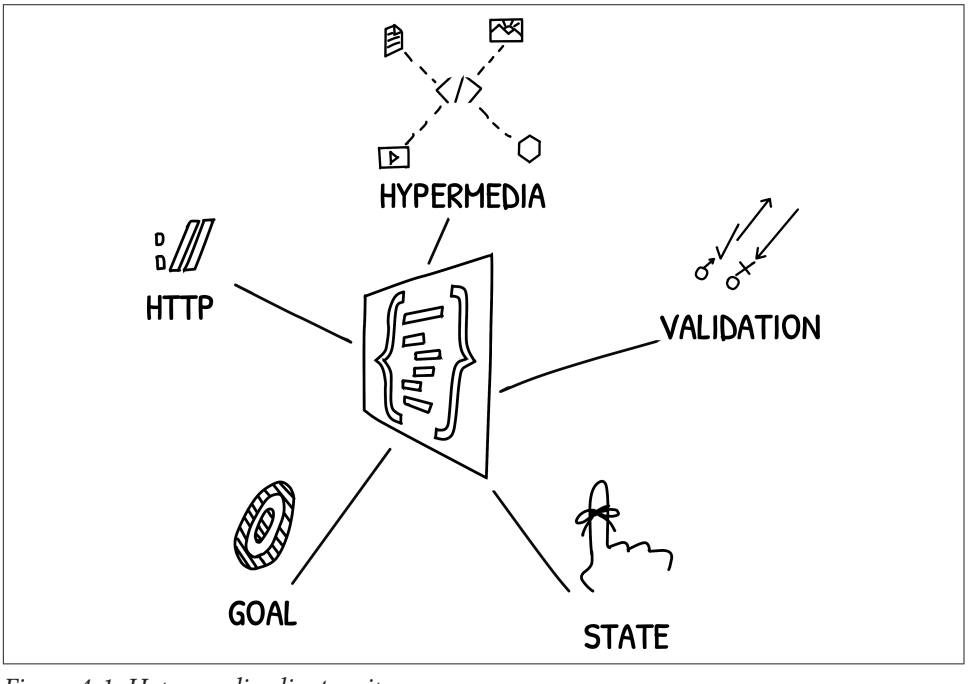

Figure 4-1. Hypermedia client recipes

Instead, for web services, we need to focus on M2M interactions and how to make up for the "missing element"—the human. The recipes in this chapter have been selected because they lead to a high level of reusability without depending on humans to be involved in every step of the client-server interaction. To that end, there is an emphasis on four elements of resilient hypermedia-driven clients:

- Taking advantage of protocols and formats
- Relying on runtime metadata
- Solving the M2M interface challenge
- Using semantic profiles as shared understanding between client and server

With that as a backdrop, let's dig into our set of client-side recipes for RESTful web APIs.

4.1 Limiting the Use of Hardcoded URLs

Changing URLs on the server can cause annoying problems for client applications. Sometimes URL changes occur due to adding or removing features—it is a challenge to protect your client application from these kinds of problems. Sometimes the service URLs change due to "re-homing" or moving the service from one platform or implementation group to another. How can you limit the impact of server URL changes on your client application?

Problem

How can you reduce the chances of a client application failing when service URLs are changed?

Solution

The best way to limit the impact of service URL changes in your client applications is to "abstract away" the actual URL values:

Use named URL variables.
: First, you should code your client so that any URL reference within the application is a named variable, not an actual URL. This is a similar pattern to how you can "localize" your client application for a single language (French, Arabic, Korean, etc.).

Store URL values in configuration files.
: Second, once you have a set of URL variable names, move the actual strings out of the source code and into a configuration file. This makes it possible to update the URLs without changing the source code or recompiling the application. Depending on your platform, you may still need to redeploy your client application after changing the configuration file.

Reduce the number of URLs you need to "memorize" to one.
: If at all possible, limit the total number of URLs your application needs to "know about" in the first place. If you're accessing a service that supports hypermedia formats (Collection+JSON, SIREN, HAL, etc.), your client application will likely only need to "memorize" one URL—the starting or "home" URL (see Recipe 5.1).

Convince the service teams to use hypermedia or configuration files.
: Lastly, if you have influence over the people who are writing the services you use (e.g., your enterprise colleagues), try to convince them to use hypermedia formats that supply URLs at runtime, or at least provide a downloadable configuration file where the service team can store all the named URL variables that your client will need at runtime.

Example

To recap, the four ways you can reduce the impact of service URL changes are:

- Use named URL variables in your code.
- Move the URLs into client configuration files.
- Limit the number of URLs your client needs to know ahead of time.
- Get services to provide the URLs for you.

Using named URL variables

The following is an example of a code snippet showing the list of named URL variables stored in client code:

```
// initializing named variables for service URLs
var serviceURLs = {};
serviceURLs.home = "http://service.example.org/";
serviceURLs.list = "http://service.example.org/list/";
serviceURLs.filter = "http://service.example.org/filter/";
serviceURLs.read = "http://service.example.org/list/{id}/";
serviceURLs.update = "http://service.example.org/list/{id}/";
serviceURLs.remove = "http://service.example.org/list/{id}/";
serviceURLs.approve = "http://service.example.org/{id}/status";
serviceURLs.close = "http://service.example.org/{id}/status";
serviceURLs.pending = "http://service.api.example.org/{id}/status";

/* later in your client code... */

// using named URL variables
const http = new XMLHttpRequest();

// Send a request
http.open("GET", serviceURLs.list);
http.send();

// handle responses
http.onload = function() {
  switch (http.responseURL) {
    case serviceURLs.home:
    ...
    break;
    ...
  }
}
```

Note that some URL values are shared for different names variables. This is often the case when the important information is passed via an HTTP body and/or HTTP method (e.g., HTTP `PUT` versus HTTP `DELETE`). You'll also notice that some URLs contain a templated value (e.g., `http://api.example.org/list/{id}/`). This means

that both client and server MUST share a similar standard for encoding templated URLs. I recommend using libraries that conform to the IETF's RFC 6570.

Loading named URL variables from configuration

Once you have your code set up for only using named URL variables, it is a short step to moving all those URLs outside your codebase into a configuration file:

```
<html>
  <head>
    <script type="text/javascript"
      src="client.example.org/js/service-urls-config.js"></script>
    <script type="text/javascript">
      ...
      // using named URL variables
      const http = new XMLHttpRequest();

      // Send a request
      http.open("GET", serviceURLs.list);
      http.send();

      // handle responses
      http.onload = function() {
        switch (http.responseURL) {
          case serviceURLs.home:
          ...
          break;
          ...
        }
      }
      ...
    </script>
  </head>
  <body>
   ...
  </body>
</html>
```

The configuration file (which looks just like the code in the previous example) is stored at the same location as the client code (*http://client.example.org/js/*). However, it could be stored anywhere the client application can access, including a local file system that is supported by the client platform.

Limit memorized client URLs to one with hypermedia

If your service is using hypermedia formats that automatically supply URLs at runtime, you can limit the number of URLs your client application needs to know ahead of time to possibly *one* URL: the starting or "home" URL. Then your client can "find" any other URLs based on inline information found in `id`, `name`, `rel`, and/or `tags` (see Recipe 5.1):

```
/* find-rel.js */
var startingURL = "http://service.example.org/";
var thisURL = "";
var link = {};

// using named URL variables
const http = new XMLHttpRequest();

// Send a request
http.open("GET", serviceURLs.list);
http.send();

// handle responses
http.onload = function() {
  switch (http.responseURL) {
    case startingURL:
      link = findREL("list");
      if(link.href) {
         thisURL = link.href;
         ...
      }
    ...
    break;
    ...
  }
}
```

Finally, if you can influence the service teams, try to convince them to either use hypermedia types so that client applications need to remember only their starting URL, or have the server team provide a remote configuration file that holds all the named variables that define the static URLs the client will need.

Discussion

The best way to limit the client-side impact of service URL changes is to keep the URLs out of the client application code. The most effective way to do that is for services to use hypermedia formats that provide URLs at runtime, or for services to provide configuration information that contains a list of URLs as named variables. If you can't get that from the service, then it is up to the client application programmers to come as close to that as possible. That means using a client-side configuration file or, if loading configuration data is not viable (e.g., security reasons), then create your own client-side named variable collection in code.

It is important to acknowledge that relying on client-side configuration files is not an ideal solution. Changes in service URLs need to be communicated to the client developer so that the client-side information (config or code) can be updated. This presents a possible lag between service changes and client changes. You can set up client-side monitoring of changes, but that's just adding more work and can only shorten the client application's reaction time to unannounced changes.

It was noted earlier that some URL variables may share the same URL data, but have very different uses. For example, the service URLs for read, update, and delete may be the same (*http://service.example.org/{id}*). This leads to the next step of client applications keeping the entire collection of request metadata in code, too (URL, method, headers, query string, and/or body information). See Recipe 4.10 for more on this option and when to use it.

See Also

- Recipe 4.9, "Relying on Hypermedia Controls in the Response"
- Recipe 4.10, "Supporting Links and Forms for Nonhypermedia Services"
- Recipe 5.1, "Publishing at Least One Stable URL"
- Recipe 7.8, "Returning All Related Actions"

4.2 Coding Clients to Be HTTP Aware

Sometimes helper libraries, wrappers, and/or API software developer kits (SDKs) can make it *harder* for API client applications to solve their problems instead of making it easier. For this reason it is important to make sure your API client applications are able to "converse" with services in HTTP (or whatever the target protocol might be).

Problem

In cases where services offer help classes or SDKs, there are times when the client cannot find the exact local function needed to solve a problem. How can you make sure that client applications can work around any limitations of any helper libraries or SDKs supplied by the service?

Solution

The best way to create API client applications that are both effective and nimble is to make sure they are "protocol-aware"—that they can, when needed, speak directly to a service using the HTTP protocol (or whatever protocol the service is using). To do that, even when your client app takes advantage of SDKs and helper libraries, you

should make sure your client app *also* knows how to "speak HTTP" directly. That will ensure your application can use the service in ways the original service authors had not thought about when creating the initial release.

Clients Teaching Services

It is worth mentioning that service providers may learn a lot by monitoring the ways client applications access their APIs (including the order in which the accesses are made). In some cases, client applications will be teaching services the features and workflows they wish to have. With this information, services can release new editions that make those tasks safer and easier to complete.

It is perfectly acceptable to mix the use of available service SDKs with direct HTTP calls when needed. In fact, it is a good idea to create your own high-level HTTP library that can easily be invoked on demand.

Example

While most modern languages supply HTTP libraries and functions, it is a good idea to create your own high-level HTTP support library. That means handling both the HTTP request and HTTP response details. Here's a simple example of a client-side JavaScript helper library for HTTP calls:

```
var ajax = {

  // setup code here...

  /************************************************************
    *** PUBLIC METHODS ***
    args = {url:string, headers:{}, queryString:{}, body:{},
      callback:function, context:string}
   ************************************************************/
  httpGet: function(args) {...},
  httpGetXML: function(args) {...},
  httpGetJSON: function(args) {...},
  httpGetPlainText: function(args) {...},
  httpPost: function(args) {...},
  httpPostVars: function(args) {...},
  httpPostForXML: function(args) {...},
  httpPostForJSON: function(args) {...},
  httpPostForPlainText: function(args) {...},
  httpPut: function(args) {...},
  httpHead: function(args) {...},
  httpDelete: function(args) {...},
  httpOptions: function(args) {...},
  httpTrace: function(args) {...},
```

```
  // implementation details below ...
}
```

This library can then be called using the following code:

```
function getDocument(url) {
  var args = {};
  args.url = url;
  args.callbackFunction = ajaxComplete;
  args.context = "processLinks";
  args.headers = {'accept':'application/vnd.collection+json'}

  ajax.httpGet(args}

  // later ...

function ajaxComplete(response,headers,context,status,msg)
{
  switch(status) {...} // handle status
  switch(context) {...} // dispatch to context
}
```

This example is psuedocode from a client-side JavaScript implementation focused on browser-based applications. The exact details of your library are not important. What is important is that your client application can quickly and safely craft an HTTP request and handle the HTTP response. Notice that the library is designed to make HTTP calls easier to deal with and does not limit access to the protocol.

Discussion

Not only is it important to offer API client apps direct access to the HTTP protocol, it is important to code the application so that the developer is always aware that HTTP calls are taking place. It might seem like a good idea to "hide the HTTP" under semantic methods like this:

```
var serviceClient = {
  findUser : function(userId) {...},
  assignUserWorkTickets(userId,ticketArray) {...}
}
```

The second call in this example (`assignUserWorkTickets`) might be a minefield. How many HTTP calls are involved? Just one? Maybe one per item in the `ticketArray`? Are the ticket calls sequential, or are they run in parallel? Developers should be able to "see" what they are getting into when they use an SDK or library. Even something as simple as changing the helper library calls from `assignUserWorkTickets` to `sendHttpRequests({ requestList:WorkTickets, parallel:true })` can help developers get a sense of the consequences of their actions.

Also, an HTTP-aware API client doesn't need to be complicated, either. Client-side browser applications can get quite a bit done with the `XMLHttpRequest` object without getting too deep into the weeds of HTTP. See Recipe 4.1 for an example.

The downside of using this approach is that some services might want to protect themselves against inexperienced or possibly malicious client-side developers. They want to use SDKs to control the way client applications interact with the service. This never works to deter malicious actors and often just frustrates well-meaning developers trying to solve their own problems in ways the service provider had not imagined. I've seen many cases where client-side teams have abandoned an API because the required SDK was deemed unsuitable.

See Also

- Recipe 4.1, "Limiting the Use of Hardcoded URLs"
- Recipe 4.6, "Managing Representation Formats at Runtime"
- Recipe 5.1, "Publishing at Least One Stable URL"
- Recipe 5.6, "Supporting HTTP Content Negotiation"
- Recipe 6.6, "Returning HTTP 200 Versus HTTP 400 for Data-Centric Queries"
- Recipe 6.9, "Improving Performance with Caching Directives"

4.3 Coding Resilient Clients with Message-Centric Implementations

To extend the life of the client application and improve its stability, it is a good idea to code client applications to "speak" in well-known media types.

Problem

How can you ensure your client application is more resilient to small changes and not coupled too tightly to a specific domain or running service?

Solution

A great way to build resilient client applications is to code your client app to bind tightly to the message returned by the service, not to a particular service or domain problem. That way, if there are minor changes in the service, as long as the media type response stays the same, your client application is likely to continue to operate correctly.

Example

You have lots of options when coding your API client. You can bind it directly to the service and its documented objects, or you can bind your client to the structured media types that the service emits. In almost all cases, it is better to bind to media types than to service interface types.

Here is an example of a ToDo domain application. This client is implemented by binding to all the actions described in the API documentation:

```
/* to-do-functions.js */
var thisPage = function() {
  function init() {}
  function makeRequest(href, context, body) {}
  function processResponse(ajax, context) {}
  function refreshList() {}
  function searchList() {}
  function addToList() {}
  function completeItem() {}
  function showList() {}
  function attachEvents() {}
};
```

The details of what goes on within each function are not important here. What is key is that each action in the domain (`refresh`, `search`, `add`, etc.) is described *in the source code* of the client. When the service adds any new features (e.g., the ability to `setDueDate` for a task), this client application code will be out-of-date and will not be able to adapt to the new feature. And it would be more challenging if the details changed, such as arguments or other action metadata (e.g., change `POST` to `PUT`, modify the URLs, etc.).

Now, look at the following client that is focused on a message-centric coding model. In this case, the code is not focused on the ToDo domain actions but on the messages (e.g., HTML, HAL, SIREN, etc.) the client sends and receives. In fact, this message-centric approach would be the same no matter the service domain or topic:

```
/* to-do-messages.js */
var thisPage = function() {
  function init() {}
  function makeRequest(href, context, body) {}
  function processResponse(ajax, context) {}
  function displayResponse() {}
  function renderControls() {}
  function handleClicks() {}
};
```

Again, details are missing but the point here is that the functionality of the client application is rooted in the work of making requests, processing the response message, displaying the results, the input controls, and then catching any click events. That's it. The actual domain-specific actions (`refresh`, `search`, `add`, etc.) will be

described *in the response message* via hypermedia control, not the code. That means when the service adds a new feature (like the `setDueDate` operation), this client will "see" the new actions described in the message and automatically render that action. There will be no need to write new code to support new operations.

Binding to the message means that new actions and most modifications of existing actions are "free"—there is no need for additional coding.

Discussion

Shifting your client code from domain specific to message centric is probably the more effective way to create resilient applications for the web. This is the approach of the HTML-based web browser, which has been around for close to 30 years without any significant interface changes. This is despite the fact that the internal coding for HTML browsers has been completely replaced more than once, there are multiple, competing editions of the browser from various vendors, and the features of HTML have changed over time, too.

Much like the recipe to make your API clients "HTTP-aware" (see Recipe 4.2), making them message centric adds an additional layer of resilience to your solution. As we'll see later in this chapter (see Recipe 4.5), there are more layers to this "client resilience cake" to deal with, too. Each layer added creates a more robust and stable foundation upon which to build your specific solution.

One of the downsides of this approach is that it depends on services to properly support strong typed message models for resource responses (see Recipe 4.3). If the service you need to connect with only responds with unstructured media types like XML and JSON, it will be tough to take advantage of a message-centric coding model for your client applications. If you have any influence over your service teams (e.g., you're in a corporate IT department), it can really pay dividends if you can convince them to use highly structured media types like HTML, HAL, SIREN, Collection +JSON, and others. See Chapter 3 for several recipes focused on the value of media types as a guiding principle for web services.

See Also

- Recipe 3.1, "Creating Interoperability with Registered Media Types"
- Recipe 3.2, "Ensuring Future Compatibility with Structured Media Types"
- Recipe 4.2, "Coding Clients to Be HTTP Aware"
- Recipe 4.3, "Coding Resilient Clients with Message-Centric Implementations"
- Recipe 4.6, "Managing Representation Formats at Runtime"

4.4 Coding Effective Clients to Understand Vocabulary Profiles

Just as it is important to code clients to be able to "speak" HTTP and bind to message formats, it is also important to code your client applications to be able to "converse" in the vocabulary of the service and/or domain. To do that, you need to code client applications to recognize and utilize domain vocabularies expressed as external profile documents.

Problem

How can you make sure that your client application will be able to "understand" the vocabulary (names of call properties and actions) of the services it will interact with, even when some of the services themselves might change over time?

Solution

To make sure clients and servers are "talking the same language" at runtime, you should make sure both client and server are coded to understand the same vocabulary terms (e.g., data names and action names). You can accomplish this by relying on one of the well-known vocabulary formats, including:

- RDF, Resource Description Framework Schema
- OWL, Web Ontology Language
- DCAP, Dublin Core Application Profiles
- ALPS, Application-Level Profile Semantics

Since I am a coauthor on the ALPS specification, you'll see lots of examples of using ALPS as the vocabulary standards document. You may also be using a format not listed here—and that's just fine. The important thing is that you and the services your client application interacts with agree on a vocabulary format as a way to describe the problem domain.

Example

Expressing a service's domain as a set of vocabulary terms allows both client and service to *share understanding* in a generic, stable manner. This was covered in Recipe 3.4. Here's a set of vocabulary terms for a simple domain:

```
# Simple ToDo

## Purpose
We need to track 'ToDo' records in order to improve both timeliness
and accuracy of customer follow-up activity.
```

```
## Data
In this first pass at the application, we need to keep track of the
following data properties:

 * **id** : a globally unique value for each ToDo record
 * **body** : the text content of the ToDo record

## Actions
This edition of the application needs to support the following operations:

 * **Home** : starting location of the service
 * **List** : return a list of all active ToDo records in the system
 * **Add** : add a new ToDo record to the system
 * **Remove** : remove a completed ToDo record from the system
```

Next, after translating that story document into ALPS, it looks like this:

```
{
  "$schema":"https://alps-io.github.io/schemas/alps.json",
  "alps" : {
    "version":"1.0",
    "title":"ToDo List",
    "doc" : {"value":"ALPS ToDo Profile (see [ToDo Story](to-do-story.md))"},

    "descriptor":[
      {"id":"id", "type":"semantic", "tag":"ontology"},
      {"id":"title", "type":"semantic", "tag":"ontology"},

      {"id":"home", "type":"semantic", "tag":"taxonomy",
        "descriptor": [{"href":"#goList"}]
      },

      {"id":"list", "type":"semantic", "tag":"taxonomy",
        "descriptor":[
          {"href":"#id"},
          {"href":"#title"},
          {"href":"#goHome"},
          {"href":"#goList"},
          {"href":"#doAdd"},
          {"href":"#doRemove"}
        ]
      },

      {"id":"goHome", "type":"safe", "rt":"#home", "tag":"choreography"},
      {"id":"goList", "type":"safe", "rt":"#list", "tag":"choreography"},
      {"id":"doAdd", "type":"unsafe", "rt":"#list", "tag":"choreography",
        "descriptor": [{"href":"#id"},{"href":"#title"}]},
      {"id":"doRemove", "type":"idempotent", "rt":"#list", "tag":"choreography",
        "descriptor": [{"href":"#id"}]}
    ]
  }
}
```

This ALPS document contains all the possible data elements and action elements, along with additional metadata on how to craft requests to the service and what to expect in response. This ALPS document can be the source material for creating client applications that will be able to successfully interact with any service that supports this profile. This is especially true for services that support hypermedia responses.

For example, assuming a ToDo-compliant service is running at localhost:8484, a ToDo-compliant client might do the following:

```
# data to work with
STACK PUSH {"id":"zaxscdvf","body":"testing"}

# vocabulary and format supported
CONFIG SET {"profile":"http://api.examples.org/profiles/todo-alps.json"}
CONFIG SET {"format":"application/vnd.mash+json"}

# write to service
REQUEST WITH-URL http://api.example.org/todo/list WITH-PROFILE WITH-FORMAT
REQUEST WITH-FORM doAdd WITH-STACK
REQUEST WITH-LINK goList
REQUEST WITH-FORM doRemove WITH-STACK
EXIT
```

Check out Appendix D for a short review of how to program in HyperLANG with the HyperCLI.

Note that the client only needs to know the starting URL, along with the names of the actions (`doAdd`, `goList`, and `doRemove`) and the names of the data properties (`id` and `body`). The client also needs to tell the service what vocabulary (`profile`) and media type (`format`) will be used during the interaction. These are all decisions made when *coding* the application. With that done, the remainder of the script invokes actions and sends data that the client knows ahead of time will be valid—based on the published vocabulary. Since, in this case, the service is known to support hypermedia responses, the client can safely assume that the vocabulary links and forms will be found in the service response and coded accordingly.

Discussion

Vocabulary documents are the glue that binds API producers and API consumers without resorting to tight coupling. As we have seen in the last example, reliance on clear vocabularies and well-known media type formats does a great deal to reduce the coupling between client and service without losing any detail.

Services often publish API definition documents (e.g., OpenAPI, AsyncAPI, RAML, etc.). This is handy for people who want to implement a single instance of the API *service*, but it has limited value for API consumers. When API client applications use the service implementation specification to create an API consumer, that application will be tightly bound to this particular service implementation. Any changes to the service implementation are likely to break that client application. A better approach is to bind the API consumer to a published *profile* document instead.

If the service you are working with does not publish a stable vocabulary document, it is a good idea to create your own. Use the API definition documentation (OpenAPI, etc.) as a guide to craft your own ALPS document (or some other format, if you wish), and then publish that document for your team (and others) to use when creating an API client for that service. It is also a good idea to include that ALPS document in your client application's source code repository for future reference.

As shown in the last example, you get the greatest benefit of vocabulary profiles when the service you are working with returns hypermedia responses. When that is not the case (and you don't have the ability to influence the service teams), you can still code your client as if hypermedia was in the response (see Recipe 4.10 for details).

See Also

- Recipe 3.4, "Describing Problem Spaces with Semantic Profiles"
- Recipe 3.3, "Sharing Domain Specifics via Published Vocabularies"
- Recipe 4.5, "Negotiating for Profile Support at Runtime"
- Recipe 4.7, "Using Schema Documents as a Source of Message Metadata"
- Recipe 5.7, "Publishing Complete Vocabularies for Machine Clients"
- Recipe 5.9, "Publishing Service Definition Documents"

4.5 Negotiating for Profile Support at Runtime

Client applications that depend on services that provide responses using predetermined semantic profiles need a way to make sure those services will be "speaking" in the required vocabulary.

Problem

You've coded your client to work with any service that complies with, for example, the ToDo semantic profile. How can you confirm—at runtime—that the service you are about to use supports the `to-do` vocabulary?

Solution

A client application can dependably check for semantic profile support in a service using the "profile negotiation" pattern. Clients can use the `accept-profile` request header to indicate the expected semantic profile, and services can use the `content-profile` response header to indicate the supporting semantic profile.

Like content negotiation (see Recipe 5.6), profile negotiation allows clients and servers to share metadata details about the resource representation and make decisions on whether the server's response is acceptable for the client. When services return resource profiles that do not match the client's request, the client can reject the response with an error, request more information from the server, or continue anyway.

Example

Clients can determine the supported semantic profiles for a service resource in a number of ways. Profile negotiation is the most direct way for clients to indicate and validate a service's profile support.

The following is a simple example of profile negotiation:

```
*** REQUEST
GET /todo/list HTTP/1.1
Host: api.example.org
Accept-Profile: <http://profiles.example.org/to-do>

*** RESPONSE
HTTP/2.0 200 OK
Content-Profile: http://profiles/example.org/to-do

...
```

In this example, the client uses the `accept-profile` header to indicate the desired vocabulary, and the service uses the `content-profile` header to tell the client what profile was used to compose the resource representation that was returned.

Servers may also return a `406` HTTP status code (`Not Acceptable`) when a client requests a semantic profile the service does not support. When doing so, the service should return metadata that indicates which profiles that service is prepared to support instead:

```
*** REQUEST
GET /todo/list HTTP/1.1
Host: api.example.org
Accept-Profile: <http://profiles.example.org/to-do/v3>

*** RESPONSE
```

```
HTTP/2.0 406 Not Acceptable
Content-Type: application/vnd.collection+json

{ "collection":
  {
    "links" : [
      {"rel":"profile", "href":"http://profiles.example.org/todo/v1"},
      {"rel":"profile", "href":"http://profiles.example.org/todo/v2"},
    ]
  },
    "error" : {
      "title" : "Unsupported Profile",
      "message" : "See links for supported profiles for this resource."
  }
}
```

It is also possible for services to provide a way for client applications to preemptively request details on the supported semantic profiles. To do this, services can offer one or more links with the relation value of "`profile`," with each link pointing to a supported semantic profile document. For example:

```
{ "collection":
  {
    "title" : "Supported Semantic Profiles",
    "links" : [
      {"rel":"profile", "href":"http://profiles.example.org/todo/v1"},
      {"rel":"profile", "href":"http://profiles.example.org/todo/v2"},
      {"rel":"profile", "href":"http://profiles.example.org/todo/v3"}
    ]
  }
}
```

These links can appear in any resource representation that supports that profile. Even when only one semantic profile is supported, it is a good idea to emit profile links in responses to help clients (and their developers) recognize the supported vocabularies. In this case, clients can look at the collection of profile links to see if their semantic profile identifier is among those listed in the response.

The list of supported profiles can be reported as HTTP headers too:

```
*** REQUEST
GET /todo/list HTTP/1.1
Host: api.example.org
Accept-Profile: <http://profiles.example.org/to-do/v3>

*** RESPONSE
HTTP/2.0 406 Not Acceptable
Content-Type: application/vnd.collection+json
Links <http://profiles.example.org/todo/v3>; rel="profile",
<http://profiles.example.org/todo/v2>; rel="profile",
<http://profiles.example.org/todo/v1>; rel="profile"
```

Discussion

It's a good idea for client applications that are "coded to the profile" to validate all incoming responses to make sure the client will be able to process the resource correctly. Since servers may actually report more than one profile link in responses, client applications should assume all profile metadata is reported as a collection and search for the desired profile identifier within that collection.

Negotiating for semantic profiles with `accept-profile` and `content-profile` is not very common. The specification that outlines that work is still a draft document and, as of this writing, the most recent WC3 work on profiles is still a work in progress. However, within a closed system (e.g., enterprise IT), implementing profile negotiation can be a good way to promote and encourage the use of semantic profiles in general.

I don't recommend getting too granular with identifying semantic profile versions (v1, v1.1, v1.1.1, etc.), as it only makes for more work at runtime for both client applications and remote services. When changes to profiles need to be made, it is a good idea to keep the variations backward compatible for as long as possible. If a profile update results in a breaking change, you should update the identifier to indicate a new version (v1 → v2).

See Also

- Recipe 3.3, "Sharing Domain Specifics via Published Vocabularies"
- Recipe 3.4, "Describing Problem Spaces with Semantic Profiles"
- Recipe 4.4, "Coding Effective Clients to Understand Vocabulary Profiles"
- Recipe 4.7, "Using Schema Documents as a Source of Message Metadata"
- Recipe 5.7, "Publishing Complete Vocabularies for Machine Clients"
- Recipe 5.8, "Supporting Shared Vocabularies in Standard Formats"

4.6 Managing Representation Formats at Runtime

Client applications that need to interact with more than one service may need to be coded to "converse" in more than one message format (HTML, HAL, SIREN, Collection+JSON, etc.). That means recognizing and dealing with multiple message formats at runtime.

Problem

How do client applications that need to be able to process more than one message type request and validate the message types they receive from services and process information from varying formats consistently?

Solution

Applications that operate should be able to manage the resource representation formats (HTML, HAL, SIREN, Collection+JSON, etc.) using the HTTP `Accept` and `Content-Type` headers. This focus on message-centric coding (see Recipe 4.3) will give stability to your client code without banding it closely to the specific domain of the service(s) you are using.

Media Type First Development

The first step in managing message formats on the web is to take advantage of HTTP's `Accept` and `Content-Type` headers to signal format preferences to the service and to validate the representation format returned by the service.

Whenever your client makes an API request, you should include an HTTP `Accept` header to indicate the structured format(s) your application "understands." Then, when the server replies, you should check the `Content-Type` header to confirm which of your supported formats was returned. If the format returned is not one your client can handle, you should stop processing and report the problem. In human-driven apps, this is usually done with a pop-up message or inline content. For M2M apps, this can be done as an error message back to the calling machine and/or a log message sent to some supervising authority:

```
*** REQUEST
GET /todo/list HTTP/1.1
Host: api.example.org
Accept: application/vnd.collection+json

*** RESPONSE
HTTP/1.1 200 OK
Content-Type: application/vnd.collection+json
...
```

The client applications also need to be coded in a way that creates a separation of concern (SoC) between the message formats exchanged with services and the internal representation of the information. When the server returns a response, the client application can route the message body to the proper handler for processing. This is where the message is translated into an internal model that the client application can use to inspect and manipulate the response.

That means implementing the Message Translator pattern internally so that clients can consistently translate between internal object models and external messages.

Example

Here is an example of code that validates the media type of the service response and handles it accordingly:

```
function handleResponse(ajax,url) {
  var ctype
  if(ajax.readyState===4) {
    try {
      ctype = ajax.getResponseHeader("content-type").toLowerCase();
      switch(ctype) {
        case "application/vnd.collection+json":
          cj.parse(JSON.parse(ajax.responseText));
          break;
        case "application/vnd.siren+json":
          siren.parse(JSON.parse(ajax.responseText));
          break;
        case "application/vnd.hal+json":
          hal.parse(JSON.parse(ajax.responseText));
          break;
        default:
          dump(ajax.responseText);
          break;
      }
    }
    catch(ex) {
      alert(ex);
    }
  }
}
```

Notice that the client can "converse" in three message formats: Collection+JSON, SIREN, and HAL. If the service returns anything else, the application sends it to the `dump()` routine for additional processing.

Discussion

Organizing your client code to be driven by message formats works best when services support highly structured formats like HTML, HAL, SIREN, Collection+JSON, etc. When services use unstructured response formats like XML and JSON, you'll need additional metadata to support message-driven client code. In that case, schema documents like XML Schema and JSON Schema can be handy. See Recipe 4.7 for more on using schema documents on the client side.

Recognizing the incoming message format is just the first step. You also need to be able to parse and often translate that message for internal use. For web browsers, you can build a single JavaScript library that converts incoming JSON formats (HAL, SIREN, Collection+JSON) into HTML for rendering. The following are the high-level methods for converting JSON-based API messages into HTML:

```
 // collection+JSON-->HTML
function parse(collection) {
  var elm;

  g.cj = collection;

  title();
  content();
  links();
  items();
  queries();
  template();
  error();

  elm = d.find("cj");
  if(elm) {
    elm.style.display="block";
  }
}

// HAL --> HTML
function parse(hal) {
    g.hal = hal;

    title();
    setContext();
    if(g.context!=="") {
      selectLinks("app", "toplinks");
      selectLinks("list", "h-links");
      content();
      embedded();
      properties();
    }
    else {
      alert("Unknown Context, can't continue");
    }

    elm = d.find("hal");
    if(elm) {
      elm.style.display="block";
    }
  }

// SIREN --> HTML
function parse(msg) {
  var elm;

  g.siren = msg;

  title();
  getContent();
  links();
```

```
    entities();
    properties();
    actions();

    elm = d.find("siren");
    if(elm) {
      elm.style.display="block";
    }
  }
```

Each message parser is designed specifically to convert the selected message format into HTML to render in the browser. This is an implementation of the Message Translator pattern. Writing these one-way translators is not very challenging for highly structured formats. Since the target (HTML) is not domain specific, a single translator can be effective for any application topic or domain. I always keep a stable browser-based media type parser on hand for quickly building rendering applications for services that use these formats.

The work gets more challenging if you need to translate the incoming message into a domain-specific internal object model or graph. In these cases, it's a good idea to use other metadata like semantic profiles (see Recipe 4.4) or schema documents (see Recipe 4.7) as your guide for translating the incoming message.

The work gets exponentially more difficult when all your client has to work with is an unstructured document like XML or JSON *and* there is no reliable schema or profile supplied. In these cases, you'll need to spend time creating custom-built translators based on the human-readable documentation. It is also likely that small changes in the documentation can cause big problems for your custom-built translator.

See Also

- Recipe 3.2, "Ensuring Future Compatibility with Structured Media Types"
- Recipe 3.4, "Describing Problem Spaces with Semantic Profiles"
- Recipe 4.3, "Coding Resilient Clients with Message-Centric Implementations"
- Recipe 4.4, "Coding Effective Clients to Understand Vocabulary Profiles"
- Recipe 4.7, "Using Schema Documents as a Source of Message Metadata"
- Recipe 5.6, "Supporting HTTP Content Negotiation"

4.7 Using Schema Documents as a Source of Message Metadata

When services do not support semantic profiles or structured media types, client applications may still be able to code to a semantic specification if that service returns references to schema documents that describe the contents of the message:

- Use schema where profiles and media types are not sufficient or are missing
- *Do not* use schema documents to validate *incoming* messages
- *Do* use schema to validate *outgoing* messages

Problem

How can we build resilient client applications based on message schema for unstructured media types (XML or JSON) instead of semantic profiles?

Solution

In cases where services consistently return schema documents with unstructured messages formats (XML or JSON) as runtime responses, you may be able to code a resilient client application without the support of semantic profiles. To do this, you need to organize client code that is driven solely by schema information.

The most common examples of these schema formats are XML Schema and JSON Schema.

Example

When services provide them, you can also use schema documents as additional metadata information for the incoming message. This works best when the response itself contains a pointer to the schema. At runtime, it is not a good idea to validate incoming messages against a schema document unless you plan on rejecting invalid inputs entirely. Usually, runtime responses will have minor differences (e.g., added fields not found in the schema, rearranged order of elements in the response, etc.), and these minor changes are not good reasons to reject the input. However, it is handy to use the provided schema URI/URL as an *identifier* to confirm that the service has returned a response with the expected semantic metadata.

Using schema documents to validate *incoming* messages runs counter to Postel's Law, which states: "be conservative in what you do, be liberal in what you accept from others." Strong typing incoming messages (instead of "be[ing] liberal in what you accept") is likely to result in rejected responses that might otherwise work well for both client and server. The specification document RFC 1122, authored by Robert Braden, goes into great detail on how to design and implement features that can safely allow for minor variations in message and protocol details. See Recipes 4.12 and 4.13 for more on this challenge.

Negotiating for schemas

It is possible for client applications to engage in schema negotiation with the service. This is useful when both client and server know ahead of time that there are multiple editions of the same general schema (user-v1, user-v2, etc.) and you want to confirm both are working with the same edition. An expired proposal from the W3C suggested the use of `accept-schema` and `schema` HTTP headers:

```
*** REQUEST
GET /todo/list HTTP/1.1
Host: api.example.org
Accept-Schema: <urn:example:schema:e-commerce-payment>

*** RESPONSE
HTTP/1.1 200 OK
Schema: <urn:example:schema:e-commerce-payment
...
```

I've not encountered this negotiation approach "in the wild." If it is offered by a service, it is worth looking into as a client developer—any semantic metadata is better than none.

Schema identifier in the Content-Type header

The schema identifier may also be passed from server to client in the `Content-Type` header:

```
*** REQUEST
GET /todo/list HTTP/1.1
Host: api.example.org
Accept: application/vnd.hal+json

*** RESPONSE
HTTP/1.1 200 OK
Content-Type: \
  application/vnd.hal+json;schema=urn:example:schema:e-commerce-payment
...
```

It is important to point out that adding additional facets (those after the ";") to the media type string can complicate message format validation. Also, some media type definitions expressly forbid including additional metadata in the media type identifier string (for example, JSON has this rule).

Schema identifier in the document

It is most common to see the schema identifier provided as part of the message body:

```
{
  "$schema":"https://alps-io.github.io/schemas/alps.json",
  "alps" : {
    "version":"1.0",
```

```
    "title":"ToDo List",
    "doc" : {"value":"A suggested ALPS profile for a ToDo service"},
    "descriptor":[
      {"id":"id", "type":"semantic", "def":"http://schema.org/identifier"},
      {"id":"title", "type":"semantic", "def":"http://schema.org/title"},
      {"id":"completed", "type":"semantic",
        "def":"http://mamund.site44.com/alps/def/completed.txt"}
    ]
  }
}
```

The schema identifier does not need to be a dereferenceable URL (*https://alps-io.github.io/schemas/alps.json*). It may just be a URI or URN identifier string (urn:example:schema:e-commerce-payment). The value of using a dereferenceable URL is that humans can follow the link to get additional information.

Coding schema-aware clients

Once you know how to identify schema information, you can use it to drive your client application code:

```
function handleSchema(ajax,schemaIdentifier) {
  var schemaType
  try {
    schemaType = schemaIdentifier.toLowerCase()
    switch(ctype) {
      case "https://api.example.com/schemas/task":
        task.parse(JSON.parse(ajax.responseText));
        break;
      case "https://api.example.com/schemas/task-v2":
        task.parse(JSON.parse(ajax.responseText));
        break;
      case "https://api.example.com/schemas/user":
      case "https://api.example.com/schemas/user-v2":
        user.parse(JSON.parse(ajax.responseText));
        break;
      case "https://api.example.com/schemas/note":
        note.parse(JSON.parse(ajax.responseText));
        break;
      default:
        dump(ajax.responseText);
        break;
    }
  }
  catch(ex) {
    alert(ex);
  }
}
```

Note that organizing code around schema definitions is often much more work than relying on semantic profile formats like ALPS. Schemas are usually domain specific and are more likely to change over time than semantic profiles. In this example, you

can see that there are two (apparently) incompatible schema references for the `task` object. Also, the example shows that while there are two different `user` schema documents, they are (at least as far as this client is concerned) compatible versions.

Discussion

The primary value of schema documents comes into play when you are creating and sending messages (see Recipe 4.12). However, client applications can also use the schema document identifiers themselves (URLs or URIs) as advisory values for processing incoming messages. Essentially, schema identifiers become confirmation values that the response received by the client will contain the information the client will need to perform its work.

I won't be talking about maintaining schema registries in this edition of the cookbook. I do, however, cover the power of registered vocabularies (much less constrained than schema documents) in Recipes 3.4 and 4.4.

Negotiating for schema identifiers has limited value as well. It can be a good way for clients to preemptively tell services what schema they can support. This allows services to *reject* requests using HTTP status code `406 Not Acceptable`. But this, too, runs afoul of Postel's Law.

Client applications that are coded to be "schema aware" are likely to be more of a challenge than client applications coded as "media type aware" (see Recipe 4.3) or "profile aware" (see Recipe 4.4). That is because schema documents typically are at a much more granular level (objects), and object schema usually changes more often than the overall vocabulary or message formats.

JSON Schema has a feature that allows for successful validation of a sample document even if that document contains additional properties not found in the schema (`additionalProperties:true|false`). By default, this is set to `true`, which allows for additional properties to appear without triggering a schema validation error. Also, altering the order of the properties within an object will not trigger a schema validation error.

Including "unknown" properties in an incoming message is not, by itself, a security problem. For example, this can be quite common in the HTML format. However, attempting to consume and process unknown properties *may* be a security hole. As you'll see in Recipe 4.13, it is best for client applications to ignore unknown properties and only process the values the application was designed to deal with.

In XML documents, schema validators are rather demanding. To be passed as valid, the elements within the XML messages need to be in the same order as indicated in the schema document, and in most cases, the appearance of a new element or property will trigger a validation error.

See Also

- Recipe 3.4, "Describing Problem Spaces with Semantic Profiles"
- Recipe 4.3, "Coding Resilient Clients with Message-Centric Implementations"
- Recipe 4.4, "Coding Effective Clients to Understand Vocabulary Profiles"
- Recipe 4.12, "Using Document Schemas to Validate Outgoing Messages"
- Recipe 4.13, "Using Document Queries to Validate Incoming Messages"

4.8 Every Important Element Within a Response Needs an Identifier

Just as every important resource in a RESTful API has an identifier (URL), every important action and/or data element *within* a response deserves its own identifier. A client needs to be able to locate the right resource, and find and use the right action or data collection in the resource response.

Problem

How do you make sure that client applications can find the actions and/or data they are looking for within a response? In addition, how can you make sure the ability to find these important elements doesn't degrade over time as the service evolves?

Solution

The key to ensuring stable, meaningful identifiers within API responses is to select media types that support a wide range of identifier types as structural elements. HTML does a good job at this. I created the Machine Accessible Semantic Hypermedia (MASH) format as an example of a media type that supports multiple structural identifiers.

The four kinds of identifiers are:

`ID`
: A document-wide unique single-value element (e.g., `id=1q2w3e4r`)

`NAME`
: An application-wide unique single-value element (e.g., `name=createUserForm`)

`REL`
: A system-wide unique multivalue element (e.g., `rel=create-form self`)

`TAG`
: A solution-specific nonunique multivalue element (e.g., `tag=users page-level`)

Another example of the last item (solution-specific nonunique multivalue element) is the HTML `CLASS` element.

Client applications should be coded to be able to find the desired FORM, LINK, or DATA block using at least one of these types of identifiers.

Example

For an example of the importance of using media types that support structured identifiers, consider the following challenge:

1. Access the starting URL of the `users` service .
2. Within the response to that request, find the form for searching for existing users.
3. Fill out the form to find the user with the nickname "mingles," and execute the request.
4. Store the resulting properties of this response to client memory.
5. Within the response to that request, find the form for updating this resource.
6. Using the data in the response, fill in the update form, change the email address to "*mingles@example.org*," and execute the request.

While we, as humans, could do this rather easily (e.g., in an HTML browser), getting a machine to do this takes some additional effort. Consider, for example, that you need to update the email addresses of 10 or more user accounts. Now add to this challenge that you need to deal with multiple versions of the user service, or one that uses multiple response formats, or one that has evolved over time. In these cases, the client might use a script like this:

```
# hyper-cli example

ECHO Updating email address for mingles

ACTIVATE http://api.example.org/users

ACTIVATE WITH-FORM nickSearch
  WITH-QUERY {"nick":"mingles"}

ACTIVATE WITH-FORM userUpdate
  WITH-BODY {"name":"Miguelito","nick":"mingles","email":"mingles@example.org"}
```

```
ACTIVATE WITH-FORM emailSearch
  WITH-QUERY {"email":"mingles@example.org"}

ECHO Update Confirmed

EXIT

# eof
```

Check out Appendix D for more on how to use the HyperCLI and HyperLANG as a hypermedia client application.

Note in this example that the client application is relying on identifiers for three actions (`nickSearch`, `userUpdate`, and `emailSearch`) and for a collection of properties (`name`, `nick`, and `email`). The actual representation format (HTML, MASH, SIREN, etc.) does not matter for the script—that is handled internally by the client. Also note that the only URL supplied is the initial one to contact the service. The rest of the URLs are supplied in the responses (as part of the forms).

Discussion

Most HTTP design practices make it clear that every *resource* deserves its own identifier (in the form of a URL). This is easy to see when you look at typical HTTP API requests:

```
http://api.example.org/customers/123
http://api.example.org/users?customer=123
```

This is also true when it comes to describing interactions within a single response. For example, consider the following response from `/users?customer=123`:

```
{
  "id": "aqswdefrgt",
  "href": "http://api.example.org/customers/123",
  "name": "customer-record",
  "rel": "item http://rels.example.org/customer",
  "tags": "item customer read-only",
  "data": [
    "identifier": "123",
    "companyName": "Ajax Brewing",
    "address": "123 Main St, Byteville, MD",
    "users": "http://api.example.org/users?customer=123"
  ]
}
```

Note that the record metadata is as follows:

`id`
: A document-wide unique ID that clients can use to locate this record in any list

`href`
: A system-wide URL that clients can use to retrieve this record from the service

`name`
: A semantic application-wide name that clients can use to return records of this "type"

`rel`
: A collection of system-wide names that clients can use to return records of this "type" or instance

`tags`
: A collection of application-wide names that clients can use to filter records in various ways

The added `href` example here shows that the system-wide identifier (*http://api.example.org/customers/123*) does not need to be the same as the internal `id` (`aqswdefrgt`). In fact, it is a good idea to *assume* they are decoupled in order to avoid problems when the URL changes due to service migration, installation of new proxies, etc.

See Also

- Recipe 3.5, "Expressing Actions at Runtime with Embedded Hypermedia"
- Recipe 4.3, "Coding Resilient Clients with Message-Centric Implementations"
- Recipe 5.4, "Expressing Internal Functions as External Actions"

- Recipe 5.14, "Increasing Throughput with Client-Supplied Identifiers"
- Recipe 7.13, "Using State-Watch to Enable Client-Driven Workflow"

4.9 Relying on Hypermedia Controls in the Response

One of the principles of REST-based services is the reliance on "hypermedia as engine of application state" or, as Roy Fielding has called it, "The Hypermedia Constraint." Services that emit hypermedia formats as their responses make it much easier for client applications to determine what actions are possible—and how to commit those actions—at *runtime*. And the hypermedia information that appears within a response can be context sensitive. Context examples include the current state of the service (does a record already exist?), the user context (does this user have permission to edit the record?), and the context of the request itself (has someone else already edited this data?).

Problem

When service providers are emitting hypermedia-based responses, how can the API consumer best take advantage of the included information to improve both the human-agent and machine-agent experience?

Solution

It is best to write API consumers to "understand" response formats (see Recipe 4.3 in this chapter). In the case of hypermedia types, each format has its own "hypermedia signature"—the list of hypermedia elements (called hypermedia factors or H-Factors) that this media type can express. Creating client applications that know the media type and understand the nine H-Factors will greatly improve the flexibility and resilience of the client application. Nine H-Factors can appear within a RESTful service response, which is described in Tables 4-1 and 4-2.

Table 4-1. Five link factors

	Title	Description	Example
LE	Link Embedded	Brings content into the current view	HTML `<img />` tag
LO	Link Outbound	Navigates the client to a new view	HTML `<a />` tag
LT	Link Template	Supports additional parameters before executing	URI templates `http://example.org/users?{id}`
LN	Link Nonidempotent	Describes a nonidempotent action	`<FORM method=POST …>`
LI	Link Idempotent	Describes an idempotent action	SIREN `"actions":[{"name":"update-item","method":"put" …}]`

Table 4-2. Four control factors

	Title	Description	Example
CR	Control for Read Requests	Describes read details	`Accept:application/vnd.hal+json`
CU	Control for Update Requests	Describes write details	`enctype="application/x-www-form-urlencoded"`
CM	Control for HTTP Methods	Describes the HTTP method to use	`<FORM method="GET" … >`
CL	Control for Link Relations	Describes the link relation for an action	`<link rel="create-form" … />`

Each hypermedia format has its own H-Factor signature.

In Figure 4-2, you can see three media types: SVG, Atom, and HTML. Note that the SVG format only supports the LE and LO factors. SVG, for example, supports embedded links like `<IMAGE href="…"…>`, and outbound links such as `<A href="…">Go Home</a>`.

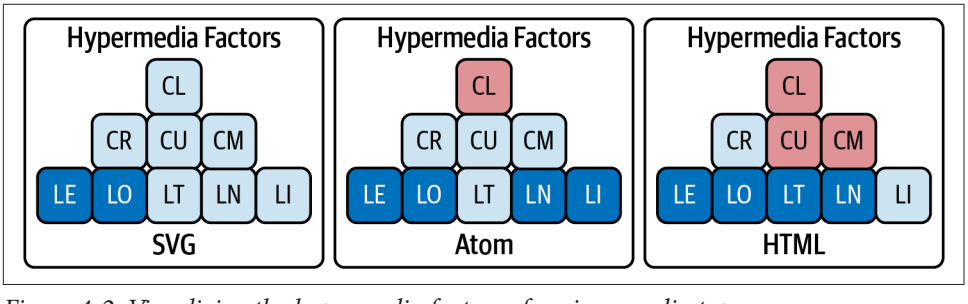

Figure 4-2. Visualizing the hypermedia factors of various media types

Example

Knowing which formats your client is likely to encounter allows the client to recognize and, where appropriate, offer options to human- and machine-driven agents in order for them to accomplish tasks.

For example, the HyperCLI client (see Appendix D) supports a number of hypermedia formats. When using a format like SIREN, you can script the HyperCLI client to add, search, modify, or remove content based on the hypermedia controls in the responses. The following is a short SIREN script written in HyperLANG:

```
#
# SIREN Edit Session
# read a record, save it, modify it, write it back to the server
#

# ** make initial request
REQUEST WITH-URL http://rwcbook10.herokuapp.com
```

```
# ** retreive the first record in the list
REQUEST WITH-PATH $.entities[0].href

# ** push the item properties onto the stack
STACK PUSH WITH-PATH $.properties

# ** modify the tags property value on the stack
STACK SET {"tags":"fishing,\.\skiing,\.\hiking"}

# ** use the supplied edit form and updated stack to send update
REQUEST WITH-FORM taskFormEdit WITH-STACK

# ** confirm the change
SIREN PATH $.entities[0]

# ** exit session
EXIT
```

See Appendix D for a short introduction to programming in HyperLANG with the Hyper CLI.

Note that in this example, only the starting URL was supplied. The remaining actions were committed by relying on the hypermedia metadata supplied in each service response.

Discussion

This recipe covers the principle of *using* that hypermedia data to solve problems directly. It relies on details supplied by services and, for that reason, this recipe is only helpful when services support hypermedia formats in the response.

The advantage of relying on supplied hypermedia in your scripts is that you reduce the amount of hardcoding you need to do within your client application. You can make it easier for services to modify hypermedia details (e.g., URLs, HTTP methods, supported formats, etc.) without breaking the client application. In the preceding example, the script does not hardcode the HTTP URL, method, or encoding type for the `taskFormEdit` action. It simply uses the details supplied by the service.

This recipe works best when the client application understands not only the response format (e.g., SIREN) but also the semantic profile (the topic domain) used by the service. For instance, in the preceding example, the client knows ahead of time that there is a possible `taskFormEdit` hypermedia control that it can use to modify an existing record. See Recipe 4.4 for more on semantic profiles.

The H-Factor signature of hypermedia formats varies widely and will affect how hardcoding is (or is not) required for an API client. For example, the SIREN format supports the full range of H-Factors, and HAL supports just a few of them. In contrast, a nonhypermedia format like `application/json` supports no built-in H-Factors, making it difficult to write a resilient API client for services that support only plain JSON responses.

In general, the more H-Factors supported by a media type, the more likely you can write an API client that will survive changes in hypermedia details in the future.

See Also

- Recipe 3.2, "Ensuring Future Compatibility with Structured Media Types"
- Recipe 3.5, "Expressing Actions at Runtime with Embedded Hypermedia"
- Recipe 4.3, "Coding Resilient Clients with Message-Centric Implementations"
- Recipe 4.7, "Using Schema Documents as a Source of Message Metadata"
- Recipe 5.4, "Expressing Internal Functions as External Actions"
- Recipe 6.7, "Using Media Types for Data Queries"

4.10 Supporting Links and Forms for Nonhypermedia Services

There are times when you need to interact with services that do not offer clear links and forms—hypermedia-style responses. There is a handy approach you can take on the client side to make up for this lack of support.

Problem

How can you get the advantages of API decoupling and the clearly described interactions available with hypermedia-style responses when the service you are working with does not offer hypermedia response formats as an option?

Solution

When you're working with services that don't offer hypermedia-style responses and you still want to be able to rely on clearly described links and forms on the client side, you can code your client application to supply its own links and forms at runtime, based on the original API documentation.

Example

A hypermedia-based API client consumes the response, identifies the action items (links and forms) in the message, and renders them (in the case of the human UI solution) along with any data that is received. A high-level look at that kind of client code would be:

```
// low-level HTTP stuff
function req(url, method, body) {
  var ajax = new XMLHttpRequest();
  ajax.onreadystatechange = function(){rsp(ajax)};
  ajax.open(method, url);
  ajax.setRequestHeader("accept",g.atype);
  if(body && body!==null) {
    ajax.setRequestHeader("content-type", g.ctype);
  }
  ajax.send(body);
}

function rsp(ajax) {
  if(ajax.readyState===4) {
    g.msg = JSON.parse(ajax.responseText);
    showTitle();
    showToplinks();
    showItems();
    showActions();
  }
}
```

When a response is received, the client looks through the message body and handles all the key tasks. For example, here's what the `showTopLinks()` method might look like:

```
// emit links for all views
function showToplinks() {
  var act, actions;
  var elm, coll;
  var menu, a;

  elm = d.find("toplinks");
  d.clear(elm);
  menu = d.node("div");

  actions = g.actions[g.object]; // get the link metadata
  for(var act in actions) {
    link = actions[act]
    if(link.target==="app") {
      a = d.anchor({
        href:link.href,
        rel:(link.rel||"collection"),
        className:"action item",
        text:link.prompt
```

```
        });
        a.onclick = link.func;
        d.push(a, menu);
      }
    }
    d.push(menu,elm);
  }
```

This example renders the top links on a user interface. The source of that rendering is the link metadata. This client gets that in a single line of code. In hypermedia responses, that link metadata is in the body of the message. But even if you only get simple JSON data responses, you can use this same code. You just need to modify where the link metadata comes from.

In the following example, the link metadata comes from an internal structure that holds all the links and forms for a particular context ("users," "tasks," etc.). All the metadata was gleaned from the APIs human-centric documentation. All the rules for possible actions and arguments were translated from the API docs into this client-code structure:

```
// user object actions
g.actions.user = {
  home:       {target:"app", func:httpGet, href:"/home/", prompt:"Home"},
  tasks:      {target:"app", func:httpGet, href:"/task/", prompt:"Tasks"},
  users:      {target:"app", func:httpGet, href:"/user/", prompt:"Users"},
  byNick:     {target:"list", func:jsonForm, href:"/user",
                prompt:"By Nickname", method:"GET",
                args:{
                  nick: {value:"", prompt:"Nickname", required:true}
                }
              },
  byName:     {target:"list", func:jsonForm, href:"/user",
                prompt:"By Name", method:"GET",
                args:{
                  name: {value:"", prompt:"Name", required:true}
                }
              },
  add:        {target:"list", func:jsonForm, href:"/user/",
                prompt:"Add User", method:"POST",
                args:{
                  nick: {value:"", prompt:"Nickname", required:true,
                    pattern:"[a-zA-Z0-9]+"},
                  password: {value:"", prompt:"Password", required:true,
                    pattern:"[a-zA-Z0-9!@#$%^&*-]+"},
                  name: {value:"", prompt:"Full Name", required:true},
                }
              },
  item:       {target:"item", func:httpGet, href:"/user/{id}", prompt:"Item"},
  edit:       {target:"single", func:jsonForm, href:"/user/{id}",
                prompt:"Edit", method:"PUT",
                args:{
```

```
                nick: {value:"{nick}", prompt:"Nickname", readOnly:true},
                name: {value:"{name}", prompt:"Full Name",required:true}
              }
            },
  changepw:   {target:"single", func:jsonForm, href:"/task/pass/{id}",
              prompt:"Change Password", method:"POST",
              args:{
                nick: {value:"{nick}", prompt:"NickName", readOnly:true},
                oldPass: {value:"", prompt:"Old Password", required:true,
                  pattern:"[a-zA-Z0-9!@#$%^&*-]+"},
                newPass: {value:"", prompt:"New Password", required:true,
                  pattern:"[a-zA-Z0-9!@#$%^&*-]+"},
                checkPass: {value:"", prompt:"Confirm New", required:true,
                pattern:"[a-zA-Z0-9!@#$%^&*-]+"},
              }
            },
  assigned:   {target:"single", func:httpGet, href:"/task/?assignedUser={id}",
              prompt:"Assigned Tasks"}
};
```

Now you can use links and forms for services that don't provide them at runtime.

Discussion

This recipe relies on someone doing the work of translating the rules buried in the API documentation into a machine-readable form. That's what hypermedia-compliant services do. If the service doesn't automatically do this work, client developers can save a lot of time by doing it once and sharing the results. Consider placing all link and form metadata for a service in a separate module or an external configuration file. Even better, if you can, convince your server teams to do this for you.

Some formats are designed specifically to make capturing and sharing link and form metadata. One is the Web Service Transition Language (WSTL). You may also be able to use JSON HyperSchema. Check out "Hypermedia Supporting Types" on page 423 for details.

There are some additional challenges not covered in the example, the biggest being user-context management. It could be that admin users should see or do things that guest users do not. In that case, a solid approach is to create a separate set of action metadata for each security role your application supports.

Another challenge you'll need to deal with is when the service changes but your local metadata information does not keep up. This is another case of versioning problems that come up when clients operate on static metadata at runtime. The good news is that, when action rules change on the server, at least the client application only needs to update configuration metadata and not the entire application codebase.

See Also

- Recipe 3.2, "Ensuring Future Compatibility with Structured Media Types"
- Recipe 3.5, "Expressing Actions at Runtime with Embedded Hypermedia"
- Recipe 4.3, "Coding Resilient Clients with Message-Centric Implementations"
- Recipe 4.6, "Managing Representation Formats at Runtime"
- Recipe 5.3, "Converting Internal Models to External Messages"
- Recipe 5.4, "Expressing Internal Functions as External Actions"

4.11 Validating Data Properties at Runtime

When supporting the collection of inputs for client applications, it can be a challenge to properly describe valid input values at runtime. However, relying on predefined input rules documented only in human-readable prose can limit the adaptability of the client application and reduce its usefulness over time.

Problem

How can API clients consistently enforce the right data properties on input values at runtime? Also, how can we make sure that changes in the input rules over time continue to be honored by API clients and that these rule changes do not break already-deployed API clients?

Solution

An effective approach for API clients to support data property validation at runtime is to recognize and honor rich input descriptions (RIDs) within HTTP responses. A great example of RIDs can be found in the HTML format. HTML relies upon the `INPUT` element and a collection of attributes to describe expected inputs (`INPUT.type`) and define valid values (`INPUT.pattern`, `INPUT.required`, `INPUT.size`, etc.).

Client applications should support any or all data property quality checks provided by the incoming message format (HAL, SIREN, Collection+JSON, etc.).

Example

The HAL-FORMS specification is a good example of an API format that supports RIDs. HAL-FORMS is an extension of the HAL media type designed specifically to add RID support to the HAL format. The HAL-FORMS specification splits RIDs into three key groups: Core, Additional, and a special category: Options.

The Core section lists the following elements that clients *should* support: `readOnly`, `regex`, `required`, and `templated`. The Additional section lists elements that clients

may support: `cols`, `max`, `maxLength`, `min`, `minLength`, `placeholder`, `rows`, `step`, and `type`. The Options support is a special class of input validation that implements describing enumerators (e.g., `small`, `medium`, `large`, etc.).

Here is an example of a HAL-FORMS Options description:

```
{
  "_templates" : {
    "default" : {
      ...
      "properties" : [
        {
          "name" : "shipping",
          "type" : "radio",
          "prompt" : "Select Shipping Method",
          "options" : {
            "selectedValues" : ["FedEx"],
            "inline" : [
              {"prompt" : "Federal Express", "value" : "FedEx"},
              {"prompt" : "United Parcel Service", "value" : "UPS"},
              {"prompt" : "DHL Express", "value" : "DHL"}
            ]
          }
        }
      ]
    }
  }
}
```

Discussion

The SIREN format supports a wide range of RIDs. It does this by simply referring to the HTML specification for input types.

The RID can often be used to determine the *rendering* of the input control (e.g., a drop-down list, date-picker, etc.). This is common when the response message is consumed by an API client that is designed for human use, but can be ignored when the client application is designed for M2M interactions.

At a minimum, supporting a regular expression property to describe valid inputs can work in many cases and limits the implementation burden on the client application. This is especially true for M2M interactions where there is no need for user interface rendering hints.

If the service response does not include inline/runtime RIDs, client application developers should review the human-readable documentation and implement the API consumer to support the same kinds of RIDs described here. Usually that means creating either an inline code implementation of RIDs or a configuration-based approach. No matter the implementation details, it is a good idea to turn the human-

readable input validation rules into machine-readable algorithms. See Recipe 4.10 for details on this strategy.

See Also

- Recipe 3.4, "Describing Problem Spaces with Semantic Profiles"
- Recipe 4.4, "Coding Effective Clients to Understand Vocabulary Profiles"
- Recipe 4.8, "Every Important Element Within a Response Needs an Identifier"
- Recipe 4.10, "Supporting Links and Forms for Nonhypermedia Services"
- Recipe 4.12, "Using Document Schemas to Validate Outgoing Messages"
- Recipe 4.13, "Using Document Queries to Validate Incoming Messages"
- Recipe 4.14, "Validating Incoming Data"
- Recipe 6.8, "Ignoring Unknown Data Fields"

4.12 Using Document Schemas to Validate Outgoing Messages

It is important for client applications to do their very best to *send* valid messages (especially message *bodies*) to servers. One way to ensure you are sending valid message bodies is to validate those bodies with a schema document. But is it always a good idea to use schema validations when sending request bodies?

Problem

How can a client application make sure it is sending a well-formed and valid request body when invoking service requests?

Solution

Sending well-formed and valid request bodies with HTTP methods like `POST`, `PUT`, and `PATCH` is important. To reduce the chance your request is rejected, you should validate the structure and content of your request bodies before sending them to the service for processing. Essentially, it is the responsibility of the *client* application to properly craft a valid request body, and it is the role of the *service* to do its best to process all valid requests. This is the embodiment of Postel's Law (aka the robustness principle).

A very effective way to meet this requirement is to use schema documents to confirm the validity of your message body. Both JSON and XML have strong schema specifications, and there is a wide array of tooling to support them.

There are two levels of confirmation that clients need to handle: is the document well-formed, and is the document valid? Well-formed documents have the proper *structure*. This means they can be successfully parsed. Valid documents are not only well-formed, but both the *structural elements* (e.g., `street_name`, `purchase_price`) and the *values* of the those elements are of the right data type and within acceptable value ranges (e.g., `purchase_price` is a numerical value greater than zero and less than one thousand).

For any cases where client applications are expected to send request message bodies, those clients should perform some type of validation before committing the request to the internet. If the body format is in JSON or XML format, using schema documents and validator libraries is the way to go. For other formats (`application/x-www-form-urencoded`, etc.), client applications should rely on their own local validation routines to confirm the message body is well-formed and valid.

Example

Here's an example of a simple message body described in FORMS+JSON format:

```
{
  id: "filter",
  name: "filter",
  href: "http://company-atk.herokuapp.com/filter/",
  rel: "collection company filter",
  title: "Search",
  method: "GET",
  properties: [
    {name: "status",value: ""},
    {name: "companyName",value: ""},
    {name: "stateProvince",value: ""},
    {name: "country",value: ""}
  ]
}
```

And here's a basic JSON Schema document that can be used to validate the message body:

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "properties": {
    "status": {
      "type": "string",
      "enum": ["pending","active","closed"]
    },
    "companyName": {
      "type": "string"
    },
    "stateProvince": {
      "type": "string"
```

```
    },
    "country": {
      "type": "string"
    }
  }
}
```

The topic of JSON Schema is too large for this book to cover. To dig more into this topic, I recommend you check out *https://json-schema.org*.

Here is a possible body a client might send:

```
{
  "status":"pending",
  "country":"CA"
}
```

Finally, here's an example using a popular JSON validator (ajv) that I frequently use:

```
/*
 * load the schema file from external source
 * pass in the JSON message to send
 * process and return results/errors
 */

function jsonMessageCheck(schema, message) {
  var schemaCheck = ajv.compile(schema);
  var status = schemaCheck(message);
  return {status:status,errors:schemaCheck.errors};
}
```

Validating XML messages with XML Schema

There are similar options for handling XML validations, too, via XML Schema documents and tooling. There are a handful of approaches, depending on your platform.

Using the same example, here's the XML Schema for filtering company records:

```
<xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="message">
    <xs:complexType>
      <xs:all>
        <xs:element name="status" type="xs:string"
          minOccurs="0" maxOccurs="1"/>
        <xs:element name="companyName" type="xs:string"
          minOccurs="0" maxOccurs="1"/>
        <xs:element name="stateProvince" type="xs:string"
          minOccurs="0" maxOccurs="1"/>
```

```
        <xs:element name="country" type="xs:string"
          minOccurs="0" maxOccurs="1"/>
      </xs:all>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

For more on the world of XML and XSD, check out XML Schema and XML Schema 1.1.

And here's a sample XML message:

```
<message>
  <status>pending</status>
  <country>CA</country>
</message>
```

And then here's a simple example of runtime validation in client code:

```
var xsd = require('libxmljs2-xsd');

function xmlMessageCheck(schema, message) {
  var schemaCheck;
  var errors;
  var status = false;

  try {
    schemaCheck = xsd.parseFile(schema);
    status = true;
  } catch {
    status = false;
    errors = schemaCheck.validate(message);
  }
  return {status:status,errors:errors}
}
```

This sample relies on a node module based on the Linux `xmllib` library.

Validating FORMS with JSON Schema

Validating `application/x-www-form-urlencoded` message bodies takes a bit more work, but as long as the message bodies are simple, you can convert the `application/x-www-form-urlencoded` messages into JSON and then apply a JSON Schema document against the results:

```
const qs = require('querystring');

/*
 * read JSON schema doc from external source
```

```
 * send in form-urlencoded string
 * convert data into JSON and forward to json checker
 * return results from json checker
 */
function formJSONValidator(schema, formData) {
  var jsonData = qs.parse(formData);
  var results = jsonMessageCheck(schema, jsonData);
  return results // {status:status,schema.errors}
}
```

Notice that this routine depends on the `jsonMessageCheck` method seen earlier in this section.

Discussion

Services should make all required schemas available online and link to them in every interaction that relies on validated messages. This means that client applications should look for schema references in services responses. Common locations are shown here.

In the HTTP link header:

```
**** REQUEST
GET /api.example.org/users/search HTTP/1.1
Accept: application/forms+json;

*** RESPONSE
200 OK HTTP/1.1
Content-Type: application/forms+json;
Link: <schemas.example.org/service1/user-schema.json>; profile=schema

{...}
```

As part of the response body:

```
{
  "id" : "za1xs2cd3",
  "type": "search",
  "schema" : "api.example.org/schemas/user-search.json",
  "links" : [
    {
      "id" : "q1w2e3r4"
      "name" : "user",
      "href" : "http://api.example.org/q1w2e3r4",
      "title" : "User Search",
      "method" : "GET",
      "properties": [
        {"name":"familyName", "value":""},
        {"name":"givenName", "value":""},
        {"name":"sms", "value":""},
      ]
    }
```

```
    ],
    ...
}
```

Validating message bodies using JSON/XML Schema can be tricky. For example, JSON Schema is much more forgiving than XML Schema. While there are some tools that support converting XML Schema to JSON Schema at runtime, it rarely works out well. It is best to continue to write schemas in their native format.

The trick of converting `application/x-www-form-urlencoded` to JSON works well as long as the resulting JSON is treated as simple name/value pairs. Most FORMs-formatted messages follow this pattern, but there are cases where services get creative with FORM field names that imply a nested hierarchy. For example:

```
user.familyName=mamund&user.givenName=ramund&user.sms=+12345678901
```

implies a server-side message that looks like this:

```
{
  "user" : {
    "familyName" :"mamund",
    "givenName" :"ramund",
    "sms" : "+12345678901"
  }
}
```

As the service get more clever with naming patterns, the suggested trick of converting FORM strings to JSON documents gets less reliable. To avoid the problem, it may be better to write a code-based validator instead.

In the best case, services will supply client application developers with properly written (and up-to-date) schema documents to use at runtime. If the target service does not do this, application developers should review the human-readable documentation for the service API and craft their own schema documents for use at runtime. These documents should be posted somewhere online (possibly behind a secured URL) that is reachable by client applications. Care must also be taken to keep these schema documents up-to-date as the service changes over time.

When in doubt, write your own custom message checking code directly to avoid any problems with poorly written or inadequately maintained schema documents.

Well-crafted services will make sure that any schema changes to production services will remain backward compatible with previous editions of the service. However, client applications should be prepared for services that do not follow this principle. If the target service is particularly bad at maintaining backward compatibility, client

applications may need to abandon runtime validation based on schema documents and instead write code-based validators to ensure compatibility going forward.

See Also

- Recipe 3.4, "Describing Problem Spaces with Semantic Profiles"
- Recipe 4.7, "Using Schema Documents as a Source of Message Metadata"
- Recipe 4.13, "Using Document Queries to Validate Incoming Messages"
- Recipe 6.7, "Using Media Types for Data Queries"
- Recipe 7.13, "Using State-Watch to Enable Client-Driven Workflow"

4.13 Using Document Queries to Validate Incoming Messages

When API client applications receive a response from a service, it is important that they validate the incoming data to make sure the response contains the expected data in the requested format. This is especially true for M2M interactions where the human agent rarely gets a chance to view and validate incoming messages.

Problem

How can an API client application consistently and safely confirm that the incoming service response contains the expected data in the requested format?

Solution

API client applications should inspect incoming service responses on three levels:

Protocol
: Does the response include the expected HTTP protocol-level details (HTTP status code, content-type, and any other important headers)?

Structure
: Is the response body (if it exists) made up of the expected structural elements? For example, is there an expected `title` property? Does the body contain one or more `link` elements with the expected `rel`, `name`, or other properties, and so forth?

Value
: Assuming the expected structural elements exist (e.g., links, forms, data properties), do those elements contain the expected *values* (e.g., `rel="self"`)? In the case of data properties, do the elements contain the expected data types (string, integer, etc.), and are the values of the data properties within acceptable ranges?

For example, do `Date` type properties contain valid calendar dates? Are numerical values within acceptable upper and lower bounds?

For details on how to use schema documents to validate *outgoing* messages, see Recipe 4.12.

Example

Here is a simplified version of validating the incoming message for protocol, structure, and values:

```
var response = httpRequest(url, options);
var checks = {};
var message = {};

// protocol-level checks
checks.statusOK = (response.statusCode===200?true:false)
checks.contentTypeCollectionJSON = (
  response.headers["content-type"].toLowercase().indexOf("vnd.collection+json")
  ?true:false
  );
checks.semanticTypeTaxes = (
  response.headers["profile-type"].toLowercase().indexOf("taxes.v1.alps")
  ?true:false
  );

// structural checks
checks.body = JSON.parse(response.body);
checks.submitLink = (body.filter("submit").length>0?true:false);
checks.rejectLink = (body.filter("reject").length>0?true:false);
checks.countryCode = (body.filter("countryCode").length>0?true:false);
checks.stateProvince = (body.filter("stateProvince").length>0?true:false);
checks.salesTotal = (body.filter("salesTotal").length>0?true:false);

// value checks
checks.submit = checkProperties(submitLink, ["href","method","encoding"]);
checks.reject = checkProperties(rejectLink, ["href","method","encoding"]);
checks.country = checkProperties(countryCode, ["value"],filters.taxRules);
checks.stateProv = checkProperties(stateProvince, ["value"],filters.taxRules);
checks.salesTotal = checkProperties(salesTotal, ["value"],filters.taxRules);

// do computation
if(validMessage(checks) {
  var taxes = computeTaxes(checks);
  checks.taxes = checkProperties(taxes,filters.taxRules);
}

// send results
```

```
if(validMessage(checks) {
  message = formBody(checks.submit, checks.taxes);
}
else {
  message = formBody(checks.reject,checks.taxes);
}
response = httpRequest(taxMessage);
```

You can also see that, after all the checks are made (and assuming they all pass), the code performs the expected task (computing taxes) and submits the results. It is easy to see from this example that most of the coding effort is in safely validating the incoming message before doing a small bit of work. This is a *feature* of well-built API client applications, especially in M2M cases.

Discussion

Although the example is verbose, it should be noted that much of this code can be generated instead of handcoded. This can greatly reduce the effort needed to compose a well-formed API client, and improve the safety and consistency of the application at runtime.

Do not rely on schemas

You may be surprised to see that I am not suggesting the use of schema documents (JSON Schema, XML Schema, etc.) for validating incoming messages. In my experience, applying schemas (particularly strict schemas) to incoming messages results in too many false negatives—rejections of messages that fail the schema but could still be successfully and safely processed by the client. This is especially true in the case of XML Schema, which is strict by default. Even valid items in a different order in the document can result in a failed schema review. Schemas also don't cover the protocol-level checks that are needed.

Consider using JSON/XML path queries

The example relies on some features of JavaScript to perform filtering checks. These lines of code can be replaced by a more generic solution using JSON Path queries. I use a JavaScript/NodeJS library called JSON Path-Plus for this task. However, caution is advised here since, as of this writing, the JSONPath specification at IETF is still incomplete and will likely change significantly.

The XML Path specification, however, is quite mature and a very reliable source for checking XML-formatted messages. There is even ongoing work at the W3C to expand the scope of XML Path version 3.1 to include support for JSON queries.

See more on filtering incoming messages in Recipe 4.14.

See Also

- Recipe 3.4, "Describing Problem Spaces with Semantic Profiles"
- Recipe 4.7, "Using Schema Documents as a Source of Message Metadata"
- Recipe 4.12, "Using Document Schemas to Validate Outgoing Messages"
- Recipe 6.7, "Using Media Types for Data Queries"
- Recipe 7.13, "Using State-Watch to Enable Client-Driven Workflow"

4.14 Validating Incoming Data

When creating API clients it is important to know how to deal with incoming messages. Essentially, every service response should be assumed to be dangerous until proven otherwise. The message may contain malicious data, smuggled scripts for execution, and/or just plain bad data that might cause the client application to misbehave.

Having a strategy for safely dealing with incoming payloads is essential to building a successful API ecosystem. This is especially true in ecosystems where M2M communication exists, since humans will have reduced opportunities to inspect and correct or reject questionable content.

Problem

How can client applications make sure the incoming message does not contain dangerous or malicious content?

Solution

API client applications should carefully inspect incoming messages (service responses) and remove or ignore any dangerous content. This can be done by relying on a few basic principles when coding the client application:

- Always filter incoming data.
- Rely on the "allow list" model instead of "deny list" for approving content.
- Maintain a "min/max" for all data values and reject any data outside that range.

These general rules are supported by another line of defense for client applications: *syntactic* validation (based on the expected data type) and *semantic* validation (based on the expected value of that data type). Examples of syntactic validation are complex types such as postal codes, telephone numbers, personal identity numbers (US Social Security numbers, etc.) and other similar predefined types. Your client application should be able to identify when these types of data are expected to appear and know

how to validate the *structure* of these data types. For example, US postal codes are made up of two sets of digits: first a required five digits and then—optionally—a dash and four more digits. Well-designed message formats provide this kind of data type information in the response at runtime.

The basic premise that API clients should adopt is "defend yourself at all costs—including to the point of refusing to process the request altogether." This flies a bit in the face of Postel's Law (aka the robustness principle), but for a good reason. API clients should only pay attention to content values they already understand, should make sure the values are reasonable, and only perform predetermined operations on those fields.

Client applications should also do their best to validate the *semantic* value of the message content. For example, if a date range is provided—for example: `startDate` and `stopDate`—it is important to validate that the value for `startDate` is earlier than the value supplied for `stopDate`.

Example

Collection+JSON is an example of this kind of support for syntactic type information:

```
{"collection" :
  {
    ...
    "items" : [
      {
        "rel" : "item person"
        "href" : "http://example.org/friends/jdoe",
        "data" : [
          {"name" : "full-name", "value" : "J. Doe",
            "prompt" : "Full Name", "type" : "string"},
          {"name" : "email", "value" : "jdoe@example.org",
            "prompt" : "Email", "type" : "email"},
          {"name" : "phone", "value" : "123-456-7890",
            "prompt" : "Telephone", "type" : "telephone"}
          {"name" : "birthdate", "value" : "1990-09-30",
            "prompt" : "Birthdate", "type" : "date"}
        ]
      }
    ]
    ...
  }
}
```

Discussion

Not many representation formats automatically include data type and range metadata for content values. Where possible, try to convince the services your client application uses to add this information as runtime metadata. Formats that do a pretty good job at this are JSON-LD, UBER, Collection+JSON, and any of the RDF formats.

For cases where the type/range metadata is not available at runtime, client developers should scan the existing human-readable documentation and create their own content metadata to apply to incoming messages. While this external source might fall out of sync over time (when the human-readable documents are updated), having a machine-readable set of content filtering rules is still quite valuable.

Here is an example of possible type and range metadata for a client that computes sales and VAT taxes:

```
var filters = {}
filters.taxRules = {
  country:{"type":"enum","value":["CA","GL","US"]},
  stateProvince:{"type":"enum","value":[...]},
  salesTotal:{"type":"range","value":{"min":"0","max":"1000000"}
}
```

When receiving a response message, it is a good practice to create an internal, filtered representation of that message and only allow your own code to operate on the filtered representation—not the original. This reduces the chances that your application will mistakenly ingest improperly formed data and limits the possible risk to your application.

```
// make request, pull body, and scrub
var reponse = httpRequest(url, options);
var message = filterResponse(response.body, filters.taxRules);
```

In this example, the `message` variable contains content deemed safe for the application to use. Note that the `filterResponse()` routine can perform a number of operations. First, it should only look for content that is interesting for this client application. For example, if this client computes sales or value-added taxes, it might only need to pay attention to the fields `country`, `stateProvince`, and `salesTotal`. This filter may only return those fields while making sure the values in them are "safe." For example, the `country` field contains one of a list of valid ISO country codes, etc. The results can be used to format another message to use in future requests.

Note that in this example, additional *semantic* validation is needed to make sure that both the `country` and `stateProvince` values are compatible. For example, `country="CA"` and `stateProvince="KY"` would be semantically invalid.

Finally, there may be cases where an API client has the job of accepting a working document, performing some processing on that document, updating it, and sending

the document along to another application or service. In these instances, it is still a good idea to filter the content of the document down to just the fields your API client needs to deal with, validate those values, perform the operation, and then update the original document with the results before passing it along. Your API client is *not* in charge of scrubbing the whole document that is to be passed along, however. Your only responsibility is to pull the fields, clean them, do the work, and then update the original document. You should assume that any other applications that work on this document are doing the same level of self-protection.

Here is a simplified example:

```
function processTaxes(args) {
  var results = {};
  var response = httpRequest(args.readUrl, args.readOptions);
  var message = filterResponse(response.body,args.rules);

  if(message.status!=="error") {
    results = computeTaxes(message);
  } else {
    results = invalidMessage(message)
  }

  options.requestBody = updateBody(response.body, message);
  response = httpRequest(args.writeUrl, args.writeOptions);
}
```

For more on scrubbing incoming data, check out the OWASP Input Validation Cheat Sheet.

See Also

- Recipe 3.4, "Describing Problem Spaces with Semantic Profiles"
- Recipe 4.7, "Using Schema Documents as a Source of Message Metadata"
- Recipe 4.12, "Using Document Schemas to Validate Outgoing Messages"
- Recipe 4.13, "Using Document Queries to Validate Incoming Messages"
- Recipe 6.7, "Using Media Types for Data Queries"

4.15 Maintaining Your Own State

In a RESTful system, it can be a challenge to locate and access the transient state of requests and responses. There are only three possible places for this kind of information: server, client, and messages passed between server and client. Which option is best under what circumstances?

Problem

How can a client successfully track the transient state of server and client interactions? Where does the application state reside when a single client application is talking to multiple server-side components?

Solution

The safest place to keep the transient state—the state of the application as defined by runtime requests and responses—is in the client. The client is responsible for selecting which servers are engaged in interactions, and the client is initiating HTTP requests. It is the client, and only the client, that has the most accurate picture of the state of the application. This is especially true when the client has enlisted multiple services (services unrelated to each other) to accomplish a task or goal.

The easiest way to do this is for the client application to keep a detailed history of each request/response interaction and select (or compute) important application state data from that history. Most programming tools support serializing HTTP streams.

Typically, the following elements should be captured and stored by the client:

Request elements

- URL
- Method
- Headers
- Query string
- Request body

Response elements

- URL
- Status
- Content-type
- Headers
- Body

Note that the URL value from the request might not be the same as the URL of the response. This might be due to an incomplete request URL (e.g., missing the protocol element: `api.example.org/users`) or as a result of server-side redirection.

Example

The following is a snippet of code from the HyperCLI client application that manages the request and response elements of HTTP interactions:

```
...
// collect request info for later
requestInfo = {};
requestInfo.url = url;
requestInfo.method = method;
requestInfo.query = query;
requestInfo.headers = headers;
requestInfo.body = body;

// make the actual call
if(body && method.toUpperCase()!=="GET") {
  if(method.toUpperCase()==="DELETE") {
    response = request(method, url, {headers:headers});
  }
  else {
    response = request(method, url, {headers:headers, body:body});
  }
} else {
  if(body) {
    url = url + querystring.stringify(body);
  }
  response = request(method, url, {headers:headers});
}
response.requestInfo = requestInfo;
responses.push(response);
...
```

This code creates a push-down stack that holds information on each HTTP interaction. The same client application can now recall the history of interactions and use the request and response details to make decisions on how to proceed.

Here is the `SHOW HELP` output for the `DISPLAY` command that returns details on the available HTTP interaction history:

```
DISPLAY|SHOW (synonyms)
  REQUEST (returns request details - URL, Headers, querystring, method, body)
  URL (returns the URL of the current response)
  STATUS|STATUS-CODE (returns the HTTP status code of the current response)
  CONTENT-TYPE (returns the content-type of the current response)
  HEADERS (returns the HTTP headers of the current response)
  PEEK (displays the most recent response on the top of the stack)
  POP (pops off [removes] the top item on the response stack)
  LENGTH|LEN (returns the count of the responses on the response stack)
  CLEAR|FLUSH (clears the response stack)
  PATH <jsonpath-string|$#> (applies the JSON Path query to current response)
```

For more on HyperLANG and the HyperCLI, see Appendix D.

Discussion

Keeping track of HTTP requests and responses takes a bit of work to set up but results in a very powerful feature for client applications. To lower the cost of supporting this functionality, it can be helpful to create a reusable library that just handles the request/response stack and share that in all client-side HTTP applications.

Just tracking the interactions on the stack is usually not enough functionality for clients. You'll also need to monitor selected values and keep track of them as "state variables." You can see from the previous example that the suggested functionality includes the use of JSON Path as a way to inspect and select properties from within a single request/response pair. You can use something like this to monitor important state values, too. See Recipe 4.16 for more on tracking important client-side variables.

If you are working with a client application that does not manage its own file space (e.g., a web browser), you can set up an external service that will track the HTTP interactions and serve them up remotely (via HTTP!). This offers the same functionality but relies on a live HTTP connection between the client and the interaction service. That means the service might be a bit slower than a native file-based system and the service might be unavailable due to network problems between the client and the interaction service.

See Also

- Recipe 4.14, "Validating Incoming Data"
- Recipe 4.16, "Having a Goal in Mind"
- Recipe 6.8, "Ignoring Unknown Data Fields"
- Recipe 7.13, "Using State-Watch to Enable Client-Driven Workflow"

4.16 Having a Goal in Mind

There are times when the client application needs to continue running (making requests, performing work, sending updates, etc.) until a certain goal is reached (a queue is empty, the total has reached the proper level, etc.). This is especially true for client apps that have some level of autonomy or automated processing. There are some challenges to this, including cases where the *client* knows the goal but the *service(s)* being used by that client do not.

Problem

How do you program a client application to continue processing until some goal is reached? Also, what does it take to create client applications that have a goal that is "private" and not shared or understood by any services?

Solution

To build client applications that can continue operations until a goal is reached, you need to program some level of autonomy for that client. The application needs to know what property (or properties) to monitor, know the target value(s) for the selected properties, be able to locate and monitor the values over time, and know when an end has been reached and how to effect a stop in processing.

Programming this solution is essentially the work of most gaming engines or autonomous robot systems. Client applications can follow the classic artificial intelligence PAGE model:

Percepts
: The properties that need to be monitored.

Actions
: The tasks clients can commit to affect the values of the properties (read, write, compute).

Goals
: The desired end values.

Environment
: The playing field or game space in which the client operates. In our case, this would be the services with which the client interacts in order to see and affect the Percepts on the way to achieving the Goals.

Example

There are a couple of different types of goal models you are likely to need: a defined exit goal (DEG) and a defined state goal (DSG). Each takes a slightly different programming construct.

Defined exit goal

With DEGs, you write a program that halts or exits some process once the defined goal is reached. The following example was taken from an autonomous hypermedia client that navigates its way out of an undetermined maze:

```
function processLinks(response,headers)
{
  var xml,linkItem,i,rel,url,href,flg,links,rules;

  flg = false;
  links = [];
  rules = [];

  // get all the links in this document
  g.linkCollection = [];
```

```
xml = response.selectNodes('//link');
for(i=0;i<xml.length;i++)
{
  rel = xml[i].getAttribute('rel');
  url = xml[i].getAttribute('href');
  linkItem = {'rel':rel,'href':url};
  g.linkCollection[g.linkCollection.length] = linkItem;
}

// is there an exit?
href = getLinkElement('exit');
if(href!='')
{
  printLine('*** Done! '+href);
  g.done = true;
  return;
}

// is there an entrance?
if(flg==false && g.start==false)
{
  href = getLinkElement('start');
  if(href!='')
  {
    flg=true;
    g.start=true;
    g.href = href;
    g.facing = 'north';
    printLine(href);
  }
}

// ok, let's go through a door
rules = g.rules[g.facing];
for(i=0;i<rules.length;i++)
{
  if(flg==false)
  {
    href=getLinkElement(rules[i]);
    if(href!='')
    {
      flg=true;
      g.facing=rules[i];
      printLine(href);
      continue;
    }
  }
}

// update pointer, handle next move
if(href!='')
{
```

```
    g.href = href;
    nextMove();
  }
}
```

Note that the client application has a goal of "finding the exit" and continues to move from room to room until that goal is reached.

Defined state goal

There are cases when you need to program a client application to maintain a desired state—a DSG. For example, you need a client app to monitor the temperature of a room and activate heating/cooling units to maintain that state. Here is a snippet of code designed to do that:

```
// set control values
var roomURL = "http://api.example.org/rooms/13";
var min = 18;
var max = 22;
var wait = (15*60*1000);

// set up periodic checks
setInterval(checkTemp(roomURL,min,max),wait));

// do the check
function checkTemp(roomURL, minTemp, maxTemp) {
  var rtn, temp;

  rtn = "";
  response = httpRequest(roomURL);
  printLine(req)

  if(response.temp<minTemp) {
    rtn = response.form("heat");
  }
  if(response.temp>maxTemp) {
    rtn = response.form("cool");
  }

  if(rtn!=="") {
    response = httpRequest(rtn);
    printLine();
  }
}
```

In the simple DSG, the application is tasked to maintain a room temperature between 18°C and 22°C. The temperature is monitored every 15 minutes and, if needed, adjusted accordingly. Of course the service used by this client doesn't know what the client has in mind, it just responds with the information about the requested room when asked. You can see in this example that the response from the service includes data on the temperature (`req.temp`) as well as hypermedia controls to modify the state of the

temperature (`req.form("heat")` and `req.form("cool")`). It is worth pointing out that the monitoring service (`http://api.example.org/rooms`) might not know anything about how to adjust the room temperature—that might be handled by separate services, too.

Discussion

Coding goal-oriented clients (DEGs and DSGs) assumes a few key things. First, that the client "knows" the end goal ahead of time. Second, that the client already knows how to achieve this goal (finding the exit, adjusting the temp, etc.). Third, the client can find an accurate representation of the current state, and fourth, is able to properly evaluate the current state against the desired goal. All of these elements *must* be present for the goal-oriented client to be successful.

For this recipe, the client examples (DEG and DSG) assume that all the planning and criteria for evaluation are determined ahead of time and/or supplied when the client starts (via a config file). No machine learning or planning skills are built into the client application here.

In the DSG temperature example, the client knows that the end goal is a room temp between 18°C and 22°C. It also knows it can use the "heat" and "cool" forms in an HTTP response to achieve the goal. The client also knows that it can use the room URL to check on the current temperature, and knows how to evaluate the returned temp to its own internal minimum and maximum values.

Sometimes the evaluation step is more involved. For example, in the temperature management case, maybe the temperature settings are different at different times of the day, or if there are (or are not) people in the room, or based on the current cost per kilowatt per hour, etc. In these cases, there may be more values to monitor and use in a more involved comparison routine before taking the determined action.

It might also be the case that the evaluation is handled by another service (not the client). For example, a stock buy/sell decision might come from a service that accepts a wide range of inputs (which your client may supply), then returns an action recommendation that the client application then executes.

Whenever possible, it is a good idea to make both the properties to monitor and the local evaluation values a configurable set of data points. This avoids hardcoding decision-making data into the client application itself. You could, for example, provide the client with the control data via a configuration file or even a remote HTTP call to another service that holds configuration data.

Finally, it is important to add an escape option for goal-oriented client applications. For example, if the DEG can't find an exit to the maze, it will probably need to give up

after some point or be forever trapped in a loop. This escape value should be completely controlled by the client, too. Depending on some other service for the decision to escape could fail if the remote service is unavailable or sending bad data to the client.

The same is true for DSG-style client applications. In our temperature monitor example, imagine that the service that returns the sensor data is broken or unavailable. What does the client do? Does it just keep checking in vain? A better option is to add some code that triggers an alert when the sensors are not responding or if they report data far outside the assumed boundaries.

See Also

- Recipe 4.14, "Validating Incoming Data"
- Recipe 5.3, "Converting Internal Models to External Messages"
- Recipe 6.8, "Ignoring Unknown Data Fields"
- Recipe 7.13, "Using State-Watch to Enable Client-Driven Workflow"

CHAPTER 5
Hypermedia Services

The best software architecture "knows" what changes often and makes that easy.

—*Paul Clements,* Software Architecture in Practice *(Pearson)*

A primary challenge in designing and implementing APIs for services is balancing usability with evolvability. It is important for service APIs to be clear and easy to understand. At the same time it is critical that these same interfaces be defined in a way that allows for future modifications. Finally, the value of service APIs is often tied to their *reliability over time*. It's fine to be able define an API that solves an immediate problem. But as that problem varies over time, operating parameters change, and needs and goals drift over time, that API—ideally—should continue to be useful. That's a lot to ask of a service interface design!

Check out "Promoting Stability and Modifiability with Hypermedia Services" on page 33 for additional discussion on the foundations of creating service interfaces.

As the chapter's opening quote implies, knowing what changes often in a software design and making that easy is a worthy goal. This is especially true for service APIs. The API is the contract—the promise that needs to be kept.

A great example of this can be found in the way the HTTP protocol was designed. Almost every key aspect of HTTP is modifiable. HTTP methods, status codes, URLs, the list of possible headers, and the list of possible body formats are all abstract collections that can be amended over time. The *structure* of HTTP messages is consistent—that's the promise. However the *content* of HTTP messages is variable—even, in some cases, the content of the *action control* information (e.g., HTTP methods, URLs,

available message formats, etc.). This is the key to creating stable, valuable APIs for services, and that's what we'll be talking about in this chapter.

Since this book is focused on *interfaces*, we will not spend time in this chapter talking about the internal details of the services *behind* the interfaces. However, it is important to have a solid understanding of the inner workings of scalable, robust, and reliable services on the web. See Appendix B for some recommended material on designing and building services.

The recipes in this section of the book are all focused on making it easy to design and implement APIs that strike the right balance between specificity and evolvability. A key feature of this type of API is the ability to include operational metadata at runtime instead of relying solely on design and build-time descriptions. In our case, this ability to modify the messages at runtime will be driven using hypermedia formats that follow the same general design pattern of the HTTP protocol itself: a clear message structure to support variable contents.

When creating service APIs, you need to balance the exposure and reliance on runtime and design-time metadata (see Figure 5-1). Another key element in successful service interface implementation is the ability to support multiple representation formats via content negotiation.

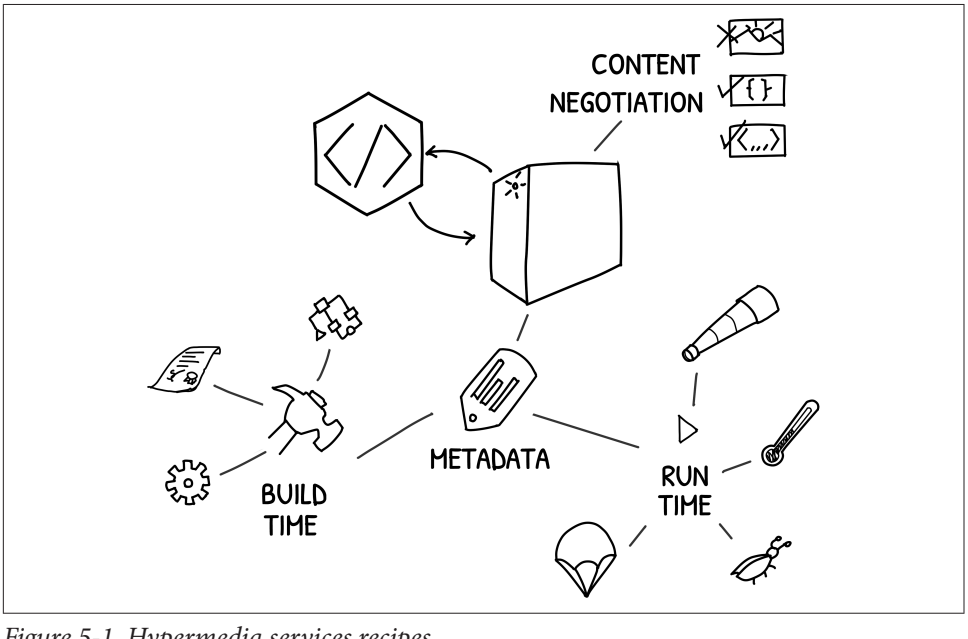

Figure 5-1. Hypermedia services recipes

5.1 Publishing at Least One Stable URL

When services publish an interface on the network, these services need to be easily (and consistently) reached by API consumers. That means publishing your API at a stable location (using a URL). But does that mean you can never change the published location? And, if you can change it, how do you tell API consumers about this change?

Problem

How can you establish a stable network location (URL) for your service API, and what steps are needed to make sure your API is findable even if the underlying service has moved to a new location?

Solution

Service interfaces should promise at least one stable URL that API consumers can use to locate and interact with the service behind that API. The exact URL does not matter. This should be the "starting point" of the API. It may be a resource that returns details about the API (including the entry point). It may be the initial listing or state of the service (e.g., the default list of users, etc.). Or it may be some other response that API consumers can rely upon to initiate access with the service (e.g., a login, a way to establish a stateful session, etc.).

The value of the URL could be something as simple as *http://api.example.com/home* or as complicated as *http://v1.home.api.example.org/q1w2e3r4t5*. What matters is that there is at least one location that consumers can memorize (i.e., write into their source code or configuration files) that will ensure that the API consumer can connect with the service API. Finally, there is no requirement that a service API have *only one* published stable URL. However, your API should have *at least one* URL that API consumers can count on over time.

Stable URLs should be published in the API human-readable documentation. They may also appear as a `Link` header in HTTP or as a `link` element in an API response body that supports inline `link` elements.

In some cases, it might be wise to register a "well-known" URL for your service API. However, this is not a trivial effort. See RFC 8615 and RFC 7595 for details.

Publishing a stable URL does not mean the service behind that URL cannot move. If the service moves, requests to the stable URL can be redirected to the new location using the HTTP status code `301 Moved Permanently`.

Example

When indicating your API's stable URL in human-readable documentation, be sure to call it out as your API's *promise* that the URL will be honored into the future and, if needed, will redirect API client applications to the new service location.

You can use the `Link` header in an HTTP response to emit a stable URL for your API:

```
**** REQUEST
GET / HTTP/1.1
Host: api.example.org

**** RESPONSE
HTTP/1.1 200 OK
Content-Type: application/vnd.collection+json
ETag: "p0o9i8u7y6t5r4e3w2q1"
Link: <http://api.example.org/home>; rel="home"
...
```

The stable URL may also appear in the body of any (or all) responses that support inline links. Here is an example shown in a Collection+JSON response:

```
{ "collection" :
  {
    "version" : "1.0",
    "href" : "http://api.example.org/friends/",

    "links" : [
      {"rel" : "home", "href" : "http://api.example.org/home"}
    ],

    "items" : [...],
    "queries" : [...],
    "template" : [...]
  }
}
```

When the service location moves, the API should continue to honor the originally published stable URL and automatically redirect the API consumer to the new location URL:

```
**** REQUEST
GET / HTTP/1.1
Host: api.example.org

**** RESPONSE
HTTP/1.1 301 Moved Permanently
Location: http://new.example.org/home

**** REQUEST
GET /home HTTP/1.1
Host: new.example.org
```

```
**** RESPONSE
HTTP/1.1 200 OK
Content-Type: application/vnd.collection+json
ETag: "p0o9i8u7y6t5r4e3w2q1"
Link: <http://api.example.org/home>; rel="home"
...
```

Discussion

Services interfaces should support *at least one* stable URL. Any stable URL is a promise to API consumers that should last the test of time. It is certainly possible to commit to promising more than one stable URL, but that just adds to the level of work API designers need to do to keep the long-term promises of *all* the stable URLs.

Keep in mind that client applications can "memorize" as many URLs as they wish in order to create an efficient API experience with a service. Just because the service only promises one URL doesn't mean the client application can't learn and remember other URLs in that API.

Services should not assume (or require) that all clients start their interactions by first visiting the stable URL. No matter the number of stable URLs promised by the API, clients are free to interact with the interface in any way they wish as long as it is making valid requests.

I typically use the link relation value `"home"` to indicate a stable URL for a service API. You can use any value you wish as long as you document it well and use the identifier consistently.

It is a good idea to emit the stable URL as a `Link` header in all your API responses. It may also make sense to emit it as part of the response body whenever possible (e.g., you won't be able to emit it for PNG or MP3 bodies!).

See Also

- Recipe 4.1, "Limiting the Use of Hardcoded URLs"
- Recipe 4.8, "Every Important Element Within a Response Needs an Identifier"
- Recipe 5.13, "Improving Service Discoverability with a Runtime Service Registry"

5.2 Preventing Internal Model Leaks

A common way in which service interfaces "break" is through changes in the underlying data, object, or process models within the service code. It's important to prevent these internal elements from "leaking" out into the external API.

Problem

How can you reduce the possibility that a service's internal data, object, and/or process model is directly exposed in the service API?

Solution

The best way to limit the possibility of exposing internal service details in the external API is to make sure the external interface is designed to exist apart from the internal service models. That means defining service API functionality on its own terms—not simply exposing the direct service code to the "outside world."

Essentially, API implementations (the interfaces) need to keep the service's internal models private. To do that you need to make sure the data properties, property collections, and input/output parameters of the API are expressed as coherent elements themselves—that they can "stand alone" separately from any internal service models. The interface should be treated as its own independent design and implementation effort, not just a "wrapper" of an existing backend service component.

Example

The quickest way to avoid leaking data/object models is to ignore them when designing the API. For example, let's assume there is a simple ToDo service that supports two local data object collections: `item` and `user`. They might look like this:

```
{"items" : [
  {"id":"q1w2e3r4","text":"This is an item","status":"active","nick":"mork"},
  ...
]}

{"users":[
  {"nickname":"mork","name":"Mark Morkelsen"},
  ...
]}
```

Let's also assume that the possible `status` field values are limited to `active` or `closed`.

You might think that the interface should also model two objects (`item` and `user`). But it would be just as valid to model three:

```
{"items" : [
  {"id":"q1w2e3r4","text":"This is an item","status":"active","nick":"mork"},
  ...
]}

{"users" : [
  {"nickname":"mork","name":"Mark Morkelsen"},
  ...
```

```
]}

{"status" : [
  {"name" : "active"},
  {"name" : "closed"},
  ...
]}
```

or even one:

```
{"todo" : [
  {"id":"q1w2e3r4","text":"This is an item",
    "status":"active",
    "nickname":"mork",
    "name":"Morkelsen"
  },
  ...
]}
```

The point here is to *not* blindly accept the service model as the interface model. Yes, if or when the service internal model changes, there may be some work needed to maintain compatibility between the published interface and the local service. But that's what API designs are meant to accomplish.

Discussion

Whenever possible, I model my API properties as a single, flat collection (like the preceding example). This simplifies the client end of the process and frees the supporting services to use any local data storage model they wish. Services can even change the storage model without adversely affecting the API. This denormalization of any internal model makes it easier to maintain a stable external model even when the internal model changes over time.

Years ago, when talking about this subject, I coined the axiom: "Remember, when designing your web API, your data model is not your object model is not your resource model is not your message model." I also gave a talk called "Web API Design Maturity Model" that expands on this notion.

It is important to remember that the API is an independent interface. It's fine if the interface expects the person creating a new record to send a "denormalized" collection of properties that will be regrouped and written to multiple storage locations. As long as the service can safely manage the data (e.g., make sure there are no lost updates or data conflicts), anything goes.

We'll dig into the details of dealing with data storage in Chapter 6.

API designers are free to create any number of HTTP resources they wish in order to support the expected interactions. Those interactions don't need to follow the CRUD (create, read, update, delete) meme so common for HTTP APIs. For example, Table 5-1 is a resource designed to support the ToDo service referred to throughout this recipe.

Table 5-1. Possible resource design for ToDo service interface

Action	URL	Method	Request body	Response body
Read List	/todo/	GET	none	[{id,text,status,nick,name}]
Filter List	/todo/?text={text}	GET	none	[{id,text,status,nick,name}]
Create Item	/todo/	PATCH	{id,text,status,nick,name}	[{id,text,status,nick,name}]
Update Item Text	/todo/	PATCH	{id,text}	[{id,text,status,nick,name}]
Update Item Nick	/todo/	PATCH	{id,nick,name}	[{id,text,status,nick,name}]
Update Item Status	/todo/	PATCH	{id,status}	[{id,text,status,nick,name}]

Notice that Table 5-1 indicates that the ToDo list is a single resource that can be modified to add and edit items from that single resource. How the service organizes this information internally and/or *stores* this information is of no real interest to the interface consumer as long as the client application can do the needed work.

Table 5-1 doesn't include additional HTTP features, such as `ETag`, `If-Match`, `Last-Modified`, and other integrity headers. See Recipe 6.2 for details.

See Also

- Recipe 3.1, "Creating Interoperability with Registered Media Types"
- Recipe 3.3, "Sharing Domain Specifics via Published Vocabularies"
- Recipe 3.4, "Describing Problem Spaces with Semantic Profiles"
- Recipe 3.5, "Expressing Actions at Runtime with Embedded Hypermedia"
- Recipe 4.7, "Using Schema Documents as a Source of Message Metadata"
- Recipe 5.3, "Converting Internal Models to External Messages"
- Recipe 5.4, "Expressing Internal Functions as External Actions"

- Recipe 6.1, "Hiding Your Data Storage Internals"
- Recipe 6.3, "Hiding Data Relationships for External Actions"

5.3 Converting Internal Models to External Messages

To support loose coupling between evolving services, you need to establish a stable basis for communication between those services. A proven successful strategy is to rely upon well-known, highly structured registered media types as the representation format for API responses.

That means your service interface needs to respond with the same *semantic* information while varying the message format.

Problem

How do we organize our service interface to support content negotiation for API consumers while still maintaining a low-cost/low-effort service API? What patterns should we support within our interface code in order to support representing the backend service data consistently in a standardized and well-known media type? Finally, what are the advantages and challenges of representing the same internal data for differing response message formats?

Solution

An important way to make sure your service *interface* design is not too tightly coupled to your service's internal data (or object) model, is to approach the representation step as a separate design effort. In other words, actively engage in the process of designing your representations—don't just settle for a simple, direct serialization of internal models into external messages.

A series of formats were created to represent information for HTTP. These formats contain enough structure (e.g., `<html>`, `<head>`, `<body>`, etc.) to support client applications that can bind to the message format instead of binding to the message *content* (e.g., `givenName`, `id`, `telephone`, etc.).

For a list of recommended response media types, see Appendix C.

The process of mapping parts of the service's internal data/object model to the interface's external object model usually takes a mix of algorithmic application and creative design thinking. For some formats, like unstructured XML and JSON, the

mapping can be quite literal: a simple serialization of the data into a message. But this can be a bad idea. For example, direct serialization of the data model into a message model exposes the internal model to the outside world and results in a brittle implementation that is likely to break if or when the internal models change—which they will! See Recipe 6.10 for details.

Instead, it is better to select a structured media type (SMT) like SIREN, Collection +JSON, or even HTML as the basis for exchanging representations. Mapping the internal data to one of these formats takes a bit more effort in the beginning—you can do this "by hand" or rely on a prebuilt representative library for that media type —but pays off over time since any future changes or even additional interfaces can use the same library to render valid external messages for API consumers.

For more on designing service interfaces independent of the underlying service, see Recipe 5.17.

Example

Service interface code that supports selectable media type responses needs to implement two key runtime support elements. First, the service API needs to be able to discover which formats the client prefers, and second, the service API needs to use that information to render internal data in the preferred format.

HTTP defines the `Accept` header to allow clients to include their format preferences when they send a request. When the service API sees an `Accept` header, it needs to parse the value of that header and make a proper selection. You can review the HTTP specification on the `Accept` header for details on how to interpret its contents.

A simple exchange might look like this:

```
*** REQUEST
GET /todo/list HTTP/1.1
Host: api.example.org
Accept: application/vnd.collection+json, application/vnd.uber+xml

*** RESPONSE
HTTP/1.1 200 OK
Content-Type: application/vnd.collection+json
```

Note that the service interface selected the `application/vnd.collection+json` format here. When more than one preference is offered, services have some freedom on which format they can return—including returning a default format that wasn't even listed by the client! See more on this in Recipe 5.5.

Once the service API selects a format to use for rendering the response, it needs to map the internal data to the output representation per the specifications of the selected response format. Often this is an easy fit, but sometimes internal data is a bit tough to render and might even be left out of a response.

Here's an example of some internal service data that needs to be represented in responses:

```
{
  "user" : {
    "id":"q1w2e3r4t5",
    "givenName":"Mark",
    "familyName":"Morkelsen",
    "nickName":"mork",
    "telephones":[
      {"type":"home", "value":"1-234-567-8901"},
      {"type":"work", "value":"1-987-654-3210"}
    ]
  }
}
```

Now, let's assume the client has sent the following request:

```
GET /users/q1w2e3r4t5 HTTP/1.1
Host: api.example.org
Accept: application/json
```

A simple (but not recommended) way to represent the internal data would be to just serialize it directly:

```
*** RESPONSE
HTTP/1.1 200 OK
Content-Type: application/json
...

{
  "user" : {
    "id" : "q1w2e3r4t5",
    "givenName" : "Mark",
    "familyName" : "Morkelsen",
    "nickName" : "mork",
    "telephone" : [
      {"type" : "home", "value" : "1-234-567-8901"},
      {"type" : "work", "value" : "1-987-654-3210"}
    ]
  }
}
```

Next, let's look at a request for a representation using the `text/csv` format:

```
GET /users/q1w2e3r4t5 HTTP/1.1
Host: api.example.org
Accept: text/csv


*** RESPONSE
HTTP/1.1 200 OK
Content-Type: application/json
...
```

```
"id","givenName","familyName","nickName","telephone_home","telephone_work"
"q1w2e3r4t5","Mark","Morkelsen","mork","1-234-567-8901","1-987-654-3210"
```

Notice the internal `telephone` object was rendered as a set of flat properties in the CSV response. Clients may not even know the internal shape of the data.

Finally, here's an example representation when the client included the `application/vnd.collection+json` media type in the request:

```
{ "collection" :
  {
    "version" : "1.0",
    "href" : "http://example.org/users/q1w2e3r4t5",

    "links" : [
      {"rel" : "users", "href" : "http://example.org/users"},
      {"rel" : "products", "href" : "http://example.org/products"},
      {"rel" : "services", "href" : "http://example.org/services"}
    ],

    "items" : [
      {
        "href" : "http://example.org/users/q1w2e3r4t5",
        "data" : [
          {"name" : "id", "value" : "q1w2e3r4",
            "prompt" : "Identifier"},
          {"name" : "givenName", "value" : "Mark",
            "prompt" : "First Name"},
          {"name" : "familyName", "value" : "Morkelsen",
            "prompt" : "Last Name"},
          {"name" : "nickName", "value" : "Mork",
            "prompt" : "Nick"},
        ],
        "links" : [
          {"rel" : "telephones", "prompt" : "Telephones"
            "href" : "http://examples.org/users/q1w2e3r4t5/telephones"}
        ]
      }
    ]
  }
}
```

In this last case, quite a bit of "extra" information was supplied by the representative in the API service, including mapping `prompt` values to each data point, including `link` elements that point to related data, and even moving the `telephone` data array to another, related resource.

The key message here is that designing representative for service data should not be limited to blindly serializing the internal models.

Discussion

During the implementation phase of a service API, it is a good practice to document the rules you are using to convert internal models to external messages. This will help share the algorithms with other developers and make it easier to debug any errors you find along the way. It is not, however, a good idea to publish this representation description to API consumers. They do not need to know the internal data or object models behind your interfaces.

As time goes on, you are likely to encounter cases where the internal models of your service change. In many cases these changes should *not* result in changes to your external messages. For example, when the internal model property `user.id` is changed to `user.identifier`, there is no need to change the external property.

However, some internal model changes include new properties (`user.hatsize`), and these new properties may be reflected in the external messages when those changes do not result in breaking existing API consumers. Adding a field at the end of a `text/csv` row might be fine, but rearranging the field order in the rows (e.g., inserting the new field as the first field in the row) is likely a bad idea.

If you anticipate the internal models will change frequently, be sure to select a representation format that is designed to reduce the impact of such changes. For example, Collection+JSON is good for varying models (all properties are represented as `data.name` and `data.value`), but SIREN and HAL (since they bind to object properties directly) are not.

While you can hardcode your service interface to *always* emit the same format (HAL, SIREN, etc.), it is usually a good idea to honor the HTTP content negotiation process (see Recipe 5.6) to allow API consumers to indicate their format preferences at runtime using the `Accept` request header. See Recipe 5.5 for additional details on how to handle a client's media type preferences.

I typically support at least two and try to support three or more, depending on the situation. First, it is a good idea to select one media type with a high Hypermedia Factor score, for example, Collection+JSON or SIREN (see Recipe 4.9 for more on H-Factors). Second, it is a good idea to support a simple serialization format like JSON, XML, and/or CSV since many API consumers will be able to use these simple formats quickly. To support additional formats, you can add any other formats you and/or your API consumers prefer to use.

If appropriate, it can also be helpful to support an HTML representation of your service API. It can help service developers while they test the interface design. It can also help API consumers who want to "play around" with the API without investing lots

of time in coding an API client. Finally, the HTML format can be a solid choice as one of the high H-Factor media types in your service collection.

See Also

- Recipe 3.2, "Ensuring Future Compatibility with Structured Media Types"
- Recipe 4.5, "Negotiating for Profile Support at Runtime"
- Recipe 4.6, "Managing Representation Formats at Runtime"
- Recipe 4.9, "Relying on Hypermedia Controls in the Response"
- Recipe 5.5, "Advertising Support for Client Response Preferences"
- Recipe 5.6, "Supporting HTTP Content Negotiation"
- Recipe 6.10, "Modifying Data Models in Production"

5.4 Expressing Internal Functions as External Actions

Once you adopt the practice of standardizing responses using structured media types (SMTs), you also need to address the challenge of properly expressing the internal functionality of the service(s) behind your interface. To do this, you need to rely on a practice of expressing not just the service's data models but also that service's action models—the functions of the service that need to be accessed via the external API.

Problem

How do you decide which internal functions within the service need to be exposed via the API, and how do you express those methods in a consistent manner for API consumers to find and operate as expected? What formats and/or patterns are available for converting internal service functions into external interface actions?

Solution

The most direct way to expose a service's internal functions as external actions is to represent those functions using hypermedia controls (see Recipe 4.9). For example, in HTML the function name (`<form name="approveUser" …>`), input arguments (`<input>` controls), and output return values can be used to represent the internal service function. However, it is not always possible (or even a good idea) to bind external actions directly to internal service operations.

It is usually better to rely upon established external vocabularies when expressing internal service functions. For example, an internal data parameter might be named `firstName`. But that same parameter would be expressed as `givenName` in the external interface (see Recipe 3.3). The API code can handle the translation between internal and external names. This has the advantage of shielding the external interface from

internal changes to the service. To extend the example, updates to the service might result in `firstName` changing to `fNameValue`. In this case, only the middleware code needs to be updated, not the actual external interface.

Finally, some internal actions—especially ones involving complicated computations or processes—are best expressed in a more simplified way. A service (`onboardingSvc`) might need to contact multiple dependent services (`userSvc`, `customerRelationsSvc`, and `accountSalesSvc`) and pass data between them in order to complete a task. It is often better to express this externally as a single action (`sendOnboardingData`) instead of multiple dependent actions (see the following example for details).

Essentially, treat the act of designing the process interactions of your API as an independent activity. You can be as creative/inventive as you wish. Design a good interface; don't just echo the service internals. Assume that there may be a "translation" step between the API exchange and the internal service innovation. Above all, work to maintain a loose coupling between the underlying service and the service interface you publish.

Example

In a direct translation from internal function to external action, you can start with an example of a service function:

```
function approveUser(userId, nickname, approver,level) {
  var approval = {};

  approval.userId = userId;
  approval.nickname = nickname;
  approval.approver = approver;
  approval.level = level;

  return data.create(approval);
}
```

And then translate these internal elements into an HTML `FORM` object in a response message:

```
<form name="approveUser" enctype="application/x-www-form-urlencoded"
  method="post" action="http://api.example.org/users/approvals"
  target="approvalDisplay">
  <input name="userNick" type="string" value="mork" />
  <input name="approverName" type="string" value="Mr. Roboto" />
  <input name="approveLevel" type="string" value="nominal" />
  <input type="submit" value="Approve User" />
</form>
<iframe id="approvalDisplay" />
```

However, as we already discussed, direct translation is not usually a good idea. Here's an example that relies on a well-known external vocabulary (in this case, values from

the Schema.org library) to express the external action using a standardized vocabulary even though that vocabulary is not supported by the underlying service.

Consider this simple internal method template for updating an existing user object:

```
function updateUser(id, fname, lname, email) {

  ...

  return userObject
}
```

Now, let's look at the interface action (expressed as a Collection+JSON template) using property names taken from the Schema.org library:

```
{
  "collection" {
    "template" : {
      "data" : [
        {"name" : "identifier", "value" : "q1w2e3r4", "prompt" : "User ID"},
        {"name" : "givenName", "value" : "Mark", "prompt" : "First Name"},
        {"name" : "familyName", "value" : "Morkelsen", "prompt" : "Last Name"},
        {"name" : "email", "value" : "mork@example.org", "prompt" : "Email"}
      ]
    }
  }
}
```

Finally, the code that accepts the update action can handle the translation between internal and external names. For this example, the method that maps to the external form uses the Schema.org property names (`identifier`, `givenName`, `familyName`, and `email`) and stores the values of these parameters into the local user object properties (`id`, `fname`, `lname`, and `email`):

```
function updateAction(identifier, givenName, familyName, email) {
  var user = data.read(identifier);
  if(user) {
    user.id = identifier;
    user.fname = givenName;
    user.lname = familyName;
    user.email = email;
    user = data.write(user);
  }
  return user;
}
```

There may also be scenarios when the interface has external actions that do not map directly to internal functions. Consider the following external action:

```
<form name="declineContract" method="post" action = "...">
   <input name="customerId" value="q1w2e3r4t5" />
   <input name="contractId" value="o9i8u7y6.t5r4" />
   <input name="salesRepName" value="Mandy Miningham" />
```

```
    <input name="reviewerName" value="Mark Morkelsen" />
    <input name="reasonCode" value="Q201.B" />
    <textarea name="comments">Unable to locate collateral</textarea>
    <input type="submit" value="Decline Contract" />
</form>
```

There is no `declineContract` method in the underlying service code behind this interface. However, there are a handful of actions that must be completed when a contract is declined:

1. Create a `declined` log record.
2. Update the `customer` record.
3. Update the `contract` record.
4. Update the `salesRep` record.
5. Update the `reviewer` record.
6. Update the `nationalCredit` agency.

In this case, the service API most likely has a set of functions it needs to perform against several dependent services. Some of these functions might be local (e.g., internal objects in the service code), and some might be remote (e.g., other external services like the `nationalCredit` agency). All these details are "hidden" from the API consumer.

The last example doesn't show any handling of possible errors, failed data updates, etc. For more on how to handle these cases, see Chapter 7.

Discussion

Adopting the practice of "designing the external actions" is a great way to improve the loose coupling aspect of your API. This is especially true when you create external actions that have no direct service equivalent, since changes to the underlying service need not be reflected directly in these independent actions. Basically, the more work you do to design interface actions, the less likely it will be that future service changes —even seemingly breaking changes—will adversely affect the API.

Sometimes a service will have a sequence of internal operations (`intializeCustomer`, `collectUserData`, `collectBankData`, and `finalizeCustomerData`). Whenever possible, it is a good idea to express these internal functions as a single external action where you collect all the important data and then let the API code sort out the sequence of actions. This can be done as a single "form" with all the inputs or, if needed, you can use the "work in progress" workflow (see Recipe 7.10). This avoids

dealing with future changes in the order of the sequence, or adding or removing steps.

There are times when internal services rely on Boolean flags to control operations. For example:

```
function approveUser(bool) {

  if(bool===true) {
    ...
  }
  else {
    ...
  }

  return results;
}
```

It is not a good idea to expose Boolean operations for external actions. This limits your ability to modify or expand this list over time. Instead, it is a good idea to expose enumerators (expressed here in HAL-FORMS) and let clients and services decide how they want to render these values internally:

```
{
  "_templates" : {
    "default" : {
      ...
      "properties" : [
        {
          "name" : "approveUser",
          "prompt" : "User Approval",
          "options" : {
            "selectedValues" : ["No"],
            "inline" : ["No","Yes"]
          }
        }
      ]
    }
  }
}
```

Now, code within the API can translate the `"Yes"` / `"No"` values to `true` or `false` before calling the underlying service. In addition, if future changes to the service result in a possible `pending` status, the `options` element of the form can be updated without any other changes to the API.

In most of my API implementations, I write up a small "mapping" function that automates the task of translating internal and external names using a set of declarations. That way I only need to update the rules in a single location for the entire API.

See Also

- Recipe 4.4, "Coding Effective Clients to Understand Vocabulary Profiles"
- Recipe 4.9, "Relying on Hypermedia Controls in the Response"
- Recipe 5.7, "Publishing Complete Vocabularies for Machine Clients"
- Recipe 6.3, "Hiding Data Relationships for External Actions"
- Recipe 7.16, "Short-Term Fixes with Automatic Retries"
- Recipe 7.17, "Supporting Local Undo or Rollback"

5.5 Advertising Support for Client Response Preferences

The HTTP protocol provides a handful of content-related parameters that clients can use to indicate preferences. HTTP supports a collection of header parameters, the `OPTIONS` method, and link relation values. However, sometimes you need a more extensive way to identify and document client preferences for interactions. The solution is to rely on a mix of HTTP protocol options and a few additional semantic references via a separate, addressable preferences resource.

Problem

How can a service advertise to API consumers all their possible preference options in a consistent and scalable way? What HTTP headers are available? When does it make sense to use the HTTP `OPTIONS` method? And when is it best to rely on link relation and other values to signal a service API's capabilities?

Solution

One of the advantages of using the HTTP protocol for accessing network services is that HTTP has a wide range of "tuneable" values that clients and servers can use to negotiate the details of request and response parameters. Essentially, services can "advertise" one or more selectable aspects of the message exchange, and clients can use this information to inform services of their preferences—all at runtime.

You'll notice that this list contains elements from a wide range of sources, such as the HTTP protocol, an HTML `FORMS` property, and even a named link relation value. The work of advertising selectable response options has a checkered history, so we have to make do with the possibilities before us.

The following is a quick review of all the "tuneable" values in HTTP and how clients and services can use them to customize message exchanges:

`Accept`
: The HTTP `Accept` header can be used to return a list of supported response media types. Although it is not required, the values in this list should be from the media types listed in the IANA Media Types document or some other source.

`Allow`
: The HTTP `Allow` header is used to return a list of supported HTTP methods for the service. The values in this list should be taken from the IANA Hypertext Transfer Protocol Method Registry.

`enctype`
: The `enctype` property of HTML forms can be used to return a list of supported request media types. Although it is not required, the values in this list should be from the media types listed in the IANA Media Types document or some other source.

`charset`
: The HTTP `charset` parameter can be used to indicate the list of supported character sets for response bodies. The values in this list should be taken from the IANA Character Sets document.

`encoding`
: The HTTP `encoding` parameter can be used to return a list of supported encodings for response bodies. The values in this list should come from the IANA Content Encoding Registry.

`language`
: The HTTP `language` tag can be used to return a list of supported natural languages for response bodies. The list of valid values should be taken from the IANA Language Subtag Registry.

`profile`
: The `profile` link relation value can be used to return a list of supported profiles for responses. The contents of this element can be a list of valid URIs that represent supported vocabularies or other descriptive documents (see Recipe 5.7 for details).

There is another client preference value that HTTP supports: `accept-ranges`. This allows clients to signal to services that they can support range values for returning responses via the `range` header. You can check out the HTTP specification for details.

These should be accessible for client applications via the HTTP `OPTIONS` method or a resource representation returned when following the `"meta"` link relation value (`rel="meta"`). See the example for details.

Clients can request the service's supported client preferences and then use values from this response in their request to indicate to the service which formats, languages, encodings, etc. they wish the service to use in responses.

Even when there is only one possible response that the service supports (e.g., `"language":"en"`), the service should still return the full `meta` resource in order to help clients discover the values for these important parameters.

It is a good idea to support a link on the "Home" resource of your service interface that points to the client preferences `meta` resource. This `meta` resource is where clients can expect to find the list of (and possible values for) all the client preferences that service supports.

Figure 5-2 is a diagram of the interaction model for supporting the `meta` resource list of client preference options.

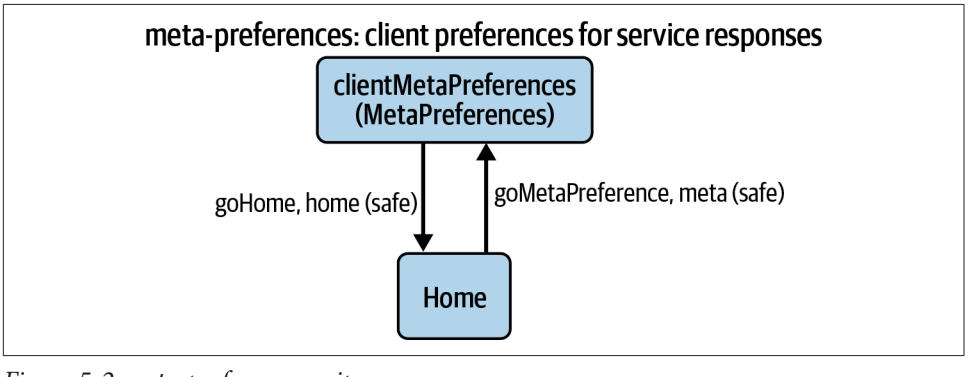

Figure 5-2. `meta` preference recipe

For details on the `meta` client preferences resource, check out the online collection of ALPS documents associated with this book.

The following section shows examples of what is returned by the API when the `meta` link is activated.

Example

When requested, the `meta` resource representation should list all the possible client preference parameters supported by the API along with all the acceptable values for

each. It is, of course, possible that the service API does not support all (or maybe any) of the client preference options. The content of the response will vary accordingly.

Here is an example (in HAL format) of the response to the `meta` link or executing the `HTTP OPTIONS` method:

```
{
  "_links" : {
    "self" : {"href" : "http://api.example.com/user-service/meta-preferences"},
    "home": {"href" : "http://api.example.com/user-service/"},
    "profile" :
   {"href" :
    "https://webapicookbook.github.io/alps-documents/meta/meta-preferences.json"
      }
  },
  "allow" : "GET PUT PATCH DELETE HEAD OPTIONS",
  "accept" : "application/vnd.hal+json application/vnd.collection+json",
  "enctype" : "application/x-www-form-urlencoded application/json",
  "charset" : "utf-8, iso-8859-1;q=0.7",
  "encoding" : "deflate gzip compress",
  "language" : "en es fr",
  "profile" : "https://alps.example.org/fhir-4.0.1"
}
```

While it is not required (by the HTTP specification) to include a body in the `OPTIONS` response, it is a good practice to return this `meta-preferences` representation anyway. In cases where the `OPTIONS` are specific to a single resource, services can modify the `meta-preferences` response accordingly. For example, a download link might support more than one response format. Here is an example `OPTIONS` response in SIREN format:

```
**** REQUEST ****
OPTIONS /file-system/download HTTP/1.1
Host: api.example.org
...

*** RESPONSE ***
HTTP/1.1 200 OK
Content-Type: application/vnd.siren+json
Content-Length: XX
Cache-Control: max-age=604800
Allow: GET PUT DELETE HEAD OPTIONS
Accept: application/zip application/gzip
Accept-Charset: utf-8
Accept-Encoding: compress
Accept-Language: en
Link: <https://webapicookbook.github.io/alps-documents/about/about.json>; \
  rel="profile"

{
  "class" : [ "meta preferences" ],
```

```
  "links" : [
    { "rel" : ["self"], "href" : "http://api.example.org/file-system/download"},
    { "rel" : ["home"], "href" : "http://api.example.org/file-system/"},
    { "rel" : ["profile"],
      "href" : "https://api.example.org/profiles/meta-preferences.json"}
  ],
  "properties" : {
    "allow" : "GET PUT DELETE HEAD OPTIONS",
    "accept" : "application/zip application/gzip",
    "enctype" : "application/json",
    "charset" : "utf-8",
    "encoding" : "compress",
    "language" : "en",
    "profile" : "https://alps.example.org/downloads/"
  }
}
```

In the `OPTIONS` response, you can see that both the header space and the response body contain the list of client preference parameters. It is a good idea to include both sets of information since some client applications may only parse the header collection and skip the content in the response body.

Discussion

It is rare that a single service interface supports the full range of the client preferences listed here. The most common ones that clients are interested in are: `allow`, `accept`, `enctype`, and sometimes `language`. The `charset` and `encoding` preferences are rarely indicated.

Even if the service interface supports only one value for the preference item, that item should be listed in the `meta-preferences` response. In this way, the `meta-preferences` response becomes the runtime documentation of service options, and that means API consumers will be able to rely on this response to get an up-to-date listing of these parameters.

Be sure to include an item in your API documentation that tells developers that your interface supports `OPTIONS` and/or the `meta` link relation value.

There is a downside to relying solely on HTTP `OPTIONS` to communicate supported client preferences. The HTTP specification explicitly states that `OPTIONS` is noncacheable. If you have lots of API consumers sending `OPTIONS` requests, you may run into scaling problems for your API. However, offering a standalone `meta-preferences` resource (for example, `<link rel="meta" href="http://api.example.org/meta-`

preferences/" />) means you can mark that resource with a long cache lifetime and reduce the expense of composing and returning client preference responses.

See Also

- Recipe 3.3, "Sharing Domain Specifics via Published Vocabularies"
- Recipe 4.5, "Negotiating for Profile Support at Runtime"
- Recipe 5.7, "Publishing Complete Vocabularies for Machine Clients"
- Recipe 5.8, "Supporting Shared Vocabularies in Standard Formats"

5.6 Supporting HTTP Content Negotiation

One of the unique aspects of the HTTP protocol is the ability to select preferred response body formats. Borrowed from the email specifications, the `content-type` header implementation for HTTP has led to the creation of almost 1,500 different media types registered in the official IANA Media Types registry, with more being added each year. The challenge is knowing which media type to use, when to use it, and how to make it possible for both clients and services to work out the details of content negotiation.

Problem

HTTP allows clients and services to engage in "content negotiation"—the process of selecting which media type to use when exchanging messages. How does this work? What is the different between proactive and reactive content negotiation? And what do service interfaces need to do to support content negotiation when it is appropriate?

Solution

An important feature of HTTP is the ability to support multiple response formats. This allows the same resource to return HTML for some consumers, CSV for others, and so forth. Even more powerful is HTTP's support for negotiating the response format at runtime. In other words, API consumers and providers can work out just which message format is used in a response at the very time the request is made.

There are two main "types" of content negotiation described in the HTTP specification: proactive and reactive. Each has advantages and challenges.

Proactive content negotiation

Proactive content negotiation (PCN) is the easiest to implement for HTTP services. With PCN, API clients send a list of one or more preferred media types for responses,

and the server, using the provided list, proactively decides which media type to use. This is sometimes called "server-driven content negotiation."

Clients indicate their response preferences using the `Accept` header in requests, and servers indicate their final section using the `Content-Type` header in responses (see the examples later in this recipe).

In PCN, clients *advise* servers of their preference, and the servers make the final determination of the response format. It should be noted that it is possible that the service will respond on a format that was *not* included in the client's `Accept` header. Using the JSON example here, if the service does not support any JSON formats, that service might respond with `text/plain` or `application/HTML`.

Reactive content negotiation

An alternative to proactive content negotiation by the service is reactive content negotiation (RCN). In RCN, the service sends the client a list of possible representation formats, and the client is expected to select the preferred format from the list and make the request a second time, specifying which response format has been selected. This puts the power of selection squarely in the hands of the client application but also means an additional "round trip" is made each time.

RCN works well when there are lots of variables (language, message format, encoding scheme, etc.), but the details of how server and client interact to select the preferred response format are not spelled out in the HTTP specification. A common approach is for servers to respond with HTTP status code `300 Multiple Choices`, with a response body that lists options for clients to review. The `300 Multiple Choices` response often carries a series of `Link` headers, and possibly a `Location` header in case the client wants to just redirect automatically.

Example

Both proactive and reactive content negotiation are supported by HTTP. The following are some examples.

Proactive content negotiation

In PCN, the client sends a list of preferred response formats in the `Accept` header, and the server determines the final selection:

```
**** REQUEST ****
GET /list HTTP/1.1
Accept: application/vnd.siren+json, application/vnd.hal+json, application/json
...

**** RESPONSE ****
200 OK HTTP/1.1
```

```
Content-Type: application/json
....
```

In this case, the server selected `application/json` as the media type. Clients can provide additional preference information in the form of a "quality" or `q` value, for example:

```
**** REQUEST ****
GET /list HTTP/1.1
Accept: application/vnd.hal+json;q=0.8, application/json;q=0.4
...

**** RESPONSE ****
200 OK HTTP/1.1
Content-Type: application/json
....
```

In this example, the client indicated a higher preference for the `application/vnd.hal+json` media type. However, the server still selected `application/json` as the response format.

For more on quality (`q`) values, check out the HTTP specification.

Reactive content negotiation

In RCN, the server returns a list of supported representations, and the client makes the final selection and resends the request:

```
**** REQUEST ****
GET /search HTTP/1.1

**** RESPONSE ****
HTTP/1.1 300 Multiple Choices
Link: <http://api.example.org/html/search>;rel="alternate html"
Link: <http://api.example.org/api/search>;rel="alternate api"
Location: http://api.example.org/html/search
```

Another RCN response returns the link values as HTML `anchor` tags in the body:

```
**** REQUEST ****
GET /search HTTP/1.1

**** RESPONSE ****
HTTP/1.1 300 Multiple Choices
Content-Type: text/html

<html>
  <title>Multiple Choices</title>
```

```
  <body>
    <h1>Multiple Choices</h1>
    <ul>
      <li><a href="http://api.example.org/html/search>HTML</a></li>
      <li><a href="http://api.example.org/api/search>API</a></li>
    </ul>
  </body>
</html>
```

Discussion

When services support multiple representation formats, the most common approach is to use proactive (server-driven) content negotiation. This is the simplest interaction even though it is possible the client may not receive its preferred format.

RCN has some major drawbacks. Servers are essentially guessing what format to return. It is great when both client and service support the same formats, but there is little to do when there is no direct match. The usual choice is for services to just return whatever *they* prefer. The only real alternative is for services to return `406 Not Acceptable` when they can't match a client's `Accept` preferences.

A drawback for PCN is that clients often don't know what the server supports. Services can solve this problem by supporting the "meta" services pattern that allows services to advertise client preferences (see Recipe 5.5).

I rarely encounter the reactive version of content negotiation in the wild. Because response details are not well specified, I don't recommend using it for M2M use cases unless both parties have worked out all the details ahead of time.

One way to avoid the content negotiation dance is to dedicate URL spaces for each representation format. For example:

- *http://api.example.org/siren/search*
- *http://api.example.org/hal/search*
- *http://api.example.org/collection-json/search*
- *http://api.example.org/html/search*

Of course, the drawback here is that service interfaces need to support many more URL paths.

In my experience the most effective approach is to document supported formats in the API design-time documentation, then advertise client preferences (see Recipe 5.5) and rely on proactive content negotiation at runtime.

See Also

- Recipe 3.1, "Creating Interoperability with Registered Media Types"
- Recipe 4.5, "Negotiating for Profile Support at Runtime"
- Recipe 5.5, "Advertising Support for Client Response Preferences"
- Recipe 6.7, "Using Media Types for Data Queries"

5.7 Publishing Complete Vocabularies for Machine Clients

When creating services that will be accessed by other services (e.g., M2M interfaces), it is important to make sure both parties have a solid "understanding" of each other. That means each message is complete and both parties recognize what is in the message. This is the essence of RESTful implementations. Roy Fielding explains it this way: "A RESTful API (done right) is just a website for clients with a limited vocabulary."

Problem

How can we make sure an API consumer will "understand" service responses adequately? What does it mean to send messages that are "self-describing"? What needs to be included when describing messages?

Solution

The best way to make sure an API consumer "understands" messages from the API provider is to produce a *complete* vocabulary document that lists all the important data and action properties that may appear in a message. Leonard Richardson refers to these values as "magic strings". These are strings that services say "mean something" and that API providers and consumers can use to "tell each other" important information. In this book, I refer to these collections of magic strings as *vocabularies* or sometimes *profiles*.

The `profile` link relation type is defined in RFC 6906 as to "allow clients to learn about additional semantics (constraints, conventions, extensions) that are associated with the resource representation." Applying a profile to a resource does not *change* the meaning of the contents in a message. Instead, profiles provide *additional* information about the message.

A Profile by Any Other Name

I've seen several terms used to identify this definitive list of data and action properties such as "dictionary" and "glossary," even "schema." But I would not use the terms "data dictionary" or "data model." That's because service vocabularies define more

than just the data properties of a service or properties of named objects. They also define all the action details, including links, forms, and query strings.

A service profile contains *all* the identifiers it uses with messages. For example, consider the following HTML FORM:

```
<form name="create-template" action="http://api.example.org/users"
  method="post" enctype="application/x-www-form-urlencoded">
  <input name="id" value="q1w2e3r4" />
  <input name="familyName" value="Mark" />
  <input name="givenName" value="Morkelson" />
  <input name="telephone" value="+1-555-123-4567" />
  <input name="email" value="mork@example.org" />
  <input name="status" value="active" />
  <input type="submit" />
</form>
```

The vocabulary information in this form is: `create-template`, `id`, `familyName`, `given Name`, `telephone`, `email`, and `status`. Both the API service and the API consumer need to recognize and "understand" what these values represent. Notice that I did not include the *values* for those elements (e.g., `id=q1w2e3r4`), just the names. Also, I did not include the media type's structural elements in my list (e.g., `name`, `action`, `method`, `enctype`, etc.).

It is not enough just to identify the vocabulary elements. You also need to offer some explanation of what they represent and how they may appear in a message. For example, the `create-template` semantic value may appear as the `name` value of a `form` element in HTTP messages. This identifies the form clients can use to add new records to the system.

You can also include additional information in your semantic profile documents, such as the source of the string value. For example, `familyName` is defined in the online dictionary Schema.org.

Example

The following are some example vocabulary documents (aka semantic profiles) for the `person` API listed in the book's repo. Here's some sample output from that service:

```
{
  "collection": {
    "version": "1.0",
    "href": "http://localhost:8181",
    "title": "BigCo Activity Records",
    "links": [
      {
        "name": "home", "href": "http://localhost:8181/",
        "rel": "home", "prompt": "Home"
```

```
      },
      {
        "name": "list", "href": "http://localhost:8181/list/",
        "rel": "list", "prompt": "List"
      }
    ],
    "items": [
      {
        "id": "22s3k36pkn4",
        "rel": "person",
        "href": "http://localhost:8181/22s3k36pkn4",
        "data": [
          {"name": "id", "value": "22s3k36pkn4", "prompt": "id"},
          {"name": "givenName", "value": "Mork", "prompt": "givenName"},
          {"name": "familyName", "value": "Mockery", "prompt": "familyName"}
        ],
        "links": [
          {
            "name": "read", "href": "http://localhost:8181/22s3k36pkn4",
            "rel": "read", "prompt": "Read"
          }
        ]
      }
    ],
    "queries": [
      {
        "name": "filter",
        "href": "http://localhost:8181/filter/",
        "rel": "filter",
        "prompt": "Search",
        "data": [
          {"name": "givenName", "value": ""},
          {"name": "familyName", "value": ""}
        ]
      }
    ],
    "template": [
      {"name": "id", "value": "", "prompt": "id"},
      {"name": "givenName", "value": "", "prompt": "givenName"},
      {"name": "familyName", "value": "", "prompt": "familyName"}
    ]
  }
}
```

At minimum, you should list all the semantic identifiers and include a description (see Table 5-2).

Table 5-2. Person API semantic profile

Identifier	Description
`id`	Record identifier for a person record

Identifier	Description
`givenName`	Given name for a person record
`familyName`	Family name for a person record
`person`	Indicates a person record
`home`	Hypermedia control to navigate to the Home view
`list`	Hypermedia control to navigate to the List view
`read`	Hypermedia control to navigate to a single person record
`filter`	Hypermedia control to filter the List view

As additional information, you can also list where—in the message—each element is likely to appear (see Table 5-3).

Table 5-3. Person API semantic profile with element locations

Identifier	Description	Element
`id`	Record identifier for a person record	`name`
`givenName`	Given name for a person record	`name`
`familyName`	Family name for a person record	`name`
`person`	Indicates a person record	`rel`
`collection`	Identifies a collection of records	`rel`
`item`	Identifies a single record	`rel`
`home`	Hypermedia control to navigate to the Home view	`rel`, `name`
`list`	Hypermedia control to navigate to the List view	`rel`, `name`
`read`	Hypermedia control to navigate to a single person record	`rel`, `name`
`filter`	Hypermedia control to filter the List view	`rel`, `name`

It is important to remember that the third column in Table 5-3 ("Element") will likely contain different elements depending on the media type returned. For example, if the service returned `person` representations using the `application/forms+json` media type, the table would look like Table 5-4.

Table 5-4. Person API semantic profile with media type locations

Identifier	Description	Element (Cj)	Element (Fj)
`id`	Record identifier for a person record	`name`	KEY
`givenName`	Given name for a person record	`name`	KEY
`familyName`	Family name for a person record	`name`	KEY
`person`	Indicates a person record	`rel`	KEY
`collection`	Identifies a collection of records	`rel`	`rel`
`item`	Identifies a single record	`rel`	`rel`

Identifier	Description	Element (Cj)	Element (Fj)
`home`	Hypermedia control to navigate to the Home view	`rel`, `name`	`rel`, `name`, `id`
`list`	Hypermedia control to navigate to the List view	`rel`, `name`	`rel`, `name`, `id`
`read`	Hypermedia control to navigate to a single person record	`rel`, `name`	`rel`, `name`, `id`
`filter`	Hypermedia control to filter the List view	`rel`, `name`	`rel`, `name`, `id`

Notice that for the FormsJSON format, some semantic values appear as structural elements in JSON (`id`, `givenName`, `familyName`, and `person`), and others appear as value elements (`collection`, `item`, `list`, `home`, `read`, and `filter`). This mix of structure and value is what makes each media type unique. If your service supports more than one media type, you'll need to supply semantic profile information that properly "maps" semantic elements to message elements:

```
{
  "person": {
    "links": [
      {
        "id": "home",
        "name": "home",
        "href": "http://localhost:8181/",
        "rel": "home",
        "title": "Home",
        "method": "GET",
        "properties": []
      },
      {
        "id": "list",
        "name": "list",
        "href": "http://localhost:8181/list/",
        "rel": "list collection",
        "title": "List",
        "method": "GET",
        "properties": []
      },
      {
        "id": "filter",
        "name": "filter",
        "href": "http://localhost:8181/filter/",
        "rel": "filter collection",
        "title": "Search",
        "method": "GET",
        "properties": [
          {"name": "givenName", "value": ""},
          {"name": "familyName", "value": ""}
        ]
      }
    ],
    "items": [
      {
```

```
        "links": [
          {
            "id": "read_22s3k36pkn4",
            "name": "read",
            "href": "http://localhost:8181/22s3k36pkn4",
            "rel": "read item",
            "title": "Read",
            "method": "GET",
            "properties": []
          }
        ],
        "id": "22s3k36pkn4",
        "givenName": "Mork",
        "familyName": "Mockery"
      }
    ]
  }
}
```

Discussion

Essentially, semantic profiles carry the domain elements (`person`, `list`, `filter`, etc.) of the service. You can think of semantic profiles as a method for documenting the service domain.

It is worth pointing out that these semantic profiles do not explain *how* the `filter` hypermedia control works—just that it might appear in the response. Semantic profiles assume the API consumer application understands that media type is independent of the profile. This separation between understanding message formats and understanding the semantics *within* the message is an important feature of well-designed hypermedia services.

When your service doesn't use a registered media type and instead uses a serialization formation like plain JSON or XML, services generally embed the semantics of the domain into the structure of the message. For example, here's a possible response for the `person` service in XML:

```
<list>
  <home href="..." rel="collection"/>
  <list href="..." rel="collection"/>
  <filter href="..." method="get" rel="collection">
    <givenName></givenName>
    <familyName></familyName>
  </filter>
  <person id="..." rel="item read" href="...">
    <familyName>...</familyName>
    <givenName>...</givenName>
  </person>
</list>
```

The semantic profile mapping would look like Table 5-5.

Table 5-5. Person API semantic profile with both JSON and XML location hints

Identifier	Description	Element (Cj)	Element (Fj)	Element (XML)
`id`	Record identifier for a person record	`name`	KEY	ATTRIBUTE
`givenName`	Given name for a person record	`name`	KEY	ELEMENT
`familyName`	Family name for a person record	`name`	KEY	ELEMENT
`person`	Indicates a person record	`rel`	KEY	ELEMENT
`collection`	Identifies a collection of records	`rel`	`rel`	`rel`
`item`	Identifies a single record	`rel`	`rel`	`rel`
`home`	Hypermedia control to navigate to the Home view	`rel`, `name`	`rel`, `name`, `id`	ELEMENT
`list`	Hypermedia control to navigate to the List view	`rel`, `name`	`rel`, `name`, `id`	ELEMENT
`read`	Hypermedia control to navigate to a single person record	`rel`, `name`	`rel`, `name`, `id`	`rel`
`filter`	Hypermedia control to filter the List view	`rel`, `name`	`rel`, `name`, `id`	ELEMENT

In this case, the domain semantics are so intertwined with the message format that changes in the internal domain elements may break structural changes in the external interface (e.g., renaming elements, adding/removing attributes, etc.). Whenever possible, keep the message details (the external interface) independent of the domain details (internal functionality). When I see cases where the domain semantics appear as structural elements within a response, this sets off alarm bells for me to rethink my message design to make sure I'm not leaking internal details to the external interface. See Recipe 5.4 for more on this topic.

See Also

- Recipe 3.2, "Ensuring Future Compatibility with Structured Media Types"
- Recipe 4.4, "Coding Effective Clients to Understand Vocabulary Profiles"
- Recipe 5.3, "Converting Internal Models to External Messages"
- Recipe 5.4, "Expressing Internal Functions as External Actions"
- Recipe 5.8, "Supporting Shared Vocabularies in Standard Formats"

5.8 Supporting Shared Vocabularies in Standard Formats

When service interfaces adopt the practice of using standard structured media types (SMTs) for exchanging messages, they also need to standardize the manner in which the domain-specific properties and actions (or *semantics*) of the service are expressed at runtime. This is similar to the message format challenge. The solution is to rely on

standardized formats to express meaning. The real work is to express internal data or object models as consistent external semantic models.

Problem

How can we standardize the way in which the meaning of content is transferred between machines? What formats are available for this work, and what is the pattern for expressing internal data and process models as standardized semantic models?

Solution

Just as API consumers rely on standardized protocols (e.g., HTTP) and standardized message formats (e.g., SIREN, Collection+JSON, etc.), they also can benefit from standardized vocabularies. That means listing all the data properties and action names in a standard format that the client understands.

This means there needs to be a single source for all the possible names of things that might appear in your API; for example, any value that might appear in attributes like `id`, `name`, `class`, `rel`, or similar structural elements. A collection of these values can cover a topic or domain (like "User Management" or "Accounting").

There are a couple of options for carrying vocabulary data in a standardized form. One of the oldest known solutions is to use a Resource Description Framework (RDF) format like RDF/XML, Turtle, and JSON-LD. There are a handful of schema-centric RDF languages, including OWL and RDF Schema, too. The JSON-LD format is the one I see more often when APIs use an RDF-based format.

It is also possible to use schema documents: XSD for XML messages and JSON Schema for JSON-based messages. While these formats are well established with lots of tooling, it is worth noting that common uses of XSD and JSON Schema are limited to defining a vocabulary's data property names and usually don't include action names (e.g., names of forms, link relation identifiers, etc.).

The format I use to document vocabularies is the Application-Level Profile Semantics (ALPS) format. There are serializations of ALPS for XML, JSON, and even YAML.

Example

The following is an RDF version of a `person` expressed in the FOAF (friend of a friend) vocabulary:

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

<#JW>
    a foaf:Person ;
```

```
    foaf:name "James Wales" ;
    foaf:mbox <mailto:jwales@bomis.com> ;
    foaf:homepage <http://www.jameswales.com> ;
    foaf:nick "Jimbo" ;
    foaf:depiction <http://www.jameswales.com/aus_img_small.jpg> ;
    foaf:interest <http://www.wikimedia.org> ;
    foaf:knows [
        a foaf:Person ;
        foaf:name "Angela Beesley"
    ] .

<http://www.wikimedia.org>
    rdfs:label "Wikimedia"
```

Figure 5-3 is a diagram of that same RDF document using the ":isSemantic" visualization service.

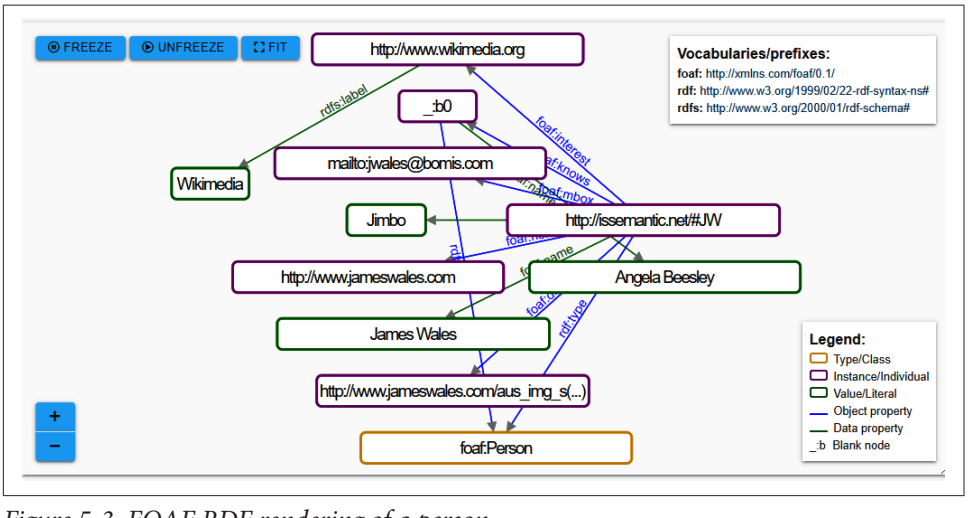

Figure 5-3. FOAF RDF rendering of a person

For another approach, the following is a portion of the ALPS document that details the `person` service API we've been referring to throughout this recipe. See Recipe 3.4 for details on the ALPS format:

```
{
  "$schema": "https://alps-io.github.io/schemas/alps.json",
  "alps":
  {
    "version": "1.0",
    "title": "Person Service API",

    "descriptor" : [
     ...
     {"id": "home", "type": "semantic",
        "title":"Home (starting point) of the person service",
```

```
            "tag":"taxonomy",
            "descriptor": [
             {"href": "#goHome"},
             {"href": "#goList"}
            ],
            "doc" : {"value" : "Person API starting point"}
         },
         {"id": "collection", "type": "semantic",
            "title":"List of person resources",
            "tag":"taxonomy",
            "descriptor": [
             {"href": "#person"},
             {"href": "#goHome"},
             {"href": "#goList"},
             {"href": "#goFilter"},
             {"href": "#goItem"},
             {"href": "#doCreate"}
            ],
            "doc" : {"value" : "List of person resources"}
         },
         {"id": "item", "type": "semantic",
            "title":"Single person resource",
            "tag":"taxonomy",
            "descriptor": [
              {"href": "#person"},
              {"href": "#goHome"},
              {"href": "#goList"},
              {"href": "#goFilter"},
              {"href": "#goItem"},
              {"href": "#doUpdate"},
              {"href": "#doStatus"},
              {"href": "#doRemove"}
            ],
            "doc" : {"value" : "A single person resource"}
          },
          ...
       ]
    }
}
```

Figure 5-4 is a graphical representation of the full `person` profile using the "app-state-diagram (ASD)" tool.

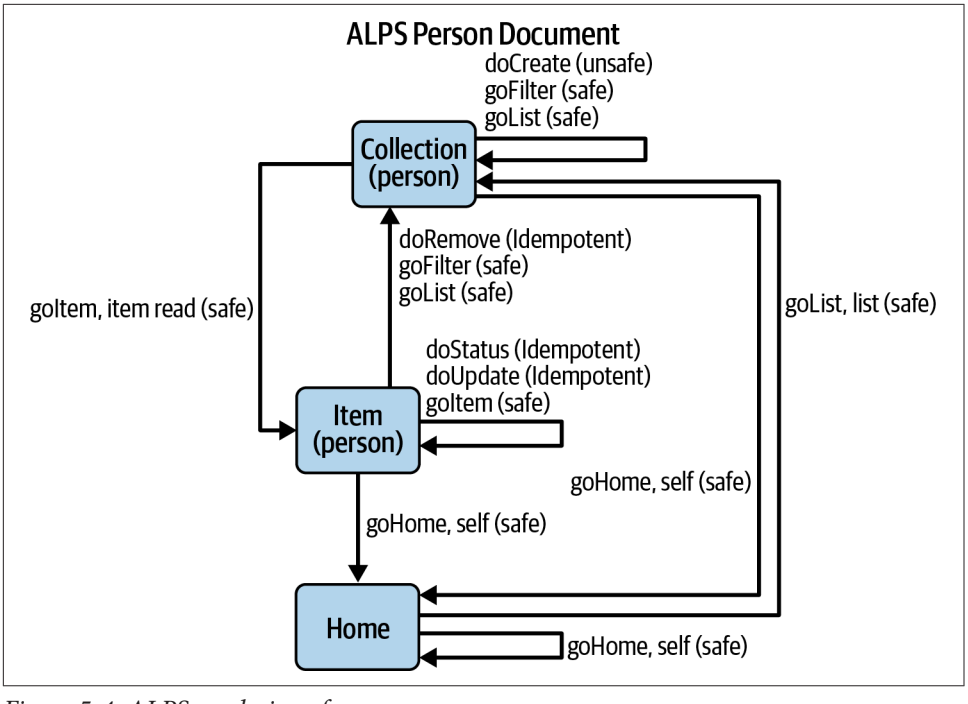

Figure 5-4. ALPS rendering of a person

A key difference between RDF and ALPS is that RDF focuses on the relationship between *data* items, and ALPS focuses on the relationship between *action* items. One vocabulary format that bridges that gap pretty well is the Hydra ontology.

Discussion

Service APIs can use semantic profile URLs as *identifiers* without needing to fully understand what is contained in a semantic profile document. For example, by including a profile URL in the header collection for responses, the service API is telling API consumers what vocabulary that service API "speaks":

```
**** REQUEST ****
GET /shopping/ HTTP/1.1
Host: api.example.org
Accept: application/vnd.collection+json

**** RESPONSE ****
HTTP/1.1 200 OK
Content-Type: application/vnd.collection+json
Link: <http://docs.alps.io/shopping-v2.json>; rel="profile"
```

In this example, the API service is "advertising" that it supports version 2 of the `shopping` profile. For more on how API providers and consumers can negotiate for profile support, see Recipe 4.5.

Profiles that detail the semantics of an interface do not fill the same role as schema documents. Schemas are used to describe objects or document messages. Semantic profiles are used to describe the way these objects (or documents) interact with each other. Publishing a profile document helps developers (and machines) understand the problem space—the possibilities within an API. Depending on the message formats you are using, you may also want to publish schema documents to help people and machines understand the types of objects and documents that will be passed around within the problem space. See Recipe 4.7 for more on how to take advantage of schema documents for APIs.

It is also worth mentioning that semantic profile documents like ALPS are not the same thing as API definition documents such as OpenAPI, AsyncAPI, Protobuf, SOAP, etc. Semantic profiles provide a complete vocabulary of properties and actions for a collection of APIs (those that support that profile). API definition documents provide details on just how a single instance of a service API is implemented—the one that lives at *http://api.example.org/shopping*, for example.

An important part of proper support for semantic profiles is mapping the profile elements to media type elements in the messages that are exchanged. See Recipe 5.7 for more on how to document profile/media type mapping.

It is a good idea to include semantic profile documents in all your service API documentation. This helps API consumers by providing a detailed list of vocabulary elements and can provide important hints to API consumers on how to navigate the "problem space" your service supports. See Recipe 5.10 for a more comprehensive list of assets services APIs should provide to consumers.

See Also

- Recipe 3.4, "Describing Problem Spaces with Semantic Profiles"
- Recipe 4.5, "Negotiating for Profile Support at Runtime"
- Recipe 4.7, "Using Schema Documents as a Source of Message Metadata"
- Recipe 5.6, "Supporting HTTP Content Negotiation"
- Recipe 5.7, "Publishing Complete Vocabularies for Machine Clients"
- Recipe 5.10, "Publishing API Metadata"
- Recipe 7.1, "Designing Workflow-Compliant Services"

5.9 Publishing Service Definition Documents

Most service APIs should publish a Service Definition Document (SDD) that explicitly describes the shared interface that both API providers and API consumers need to understand in order to successfully interact. Today, the most common way to do this is to publish your API's definition in one of several standard formats.

Problem

What are the common SDD formats? How can you easily share SDDs with API consumers? What URLs and/or link relation values should be used when sharing SDDs? Along with direct links in the API, should links to SDDs be listed in other locations? Appear in other documents?

Solution

For most APIs, there is a fixed set of URLs and message bodies that are shared at runtime. These fixed "endpoints" and "objects" can be documented in a number of different formats based on the style of the service interface implementation.

The most common API styles (and their related SDDs) are:

HTTP CRUD
: OpenAPI, Web Application Description Language (WADL)

Event-driven
: AsyncAPI, CloudEvents

Remote procedure
: Protocol buffers, XML-RPC, JSON-RPC

Remote data query
: Schema Definition Language (for GraphQL), OData

Remote messaging
: Web Service Definition Language (WSDL)

Hypermedia
: Application-Level Profile Semantics (ALPS)

Typically, SDDs are designed to streamline the work of generating human-readable documentation and/or generating API consumer code. Sometimes SDDs are actually generated *from* code. For example, both OpenAPI and WSDL can be generated from existing source code. As of this writing, many of the formats are authored "by hand" as part of the service interface design process.

However the document is created, it is a good idea to publish that document at an easily found location. This will help API client application developers quickly get up

to speed on consuming the service API successfully. The best way to do this is to return a link to the SDD in HTTP responses either as a `link` header, within the response body, or both. RFC 8631 defines a link relation value of `service-desc` to use when identifying links that point to SDD documents. It is also a good idea to return a pointer to the SDD in a `link` header in response to an HTTP `OPTIONS` request. See the examples in this recipe for details.

It is a good idea to include a link to the SDD document in your API metadata document (see Recipe 5.10).

Example

There are three good places to advertise your API's SDD location: the HTTP `link` response header, as part of your interface's service metadata, and as part of the HTTP `OPTIONS` response.

SDDs in HTTP responses

You can return a pointer to the API's SDD as a `link` header in any API response:

```
HTTP/1.1 200 OK
Content-Type: application/vnd.hal+json
Link: <http://api.example.org/service-desc>; rel=service-desc
...
```

You can also return this information in the body of the response:

```
HTTP/1.1 200 OK
Content-Type: application/vnd.collection+json
...

{ "collection" : {
  "links" : [
    {"rel": "service-desc", "href": "http://api.example.org/service-desc"}
  ]
  ...
}}
```

When the format supports it, you can return the SDD location in both the header and body of a response.

SDDs in service metadata responses

You can also return the location of the SDD in your API's metadata document (see Recipe 5.10). The format I recommend for service metadata is *APIs.json*, and that specification has reserved keywords for the most common SDD formats.

Here's a snippet of an *APIs.json* document that cites support for an OpenAPI SDD:

```
{
  "name": "Example API",
  "type": "Index",
  "created": "2014-04-07",
  "modified": "2020-09-03",
  "url": "http://example.com/apis.json",
  "specificationVersion": "0.14",
  "apis": [
  {
    "name": "Example API",
    "humanURL": "http://example.com",
    "baseURL": "http://api.example.com",
    "properties": [
      {"type": "OpenAPI", "url": "http://example.com/openapi.json"}
    ]
  }
  ]
  ...
}
```

If you are using an SDD format that is not currently covered by the *APIs.json* specification, you can add your own format as an extension, too.

SDDs in HTTP OPTIONS responses

You can include the SDD in a `link` header and/or link the body of the `OPTIONS` response too:

```
**** REQUEST ****
OPTIONS / HTTP/1.1
Host: api.example.org
...

**** RESPONSE ****
HTTP/1.1 200 OK
Content-Type: application/vnd.siren+json
Content-Length: XX
Cache-Control: max-age=604800
Allow: GET PUT DELETE HEAD OPTIONS
Accept: application/zip application/gzip
Accept-Charset: utf-8
Accept-Encoding: compress
Accept-Language: en
Link: <https://api.example.org/service-desc>; rel="service-desc"
```

See Recipe 5.5 for more on using the `OPTIONS` method.

Discussion

It is a good idea to store the SDD document in the root folder of the service interface when possible. It can also be helpful to name the document *service-desc*. If you do these two things consistently, your API consumers can "assume" the location of the SDD, which will make it easier on API developers.

The formats collection is not an exhaustive list. The reader may be using other SDD formats, and some large organizations even have their own in-house SDDs that they use. I've included the SDDs that, in my experience, have appeared most often "in the wild."

In some cases, the service supports multiple interface styles (e.g., HTTP CRUD and GraphQL). In that case, you can return multiple SDD documents in the header or body of your responses:

```
HTTP/1.1 200 OK
Content-Type: application/vnd.hal+json
Link: <http://api.example.org/openapi/service-desc>; rel="service-desc"; \
type="application/openapi+json"
Link: <http://api.example.org/graphql/service-desc>; rel=service-desc"; \
type="application/sdl+json"
...
```

You can use the `type` property of the `Link` header to indicate the expected format of the response (see the preceding example). The challenge here is that most SDD formats do not have their own media type identifier string registered with the IANA. For my own work, I've adopted a set of media type identifiers and include them in my API documentation. This is less than ideal but works as expected. Hopefully, by the time you are reading this book, more SDD formats will have acquired their own registered media type identifiers.

Note that there isn't an agreed standard SDD format for hypermedia-style APIs. For now, I recommend including an ALPS document as part of the API's `profile` (see Recipe 5.7).

See Also

- Recipe 3.4, "Describing Problem Spaces with Semantic Profiles"
- Recipe 4.4, "Coding Effective Clients to Understand Vocabulary Profiles"
- Recipe 4.6, "Managing Representation Formats at Runtime"
- Recipe 5.7, "Publishing Complete Vocabularies for Machine Clients"
- Recipe 5.10, "Publishing API Metadata"

5.10 Publishing API Metadata

As the number of application interfaces on the open web grows, there is a need for a common format and common vocabulary for describing metadata *about* the API. This is especially important when service providers want to communicate important details about the API without the need for physical meetings, interactions, and/or bespoke description and documentation efforts. Luckily, there is a strong contender for carrying this information: the *APIs.json* specification.

Problem

How can we represent important metadata about a service interface (supported formats, vocabularies, documentation, security, etc.) in a standardized way that allows API consumers to easily find and understand the metalevel details of the interface?

Solution

A very dependable way to catalog and publish API metadata is to use the *APIs.json* open specification. It has a vocabulary and structural layout that covers most of the important metadata elements for APIs (basic definitions, properties of the API, important URLs, key contacts and maintainers, etc.). It is also extensible so API providers can, when needed, expand the *APIs.json* document (in a backward-compatible way) to fit their needs.

APIs.json, APIs.yaml, and APIS.txt

APIs.json documents can be formatted as JSON, YAML, or TXT files. You can learn more about *APIs.json* by visiting the specification site. You can also find a publicly hosted version of a search engine for *APIs.json* documents at *http://apis.io*.

The *APIs.json* specification defines several key sections of the document:

Root
: The name, description, URL, and other basic information about the API

APIs
: The list of APIs identified in the document

Tools
: The list of open source tooling available as part of operations

Specifications
: The list of open specifications adopted as part of operations

Resources
: The list of resources available for API operations

Common
: A list of common properties for use across all APIs and tools

Include
: Other *APIs.json* documents to include in this document

Maintainers
: A collection of persons and/or organizations maintaining this API

Providers can document the API's metadata in a single document (named *apis.json*) available at the root folder of the service (e.g., *http://api.example.org/apis.json)*. APIs an also provide a link relation value that can be included in an HTTP header and/or within the body of an HTTP response. The `service-meta` relation value (defined in RFC 8631) is the recommended value to use when pointing to the API's *APIs.json* document.

It's a good idea to support a link on the "Home" resource of your service interface that points to the `service-meta` resource (see Figure 5-5).

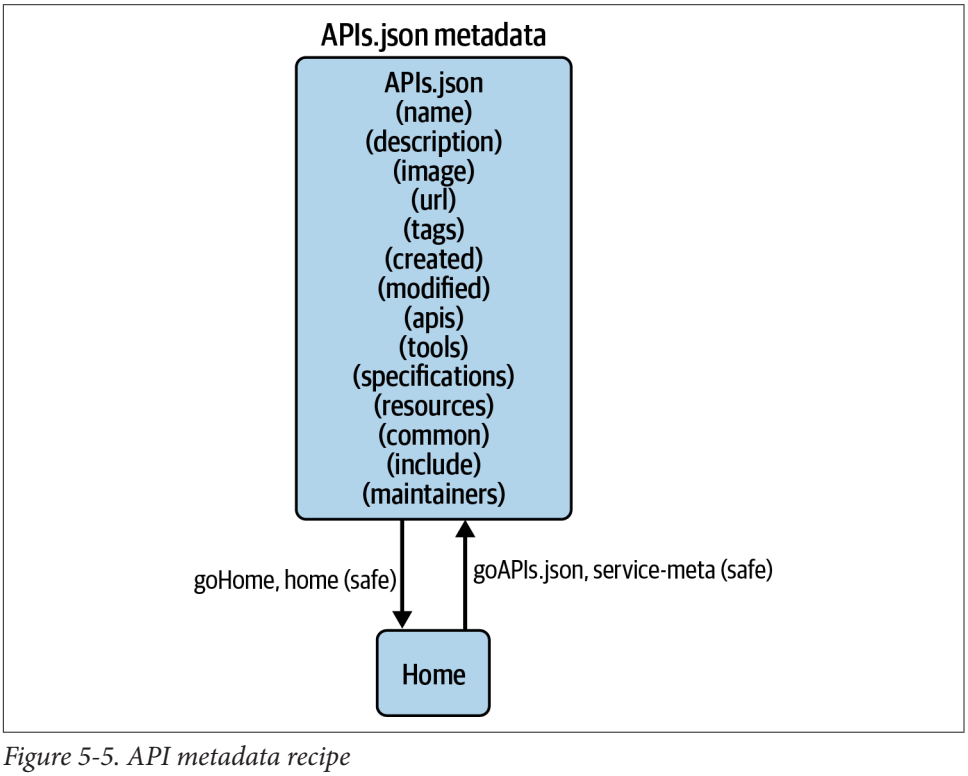

Figure 5-5. API metadata recipe

For details on the `service-meta` *APIs.json* resource, check out the online collection of ALPS documents associated with this book.

Example

Here is a short example of an *APIs.json* document:

```
HTTP/1.1 200 OK
Content-Type: application/apis+json
...

{
  "name": "Example API",
  "type": "Index",
  "description": "This is an example APIs.json file.",
  "image": "https://api.example.org/logo.jpg",
  "tags": ["Application Programming Interface","API"],
  "created": "2014-04-07",
  "modified": "2020-09-03",
  "url": "http://example.com/apis.json",
  "specificationVersion": "0.14",
  "apis": [
    {
      "name": "Example API",
      "description": "This provides details about a specific API.",
      "humanURL": "http://example.com",
      "baseURL": "http://api.example.com",
      "tags": ["API","Application Programming Interface"],
      "properties": [
         {"type": "Documentation","url": "https://example.com/documentation"},
         {"type": "OpenAPI","url": "http://example.com/openapi.json"}
      ],
      "contact": [{"FN": "APIs.json","email": "info@apisjson.org"}]
    }
  ],
  "specifications": [
    {"name": "OpenAPI", "url": "https://openapis.org"},
    {"name": "JSON Schema","url": "https://json-schema.org/"}
  ],
  "common": [
    {"type": "Signup","url": "https://example.com/signup"},
    {"type": "Authentication","url": "http://example.com/authentication"},
    {"type": "Login","url": "https://example.com/login"}
  ],
  "maintainers": [{"FN": "Mark Morkelson","email": "mork@example.org"}]
}
```

Services can advertise availability of the *APIs.json* document in HTTP responses:

```
HTTP/1.1 200 OK
Content-Type: HTML
Link: <http://api.example.org/apis.json>; rel=service-meta
...

<html>
  <head>
    <link rel="service-meta" href="http://api.example.org/apis.json" />
  </head>
  <body>
  ...
  </body>
</html>
```

API client applications can also automatically "look" for *APIs.json* documents by calling to the default location of the file as stated in the specification:

```
GET /apis.json HTTP/1.1
Accept: application/apis+json, application/yaml, text/plain
Host: http://api.example.org
```

Discussion

It is important to note that the *APIs.json* specification is meant to carry related information about a service interface, but it is not meant to *describe* or *define* that interface in detail. Other formats, such as OpenAPI, AsyncAPI, Protobuf, etc., were designed for that purpose. See Recipe 5.9 for details on service definition formats and how to share them with API consumers.

As of this writing, the *APIs.json* specification is not yet finalized. However, I've found the format (and the related search engine) stable and useful, and I encourage readers to keep an eye on this specification.

One of the reasons to adopt the *APIs.json* format for service metadata is that there is a related open source project that provides a search engine that uses the *APIs.json* format as input. You can find the current edition of the *APIs.json* search engine project online. As of this writing, you can fork or download this project and start up your own API search engine to host *APIs.json* documents.

An added feature of the *APIs.json* search engine project is support for its own API dedicated to automating the uploading and validation of *APIs.json* documents for that search engine. This would make it possible to create an automated process for registering APIs with target API search engines.

The *APIs.json* specification contains more than 50 reserved words. Most of them are known as "Properties Elements" and are used for the `type` values for entries in the `common` section of an *APIs.json* document. Here are just a few samples:

- `Signup`
- `Login`
- `TermsOfService`
- `InterfaceLicense`
- `PrivacyPolicy`
- `Security`
- `StatusPage`
- `Pricing`
- `Rate Limits`

You can also add your own names to this list of "Properties Elements" as extensions to the specification.

Another specification that could be used to carry API metadata is the JSON Home specification. However, this specification proposal has not been updated in several years, and I have not seen examples of JSON Home "in the wild" as often as I have seen the *APIs.json* specification discussed here.

See Also

- Recipe 3.3, "Sharing Domain Specifics via Published Vocabularies"
- Recipe 3.4, "Describing Problem Spaces with Semantic Profiles"
- Recipe 4.4, "Coding Effective Clients to Understand Vocabulary Profiles"
- Recipe 5.7, "Publishing Complete Vocabularies for Machine Clients"
- Recipe 5.9, "Publishing Service Definition Documents"
- Recipe 7.1, "Designing Workflow-Compliant Services"

5.11 Supporting Service Health Monitoring

One of the advantages of using microservices on the web is that you don't have to program all the functionality yourself; you can rely on the creativity of others. However, adding dependent services to your own increases the likelihood of failure on a distributed network. A good way to monitor the status of dependent services is to make regular "health checks" to other services to make sure they are up and running properly.

Problem

How can we consistently and regularly monitor the status (or health) of services on the web to make sure they are up and running and performing as expected? What are the common values we should monitor to assess the health of a service? What is the standard format for passing those status values? How do we advertise support for health monitoring, and how do we share that status data?

Solution

Health and status monitoring of services running on the web is a common challenge. The good news is that there has been lots of good work in the area of standardizing this process over the last few years. There is currently an expired draft document called "Health Check Response Format for HTTP APIs". This specification was first published in 2018, and I've been using it as a guide for implementing health and status checking for quite some time.

The Health Check specification defines several properties, including these basics:

`status`
: Indicates the service status (`pass`, `fail`, `warn`)

`version`
: Public version of the service

`releaseId`
: Service release/version

`notes`
: Array of notes relevant to current state of health

`output`
: Raw error output, in case of `fail` or `warn` states

`checks`
: An object providing detailed health statuses of additional downstream systems

`links`
: An object containing link relations and URIs for external links that *may* contain more information about the health of the service

`serviceId`
: A unique identifier of the service

`description`
: A human-friendly description of the service

The `checks` object represents a collection of related "downstream" services. It is a bit of a "look ahead" to the health of other dependent services. This object has several more properties defined, and you can review the specification for details.

Here are some additional things to keep in mind when implementing the Health Check specification:

Health resource
: Services should expose an "endpoint" that can be used to request a health status document. The spec suggests (but does not require) that services use `/health` as the URL for this.

Health link relation value
: Although not mentioned in the specification document, I also use the `healthcheck` link relation value to identify the URL that will return the health status information.

Health media type
: When returning the health status information, services should use the `application/health+json` media type identifier and return a valid (based on the specification) health status document.

Caching
: Services should include an HTTP caching directive with the response (`max-age`, `ETag`, etc.) to reduce the impact of frequent requests (and avoid a denial of service attack vector). Of course, clients should honor the caching directives, too.

See the examples for details.

It is a good idea to support a link on the "Home" resource of your service interface that points to the `health` resource (see Figure 5-6).

For details on the `health` Health Checks resource, check out the online collection of ALPS documents associated with this book.

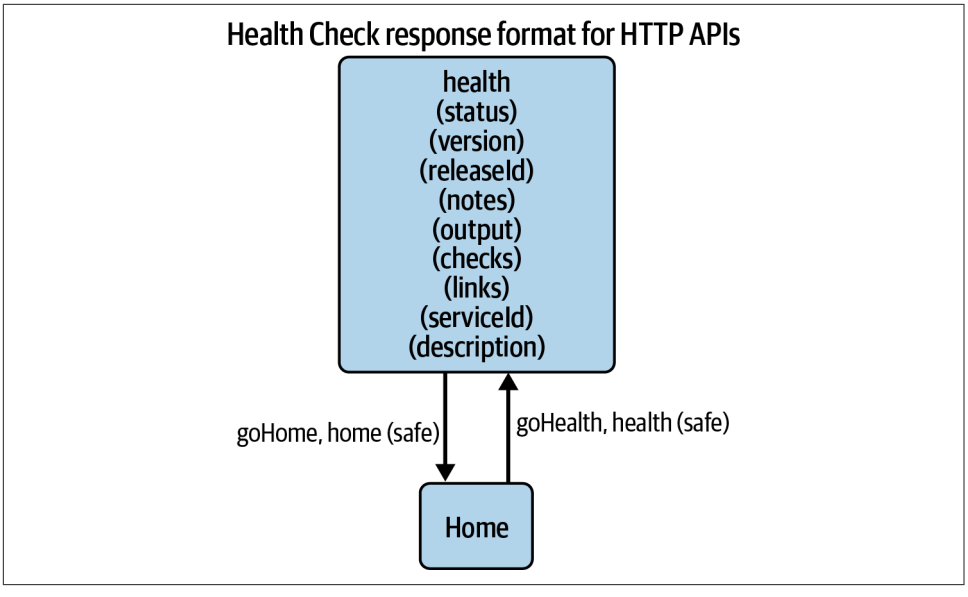

Figure 5-6. Service health recipe

Example

A typical `application/health+json` response looks like this:

```
HTTP/1.1 200 OK
Content-Type: application/health+json
Cache-Control: max-age=3600
ETag: "w\i8u7y6t5r4e3w2"
...

{
  "status": "pass",
  "version": "1",
  "releaseId": "1.2.2",
  "notes": [""],
  "output": "",
  "serviceId": "f03e522f-1f44-4062-9b55-9587f91c9c41",
  "description": "health of authz service",
  "checks": {
    "cassandra:responseTime": [
      {
        "componentId": "dfd6cf2b-1b6e-4412-a0b8-f6f7797a60d2",
        "componentType": "datastore",
        "observedValue": 250,
        "observedUnit": "ms",
        "status": "pass",
        "affectedEndpoints" : [
          "/users/{userId}",
          "/customers/{customerId}/status",
```

```
            "/shopping/{anything}"
          ],
          "time": "2018-01-17T03:36:48Z",
          "output": ""
        }
      ],
      ...
    }
  }
```

Note the use of `cache-control` and `ETag` headers to instruct client applications to retain local copies of the document to reduce traffic loads to the service.

Health check information in OPTIONS responses

You can "advertise" support for the health check pattern by including a `link` relation value in responses. I do this for all my `OPTIONS` responses:

```
**** REQUEST ****
OPTIONS / HTTP/1.1
Accept: application/vnd.collection+json, application/html, application/json
...

**** RESPONSE ****
HTTP/1.1 200 OK
Content-Type: application/vnd.collection+json
Link: <http://api.example.org/health>; rel="health-check"
...
```

Health check information in service-meta responses

I also include support for health checks in my service metadata document (see Recipe 5.10):

```
HTTP/1.1 200 OK
Content-Type: application/apis+json
...

{
  "name": "Example API",
  "type": "Index",
  "description": "This is an example APIs.json file.",
  "image": "https://api.example.org/logo.jpg",
  "tags": ["Application Programming Interface","API"],
  "created": "2014-04-07",
  "modified": "2020-09-03",
  "url": "http://example.com/apis.json",
  "specificationVersion": "0.14",
  "apis": [
    {
      "name": "Example API",
      "description": "This provides details about a specific API.",
      "humanURL": "http://example.com",
```

```
        "baseURL": "http://api.example.com",
        "tags": ["API","Application Programming Interface"],
        "properties": [
           {"type": "Documentation","url": "https://example.com/documentation"},
           {"type": "OpenAPI","url": "http://example.com/openapi.json"}
           {"type": "Health","url": "http://example.com/health.json"}
        ],
        "contact": [{"FN": "APIs.json","email": "info@apisjson.org"}]
      }
    ],
    ...
}
```

Note that the "health" type in *APIs.json* is an extension value not included in the *APIs.json* specification. See Recipe 5.10 for details.

Discussion

In most cases, health checks are helpful when they stick to the basics. Reporting `pass`, `fail`, and `warn` is usually more than enough for service consumers. It is not a good idea to try to use health checks as a debugging or diagnostic tool. The health check reflects the state of the *interface*, not the service behind that interface.

This recipe is based on the draft-06 version of the "Health Check Response Format for HTTP APIs" document. By the time you read this book, there may be new drafts or the final approved RFC may be published. Be sure to keep an eye on this specification document to stay informed on changes.

Some readers might want to set up a "callback" health endpoint that allows others to subscribe to regular responses from your service. This is not recommended. As the number of users of your service grows, it is possible that you'll get hundreds, possibly thousands of callback requests to handle. This can easily overwhelm your service. For that reason, stick to offering an HTTP `GET` endpoint that has a `cache-control` value that reduces the impact of lots of health requests. It would be the ultimate irony that health checks cause the service to be unable to meet your service-level agreements!

The specification points out that responses for health checks are "dynamic"; the contents of the response may be customized based on the calling context. For example, anonymous requests might return only the `status` value (`pass`, `fail`, `warn`), but authorized requests (e.g., admin users) might see detailed information on the status of the service.

The specification does not provide lots of details on extending the `application/health+json` responses. If you want to include additional information, not covered by the defined properties, be sure to do it in a backward-compatible manner, and provide links to human-readable documentations explaining your extension.

See Also

- Recipe 3.8, "Designing for Repeatable Actions"
- Recipe 3.9, "Designing for Reversible Actions"
- Recipe 5.12, "Standardizing Error Reporting"
- Recipe 7.16, "Short-Term Fixes with Automatic Retries"
- Recipe 7.17, "Supporting Local Undo or Rollback"

5.12 Standardizing Error Reporting

All service interfaces encounter errors. The challenge is handling them properly. The first step is to *report* the initial error in a way that is consistent and usable for API consuming applications. That's what RFC 7807 does.

Problem

What is the best way for web-based services to report runtime errors? What information should be returned to the caller when errors occur? What is the best format for returning that information? What can services do to make sure error reporting does not result in unexpected halting or "crashing" of the API client application that experiences the error?

Solution

Error reporting is a critical part of implementing stable, usable service interfaces. It is important to handle it in a way that, whenever possible, allows both the API provider and consumer to resolve any errors and continue to function as designed. The key to this is reporting errors in a standardized way: a way that API consumers will recognize ("oh, that's an error") and that gives them the opportunity to resolve the error.

An excellent way to report errors to API consumers is to use the "Problem Details" media type defined in RFC 7807. When you use this media type for reporting errors, clients are more likely to recognize the error and can be more prepared for resolving the problem described in the message (see the example in this recipe).

Errors Are Built In

In some cases, message formats have error reporting as part of the design. For example, the CollectionJSON format includes an `error` object. When the format accounts for error reporting directly in the message, it is better to report errors within that format instead of using an RFC 7807 message to report the same information.

The key to success is to treat errors as an "alternate response" instead of a failed request. In other words, always include error reporting as a *feature* of your service interface, not a failure condition.

Example

The Problem Details specification (RFC 7807) defines a small set of elements:

`type`
: A URI reference (RFC 3986) that identifies the problem type

`title`
: A short, human-readable summary of the problem type

`status`
: The HTTP status code generated by the service (this is a number, not a string)

`detail`
: A human-readable explanation specific to this occurrence of the problem

`instance`
: A URI reference that identifies the specific occurrence of the problem (optional)

The specification defines a default value for `type`: `"about:blank"`. If a URI appears in the `type` property, the specification encourages this URI to point to a human-readable message for that problem type.

Here's a typical problem details message:

```
HTTP/1.1 403 Forbidden
Content-Type: application/problem+json
Content-Language: en

{
  "type": "https://example.com/probs/out-of-credit",
  "title": "You do not have enough credit.",
  "detail": "Your current balance is 30, but that costs 50.",
  "instance": "/account/12345/msgs/abc",
  "status": 403
}
```

It is possible to extend problem detail records using your own properties:

```
HTTP/1.1 403 Forbidden
Content-Type: application/problem+json
Content-Language: en

{
  "type": "https://example.com/probs/out-of-credit",
  "title": "You do not have enough credit.",
  "detail": "Your current balance is 30, but that costs 50.",
  "instance": "/account/12345/msgs/abc",
  "status": 403,
  "balance": 30,
  "accounts": ["/account/12345", "/account/67890"]
}
```

Note that the URL in the `type` element should point to a document that defines the use and meaning of the `out-of-credit` problem. This documentation should also include definitions of the `balance` and `accounts` elements. In this way, the `type` property points to a semantic profile for this problem representation. See Recipe 5.7 for details on semantic profiles.

The `type`, `title`, and `status` of the problem detail message are "fixed"—always the same, no matter which API or service returns the message. However, the `detail` and `instance` properties are specific to the current case. Here is an illustration:

```
HTTP/1.1 403 Forbidden
Content-Type: application/problem+json
Content-Language: en

{
  "type": "https://example.com/probs/out-of-credit",
  "title": "You do not have enough credit.",
  "detail": "Your current balance is 20, but that costs 50.",
  "instance": "/account/12345/msgs/q1w2e3",
  "status": 403,
  "balance": 20,
  "accounts": ["/account/r4er3w2", "/account/y6t5r4"]
}
```

Now compare this problem details response to the previous one. In both cases, the `type`, `title`, and `status` values are the same. But the `details`, `instance`, `balance`, and `accounts` values are different.

The RFC 7807 specification registers both `application/problem+json` and `application/problem+xml` media type identifiers. However, it does not supply any examples for an XML version of the problem details message. The following response:

```
HTTP/1.1 403 Forbidden
Content-Type: application/problem+xml
Content-Language: en
```

```
<problem-details>
  <type>https://example.com/probs/out-of-credit"</type>
  <title>You do not have enough credit.</title>
  <detail>Your current balance is 30, but that costs 50.</detail>
  <instance>/account/12345/msgs/abc</instance>
  <status>403</status>
<problem-details>
```

As of this writing, RFC 7807 is undergoing an update at the IETF standards committee. While this update is expected to result in a nonbreaking change, readers should keep a close eye on this specification and be prepared for any changes that may occur in the future.

Discussion

The specification makes a point to say that problem details messages should not be used when a simple HTTP 4xx or 5xx status report would suffice. For example, if a user attempts to update an existing record and does not have rights to do so, returning a 403 response to a `PUT` would likely suffice. However, if the user sent an invalid body for a `PUT` message, it might be helpful to return a problem details response that describes the problem and offers a way to fix the error before resubmitting.

Services should not use this format to return "debugging" information to the client application. The format is meant to share details about the *interface*, not about the underlying service behind the API.

When creating a new problem details `type`, you should document three things:

- A `type` URI (e.g., *http://api.example.org/problems/insufficient-funds)*
- A `title` (e.g., "Your account has insufficient funds for this transaction.")
- A `status` code that is appropriate for this problem (e.g., 403)

Whenever possible, it is a good idea to limit the number of new problem details `types` you define and create ones that are general enough to be reusable. You can also customize the problem representations using the `detail` (text) and `instance` (URL) elements of the message.

It is also possible to indicate the use of a `Retry-After` header with a problem details response. This makes it possible to indicate retry details to API consumers when needed.

See Also

- Recipe 3.8, "Designing for Repeatable Actions"
- Recipe 3.9, "Designing for Reversible Actions"
- Recipe 5.11, "Supporting Service Health Monitoring"
- Recipe 7.17, "Supporting Local Undo or Rollback"
- Recipe 7.16, "Short-Term Fixes with Automatic Retries"

5.13 Improving Service Discoverability with a Runtime Service Registry

A key goal for all API interfaces is the ability to easily discover services you want to use and consistently (re)use those services across a wide range of products and applications. This recipe focuses on the work of making sure your provider service is *discoverable* and *reusable* at runtime (not design or build time) by registering with a shared Runtime Service Registry (RSR).

Problem

What should service providers do to make sure their service is easily and consistently registered with target RSRs? What internal functions should each service interface support in order to be discovered? How can services report their runtime health statistics to the RSRs? What service metadata should services supply to improve the findability of the service for both machines and people?

Solution

Maintaining an RSR can improve the findability and reusability of running services on the web. When a service is deployed on the web (and starts up), that service should "publish" itself to one or more RSRs. This essentially tells the registry where the service is running, what kind of functionality it has, and other supportive information, such as the media types (Recipe 3.1), shared vocabularies (Recipe 3.3), interface definitions (Recipe 5.9), and health status (Recipe 5.11). You can think of RSRs as a kind of Domain Name System (DNS) for services—a reliable place to find and connect to other services.

This recipe is focused on supporting M2M runtime discovery and use of an existing API. There are other options for supporting human-driven design- or build-time discovery such as API portals, catalogs, etc. This recipe (along with Recipe 5.10) can be adapted to help support design- and build-time API discovery, too.

For a service to be easily discovered and (re)used on the web at runtime, it should support the following:

Register on startup
: When the service is deployed, it should automatically self-register with one or more RSRs. This makes sure other services can find it when it is needed.

Periodic health reports
: On a regular basis (sometimes this interval is set by the RSR), the service should "ping back" to the RSR (using a URL supplied at registration) to confirm that the service is up and running. Optionally, it may report some usage statistics, like how many times it has been called in the last reporting period, number of reported errors (400/500), and response times.

Unregister on shutdown
: When the service goes offline (via a controlled shutdown or fatal error), it should automatically "unregister" with any associated RSRs. This ensures that other services will not try to connect to it when it is not available.

Note that all three of these actions are internal to the service and there is no need to expose this functionality as part of the usable service API interface.

Example

Creating a service that supports RSRs means enabling your service to register at startup, send health pings to the RSR, and unregister at shutdown. Here is a simple example of registering with a preconfigured RSR at startup using a package called `discovery`:

```
var srsResponse = null;
var srsRegister({Url:"...","name":"...", .....});

// register this service w/ defaults
discovery.register(srsRegister, function(data, response) {
  srsResponse = JSON.parse(data);
  initiateKeepAlive(srsResponse.href, srsResponse.milliseconds);
  http.createServer(uuidGenerator).listen(port);
  console.info('uuid-generator running on port '+port+'.');
});
```

Upon registration, a collection of service metadata should be supplied to the RSR, such as:

`serviceURL`
: URL of the service you are registering

`serviceName`
: Text name of the registered service

`semanticProfile`
: Space-separated list of profile URIs (see Recipe 3.4)

`mediaType`
: Space-separated list of `mediaType` identifiers (see Recipe 3.1)

`apiDefintions`
: Collection of annotated links pointing to API definition documents (see Recipe 5.9)

`tags`
: Space-separated list of searchable keywords (e.g., `"gdpr banking loans"`)

While the service is up and running, it should periodically ping the RSR to confirm that the service is still up and running. The example shows this using the `initiate KeepAlive(…)` method. This method may also send additional health information (or an `href` that points to that data; see Recipe 5.11 for more on this detail).

Whenever the service is taken offline, either in a controlled form or due to a fatal error, it should be evicted from the RSR registry to prevent others from attempting to use that missing instance (or cluster). Here's a NodeJS example showing how to unregister the service at shutdown:

```
// set up proper discovery shutdown
process.on('SIGTERM', function () {
  discovery.unregister(null, function(response) {
    try {
      uuidGenerator.close(function() {
      console.log('gracefully shutting down');
        process.exit(0);
      });
    } catch(e){}
  });
  setTimeout(function() {
    console.error('forcefully shutting down');
    process.exit(1);
  }, 10000);
});
```

Note the added `setTimeout` method to force the service shutdown, if needed.

Since a fatal service crash might preempt sending the `discovery.unregister` message, an RSR usually has a maximum amount of time that it will wait for a pingback. If the wait exceeds the maximum value, the RSR will most likely delete the registration entry or set its status to "down" (or some other value).

Discussion

Each service instance or cluster that is up and running should be listed in an RSR. For example, if an instance of a `netPresentValueCalculator` service is running on a machine in North America and another running on a machine in North Africa, that would be represented by two entries in the RSR. Conversely, the same instance of a service might be listed in multiple RSRs. For example, the instance might be registered in a registry hosted in North America as well as one hosted in Australia.

Keep in mind that a single service (e.g., `computeInterest`) might have multiple *versions*. If those versions are running as standalone instances, each should be listed in the RSR. This allows API consumers to find the version they want to use.

The RSR quite likely will require an authenticated user in order to register your service. This is especially true for RSRs that are available on the web.

Services can use an RSR to locate and connect to internally dependent services, too. For example, if your `computeInterest` service also relies upon the external `taxTable` and `federalRates` services, your service can use an RSR to find and bind to the preferred version of each of those external services, too.

When registering your service, it's advantageous to provide as much metadata as possible to improve your service's "findability." Be sure to include keywords in the `tags` field as well as the `serviceProfile`, `apiDefinitions`, and supported `mediaTypes`.

See Also

- Recipe 3.4, "Describing Problem Spaces with Semantic Profiles"
- Recipe 4.4, "Coding Effective Clients to Understand Vocabulary Profiles"
- Recipe 5.7, "Publishing Complete Vocabularies for Machine Clients"
- Recipe 5.9, "Publishing Service Definition Documents"
- Recipe 7.6, "Supporting RESTful Job Control Language"

5.14 Increasing Throughput with Client-Supplied Identifiers

It is common practice to expect servers to generate and return identifiers for any resources created by client applications. This simplifies the process of ensuring unique resource URLs, but has drawbacks. This is especially true when multistep processes are involved (create A, get unique ID from A, use that to create B, etc.).

Problem

When is it better to allow clients to supply unique identifiers, and how can that be done safely and consistently? What is the easiest, most reliable way to support API consumer-supplied resource IDs?

Solution

A simple and reliable way to generate unique identifiers is to use date-time stamp information or an output of a random number generator to create a string. Here's an example using a random number generator in JavaScript:

```
var id = parseInt(String(Math.random()).substring(2)).toString(36);
// id = "906lem09xu8"
```

You can also use date-time and/or random numbers to generate a Universally Unique Identifier (UUID) (see the examples). There is a specification for creating UUIDs (RFC 4122) that you can use as a guide to write a simple routine. Even better, there are quite a few open source RFC 4122-compliant UUID generators for just about all programming languages. Some programming platforms even offer a direct call for generating UUIDs (e.g., NodeJS v17 supports a UUID function in the `cryptoAPI` module).

Client-supplied identifiers can make it possible to reduce sequential processing of multistep workflows and replace them with parallel processing implementations (see the example).

Client applications should be instructed to generate their own unique identifier and either include that in the URL or supply it in the body of the HTTP `POST`, `PUT`, or `PATCH` request sent to the service API. Service APIs can then inspect the value and confirm it is unique before processing the request. If the value supplied by the client is not unique, the service can return a `409 Conflict` status code with additional information on how clients can fix the problem.

Example

A very simple way to generate a unique identifier is to rely on your programming language's random number generator. Here's a sample implementation in JavaScript:

```
// generating unique identifiers
function makeId() {
  var rtn;

  rtn = String(Math.random());
  rtn = rtn.substring(2);
  rtn = parseInt(rtn).toString(36);

  return rtn;
}
```

Calling the `makeId()` function will generate compact identifier strings that look like this: `"1oyte4x0zep"`.

Some random number generators are not quite as "random" as most of us would like. For that reason, you can mix a date-time value with a random number and generate an RFC 4122-compliant UUID value. Here is an example function that does the trick:

```
// generating UUIDs (Public Domain/MIT)
function generateUUID() {
  var d = new Date().getTime();
  var d2 = ((typeof performance !== 'undefined') &&
    performance.now && (performance.now()*1000)) || 0;
  return 'xxxxxxxx-xxxx-4xxx-yxxx-xxxxxxxxxxxx'.replace(/[xy]/g, function(c) {
    var r = Math.random() * 16;
    if(d > 0){
      r = (d + r)%16 | 0;
      d = Math.floor(d/16);
    } else {
      r = (d2 + r)%16 | 0;
      d2 = Math.floor(d2/16);
    }
    return (c === 'x' ? r : (r & 0x3 | 0x8)).toString(16);
  });
}
```

The output of UUID functions is much less likely to result in a collision, but the resulting output (e.g., `"e6db8698-7128-478d-8658-12c2cb9dd126"`) is longer than the simple example shown earlier.

The unique values can also be shipped from the client application as part of the message body for HTTP `POST` operations:

```
<p>Create a New Person</p>
<form name="create" action="/persons/" method="post">
  <input type="hidden" name="unique-id" value="1oyte4x0zep" />
  <input type="text" name="name" value="Mark Morkleson" />
```

```
  <input type="submit" />
</submit>
```

Or as part of the URL for HTTP `PUT` operations:

```
<p>Create a New Person</p>
<form name="create" action="/persons/1oyte4x0zep" method="put">
  <input type="text" name="name" value="Mork Markleson" />
  <input type="submit" />
</submit>
```

See Recipe 5.15 for more on using HTTP `PUT` to create new resources.

It is up to the server-side component to confirm that the client-supplied unique identifier is acceptable. For example, does this identifier (and a related resource) already exist? Is the identifier properly formatted (e.g., a proper length string with appropriate characters, etc.)? If, for any reason, the client-supplied identifier is not acceptable, the service interface should reject the write operation with an HTTP status of `409 Conflict`.

Discussion

Some client application developers will balk at the idea of supplying their own unique values. You can ease their worries by supplying a code library that generates unique identifiers and encouraging client application developers to use them as they would an SDK (software development kit).

An optional approach to requiring client-supplied identifiers is to provide a fallback approach in the service interface that will generate the identifier if it is not provided by the client. A simple "if-test" can be used:

```
function addItem(id, body) {
  var item;

  if (id) {
    item.id = id;
  } else {
    item.id = makeId();
  }
  ...
}
```

This is especially handy if the API is just a thin proxy for an existing backend service.

One of the advantages of relying on client-supplied identifiers is that it can help reduce the need for sequential workflow processing. For example, consider the case where three resources need to be updated: `customer`, `account`, and `salesRecord`. Also assume that the `account` record needs to have the `customer` identifier as a property, and that the `salesRecord` needs to have both the `customer` and `account` identifiers as properties. In this scenario, if the services are supplying the identifiers, your workflow might look like this:

```
writeCustomer()
  .then(function(customerResponse) {
    return writeAccount(accountResponse);
}).then(function(nextResponse) {
    return writeSalesRecord(salesRecordResponse);
}).then(function(finalResponse) {
    console.log('Final response: ' + finalResponse);
}).catch(failureCallback);
```

Essentially, each step must wait on the previous step to complete before continuing. However, if you rely on client-supplied identifiers, you might be able to do this:

```
var cId = makeId();
var aId = makeId();
var sId = makeId();
Promise.all([
  writeCustomer(cId),
  writeAccount(cId,aId),
  writeSalesRecord(cId,aId,sId)
])
.then(() => console.log('All done!'))
.catch(function(err) {
  rollbackAll(cId,aId,sId);
  console.log('Write failed!');
});
```

A side effect of the client-supplied identifier recipe is that the resource identifiers are not sequential and are rarely easy for people to read or remember. This lack of sequence can make it harder for malicious coders to guess your resource identifiers. It can also make it harder for legitimate developers to navigate a collection of records. You can mitigate this second problem by allowing clients to supply "friendly IDs" that can be displayed in lists and other output while the generated identifier is hidden in the user interface:

```
<ul>
  <li id="q1w2e3r4">Sally-R</li>
  <li id="p0o9i8u7">Jane-Q</li>
  <li id="6y5t4r3e">Barb-Z</li>
</ul>
```

See Also

- Recipe 3.6, "Designing Consistent Data Writes with Idempotent Actions"
- Recipe 3.8, "Designing for Repeatable Actions"
- Recipe 3.9, "Designing for Reversible Actions"
- Recipe 5.15, "Improving Reliability with Idempotent Create"
- Recipe 6.2, "Making All Changes Idempotent"
- Recipe 7.16, "Short-Term Fixes with Automatic Retries"

5.15 Improving Reliability with Idempotent Create

While the common practice is to use HTTP `POST` to create new resources in web APIs, it is not the most reliable. In fact, when it comes to M2M interactions (e.g., no humans in the loop), using HTTP `POST` can be problematic because of the possibility of the *lost response* problem. What is needed is a solution to creating resources that is repeatable and reliable, even when the network connections may be faulty.

Problem

When using HTTP `POST` to create new resources, it is possible to experience the lost response problem. For example, an API client sends a `POST` that transfers $500 from account A to account B, and that API client never receives an HTTP response. No `200 OK`, no `400 Bad Request`, no `500 Server Error`—nothing.

Now what is the client to do? Did the request ever make it to the server? Was there a server-side error that rejected the request? What if the request made it to the server and was completed, but the `200 OK` response got dropped on the network? In that last case, repeating the request might result in executing the transfer twice.

How can we avoid the bad effects of the lost response problem when writing data to the service?

Solution

A lost response for HTTP `POST` actions is troublesome because the state of the server resource is unclear. Was the server updated or not? To compound the problem, the HTTP specification for `POST` does not define the method as *idempotent*. That means, unlike HTTP `PUT`, an HTTP `POST` request cannot be "repeated automatically if a communication failure occurs before the client is able to read the server's response."

An important element of this recipe is the use of client-supplied unique URLs. See Recipe 5.14 for more details.

The solution to this problem is to use `PUT` for any case where API clients want to modify the resource on a service, whether the intent is to create a new record or to update an existing record. Since HTTP `PUT` is an idempotent method, client applications can confidently repeat requests that intend to modify data on the service without the worry of "double-posting" resources by mistake.

There are other options for supporting idempotent updates while still using HTTP `POST`. A recent example is "The Idempotency-Key HTTP Header Field". I continue to encourage implementing solutions along the lines of this recipe since they work well for existing HTTP versions without the need for new headers or other extensions.

Client applications can also use the `If-None-Match` header to indicate they want to create a new resource, or the `If-Match` header when the client intends to update an existing resource.

In this way, client applications avoid the ill effects of the list response problem.

Example

The key to the lost response solution is to always support an idempotent HTTP method (`PUT`) to write data to the server. When you want client applications to create new resources, instruct them to use the `PUT` method, supply a complete URL, and include the `If-None-Match` header with a value of `"*"`.

Here is an example of using `PUT` for creating HTTP resources:

```
PUT /persons/q1w2e3r4
Host: api.example.org
Content-Type: application/json
Content-Length: XXX
If-None-Match : "*"

{"name":"Mark Morkelson"}
```

Upon receiving this request, the API service can check if there is already a resource at the provided URL, and "if none match" the wildcard entity tag (`"*"`), the server can create the resource and return the following response:

```
201 Created
Location: http://api.example.org/persons/q1w2e3r4
```

If, however, a resource already exists at that URL, the server can return the following response:

```
209 Conflict
Content-Type: text/plain

Unable to create. Resource already exists.
```

When client applications want to update existing resources, they need to supply that resource's unique entity tag (provided by the server) when sending the updating `PUT` request. Here is a sample exchange:

```
**** REQUEST ****
GET /persons/q1w2e3r4
Accept: text/plain

**** RESPONSE ****
200 OK
Content-Type: application/vnd.collection+json
ETag: "w/p0o9i8u7y6yt5r4"

{"collection": {
  "items": [
    {"href" : "/persons/q1w2e3r4", "data" : [ {"name" : "Mark Morkleson"} ]}
  ],
  "template" : { "data" : [ {"name" : "Mork Markleson"} ] }
}}

**** REQUEST ****
PUT /persons/q1w2e3r4
If-Match: "w/p0o9i8u7y6yt5r4"
Content-Type: application/x-www-form-urlencoded
Accept: application/vnd.collection+json

name=Mork%20Markleson

**** RESPONSE ****
200 OK
Content-Type: application/vnd.collection+json
ETag: "w/i8u7y6t5r4e3"

{"collection": {
  "items": [
    {"href" : "/persons/q1w2e3r4", "data" : [ {"name" : "Mork Markleson"} ]}
  ]
}}
```

Note that, for the update scenario, the `If-Match` header is sent with the value of the existing resource's `ETag` header. The server, upon seeing the `PUT` request, can use the

value of the `If-Match` header to confirm the resource exists—and that it was not altered by some other update process (which would result in a new entity tag value). If the update fails (e.g., no existing resource or entity tag mismatch), the server can return a `409 Conflict` response.

Discussion

Supplying a complete URL when creating records may look a bit strange to some API designers. But this is typically the way most HTTP client applications implement file uploads (images, PDF documents, etc.).

This recipe relies on client applications interacting with HTTP headers (`ETag`, `If-None-Match`, and `If+Match`). This might be a challenge for some API consumers (e.g., browsers limited to HTML `FORMS`). You can still make this work by supplying the entity tag related information as hidden fields in server-supplied forms. Technically, you could even use HTTP `POST` for this recipe, too. The HTML browser implementation would look like this:

```
<p>Create a New Person</p>
<form name="create" action="/persons/q1w2e3r4" method="post">
  <input type="hidden" name="if-none-match" value="*" />
  <input type="text" name="name" value="Mark Morkleson" />
  <input type="submit" />
</submit>

<p>Update an Existing Person</p>
<form name="update" action="/persons/q1w2e3r4" method="post">
  <input type="hidden" name="if-match" value="w/p0o9i8u7y6yt5r4" />
  <input type="text" name="name" value="Mork Markleson" />
  <input type="submit" />
</submit>
```

While this will work for HTTP `POST`, I still encourage you to implement servers to use HTTP `PUT` instead, since `PUT` is defined as idempotent (reliably repeatable) and this use of `POST` is only "treating" the interaction as repeatable by convention.

By sticking to idempotent methods for all write operations, services can also provide support for automatic retries for updates. Most HTTP libraries support automatic retries for HTTP `GET`, but I have yet to find one that does this for `PUT` or `DELETE`, too. See Recipe 5.16 for more on this option.

See Also

- Recipe 3.6, "Designing Consistent Data Writes with Idempotent Actions"
- Recipe 5.14, "Increasing Throughput with Client-Supplied Identifiers"
- Recipe 5.16, "Providing Runtime Fallbacks for Dependent Services"

- Recipe 6.2, "Making All Changes Idempotent"
- Recipe 7.16, "Short-Term Fixes with Automatic Retries"

5.16 Providing Runtime Fallbacks for Dependent Services

In cases where a service calls out another service, that first service is *dependent* on the second service. Since these service calls rely on the network, a long list of problems may arise. While the chances that a single dependent service call may fail might be small, if your service depends on several external services, your likelihood of failure grows—exponentially. When implementing service APIs, you need to account for this failure possibility and mitigate any problems.

Problem

In cases where a service depends upon another external service, how can you reduce the risk of failure? What patterns can you use to limit the possibility of these kinds of "fatal dependencies," and how can you work around them at runtime when they happen? How can we make sure to maintain service reliability both in the case of *reading* data from a dependent service and *writing* data to those services?

Solution

There will be cases when an API is actually an interface that aggregates other APIs. For example, a shopping API might be implemented as a mix of three other APIs: a shopping-cart API, a payment API, and a delivery API. The danger is that one or more of these dependent APIs is unreachable at runtime due to network or local service problems. What's needed is a well-defined "backup plan" or fallback option for each of the dependent APIs.

It is important to keep in mind that your solutions need to take into account both the "read-from" and "write-to" scenarios.

Protecting your service interface from the failures of the network, or the failures of *other* services upon which you depend, can be mitigated through a handful of methods:

Automatic retries
: One option is to assume the failure is temporary, and that a short pause and retry will solve the problem. Most HTTP libraries have this auto-retry feature for HTTP `GET` requests built in, but few HTTP modules support auto-retry for

unsafe actions (e.g., `PUT` and `DELETE`). However, modifying your own code to handle this is not too difficult.

Static fallback options
: Just as you can configure (or hardcode) the initial API dependencies (for example, `var shoppingAPI = "http://shopping.example.com/home"`), you can expand the configuration to include a second, "fallback," location for the same service (`var shoppingAPIFallback = "http://other-shopping.example.com/home"`). Then, when the initial location fails to respond, you can update the runtime code to point to the alternate service. This has some implications for state management—see Recipe 3.7 for details.

Dynamic fallback options
: You can also write your API aggregator to be able to enlist a service registry to find another available alternative for your needed functionality. This is, essentially, a runtime service lookup similar to the runtime DNS that locates machines. Check out Recipe 5.13 for more details.

Queuing requests for later replay
: If you can't gain a connection to either your initial service or an expected replacement service, you may be able to simply "hold on to" the request in a queue and replay it later, when the currently failing service is once again available. This requires setting up and managing a local queue and some code to process that queue's content. Typically, the initial response in this case is an HTTP `202 Accepted` with a response body that describes details on how the API consumer can monitor the handling of the delayed request (see Recipe 7.15). Of course, you'll need to include this possibility in your documented interface design so that API consumers can be prepared for this kind of response.

Give up
: The final option is to just stop processing requests, tell the API consumer that is calling your API that you are "unable to process the request at this time," and return an HTTP `500` error. This is the safest and the least functional option. In this case, if possible, you should return a response body that indicates some estimated wait time before the API consumer can try this request again. Keep in mind that *your* `500 Internal Service Error` might trigger the API consumer to kick in its own mitigation code, which might also affect that API consumer application's consumers, and so forth.

If the dependent service is used primarily as a simple data source (e.g., list of postal codes, countries, states/regions, product numbers, etc.), you may be able to use some of the data caching recipes covered in Chapter 6.

Basically, you want to implement your API to assume failure on the part of other elements of the system you are dependent upon and incorporate at least one of the preceding mitigating solutions for each possible failure point. Ultimately, you cannot *prevent* the failures, but you may be able to mitigate its effects. To quote John Gall: "Any large system is going to be operating most of the time in failure mode."

What About the Circuit-Breaker Pattern?

You might have expected to see some other solutions in this recipe such as the Circuit Breaker. Since this book is focused primarily on the M2M interaction level, I've left out many of the established code-centric solutions like the Circuit Breaker and others. See the preface for some books that I highly recommend as guides for coding microservices, including Circuit Breaker and other patterns.

Example

This recipe outlines several possible solutions. Up next are examples of each.

Automatic retries

You can arrange your API calling code to automatically retry requests. The key elements to manage are:

- The type of request failure (e.g., HTTP `502` is returned)
- The type of request that was made (e.g., `GET`, `PUT`, `DELETE`)
- The time to wait before trying again (e.g., 250 ms wait before retry)
- The number of times to retry before giving up (e.g., retry three times)

For example, internal API code might look like this:

```
var reqParams = {} // request params
reqParam.host = "https:/api.example.com"
reqParams.url = "/users/q1w2e3";
reqParams.body = "mork=mamund&name=Mike Morkelsen";
reqParams.method = "PUT";
reqParams.waitMS = 300;
reqParams.retryAttempts = 3;
reqParams.successFunction = requestSucceeded;
reqParams.failFunction = requestFailed;

httpLib.request(reqParams);
```

After this fails, you may wish to repeat the attempt (with a modified wait period value) or you can attempt one of the other mitigations in this recipe (static fallback, dynamic fallback, queuing requests, or give up).

Static fallback

You can update your request functionality to include attempts to call an alternate host by including the alternate location in your request collection and adding code to your request method to switch hosts and make additional request attempts:

```
var reqParams = {} // request params
reqParams.host = "https:/api.example.com"
reqParams.url = "/users/q1w2e3";
reqParams.body = "mork=mamund&name=Mike Morkelsen";
reqParams.method = "PUT";
reqParams.waitMS = 300;
reqParams.retryAttempts = 3;
reqParams.successFunction = requestSucceeded;
reqParams.failFunction = requestFailed;
reqParams.alternateHost = "https://alternate-api.example.com";

httpLib.request(reqParams);
```

After that fails, you can try the other options.

Dynamic fallback

This one is a bit trickier since you don't have a fixed alternate host but need to go "find" one instead. See Recipe 5.16 for details on how to implement dynamic fallbacks.

Queuing requests

If retries and fallbacks fail, you can offer to queue the request and try it again later. This can be added as an option in your local request implementation:

```
var reqParams = {} // request params
reqParams.host = "https:/api.example.com"
reqParams.url = "/users/q1w2e3";
reqParams.body = "mork=mamund&name=Mike Morkelsen";
reqParams.method = "PUT";
reqParams.waitMS = 300;
reqParams.retryAttempts = 3;
reqParams.successFunction = requestSucceeded;
reqParams.failFunction = requestFailed;
reqParams.queuingFunction = queueRequest;
reqParams.alternateHost = "https://alternate-api.example.com";

httpLib.request(reqParams);
```

When you do this, you will need to return an HTTP `202 Accepted` response to the API consumer along with a body that includes additional information. See Recipe 7.15 for details.

Give up

At some point you'll need to admit failure and just stop trying to complete the request. In this case, you should return the appropriate response (an HTTP 5xx error) along with a response body that indicates the inability to complete the request. Since this request might be one of a series of API requests made by the API consumer, you should include any information that might be needed by the consumer to support additional rollbacks of other requests in the series. See Recipe 7.16 for details.

Discussion

The good news is that the first few options listed (automatic retires, static fallbacks, and dynamic fallbacks) can all be implemented without coordination with the API client. They're "hidden" implementation details.

Conversely, the queuing requests option requires substantial coordination with API clients since it not only requires the client be prepared for the `202 Accepted` response, it may need to tell other services about this queuing issue. Consider the case where our service handles payment processing and some dependent service is unavailable. Returning `202 Accepted` to the API caller might create a mess on their end. Maybe they have already committed to decrementing inventory and scheduling a delivery of purchased goods. They will now need to decide if they should "unroll" those other activities, leave them in place, and not tell the calling API about it, or offer the calling API an option to wait or cancel the activities.

Don't attempt retries on nonidemptotent HTTP methods like `POST` or `PATCH`. Instead only support these mitigations for `GET`, `HEAD`, `PUT`, and `DELETE`. In rare cases, HTTP `POST` operations can be interpreted as failures even when they are completed successfully; for example, when the target service's `201 Created` response never makes it back to the API consumer that made the call. See Recipe 5.15 for more on this scenario. In the case of `PUT`, the consequence of this "false failure" is much less likely to result in corrupted or duplicate data on the target service.

With the exception of the give up option, all the mitigations mentioned here involve some additional processing on the part of the API implementation. These added features (retries, fallbacks, queuing) will all need to be implemented locally to provide any benefit.

Don't make the mistake of implementing these mitigations as shared external services that get called by your API. This turns them into (possibly fatal) dependencies, too! If your API code can't reach other underlying services to do its work, it might not be able to reach your external mitigation services, either.

While the most common case is to use one of these mitigations for any HTTP 5xx level response, you might want to limit when you reset to these mitigations. For example, HTTP `408 Request Timeout` is a good candidate for these solutions.

There may be new 5xx-level HTTP responses added in the future, and it may not be a good idea to engage in these mitigations for just *any* 5xx response. The ones I recommend as good candidates for retries, fallbacks, and queuing are: HTTP 500, 502, 503, and 504. Other 5xx values might be better handled by just returning the give up option right away.

These options do not require the agreement of the underlying (failing) service. For example, your service interface can implement retries, fallbacks, or queuing mitigations without involving the service that has failed to respond in a timely manner.

When implementing the retry option, you should check the documentation of the target service (the one you plan to send retries) to make sure you don't program your interface code to trigger a denial of service response. Be sure not to abuse the underlying services by sending retries too often or too quickly.

It is a good idea to keep a running history of all requests processed by your service. This is especially true in cases (like those covered in this recipe) where your service is doing additional work to store, forward, and process API consumer requests.

When implementing the fallback options, you may need to provide additional request/response management if you plan to keep track of the activities (e.g., keep a history of past actions for the API consumer). By setting up the use of other services as a fallback, you are making the processing target a *variable*—not all requests may have been processed by the same service. If this detail matters, you will need to arrange for your service interface to keep track of—and possibly offer additional request tracking and management options for—these requests.

See Also

- Recipe 5.4, "Expressing Internal Functions as External Actions"
- Recipe 6.9, "Improving Performance with Caching Directives"
- Recipe 7.16, "Short-Term Fixes with Automatic Retries"
- Recipe 7.17, "Supporting Local Undo or Rollback"

5.17 Using Semantic Proxies to Access Noncompliant Services

At the heart of the recipes in this book is the notion of supporting resilient API clients by using hypermedia-based service interfaces. But there are times when the services we need to expose to the network do not implement hypermedia interactions (e.g., an FTP upload service). In some cases, we don't have the ability to change those services since they are operated by a third party outside our own IT ecosystem.

Problem

How can we safely expose noncompliant services in our RESTful web microservices ecosystem? When is it a good choice to create a local proxy service that wraps other, possibly remote third-party services? What does it take to create stable, reliable proxies for services we did not design and do not maintain?

Solution

There are times when we need the functionality of an existing service, but the service design was not meant to support adaptable, evolvable interfaces in a RESTful ecosystem. In these cases, we need to create local, compliant service proxies that act as "translation" devices between those noncompliant services and the rest of our service ecosystem.

Typically, we need to design the *desired* service interface and then work up our own local code within the API "wrapper" that translates RESTful requests into ones that are supported by the noncompliant services. These proxies also need to translate any responses from the noncompliant services into RESTful resource representations.

Sometimes the only challenge for an existing service is that it doesn't operate with the preferred semantic vocabulary or it doesn't support the desired media types for message exchange. In these cases, it isn't too much work to introduce a kind of "semantic translation" proxy that hides the noncompliant service from API consumers.

The proxy solution makes sense when you need to expose functionality that is "trapped" within noncompliant (see "Shared Principles for Scalable Services on the Web" on page 16) services and it is not possible (or viable) to modify those services to bring them into line with RESTful web microservices principles.

There are three types of service proxies we'll discuss in this recipe: the enterprise-level proxy (ELP), the custom one-off proxy (COP), and the semantic profile proxy (SPP).

Enterprise-level proxy

In some cases, you might be able to create an ELP that algorithmically translates between the noncompliant and compliant services. A good example of an ELP is IBM's CICS External Call Interface (EXCI). This solution works well when your organization has invested a great deal of effort in developing a domain-specific service platform (e.g., an accounting system) that was never implemented as a RESTful set of services.

The ELP approach works well when you need to translate an established collection of related services (or a single monolith offering a wide range of functionality from a single source).

Custom one-off proxy

A more incremental approach is to create COP to translate existing functionality into an RWA-compliant interface. For example, you might want to add a RESTful interface to an existing FTP file upload service. The COP approach limits the effort needed to solve a compliance problem and can also be easier to modify and scale as you go along.

Use the COP approach when you have a heterogeneous mix of services you need to bring into compliance or when you anticipate a small set of services needing this kind of attention.

Semantic profile proxy

Sometimes the only barrier to interoperability for an external service is its inability to "speak" in a language the other parts of your system understand. For example, you might be dependent on services that only exchange messages in `text/csv` instead of a hypermedia format like SIREN or Collection+JSON. Or the responses may not be expressed in your organization's preferred vocabulary (e.g., FHIR for health data or BIAN for banking information). In these cases, you might be able to set up an SPP to "normalize" the data exchanges.

SPP implementations can be as simple as converting XML to SIREN or as complex as adding hypermedia forms to a set of exchanges originally designed to only support CSV files. For this reason, it is sometimes difficult to estimate the level of effort needed to design and implement an SPP solution. You might be able to use an ELP-like approach to establish standardized transformations for messages. For example, you might be able to write an XLST transformation proxy that converts between Collection+JSON and XML.

No matter which approach you are using (SPP, COP, or ELP), all service proxies need to deal with the following elements:

- Create a semantic profile that delineates the domain in use (Recipe 5.7)
- Publish an API definition document outlining the functionality of your proxy (Recipe 5.9)
- Write code that implements all the external actions using the underlying services, internal functions (Recipe 5.4)

Essentially, writing a service proxy is the work of designing, defining, and implementing a new service interface. The key difference here is that there is already a working service that relies upon another, noncompliant implementation.

Example

Each proxy model (ELP, COP, and SPP) requires a slightly different implementation approach.

COP example

This is an example of a COP implementing a RESTful proxy for an FTP file upload service. The following is the underlying service interface we need to "cover," along with a RESTful interface implementation:

```
// HTTP upload external action
function httpUpload(file) {
  var uploader = new httpService();
  var file = uploader.read();
  return file;
}

// FTP client service
function ftpUpload(file) {
  var client = new ftpService();
  var results = client.put(file);
  return results;
}

// proxy function for file uploads
function proxyUpload(file) {
  var results = null;;
  var file = httpUpload(file);
  if(file) {
    results = ftpUpload(file)
  }
  return results;
}
```

Here's a RESTful interface for uploading files (in HTML):

```
**** REQUEST ****
GET /upload-file/ HTTP/1.1
Accept: text/html
...

**** RESPONSE ****
HTTP/1.1 200 OK
Content-Type: text/html
....

<form method="post" action="https://api.example.org/uploads/"
  enctype="multipart/form-data" >
  <input type="file" name=file" value="daily-batch.txt" />
  <input type="submit" value="Upload" />
</form>

**** REQUEST ****
POST /uploads/ HTTP/1.1
Accept: text/html
...

Content-Disposition: form-data; name="file";
...

**** RESPONSE
HTTP/1.1 200 OK
Content-Type: text/html
....

<p>File has been uploaded</p>
```

ELP example

In the case of a enterprise-level proxy (ELP), you can approach this as a translation gateway similar to the one used for the Wireless Application Protocol (WAP) example cited earlier in this recipe. Other examples of algorithmic proxies are ones that convert XML-based messages to JSON-based formats, natural language translators (English to Spanish, etc.), and even protocol translators like the preceding FTP<→HTTP proxy or the WAP<→HTTP example referred to earlier.

SPP example

Simple SPPs (e.g., format translators) are usually easy to implement as long as the responses from the underlying service are consistent and the level of customization for output formats is not too high. For example, you might have an XSLT implementation that consistently converts XML to Collection+JSON:

```
results = convert(xmlDocument, xsltCollectionJSON);
```

However, you might need to inject hypermedia controls in the responses, too. This works well if the underlying service provides a status interface (e.g., a CRUD-style model). In that case, you can simply code-in the standard Create-Read-Update-Delete operations in the transformation script.

Finally, if you need to convert the vocabulary of the underlying service (e.g., `firstName → givenName`, etc.), you'll need to do more work. Usually, it is difficult to use a transformation language with XSLT. Instead you may need to write additional conversion code that runs separately from any format translations.

Discussion

Implementing COP works when the functionality you want to expose is limited to and/or focused on a small set of static operations. ELPs should be reserved for large-scale efforts to convert an existing investment into a more modern, accessible set of external actions. I've worked on a handful of ELP-type projects and most of them take quite a bit more effort than expected, and almost all of them have a limited lifetime before some other technology supersedes them.

It is rare for service proxies to scale at speed. Most of the time, the work of translating between the two interfaces is a fiddly business of string manipulation and protocol hoping. For this reason, translation proxies should be limited to parts of the system that do not require a high transaction volume or a low latency. Queue-based implementations are usually a common target of service proxying.

Implementing an ELP is not something to take on unless you are sure you have sufficient resources. Not just money, but also the time it will take to complete the first release and maintain it going forward.

When implementing an ELP translator, your semantic profile and API definition files document the gateway-level semantics, not the domain-level semantics of the information that passes through the gateway. In this way, implementing an ELP is very close to designing, defining, and implementing a new message protocol or an end-to-end programming framework.

In general, protocol translation proxies (FTP<→HTTP, etc.) are relatively easy to implement because application-level protocols are mature and stable specifications. There are lots of "rabbit holes" when it comes to metadata (HTTP headers, dealing with FTP status codes, etc.), but most of the time you can smooth these over with generic responses.

Conversely, SPPs are often the most challenging proxies to successfully implement. Often the poorly specified vocabularies, ambiguous meanings of terms, and long-

standing organizational knowledge buried in local service implementations all conspire to make building stable SPPs a daunting task. The smaller the vocabulary (e.g., the target domain), the more likely you are to succeed.

In my experience, there are two types of successful service proxies. The first is one that works "just well enough" and just long enough until a better, more stable solution comes along (e.g., a new product) to replace it; and the sooner the better. The other successful proxy implementation is one that has been around for years, humming along, and no one has any idea how it works. None of the original designers or programmers are around anymore, and only a fool would ever attempt to modify any part of that mysterious (but stable) proxy.

See Also

- Recipe 3.4, "Describing Problem Spaces with Semantic Profiles"
- Recipe 4.4, "Coding Effective Clients to Understand Vocabulary Profiles"
- Recipe 5.4, "Expressing Internal Functions as External Actions"
- Recipe 5.7, "Publishing Complete Vocabularies for Machine Clients"
- Recipe 5.9, "Publishing Service Definition Documents"
- Recipe 6.11, "Extending Remote Data Stores"

CHAPTER 6
Distributed Data

> *First step in breaking the data centric habit, is to stop designing systems as a collection of data services, and instead design for business capabilities.*
>
> —*Irakli Nadareishvili, JPMorgan Chase*

This chapter is devoted to recipes for data-centric service interfaces. Data-centric interfaces need to follow all the same principles covered in Chapter 5 along with some additional details that come from the responsibility of storing and managing data. These details involve assuring data integrity, hiding internal data models and implementation technology, and dealing with a wide range of possible network failures without invalidating any existing data. This is especially important now that data can easily travel around the world and come into contact with regulations such as General Data Protection Regulation (GDPR) and others.

Not all services need to manage their own data, but most have some level of data support responsibilities. The challenge of data-centric services is that they typically support persistent data. Even when a service goes offline, the data must continue to exist and/or be accessible by other services. In some cases the service interface has the task of mixing locally managed data with data from other external services. This compounds the integrity and reliability problem since the target interface must now rely on *other* data services to complete the request work. And, especially in the case of writing data, the more services involved in an action, the more likely an error and the more complicated it is to rectify any problems.

Supporting distributed data in a hypermedia environment (see Figure 6-1) means representing data responses as messages and returning hypermedia controls to communicate possible actions on that returned data. It also means presenting a common information retrieval query language (IRQL) independent of any internal custom data query technology, such as SQL, GraphQL, etc. Editing data on the network also means supporting idempotency for all changes in order to improve the likelihood of

successful writes. Lastly, enabling distributed data on the web means you need to support the ability to modify and extend backend data models without breaking the service interface.

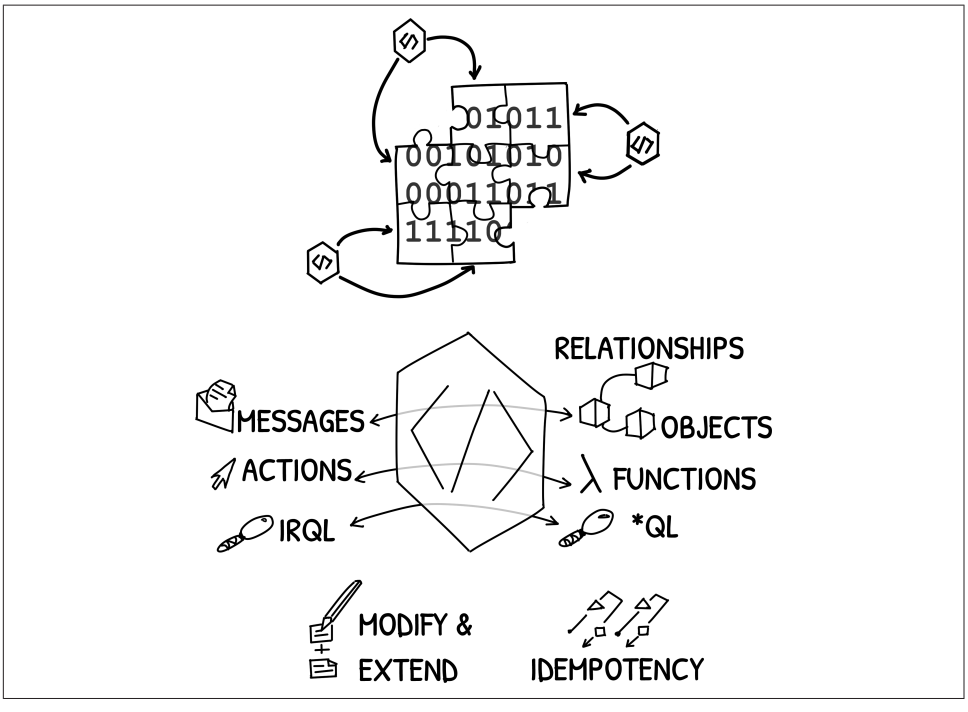

Figure 6-1. Hypermedia data recipes

The Rule of Least Power

The recipes in this chapter were selected to highlight common challenges, identify common principles, and generally offer advice on how to implement APIs that follow Tim Berners-Lee's "Rule of Least Power": "[T]he less powerful the language, the more you can do with the data stored in that language."

Data services are hard; no doubt. The solutions here try to use the least powerful "data languages" in order to support the most you can do with the stored data.

For additional background on designing and implementing data-centric services, see "Supporting Distributed Data" on page 40.

6.1 Hiding Your Data Storage Internals

Data storage technology has changed dramatically over the decades. At the same time, interface designers often need to create APIs that make it possible to interact with data services written using technology and features clearly designed for local access instead of distributed network support. A guiding principle in creating data-centric services is to hide the technology behind the interface and to always present APIs that "speak the language of the API consumer," not the language of the data storage technology.

Problem

What is the best way to ensure APIs for data-centric services don't "leak" the underlying storage and query technology used by that service? How can we shield API consumers from changes in foundational data technology over time? When does it make sense to expose the syntax of current storage tech, and when does it make sense to adopt a more generic query, data management, and storage language for your service interfaces?

Solution

As a rule, it is good idea to *hide* the data storage technology from your API consumers. They should not know whether you are using SQL-based tech, GraphQL, or simple filed-based storage. Ideally, you should design your interfaces in a way that allows you to change the underlying data storage without adversely affecting existing API consumers. The best way to do this is to make sure the service interface focuses on the "job to be done" (e.g., `updateCustomer` or `findUnpaidInvoices`) instead of relying on "data-language" (e.g., `writeToDB(customerObject)` or `queryData('invoices where balance>0')` in your external interface.

New Data Service, Anyone?

The exception to this rule comes up when you are designing and implementing a data management service itself. If your goal is to create the next GraphQL clone or improve on the technology of Apache Lucene, then you need to design and implement your own data platform, including storage, query, and data management languages. That's a big job but certainly a worthy one. But designing a data engine is not the same job as supporting data services for a user management interface.

When adding data functionality to your API, it is best to expose domain-specific actions as your external interface and keep your chosen data technology hidden as part of your API's internal implementation details.

Example

Consider the case of a service that needs to support updating existing objects in a handful of stored collections (e.g., `customers`, `salesReps`, `products`, etc.). Let's also assume that our internal service exposes a generic method (`update(object)`) where the object is one of the ones mentioned previously.

For classic HTTP implementations, you can supply a FORM that supports modifying existing records:

```
**** REQUEST ****
GET /customers/q1w2e3 HTTP/1.1
Accept: application/vnd.siren+json
...

**** RESPONSE ****
HTTP/1.1 200 OK
Content-Type: application/vnd.siren+json
ETag: "w\p0o9i8u7"
...
 "class": ["customer"],
  "properties": {
    "id": "q1w2e3r4",
    "companyName": "BigCo, Inc.",
    ...
  },
  "actions": [
    "name": "update", "type": "application/x-www-form-urlencoded",
    "method": "PUT", "href": "http://api.example.org/customers/q1w2e3r4",
    "fields": [
      {"name": "id", "value": "q1w2e3r4"},
      {"name": "companyName", "value": "BigCo, Inc.",
      ...
    ]
  ]
}
```

In this example, you don't know what data technology is used to implement the update—and that's the way it should be. For example, on first release, the data tech might have been an SQLite database. But now the data storage is implemented using GraphQL. The good news is this change in data platforms could be implemented without adversely affecting the clients using the API.

Exposing data writes (create, update, delete) are pretty easy to do without leaking the underlying technology. But it is a bit more challenging when it comes to safely exposing queries as external actions. Too often it is too easy to slip into just exposing a generic query engine in your service interface. That makes the initial release faster but opens you up to challenges when the underlying tech changes or the underlying table/relationship layout changes. Plus, the data *provider* perspective is not always the

same as the data *consumer* perspective. It is always better to put the consumer's needs above the provider's.

Exposing the underlying query language is a bad idea. Soon you'll be spending time managing the query language instead of managing the problem domain defined by the service interface. Whenever possible, it is a better idea to expose simple links (and possibly forms) that describe the query. This follows the "Rule of Least Power" mentioned at the start of this chapter.

For example, let's assume a case where your service interface needs to support finding all `customers` with outstanding balances more than 30 days, 60 days, and 90 days past due. Here is an example implementation:

```
**** REQUEST ****
GET / HTTP/1.1
Host: api.customers.org
Accept: application/vnd.collection+json
...

{"collection": {
  "title": "Customers",
  "links" : [...]
  "items" : [...]
  "queries" : [
   {"name": "unpaid30", "href": "..."},
   {"name": "unpaid60", "href": "..."},
   {"name": "unpaid90", "href": "..."}
  ]
}}
```

Notice that there is no indication of the data storage or query technology in the service interface. Another possible, slightly more involved, solution would be:

```
**** REQUEST ****
GET / HTTP/1.1
Host: api.customers.org
Accept: application/vnd.collection+json
...

{"collection": {
  "title": "Customers",
  "links" : [...]
  "items" : [...]
  "queries" : [
   {"name": "unpaid", "href": "..."
    "data" : [
      {"name": "days", "value": "30", "required": "true"}
    ]
   }
  ]
}}
```

Adding required parameters to your API means that changing it in the future will be more difficult. If the underlying service changes from using `days` as a query value to `months`, your interface will need to be modified to 1) continue to support `days` and do a local conversion or 2) introduce a backward-incompatible breaking change. Whenever possible, avoid required parameters.

The required parameter could be made optional if the interface also offered the promise to supply a default value (e.g., `"30"`) if no value was passed in the query. This would simplify the query interface, but add more to the implementation.

Discussion

Hiding data tech for writes is usually not a problem. Using HTTP to pass a message (e.g., `customer`, `product`, etc.) with an HTTP method is all you need to support it. See Recipe 6.2 for more on data writes.

Hiding data tech for queries can be a challenge. It is easy to end up exposing whatever query technologies are used by the service itself. This is especially true if the API is the only layer between the API consumer and the data storage itself. Consider the folly of the following data-centric API code:

```
function sqlExecute(connectionString, sqlStatement) {
  var sql = sqlConnection(connectionString);
  var results = sql.query(sqlStatement);
  return results;
}
```

And now here's the (terrible) HTTP interface to match:

```
POST /data/ HTTP/1.1
Host: api.example.org
Accept: application/vnd.collection+json
Content-Type: application/x-www-form-urlencoded
...

sqlConnection=user=mork,pw=m04k,server=db1,database=products&
sqlStatement=select%20*%20from%20products
```

The apparent good news is that the interface designer gets a fully functional data-centric interface without doing much work. The bad news is that what we have here is a security and evolvability nightmare. Don't do this!

There may be times when you want to implement an API that relies on an independent query language like Lucene or some other IRQL. In those cases, exposing the query language is much less of a risk. See Recipe 6.7 for more on how to safely implement language-specific query support.

See Also

- Recipe 3.2, "Ensuring Future Compatibility with Structured Media Types"
- Recipe 5.2, "Preventing Internal Model Leaks"
- Recipe 6.3, "Hiding Data Relationships for External Actions"
- Recipe 7.1, "Designing Workflow-Compliant Services"

6.2 Making All Changes Idempotent

One of the important responsibilities of data services and their interfaces is to ensure data integrity. This can be a challenge when using HTTP over the web, especially when the network is slow or unreliable. This recipe describes how you can improve the reliability of your data writes over HTTP.

Problem

There are times when data writes over HTTP fail, either due to a faulty network or inconsistency in the data to be written. What is the best way to improve the reliability of data writes at a distance when using HTTP?

Solution

The simplest way to improve the reliability of data writes on the web is to limit write actions to rely only on the HTTP `PUT` method. Do not use `PATCH` or `POST` to write data over HTTP. The HTTP method `PUT` is idempotent—it was designed to return the same results, even when repeated multiple times. Neither `POST` or `PATCH` have this feature.

A PATCH Is Not a PATCH

The HTTP Method Registry entry for HTTP `PATCH` identifies the method as non-idempotent. However, the full specification points out that "a `PATCH` request can be issued in such a way to be idempotent." Despite this observation, I continue to recommend using only `PUT` for data writes over HTTP, as I find implementing `PATCH` more of a challenge than simply using `PUT`.

The HTTP `PUT` method was designed such that "the state of the target resource be created or replaced with the state defined by the representation." For this reason, I use `PUT` when attempting to *create* new data records and when *updating* existing data records.

For details on how to use HTTP `PUT` to create instead of using HTTP `POST`, see Recipe 5.15.

Service interfaces should always make HTTP `PUT` actions *conditional* requests. To do this, when HTTP `PUT` is used to create a new data resource, the `If-None-Match: *` header should be sent. This ensures that the record will only be created if there is no resource at the URL used in the request.

When using HTTP `PUT` to update an existing data resource, the API client should send the `If-Match: "…"` header with the value set to the `Etag` header received when reading the record to update. This will make sure that the `PUT` will only be completed when the entity tags (`ETag` in the response and `If-Match` in the request) are the same.

While including the existing record's entity tag values (or the * wildcard), you can make sure the HTTP request only completes when the conditions are right. Using entity tags in this way also makes it possible to resend the same request multiple times without fear of overwriting an update from another API client. In the case of using HTTP `PUT` for creating records, resending the record will not inadvertently overwrite a new record. Finally, in cases where the API consumer gets an initial failure (or a missing response entirely), that client can confidently resend the update without worrying that it will overwrite someone else's work.

Since HTTP `PUT` is idempotent, it can be safely repeated with confidence. This is especially handy in M2M scenarios where humans would not be able to intercept and sort out any failed request problems.

Example

The following are several examples of using HTTP `PUT` for creating and updating an existing resource record.

Using HTTP PUT to create a new resource

Here is a quick example of using HTTP `PUT` to create a new data resource:

```
**** REQUEST ****
PUT /tasks/q1w2e3r4 HTTP/1.1
Host: api.example.org
Content-Type: application/x-www-form-urlencoded
If-None-Match: *
Accept: application/vnd.hal+json
...

id=q1w2e3r4&title=Meet%20with%Dr.%Bison
```

```
**** RESPONSE ****
HTTP/1.1 201 Created
ETag: "w/p0o9i8u7"
Location: http://api.example.org/tasks/q1w2e3r4

**** REQUEST ****
GET /tasks/q1w2e3r4 HTTP/1.1
Host: api.example.org
ETag: "w/p0o9i8u7"
Accept: application/vnd.hal+json

**** RESPONSE ****
HTTP/1.1 200 OK
Content-Type: application/vnd.hal+json
ETag: "w/p0o9i8u7"
....

{
  "_links": {...},
  "id": "q1w2e3r4",
  "title": "Meet with Dr. Bison",
  "dateCreated": "2022-09-21"
}
```

Using HTTP PUT to update an existing resource

Note that, in the first response (`201 Created`), the `ETag` was included. This provides the API client application the proper value for the `ETag` used in the following `GET` request.

Now let's assume the API client wants to update that same `task` record:

```
**** REQUEST ****
PUT /tasks/q1w2e3r4 HTTP/1.1
Host: api.example.org
Content-Type: application/json
Accept: application/vnd.hal+json
ETag: "w\p0o9i8u7"
...

{ "id": "q1w2e3r4", "title": "Meet with Dr. Bison at 16:00"}

**** RESPONSE ****
HTTP/1.1 200 OK
Content-Type: application/vnd.hal+json
ETag: "w\y6t5r4e3"
...

{
  "_links": {...},
  "id": "q1w2e3r4",
  "title": "Meet with Dr. Bison at 16:00",
```

```
  "dateCreated": "2022-09-21"
}
```

Note the updated `ETag` value that was returned in the update response. This represents the new edition of the task resource representation.

Entity tags (`ETag` headers) are unique for each *representation* of the resource. For example, the `ETag` for a `person` resource returned as an HTML document is not the same as the `ETag` for the same `person` returned as a JSON document.

Handling a failed HTTP PUT update

Finally, let's look at a case where the API client application attempts to update the resource again. But this time, the resource was updated by another client application somewhere else. For that reason, the update will fail and needs to be repeated:

```
**** REQUEST ****
PUT /tasks/q1w2e3r4 HTTP/1.1
Host: api.example.org
Content-Type: application/json
Accept: application/vnd.hal+json
ETag: "w\p0o9i8u7"
...

{ "id": "q1w2e3r4", "title": "Meet with Dr. Bison at 16:30"}

**** RESPONSE ****
HTTP/1.1 412 Conflict
Content-Type: application/problem+json

{
  "type": "https://api.example.org/probs/lost-update",
  "title": "The resource has already been updated",
  "detail": "The title properties do not match",
  "instance": "http://api.example.org/tasks/q1w2e3r4",
}
```

When a `412 Conflict` is returned, the API client application should immediately make a `GET` request to retrieve the updated record:

```
**** REQUEST ****
GET /q1w2e3r4 HTTP/1.1
Host: api.example.org
Accept: application/vnd.hal+json
ETag: "w/o9i8u7y6"
...
{
  "_links": {...},
  "id": "q1w2e3r4",
  "title": "Meet with Dr. Bison at downtown office",
```

```
  "dateCreated": "2022-09-21"
}
```

With the refreshed record in hand, the API client can now resubmit the update request:

```
**** RESPONSE ****
HTTP/1.1 200 OK
Content-Type: application/vnd.hal+json
ETag: "w/o9i8u7y6"
...

{
  "_links": {...},
  "id": "q1w2e3r4",
  "title": "Meet with Dr. Bison at the downtown office at 16:00",
  "dateCreated": "2022-09-21"
}
```

In this last example, the failed response was caused by a previous update from another client application. This is often referred to as the "lost update problem". Fixing this problem requires performing a `GET` request on the existing record and then modifying *that* record before resending the `PUT` request.

Handling network failures

Another possible error condition is a network failure. In this case, either the update never reaches the server or the server's response never reaches the client. You can think of this as the "lost response problem". Fixing this problem is easier, if not more reliable. The API client can simply resend the existing request in hopes that the server is once again reachable. If the repeated attempts fail, the API client needs to log the error and stop bothering the target server.

Discussion

Making all write actions idempotent (via HTTP `PUT`) doesn't remove possible failures. Instead, using HTTP `PUT` makes handling the error conditions encountered on the web less complicated and more easily resolved.

Using `PUT` to create resource records means the client application needs to supply the unique resource identifier (e.g., `/tasks/{id}`). A dependable way to do this is to instruct the client to generate a UUID or use a high-precision date-time stamp as the

resource identifier. In both cases there is a lot of guidance on doing this safely and effectively. See Recipe 5.15 for additional details.

Handling 410 Gone

It is possible that an API client will attempt to use HTTP `PUT` to update a record that has already been removed by some other API consumer via an HTTP `DELETE` request. In this case, the API client should have received a `410 Gone` response and logged the condition for later inspection.

Data service interfaces should always return entity tags (`ETag`) in responses. This value is the best way for both API provider and consumers to ensure resource integrity when attempting any write requests. It is important that data-centric service interfaces *reject* any attempt to modify resource state (via `PUT`, `POST`, `PATCH`, and `DELETE`) that does not include an entity tag header (`If-None-Match` for create, and `If-Match` for update and delete).

While I focus on `If-None-Match` and `If-Match` headers, the HTTP specification also defines `If-Modified-Since` and `If-Unmodified-Since` headers based on date-time stamps. I continue to recommend using the headers based on entity tags instead of date-stamps, but both approaches are acceptable.

There will be times when an update fails and there is no simple machine-driven way to resolve the problem. In this case, the API consumer application should write a log record or send a text/email message indicating the failure and ask a human to deal with the problem.

See Also

- Recipe 3.6, "Designing Consistent Data Writes with Idempotent Actions"
- Recipe 4.8, "Every Important Element Within a Response Needs an Identifier"
- Recipe 5.15, "Improving Reliability with Idempotent Create"
- Recipe 7.16, "Short-Term Fixes with Automatic Retries"

6.3 Hiding Data Relationships for External Actions

When following the principle of hiding data technology from the service interface, it is also important to hide any data model relationships employed by that technology. This recipe helps you keep the data relationships hidden from the API while still supporting those relationships at runtime.

Problem

What's the best design to hide any data model relationships (e.g., `person` as one or more address records) from the service interface? Is there a way to support the back-end relationships without exposing the data model or data technology behind that interface?

Solution

The safest approach to writing data that includes entity relationships in the underlying data model is to represent the data properties using a "flat view" of the associated data. For example, in our `person` and `address` case, a single write operation would contain all the fields needed for both entities. It would be up to either the API code or the code in the service *behind* the API to split the single message into as many write operations as needed.

It is also a helpful practice to include in the write response a pointer that allows the client to add another related entity. For instance, the response from a successful write of a new `person` and `address` entity might include a link or form to "Add another address" in the stored model.

Example

Figure 6-2 is an example entity-relationship diagram showing how `person` records and `address` records are related in the underlying data model of the `person` interface.

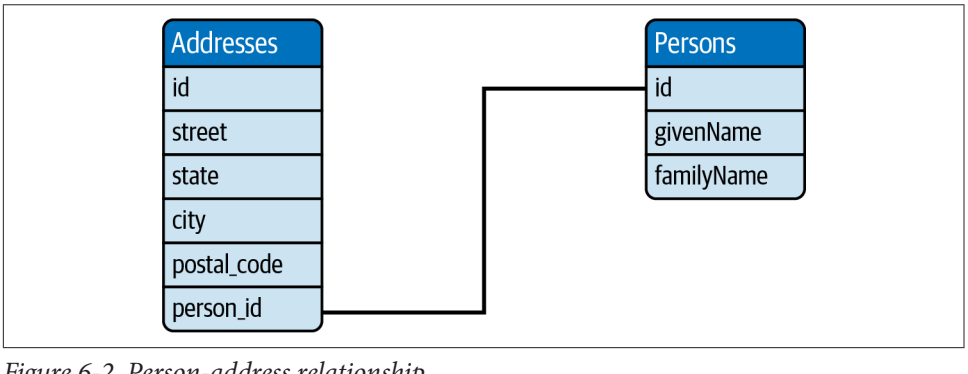

Figure 6-2. Person-address relationship

And here is the Collection+JSON template from the published API:

```
{"collection": {
  "title": "Persons",
  "links": [...],
  "items": [...],
  "template": {
```

```
    "data": [
      {"name": "id", "value": "q1w2e3r4", "type": "person"},
      {"name": "givenName", "value": "Mork", "type": "person"},
      {"name": "familyName", "value": "Markelson", "type": "person"},
      {"name": "street", "value": "123 Main", "type": "address"},
      {"name": "city", "value": "Byteville", "type": "address"},
      {"name": "state", "value": "Maryland", "type": "address"},
      {"name": "postal_code", "value": "12345-6789", "type": "address"}
    ]
  }
}}
```

In this example, both the `person` and `address` data properties are included in a single write message. A minor implementation detail has been added: the inclusion of the `type` property that provides hints on how the data might be organized in data storage. This is purely optional.

Behind the scenes, the API code might turn this into two write operations:

```
var message = http.request.body.toJSON();
var person = personFilter(message);
var adddress = addressFilter(message);
address.person_id = person.id;

Promise.all([writePerson(person), writeAddress(address)]).then(...);
```

In this case, the write operations of both the parent record (`person`) and related child record (`address`) are sent in a single HTTP request allowing the API code to sort out the details. The response to a successful write request can be a representation that includes a link to continue to add more child records:

```
{"collection": {
  "title": "Persons and Addresses",
  "links": [
    {"rel": "person collection", "href": "...", "prompt": "List persons"},
    {"rel": "address create", "href": "...", "prompt": "Add another address"},
    ...
  ],
  "items": [...],
  "queries": [...],
  "template": {...}
}}
```

Another way to hide the underlying data model relationships is to use the "Work in Progress" recipe to collect all the related information (`person`, `addresses`, `online-contacts`, etc.) incrementally as a series of HTTP write requests. Then, once all the data is collected, offer a "submit" operation to ship the complete collection as a final HTTP request. For more on this, see Recipe 7.10.

The key to using this recipe is to model the interface operations independently of any internal data models. This will allow interface designers the freedom to change the API details and/or the storage details in future releases without requiring a breaking change.

Discussion

While the example here illustrates a single relationship (`person` and `address`), this recipe works for any number of entity relation models. For example, the following is a single write request that might be modeled as three storage collections (`company`, `contact`, and `salesRep`):

```
<form name="createEntry" method="PUT" action="...">
  <input name="companyId" value="p0o9i8" class="company" />
  <input name="companyName" value="BigCo, Inc" class="company" />
  <input name="contactId" value="y6t5r4: class="contact" />
  <input name="contactName" value="Mork Markleson" class="contact" />
  <input name="salesRepId" value="w2e3r4" class="salesRep" />
  <input type="submit" />
</form>
```

Note that the last data property in this write message is a reference to a `saleRepId` and no other data properties for `salesRep`. In this case, this is a reference to an existing `saleRep` resource. The interface and/or service layers are expected to know how to deal with this situation. The data model (e.g., an SQL engine) might call for just a reference field (as shown here) or, in a document interface model (e.g., a file-based engine), the `salesRep` data may need to be read from storage and then included in the final storage document.

It is a good idea to log the complete message that was submitted by the client application. This is especially handy if clients request to "back out" or cancel a write operation.

In the preceding example, the "entity hint" pattern (`class="company"`) is employed. This is a handy way to provide additional metadata to the API layer (or backend service) on how the client expects the data properties to be "grouped." But it has its limits. As long as the grouping is supplied by the party in charge of collecting the data properties, it can work well. But it has a kind of "leaky" quality to it.

It is important that client applications not become too tied to any data storage "hints" that appear in responses. This is because the storage modeling might change at some future point, rendering those "hints" incorrect at best and dangerous at worst.

Typically, M2M interactions can handle large payloads (ones that include several entity relations) better than human-to-machine interactions. People have a hard time keeping lots of data in their heads, and really long input forms can be trouble for users. If you are designing a human-to-machine interface, feel free to use the "flat" model approach shown here. If you are designing a human-to-machine interface, consider using the "Work in Progress" Recipe 7.10 instead.

See Also

- Recipe 3.5, "Expressing Actions at Runtime with Embedded Hypermedia"
- Recipe 4.9, "Relying on Hypermedia Controls in the Response"
- Recipe 6.11, "Extending Remote Data Stores"
- Recipe 7.1, "Designing Workflow-Compliant Services"

6.4 Leveraging HTTP URLs to Support "Contains" and "AND" Queries

HTTP has a simple information retrieval query language (IRQL) built into the protocol. The URL query string can be used to support a simple, yet powerful search tool by implementing the "contains" predicate along with the "AND" search condition. This recipe shows you how to quickly and easily add solid search support to your service interfaces.

Problem

What's the easiest way to add IRQL support to HTTP APIs? What search operators, conditions, and predicates are needed for most HTTP queries?

Solution

The simplest search support you can implement for your HTTP interfaces is to leverage the URL query string pattern for name/value pairs (`?name=value`). For HTTP URLs, the implementation you need to support queries is to treat the `=` (equals) operator that separates a name/value pair to actually check for "contains" or "in" rather than a strict "equals" check.

Assume you have the following two records in some storage:

ID	NAME	CITY
q1w2e3	Mork	Middletown
e3r4t5	Mick	Morganville

By implementing `=` as contains, you can implement the following queries:

`?ID=q1w2e3` returns the first record

`?ID=e3` returns both rows

In the first query, the search returns the same results as when the `=` operator is implemented as "equals." However, the second query shows that the `=` operator is really implemented to return any row in which the supplied value (`e3`) is "contained" within the supplied field (`ID`).

You can also leverage the `&` reserved character in HTTP queries to implement the "AND" search condition. That means:

`ID=3e&NAME=Mi` would return only the second row from the preceding table.

The URI specification indicates that the query portion of the URI is defined as case-sensitive (e.g., X=m is not the same query as x=M). However, in most implementations that I encounter, the query portion is implemented as case-insensitive. I recommend only implementing case-sensitive searches where absolutely necessary, and when doing so, you make this implementation detail easily discovered to reduce confusion and frustration for your API users.

Implementing your HTTP search support using name/value pairs with `=` operation supporting "contains" and the `&` supporting "AND" is the simplest IRQL you can implement for your HTTP APIs, and in my experience is often the only one you need. For more on URI query syntax, check out the related IETF specification.

Example

Most hypermedia formats support some version of query forms. Here are some snippets of representations using the data properties in the preceding table.

HTML support IRQL using FORMS:

```
<form method="GET" action="http://api.example.org/persons/" rel="search">
  <input name="ID" value="e3" />
  <input name="NAME" value="" />
  <input name="CITY" value="Mo" />
  <input type="submit" />
</form>
```

Which would be serialized as the following HTTP request:

```
GET /persons/?ID=e3&NAME=&CITY=Mo HTTP/1.1
Host: api.example.org
...
```

Note that all three fields in the form are converted to name/value pairs in the URL, even the field (NAME) with no search value. Be sure to account for this in your interface implementation. For example, you might not pass this name to the search service unless you are sure the search service ignores empty values.

Here's how the same query looks using the Collection+JSON format:

```
{"collection": {
  "title": "Persons",
  "links": [...],
  "item": [...].
  "queries": [
    {"name": "search", "href": "http://api.example.org/persons/",
      "data": [
        {"name": "ID", "value": "e3"},
        {"name": "NAME", "value": ""},
        {"name": "CITY", "value": "Mo"}
      ]
    }
  ],
  "template" : {...}
}}
```

The Collection+JSON example here would result in the same HTTP `GET` request shown in the previous HTML example.

As a final example, the HAL media type does not support inline forms but it does support inline URI templates. Here is an example of the same query as a HAL link template:

```
{
  "_links": {
    "search": {"href": "http://api.example.org/persons{?ID,NAME,CITY}"}
  }
}
```

Notice that, while HAL supports URI templates, these templates do not support including the values inline.

Discussion

Technically, any string value in the URL between the `?` and the `#` (or the end of the URL) is considered the query portion of the URL. You are free to insert any content there you wish. However (again), if you create your own query syntax, you'll need to make sure your API clients understand this syntax before they attempt to create and submit a query. I recommend against creating a new query string format.

It is valid to include the same `name` in a name/value pair multiple times in the same query:

```
?city=northville&city=southland
```

Be sure to take this into account when you implement query support in your API.

Creative Queries

There are lots of creative ways to use the HTTP query string to support IRQL features. I've stuck to the simplest support here since that is often sufficient. It is also the one most commonly understood by HTTP client applications. Implementing other operators (not-equals, less-than, greater-than, etc.) is certainly possible. So is supporting features such as BETWEEN, LIKE, and so forth. However, the IETF has not established standards for expressing these in URIs and, if you really need them, you can move on to other query technologies, like Lucene, SQL, and others. See Recipe 6.7 for more on this topic.

A common (anti)pattern for implementing IRQL requests over HTTP is to include a single URI query string property (e.g., `?q=` or `?query=`) followed by a custom data query. Typically, it looks like this:

```
GET /?query=select * from users where id='q1w2e3r4'
```

Or sometimes the query value is URL encoded:

```
GET /?query=select%20%2A%20from%20users%20where%20id%3D%27q1w2e3r4%27
```

In both cases, you're basically smuggling a data query as part of the URL. This is usually a bad idea. Lots of things can (eventually) go wrong:

The query is too long.
: The length of the query gets so long it may be truncated by some server or gateway along the way.

The query contains reserved characters.
: The query content may include a reserved URI character, which invalidates the request.

URL encoding introduces errors.
: Encoding the query can result in the service behind the API to misunderstand the query, returning unexpected results.

The query exposes a security problem.
: If you are using an exact data query (e.g., SQL statement), you're creating an opening for others to exploit a potential security hole, resulting in data theft or damage.

The query introduces tight coupling.
: By exposing a direct query tied to the data storage model (e.g., SQL, OData, etc.), you're creating a tight coupling between the data model and the API. For

example, changes to the data model (adding/removing/renaming fields) can break the API.

See Also

- Recipe 4.9, "Relying on Hypermedia Controls in the Response"
- Recipe 4.10, "Supporting Links and Forms for Nonhypermedia Services"
- Recipe 6.5, "Returning Metadata for Query Responses"
- Recipe 6.6, "Returning HTTP 200 Versus HTTP 400 for Data-Centric Queries"
- Recipe 6.12, "Limiting Large-Scale Responses"
- Recipe 7.14, "Optimizing Queries with Stored Replays"

6.5 Returning Metadata for Query Responses

It is a good practice to return more than just the query *results* when returning data-centric HTTP responses. You can include information that helps API consumers better evaluate the quality of the response as well as determine if the query needs to be modified and resubmitted to get the desired results.

Problem

What's the best way to represent HTTP IRQL results? What types of metadata should be considered when returning the results of a data-centric HTTP query? How should that metadata be represented in the query response?

Solution

For most data queries, it is not enough to just return the resulting data. It can also be important to return metadata about the query to help client applications that made the request better understand the quality and usefulness of the returned data. In this section, we show a set of metadata values that should be considered as part of the response representation for HTTP queries (see Figure 6-3).

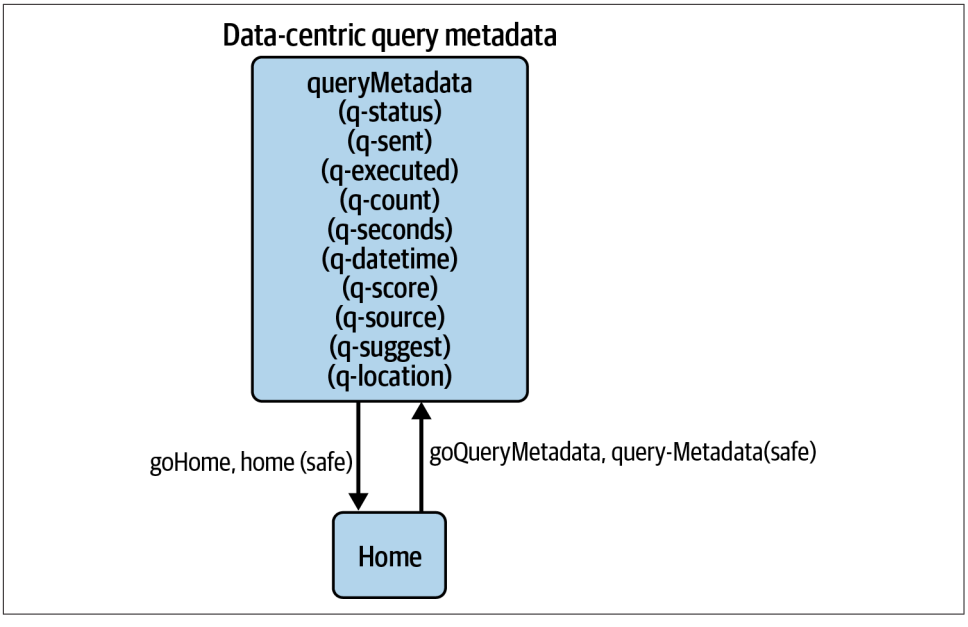

Figure 6-3. Query metadata recipe

How you deliver these metadata values depends on the number of properties you are returning. If a small set of these values is returned, you can include the values in the body of the response. If there are just a couple values to be returned, you can include them with the actual data response as additional data properties. If there are several relevant metadata values, you can return an HTTP link (either in the body or the header collection) that points to a resource that contains all the query metadata. You can also implement a mix of both. See the "Examples" section for details on how to return these values:

Query status (`q-status`)
: This is a simple description of the status of the requested query. Usually this response would be "completed successfully," but if the query was malformed, timed out, or in some other way halted, this string can contain details on what occurred along with possible changes to the query in order to complete successfully. Note that this is not the same as the "error" status of an unsuccessful HTTP request (aka an HTTP 400 response). See Recipe 6.6 for details.

Query sent (`q-sent`)
: This is typically a copy of the HTTP query string or HTTP request body sent by the API client application to the service interface. This can be used to remind humans of the query that was sent as well as allow machines to validate that the service interface parsed the submitted query properly.

Actual query executed (`q-executed`*)*
: In some cases the service interface may modify the query information it received in order to submit that query to the data storage. For example, a simple HTTP name/value pair query might be converted into an SQL "SELECT…" statement. This property contains that resulting modified query. NOTE: this may introduce security problems (see the "Discussion" section).

Number of returned records (`q-returned`*)*
: The number of records returned in this response. This may be a subset of the total number of records matching the query criteria (see `q-count`). For example, the total records found might be 11,000, but the number of records returned in this response might be 1,000.

Estimated size of the result set (`q-count`*)*
: This holds the estimated total number of records selected in the query. For example, a query might result in thousands of selected records, but only the first 100 are returned to the client that initiated the request. If the estimated number is quite high, this may indicate the query is too general and needs to be adjusted (see "Query suggestions").

Elapsed time for query execution (`q-seconds`*)*
: This is the total time (in seconds) that it took for the service behind the interface to execute the query. This can be handy as a diagnostic value. For example, is the query service running slowly? Was this particular query "costly" in terms of resources?

Date/time the query was executed (`q-datetime`*)*
: You should always return the recorded date/time the query was executed. This is especially true if the query response will be cached for later review.

Results relevance (`q-score`*)*
: If available, it can be helpful to pass along any indication of the relevance of the query results that may be returned from the query engine. For example, Lucene supports returning the "score" of the query (usually at the document level). If there is any characterization of the "relevance" of a query, it can be handy to return it in the response metadata.

Data source(s) (`q-source`*)*
: In some cases, it might be handy to return the data sources that were used in fulfilling the query. This can be helpful for those who want to "tweak" the query to improve results. Note that this may introduce security and/or coupling problems; see the "Discussion" section for details.

Query suggestions (`q-suggest`)
: If the query returns too many records, or none at all, there might be suggestions you can return to help client applications improve the quality of their results. These might be suggestions to include or exclude certain fields, add/remove ranges, etc.

Replay URL (`q-location`)
: If the query is stored for later use by the interface or the underlying service, you can generate a unique URL that can make it easy to replay the same query. This replay might be just a return of the same result set or a repeated query using the same (stored) query parameters.

You don't need to include all these data fields in your query metadata. Select the ones that make sense for each use case. For example, if the query was designed to return just one record (`?id=q1w2e3r4`), you might include just the date/time metadata.

Example

When you want to return query metadata, you can use three possible locations:

- The HTTP response header collection
- The HTTP response body
- A link to a separate HTTP resource that contains all the query metadata

You can also do a mix of all of these.

You can pass the query metadata in the HTTP header collection:

```
**** RESPONSE ***
HTTP/1.1 /persons/?id=q1w2e3r4 200 OK
Q-Status: successful
Q-DateTime: 2024-12-12:00:12:0012TZ
...
```

You can also return query metadata as part of the HTTP response body:

```
**** RESPONSE ***
HTTP/1.1 /persons/?id=q1 200 OK
Content-Type: application/vnd.collection+json
...

{"collection": {
  "title": "Person",
  "metadata" : [
    {"name": "q-sent", "value": "?id=q1"},
    {"name": "q-datetime", "value": "2024-12-12:00:12:0012TZ"},
    {"name": "q-status", "value": "result set too large, query canceled"},
    {"name": "q-seconds", "value": "120"},
    {"name": "q-count", "value": "10000+"},
```

```
    {"name": "q-suggest",
      "value": "reduce return set with additional query parameters"},
  ]
  ...
}}
```

If the query metadata is extensive, or if you want to reduce the amount of information in the query response, you can include a link to the query metadata in the header and/or the body of the response representation:

```
HTTP/1.1 /persons/?id=q1 200 OK
Content-Type: text/html
Link: <http://api.example.org/queries/u7y6t5r4>;rel="query-metadata"
...
<html>
  <title>Person</title
  <meta name="query-metadata" href="http://api.example.org/queries/u7y6t5r4" />
  ...
  <body>
    <a href="http://api.example.org/queries/u7y6t5r4">Query Metadata</a>
    ...
  </body>
</html>
```

In this example, the query metadata is returned when the client application follows the `query-metadata` link (*http://api.example.org/queries/u7y6t5r4*). Here is an example of a `query-metadata` response in HAL format:

```
{
  "_links": {
    "self": {"href": "http://api.example.org/queries/u7y6t5r4"},
  },
  "q-status": "success",
  "q-sent": "?state=MN",
  "q-executed": "SELECT * from persons where contains(state,'MN')",
  "q-count": "750",
  "q-seconds": ".5",
  "q-datetime": "2024-12-12:12:12:12",
  "q-score": "100%",
  "q-source": "bigco.persons",
  "q-suggest":  "none",
  "q-location", "http://api.example.org/queries/t5y6r4u7"
}
```

Of course, you can mix and match these options by including some basic data in the headers (e.g., `Q-DateTime`), some in the body (e.g., `q-status`, `q-count`, `q-suggest`, etc.), and all of the information in a separate `query-metadata` resource (via an HTTP link).

Discussion

Returning query metadata is a "mixed bag" of good and bad. Too much metadata—for example, actual query executed, query sources—can expose too much information about your services and data technology, creating security challenges and introducing the possibility of tight coupling. At the same time, having no metadata at all can result in inefficient queries, challenges responding to invalid (or malformed) requests, and generally make it hard for API consumers to easily and safely craft quality queries. You'll need to use caution and experience to find the right balance in each unique case.

Are These Standard Metadata Properties?

This recipe introduces a handful of query metadata properties (`q-status`, etc.). These are not standardized and are not registered header or link relation values. They are values I've used over the years for the projects I've worked on, and I've found them handy. It is possible you have some metadata values of your own, too. Feel free to use them (after documenting them properly using Recipe 3.4) and share them with the wider community.

Ultimately, it is the service *behind* the interface that is responsible for protecting itself against malformed or malicious queries. However, the service interface can often take on some of that responsibility, too. Remember that it is the *interface* that defines the limits of HTTP responses. You can (and should) adopt documented limits to query responses (record count, execution time, etc.). See Recipe 6.12 for more on this topic.

The `q-location` metadata property is meant to hold a "quick link" to the query results. You can use this to recall a previously executed (and saved) query or as a way to re-execute the saved query parameters to get updated results.

Resist the temptation to load the query metadata with service-specific internal information (e.g., debugging, performance statistics, etc.). This kind of data *should not* be shared at the HTTP level with other services and API client applications. Limit the shared values in `query-metadata` to those that are directly related to the API consumer's needs, not the underlying service's needs.

Supporting a `query-metadata` resource link is a good idea if you have lots of query metadata and/or the query is used often (whether accessing an old result or generating a new one). For cases where you are using a simple HTTP name/value query to return a single record, this additional resource may not be very helpful.

See Also

- Recipe 5.8, "Supporting Shared Vocabularies in Standard Formats"
- Recipe 5.10, "Publishing API Metadata"
- Recipe 5.12, "Standardizing Error Reporting"
- Recipe 6.6, "Returning HTTP 200 Versus HTTP 400 for Data-Centric Queries"
- Recipe 7.7, "Exposing a Progress Resource for Your Workflows"

6.6 Returning HTTP 200 Versus HTTP 400 for Data-Centric Queries

When implementing data-centric APIs, there will be some cases where the response body contains no records. In these cases, it is important to return the proper HTTP status code: the one that accurately communicates the meaning behind the empty collection in the response.

Problem

When returning an HTTP response to a data-centric request, what is the proper HTTP status code to use when the response collection is empty? Should the response always be `200 OK`? Or always be a 404 or some other 4xx status code? How do you decide which class of status codes to return, and what is the best way to handle "empty" responses to data-centric HTTP requests?

Solution

Like any other HTTP request, the HTTP status code is an important way to indicate the condition of the response that was returned. And, even when the HTTP request is essentially a data query, the same rule applies. If the request was properly formatted and the server was able to parse and process the response, the HTTP status code should be in the 2xx class. However, if the server was unable to fulfill, understand, or process the request due to errors either in the requests or in the service tasked with fulfilling that request, the HTTP status should be either 4xx (due to a client-side problem) or 5xx (due to a service-side problem).

There are some minor variations on this general rule. See the following list for details:

`200 OK`
: If the request was well-formed, and was meant to return a collection of records that meet a filter rule (e.g., `/persons/?status=pending`), and the result of fulfilling the request is an empty collection (e.g., the service can find no records that match the search criteria), the HTTP status should be `200 OK`.

`404 Not Found`
: If the request is meant to return a single, existing resource (e.g., `/persons/?id=q1w2e3`) and that URL points to a resource that *does not exist*, then the API should return an HTTP `404 Not Found` status code with details.

`4xx Bad Request`
: If the client application has made an invalid request (e.g., a bad URL, improperly formed data query, etc.), then the API should return a `400 Invalid Request` response (along with details on how to correct the problem).

`5xx Server Error`
: If the client request was correct but the underlying service is unable to fulfill the request due to a server-side problem (e.g., unable to reach the data store, network failures, etc.), then the API should return a `5xx` class status code with details.

Using URL query strings to return a single resource is a weak design pattern that should be avoided. Instead of using `persons/?id=q1w2e3` to return an existing HTTP resource, a better design choice would be `/persons/q1w2e3`.

Of course, we don't always get to choose our URL designs; we sometimes need to learn to work with URLs designed by other people for services we do not control.

In general, return `200 OK` with an empty collection for any well-formed queries. Reserve 4xx and 5xx status codes for cases where the client or service encounters an error. The single-resource query that returns 404 is the exception to this rule.

Example

There are four common cases to deal with when determining the proper HTTP status code for "empty" responses to a data-centric query.

Return 200 OK when the filter criteria results in an empty collection

When returning an empty collection in response to a data-centric HTTP request, you should use the `200 OK` status along with some metadata about the query (see Recipe 6.5 for details):

```
**** REQUEST ****
GET /persons/?status=pending HTTP/1.1
Host: api.example.org
Accept: application/vnd.collection+json
...

**** RESPONSE ***
HTTP/1.1 200 OK
```

```
Content-Type: application/vnd.collection+json
...

{"collection": {
  "title": "Persons",
  "metadata": [
    "name": "q-status", "value": "success",
    "name": "q-sent", "value": "?status=pending",
    "name": "q-count", "value": "0"
  ],
  "items": []
}}
```

Note the query metadata (see Recipe 6.5) in the response body. Adding this kind of information in the response can clarify the meaning of the "empty collection" that was returned.

Return 404 when the query string points to a resource that does not exist

When using the URL query string to locate an existing resource, you should return HTTP status `404 Not Found` to indicate that the requested HTTP resource does not exist:

```
**** REQUEST ****
GET /persons/?id=q1w2e3 HTTP/1.1
Accept: text/html
...

**** RESPONSE ****
HTTP/1.1 404 Not Found
Content-Type: text/html
...

<html>
  <title>Not Found</title>
  <body>
    <h1>Not Found</h1>
    <div class="error">
      Unable to locate the requested resource
      <span class="q-sent">?id=q1w2e3</span>
    </div>
  </body>
</html>
```

Remember that this use of HTTP 404 is limited to cases where the URL plus query is *designed* to return a single resource instead of a collection of data rows or resources.

Return 4xx when the client application has sent an invalid request

In cases where the client application has sent an invalid request, you should return the appropriate 4xx status code with hints on how the client can fix the problem:

```
**** REQUEST ****
GET /persons/?hatsize=13 HTTP/1.1
Accept: application/vnd.collection+json
...

**** RESPONSE ****
HTTP/1.1 400 Bad Request
Content-Type: application/vnd.collection+json
...

{"collection": {
  "title": "Persons List",
  "error": {
    "title": "Data Filter Error",
    "message": "The data property 'hatsize' does not exist"
  }
}}
```

Note that in this example, the request is not malformed at the HTTP level. Instead, the data service cannot process the request since the data in the query is invalid. Of course, there may be cases where the HTTP request itself is invalid, which will result in a 4xx response.

Return 5xx when the service cannot fulfill a valid request

There will be cases where the client sends a valid query but the underlying service is unable to properly fulfill the request. For example, the service interface cannot reach the source data or the underlying data request take too long to process, etc. In these cases, the API should return the proper 5xx class status code along with information on the state of the service interface:

```
**** REQUEST ****
GET /persons/?status=active HTTP/1.1
Accept: application/vnd.hal+json
...

**** RESPONSE ****
HTTP/1.1 504 Gateway Timeout
Content-Type: application/problem+json
....

{
  "type": "https://api.example.org/problems/time-out",
  "title": "Data query timed out.",
  "detail": "The remote data storage took too long to reply.",
  "instance": "/persons/?status=active",
  "q-status": "failed",
  "q-seconds": "120",
  "q-suggest": "Try again later."
}
```

Note the use of the HTTP problem details media type (see Recipe 5.12) along with query metadata (see Recipe 6.5) to improve the quality of information returned to the client application.

Discussion

When attempting to decide whether to use 200 or 4xx for data-centric responses, be sure to take into account whether the request is designed to return *collections* or designed to return a *single resource*. In the first case, an empty response is "OK." In the single resource case, an empty response means the resource was *not found*.

Returning 4xx and 5xx responses in your service interfaces may result in exposing internal details of your network or services. When crafting HTTP error responses, always be careful not to reveal too many details on where (and how) your data is stored.

In some cases, your API might be calling other service interfaces that do not follow this recipe. For example, a service might return 404 when the collection query results in zero rows returned. Whenever possible, *do not* echo this behavior back to your own API clients. Instead, modify the return you pass on to downstream clients to follow the "200 OK" rule. This will result in a more consistent and reliable interface for your APIs.

See Also

- Recipe 4.3, "Coding Resilient Clients with Message-Centric Implementations"
- Recipe 5.12, "Standardizing Error Reporting"
- Recipe 6.5, "Returning Metadata for Query Responses"
- Recipe 6.7, "Using Media Types for Data Queries"
- Recipe 7.14, "Optimizing Queries with Stored Replays"

6.7 Using Media Types for Data Queries

Implementing data query support for your service interfaces can be challenging because you're often forced into selecting—and exposing as part of your interface—a single data technology (SQL, GraphQL, Lucene, etc.). This runs counter to the notion of creating loosely coupled APIs that hide the underlying implementation details.

One alternative is to create your own local data query language; one that is not tied to a single data technology. However, doing this can lead you into creating and maintaining a complete data language, which can result in a new, tight coupling. While this seems easy at first, it can become impractical at scale.

This recipe offers a way to decouple data technology from your interface while still taking advantage of powerful established data query languages.

Problem

What's the best way to implement data query support for a service interface? Do you need to select one of the existing query languages (SQL, GraphQL, Lucene, etc.)? Or do you invent your own data query language that is not tied to a single backend technology? How can you support more than one query language for the same service interface? Is there a way to support well-known data query languages without locking your API into a single tightly coupled backend service? What happens when that backend service changes in the future (e.g., migrates from SQL to GraphQL)?

Solution

Balancing the desire for loosely coupled implementations for your APIs against the power of embracing a well-known data query language like SQL or GraphQL can be a challenge. There is a rather simple solution that I have found helpful. However, I rarely see it employed by others. That solution is to implement your data query language support over HTTP as a *media type*. Elsewhere in this book, I've discussed the power of using registered media types (RMTs) (see Recipe 3.1) and semantic profile documents (SPDs) (see Recipe 3.4), as well as the advantage of supporting content negotiation (see Recipe 5.6). The same principles can be applied to supporting negotiable data query languages, too.

The first step is to adopt the policy of using media type definitions for data query languages. This has already been done for the SQL format with RFC 6922. That specification documents the `application/sql` media type as a way to submit SQL queries over HTTP (see the "Example" section).

Although not officially registered, I've used the same technique to support other query languages for my APIs, including:

- *application/prs.solr+json* for Solr
- *application/prs.odata+json* for OData
- *application/prs.graphql+json* for GraphQL

As of the writing of this book, there are no registered media types in the IANA registry for Solr, OData, and GraphQL. I use an unregistered media type identifier that contains the `prs.` prefix (indicating a "personal" or "vanity" registration). Be sure to monitor the IANA Media Type Registry to learn when these and other data query languages register their official media type identifiers.

The key to making this recipe work is to leverage HTTP request bodies to send the data query. For example, the API client can craft a valid query using the designated language (SQL, OData, etc.). That message becomes the body in an HTTP `POST` or `PUT` request (see Recipe 5.15) sent to the service interface. The API accepts the request and then, if needed, converts that into a form usable by the underlying data engine and executes the query, returning the results to the API caller.

Using `PUT`/`POST` to submit data queries means it's possible to easily store and replay those same queries in the future. See Recipe 7.14 for a recipe that shows you how to do this.

By converting your data query requests to media type requests, you also get to take advantage of HTTP content negotiation (see Recipe 5.6). You can include supported data query media types in your service metadata (see Recipe 5.10) and list the media type strings in your client preferences responses (see Recipe 5.5).

Example

Here's the HTTP exchange for submitting as data query using SQL:

```
**** REQUEST ****
PUT /person/queries/q1w2e3r4 HTTP/1.1
Host: api.example.org
If-None-Match: *
Content-Type: application/sql
Accept: text/html
...

SELECT id,name,city FROM persons where city LIKE '%ville%'

**** RESPONSE ****
HTTP/1.1 301 Moved Permanently
Location: http://api.example.org/person/results/p0o9i8u7

**** REQUEST ****
GET person/queries/p0o9i8u7 HTTP/1.1
Accept: text/html

**** RESPONSE ****
HTTP/1.1 200 OK
Content-Type: text/html
ETag: "w/o9p0i8y6"
...
<html>
  <title>Persons</title>
  <body>
    <div class=query-metadata>
```

```
        <span class="q-status">success</span>
        <span class="q-sent">
          SELECT id,name,city FROM persons where city LIKE '%ville%'
        </span>
      </div>
      <div class="results">
        ...
      </div>
    </body>
  </html>
```

Note that the query was sent using the `application/sql` content type to communicate to the service interface which query engine to use for this request. You can also see the use of query metadata (see Recipe 6.5) in the response. See Recipe 5.15 for details on how to use `PUT` to create new resources instead of `POST`.

Two Resources Are Better than One

In all the examples in this recipe, the `PUT` request used to create a new resource (`/person/queries/q1w2e3r4`) creates a *data query* resource and then uses that data query resource to create the *data results* resource (`person/results/p0o9i8u7`). These are two separate resources. One holds the query. The other holds the results. The results resource can be called to return the static results from the data query. The query resource can be used to rerun the query against the data source and retrieve a new result set (and possibly a new URL). For more on this, see Recipe 7.14.

If the interface supports it, the same query could be crafted using Solr:

```
**** REQUEST ****
PUT /person/queries/q1w2e3r4 HTTP/1.1
Host: api.example.org
If-None-Match: *
Content-Type: application/prs.solr
Accept: text/html
...

fl=id name city
city:ville
```

Or maybe an OData query:

```
**** REQUEST ****
PUT /person/queries/q1w2e3r4 HTTP/1.1
Host: api.example.org
If-None-Match: *
Content-Type: application/prs.odata
Accept: text/html
...

$select=id, name, city
$filter=contains(city,'ville')
```

OData has its own rules for how to craft queries for use in HTTP `POST` and `GET` requests. It should be noted that, as of OData 4.1, the rules have changed significantly, too. I have adopted this format (which is slightly easier to read than the current version) and, depending on what version the backend OData engine is running, will rewrite the query to be compliant with the engine. Adopting this, or a similar approach, can keep your service interface stable, even when the backend query engine changes over time.

I found this article by John Gathogo covering the use of OData over HTTP very helpful.

For one more example, here's the same request crafted for the GraphQL query engine:

```
**** REQUEST ****
PUT /person/queries/q1w2e3r4 HTTP/1.1
Host: api.example.org
If-None-Match: *
Content-Type: application/prs.graphql
Accept: text/html
...

{
  person(city: {regex: "/ville/"}) {
    id
    name
    city
  }
}
```

Like the OData query engine, GraphQL has its own rules for sending data queries via HTTP `POST` and `GET`. Be sure to check the documentation and, if needed, insert a rewrite step to convert the `application/prs.graphql` message to one acceptable to the GraphQL engine.

The thing to keep in mind for this recipe is that your goal is to create a stable, decoupled way to submit technology-specific data queries.

Discussion

Communicating data queries via a media type encapsulates the data technology to a strongly typed message format. It also makes it possible to implement multiple data query languages for the same data store without making changes to the interface. That means you can implement support for `application/sql` when you first release the API and, if needed, you can also safely add support for Solr or GraphQL at some future date.

It bears repeating that the recipe shown here uses `PUT` to create a new resource, not `POST`. However, the underlying service *may* need to use `POST` against, for example, a GraphQL or OData query. Or it might use `GET` for Solr, for example. The `PUT` method used here is just for our service interface and is independent of any HTTP request made to backend services.

It might seem better to just adopt the HTTP conventions of the data engine directly and *not* introduce an additional media type implementation for your data query support. This is fine as long as you don't plan on changing your query engine in the future and you don't plan to honor any breaking changes the query engine makes in the coming years. Even though there is some up-front cost (establishing a media type string, setting up an HTTP `PUT`/`POST` endpoint, implementing a rewriter, etc.), adopting this recipe can pay off over time.

The SPARQL query language was designed to be used over HTTP. Since it already has its own set of associated media types, supporting SPARQL fits well into this recipe.

It should be noted that there is no rule that requires the SQL query language to be applied only to SQL-based storage. It is certainly possible to maintain support for SQL queries after you have converted all your SQL databases to file-based storage. The key is to treat the query language as a separate layer in your interface implementation. That way, if needed, you can change backend storage without changing the service interface.

Be careful to not fall into the trap of designing your own data query language. Unless you plan on creating your own unique data engine, it is better to use this media type pattern to create a kind of interoperability between data services and query languages. This offers lots of possibilities without introducing a new language that you may not have the resources to support over time.

See Also

- Recipe 3.1, "Creating Interoperability with Registered Media Types"

- Recipe 3.4, “Describing Problem Spaces with Semantic Profiles”
- Recipe 4.6, “Managing Representation Formats at Runtime”
- Recipe 5.6, “Supporting HTTP Content Negotiation”
- Recipe 5.10, “Publishing API Metadata”
- Recipe 7.14, “Optimizing Queries with Stored Replays”

6.8 Ignoring Unknown Data Fields

Sometimes the data messages being exchanged have more data properties than expected or desired. In those cases, it is a good idea to just ignore the “additional” data properties and continue to process the parts of the message you are interested in.

Problem

There will be times when the data messages exchanged between services contain more data properties than the API consumer wants. For example, a service designed to manage email addresses may be talking to a data service that returns complete addresses for physical and electronic delivery. What is the best way for the consuming service to ensure data integrity when it only reads/writes the properties it knows about? Does the consuming service only return the properties that were modified? Or does it return all fields, even ones it does not understand?

Solution

When reading and then updating a data record from another service, you should always exchange the *complete* data record, even when that record contains fields you did not edit and/or don’t understand. This is true when you are updating the record yourself and returning it to the source. It is also true when you are forwarding the data record to another service for processing and supporting updates from the forwarding service.

In general, it is not a good idea to strip data from an incoming data message if you (or some other service you are passing the data from) also support *writing* that data back to the source. For example, attempting to write partial values to a data store can lead to cases where the updated values are not compatible with the unedited values. That means that the stored data loses integrity, or the write is rejected. Both results diminish the reliability of your service.

At the heart of this solution is what is loosely referred to as the "Must Ignore" rule. When receiving a message from another party, just ignore the parts of the message you don't understand. Operate on the parts you were designed to understand, and return the entire block if you are attempting an update.

Example

Consider the case shown in Figure 6-4 where you create an interface that supports reading/writing email addresses for users (`emailUpdater`).

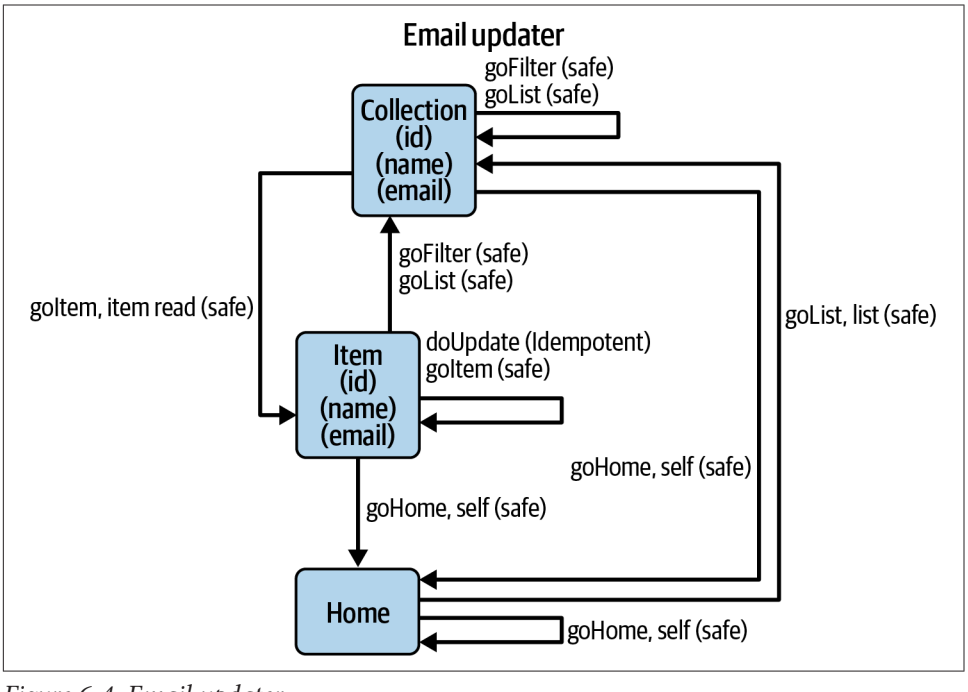

Figure 6-4. Email updater

However, it turns out your interface gets its source data from another service (`addressManager`) that holds lots of information about email and physical delivery addresses for users (Figure 6-5).

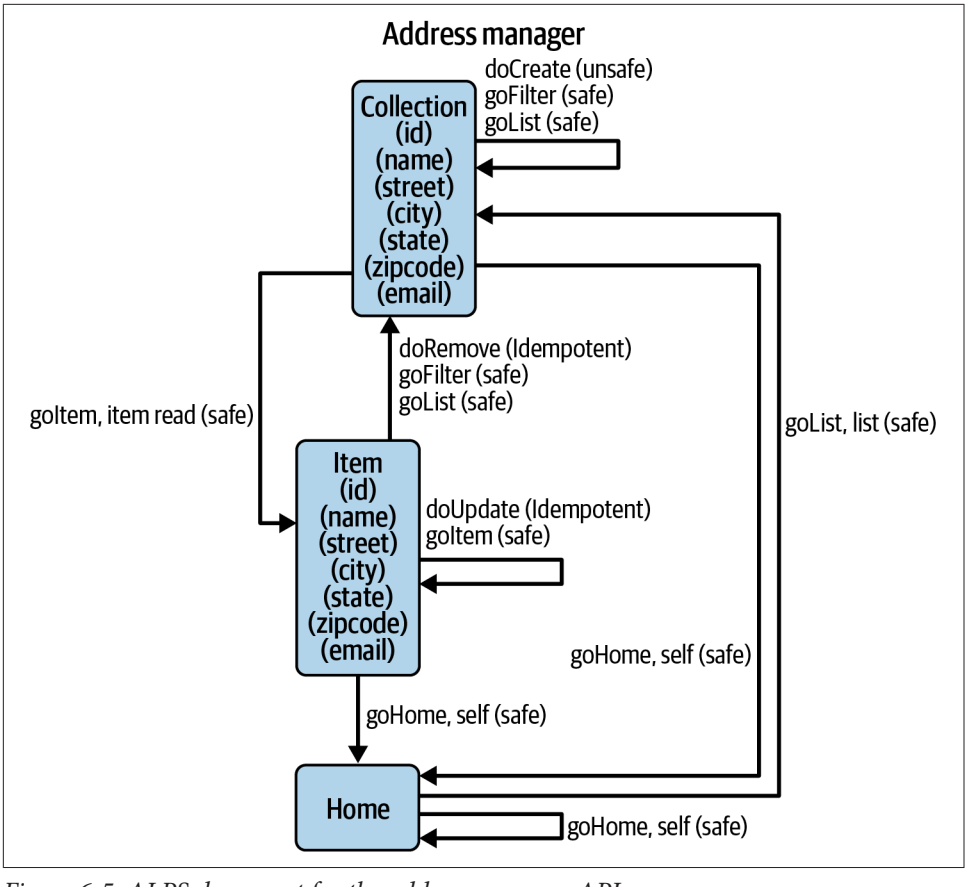

Figure 6-5. ALPS document for the address manager API

When requested, the `emailUpdater` gets the record identifier from the API caller (`GET` *http://api.emailupdater.org/q1w2e3r4)* and uses that to make a call to the `address Mananger`, which returns the following message:

```
**** REQUEST ****
GET /q1w2e3r4 HTTP/1.1
Host: api.addressManager.org
Accept: application/vnd.collection+json
...

**** RESPONSE ****
HTTP/1.1 200 OK
Content-Type: application/vnd.collection+json
Content-Length: XX
ETag: "w\u7y6t5r4"

{"collection": {
  "title" : "Address Manager",
```

```
  "links" : [...],
  "items" : [
    {"href": "http://api.addressmanager.org/q1w2e3r4",
      "rel" : "address",
      "data" : [
        {"name": "id", "value": "q1w2e3r4"},
        {"name": "name", "value": "Mork Mickelson"},
        {"name": "street", "value": "123 Main"},
        {"name": "city", "value": "Byteville"},
        {"name": "state", "value": "MD"},
        {"name": "zipCode", "value": "12345"},
        {"name": "email", "value": "mork@example.org"}
      ]
    }
  ],
  "template": {...}
}}
```

Next, the `emailUpdater` modifies the email as requested and returns the *complete* address record to `addressManager`:

```
**** REQUEST ****
PUT /q1w2e3r4 HTTP/1.1
Host: api.addressManager.org
Accept: application/vnd.collection+json
Content-Type: application/vnd.collection+json
If-Match: "w\u7y6t5r4"
...
```

```
{"template": {
  "data": [
    {"name": "id", "value": "q1w2e3r4"},
    {"name": "name", "value": "Mork Mickelson"},
    {"name": "street", "value": "123 Main"},
    {"name": "city", "value": "Byteville"},
    {"name": "state", "value": "MD"},
    {"name": "zipCode", "value": "12345"},
    {"name": "email", "value": "mickle@example.org"}
  ]
}}
```

```
**** RESPONSE ****
HTTP/1.1 200 OK
Content-Type: application/vnd.collection+json
Content-Length: XX
ETag: "w\i8u7y6t5"
```

```
{"collection": {
  "title" : "Address Manager",
  "links" : [...],
  "items" : [
    {"href": "http://api.addressmanager.org/q1w2e3r4",
      "rel" : "address",
      "data" : [
```

```
            {"name": "id", "value": "q1w2e3r4"},
            {"name": "name", "value": "Mork Mickelson"},
            {"name": "street", "value": "123 Main"},
            {"name": "city", "value": "Byteville"},
            {"name": "state", "value": "MD"},
            {"name": "zipCode", "value": "12345"},
            {"name": "email", "value": "mickle@example.org"}
          ]
        }
      ],
      "template": {...}
    }}
```

Finally, imagine that—at some future point—the `addressManager` adds some new data properties to its messages (e.g., `telephone` and `sms`). The `emailUpdater` can continue to do the same task of modifying the one field it knows about and simply passing back *all* the fields it was returned. In this way, the service behind the `emailUpdater` interface does not need to be modified each time a new field is added by the `addressManager` service.

Discussion

The idea of ignoring elements of a message is a handy way to add resilience to services and APIs that interact with other interfaces. When you are working with other service interfaces that you do not control, you can't be sure of what is going on behind the scenes. You only know what the interface promises. Over time it is possible that the interface will change and, as long as it doesn't change the promises you count on, adding/removing fields or actions outside the scope of your needs is of no real interest to you.

The "Must Ignore" Rule

This pattern of passing along any fields you don't understand instead of removing them is a key element in supporting future compatibility for services and interfaces. The web itself was built with this assumption in mind, too. A good document for reading more about what is loosely called the "Must Ignore" rule can be found in the document "Extending and Versioning Languages Part 1" from the W3C website.

This recipe depends on the ability of a service to recognize and use hypermedia forms in responses. Like all service interfaces, data services should return hypermedia-rich messages (see Recipe 3.5 and Recipe 4.9) that include instruction on how callers can execute additional actions. If you are using a return format that does not support hypermedia (e.g., HAL, CSV, plain JSON, etc.), then you should return a pointer to another document that contains action descriptions (e.g., HAL-FORMS, XML/JSON Schema, ALPS, etc.).

This recipe is closely related to Recipe 6.13 (about pass-through proxies). In fact, you can see this recipe as advice for API consumers that might be talking to an interface that is acting (behind the scenes) as a data proxy. Of course, your service will never actually *know* if it is talking to a data proxy. Your service will just know that it is talking to a service interface that is fulfilling action promises.

See Also

- Recipe 3.3, "Sharing Domain Specifics via Published Vocabularies"
- Recipe 3.4, "Describing Problem Spaces with Semantic Profiles"
- Recipe 4.11, "Validating Data Properties at Runtime"
- Recipe 4.13, "Using Document Queries to Validate Incoming Messages"
- Recipe 5.7, "Publishing Complete Vocabularies for Machine Clients"
- Recipe 6.5, "Returning Metadata for Query Responses"
- Recipe 6.13, "Using Pass-Through Proxies for Data Exchange"

6.9 Improving Performance with Caching Directives

HTTP has a rich caching model built into the protocol. For data-centric services, this caching model can improve perceived performance and reduce latency—all without much effort.

Problem

When the service depends on remote data sources, it can lead to perceived poor performance due to slow networks and, at times, network-related failures. How can you improve the perceived performance of your service even when you depend on remote data-centric services that you do not control? How can the HTTP caching model be used on both the service and client side to improve reliability and availability of runtime services?

Solution

When your service interface relies on other, remote, services for data, the easiest way to improve perceived performance is by leveraging the HTTP caching models built into the protocol. That means service providers need to mark every response with caching metadata to help consumers understand the lifetime of the response. This

also means service consumers need to check for, and honor, the caching metadata in order to improve the quality of API responses.

Keep Your Data Close

Caches can be kept on the provider's machine, a third-party Content Delivery Network (CDN) machine, or even on the API consumer's machine. The closer the cache is to the client, the faster the perceived speed of your service.

For a better understanding of the HTTP caching model, see the HTTP specifications for caching in RFC 7234.

In this recipe, I'll only be covering the caching model metadata for HTTP 1.1 and later. Caching directives are also sprinkled throughout the HTTP header collection that work for HTTP 1.0 services: `Age`, `Expires`, and `Pragma`. See the HTTP caching specification for details.

For a full list of caching directives, see the HTTP Cache Directive Registry.

Response caching metadata for service providers

Service providers can mark their responses with caching metadata. Typically, this means indicating the caching scope (`public` or `private`), the maximum length the response can be held (`max-age`), and how responses should be handled (`no-store`, `no-cache`, `must-revalidate`, etc.). This works well for responses that don't change very often; for example, things like static lookup lists or written documentation that rarely changes.

If you are consuming a data-centric service API that does not use caching directives, try to encourage them to add caching support to their responses. This can be especially effective if you are in an enterprise setting where the data-centric services are authored and supported by another team within your company.

For cases where the response contains information that might change frequently, you can use directives that tell API consumers to check back with the origin server before replaying cached responses. Service interface providers can return the `ETag` along with `Cache-Control: must-revalidate` to tell API consumers that they should craft conditional requests (`GET` or `PUT/DELETE`) by returning the value of the `ETag` in the `If-Match` header.

Immutable Caching

A cache directive named `immutable` can be used to help API consumers better understand that the response has a long, dependable life. This is typically used for subresources like news photos accompanying text. Data-centric services can use this directive when returning long-lived responses like static lists, product images, etc. To learn more about how to use this directive, see RFC 8246.

This pattern also works well if service providers anticipate that API consumers will send updates (`PUT/POST/PATCH/DELETE`) for the same resource and want to make sure the action is not completed if the server's version of the resource has already been modified.

Caching metadata for API consumers

When sending requests to a service interface, API consumers can use caching metadata to qualify the type of response that the consumer is willing to accept. For example, including the directive `Cache-Control: max-age=600, min-fresh=300` in an API request tells the provider that the consumer wants to make sure the response is no older than 10 minutes (`max-age=600`) and has at least another 5 minutes of "freshness" left (`min-fresh=300`).

These kinds of caching directives make sense when the API consumer wants to be sure to get a response that can be held for a while in order to reduce "refresh churn" on the client side.

API consumers can also use caching metadata in the request to force service interfaces to deliver a "brand-new" response by using the `Cache-Control: no-cache` directive. This is handy when the API consumer wants the most recent resource representation in order to, for example, edit that record.

Clients can also use the `Cache-Control:max-stale, stale-if-error` directives to tell service interfaces it's OK to return a "stale" copy of the resource if that is the only one available. This is a good idea when the API consumer wants to fulfill a read-only request (no editing anticipated) and would rather get an old response instead of having to report a 4xx or 5xx status.

Example

What follows are examples of applying caching metadata to provider responses and client requests. There is also an extended example that walks through a scenario where both client and server are using caching metadata to improve both the responsiveness and quality of the interchange.

Example provider response caching metadata

Here is a simple response with caching metadata added:

```
**** REQUEST ****
GET /provinces/list HTTP/1.1
Host: api.example.org
Accept: application/vnd.collection+json
...

**** RESPONSE ****
HTTP/1.1 200 OK
Content-Type: application/vnd.collection+json
Content-Length: NN
Date: Tue, 15 Nov 2022 08:12:31 GMT
Cache-Control: public, max-age=600
...
```

This response caching metadata tells the API consumers that the response is cacheable by any proxy (`public`), and that it can be stored (and replayed) for up to 10 minutes following receipt of the response.

The following shows how a service interface can provide caching metadata that instructs API consumers to craft conditional requests (`GET` or `PUT`/`DELETE`) using the HTTP entity tag header and the `must-revalidate` directive:

```
GET /user/q1w2e3r4 HTTP/1.1
Host: api.example.org
Accept: application/vnd.siren+json
...

**** RESPONSE ****
HTTP/1.1 200 OK
Content-Type: application/vnd.siren+json
Content-Length: NN
ETag: "w/p0o9i8u7y6t5"
Date: Tue, 15 Apr 2022 11:12:13 GMT
Cache-Control: public, max-age=300, must-revalidate, stale-if-error
...
```

In this case, API consumers are told they can keep the current response for up to five minutes and, if asked for it, they first need to confirm with the backend server that both server and cache have the same version (via the `ETag` header) before returning the cached copy to the API consumer. Note the use of the `stale-if-error` directive,

which tells the cache holder that, in cases where the validation request fails (e.g., a network error), it is OK to return this copy of the response, even if the "freshness date" has passed.

Consumer request caching metadata

The following is an example of an API request where the consumer wants to make sure the response is no older than 10 minutes (`max-age=600`) and has at least another 5 minutes of "freshness" left (`min-fresh=300`):

```
**** REQUEST ****
GET /users/list HTTP/1.1
Host: api.example.org
Accept: application/vnd.hal+json
Cache-Control: max-age=600, min-fresh=300
```

API consumers can also use caching directives in the request to force service interfaces to deliver a "brand-new" response:

```
**** REQUEST ****
GET /users/q1w2e3r4 HTTP/1.1
Host: api.example.org
Accept: application/vnd.hal+json
Cache-Control: no-cache
```

Here is an example of an API consumer telling the service interface it is OK to send a "stale" copy of the resource if that is the only one available right now:

```
**** REQUEST ****
GET /users/q1w2e3r4 HTTP/1.1
Host: api.example.org
Accept: application/vnd.hal+json
Cache-Control: max-stale, stale-if-error
```

Extended caching metadata example

When service interfaces return caching metadata in HTTP responses, it is important that the API consumer honor those directives and use them as intended. For example, API providers might indicate the response can be locally cached for up to 10 minutes after receipt (`Cache-Control: max-age=600`). When true, the API consumer should save a copy of this response and replay that stored response whenever that resource is needed within the time window indicated.

Consider a case where there is a service that keeps track of `customer` data and one that keeps track of `order` data. Your service returns an aggregation of those two data sources in the form of a `customer-order` read-only service. As the API consumer for `customer` and `order` APIs, you should honor the caching metadata returned by these services.

Here's the request for a single record from the `customer` service:

```
**** REQUEST ****
GET /customer/e3r4t5y6 HTTP/1.1
Host: api.example.org
Accept: application/vnd.collection+json
Cache-control: no-cache

**** RESPONSE ****
HTTP/1.1 200 OK
Host: api.example.org
Content-Type: application/vnd.collection+json
Cache-Control: private, max-age=600, must-revalidate
ETag: "w/i8u7y6t5r4er3"
```

And here's the request for related content from the `orders` service:

```
**** REQUEST ****
GET /orders/filter?customer=e3r4t5y6 HTTP/1.1
Host: api.example.org
Accept: application/vnd.collection+json
Cache-control: max-stale

**** RESPONSE ****
HTTP/1.1 200 OK
Host: api.example.org
Content-Type: application/vnd.collection+json
Cache-Control: private, max-age=1800
ETag: "w/u7y6t5r4e3w2"
```

Note that, in the first case, the request to the `customer` service asks for a "fresh" copy of the resource (`Cache-Control: no-cache`). But in the case of the list of related records (the `order` service), the request says it will accept a stale response if that is all that is available (`Cache-Control: max-stale`). This is a common arrangement. Individual records (ones that might be updated) often need to be "fresh" copies, but related lists could be "not so fresh." However, when a single resource from the `order` service is needed, it would be wise to mark that request `Cache-Control: no-cache` in order to get the most recent one.

Discussion

It is good practice for all services to provide caching metadata in responses. This is especially important for data-centric services where it is likely that the responses (essentially, the data records) will be reused often.

The rate of expected data change (e.g., how often a record is modified) is a good guide for how long the resource can be cached. If the underlying data is not changed often—for example, a list of states or provinces, street addresses, etc.—then the caching period for this data can be relatively long: hours or days. But if the base data is

more volatile—say, the contents of a shopping cart—then the caching period should be very short: possibly only a few seconds.

API clients should use the `no-cache` directive sparingly. It is essentially a "cache-buster" that can add additional burden to service interfaces. If the client expects to *edit* the resource, it makes sense to use the `no-cache` directive to get the most recent copy of the data. Otherwise, the client should just allow the provider to send the default (cached, if available) response.

Your Mileage May Vary

An important caching-related header not covered in this recipe is the `Vary` header. This header can be used to indicate which HTTP elements (besides the URL, host, and method) should be considered when caching a response. For example, `Vary: Authorization` indicates that the response can only be replayed if the values of `Authorization` on subsequent requests match. Another common `Vary` directive includes things like `accept-language`, `content-type`, or other negotiable elements.

Providers should keep these things in mind when marking a response with `Cache-Control` directives to make sure the wrong representation is not "replayed."

In cases where the data is long-lived (static lists, stable documents, etc.), API consumers can use caching to build up their own local copy of commonly used data records. This is a kind of "low-fidelity" replication pattern. For more along these lines, see Recipe 7.14.

See Also

- Recipe 4.15, "Maintaining Your Own State"
- Recipe 5.14, "Increasing Throughput with Client-Supplied Identifiers"
- Recipe 5.16, "Providing Runtime Fallbacks for Dependent Services"
- Recipe 7.14, "Optimizing Queries with Stored Replays"

6.10 Modifying Data Models in Production

At some point you may need to update the data model your service is using to create resources. When that happens, you need to come up with a way to modify your data model without breaking any existing data consumers.

Problem

How can you safely update your service's data model after it has already been released into production? What aspects of the model can be changed without breaking API consumers? What happens if you update your model in a new release and need to roll back that model change?

Solution

The most effective approach for supporting data model changes for production data services is to design the data store to support changes from the very start. That means modifications are not "bolted on" or an "afterthought," but are actually part of the original design.

A good way to build in model changes for your data storage is to adopt a two-tier approach to modeling data through the use of *explicit and implicit data properties*. Explicit fields are part of the model or schema (e.g., `{"familyName" : "Quarkus"}`). Implicit fields are part of a name-value collection associated with the model (e.g., `{"nvp": [{ "name": "familyName", "value": "Quarkus" }]}`). When you create an array or collection property that can hold an arbitrary number of name and value elements, you build in the ability to add new properties to your data model without changing the existing model or schema.

A key advantage to this approach is that new properties can be added to the name-value collection without requiring schema updates for all your API consumers. It also means that, if you release a new edition of your service API that collects additional data fields and then need to back out that release, your data model does not need to change and you don't lose all the data collected in the new name-value pair fields. Then, when you fix any bugs and create a new release, you can take advantage of the previously collected data. Essentially, you're creating a data modeling pattern that supports both forward and backward compatibility.

Example

Usually, local data stores are modeled as a strongly typed data object. For example, the following is a `person` storage object:

```
{
  "givenName": "John",
  "familyName": "Doe",
  "age": 21
}
```

The `person` message can be defined by the following schema document:

```
{
  "$id": "https://api.example.org/person.schema.json",
```

```
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "title": "Person",
  "type": "object",
  "additionalProperties": false,
  "properties": {
    "givenName": {
      "type": "string",
      "description": "The person's first name."
    },
    "familyName": {
      "type": "string",
      "description": "The person's last name."
    },
    "age": {
      "description": "Age in years. Must be equal to or greater than zero.",
      "type": "integer",
      "minimum": 0
    }
  }
}
```

The JSON Schema examples used in this pattern all have their `additionalProperties` value set to `false`. XML-based schemas have this as the default, too.

Adding a new property

Now assume we want to add a new property—`middleName`—to this `person` object. If we add a new field in the schema, we run the risk of breaking existing API consumers. We'd need to make sure to roll out updates for all consuming services at the same time. This might be possible for services where all the API consumers and producers are maintained by the same team (or two teams in close coordination), but even this can be tricky. Instead, we can employ the name-value pair pattern to add the new property to the `person` object:

```
{
  "givenName": "John",
  "familyName": "Doe",
  "age": 23,
  "nvp" : [
    {"name" : "middleName", "value" : "Seymore"}
  ]
}
```

And here's the new schema for this model:

```
{
  "$id": "https://example.com/person.schema.json",
  "$schema": "https://json-schema.org/draft/2020-12/schema",
```

```
    "title": "Person",
    "type": "object",
    "additionalProperties": false,
    "properties": {
      "givenName": {
        "type": "string",
        "description": "The person's first name."
      },
      "familyName": {
        "type": "string",
        "description": "The person's last name."
      },
      "age": {
        "description": "Age in years which must be equal to or greater than zero.",
        "type": "integer",
        "minimum": 0
      },
      "nvp" : {
        "type" : "array",
        "items" :  { "$ref": "#/$defs/nvp"},
        "description" : "List of name/value pairs.",
      }
    },
    "$defs": {
      "nvp": {
        "type": "object",
        "required": [ "name", "value" ],
        "properties": {
          "name": {
            "type": "string",
            "description": "The name of the property."
          },
          "value": {
            "type":["number","string","boolean","object","array", "null"],
            "description": "The value of the property."
          }
        }
      }
    }
  }
```

Now, instead of a "strongly typed" schema for our data model, we have a "mildly typed" approach. Any properties we wish to add to the model can be placed in the `npv` array, and the resulting object will still pass validation for the published schema.

A single-access function

This two-tier approach can make it a bit challenging for services to manage when trying to locate a desired property. The question becomes: "I am looking for the `middleName` field. Is that an implicit or explicit property?" You can hide the dual nature of

your data model with a single access function that can look in both places automatically. Here's one example implementation in JavaScript:

```
// return a single property
// whether explicit or implicit
// args = {name:n,message:m,nameValuePair:p}
function find(args) {
  var a = args || {};
  var n = a.name || "";
  var m = (a.message || local.m) || {};
  var p = (a.nameValuePair || local.p ) || "nvp";
  var r = undefined;

  if(m==={} || n==="") {
    r=undefined;
  }
  else {
    if(m.hasOwnProperty(n)) {
      r=m[n];
    }
    else {
      if(m.hasOwnProperty(p)) {
        try {
          r = m[p].filter(function(i) {return i.name===n})[0].value;
        }
        catch {
          r = undefined;
        }
      }
    }
  }
  return r;
}
```

Now, whether the property is explicit or implicit, the same function call can be used:

```
console.log(find({name:"givenName", message:person}));
console.log(find({name:"middleName", message:person}));
```

This implicit model works for complex values, too:

```
{
  "givenName": "John",
  "familyName": "Doe",
  "age": 21,
  "nvp" : [
    {"name" : "hatsize", "value" : null},
    {"name" : "middleName", "value" : "Seymore"},
    {"name" : "nicknames", "value" : ["J","JJ","Johnboy","Jack"]},
    {"name" : "address", "value": {"street":"123 main", "city": "Byteville",
      "state": "MD", "zip": "12345"}}
  ]
};
```

Support for multiple schema versions

There's another advantage to this pattern when you use a single-access function, as shown previously. Since it does not matter where the property is located in the message, the same client application can be coded to support multiple schema instances of the `person` message.

For example, your service can update the data model to make `middleName` an explicit field:

```
{
  "givenName": "John",
  "middleName": "Seymore",
  "familyName": "Doe",
  "age": 21
};
```

And the same `find` function works as expected:

```
console.log(find({name:"middleName", message:person}));
```

Reverting models

Finally, in cases where you need to revert to a previous model/schema, you can move properties that are explicit in one model and represent them as implicit in another model with minimal side effects.

Discussion

This pattern is primarily designed to make it easy for services to safely support internal data model changes over time. It is not recommended that you expose these models directly via your service API. Instead, you should represent your resources using a well-defined media type (see Recipe 5.3 for details).

It may seem like a good idea to simply add new explicit fields to JSON-based objects and rely on the JSON Schema's default behavior of ignoring additional properties when validating objects. This, however, can lead to problems. First, JSON Schema and XML Schema behaviors are not the same. By default, XML rejects messages with unknown elements. Second, assuming that JSON Schema documents will always have their `additionalProperties` value set to `true` is dangerous. Some data consumer teams might write their own local JSON Schema that sets this value to `true` and start rejecting messages with unknown properties. Finally, by adopting this explicit/implicit model, you make it clearer to API consumers that some fields have been added. This makes it easier for consumers to ignore them when appropriate and gives them a clear path toward discovering new properties in the future.

If you end up supporting multiple model schemas at runtime (e.g., some previously implicit properties are upgraded to explicit), be sure to link to the correct revision of

the JSON (or XML) schema documents, since client applications may be relying on schema documents at runtime (see Recipe 4.7).

This recipe doesn't explain how to *store* the explicit and implicit data properties, since the details change depending on what data format you are using. JSON documents can be structured to fit the given example objects directly. However, in cases where data is stored using SQL tables, you can create a table to hold the explicit fields and then add the implicit properties in a standalone table (`nameValuePairs`), where each row has an index field pointing to the explicit row along with the name and value for that implicit property. See Figure 6-6.

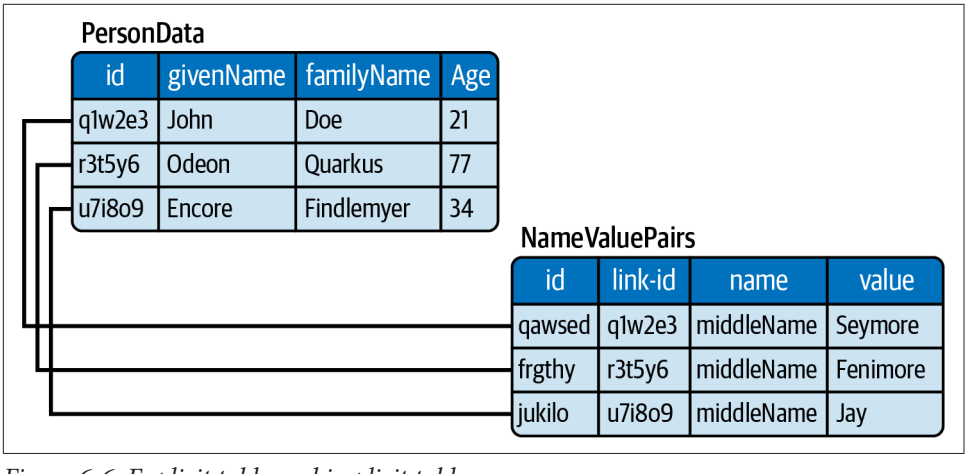

Figure 6-6. Explicit table and implicit table

Accepting unknown implicit fields can be a security risk. It's crucial that all services that attempt to read/write data check both the name and value contents to protect themselves from any attempt to place dangerous values in data storage.

It is important to remember that services that support writing data using this recipe will need to know which fields are explicit and which are implicit. An easy approach is to keep a list of explicit fields as a guide when writing a record. If any field appears that is not listed, you can assume it is an implicit field that should be added to the `nvp` collection. You may also be able to use a local schema document to guide the write operation.

See Also

- Recipe 3.10, "Designing for Extensible Messages"
- Recipe 3.11, "Designing for Modifiable Interfaces"

- Recipe 4.7, "Using Schema Documents as a Source of Message Metadata"
- Recipe 4.14, "Validating Incoming Data"
- Recipe 5.2, "Preventing Internal Model Leaks"
- Recipe 6.1, "Hiding Your Data Storage Internals"
- Recipe 6.8, "Ignoring Unknown Data Fields"

6.11 Extending Remote Data Stores

There are times when you want to use the data from another service even though that service does not store all the data properties you need. In cases like this, it may be possible to *extend* that remote data source with your own local property storage using an associative key.

Problem

How can you safely and effectively extend an existing data store (one that you don't control) with additional property values? When is this a good idea? How can you maintain data integrity between the remote and local data stores?

Solution

In cases where you want to extend an existing remote data store with additional values, you can establish a local data store that contains the additional values, and an associative key that links the two data stores together.

Example

To start, let's lay out an example remote and local data store and then make the association.

As an example, assume you are using a remote data store that keeps track of `user Accounts`. It has a `uniqueId` property along with a collection of other properties, such as `givenName`, `familyName`, `email`, etc.:

```
**** REQUEST ****
GET /users/q1w2e3r4 HTTP/1.1
Host: user-accounts.example.org
Accept: application/vnd.collection+json
...

**** RESPONSE ****
HTTP/1.1. 200 OK
Host : user-accounts.example.org
Content-Type: application/vnd.collection+json
Content-Length: XXX
```

```
{ "collection" :
  {
    "version" : "1.0",
    "href" : "http://user-accounts.example.org/users/q1w2e3r4",

    "links" : [...],
    "items" : [
      {
        "href" : "http://user-accounts.example.org/users/q1w2e3r4",
        "data" : [
          {"name": "uniqueId", "value": "q1w2e3r4"},
          {"name": "givenName", "value": "Marquis"},
          {"name": "familyName", "value": "Quarkus"},
          {"name": "email", "value": "user@example.org"}
        ]
      }
    ]
  }
}
```

Also assume you are creating a service that keeps track of the request history for users of your shopping site. You want to track things like `visitorId`, `pageUrl`, `dateTime`, `dwellTime`, etc.:

```
**** REQUEST ****
GET /history/p0o9i8u7/1 HTTP/1.1
Host: shopping.example.org
Accept: application/vnd.collection+json
...

**** RESPONSE ****`
HTTP/1.1. 200 OK
Host : shopping.example.org
Content-Type: application/vnd.collection+json
Content-Length: XXX

{ "collection" :
  {
    "version" : "1.0",
    "href" : "http://shopping.example.org/history/p0o9i8u7/1",

    "links" : [...],
    "items" : [
      {
        "href" : "http://shopping.example.org/history/p0o9i8u7/1",
        "data" : [
          {"name": "historyId", "value": "p0o9i8u7"},
          {"name": "pagerUrl", "value": "..."},
          {"name": "dwellTime", "value": "30000ms"},
          {"name": "dateTime", "value": "20230203T141529Z"}
        ]
      }
    ]
```

```
    }
  }
```

Finally, you want to be able to share this request history with users on demand. Essentially, you need to merge both the `userAccount` and `requestHistory` data properties.

A direct way to do this is to use the runtime URL of the remote data store (*http://user-accounts.example.org/users/q1w2e3r4*) as the associative key and add that to the local record:

```
**** REQUEST ****
GET /history/p0o9i8u7/1 HTTP/1.1
Host: shopping.example.org
Accept: application/vnd.collection+json
...

**** RESPONSE ****`
HTTP/1.1. 200 OK
Host : shopping.example.org
Content-Type: application/vnd.collection+json
Content-Length: XXX
```

```
{ "collection" :
  {
    "version" : "1.0",
    "href" : "http://shopping.example.org/history/p0o9i8u7/1",

    "links" : [...],
    "items" : [
      {
        "href" : "http://shopping.example.org/history/p0o9i8u7/1",
        "data" : [
          {"name": "historyId", "value": "p0o9i8u7"},
          {"name": "pagerUrl", "value": "..."},
          {"name": "dwellTime", "value": "30000ms"},
          {"name": "dateTime", "value": "20230203T141529Z"},
          {"name": "associativeKey",
            "value": "http://user-accounts.example.org/users/q1w2e3r4"}
        ]
      }
    ]
  }
}
```

Now, when you want to construct a resource that includes both your locally stored data and fields from the remote data source, you just need to use the `associativeKey` in your local store to retrieve the related data.

Discussion

It is possible that the remote service may not respond when you attempt to retrieve the associated resource. In this case, you may be able to use a local copy (see Recipe 6.9) instead.

It is also possible that the remote service is up and running but unable to find the resource you are requesting. For example, that resource might have been deleted. In this case, your service needs to either supply default data to replace the missing information, or return a 400-level response telling the calling application that the information is no longer available. If the local data is no longer useful when the remote resource is missing, you can delete your local data associated with that record.

Just because a remote service reports 404 does not mean that the data has been deleted. It may just be *temporarily* missing. Don't be too quick to delete your local data if the remote service reports a 4xx level response. If, however, the remote service responds with a `410 Gone`, you can be confident this condition is permanent.

It is not a good idea to create local data properties with the same name as properties in the remote data store. For example, if the remote resource has an `email` field that contains the user's personal email address (*marquis@example.org*) and you want to store a work email instead (*work@example.org*), don't create a local `email` property with the desired value. Instead, create a uniquely named local property (`workEmail`) and use that to hold your value.

Even if your local service displays a single resource that is a mix of both local and remote properties, it is not a good idea to try to enforce any data integrity for the combined record. For example, you might want to enforce a rule that any local resource that has an `email` property also *must* have a `fullName` property. If one of the properties is part of the remote resource (`email`) and the other is part of the local resource (`fullName`), you can't be assured that both records will enforce the same rules.

But My Schema!?!

In cases where service interfaces are returning responses made from multiple sources, using schema documents to validate messages gets complicated quickly. This is especially true if you are using a storage service that supports field selection (e.g., OData, GraphQL, etc.). See Recipes 4.12 and 4.13 for more on using schema documents for APIs.

It is rarely a good idea to use your local resource to hold copies of the values of some remote resource properties. The values of the remote properties might change, and your local service will not know about the updates. See Recipes 6.9 and 6.13 for more on this topic.

Over time, it is likely that your local data store will contain resources with "broken" associations—the `associativeKey` URL returns a 4xx or 5xx status. If needed (e.g., to recover space), you can create a job or script that crawls your local data store and removes any local records that no longer have remote associations. However, it is often important to retain your local data (to keep a complete history, etc.), and in that case, your local service should just ignore the association and/or replace the values with the local default (e.g., "No Longer Available," etc).

See Also

- Recipe 3.10, "Designing for Extensible Messages"
- Recipe 3.11, "Designing for Modifiable Interfaces"
- Recipe 4.11, "Validating Data Properties at Runtime"
- Recipe 4.12, "Using Document Schemas to Validate Outgoing Messages"
- Recipe 4.14, "Validating Incoming Data"
- Recipe 6.1, "Hiding Your Data Storage Internals"
- Recipe 6.9, "Improving Performance with Caching Directives"

6.12 Limiting Large-Scale Responses

For data services that get a large amount of traffic and/or serve up lots of content, limiting the size of the responses can be an effective way to support API clients "at scale." Conversely, failing to regulate the size and/or count of data returned from queries against large data sets can easily bog down your data service and, if this is a critical data set, adversely affect many other dependent services.

Problem

How can you make sure query responses from data-centric services remain responsive even as the data set grows? What are common controls to the size of the data collections returned? When is it important to impose limits on returns from large data sets?

Solution

The best way to ensure that your service interface for data-centric services does not suffer performance problems as the data set grows is to limit the number of records returned in HTTP responses. The proper way to do this is to set (and properly document) a *default* maximum records value for all data queries. Even if the backend data service does not already set limits on response collections, it is a good idea for you to implement response limits for the service interface that stands between the API client and the underlying service.

This recipe addresses the ability to limit returns in a query to underlying data stores. You can also use the page navigation recipe (Recipe 7.11) to implement a more interactive way to control query returns for large data sets.

Most data engines support the ability to limit the number of records returned in a single response. For example, OData supports the `$top` directive and the `maxpagesize` setting. GraphQL supports the use of `first:nn`. SQL supports the `TOP` and `LIMIT` directives. And Lucene has the `rows` query parameter. So, for cases where your service interface is crafting the query for the underlying data service, you can make sure to employ these settings in your data requests.

You should *always* include a maximum value in your queries to data storage, and you should always replace any client-supplied limit values that are out of range (e.g., set too high). Sending data queries without indicating a maximum record count can introduce performance and possibly security problems.

You can make the return limit a part of the query that API clients commit (in other words, allow API clients to determine the limit value). You can also set the limit value within the service interface *before* you send the query to the underlying service (essentially you edit the client query before sending it). It is also possible (but a bit more work) to support both options. That is, to allow API clients to set the limit value and, once submitted to the service interface, that code can look for the limit value. If the limit value does not exist, the code can insert the default value; or if the limit value does exist but is outside of acceptable limits (e.g., set to 10,000,000), the service interface code can modify the value accordingly.

In cases where the underlying data source does not support a maximum count setting, you will need to implement your own return limits at the service interface level. That means accepting the return collection from the underlying data source and then, if needed, truncating the collection at the maximum record count.

Whether you communicate the limits via the actual query to the data store or implement the limit by truncating the collection returned to the API caller, you should *always* indicate the use of the limit value in the query metadata (see Recipe 6.5) returned to the API caller.

Example

There are two ways to implement return count limits on data-centric service interfaces:

`DirectLimit`
: Include the limit directive in the data query sent to the data store.

`TruncatedLimit`
: Truncate the resulting data collection within the service interface.

Next up are examples of each method.

Implementing direct limits

The easiest way to support direct limits is to simply allow API client applications to send in the proper directives with each data query. For example, here's a query using the Solr search engine. Note the use of the `limit=100` setting:

```
**** REQUEST ****
GET /persons/search/?q=%22Bob%22&rows=100 HTTP/1.1
Host: api.example.org
Accept: application/vnd.collection+json

**** RESPONSE ****
HTTP/1.1 200 OK
Content-Type: application/vnd.collection+json
Q-Status=success
Q-Returned=100
Q-Count=10000
...
```

In this example, the `rows=100` query parameter was sent by the client application to the service interface. But there will be times when the client application does not send the limit value. In that case, it is up to the service interface to send the limit value to the data store. Here is an example using the OData engine:

```
**** REQUEST ****
GET /persons/search/?$search=%22Bob%22 HTTP/1.1
Host: api.example.org
Accept: application/vnd.collection+json

**** RESPONSE ****
HTTP/1.1 200 OK
```

```
Content-Type: application/vnd.collection+json
Q-Status=success
Q-Sent=$search=%22Bob%22
Q-Executed=$search=%22Bob%22$top=100
Q-Returned=100
Q-Count=10000
...
```

Note the use of the `q-sent` and `q-executed` query metadata values (see Recipe 6.5). In cases where the service interface modifies the client query before sending it to the data store, these two metadata properties should be returned as either HTTP headers or as part of the response body.

Implementing truncated limits

There will be times when the service interface has to communicate the maximum return limit from a data query. Sometimes this maximum value is hardcoded into the data store, and sometimes the data store has no maximum limits at all. In both cases, it is important for the service interface to protect itself against return sets that are "too large" (whatever that means for the client making the query). This usually means adding support for *truncating* the data collection returned from the data store (in the API code) before sending the results to the client application.

Implementing data set truncation at the service interface layer will vary based on the programming languages, return formats, and other factors. A crude solution is to simply walk through the returned collection and retain the maximum number of records to return:

```
function executeQuery(dataStoreAddress, dataQuery) {
  var ix=0;
  var maxLimit=100;
  var responseCollection = [];
  var dataCollection = httpRequest(dataStoreAddress, dataQuery);
  for (let item of dataCollection) {
    if(ix>maxLimit) {
      break;
    } else {
      responseCollection.push(item);
    }
    ix++;
  });
  return responseCollection;
}
```

In some cases, you may be able to retain the complete collection and implement local page navigation against the saved data collection. See Recipe 7.11 for details.

Once the service interface has created its own local copy of the collection (limited to the maximum allowed record count), the API can return the results shown in the direct limit example:

```
**** RESPONSE ****
HTTP/1.1 200 OK
Content-Type: application/vnd.collection+json
Q-Status=truncated
Q-Returned=100
Q-Count=10000
...
```

Note the use of the `q-status=truncated` metadata property to help the API client understand the meaning of the results.

Discussion

The safest approach for implementing data request limits is to set the limit in the API code using the parameters in the underlying query language. If you allow client applications to set the limit value, *always* check it to make sure it does not fall outside acceptable limits (e.g., set to `-100` or `1000000`).

Using the truncated limit approach has its drawbacks. For example, you do not get to control how large a data set may be returned to you from the underlying data source. Even if your API limits response collections to 100 records, you may still need to wait for the underlying (unlimited) data source to search for and return 100,000 records before you can implement your internal truncation routine. In the case of very large return sets, your API performance is likely to suffer.

Limits Are Not Pages

It is important to not confuse query limits (like the ones discussed here) with page sizes (covered in Recipe 7.11). For example, you might set your API query limit to 1,000, and the client application might set its page-size value to 100. In that case, the client will be expecting no more than 100 records in any single returned response.

Whenever you modify a direct limit query or invoke a truncated limit on a return set, you should communicate this information back to the requesting application (preferably using query metadata [Recipe 6.5]).

See Also

- Recipe 4.10, "Supporting Links and Forms for Nonhypermedia Services"
- Recipe 4.15, "Maintaining Your Own State"

- Recipe 5.11, "Supporting Service Health Monitoring"
- Recipe 5.16, "Providing Runtime Fallbacks for Dependent Services"
- Recipe 6.5, "Returning Metadata for Query Responses"
- Recipe 6.6, "Returning HTTP 200 Versus HTTP 400 for Data-Centric Queries"
- Recipe 7.11, "Enabling Standard List Navigation"
- Recipe 7.15, "Synchronous Reply for Incomplete Work with 202 Accepted"

6.13 Using Pass-Through Proxies for Data Exchange

There are times when services want to edit a subset of the fields in a stored record (e.g., the `address` data in a `person` record). In these cases, it is important to make sure that changes to the subset of fields does not result in invalidating the data in the complete record. At the same time, "upstream" clients do not need to know the details of the source ("downstream") services. This is where the pass-through proxy recipe can help.

Problem

How can you maintain data storage integrity when other services want to edit just a portion of your storage record? When updating a partial set of data fields in a record (e.g., the `address` fields in a `person` record), what can you do to make sure the full set of fields in the record contain no conflicts? And when you discover internal data inconsistencies, what response should you return to the service attempting the invalidate update?

Solution

A good rule to follow is to *always* pass complete storage data records back and forth between services. This is the easiest, most reliable way to ensure information integrity of the record. Even if there is a service that wants to edit just a portion of the record (for example, the `address` portion of a `person` record), that service should be sent a complete record, modify the portion of the record it wants to work with, then return the entire record to the data service for validation and storage.

In cases where the consuming service is an intermediate (or pass-through) service between two other parties, (e.g., an `address` service that exchanges data with a `person` service), that service should still exchange complete records with the `person` service. This can be done by reading a `person` record, extracting the `address` portion to send along, holding on to the related `person` record, and, upon receiving the updated `address` information, loading the `address` data into the held `person` record and sending that back to the `person` service. Of course, the `address` service needs to be

prepared for a failed write to the `person` service and report that back to the `address` consumer when appropriate.

It's Turtles All the Way Down

The example given here (`person` and `address`) is a reminder that API consumers can never be certain whether they are talking to a "pass-through" service or "storage" service. In fact, you can imagine another service called the `telephone` service that allows consumers to update the telephone value of an `address` record. Now we have three layers to deal with (`telephone`, `address`, and `person`). For all we know, the `person` service may be changed in the future from a "storage" type to a "pass-through" type. That would be *four* layers of data exchange.

Services that are built to edit subsets of data (`address` for a `person`) may be limited to updates and not be able to support create or delete actions. For example, the only way to create an additional `address` or remove an existing `address` for a `person` is to use the `person` service directly.

Note that it is a good idea to keep track of update metadata when exchanging subsets of records. See Recipe 6.5 for details.

Example

Figure 6-7 is a sequence diagram that gives you an omniscient view of how the three services interact in our example. Of course, each participant (`apiClient`, `addressAPI`, and `personAPI`) only knows about its direct neighbors. For example, `apiClient` only knows about `addressAPI`, `personAPI` only knows about `addressAPI`, and `addressAPI` knows about both other parties.

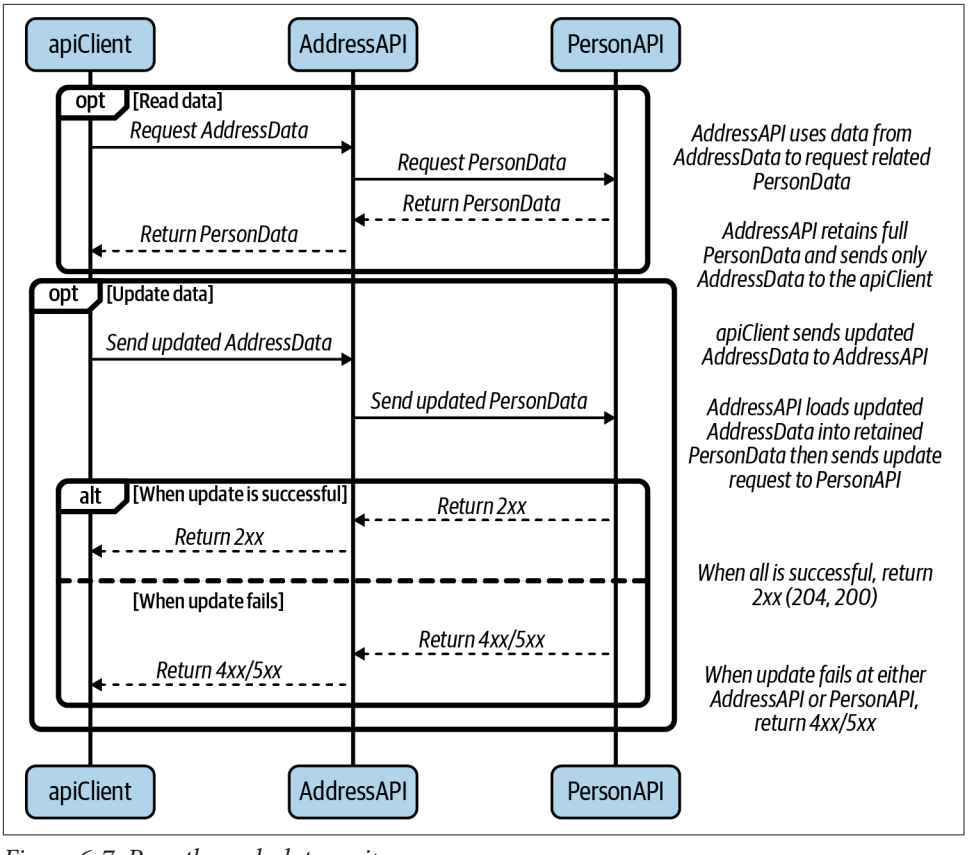

Figure 6-7. Pass-through data recipe

The following is an example of an `address` service interacting with the `person` service to support updating the `address` subset info for a `person`.

First, the `address` service receives a request for a record:

```
**** REQUEST ****
GET /q1w2e3r4 HTTP/1.1
Host: api.address.org
Accept: application/vnd.collection+json
```

The `address` service then needs to contact the `person` service for a record:

```
**** REQUEST ****
GET /q1w2e3r4 HTTP/1.1
Host: api.person.org
Accept: application/vnd.collection+json

**** RESPONSE ****
HTTP/1.1 200 OK
```

```
Content-Type: application/vnd.collection+json
Content-Length: nn
ETag: "w\p0o9i8u7y6"
...

{"collection" : {
  "title": "Person Service",
  "links": [...],
  "items": [
    { "href": "https://api.person.org/q1w2e3r4",
      "data": [
        {"name": "id", "value": "q1w2e3r4"},
        {"name": "fullName", "value": "Mork Markleson"},
        {"name": "streetAddress", "value": "123 Main St"},
        {"name": "cityTown", "value": "Byteville"},
        {"name": "stateProvince", "value": "MD"},
        {"name": "postalCode", "value": "12345"}
      ]
    }
  ]
}}
```

Now the `address` service can store a copy of the `person` record locally (be sure to store the response headers, too) and then respond to the caller with the subset of `address` fields:

```
**** RESPONSE ****
HTTP/1.1 200 OK
Content-Type: application/vnd.collection+json
Content-Length: nnn
ETag: "w\y6t5r4e3w2"

{ "collection" : {
  "title": "Address Service",
  "links": [...],
  "items": [
    { "href": "https://api.address.org/q1w2e3r4",
      "data": [
        {"name": "id", "value": "q1w2e3r4"},
        {"name": "street", "value": "123 Main St"},
        {"name": "municipality", "value": "Byteville"},
        {"name": "region", "value": "MD"},
        {"name": "zipCode", "value": "12345"}
      ]
    }
  ],
  "template": {
    "rel": "edit",
    "data": [
      {"name": "id", "value": "q1w2e3r4"},
      {"name": "street", "value": "123 Main St"},
      {"name": "municipality", "value": "Byteville"},
      {"name": "region", "value": "MD"},
```

```
        {"name": "zipCode", "value": "12345"}
      ]
    }
  }}
```

There are a couple of things to notice here. First, the `address` service has generated its own `ETag` value, one that matches the `address` representation. This can be used in case more than one party attempts to update this record (see Recipe 6.2 for details on the lost update problem).

In this example, both the `address` service and the `person` service use the same `id` value *and* use that value in the URL. This is not always the case and should be taken into account. You may need to keep track of two different identifiers and two separate URLs when working as a pass-through service.

Second, note that the field *names* used by the `address` service do not need to be the same as those used by the `person` service. It is handy if they *are*, but quite often we don't control these things and a translation step is needed to pass data from one party to the next.

Next, the `address` service can wait for someone to send an update record back:

```
**** REQUEST ****
PUT /q1w2e3r4 HTTP/1.1
Host: api.address.org
Content-Type: application/vnd.collection+json
Content-Length: nn
If-Match: "w\y6t5r4e3w2"
...

"template": {
  "data": [
    {"name": "id", "value": "q1w2e3r4"},
    {"name": "street", "value": "123 Main St, Apt 3G"},
    {"name": "municipality", "value": "Byteville"},
    {"name": "region", "value": "MD"},
    {"name": "zipCode", "value": "12345-6789"}
  ]
}
```

Then the `address` service must load the data from the caller into its own local copy of the `person` record and pass that along to the `person` service:

```
**** REQUEST ****
PUT /q1w2e3r4 HTTP/1.1
Host: api.person.org
Content-Type: application/vnd.collection+json
Content-Length: nn
```

```
If-Match: "w\p0o9i8u7y6"
...

"template": {
  "data": [
    {"name": "id", "value": "q1w2e3r4"},
    {"name": "fullName", "value": "Mork Markleson"},
    {"name": "streetAddress", "value": "123 Main St, Apt 3G"},
    {"name": "cityTown", "value": "Byteville"},
    {"name": "stateProvince", "value": "MD"},
    {"name": "postalCode", "value": "12345-6789"}
  ]
}

**** RESPONSE ****
HTTP/1.1 204 No Content
```

Assuming all goes well, the `person` service returns a 2xx response to the `address` service, and the `address` service does the same to its caller.

If, however, there was a problem writing to the `person` service, the appropriate status code will be returned to the `address` service and then passed along to the caller:

```
**** RESPONSE ****
HTTP/1.1 412 Precondition Failed
Content-Type: application/vnd.collection+json
Content-Length: nn

{"collection": {
  "title": "Person Service",
  "links": [...],
  "error": {
     "title": "Unable to update record",
     "code": "http://api.person.org/reasons/precondition",
     "message": "The record you are trying to update has been changed.",
     "data" [{"name": "id", "value": "q1w2e3r4"}]
  }
}}
```

You may need to alter the `error` information depending on the type of error and where in the chain (storage service, pass-through service) you are reporting.

Discussion

The concept of a pass-through data interface is only important to the interface that is acting as a proxy between two services (as in the preceding `address` service example). Consumers of pass-through proxies don't need to know anything about the pass-through nature of the target service. The same goes for any service interface that is *behind* the pass-through proxy.

As an alternative to the pass-through proxy recipe, services can ignore unknown data fields (see Recipe 6.8).

A key element of this recipe is to be sure that pass-through proxies keep track of the connection between the "downstream" (source) record and the "upstream" (exposed) record. To do this, be sure to always store a complete copy of the "downstream" record, including metadata fields such as `ETag` and other related resource integrity information. The best way to do that is to store the complete HTTP response, not just the body. See Recipe 6.5 for details.

As an implementation detail, it is always a good idea for the pass-through proxy to use a translation step to handle any differences between the data property list for the proxy interface and the data property list for the source service:

```
function fetchPerson(url,propertyMap) {
  var personRecord = person.read(url);
  var addressRecord = mapProperties(personRecord, propertyMap)
  return addressRecord;
}
```

The `mapProperties(personRecord, propertyMap)` step does the work of pulling the expected fields from the source ("downstream") record and constructing a valid exposed ("upstream") record. In some cases this is not just a one-to-one renaming of data properties (`person.givenName = +address.firstName`). In some cases, fields are combined or split between interfaces:

```
var address.fullName = person.familyName+ ", "+ person.givenName
```

Source Today, Pass-Through Tomorrow

Since we don't control all the services on the web, we can't always know whether an interface is connected to a data source service or a data pass-through service. In fact, a service might start as a data source and later become a pass-through. From the interface point of view, it doesn't matter. Interfaces should continue to make promises about message exchanges independent of the implementation details of the services behind that interface.

Pass-through proxies need to be prepared for cases where a poorly managed "downstream" source will change its response (e.g., add/remove/rename data fields). This can't be prevented, but you can mitigate the problem by always inspecting the incoming downstream record to make sure it contains the data properties the "upstream" interface has promised. If not, the pass-through proxy should return a `502 Bad Gateway` response with a body indicating the problem:

```
**** REQUEST ****
GET /q1w2e32r4 HTTP/1.1
Host api.address.org
Accept: application/vnd.collection+json
...

**** RESPONSE ****
HTTP/1.1 502 Bad Gateway
Content-Type: application/vnd.collection+json
Content-Length: nn
...
{"collection": {
  "links": [...],
  "error": {
    "title": "Invalid Response",
    "code": "SRC-077",
    "message": "Data source is missing the [givenName] data property"
  }
}}
```

Note that in this example, only the details related directly to the current interface exchange (`client` and `address` service) are included in the status report. There is no mention of the "downstream" source (`person` service).

See Also

- Recipe 3.4, "Describing Problem Spaces with Semantic Profiles"
- Recipe 3.10, "Designing for Extensible Messages"
- Recipe 4.3, "Coding Resilient Clients with Message-Centric Implementations"
- Recipe 5.7, "Publishing Complete Vocabularies for Machine Clients"
- Recipe 5.17, "Using Semantic Proxies to Access Noncompliant Services"
- Recipe 6.1, "Hiding Your Data Storage Internals"
- Recipe 7.20, "Using Workflow Proxies to Enlist Noncompliant Services"

CHAPTER 7
Hypermedia Workflow

Productivity is never an accident. It is always the result of a commitment to excellence, intelligent planning, and focused effort.

—Paul J. Meyer

For many, the ultimate goal of creating service interface APIs is to foster flexible and robust *workflow* solutions that mix existing services in unique ways to solve new problems. In fact, this goal is embodied in the keystone principle discussed in "Shared Principles for Scalable Services on the Web" on page 16. The process of linking services together is not a great challenge when these services were designed from the start to work in concert. However, enlisting services that were not built as a group, know nothing of each other, and are oblivious to the solution being created is a much bigger challenge. This second approach depends on important implementation details that make a number of activities possible, such as sharing solution state, progressing through multiple steps to accomplish a goal, and tracking/displaying that progress. Anytime you mix service interfaces together, there is a chance of execution errors that might require rerunning some steps, undoing others, and knowing when this is needed.

Often, we build our own services as a related collection. In fact, if the same team is building more than one service, those services are bound to have hidden dependencies that are difficult to untangle. Keeping this tangle at bay takes discipline, design, and determination.

The bad news is that designing and supporting successful workflow on the web requires additional effort. The good news is that, as long as you implement each individual service with a simple API and establish a general processing and tracking model that all services support, you can create a resilient, flexible, and evolvable workflow environment that clients can safely use to solve new and interesting problems.

For more background on the important elements of a well-designed and -implemented workflow on the web, see "Empowering Extensibility with Hypermedia Workflow" on page 46.

Supporting a hypermedia workflow (see Figure 7-1) means focusing on common patterns like page navigation, works in progress, most recently used, etc. It also means supporting a generalized hypermedia-driven workflow via jobs and tasks, along with some optimization techniques to keep things flowing. Finally, there is a second-level set of activities centered around managing the workflow process itself. These four elements (patterns, generalized flow, optimization, and management) make up a healthy hypermedia workflow system.

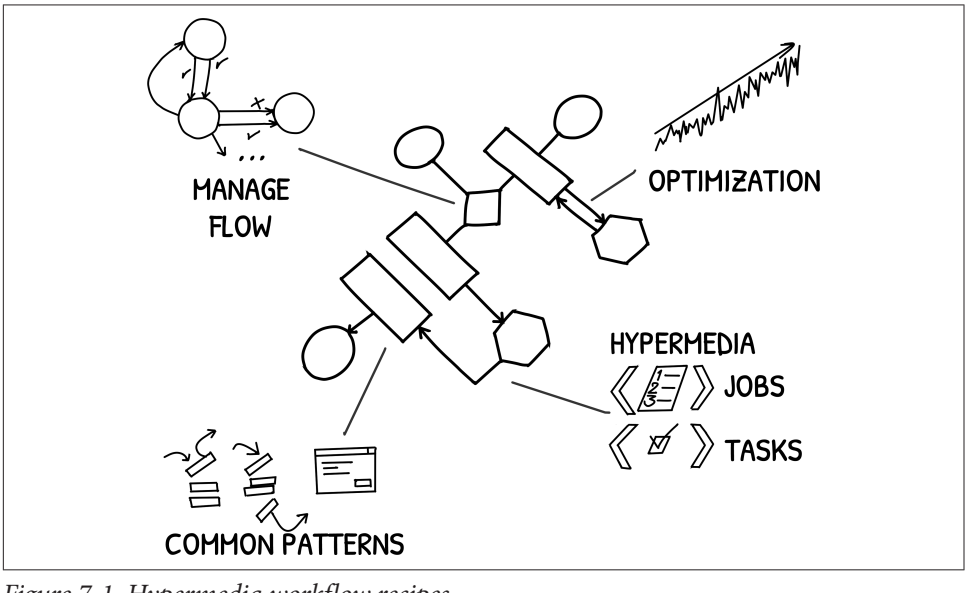

Figure 7-1. Hypermedia workflow recipes

This chapter contains two types of recipes. The first type is a set of common workflows like list navigation, work-in-progress patterns, forms handling, and similar patterns (Recipes 7.8 to 7.14). You can use these prototype patterns as a guide when you create your own semantic profiles (Recipe 3.4) for your service interfaces.

The second set of recipes here describe a general workflow interaction language that you can use to enlist selected independent services and track the progress of these multiple-service solutions. This is a much more abstract set of patterns that cover everything from how you can design in workflow support for your services to the actual implementation of workflow-compliant services.

Finally, you'll find several entries here that detail general principles and patterns for dealing with coordinating and managing these services. These include dealing with errors, rollbacks, and even some internal patterns to help scaling and proxying other service interfaces.

You'll find that many of the recipes in this chapter depend upon, or call back to, other recipes from earlier in the book. If you skipped over those chapters, this is a good time to follow the threads back to see where some of these workflow-solution recipes have their roots.

7.1 Designing Workflow-Compliant Services

A shared goal of many service interface designers is the ability to easily connect multiple services to form a workflow. These are often called *composable* services. This recipe describes a set of basic design features every workflow-compliant (composable) service should have.

Problem

What does it take to design a workflow-compliant service? What are the key features every composable service interface shares? How can you make it safe, cheap, and easy to implement workflow using these composable services?

Solution

The key to designing workflow-compliant services is to offer a consistent set of actions and make it easy to share state data among services. These elements—shared actions and shared state—are at the heart of stable, composable service interfaces.

In addition to a shared set of actions and state data, workflow-compliant services support shared identifiers for the job (as `correlation-id`) and as each task within the workflow job (as `request-id`).

Workflow actions

In hypermedia-driven services, actions are expressed at input forms. Each form describes all that is needed to complete the action: the URL, the HTTP method, the supported media types, and the complete set of inputs.

See Recipe 7.12 for details on how to design better forms.

While actions can be very specific (e.g., `onboardCustomer`, `computeSalesTax`, etc.), they can also be generic (`writeRecord`, `filterData`, etc.), but they *must* be completely described with all the inputs and related HTTP metadata included (see the "Example" section).

The list of actions also needs to support the following:

Execute
: The actual work to be done (e.g., `applySalesTax`).

Repeat
: Repeat the work again in an idempotent manner (e.g., `applySalesTax` can be safely repeated and still be the same expected results).

Revert
: The ability to undo a completed action (e.g., `revertSalesTax`).

Continue
: The ability to stop a workflow and then, after some pause, continue where you left off to complete the work.

Rerun
: The ability to start from the beginning of a set of tasks in a workflow and rerun all the steps even if the workflow has been run before.

It is a good idea for all workflow-compliant services to support all these actions, even if the action doesn't "do anything" (e.g., no support for Revert). That will make supporting them easier over time.

All these actions *should* have an associated URL and an associated FORM. The first three actions apply to each step (referred to here as a *task*) within a workflow. The last two items apply to the complete set of tasks (called a *job*).

Shared state

Two other actions that are needed for workflow-compliant service interfaces are:

ReadState
: The ability to load up related state properties for use by each *task* in the *job*.

WriteState
: The ability to store the related state properties for use by other *tasks* within the same *job*.

See Recipe 7.2 for details on how to handle shared state for HTTP workflows.

Both actions need an associated URL that points to a shared HTTP resource. This resource URL should be passed to each *task* in the workflow. It is up to each service to know how to recognize state values in the state resource and, when needed, update existing properties or add new properties to the state resource. Often, the properties in this resource are used to fill in the FORMS exposed by the composable service's actions.

For an example of a generic workflow engine via a job control language, see Recipe 7.6.

Correlation IDs for tasks and jobs

Each workflow job and each task within that job needs unique identifiers, typically a UUID or some other globally unique value. The `jobID` value should be used as the correlation identifier. This can be passed in the HTTP `correlation-id` header. The `taskID` value should be used as the request identifier. This can be passed in the HTTP `request-id` header. Like the `correlation-id`, the `request-id` should be included in both the HTTP request and the HTTP response.

The `correlation-id` and `requestid` headers are not standard, registered header values. Your organization might use some other header values or possibly even a different way to properly track workflow jobs and tasks. Whatever your organization does, the key to success is consistency and good documentation.

Workflow-compliant services support both `correlation-id` for the job and `request-id` for each task in the job.

Example

Consider a composable service that computes sales taxes for a shopping cart of goods. This service (along with other workflow-compliant services) can be used as a task in a job to handle checking out from a shopping site. The following is an imaginary DSL example of what that might look like:

```
*** Checkout Job
***
READ sharedState WITH urlState
EXECUTE shoppingCartService->checkOutForm WITH sharedSTATE
IF-NOT-OK EXIT
EXECUTE salesTaxService->applyTaxesForm WITH sharedSTATE
IF-NOT-OK EXECUTE shoppingCartService->revertCheckoutForm WITH sharedSTATE
STORE sharedState WITH urlState
EXIT
***
*** End Job
```

In this example, the `sharedState` is loaded into the client application. That application then attempts to execute the `shoppingCartService→checkOutForm` and the `salesTaxService→applyTaxesForm`, and then store the resulting `sharedState`. Note the support for the `shoppingCartService→revertCheckoutForm` in case something goes wrong when applying the sales taxes.

Discussion

Any service interface that offers an action that writes data to the shared state *should* offer a way to reverse that action. This may be as simple as restoring the old values in the shared state resources (as in our `applySalesTax` example) or as complex as launching another job to undo some set of steps related to the action that needs to be reversed. See Recipe 7.6 for more discussion on handling related jobs.

Service interfaces for each task *should* support calls to Repeat and Revert even if the Execute action has not yet been called. For example, calls to Revert *should not* result in an error (unless the actual revert action experiences a problem). This means services should "know" whether the Execute action has been previously completed. If yes, the calls to Revert will likely require some processing. If the Execute action was not yet completed, calls to Revert should just return `200 OK` without any processing. Think of Revert like a `DELETE` action. Deleting an already-removed resource doesn't create any new problems, so a return of `200 OK` is an acceptable response.

Revert Them All

If any task fails or times out (e.g., reaches `taskMaxTTL`), the *job* is invalid and the Cancel action should be called. This should result in a call to Revert all the tasks in the job. This vastly simplifies implementing cross-service rollbacks or undo (see Recipe 7.17).

Support for the Repeat of an action should be easy to handle since (as discussed in Recipe 3.6) all service actions should be idempotent. If you find you're having trouble safely repeating an action, you probably have a nonidempotency problem to untangle.

Support for shared state means that each job has a state document that is passed between tasks within the job. It might seem like a better idea to use a database record for shared state, but it is not. Shared state should be supported as a standalone HTTP resource that supports idempotent updates and possibly varying representation media types. This is the only set of properties that should be shared between services in a job. See Recipe 3.7 for more on this topic.

If there are many steps in a job (some of which might take lots of time), and that job gets stopped for some reason (e.g., problems with one of the tasks that needs to be fixed by a person), then the ability to Continue where the job left off is very useful. It can save time and computing resources.

Support for the Rerun action is handy when you want to force-restart a job that has multiple tasks. For example, you might find that after running a job, you notice that the results are not quite what you expected. You might be able to fix some state information and then rerun the complete job to get the proper results.

Although not covered here, you can create job *templates* where the tasks are the same but the contents of the shared state resource vary. This makes it possible to quickly define and run jobs based on configuration data (via the shared state resource). If you plan to do a lot of this kind of work, see Recipe 7.6 for an example of a generic RESTful job control language that makes defining and running jobs easier.

See Also

- Recipe 3.4, "Describing Problem Spaces with Semantic Profiles"
- Recipe 4.4, "Coding Effective Clients to Understand Vocabulary Profiles"
- Recipe 5.16, "Providing Runtime Fallbacks for Dependent Services"
- Recipe 6.13, "Using Pass-Through Proxies for Data Exchange"
- Recipe 7.2, "Supporting Shared State for Workflows"
- Recipe 7.6, "Supporting RESTful Job Control Language"
- Recipe 7.16, "Short-Term Fixes with Automatic Retries"
- Recipe 7.17, "Supporting Local Undo or Rollback"

7.2 Supporting Shared State for Workflows

Workflow-compliant services need to support the ability to share state properties between services. This recipe defines a simple and effective way to share state without the need to share domain-specific data or data models.

Problem

How can you design services to safely share data between each other without creating unnecessary tight coupling to data models or domain-specific service information? What does a minimum design for shared state look like?

Solution

The most straightforward way to share data between services is to use a *shared state resource*. This is a standalone HTTP resource that contains all the data related to a workflow job. Each workflow-compliant service that is responsible for a task in the job should be able to read and, in some cases, write to that shared state resource.

Usually, this shared state resource is used by services as a source for filling in HTTP FORMS that describe an action (e.g., `onboardNewUser`). Sometimes, the results of an action (e.g., `applySalesTax`) are written to the shared state resource.

Once the job is completed, this shared state resource should be archived for possible later reference.

This shared state resource is not the same as the workflow progress resource (see Recipe 7.7). You will need both when implementing a robust HTTP workflow system.

An easy way to create shared state resources is to use a unique identifier (usually the job identifier) as the public URL of the resource:

```
<link rel="sharedState" href="http://api.example.org/shared-state/u7y6t5r4e3 />
```

This URL should be passed around with each HTTP request and response related to the job being executed.

Example

You can pass the URL for the shared state resource as part of the HTTP header collection:

```
**** REQUEST ****
GET /onboarding/job1
```

```
Accept: application/vnd.collection+json
Link: <http://api.example.org/state/q1w2e3r4t5>>;rel="sharedState"

**** RESPONSE ****
200 OK
Content-Type: application/vnd.collection+json
Length: XX

{"collection" : {
  {"title": "Pricing Update Job q1w2e3r4t5",
  "links" : [...],
  "items" : [...]
}}
```

Or you can pass the shared state URL as part of the message body:

```
**** REQUEST ****
GET /onboarding/job1
Accept: application/vnd.collection+json

**** RESPONSE ****
200 OK
Content-Type: application/vnd.collection+json
Length: XX

{"collection" : {
  {"title": "Pricing Update Job q1w2e3r4t5",
  "links" : [
    {"rel": "sharedState", "href": "http://api.example.org/state/q1w2e3r4t5"}
  ],
  "items" : [...]
}}
```

The shared state resource can be "primed" with data before the job is started. For example, if you are running a job to update the prices of a select set of items in your warehouse, you can load an array of product identifiers along with price increase information into the shared state resource:

```
{"collection" : {
  "title" : "Shared State for Job ID q1w2e3r4t5",
  "links" : [
    {"rel":"read, sharedState",
      "href": "http://api.example.org/shared-state/q1w2e3r4t5"},
    {"rel": "create, sharedState",
      "href": "http://api.example.org/shared-state/q1w2e3r4t5/form"}
    {"rel": "filter, sharedState",
      "href": "http://api.example.org/shared-state/q1w2e3r4t5/search"}
  ],
  "items" : [
    { "id": "pid-00122356",
      "href": "http://api.example.org/shared-state/q1w2e3r4t5#pid-00122356",
      "data": [
        {"name": "currentPrice", "value": "10"},
```

```
          {"name": "updatedPrice", "value": "11"},
          {"name": "state", "value": "pending"},
          {"name": "text", "value": ""}
        ]
      },
      {...}
    ]
  }}
```

Once the document is loaded, the job can be started. Here is the pseudocode:

```
FOR-EACH item in http://api.example.org/shared-state/q1w2e3r4t5
  updatePrice(item)
END-EACH
```

In this case, the `item` is read, the function (`updatePrice`) uses the item data to look up the product in the catalog, and (assuming the `currentPrice` agrees with the one in the shared state document) the `updatedPrice` is applied. The shared state document is then modified to indicate the status of the work (e.g., from `pending` to `completed`), and any notes are supplied in `text` (e.g., "applied successfully" or "currentPrice does not agree," etc.).

Each time the job completes a loop and modifies the shared state document, it should write the updated document to the same URL (using an idempotent HTTP `PUT`). This will make sure to most accurately record the state of the work in progress. This makes it easy to support the progress resource (see Recipe 7.7) and helps in diagnosing any problems encountered along the way (see Recipe 7.18).

Once the work is completed, the resulting shared state resource can be archived for later reference.

Discussion

In the preceding example, the data properties (`currentPrice`, `updatedPrice`, `status`, and `text`) must be semantic identifiers that are agreed upon ahead of time. This is typically handled through a shared vocabulary (see Recipe 3.3).

Note that in the preceding example, the task was described using idempotent language. For example, the target property of the change is expressed as `updatedPrice` and not `percentIncrease`. This makes supporting repeating and reverting changes much easier.

You might think a simple name-value pair document is all that is needed for your shared state resource. This may work, but it is much better to include lots of metadata to help services keep track of what work needs to be done and how much has been completed. Whenever possible, make your shared state resource a full-fledged HTTP resource, not just a serialized data file.

Often, you will need to use the shared state document as a source for filling in a data input form for another service. The easiest way to do this is to write a *mapping* function that takes the form and the shared state data as inputs, and applies the shared state properties to the form automatically:

```
var sharedState = httpRead("http://api.example.org/state/q1w2e3r4t5");
var inputForm = httpRead("http://api.example.org/users/onboardingform");
var completedForm = propertyMap(sharedState, inputForm);
var results = httpExecute(completedForm);
```

One of the advantages of the shared state recipe is that, over time, you can add new state data to the document without needing to update any existing services that rely on the resource. Each service can use the contents of the state resource as it sees fit. This loosens the coupling between the services and the state, and leads to a more configuration-based kind of implementation.

See Also

- Recipe 3.7, "Enabling Interoperability with Inter-Service State Transfers"
- Recipe 4.15, "Maintaining Your Own State"
- Recipe 5.14, "Increasing Throughput with Client-Supplied Identifiers"
- Recipe 6.11, "Extending Remote Data Stores"
- Recipe 7.1, "Designing Workflow-Compliant Services"
- Recipe 7.13, "Using State-Watch to Enable Client-Driven Workflow"

7.3 Describing Workflow as Code

When you want to enlist multiple service interfaces to compose a single solution, you need some way to describe the list of services and the ways they can interact. There are multiple approaches. This recipe shows the simplest method: source code behind a solution interface.

Problem

What is the least complicated way to compose a solution with multiple existing independent service interfaces using service component source code? When does it make sense to use source code instead of some other, more abstract and flexible approach?

Solution

When you have a rather simple workflow solution and/or one that will not change over time and does not support customization, it usually makes sense to describe that workflow solution using source code within a single service interface. That means

creating an interface that has all the features of your desired solution and then, behind the interface, using source code to enlist and execute other remote services to complete the work.

In this way, you present a stable, straightforward API for the consumer applications and allow yourself maximum design and build flexibility behind that stable interface. Over time, the enlisted services might change their own APIs or you might replace one service (e.g., the `taxComputation` service) with another one (e.g., `newTaxComputation`). As long as you do not change the external interface, you can make substantial changes to the internal implementation without breaking your promise to the solution consumers. See Recipe 5.2 for details.

Example

Consider an organization that needs to support an onboarding workflow for new employees. Let's also assume that, independently, multiple parts of the organization have online services that support their team's responsibilities. That means you have a set of valuable service interfaces you can use to compose a solution in source code.

One source code solution that gathers the needed inputs and then calls a set of external services to handle the work might look like this:

```
(TK check code—ma)
// provide input screens to gather important data
var stateData = collectInputs(personalData, backgroundCheck,
      healthInsurance, salaryDetails);

// execute work in parallel
Promise.all([apiPersonalData(stateData), apiBackgroundCheck(stateData),
    apiHealthInsurance(stateData), apiSalaryDetails(stateData)])
  .then((results) => {
    emitSuccessResponse(results);
  }).catch(error => {
    emitFailedResponse(results);
});
```

In this example, the required inputs are collected and then a set of parallel calls are made to external service interfaces to handle each required task in the workflow. The successful results are then returned to the caller using either the `emitSuccessResponse(results)` method or, if there were any processing or network problems, via the `emitFailedResponse(results)` method.

The internal code can be exposed via HTTP as follows:

```
*** REQUEST ***
GET /onboarding/jobs/
Accept: text/html
Link: <http://webapicookbook.com/profiles/jobs>;rel=profile
...
```

```
*** RESPONSE ***
200 OK
Content-Type: text/html
Link: <http://webapicookbook.com/profiles/jobs>;rel=profile
...

<html>
  ...
  <h1>Onboard New Employee</h1>
  <form name="submit-job" action="/onboarding/q1w2e3r4" method="put">
  ...
  </form>
</html>

*** REQUEST ***
PUT /onboarding/jobs/q1w2e3r4
Content-Type: application/x-www-form-urlencoded
Accept: text/html
Link: <http://webapicookbook.com/profiles/jobs>;rel=profile

jobID=q1w2e3r4&...

*** RESPONSE ***
202 Accepted
Content-Type: text/html
Link: <http://webapicookbook.com/profiles/jobs>;rel=profile

<html>
  ...
   <h1>Job q1w2e3r4</h1>
   <div name="progress">
      <span id="jobStatus">pending</span>
      ...i
      <a href="..." rel="refresh">Check Job Status</a>
   </div>
</html>
```

In this HTTP exchange, the API consumer requests the resource that supplies the FORM inputs for describing an onboarding job. Then, after filling in the FORM, it is submitted to the URL (supplied in the form) that will execute the job. Notice that `202 Accepted` is returned by the workflow engine while the job is in process.

The API consumer can use the `refresh` URL to monitor the status of the job, and eventually, the response will be either a `200 OK` or a 4xx/5xx status code indicating the completion of the job run.

Discussion

Note that this example uses the javascript `promise.all()` construct. That assumes all the services can run in parallel and they don't have a fixed sequence. If you need to implement a workflow that has one or more sequential steps, you should treat each fixed step as *separate* workflow elements (see Recipe 7.1 for details).

The work of collecting inputs can be handled using Recipe 7.10 and/or Recipe 7.12.

Errors can be tricky. In some cases, you may need to support a *rollback* option to undo the work already completed within the workflow. See Recipe 7.17 for a way to support the "undo" option in your workflows.

While using source code is a common way to solve network workflow problems, it has its limits. For example, any changes to the workflow will likely require additional rounds of design, code, test, and deployment. Depending on your resource availability (e.g., money and time), changes can become costly bottlenecks in your workflow.

Even when you have sufficient resources, using source code as your workflow language can result in many one-off solutions that are not easily reusable (e.g., `onboardEmployee`, `onboardPartTimeEmployee`, `onboardTemporaryWorker`, etc.). If you need to add customizations to existing solutions and/or want to create more robust workflows, check out Recipes 7.4 and 7.5. For a more complex, but more flexible solution, see Recipe 7.6.

See Also

- Recipe 3.4, "Describing Problem Spaces with Semantic Profiles"
- Recipe 4.11, "Validating Data Properties at Runtime"
- Recipe 4.15, "Maintaining Your Own State"
- Recipe 5.2, "Preventing Internal Model Leaks"
- Recipe 5.16, "Providing Runtime Fallbacks for Dependent Services"
- Recipe 6.7, "Using Media Types for Data Queries"
- Recipe 7.1, "Designing Workflow-Compliant Services"
- Recipe 7.4, "Describing Workflow as DSL"
- Recipe 7.5, "Describing Workflow as Documents"

7.4 Describing Workflow as DSL

When you are writing a lot of workflow solutions, relying on source code may not scale well. In these cases, it can be better to use a workflow-aware domain-specific language (DSL) to design, code, test, and deploy your solutions.

Problem

What does a workflow-aware DSL look like, and when does it make sense to use it instead of native code implementations? What are the advantages and challenges of supporting a workflow DSL for your organization?

Solution

As mentioned in Recipe 7.3, using native source code as your workflow language works well when you have rather straightforward solutions that don't require much customization and/or rarely change over time. However, in cases where your workflows need to be more flexible or when you need to support lots of workflow solutions within an organization, it is usually better to adopt an approach that assumes a workflow-aware environment that supports general workflow processes and lowers the barrier for creating, testing, and deploying them into production.

Example

Here's an example of handling the ordering and checkout of groceries using Hyper-CLI, a hypermedia DSL that I created for this book:

```
# complete grocery checkout

# load job config
CONFIG LOAD grocery.config
grocery.exitOnError = true

# load shared state
REQUEST WITH-URL $$sharedstateURL$$
STACK PUSH WITH-RESPONSE ##sharedState#

# get update cart
REQUEST WITH-URL $$scartURL$$
REQUEST WITH-FORM cartCheckout WITH-STACK
STACK PUSH WITH-RESPONSE ##sharedState##

# get shipping details
REQUEST WITH-URL $$shippingURL$$
REQUEST WITH-FORM bookShipping WITH-STACK
STACK PUSH WITH-RESPONSE ##sharedState##

# settle payment
```

```
REQUEST WITH-URL $$paymentURL$$
REQUEST WITH-FORM settlePayment WITH-STACK
STACK PUSH WITH-RESPONSE ##sharedState##

# update shared state
REQUEST WITH-URL $$sharedStateURL$$
REQUEST WITH-FORM updateState WITH-STACK

# end job
```

In this example, three tasks are completed (`cartCheckout`, `bookShipping`, and `settlePayment`). At the start and end of the script, the shared state is first loaded then, after it is updated by the three tasks, saved again. Note also the `grocery.config` file loaded at the start. This holds the various URLs needed to complete the job.

A hypermedia DSL like the one shown in this example can streamline workflow authoring since many of the details (e.g., handling HTTP requests, locating and filling in forms in a response, etc.) are handled by the DSL directly. If you were using code (see Recipe 7.3), you'd need to deal with this yourself (possibly through a shared code library).

Alternatively, you can use shell scripting (MS-DOS, Linux bash, etc.) to do the same work. It takes a bit more effort but can be quite effective. You need to lean on a handful of utilities, like `cURL` for handling HTTP, `xslt` and/or `jq` for parsing XML and JSON, respectively, and a heavy dose of `grep`, `awk`, and other string manipulation tools.

Discussion

DSLs can make it safer and easier to create workflow scripts since they constrain the possible actions due to the limited power of the language itself. This can also be a bit of a frustration for programmers who are used to a fully operational programming language like Java, NodeJS, etc.

Using shell scripting and utilities offers more power and flexibility than a dedicated DSL like the one shown in this example, but with great power comes great responsibility. It may be harder to craft a safe, effective shell script and you are more likely to introduce bugs along the way.

Whether you use a dedicated DSL or pull together your own using command-line shell scripting, expressing workflow as script, can reduce the turnaround time it takes to code, test, and deploy your workflow solutions.

See Also

- Recipe 3.4, "Describing Problem Spaces with Semantic Profiles"
- Recipe 4.11, "Validating Data Properties at Runtime"

- Recipe 4.15, “Maintaining Your Own State”
- Recipe 5.14, “Increasing Throughput with Client-Supplied Identifiers”
- Recipe 6.7, “Using Media Types for Data Queries”
- Recipe 7.1, “Designing Workflow-Compliant Services”
- Recipe 7.3, “Describing Workflow as Code”
- Recipe 7.5, “Describing Workflow as Documents”

7.5 Describing Workflow as Documents

The most robust and resilient way to describe service workflow is using declarative documents. This recipe provides an example of what a declarative, document-based workflow language looks like.

Problem

You want to describe workflow in a declarative way—one that does not involve a step-by-step execution plan, but instead, describes all the work to be done and allows document consumers to decide how to accomplish the work. What does that document look like? What are the required elements of such a document, and what are the limits to using this approach to supporting workflow in a hypermedia environment?

Solution

The best way to describe workflow in a declarative way is to create a workflow vocabulary (see Recipe 3.3) that outlines the properties and actions needed to fulfill all the requirements of a workflow-compliant service, as defined in Recipe 7.1. That means supporting both tasks and jobs as well as supporting shared state. The documents also need to make it easy for consumers and producers to use `correlation-id` and `request-id` headers to track and manage the work being done.

For more on how to describe and enable declarative workflow, see Recipe 7.6.

Declarative documents for workflow include all the necessary properties and actions (see the example) needed to carry out the work. These documents are typically expressed as HTTP resources (e.g., *http://api.example.org/jobs/q1w2e3r4*). They may also be expressed as messages in a queue (see Recipe 7.19). This is especially useful if the number and/or rate of jobs to process is large, or the traffic for job processing is sporadic, tends to cause bottlenecks, or experience wait times.

Example

The following is a sample document that describes a workflow job following the recommendations of Recipe 7.1 (in an HTML representation):

```
<html>
  <head>
    <title>Shopping Checkout Workflow</title>
  </head>
  <body>
    <h1>Shopping Checkout Workflow</h1>
    <div class="job">
      <span class="jobID">q1w2e3r4t5</span>
      <span class="jobStatus">working</span>
      <span class="jobDateCreated">2023-02-23:14:00:00</span>
      <span class="jobDateUpdated">2023-02-23:14:00:30</span>
      <span class="jobMaxTTL">300</span>
      <span class="jobDescription">...</span>
      <a href="..." rel="jobURL">Job</a>
      <a href="..." rel="jobSharedStateURL">Shared State</a>
      <a href="..." rel="jobProgressURL">Shared State</a>
      <a href="..." rel="jobSuccessURL">Success</a>
      <a href="..." rel="jobFailedURL">Failed</a>
      <a href="..." rel="jobContinueURL">Continue</a>
      <a href="..." rel="jobRestartURL">Restart"</a>
      <a href="..." rel="jobCancelURL">Cancel"</a>
      <div class="tasks">
        <div class="task">
          <span class="taskID">p0o9i8u7y6</span>
          <span class="taskStatus">completed</span>
          <span class="taskDateCreated">2023-02-23:14:00:00</span>
          <span class="taskDateUpdated">2023-02-23:14:00:30</span>
          <span class="taskMaxTTL">60</span>
          <span class="taskDescription">ComputeTaxes</span>
          <span class="taskMessage">Success (200)</span>
          <a href="..." rel="taskURL">Task</a>
          <a href="..." rel="taskStartURL">Start</a>
          <a href="..." rel="taskRollbackURL">Rollback</a>
          <a href="..." rel="taskRerunURL">Rerun</a>
          <a href="..." rel="taskCancelURL">Cancel</a>
        </div>
        ...
        <div class="task">
          <span class="taskID">u8y7t6r5e4</span>
          <span class="taskStatus">pending</span>
          <span class="taskDateCreated">2023-02-23:14:00:00</span>
          <span class="taskDateUpdated">2023-02-23:14:00:30</span>
          <span class="taskMaxTTL">60</span>
          <span class="taskDescription">Schedule Shipping</span>
          <a href="..." rel="taskURL">Task</a>
          <a href="..." rel="taskStartURL">Start</a>
          <a href="..." rel="taskRollbackURL">Rollback</a>
```

```
            <a href="..." rel="taskRerunURL">Rerun</a>
            <a href="..." rel="taskCancelURL">Cancel</a>
          </div>
        </div>
      </div>
    </body>
  </html>
```

This document/vocabulary can be used by a *workflow-engine* application to handle all the details of running and monitoring the job. See Recipe 7.6 for more on how to handle each action element.

Discussion

The advantage of using declarative documents for workflows is that you are separating the description details from the processing code. This makes it possible to change the code frequently and improve it over time without needing to change the document format or contents.

Document-driven workflows are effective when you are dealing with workflow-compliant service interfaces, as described in this book. If you are trying to enlist non-compliant services, you probably need to use a DSL (Recipe 7.4) or code (Recipe 7.3) approach instead.

Be careful to *not* introduce imperative-style step-by-step processing into your workflow documents. Elements like branching (`if..then..else`), variable-checking (`when $state.value==="13"`), and other similar constructs turn your declaration document into an imperative programming language. These decision elements should be kept within the *tasks* (the service being used) and not within the *job* document. If you think you need this level of programming, you probably need to use a DSL (Recipe 7.4) or maybe a code-based (Recipe 7.3) approach instead.

See Also

- Recipe 3.4, "Describing Problem Spaces with Semantic Profiles"
- Recipe 3.5, "Expressing Actions at Runtime with Embedded Hypermedia"
- Recipe 3.9, "Designing for Reversible Actions"
- Recipe 4.3, "Coding Resilient Clients with Message-Centric Implementations"
- Recipe 4.4, "Coding Effective Clients to Understand Vocabulary Profiles"
- Recipe 4.15, "Maintaining Your Own State"
- Recipe 5.4, "Expressing Internal Functions as External Actions"
- Recipe 5.16, "Providing Runtime Fallbacks for Dependent Services"
- Recipe 7.1, "Designing Workflow-Compliant Services"

- Recipe 7.3, "Describing Workflow as Code"
- Recipe 7.5, "Describing Workflow as Documents"

7.6 Supporting RESTful Job Control Language

One of the persistent challenges for creating multiservice workflows on the web is a consistent and robust language for describing and implementing workflow. This recipe provides a basic set of actions and metadata for orchestrating independent workflows on the open web.

Problem

When you have a lot of workflow processing to deal with, it can be helpful to implement a generic workflow engine that supports managing multiple jobs and tasks. What does it take to support a robust job control language (JCL), and how can you confidently mix and control independent service interfaces that might be running in various places across the web?

Solution

As your use of workflow grows, you'll need a consistent way to describe, manage, and execute those workflow jobs. I've already covered the basics of workflow-compliant, composable services (see Recipe 7.1). Now, here's a recipe that defines a robust and reliable JCL for the web.

I've used various versions of this RESTful JCL recipe a number of times over the years, each tuned for local needs. What I'm including here is a generic recipe that takes into account lots of features used in the past, bringing them together in a single place. Feel free to tweak and adjust this recipe to fit your specific needs.

Wait, What About iPAAS?

There are definitely other workflow platforms you can use to meet your interoperability needs. In the limited space we have here, I've chosen to focus on a hypermedia-driven solution since that is one that I rarely see in use, even though I've had great success with it in the past. Most other solutions result in hardcoding connections and parameter sharing while a REST-based approach achieves the same ends without the tightly coupled implementation.

This recipe provides a basic set of actions and metadata for coordinating independent services on the open web. It also includes actions for managing (list, filter, add, update, and remove) job records. Figure 7-2 shows the RESTful job control workflow.

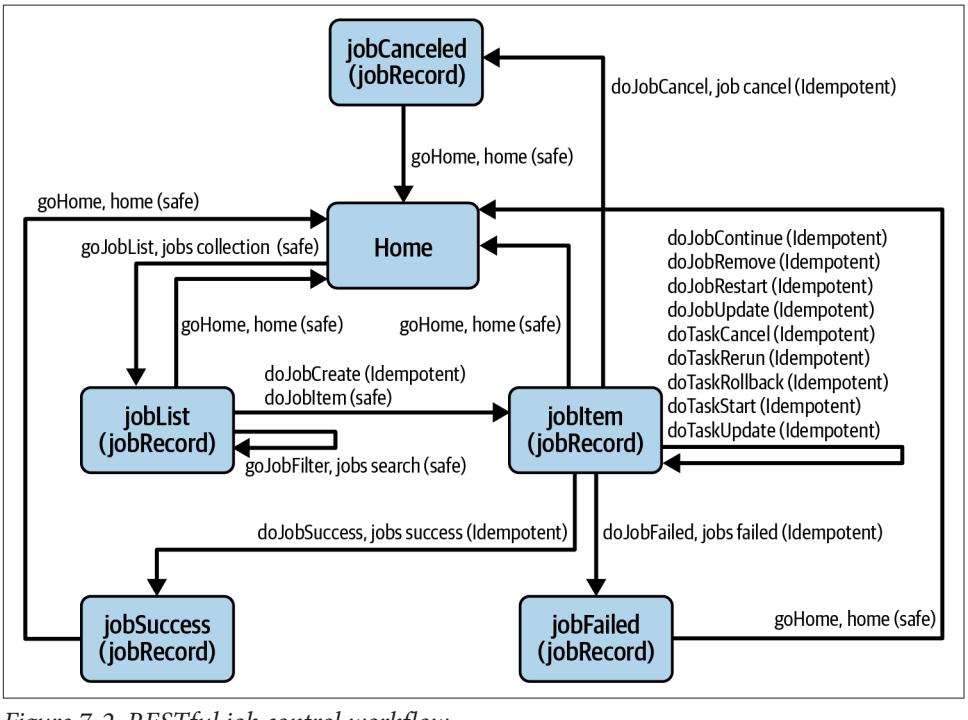

Figure 7-2. RESTful job control workflow

Executing workflow with RESTful JCL

There are two key elements to defining and executing workflow records:

Tasks
: A task is a single action in a workflow (`computeTaxes`, `DoCreditCheck`, `UpdateCustomerAccounts`, etc.) A task supports execute (`taskStartURL`), repeat (`taskRerunURL`), revert (`taskRollbackURL`), and cancel (`taskCancelURL`).

Jobs
: Each job contains one or more tasks. The set of tasks in each job *must* be able to run in parallel. A job supports restart (`jobRestartURL`), continue (`jobContinueURL`), and cancel (`jobCancelURL`). It also has a URL to call when the job completes successfully (`jobSuccessURL`) and when the job fails (`jobFailedURL`).

To run a job, the job document should be submitted to a workflow engine. Upon receiving the document, the workflow engine will return a `202 Accepted` (see Recipe 7.15) while the tasks are processed and will eventually return a `200 OK` or a 4xx/5xx

HTTP status with the completed response. Upon completion of a job, that job document *may* be archived for future reference (or re-execution), and the appropriate link (`jobSuccessURL` or `jobFailedURL`) *should* be activated.

Compliant workflow engines *must* be prepared to handle `jobCancelURL`, `jobContinueURL`, and `jobRestartURL`. They should also support `taskStartURL`, `taskRollbackURL`, `taskRerunURL`, and `taskCancelURL` properties, as well as monitor or update progress via `taskStatus` and `jobStatus`. The engines *should* also monitor the `jobMaxTTL` and `taskMaxTTL` values, and cancel jobs whenever the maximum runtime has been exceeded.

Tasks that need to be run in a fixed sequence *must* be separated into individual job documents, and those job documents can be run in fixed order as each job successfully completes.

Managing RESTful JCL records

Along with the work for *running* workflows, you also need to support authoring, filtering, and removing workflow documents. A handful of actions are included in this recipe to handle workflow document management:

`jobList`
: Return a list of job records

`jobFilter`
: Search for job records

`jobRead`
: Return a single job record

`jobCreate`
: Add a new job record to the collection

`jobUpdate`
: Modify an existing job record

`jobRemove`
: Delete an existing job record

You'll notice that this list does not include actions for adding, editing, and removing task records. In this case, the `jobItem` is the smallest manageable resource in the system. If you wish, you can implement another set of management actions (essentially applying Recipe 7.11) at the task level. In my experience, this is rarely needed.

Shared state, correlation IDs, and progress

An element not shown here is support for passing state between tasks. See Recipe 7.2 for details on how to implement that.

Another important element is sharing `jobID` and `taskID` via headers at runtime. As covered in Recipe 7.1, each workflow-compliant service *must* share a `jobID` (as `Correlation-ID` header) and `taskID` (as `Request-ID` header) for every action within the job.

Workflow-compliant services used for this recipe should support progress reporting (see Recipe 7.7) and possibly the "call for help" pattern (Recipe 7.18).

Discussion

RESTful JCL only works if all the services you enlist as tasks follow the rules laid out in Recipe 7.1. The good news is you can implement workflow-compliant services without having to also support a full local job control language. RESTful JCL comes *after* authoring workflow-compliant services. If you're just starting out on the road to workflow systems, consider focusing on compliant service interfaces first, then moving on to exploring JCL options.

Are You Sure You Want to Do This?

Implementing a general job control system like the one described here is not a simple task. Existing alternatives to this recipe are available as off-the-shelf software and as online SAAS (software as a service) implementations. Choosing to "roll your own" JCL is a decision not to be made lightly. If you need to support multiple hosting platforms (including your own local servers), this recipe can be helpful. If, however, you are only hosting services that run on a single platform (e.g., AWS, Azure, Google, etc.), using their workflow tooling may be a better choice.

Workflow authoring services *may* be separate from workflow execution services. The editor and the engine don't need to be hosted within the same service interface. In fact, it can be advantageous to keep the editing work and the execution work separate since they can move forward at their own individual paces.

The premise of this recipe is that you want to create workflows that take advantage of service interfaces you have not created and do not control (see Appendix A). Even if you are working with services built by other teams inside your own organization, RESTful JCL can improve your success in reusing existing services from other parts of the company.

This recipe works well when you commit to describing your workflow as documents (Recipe 7.5). It is much harder to pull off if you are relying on code (Recipe 7.3) or DSL (Recipe 7.4) to express workflow.

See Also

- Recipe 3.8, "Designing for Repeatable Actions"
- Recipe 3.9, "Designing for Reversible Actions"
- Recipe 4.4, "Coding Effective Clients to Understand Vocabulary Profiles"
- Recipe 4.9, "Relying on Hypermedia Controls in the Response"
- Recipe 4.15, "Maintaining Your Own State"
- Recipe 5.7, "Publishing Complete Vocabularies for Machine Clients"
- Recipe 5.16, "Providing Runtime Fallbacks for Dependent Services"
- Recipe 7.1, "Designing Workflow-Compliant Services"
- Recipe 7.16, "Short-Term Fixes with Automatic Retries"
- Recipe 7.17, "Supporting Local Undo or Rollback"
- Recipe 7.18, "Calling for Help"

7.7 Exposing a Progress Resource for Your Workflows

When executing and monitoring workflow jobs, it is a good idea to offer up a progress report on the job so that API consumers and producers can see how things are going and what, if any, problems were encountered. This recipe provides an example progress resource you can use for your workflow implementations.

Problem

How can I keep an eye on long-running workflow jobs? Where can I go to monitor transaction details or check for errors? What does it take to create a general workflow progress resource that works whether you are describing your workflows using code (Recipe 7.3), a DSL (Recipe 7.4), or documents (Recipe 7.5)?

Solution

Like so many other solutions offered in this book, you can often solve your problem with an HTTP metadata resource (see Recipes 5.5, 5.11, and 5.12). In this case, we'll create a workflow-progress resource that holds any and all execution data related to a workflow job and its tasks (see Recipes 7.1 and 7.6 for more on jobs and tasks for workflow).

Every workflow job should have an associated progress resource. This resource can be tied to the job using the `jobID`, which is often the same as the `correlation-id` value stored in an HTTP header. That resource can be a collecting point for any progress reporting or other tracking for the job. You should track both the job metadata and the metadata for each task in the job as they are executed. Figure 7-3 illustrates the workflow progress.

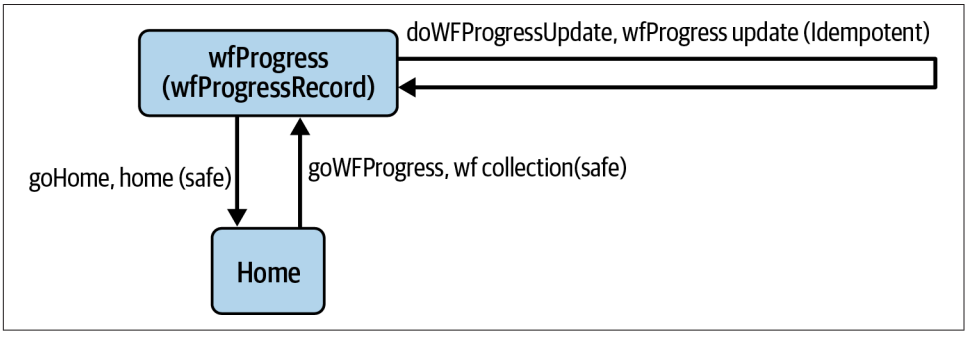

Figure 7-3. Workflow progress

Important job metadata is:

`jobID`
: Unique identifier for the job resource (often the `correlation-id`)

`jobURL`
: URL of the job resource document

`jobDescription`
: Human-readable text that describes the job

`jobStatus`
: Indicates job status (pending, working, completed, or failed)

`jobDateCreated`
: Indicates the date/time (UTC) the job resource was first created

`jobDateUpdated`
: Indicates the last date/time (UTC) the job resource was modified

`jobMaxTTL`
: Indicates the maximum amount of time (in seconds) this resource is considered valid

Important task metadata is:

`taskID`
: Unique identifier for this task (often the `request-id`)

`taskURL`
: URL of the task resource

`taskDescription`
: Human-readable text that describes the task

`taskStatus`
: Indicates task status (pending, working, completed, or failed)

`taskStartDateTime`
: Indicates the date/time (UTC) the task started

`taskStopDateTime`
: Indicates the date/time (UTC) the task stopped

`taskMaxTTL`
: Indicates the maximum amount of time (in seconds) this task is considered valid

`taskMessage`
: Human-readable text that contains any messages related to the task

The last item for each task (`taskMessage`) can contain multiple lines of progress reporting, such as each attempt to complete the task. Writing text lines into the `taskMessage` field is often sufficient for most monitoring needs. Another useful way to record that is to include the HTTP request details (URL, headers, and body) as well as the HTTP response details (HTTP status, headers, and body).

If you include the HTTP details as part of your progress resource, you may be exposing a security leak. Make sure only duly authorized users can read this part of the progress resource.

You can format the progress resource for display in any of the common media types (HTML, Collection+JSON, HAL, SIREN, UBER, etc.). When displaying the progress data, you should also include a refresh link and caching data to give the API consumer hints on the freshness of the resource.

Example

Here is a sample progress resource formatted in Collection+JSON:

```
**** REQUEST ****
GET /jobs/q1w2e3r4t5;progress
Accept: application/vnd.collection+json
...

**** RESPONSE ****
```

```
200 OK
Content-type: application/vnd.collection+json
Link: <<http://api.example.org/jobs/q1w2e3r4t5;progress>>; rel="refresh"
Cache-control: max-age 10000, public

{"collection" : {
  "title": "Job q1w2e3r4t5 Progress",
  "links": [
    {"rel":"refresh","href":"http://api.example.org/jobs/q1w2e3r4t5;progress"}
  ],
  "items": [
    {"id": "q1w2e3r4t5",
     "type": "job",
     "data": [
       {"name":"jobID", "value":"q1w2e3r4t5"},
       {"name":"jobURL", "value":"http://api.example.org/jobs/q1w2e3r4t5"},
       {"name":"jobDescription", "value":"Employee Onboarding Job"},
       {"name":"jobStatus", "value":"working"},
       {"name":"jobDateCreated", "value":"2024-05-01:13:12:11"},
       {"name":"jobDateUpdated", "value":"2024-05-01:13:15:14"},
       {"name":"jobMaxTTL", "value":"30000"}
     ]
    },
    {"id": "u8y7t6r5",
     "type": "task",
     "data": [...]
    },
    {"id": "5r4e3w2q",
     "type": "task",
     "data": [...]
    },
    {"id": "p0o9i8u7",
     "type": "task",
     "data": [
      {"name":"taskID", "value":"p0o9i8u7"},
      {"name":"taskURL,
        "value":"http://api.example.org/jobs/q1w2e3r4t5#p0o9i8u7"},
      {"name":"taskDescription, "value":"Credit Check"},
      {"name":"taskStatus, "value":"completed"},
      {"name":"taskStartDateTime, "value":"2024-05-01:13:13:11"},,
      {"name":"taskStopDateTime, "value":"2024-05-01:13:14:11"},,
      {"name":"taskMaxTTL, "value":"5000"},
      {"name":"taskMessage, "value":"completed successfully."},
     ]
    }
  ]
}}
```

Note that in this example, only basic progress information is shared for each task (e.g., `"completed successfully"`). This reduces the chances of leaking personal data or other intellectual property that might be a breach of security.

Once a job is completed, you should archive these progress records for later review, as needed.

Discussion

The simple goal of workflow progress resources is to offer additional feedback to API callers as the jobs and tasks execute. For that reason, it is a good idea to keep workflow progress resources very basic. Don't try to include too much data, and be sure to not include any internal or private programming information as this can constitute a security leak.

Workflow progress resources *should not* be used as trace logs for your services, and they *should not* include any internal values and/or debugging information. If you need that kind of detail, use something available within your programming/platform internals, and do not expose that information via HTTP.

You might be tempted to include several links in the progress document that allow the caller to restart a task, perform a rollback, or run other workflow management options. This is not a good practice. If you are using code (Recipe 7.3) or DSL (Recipe 7.4) to describe your workflow, these added features can cause conflict and confusion at runtime.

See Also

- Recipe 3.3, "Sharing Domain Specifics via Published Vocabularies"
- Recipe 4.4, "Coding Effective Clients to Understand Vocabulary Profiles"
- Recipe 5.5, "Advertising Support for Client Response Preferences"
- Recipe 5.10, "Publishing API Metadata"
- Recipe 6.5, "Returning Metadata for Query Responses"
- Recipe 7.14, "Optimizing Queries with Stored Replays"

7.8 Returning All Related Actions

There are times when the list of possible actions is quite long. In some cases, there may be more than a dozen possible state transitions (links or forms) available from a single response. Returning all those forms (with default values filled in) for each request can be very costly and affect service response times.

Instead of returning all possible state transitions on every request, you can create a separate resource response that does that independently of the current workflow or state of the application. This reduces response payload size, increases response times, and standardizes a repeatable way to get an up-to-date list of available transitions.

Problem

For some applications, the possible number of state transitions (or links) at any single moment can be quite long. Common workflow interactions—like paging through a list (see Recipe 7.11), for example—can have lots of valid possible actions at a single moment, such as `top`, `next`, `previous`, `last`, `item`, `create`, `refresh`, and `self`. That does not include any transitions to other collections or other related activities (`list-users`, `list-companies`, `list-reports`, etc.).

Listing all these options with every response can be tedious and take up quite a bit of payload space. What is needed is a standardized way to ask for all the possible transitions as a separate step (e.g., a transition!) to reduce the payload size without reducing the available information.

Solution

A simple way to solve the "too many links in the response" challenge is to expose a separate resource that always lists all the possible actions at any single moment in time for the application. You can do this via the `related` IANA link relation value. Now, the service interface can be designed to offer helpful links within the body of the payload and then offer all the other possible links as a separate resource (see the example).

An advantage of this approach is that the service behind the interface can take into account various context-related variables when assembling the contents of the `related` resource. This includes the state of the data behind the service (is data available right now?), the state of application processing (is the service in maintenance mode?), the state of the client application (is the client in the middle of a set of actions?), and even the calling context identity (is the logged-in user an admin?).

Figure 7-4 is a simple diagram of the `related` link that you can add to each response. See the "Example" section for details.

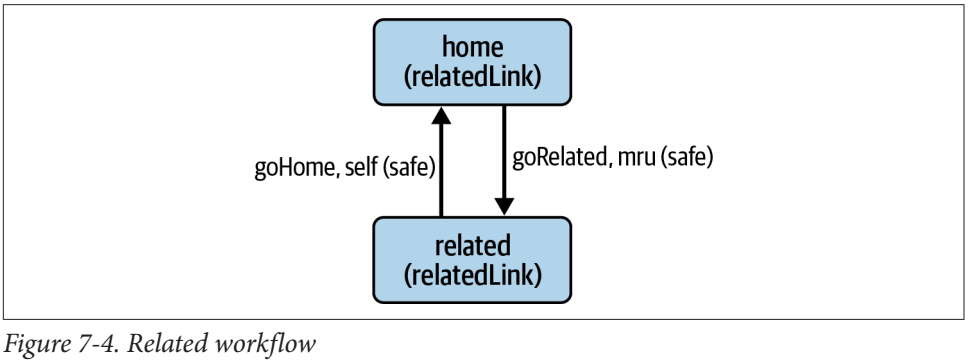

Figure 7-4. Related workflow

Example

By adding a `related` link to your response, you can "bury" the additional transitions that you don't want to expose in the default response. In this example, the `self` transition is always added to responses. Now, there is also a `related` link that clients can use to get a more complete list of all the possible actions:

```
{
  "collection" : {
    "links": [
      {
        "name": "self",
        "rel" : "home self",
        "href": "https://api.example.org/"
      },
      {"$comment" : "link that points to the list of related actions"},
      {
        "name": "related",
        "rel": "collection",
        "href": "https://api.example/org/related"
      },
      {
        "name": "list",
        "rel": "collection",
        "href": "https://api.example.org/list"
      }
    ],
    "items": [ ... ],
    "queries": [ ... ],
    "template": { ... }
  }
}
```

The `text/uri-list` IANA media type is an excellent format for returning lists of related links.

When the client application follows the `related` link, the response returns a complete list of all the possible actions for the current calling context:

```
{
  "collection" : {
    "links": [
      {
        "name": "self",
        "rel" : "related",
        "href": "https://api.example.org/"
      },
      {"$comment" : "list of related available actions"},
```

```
        {
          "name": "list",
          "rel": "collection",
          "href": "https://api.example.org/list"
        },
        {
          "name": "filter",
          "rel": "collection",
          "href": "https://api.example.org/filter"
        },
        {
          "name": "create",
          "rel": "collection",
          "href": "https://api.example.org/create-form"
        },
        {
          "name": "needs-approval",
          "rel": "collection",
          "href": "https://api.example.org/list?status=needs-approval"
        },
        {
          "name": "home",
          "rel": "home",
          "href": "https://api.example.org/"
        }
      ],
      "items": [ ... ],
      "queries": [ ... ],
      "template": { ... }
    }
  }
```

It is worth pointing out that in these two examples, the `list` link relation appears in both lists. There is no rule that says a link can occur in only one location. Also, the contents of the `related` resource can change over time. The list of links can reflect the current state of the service, the clients, and the user context.

When service interfaces support the `related` pattern, client applications can easily be coded to "search" for a link or form that was not initially represented in the service interface response. For example:

```
// get root response and perform checkout
var output = request(apiRootUrl);
var link = output.findLink("checkout");
if (link === null) {
  link = output.findLink("related") {
    if(link !=== null) {
      link output.findLink("checkout");
  }
}
if(link !=== null) {
  runCheckout(output, stateVariables);
```

```
}
else {
  runUnableToCheckout(output, stateVariables);
}
```

If you wish, you can also write API consumer code that "hides" the use of `related` behind a facade. Then all you need to write in the client code is `output.findLink("related");`, and the code can look in the local response first and, if needed, attempt to activate the `related` link to see if it can be found there.

Discussion

This recipe is similar to Recipe 7.9 in that it helps to optimize link finding for client applications. Humans are really good at finding links and/or crafting their own from scratch, but machines are not. This recipe is a way to compensate for machine shortcomings without having to rely on some form of artificial intelligence.

Using the `related` recipe makes sense when there are a lot of possible transitions at any given moment. Interface designers can focus on creating a usable set of responses without having to include all possible transitions for all possible responses. This reduces clutter and disagreements about just what links or forms should be included in responses.

See Also

- Recipe 3.5, "Expressing Actions at Runtime with Embedded Hypermedia"
- Recipe 4.9, "Relying on Hypermedia Controls in the Response"
- Recipe 5.4, "Expressing Internal Functions as External Actions"
- Recipe 6.7, "Using Media Types for Data Queries"
- Recipe 7.9, "Returning Most Recently Used Resources"
- Recipe 7.14, "Optimizing Queries with Stored Replays"

7.9 Returning Most Recently Used Resources

You can make it easier on API consumers when you include a list of the most recently used links/forms in your responses. This can help programmers better understand common uses of your service interface and make it easier for developers to access the common links they need to solve their problems.

Problem

It can be a challenge for developers and designers to agree on what links and forms should be returned in hypermedia responses. Designers typically try to create an

interface that is both usable and useful. This usually means leaving some details out. Developers and architects often aim for "completeness" and want to include all possible options in each response (see Recipe 7.8). What's a good way to balance these two options and still include helpful links in responses? What happens if the kinds of links commonly needed by users change over time?

Solution

This recipe is closely connected to Recipe 7.8. That recipe offers a way to `stash` lots of related links in a separate document to keep the responses reasonably small, but still offer a way to access all the possible actions. However, if taken too far, most hypermedia responses will contain a single link (`related`), and now all clients need to do additional HTTP requests to get things done. Not a good idea. Instead, it can be helpful to keep track of the most recently used (MRU) resource links and include those in responses.

A Window into MRUs

The idea of identifying and exposing the most recently used application options became common with the graphical user interfaces of the '80s and '90s. As application menu options got longer, there was a move to improve the interface by reducing options presented to the user. Introducing the MRU pattern allowed applications to offer up the most recently edited documents, used commands, or even requested applications. This MRU concept is also used for caching services to keep track of most recently requested resources.

This recipe can be used in conjunction with Recipe 7.8 to include additional links in commonly used responses. In this approach, the service interface can keep track of commonly requested resources and include a link to them in responses. For example, if client applications typically follow up a request for the `list` resource with a second request for the `add` resource, service interfaces can track that pattern and automatically include the `add` resource link as part of the `list` resource response.

This tracking of commonly used resources can be done at the application level or user level. This makes it possible to customize the MRU list for the calling context—including the access rights of the logged-in user.

Keeping track of MRUs for a context (service wide or consumer specific) takes a bit of work but is not very complicated. A history list of the recently requested resources is all that is needed. As each resource is requested, that link is placed in a list (URL and count). When it is time to represent a resource response, the service interface can consult the MRU listing and (after filtering for access context, if needed), inject the last most recent links into the response.

Returning MRUs can reduce the reliance on Recipe 7.8 and help new developers learn the most common actions for the service. The list of MRUs is usually presented as normal links in the response (Figure 7-5). But you can also create an `mru` resource (similar to the `related` resource from Recipe 7.8).

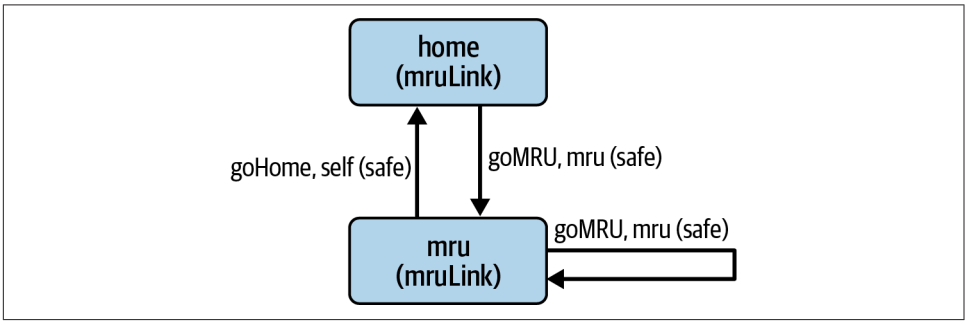

Figure 7-5. MRU workflow

Example

Here's an example response (in Collection+JSON) that shows both the MRU link (`"name":"mru"`) and a few of the most recently used links (the ones with `rel` set to `mru`):

```
{
  "collection" : {
    "links": [
      {
        "name": "self",
        "rel" : "home self",
        "href": "https://api.example.org/"
      },
      {"$comment" : "link that points to the MRU list"},
      {
        "name": "mru",
        "rel": "collection",
        "href": "https://api.example/org/mru"
      },
      {"$comment" : "some MRU links in the response"},
      {
        "name": "list",
        "rel": "mru collection",
        "href": "https://api.example.org/list"
      },
      {
        "name": "pending",
        "rel": "mru collection",
        "href": "https://api.example.org/list?status=pending"
      },
      {
```

```
        "name": "home",
        "rel": "mru home",
        "href": "https://api.example.org/"
      }
    ],
    "items": [ ... ],
    "queries": [ ... ],
    "template": { ... }
  }
}
```

Services that support the MRU recipe should return at least the MRU link in every response. This reminds clients where they can go for additional details. It is also valid to return the first few MRUs (see the example) in responses. These included MRUs should reflect the current user context (e.g., the most recently used links by the currently logged-in user) as well as the current service context (e.g., the most recently used links that are valid at this time). Not all of the most recently used links might be valid at a given moment. For example, it might not be possible to query the list of customers while database maintenance is underway on the service.

Discussion

The work of tracking the most recently used URLs can be handled by creating a simple first in, first out (FIFO) stack of a fixed length (I use a depth of three to five for most cases). After completing the request, you can add the URL value to the stack and, if needed, drop URLs off the "bottom" of the list as it grows too long. Then you can just return this list of URLs in your response.

Don't try to "remember" too much about the previous requests. For example, it is not a good idea to include header information or attempt to derive the "intent" of the request with other metadata. Just keep the links and return those.

The MRU pattern is a great way to help API consumers know which are the more frequent calls for your service. You can think of MRUs as a kind of "help" feature for novice users. You can implement them that way by offering a "toggle" option where consumers can turn MRUs on or off in responses. See Recipe 5.5 for a recipe that supports this kind of toggle option.

You can also use Recipe 5.5 to hold a variable on how long the list of MRUs should be in each response. Typically returning three to five MRUs is more than enough and makes a good default value.

It is a good idea to share the MRU values as simple links—not forms. This keeps down the amount of work the server-side code needs to do to include them and reduces the amount of content in the responses.

When rendering the MRUs in your response, you can make things easy for API consumers by identifying the list using common link relation values (`"mru"`), a block

within the response (`<div id="mru">…</div>`), or some other method common to the response format.

It is not a good idea to return MRU values in HTTP headers (using a `link` element). The MRU list might get long and is best rendered in the body of the message. Alternatively, you can include a single `link` element that points to a standalone MRU list:

```
**** REQUEST ****
GET /users/list HTTP/1.1
Host: api.company.org
Accept: application/vnd.siren
Authentication: BASIC o9eir8tu5ry6362yeir...

**** RESPONSE ****
200 OK HTTP/1.1
Content-Type: application/vnd.siren
Link: <https://company.org/users/mru>;rel="mru"

...
```

If your service supports user identities, you can build in support for tracking MRUs by identity, too. This adds another level of customization to the response list.

Even if you don't want to keep track of the actual commonly used URLs, you can use the MRU recipe to return URLs that are likely to be handy, like the home URL or the collection list URL, etc.

See Also

- Recipe 3.5, "Expressing Actions at Runtime with Embedded Hypermedia"
- Recipe 4.9, "Relying on Hypermedia Controls in the Response"
- Recipe 5.4, "Expressing Internal Functions as External Actions"
- Recipe 6.7, "Using Media Types for Data Queries"
- Recipe 7.8, "Returning All Related Actions"
- Recipe 7.14, "Optimizing Queries with Stored Replays"

7.10 Supporting Stateful Work in Progress

There are times when you need to gather quite a bit of data from API consumers, and a single FORM resource is not enough. In these cases, a set of related steps for data collection and review is handy. This recipe offers a "work-in-progress" pattern for collecting data and submitting the final results.

Problem

How can you collect a large amount of data in small related chunks and then submit the collected result in a single step? What does the data collection involve with multiple people filling in the same resource? What if the collection process is expected to take a long time due to waiting on data from other parties?

Solution

The work-in-progress (WIP) recipe is a great way to gather lots of data over a stretch of time before reviewing it and submitting the data collection for processing. A WIP approach lets you define a persistent "work document" and then, through a set of one or more interactions, continue to collect inputs until you have amassed a sufficient amount of data to submit to a service for processing.

You can also use the WIP recipe to manage multiple document streams in flight. For example, you might need to start two or three WIP documents to complete a task like approving a customer for a credit purchase.

Here is the list of actions in the WIP recipe:

`listWIP`
: Go to the WIP list resource.

`filterWIP`
: Get a filtered list of WIP records.

`createWIP`
: Create a new WIP resource.

`readWIP`
: Retrieve a single WIP resource.

`updateWIP`
: Update a single existing WIP resource.

`cancelWIP`
: Cancel an existing WIP resource.

`shareWIP`
: Share/save an existing WIP resource.

`submitWIP`
: Submit (for final acceptance) a completed WIP resource.

The first two actions (`listWIP` and `filterWIP`) help in managing the list of WIP documents. The next handle the work of a single WIP document (`createWIP`, `readWIP`, `updateWIP`, and `cancelWIP`). `shareWIP` gives you the option of sending the

document to another user or machine for processing, and `submitWIP` supports sending the completed WIP document for processing.

Usually, this recipe is used in conjunction with Recipe 7.12, which can be used to collect the input data.

Figure 7-6 shows the WIP workflow.

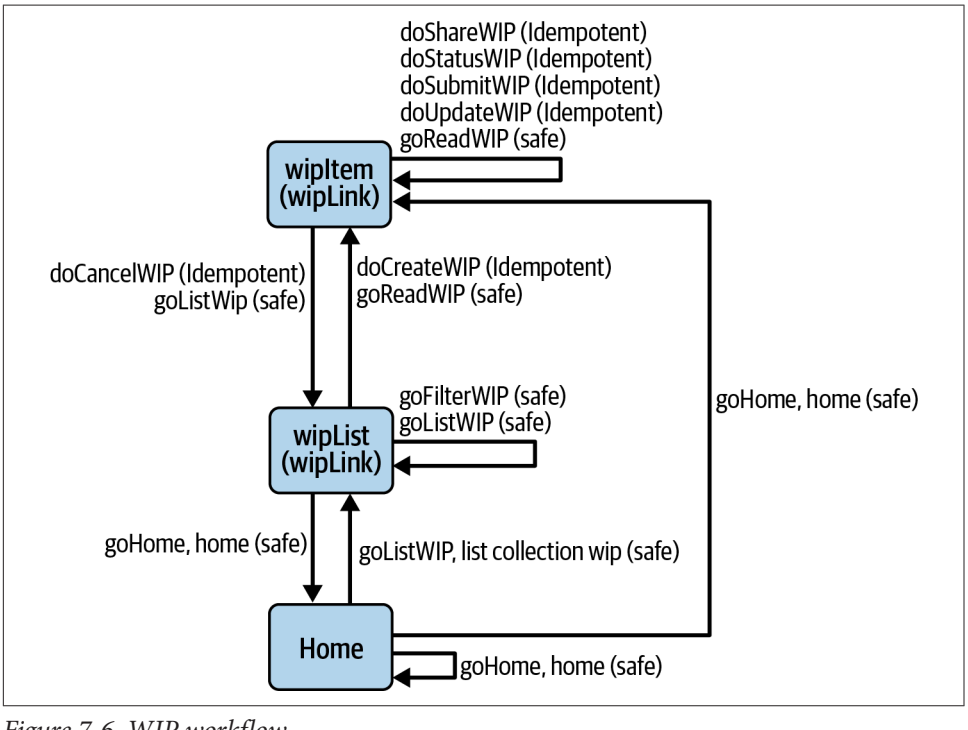

Figure 7-6. WIP workflow

Take the example of employee onboarding. This may require lots of data entry from multiple parties and even the possibility of having to wait on approvals or some other data input that might not arrive for a day or two. You can use a WIP document to handle this process. You may need to collect employee data, inputs from human resource department interviews, the results of a background check from a third party, and details on an employment contract. You can use the WIP recipe to start a work document that will eventually hold all these inputs and share that with various parties within the organization to edit when they have data available. Eventually, someone with the proper authority can submit the completed data for final processing.

Example

Here is an example that works through the employee onboarding scenario:

```
**** REQUEST ****
GET /onboarding/
Accept: application/vnd.siren+json

**** RESPONSE ****
200 OK
Content-Type: application/vnd.siren+json

{
  "title" : "Onboarding",
  "links" : [
    {"rel" : ["listwip"], "href" : "/onboarding/wip/"}
    ...
   ]
}
```

Following the `listwip` link returns a list of any current WIP documents in progress:

```
**** REQUEST ****
GET /onboarding/wip
Accept: application/vnd.siren+json

**** RESPONSE ****
200 OK
Content-Type: application/vnd.siren+json

{
  "title" : "Onboarding WIP List",
  "links" : [
    {"rel" : ["self", "listwip"], "href" : "/onboarding/wip/"},
    {"rel" : ["home"], "href" : "/onboarding/"},
    ...
   ],
   "actions" : [
     {
       "name" : "createwip",
       "method" : "PUT",
       "href" : "/onboarding/wip/p0o9i8",
       "fields" : [
          {"name" : "dateCreated", "value" : "2024-MAY-30"},
          {"name" : "wipowner", "value" : "Mandy Mayberry"}
       ]
     }
     ...
   ]
}
```

The `createwip` form can gather basic data like the date/time it was created and the user associated with creating the document, and then present an interface prompting for data collection. It might look like this:

```
**** REQUEST ****
GET /onboarding/wip/p0o9i8
Accept: application/vnd.siren+json

**** RESPONSE ****
200 OK
Content-Type: application/vnd.siren+json

{
  "title" : "Onboarding Mick Mickelsen",
  "links" : [
    {"rel" : ["self", "listwip"], "href" : "/onboarding/wip/"},
    {"rel" : ["home"], "href" : "/onboarding/"},
    {"rel" : ["employee"], "href" : "/onboarding/wip/p0o9i8/employee"},
    {"rel" : ["interviews"], "href" : "/onboarding/wip/p0o9i8/interviews"},
    {"rel" : ["background"], "href" : "/onboarding/wip/p0o9i8/background"},
    {"rel" : ["contract"], "href" : "/onboarding/wip/p0o9i8/contract"},
    ...
   ],
   "actions" : [
     {
       "name" : "cancelwip",
       "method" : "DELETE",
       "href" : "/onboarding/wip/p0o9i8",
     },
     {
       "name" : "updatewip",
       "method" : "PUT",
       "href" : "/onboarding/wip/p0o9i8",
       "fields" : [
         {"name" : "status", "value" : "working"},
         ...
       ]
     },
     {
       "name" : "submitwip",
       "method" : "PUT",
       "href" : "/onboarding/wip/p0o9i8",
       "fields" : [
         {"name" : "status", "value" : "submitted"}
       ]
     }
     ...
   ]
}
```

Notice that the body of the response contains the native WIP actions (`cancel`, `submit`, and `update`) and that the `link` section of the document includes several substeps for completing the process. Typically each step (`employee`, `interviews`, etc.) contains a form for inputs that supports the `updateWIP` action directly. In this way, each step is expressed as a resource that updates the shared state for that WIP document.

This recipe has lots of variations and implementation options. Be sure to check out the "Discussion" section for hints on how to take advantage of this recipe.

In some cases, the work needs to be handled by some other party and that may mean sharing the WIP document. You can use the `shareWIP` action for that. Depending on how your implementation works, you might just change the "owner" property of this WIP document to allow it to appear in a new user's list of things to process. Or you might generate an email asking someone to follow a link to complete the needed inputs. You might even generate a new resource for another system (e.g., a work ticket, etc.) and use an additional workflow to incorporate the new data into this WIP document.

Eventually, someone needs to either select the `cancelWIP` or `submitWIP` action. Once this is done, the document can be queued for processing (see Recipe 7.15) or be canceled.

Discussion

The WIP recipe has lots of variations, and there aren't many hints in Figure 7-6. My version has just the minimum elements, and you'll need to enhance this recipe with others mentioned earlier.

It is best not to get too intricate with field submissions for a WIP document. Too many dependent fields or fields that are required to be submitted together can turn a WIP workflow into a confusing stateful experience that is hard to understand, predict, or debug. Instead, make sure the service accepts *any* combination of inputs (or just one input at a time) for each `updateWIP` action. You can use the `submitWIP` action as a signal that the API consumer has completed their entry and is looking for input review.

It can be tricky, but you can link WIP documents together to create multistep workflow without much complexity. Using the preceding example, links to each collection step (`employee`, `interviews`, etc.) could point to another WIP document (e.g., `interviewsWIP`). And that one could point to another, and so on. In this way, you can use the WIP recipe to implement a human-centric variable workflow system.

If you want to create more sophisticated machine-driven workflows, see Recipe 7.6 and Recipe 7.15 for approaches that are better suited for "headless" or automated multistep flows.

It's a good idea to archive any completed (or even abandoned) WIP documents for future reference. You may even want to add support for reusing WIP documents in the future. See Recipe 7.14 for hints on how to implement a reuse pattern.

The WIP model is a good way to implement, for example, a "ticketing" system for customer service, support, etc. You can make each set of data another resource for inputs, and display and automate the process of sharing WIPs (via `shareWIP`) based on the state of the document (e.g., fields that are still missing). This usually means establishing some checking logic within the service each time you submit data (via `updateWIP`):

```
var wipDocument = parseUpdateRequestBody();

if(wipDocument.onboardingStatus = "needs background check") {
  wipDocument.assignedTo = "Moira";
  writeStoredItem(wipDocument);
}
...
```

You can use data points within each WIP document to help with filtering (via `filterWIP`) any outstanding documents. For example, you might add some metadata fields like `assignedTo`, `dueDate`, etc., and even automatically filter the complete list based on those fields. In this way you can create customized to-do lists for API consumers to work with.

An easy implementation of this recipe is to just return a single WIP document with all the requested fields, possibly with dozens of inputs. For machine cases, this is not a problem at all, especially if you use Recipe 7.12 to collect the data. You can even use the single-document approach for human UI if you decorate the inputs in a way that results in a kind of "tabbed" interface (see Figure 7-7):

```
<html>
<head><title>Onboarding Mick Mickelsen</title></head>
<body>

  <div id="wip-actions">
    <a rel="cancelWIP" href="...">Cancel</a>
    <a rel="updateWIP" href="...">Update</a>
    <a rel="submitWIP" href="...">Submit</a>
  </div>
  <div> id="wip-inputs">
     <form name="employee" ... >
       <submit name="updateWIP" />
       <input name="field1" ... />
       <input name="field20" ... />
     </form>
     <form name="interviews">
       <submit name="updateWIP" />
       <input name="field21" ... />
       <input name="field37" ... />
     </form>
     <!-- other forms go here -->
  </div>
</body>
```

In this example, each set of collected inputs is wrapped within an HTML FORM element that can be submitted to the WIP engine at any time. For humans, this might be rendered as shown in Figure 7-7.

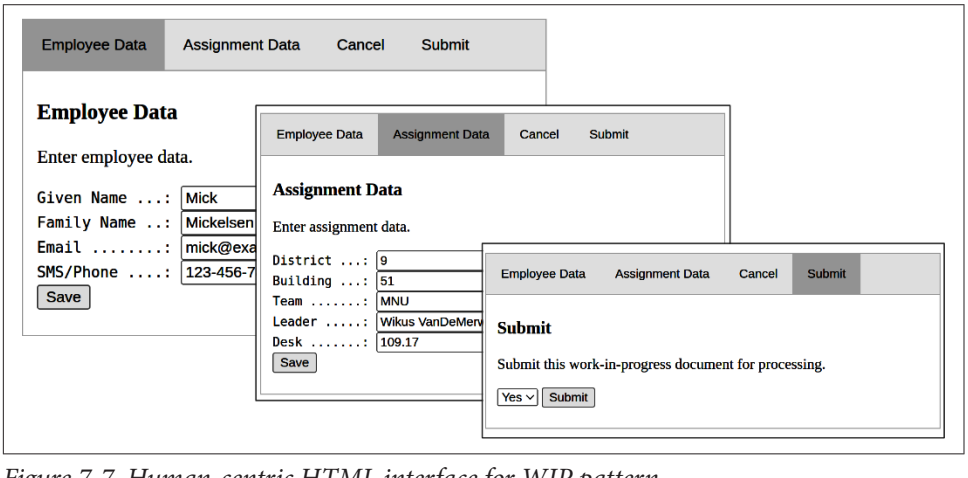

Figure 7-7. Human-centric HTML interface for WIP pattern

See Also

- Recipe 3.5, "Expressing Actions at Runtime with Embedded Hypermedia"
- Recipe 4.9, "Relying on Hypermedia Controls in the Response"
- Recipe 5.4, "Expressing Internal Functions as External Actions"
- Recipe 6.4, "Leveraging HTTP URLs to Support 'Contains' and 'AND' Queries"
- Recipe 7.11, "Enabling Standard List Navigation"
- Recipe 7.12, "Supporting Partial Form Submit"
- Recipe 7.14, "Optimizing Queries with Stored Replays"

7.11 Enabling Standard List Navigation

Most service interfaces need to support list navigation options like getting the "next page," selecting an item from the list, etc. This recipe provides a simple pattern that can be applied to any exposed resource list.

Problem

What's the best way to support paging through a long list of data?

Solution

A good way to add "pagination" support to your service interface is to expose a fixed set of actions that can be applied to any resource list. This set of actions should include not only list navigation actions (first, previous, next, and last) but also actions to pick a particular item from the list and even exit the list navigation.

Here is the list of actions your list navigation should support:

List
: Go to the list (usually, the start of the list).

First
: Move to the first page in the list.

Previous
: Move back to the previous page in the list.

Next
: Move forward to the next page in the list.

Last
: Move to the last page in the list.

Select
: Select an item from the list.

Exit
: Exit the list navigation.

Home
: Go to the "Home" resource (exiting the list).

A common approach is to offer a link in the document with `rel="list"` or `rel="collection"` that "starts" the navigation process. The returned resource then includes links to support the other actions in the preceding list, including the ability to leave the list via `rel="exit"` and/or `rel="home"`. You don't need to support both, but there may be cases where you can do some state cleanup and use `rel="exit"`.

The `rel="select"` action makes it clear that API consumers can use that link to navigate to a single item resource. This is handy when not all members of the list support single reads. Think, for example, of a list of configuration values.

You can enhance this recipe with Recipe 6.4 to add support for creating filtered lists. You can also use Recipe 7.12 to support editing or adding resources to the list. Figure 7-8 shows the list navigation workflow.

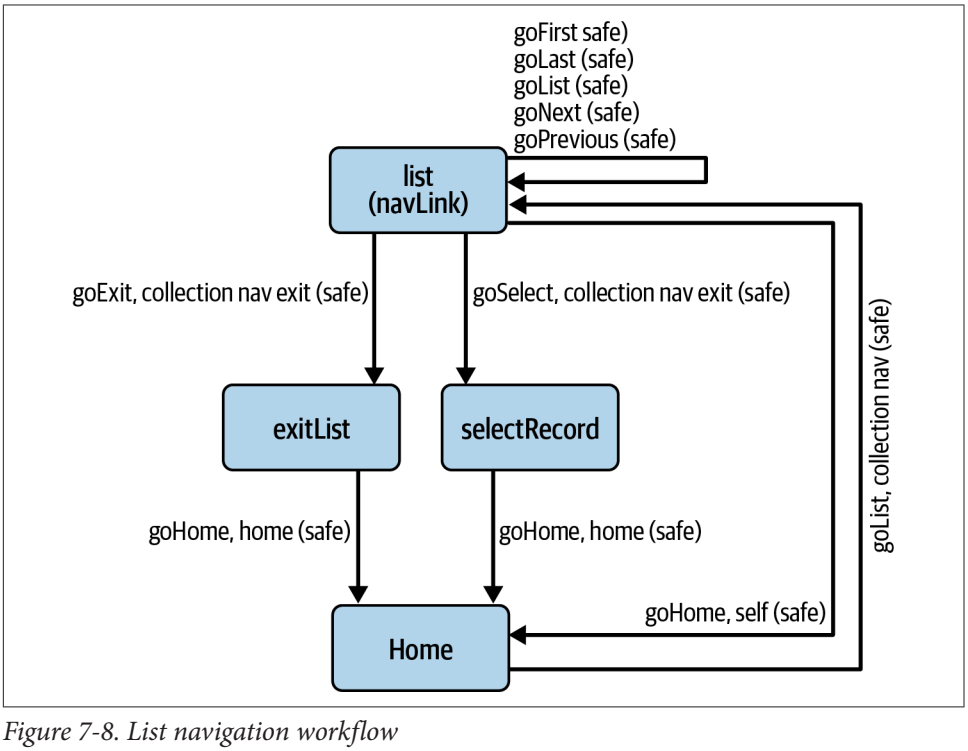

Figure 7-8. List navigation workflow

Example

Usually, you'd see a resource that offers the ability to navigate a list:

```
{"collection" : {
  "title" : "Manage Customers",
  "links" : [
    {"rel" : "home self", "href" : "/customers/"},
    {"rel" : "list collection", "href" : "/customers/list"}
  ]
}}
```

Activating the `list` link then returns the `list` resource with accompanying actions:

```
{"collection" : {
  "title" : "Customer List",
  "links" : [
    {"rel" : "self list collection", "href" : "/customers/list"},
    {"rel" : "home", "href" : "/customers/"},
    {"rel" : "first", "href" : "/customers/list/q1w2"},
    {"rel" : "next", "href" : "/customers/list/t5y6"},
    {"rel" : "exit", "href" : "/customers/p0o9"}
  ],
  "items" : [
   ...
  ]
}}
```

In this example, we can assume this is the "first" page in the list. Note that the `previous` link is not offered here since there is no way to navigate before the start of the list. The `last` link is also missing. This is common in cases where the list is very long (thousands of records) and there is no easy way to compute the last page of a collection.

You'll also notice that the `href` values for `first` and `next` are not semantically meaningful (e.g., "page1," page2," etc.). There is no need to offer link strings that humans can easily read. See the "Discussion" section for more on link values for navigation.

The `exit` link here might point to a temporary resource that asks for confirmation of exiting the navigation (and maybe losing some collected state data). This is common when creating the list is costly and exiting would mean throwing away all that work. The `exit` link may also point to a resource that allows the API consumer to store, collect, or in some other way take advantage of accumulated state data (e.g., a list of selected records, the last one read, etc.).

Here is a resource that might be returned once the end of the list is reached:

```
{"collection" : {
  "title" : "Customer List",
  "links" : [
    {"rel" : "self list collection", "href" : "/customers/list/ht5r"},
```

```
      {"rel" : "home", "href" : "/customers/"},
      {"rel" : "first", "href" : "/customers/list/q1w2"},
      {"rel" : "previous", "href" : "/customers/list/t5y6"},
      {"rel" : "exit", "href" : "/customers/p0o9"}
    ],
    "items" : [
     ...
    ]
  }}
```

Notice that there is no link for the `next` or `last` actions since they would be inappropriate here.

Finally, if the navigation supports selecting or viewing individual resources from the list, you can use the `select` link identifier:

```
{"collection" : {
  "title" : "Customer List",
  "links" : [
    {"rel" : "self list collection", "href" : "/customers/list/ht5r"},
    {"rel" : "home", "href" : "/customers/"},
    {"rel" : "first", "href" : "/customers/list/q1w2"},
    {"rel" : "previous", "href" : "/customers/list/t5y6"},
    {"rel" : "next", "href" : "/customers/list/r4e3"},
    {"rel" : "exit", "href" : "/customers/p0o9"}
  ],
  "items" : [
    { "href" : "/customers/aq1sw2de3",
      "data" : [
        ...
      ],
      "links" : [
        {"rel" : "select item", "href" : "/customers/aq1sw2de3"}
    },
    ...
  ]
}}
```

Discussion

Supporting lists usually seems easy at first—when the list is small. But the effort quickly ramps up as the list begins to grow. It is a good strategy to assume you are supporting a list with thousands of members and implement accordingly. Don't assume it will be easy to access, filter, or display list contents. Do your best to limit the effort involved in responding to list requests, and you'll avoid lots of problems.

It is common to want to offer metadata controls that allow API consumers to control the list of each page of data (e.g., 50, 100, etc.). You can use client preferences (see Recipe 5.5) to handle that. Be sure to always have a default value for page size, too. A

reasonable value varies depending on the list of each resource returned in the list. Usually, 50 is a good start.

When returning resources in a list, you don't need to include the complete resource. Often it is a good idea to just return basic information like summary data, commonly used fields, and a link to a detailed view (via `select` links).

Keep in mind it may be costly to generate a `list` resource. For example, listing all the computers connected to the server on the network might be a long process. In this case, you can compute partial lists—enough to complete a single page—and only compute the following pages if and when the API consumer asks for another one.

It may seem like a good idea to support navigation that allows API consumers to "go to page 11" in your list. But this can be an expensive request. It assumes you already know the contents of the complete list. If you are listing all the users in a live chat room, that membership is likely to change frequently. Even in cases where the list is fixed, if it is quite long (thousands of items), you may be forced to spend vital runtime resources trying to navigate to page 133 in a list of 600 pages.

You can enhance your list navigation support by adding search capabilities (see Recipe 6.4). However, it can be difficult to navigate through a filtered list of data. If you offer this option, it may make sense to internally create a single snapshot of the filter collection, store it in memory, and use that in-memory list as your source for navigation. Of course, now you need to pay the cost of creating a complete filtered collection before you can offer the first page of data!

See Also

- Recipe 3.5, "Expressing Actions at Runtime with Embedded Hypermedia"
- Recipe 4.9, "Relying on Hypermedia Controls in the Response"
- Recipe 5.4, "Expressing Internal Functions as External Actions"
- Recipe 6.6, "Returning HTTP 200 Versus HTTP 400 for Data-Centric Queries"
- Recipe 6.7, "Using Media Types for Data Queries"
- Recipe 7.9, "Returning Most Recently Used Resources"
- Recipe 7.13, "Using State-Watch to Enable Client-Driven Workflow"

7.12 Supporting Partial Form Submit

While input forms are a key element of any hypermedia-driven system, they can be tricky to navigate, especially for machines or automated processes. This recipe offers a simple, standard way to handle form input that reduces the chances for rejected inputs and runtime. This recipe was inspired by UX pioneer Donald Norman, who said, "Think of each action by the user as an attempt to step in the right direction."

Problem

Input forms, especially complex ones, can be difficult for users to successfully complete, especially when the user is a *machine* rather than a person. Features like dependent drop-down lists, detailed client-side validation, and just long lists of input increase the likelihood that the submission will be rejected by the server. How can you design and implement input forms that are easier to navigate and more likely to be completed successfully? How can you make input forms more "machine-friendly"?

Solution

You can do a great deal to improve input forms by supporting the Partial Form Submit (PFS) recipe. The notion of PFS is that the forms can be submitted to the server even if they have not yet been completed. This allows the client application to fill in just parts of the form, save those inputs to the server, and then add more inputs along the way. Once all the inputs have been filled in, the client application can execute a "final submit"—telling the service that all inputs are complete and signaling the service to do any post-input processing.

This recipe is related to the work-in-progress (WIP) recipe (see Recipe 7.10). But this recipe doesn't support long-running (e.g., hours/days), multipage input processes.

The PFS recipe is particularly useful for machine-driven client applications since they can submit one input field at a time and ask the server to validate and store that one input. This makes fixing validation errors much simpler for the machine-driven client. Implementing PFS means returning an input form with a collect of inputs and a set of submission options, including:

`partialSubmit`
: Submit this form and hold on to the inputs. It is not yet complete.

`resetSubmit`
: Clear any inputs you have so we can start again.

`refreshSubmit`
: Return the current state of the form (including any supplied inputs).

`cancelSubmit`
: Cancel the input process completely and clear any inputs you have.

`finalSubmit`
: The client app is done supplying inputs; submit this for final processing.

As each input is submitted (via `partialSubmit`), the service can check to make sure it contains valid content. If it does, the server should store the inputs and then return an updated form to the client application that includes the filled-in values as well as any remaining empty inputs. This continues until the client application selects `cancelSubmit` or `finalSubmit`. At any time, the client application can also use `resetSubmit` to reset the form and start the process form the beginning.

Figure 7-9 shows the PFS workflow.

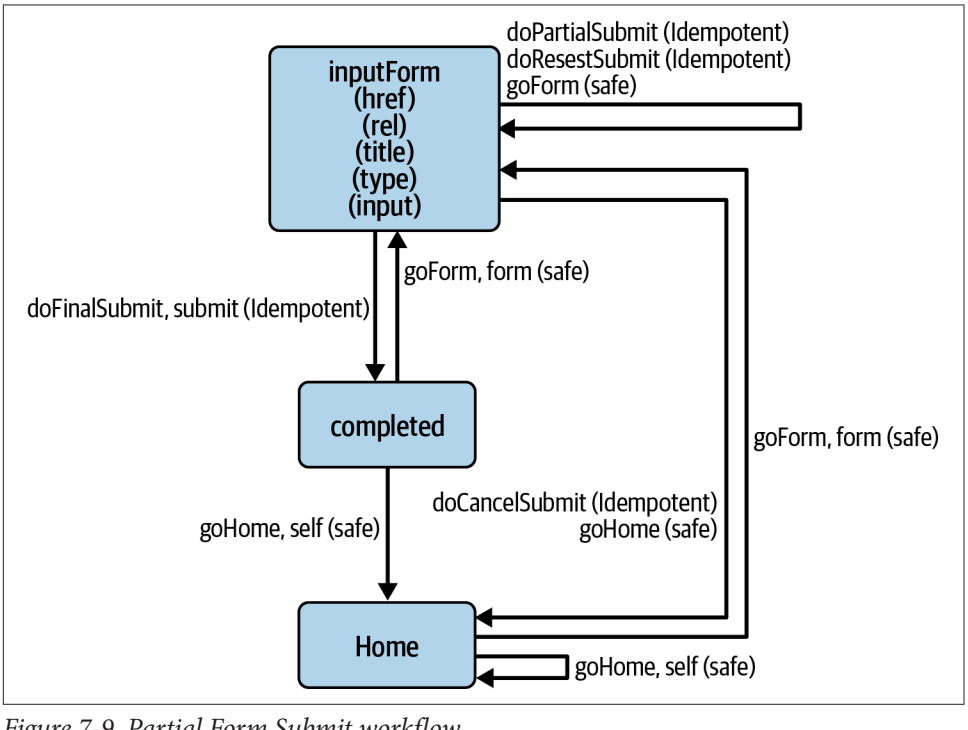

Figure 7-9. Partial Form Submit workflow

Example

Consider the following HTML input form:

```
<form name="search-users" action="/users/filter" method="GET">
  <select name="type">
    <option>customer</option>
    <option>prospect</option>
  </select>
  <select name="region">
    <option>west</option>
    <option>south</option>
  </select>
  <input type="text" name="name" />
  <input type="text" name="salesRep" />
  <input type="submit" name="submit" value="partialSubmit" />
  <input type="submit" name="submit" value="refreshSubmit" />
  <input type="submit" name="submit" value="resetSubmit" />
  <input type="submit" name="submit" value="cancelSubmit" />
  <input type="submit" name="submit" value="finalSubmit" />
</form>
```

Now, let's assume that we have written an autonomic bot that looks for all the records where the `type` is set to "customers" and the `salesRep` value contains "Mork." This bot is pretty simple and fills in values one at a time before executing a `finalSubmit` action. So our bot would make the following three HTTP requests:

```
GET /users/filter?type=customer&submit=partialSubmit
GET /users/filter/?salesRep=Mork&submit=partialSubmit
GET /users/filter/?submit=finalSubmit
```

As each input is sent individually, the service interface can perform validation and provide feedback. For example, if the bot tried to send `type=friend` as the first input, the service can send an HTTP 400 status response to help the client fix the problem and try again (via `refreshSubmit` to reload the form).

Note that the PFS recipe does not assume *all* inputs are required to execute the `finalSubmit` action. Also, once `finalSubmit` has been sent, the service interface can still perform a validation that might result in a 400 HTTP status with corrective action required.

Discussion

The key point of this recipe is to make it easy for users of any kind to submit data to the server. By separating "save my data" from "process this submission," you can simplify interactions and make it easier to fix errors in the input. This really pays off when you are using an automated client to submit that data.

This recipe actually makes it harder to create input forms that require "dependent input validation." For example, it is difficult to support any rule that states, "If you

select `customer` in `inputA`, you *must* not fill in any value for `inputB` and so forth." In this case, you can write code on the server that uses the `finalSubmit` action to validate these dependencies and return error messages with hints for solving the problem. If at all possible, you should avoid these cases, as they can be very difficult for automated clients to resolve.

Since the server is returning the form each time with more inputs filled in, it is also possible for client applications to modify values for already-submitted fields. This is perfectly fine and can be especially handy for human users as they may recognize mistakes made earlier in the process.

See Also

- Recipe 3.5, "Expressing Actions at Runtime with Embedded Hypermedia"
- Recipe 4.9, "Relying on Hypermedia Controls in the Response"
- Recipe 4.10, "Supporting Links and Forms for Nonhypermedia Services"
- Recipe 5.4, "Expressing Internal Functions as External Actions"
- Recipe 6.7, "Using Media Types for Data Queries"
- Recipe 7.9, "Returning Most Recently Used Resources"
- Recipe 7.10, "Supporting Stateful Work in Progress"
- Recipe 7.15, "Synchronous Reply for Incomplete Work with 202 Accepted"

7.13 Using State-Watch to Enable Client-Driven Workflow

While most recipes for workflow services focus on examples where the *service* knows the end goal through code (Recipe 7.3), DSL (Recipe 7.4), or document (Recipe 7.5), there are times when the servers do not know what goal the API consumer (client) has in mind. In these cases, you can use this recipe to create a service workflow where the end goal is known by the client alone and the services provide information to help that client determine when the goal is reached.

Problem

There are times when the client application knows the desired end state and the services do not. In these cases, how do you design workflow to meet client (API consumer) needs? When is this client-driven goal applicable? What are the pitfalls of allowing client applications to "drive the workflow"?

Solution

The solution to this problem of client-driven workflow was first mentioned in Recipe 4.16 when we talked about Defined End Goals (DEGs) and Defined State Goals (DSGs). This recipe focuses on how to enable DSGs. The key to making this work is making sure there is service interface support for sharing client-selected state variables. That means supporting a way for client applications to *tell services* what data to return.

But GraphQL?!?

The data-centric protocol GraphQL is an excellent example of a platform that supports client-driven queries. I won't be using GraphQL in my example here, but if you already have GraphQL in place in your technology mix, you can apply this recipe using that platform.

What is state-watching?

Consider an example where a client application (`temp03`) needs to monitor the temperature sensors in a large building. That client application "knows" a range of proper temperatures for, let's say, the third floor of the building. It also knows that it needs to monitor the temperature sensors on that floor and operate the thermostat for the same zone. This means the client needs to get feedback on each `thermo-sensor` on the floor and access to control the `thermostat` as well.

Now consider another application (`light03`) that needs to monitor movement on the same floor and turn on/off the overhead lights depending on whether people are moving around in the rooms. This application needs access to all the `motion-senors` and `light-switches` on the third floor.

Finally, let's assume there is a single building management service interface, one that supports access to all the sensors and all the actuators throughout the building. This is the service that `temp03` and `light03` "talk to" to do their work. The ideal situation is to have each client request only the data properties it is interested in monitoring and only returning the action elements (FORMS) that each application wants to access.

Client applications need a way to tell the service what information it wants to see for each HTTP request response. This amounts to the client telling the service interface with state values it will "watch." Hence the name of this recipe: *state-watch*.

Is this watchable?

To make this work, the service interface needs to be able to tell the API consumer just what data properties are "watchable"—which properties can be selected for inclusion

in HTTP responses. This is essentially an instance of the list navigation recipe (Recipe 7.11). Since that has been covered earlier, I won't repeat the details here. For our purposes, let's assume the list of available, watchable elements includes all the motion sensors, thermo-sensors, light switches and thermostats for the building. Let's make this easier and assume that our application identity only has access to the element on the third floor of the building. That means making a call to the `watchList` will return available elements.

Now we need to be able to tell the service just which elements on that list each client applications wishes to "see."

I'll take motion sensors for the third floor, please

There are three ways a client application can signal the desired elements to monitor, each with their own advantages and drawbacks:

Query string
: The simplest way to pass the list of watched elements is to include them as part of an HTTP query string. The advantage here is that it is a well-known and easily supported solution. The drawback is that the list may get quite long and risks complicating URL parsing for intermediates.

HTTP prefer headers
: RFC 7240 defines the `Prefer` header, which allows clients to pass metadata information about how the client prefers the service to modify the HTTP response body. The advantage is that this solution keeps the information out of the URL and has an established standard. The drawback is that there is currently no preference property defined (see HTTP Preferences) for use with state-watch lists. However, you can apply to define one, if you wish, or use this header without registering a preference property.

State-watch resource
: Another solution is to define a state-watch resource to associate with client requests. In this case, a separate resource can be used to host the list of elements the service should return. A pointer to this resource can be shared via a `Link` header with the `rel="watch"` identifier. The advantage is that it is a clear and simple solution. The drawback is that it is a *separate* resource that both the API consumer and API provider must honor at runtime.

Data query body
: To prevent problems with long URLs, the state elements to watch can be passed as part of a request body. This is how GraphQL works. The advantage is that there is existing technology for doing this. The drawback is that it requires the use of an unsafe HTTP method (`PUT` or possibly `POST`), and this complicates caching, retries, and other issues.

For various reasons, I recommend the state-watch resource solution. It is the most compatible with proper HTTP protocol usage and avoids the problems related to long lists of data properties. It also allows clients to set their preferences at the start of a workflow session and not have to resend those preferences until it is time to modify the list.

State-watch properties, resources, and actions

Supporting the state-watch pattern involves the follow properties: resources and actions. State-watch properties include:

`watchElementID`
: Unique identifier of a watch element

`watchElementURL`
: URL of a watch element

`watchElementName`
: Text name of a watch element

`watchElementValue`
: Current value of a watch element

`watchElementTag`
: Filter tag of a watch element

Here are the state-watch resources:

`watchListResource`
: Contains the list of all available watchable elements

`watchResource`
: Represents the list of *selected* watchable elements

State-watch actions include:

`doWatchClear`
: Unsafe action to clear any selected watch elements

`doWatchCreate`
: Unsafe action to create a new list of selected watch elements

`doWatchUpdate`
: Unsafe action to update the current list of selected watch elements

`goWatchList`
: Safe action to return the list of available watchable elements

`goWatchResource`
: Safe action to return the list of selected watchable elements

Typical State-Watch Interaction

Using the state-watch resource option as a guide, the process of selecting and reporting to the server state elements the client prefers to watch would look like this:

1. The client locates the `rel=watch` link for the service (see Recipe 7.8).
2. The client selects the elements it wishes to watch.
3. The client uses the `updateWatchList` form and sends the list of watched elements to the service interface using a unique URL generated *by the API client*.
4. Now, on each request to the service (e.g., `listStates`), the status of each element is included in the response.
5. This continues until the client wishes to update (or clear) the `watchList`.

See Figure 7-10 and the "Example" section for details.

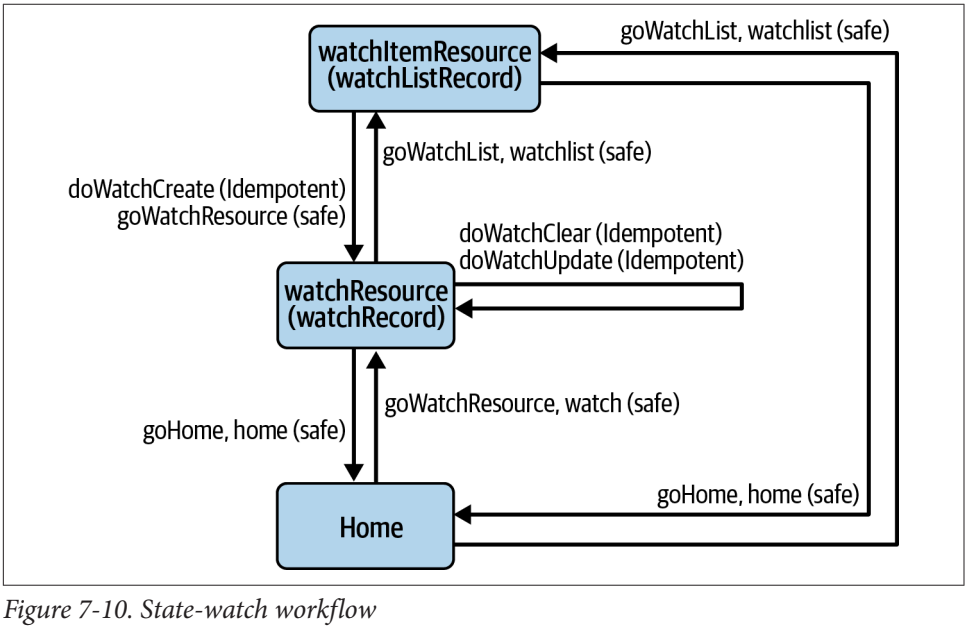

Figure 7-10. State-watch workflow

Example

A service interface that supports the state-watch pattern can advertise this using an HTTP link header and/or a link within the body of a response:

```
**** REQUEST ****
GET /building
Host: api.example.org
Accept: application/vnd.siren+json
...

**** RESPONSE ****
200 OK
Content-Type: application/vnd.siren+json
Link: <<http://api.example.org/building/elements/list>>; rel="watchList"
Link: <<http://api.example.org/building/selected;t6r5e4ed>>; rel="watchSelected"

{"class": ["building"],
  "links" : [
   {"rel" : ["self", "home"],
    "href": "/building"},
   {"rel" : ["watchList", "collection"],
    "href": "/building/elements/list"},
   {"rel" : ["watchSelected", "collection"],
    "href": "/building//building/selected;t6r5e4"}
  ],
  "properties": {...},
  "entities": {...}
}
```

Note that there are two links in the response related to the state-watch pattern. The first (`rel="watchList"`) is the list of available elements.

It is important that each API consumer have its own associated `rel=watchSelected` resource. Generating this URL is mostly the responsibility of the API consumer (remember, the consumer is driving the workflow!). See the "Discussion" section for more on this topic.

Clients can use this to review and select elements to watch. The second link (`rel=watchSelected`) is the list of currently selected elements that are returned with each response:

```
**** REQUEST ****
GET /building/elements/list
Host: api.example.org
Accept: application/vnd.siren+json

**** RESPONSE ****
200 OK
Content-Type: application/vnd.siren+json
Link: <<http://api.example.org/building/elements/list>>; rel="watchList"
Link: <<http://api.example.org/building/selected;t6r5e4ed>>; rel="watchSelected"

{"class": ["building"],
```

```
  "links" : [
   {"rel" : ["home"],
    "href": "/building"},
   {"rel" : ["self", "watchList", "collection"],
    "href": "/building/elements/list"},
   {"rel" : ["watchSelected", "collection"],
    "href": "/building/elements/selected"}
  ],
  "entities" : [
    {"class": ["watch", "element"],
     "properties": {
       "id": "q1w2e3r4",
       "href": "/building/elements/list/q1w2e3r4",
       "type": "thermo-sensor",
       "name": "ts-01",
       "location": "Third floor"
     },
     ...
    {"class": ["watch", "element"],
     "properties": {
       "id": "u8y7t6r5",
       "href": "/building/elements/list/u8y7t6r5",
       "type": "thermostat",
       "name": "stat-0301",
       "location": "Third floor"
     }
  ],
  "actions": [
    {
      "name": "select-elements",
      "title": "Select Elements",
      "method": "PUT",
      "href": "http://api.example.org/building//building/selected;t6r5e4",
      "type": "text/plain",
      "fields": [
        { "name": "elements", "type": "value": "q1w2e3r4,u8y7t6r5" }
      ]
    }
  ]
}
```

In this example, the client will send two elements to the `watchSelected` list (`q1w2e3r4,u8y7t6r5`). Now each response for the related resource (`/building/`) will include the actual status of these two elements:

```
GET /building
Host: api.example.org
Accept: application/vnd.siren+json
...

**** RESPONSE ****
200 OK
```

```
Content-Type: application/vnd.siren+json
Link: <<http://api.example.org/building/elements/list>>; rel="watchList"
Link: <<http://api.example.org/building/selected;t6r5e4ed>>; rel="watchSelected"

{"class": ["building"],
  "links" : [
   {"rel" : ["self", "home"],
    "href": "/building"},
   {"rel" : ["watchList", "collection"],
    "href": "/building/elements/list"},
   {"rel" : ["watchSelected", "collection"],
    "href": "/building/elements/selected"}
  ],
  "properties" {...},
  "entities": [
    {"class":["thermo-sensor"],"id":"q1w2e3r4","name":"ts-01","temp": "-3C"},
    {"class":["thermostat"],"id":"u8y7t6r5","name":"stat-0301","state": "off"},
    ...
  ],
  "actions": [...]
}
```

Note that the two selected elements have been included in the response to the `/building` resource. It is also possible to get just the list of selected elements:

```
GET /building//building/selected;t6r5e4
Host: api.example.org
Accept: application/vnd.siren+json
...

**** RESPONSE ****
200 OK
Content-Type: application/vnd.siren+json
Link: <<http://api.example.org/building/elements/list>>; rel="watchList"
Link: <<http://api.example.org/building/selected;t6r5e4ed>>; rel="watchSelected"

{"class": ["watchSelected"],
  "links" : [
   {"rel" : ["home"],
    "href": "/building"},
   {"rel" : ["watchList", "collection"],
    "href": "/building/elements/list"},
   {"rel" : ["home", "watchSelected", "collection"],
    "href": "/building/selected;t6r5e4"}
  ],
  "properties" {...},
  "entities": [
    {"class":["thermo-sensor"],"id":"q1w2e3r4","name":"ts-01","temp": "-3C"},
    {"class":["thermostat"],"id":"u8y7t6r5","name":"stat-0301","state": "off"},
    ...
  ],
  "actions": [
   {"name": "updateWatchList", "method": "PUT", ...},
```

```
    {"name": "clearWatchList", "method": "DELETE", ...}
  ]
}
```

In this example, not only were the selected elements returned (with their current values), there are also two forms in the response. The `updateWatchList` action can be used to modify the watch list, and the `clearWatchList` action can be used to quickly clear all the elements from the watch list.

Using our previous example of two API consumers (one watching temperatures and the other watching motion and lighting), we could construct another scenario.

Discussion

The state-watch pattern works well when the API consumer needs to evaluate a small set of available data to make a decision and/or commit an action. Monitoring sensors of some type (temperatures, motion, liquids, etc.) is a good use case for the state-watch pattern.

This recipe is one of the more complicated ones in the collection. It requires both client and server to share a resource that's created—at runtime—by the server based on input (unique ID) from the client. Both parties then need to be able to recall that URL when passing HTTP requests and responses.

Because the state-watch pattern relies on two key HTTP resources (the list of available elements to watch and the list of watched elements for each API client), you need to be clear on how to create a unique URL for the client's selected elements. The details of the URL are not interesting, but you will need to be sure the client can generate a unique ID for its own `watchResource` (see Recipe 5.14), and that the client can *remember* the URL and pass it to the service when appropriate. This last step usually means the service should return the URL with each response to that client (via HTTP headers and/or the body).

In the examples shown here, the `updateWatchList` form uses a `text/plain` body to send a comma-delimited list of watched elements. You can use other serializations but will need to work out a way for client applications to know ahead of time how to parse the watch list responses.

See Also

- Recipe 3.5, "Expressing Actions at Runtime with Embedded Hypermedia"
- Recipe 4.9, "Relying on Hypermedia Controls in the Response"
- Recipe 4.14, "Validating Incoming Data"

- Recipe 4.15, “Maintaining Your Own State”
- Recipe 4.16, “Having a Goal in Mind”
- Recipe 5.4, “Expressing Internal Functions as External Actions”
- Recipe 6.5, “Returning Metadata for Query Responses”
- Recipe 7.7, “Exposing a Progress Resource for Your Workflows”
- Recipe 7.15, “Synchronous Reply for Incomplete Work with 202 Accepted”

7.14 Optimizing Queries with Stored Replays

For systems that rely heavily on custom queries, this recipe offers a standardized way to create, manage, and execute stored queries.

You might have expected this recipe to appear in Chapter 6. However, since this recipe involves several resources acting in concert, I decided to place it here in the workflow chapter.

Problem

Some queries are complex and/or expensive to run. Sometimes HTTP queries require extensive filtering parameters, resulting in a very long query string. Since most HTTP servers have a fixed limit on the length of a URL, complex queries may be misinterpreted as a security attack. What can you do to make storing and *replaying* those complex queries safe, cheap, and easy? Along the way, how can you better manage the list of collected queries, even implementing a simple sharing support to allow users or machines to author and then “publish” those complex queries for others to use?

Solution

The first step to making complex queries safe and easy to use is to implement the query itself as a resource. That means expressing the query details as an HTTP body and then using that message body to create a new *query resource* that can then be executed directly upon request. That’s step two, using the query resource URL to execute the query and return results in the response.

Not All URLs Are Equal

There is a slight trick buried in this recipe. API consumers need to know the URL for *executing* a query and may also need to know the URL for *editing* the query. See the “Example” section to see how this recipe handles this.

Essentially, what you're doing is creating a collection of query resources, and that collect needs to be managed, too. This recipe takes advantage of list navigation (Recipe 7.11) to support listing, filtering, editing, and removing query resources.

Finally, an optional element for this recipe is the `shareQuery` action. This allows you to support sending the query resource to someone else to use and/or set the "owner" of the query resource for future management.

There is a collection of metadata associated with stored queries. You can use these values (and others, if you wish) to manage and track the queries that are created and replayed over time:

`identifier`
: Unique identifier for the stored query.

`URL`
: URL of this delayed response document.

`resourceQuery`
: Complete (original) HTTP query.

`cacheTTL`
: How long (in milliseconds) the response should be cached by the client.

`cachingDirectives`
: Additional caching directives for the query.

`description`
: Human-readable text that describes this stored query.

`owner`
: Identifier (URL or text) of the identity that controls this query resource.

`tags`
: Free-form string space-separated values. Use this for filtering or otherwise grouping stored queries.

`dateCreated`
: Indicates the date/time (UTC) this replay resource was first created.

`dateUpdated`
: Indicates the last date/time (UTC) this replay resource was modified.

`dateLastRun`
: Indicates the last date/time (UTC) this stored query was executed.

The `resourceQuery` property holds the full query string passed to the service when the query was created. This may be updated, too. The `cacheTTL` and `cachingDirectives` properties may contain details on the kind of caching metadata that should be

returned with the query response. The `owner` and `tags` properties are handy for keeping track of management details like who is responsible for the resource and how you can group or filter this resource in the collection.

Figure 7-11 shows the stored query replay workflow.

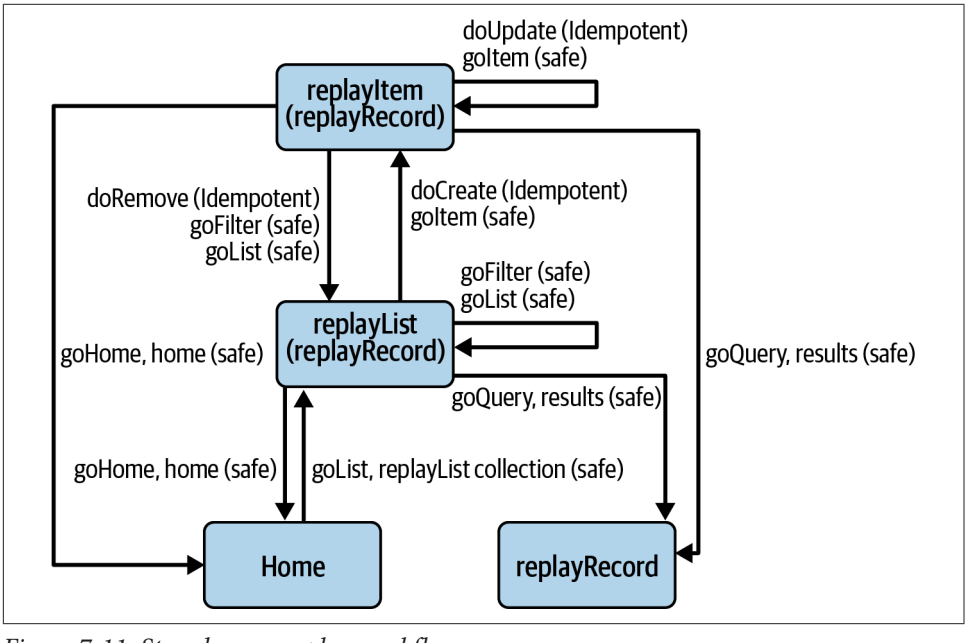

Figure 7-11. Stored query replay workflow

Example

This recipe has two distinct parts. The first part covers creating and replaying query resources. The second part covers managing the list of stored query resources.

Creating and replaying query resources

The first action creates the query resource and stores it on the server. Here's how that looks:

```
**** REQUEST ****
PUT /queries/q1w2e3r4
Content-Type: application/x-www-form-urlencoded
Accepted: application/vnd.collection+json
Length: NN
If-Not-Match: "*"

type=customer&region=south&name=Andrews&balance-is-great-then=10000
&past-due-days-is-great-than=90&salesrep=Mork
```

```
**** RESPONSE ****
200 OK
Content-Type: application/vnd.collection+json
Length: NN

{"collection" : {
  "title" : "Query Results",
  "metadata" : [
    {"name": "q-sent", "value": "type=customer&region=south&name=Andrews& +
      balance-is-great-then=10000&past-due-days-is-great-than=90&salesrep=Mork"},
    {"name": "q-datetime", "value": "2024-12-12:00:12:0012TZ"},
    {"name": "q-status", "value": "sucess"},
    {"name": "q-seconds", "value": "120"},
    {"name": "q-count", "value": "100"},
  "links" : [...]
  "items" : [...]
}}
```

As you can see, the API consumer sent the complex query as a `PUT` to create the new resource (see Recipe 5.15 for details on this recipe). Assuming there are no problems, the service creates the new query resource, executes the query, and returns the results. You'll also notice that the server has returned query metadata (see Recipe 6.5) in the response. This is not a requirement but can be especially handy when relying on this reply recipe to execute queries.

Now that the query is stored on the server, API consumers can call the query resource directly to replay the query:

```
**** REQUEST ****
GET /queries/q1w2e3r4
Accept: application/vnd.collection+json
...

**** RESPONSE ****
200 OK
Cache-Control: public, max-age: 3600
ETag: "w/ki8ju7hy6gt5"
...

{"collection" : {
  "title" : "Query Results",
  "metadata" : [...],
  "links" : [...],
  "items" : [...]
}}
```

Here, the API consumer just needs to issue an HTTP `GET` (not `PUT`) to return the results. In this case, the service has "replayed" the stored query and returned updated results. Note also the use of the `Cache-Control` header to give the consumer directives on how long to hold the query locally. See Recipe 6.9 for more on this recipe.

Managing stored query resources

Creating and replaying stored queries is only part of this recipe. The other part involves managing the collection of stored queries. That means offering `list`, `filter`, `update`, and `remove` actions from Recipe 7.11. In this recipe, I also introduce the optional `share` action.

A good way to expose the management elements of the recipe is to include links in the query responses. They can be shipped as part of the media type body:

```
{"collection" : {
  "title" : "Query Results",
  "metadata" : [...],
  "links" : [
    {"rel":"list", "href":"http://api.company.org/queries/list"},
    {"rel":"filter", "href":"http://api.company.org/queries/filter"},
    {"rel":"update", "href":"http://api.company.org/queries/q1w2e3r4#update"},
    {"rel":"share", "href":"http://api.company.org/queries/q1w2e3r4#share}
  ]
  "items" : [...]
}}
```

The links can also appear as part of the HTTP headers collection. In the following example, the current user context only has permission to share this query, not edit or remove it or see the complete list:

```
**** RESPONSE ****
200 OK
Content-Type: application/vnd.collection+json
Link: <<https://api.company.org/queries/q1w2e3r4#share>>; rel=share
```

I won't go into the details of the `list`, `filter`, `update`, and `remove` actions here. Check out Recipe 7.11 for that. But I will add a comment here about the URLs shown in the preceding example. There's a bit of a challenge when creating URLs for this recipe. We need a URL for *executing* the query, and we need a URL managing the query (e.g., read, edit, delete, and share).

There are a few options available. The one I use is to modify the query resource URL to signal to the service which actions I intend. This might annoy some who don't like to see verbs in URLs. If you wish, you can use a different URL for managing the query resource (e.g., *https://api.company.org/queries/list/q1w2e3r4*) than for executing it (*https://api.company.org/queries/q1w2e34*).

The exact URL you use is up to you. Just remember that you'll need to keep track of them at runtime and keep the API consumer informed of the differences in order to implement this recipe.

Discussion

One of the big advantages of implementing the replay recipe is to solve the problem of very long query strings. I see lots of use of HTTP `POST` to support large query strings; while it works, you don't get the advantage of caching on `POST`. The "create-a-query-resource-then-execute-it" works because you can return the URL of the new resource and do a simple `GET` on that instead. That way you get all the options of caching in the bargain.

This recipe really pays off when you *replay* the queries later. If you don't want to do that, you can skip implementing the resource management part completely and just support the `PUT`/`POST` create and execute part. In those cases, you can keep the created resources around for a short time and then just delete them. Later attempts to call those temporary URLs will just return an HTTP 410 error.

The `owner` metadata is listed here but there is nothing to cover, just how to implement that feature. Typically, this gets folded into user or machine identities and related security issues. That is more than I cover in this edition of the book.

Sharing queries is a way to make it easy for people to reuse existing resources. I often make all the replay queries "public" and include an interface that allows users or machines to select one or more of them and "claim" them for their own use. If you do this, you'll need to deal with who can edit the reusable queries, and you might even need to implement a `copy` action to allow people to get a copy of their own to edit without changing the source query.

See Also

- Recipe 3.5, "Expressing Actions at Runtime with Embedded Hypermedia"
- Recipe 3.8, "Designing for Repeatable Actions"
- Recipe 4.9, "Relying on Hypermedia Controls in the Response"
- Recipe 4.14, "Validating Incoming Data"
- Recipe 5.4, "Expressing Internal Functions as External Actions"
- Recipe 6.4, "Leveraging HTTP URLs to Support 'Contains' and 'AND' Queries"
- Recipe 6.9, "Improving Performance with Caching Directives"
- Recipe 7.16, "Short-Term Fixes with Automatic Retries"

7.15 Synchronous Reply for Incomplete Work with 202 Accepted

There are times when the work that needs to be completed may take some time. In these cases, HTTP (which relies on a synchronous request/response pattern) supports a `202 Accepted` response status. You can use this status to create a pattern that allows for delayed responses that may take hours or even days to resolve.

Problem

Sometimes you know ahead of time that a resource request is likely to take a long time to complete—more than a few seconds. In this case, you need to employ a "delayed response" pattern that tells the client, "This will take some time," and supplies enough metadata to allow the client to decide how to handle this delay. So, what does it take to establish and support this pattern? What metadata should be supplied to the client? What possible workflow options should be supported?

Solution

In cases where the response to an HTTP request is likely to be delayed (take more than a few seconds), you can use a delayed response pattern. This is done by returning a `202 Accepted` status from the service along with metadata (via body and/or headers) that informs the client application of the status of the request, plus options to take additional action (like `refresh` or `cancel`).

The initial response can be returned even before the requested work (e.g., save a record, compute a regression, etc.) is started. See the "Example" section for a typical response body and possible headers.

Response bodies for the delayed response can include the following metadata:

`identifier`
: Unique identifier for this delayed response

`acceptedURL`
: URL of this delayed response document

`completedURL`
: URL of the completed status resource

`failedURL`
: URL of the failed status resource

`description`
: Human-readable text that describes this document

`refresh`
: A value (in milliseconds) that represents the refresh interval

`percentCompleted`
: A value (0–100) of completion

`status`
: Indicates document status (pending, working, completed, or failed)

`dateCreated`
: Indicates the date/time (UTC) this delayed response was first created

`dateUpdated`
: Indicates the last date/time (UTC) this delayed response was modified

`dateEstimated`
: Indicates the date/time (UTC) this delayed request is expected to complete

Common actions that may also appear include:

`goAccepted`
: Navigate to the delayed response resource (safe)

`doCancel`
: Request to cancel this outstanding delayed task using `DELETE` (idempotent)

`goCompleted`
: Navigate to the completed resource (safe)

`goFailed`
: Navigate to the failed resource (safe)

`goHome`
: Navigate to the "Home" resource (safe)

Some of this metadata can be returned using `Link` headers, too.

Figure 7-12 shows an HTTP 202 Accepted workflow.

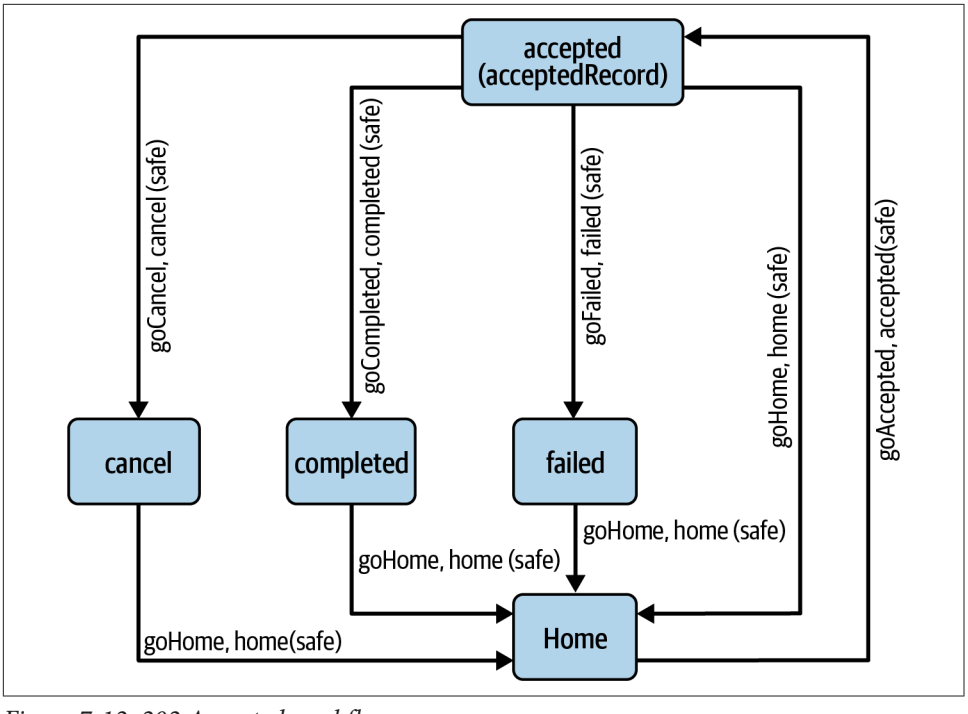

Figure 7-12. 202 Accepted workflow

Example

A typical example of the delayed response pattern looks like the following.

First, the API consumer sends a request to the service interface, and the service interface starts (or queues up) the work and returns a `202 Accepted` response:

```
**** REQUEST
PUT /services/compute-results HTTP/1.1
Content-Type: application/vnd.collection+json
...

{"collection": {...}}

**** RESPONSE
202 Accepted HTTP/1.1
Content-Type: application/vnd.collection+json
Link: <http://api.example.org/services/compute-results/q1w2e3>;rel=self;
  refresh=60000
Link: <http://api.example.org/services/cancel-form/q1w2e3>;rel=cancel

{"collection":
  {
    "links": [
```

```
      {"rel":"self", "name":"acceptedURL",
      "href":"/services/compute-results/q1w2e3"}
    ],
    "items": [
      {
        "href": "/services/compute-results/q1w2e3",
        "data" : [
          {"name":"identifier", "value":"q1w2e3"},
          {"name":"description",
            "value":"Compute net present value for property"},
          {"name":"refresh", "value":"60000"},
          {"name":"percentCompleted", "value":"10"},
          {"name":"status", "value":"working"},
          {"name":"dateCreated", "value":"2024-02-01:22:11:00"},
          {"name":"dateUpdated", "value":"2024-02-01:22:11:00"},
          {"name":"dateEstimated", "value":"2024-02-01:22:15:00"}
        ]
      }
    ]
  }
}
```

Next, the API consumer uses metadata in the response to periodically refresh the DR-Doc (delayed response document) to check the status of the work:

```
**** REQUEST
GET /services/compute-results/q1w2e3 HTTP/1.1
Accepted: application/vnd.collection+json
...

**** RESPONSE
200 OK HTTP/1.1
Content-Type: application/vnd.collection+json
Link: <http://api.example.org/services/compute-results/q1w2e3>;rel=self;
  refresh=60000
Link: <http://api.example.org/services/cancel-form/q1w2e3>;rel=cancel

{"collection":
  {
    "links": [
      {"rel":"self", "name":"acceptedURL",
      "href":"/services/compute-results/q1w2e3"}
    ],
    "items": [
      {
        "href": "/services/compute-results/q1w2e3",
        "data" : [
          {"name":"identifier", "value":"q1w2e3"},
          {"name":"description",
            "value":"Compute net present value for property"},
          {"name":"refresh", "value":"60000"},
          {"name":"percentCompleted", "value":"75"},
          {"name":"status", "value":"working"},
```

```
          {"name":"dateCreated", "value":"2024-02-01:22:11:00"},
          {"name":"dateUpdated", "value":"2024-02-01:22:14:00"},
          {"name":"dateEstimated", "value":"2024-02-01:22:15:00"}
        ]
      }
    ]
  }
}
```

Once the DR-Doc has a status of `completed` or `failed`, the API consumer follows the appropriate link:

```
**** REQUEST
GET /services/compute-results/q1w2e3 HTTP/1.1
Accepted: application/vnd.collection+json
...

**** RESPONSE
200 OK HTTP/1.1
Content-Type: application/vnd.collection+json
Link: <http://api.example.org/services/npv/p0o9i8u7y6>;rel=completed
Link: <http://api.example.org/services/compute-results/q1w2e3>;rel=self;
  refresh=60000

{"collection":
  {
    "links": [
      {"rel":"self", "name":"acceptedURL",
      "href":"/services/compute-results/q1w2e3"}
    ],
    "items": [
      {
        "href": "/services/compute-results/q1w2e3",
        "data" : [
          {"name":"identifier", "value":"q1w2e3"},
          {"name":"description",
            "value":"Compute net present value for property"},
          {"name":"percentCompleted", "value":"100"},
          {"name":"status", "value":"completed"},
          {"name":"dateCreated", "value":"2024-02-01:22:11:00"},
          {"name":"dateUpdated", "value":"2024-02-01:22:16:00"},
          {"name":"dateEstimated", "value":"2024-02-01:22:15:00"}
        ],
        "links": [
          {"rel":"completed", "href":"/services/npv/p0o9i8u7y6"}
        ]
      }
    ]
  }
}
```

Finally, if available in the response, the API consumer may activate the `cancel` action to abandon the request.

Discussion

The key to a successful delayed response implementation is to include as much metadata as possible in the interim responses. The most important is the `refresh` interval and the `status` value. `dateEstimated` and `percentCompleted` are also helpful values.

Most API consumers will be unable to do much with an unexpected `202 Accepted` HTTP response from the service interface. For this reason it is important to tell API consumers ahead of time which actions might return a `202` status code. This should be done in the documentation of the interface itself.

Not all DRs (delayed responses) will support the ability to cancel the request. If the underlying work is limited to handling a long computation (like our net present value example) or generating a long report, then supporting cancel is pretty straightforward. However, if the request involves changing state somewhere (writing a record, deleting or adding data for one or more existing records, etc.), then supporting the cancel option is more involved. See Recipe 7.17 for more on supporting the cancel option for RESTful web services.

Depending on the representation format used in delayed document responses, the cancel action might be represented as an inline FORM or a link, or a link to a FORM. When implementing responses, be sure to follow the conventions for representing hypermedia actions in the body of the response.

It can be tempting to use HTTP headers to contain lots of delayed response metadata, but it is best to keep header use to a minimum. This is especially true in cases where you want to communicate transient information like `percentCompleted`, `dateUpdated`, `status`, etc. Including variable data in headers complicates caching.

The actual URL values used throughout the delayed response process (`acceptedURL`, `completedURL`, `failedURL`, or `cancelURL`) are not important. It is not important, for example, that the `acceptedURL` is the same as the `completedURL`.

It is a good idea to keep the completed delayed response documentation in storage for at least some set period of time. They might come in handy for reporting, audit purposes, internal debugging, or some other needs.

See Also

- Recipe 3.5, "Expressing Actions at Runtime with Embedded Hypermedia"
- Recipe 3.7, "Enabling Interoperability with Inter-Service State Transfers"
- Recipe 4.6, "Managing Representation Formats at Runtime"
- Recipe 4.9, "Relying on Hypermedia Controls in the Response"
- Recipe 5.4, "Expressing Internal Functions as External Actions"

- Recipe 5.10, "Publishing API Metadata"
- Recipe 6.5, "Returning Metadata for Query Responses"
- Recipe 7.10, "Supporting Stateful Work in Progress"

7.16 Short-Term Fixes with Automatic Retries

There will be times when an HTTP request fails due to a network problem. Often, the network problem is temporary and will resolve itself within a matter of seconds. This recipe offers a structured way to build in "automatic retries" to your service interface that allows you to try the unsuccessful network call again before giving up and returning a 5xx HTTP error.

Problem

There are times when transient network errors or temporary outages will impede the completion of an HTTP request. What do you need to do to automatically retry the request in cases where the network problem is intermittent? How do you make sure that problems that persist (nontransient) don't end up blocking your service requests entirely?

Solution

The first thing to keep in mind when arranging to retry requests is that you need to make sure the request itself is idempotent. In HTTP, `GET`, `PUT`, and `DELETE` are designed as idempotent on the network. This is usually sufficient, but not always. See the "Discussion" section for details.

This recipe, like Recipe 7.19, deals with internal server-side programming details; all the elements here should be treated accordingly. This is not information that should be easily reachable on the web.

Second, you need to make sure to inspect the initial response to your request to determine if a retry is warranted. For example, if the response is an HTTP 4xx status (meaning a client-side problem), then retrying that same request without modification will not result in success. If the error condition is an HTTP 5xx status, it may be helpful to retry the same request. Finally, if the error happens at the local connection (e.g., "network connection lost"), then it makes sense to attempt a retry.

If you'd like to learn more about transient network problems and how to deal with them, check the article "Transient Fault Handling" at Microsoft's Azure documentation portal.

There are a handful of options when it comes to handling transient network problems; some are better than others. The following are the reasonable possibilities, in order of preference, along with some comments:

Exponential backoff (EBO)
: The best option is to try an exponential backoff or EBO retry. For example, after experiencing a failure, you can wait two seconds and try again. If that fails, wait 4 seconds, then 16. It is not a good idea to retry the same request more than three times (see the "Discusson").

Incremental backoff (IBO)
: You can also arrange an incremental backoff value (e.g., four seconds) for your repeated attempts. That means you'd first wait 4 seconds, then 8, then 16 before giving up.

Regular interval retries (RIR)
: You can get success by trying at the same regular interval (e.g., every three seconds). One of the downsides of this approach is that if there are multiple instances of the same service attempt retries at the same rate, you could be causing more problems than you think.

Immediate retry (IR)
: If the loss of network is very brief, it could make sense to immediately try again. This can work if the problem is happening at a gateway or some other cluster implementation that is using multiple machines to route the same request. If you use this option, be sure not to try it more than a couple of times since you're really not giving much of a chance for network conditions to change.

Randomized retry (RR)
: Randomized retries can help avoid bottlenecks caused by IR and other options. In this case, you can generate a random wait time (keep it within 15 seconds) and try multiple times. The downside to this technique is that it might randomly generate long wait times (all 3 tries over 10 seconds apart) when all you really needed was to wait a couple of seconds for the network connection to be restored.

Once you settle on an option, you can set up your retry parameters in a configuration resource (see Recipe 5.5). This resource should established for each service interface (or workflow task—see Recipe 7.1). The exact metadata in the configuration depends on which method you choose (see the "Example" section), but here's a list of important properties for a complete `retry` block:

`Method`
: The retry option supported (EBO, IBO, RIR, IR, or RR)

`Max-Retries`
: The maximum number of times the request should be retried

`Max-Wait-Seconds`
: The maximum number of seconds to wait before a retry attempt

`Starting-Value-Seconds`
: The number of seconds to wait before the first retry attempt

`Increment-Value-Seconds`
: The number of seconds to wait for each incremental retry

If the retry attempts fail, the service should report the last HTTP status and request details received.

Keep in mind that the target service *might* assume your repeated attempts are a denial of service attack (see the "Discussion").

You also need to make sure to record your attempts in a local log file for later review. You do not need to report this information to the workflow progress record (see Recipe 7.7) since this is an internal detail that only service interface developers should see.

Example

The following are example metadata properties you can use when recording retry options in a configuration file or other resource (see Recipe 5.5). Here is an example of retry configuration values expressed in XML:

```
<!-- For EBO methods -->
<retries>
  <method>EBO<method>
  <starting-value-seconds>2</starting-value-seconds>
  <max-retries>3</max-retries>
</retries>

<!-- for IBO method -->
<retries>
  <method>RIR<method>
  <increment-value-seconds>3</increment-value-seconds>
  <max-retries>3</max-retries>
</retries>
```

```
<!-- for RIR method -->
<retries>
  <method>RIR<method>
  <starting-value-seconds>2</starting-value-seconds>
  <increment-value-seconds>3</increment-value-seconds>
  <max-retries>3</max-retries>
</retries>

<!-- for IR method -->
<retries>
  <method>IR<method>
  <max-retries>3</max-retries>
</retries>

<!-- for RR method -->
<retries>
  <method>RR<method>
  <max-retries>3</max-retries>
  <max-wait-seconds>10</max-wait-seconds>
</retries>
```

The `retries` block shown here can be added to the existing client preferences document (see Recipe 5.5) or some other configuration resource that the service uses. Be careful to *not* expose this internal information to unauthorized parties.

Discussion

This recipe is designed to *hide* transient network problems and try to resolve them without disrupting the flow of traffic on the network. This has its downsides. You may be dealing with a regularly problematic connection and never notice it unless you inspect your local service logs. For this reason, it's a good idea to review your local logs often.

It is important to remember that repeated attempts to execute the same request multiple times *might* be flagged as malicious behavior (e.g., a denial of service attack) by the target service. This includes making the exact same request as well as being the "source" for lots of repeated requests for various related URLs. Be conservative in setting up your automatic retries and do everything you can to act as a "good netizen."

When logging retries, be sure to log them as warnings and not errors. Once the retries fail, you can report the last attempt as an error. This keeps your logs easier to read and process when you are just looking for retries, for example.

If your log review shows the same services repeatedly causing retry attempts, consider fixing that connection (if you can) or replacing the service with something more reliable.

The EBO approach is handy for M2M interactions. They will not be bothered by the growing lag time between attempts. If you are coding a human-to-machine interface (e.g., a UI instead of an API), the better approach is to implement an IR retry and stop sooner.

Keep in mind that any delays in making requests may get compounded by a long set of sequential actions. For example, if you have 3 sequential requests going through the same faulty gateway and your maximum wait time is 10 seconds for 3 attempts, those 3 requests might add up to as much as 90 seconds of delay. The only way to avoid this possibility is to rely on workflow-compliant services (see Recipe 7.1) that support `jobMaxTTL` and `taskMaxTTL` options.

Conversely, when applying retries to a set of parallel actions, the maximum delay in this case might be limited to 30 seconds—still a long time!

See Also

- Recipe 3.5, "Expressing Actions at Runtime with Embedded Hypermedia"
- Recipe 3.8, "Designing for Repeatable Actions"
- Recipe 5.12, "Standardizing Error Reporting"
- Recipe 5.16, "Providing Runtime Fallbacks for Dependent Services"
- Recipe 7.7, "Exposing a Progress Resource for Your Workflows"
- Recipe 7.17, "Supporting Local Undo or Rollback"

7.17 Supporting Local Undo or Rollback

Whenever possible, your service interface should offer API consumers an undo option—the ability to reverse the last action. This might be reversing a data update, removing the resource that was just created, or unwinding a process that was executed previously. There are a few challenges to face, including the likelihood that the underlying service does not offer an undo action on its own. However, that doesn't mean the API can't offer one anyway.

Problem

It is a good practice to support an undo option for the most recent interface action (e.g., add, edit, or delete). However, not all services (the code behind the API) will offer an undo option. What does it take to support undo even for cases where services do not support it? Under what conditions is it recommended versus when is it not recommended to provide an undo option?

Supporting local undo is a kind of API proxy pattern. Check out Recipe 5.17 for a more general approach to the proxy pattern.

Solution

As a general practice, all services are better off when they offer some level of undo functionality. The ability to reverse the last action is assumed for most all human-centric interfaces and doing the same for machine-centric interfaces is just as valid.

When services provide their own undo option, you should definitely expose that through the external action interface (see Recipe 5.4). In these cases, supporting undo is easy; just implement the service option in your API. However, there will be times when an undo action is not available through the underlying service(s). In that case, it is a good idea to implement undo at the API level whenever possible. That means working up a "faux undo" that works with the underlying service (or services).

Direct undo

For simple CRUD-style services, implementing an undo is pretty straightforward. You need to implement the service action (e.g., `updateCustomer`), keep a history of undoable actions in a local durable storage index by an identifier (`contextId`), and expose your own external action (for example, `undoUpdateCustomer` that takes the `contextId`).

When implementing your local undo functionality, it is wise to establish a time frame for valid rollbacks. For example, you might offer to reverse the action as long as the rollback request comes within five seconds of the original action. This reduces the likelihood that undoing something from last week will have adverse affects in other parts of the system.

When you are dealing with simple CRUD-style services, you can consider the "direct undo" approach. When supporting changes to multiple underlying services, or in cases where you are not sure of the implications of implementing a direct reversal of an action, you can try the "undo with delays" approach instead.

Supporting undo with delays

A less worrisome way to support undo is to simply delay the requested change. For example, you can support the `updateCustomer` action and, when it is executed, you put the request "on hold" for some set time (e.g., five seconds). If, in that time frame, the API caller discovers its mistake and sends the `undoUpdateCustomer` message, you can simply ignore the original request. In this case, the underlying service(s) never

sees the original `updateCustomer` action and there is no possibility that undoing a committed transaction will invalidate some other part of the system.

The downside of this approach is that you are committing to a built-in delay of some value for all write operations that support undo. For this reason, it is a good idea to only use this delay approach for actions that can support delayed completion without breaking any performance commitments. For example, use cases such as nightly batch updates, patterns that typically work via queues, etc., make good candidates for the delayed undo.

Example

There are two possible implementation patterns for supporting undo for your service interfaces: direct and delayed.

Direct undo

For the direct approach, consider a simple CRUD-style service that has this as its internal function model:

```
// internal service code
function internal_actions(action, id, object) {
  var rt = null;
  switch(action) {
    case "create":
      rt = create(id, object);
      break;
    case "read":
      rt = read(id);
      break;
    case "update":
      rt = update(id, object);
      break;
    case "delete":
      rt = remove(id);
      break;
  }
}
```

One way to support undo for your API is to add a new action that looks like this:

```
// external actions in api code
function createAction(id, object, context) {
  var rt = null;
  if(writeLogAction(id, object, context)) {
    rt = internal_actions("create", id, object);
  }
  return rt;
}

function undoCreateAction(id, context) {
```

```
  var rt = null;
  rt = readLogAction(id, context);
  if(isUndoable(rt)) {
    rt = internal_actions("remove", id);
  }
  else {
    rt = errNotUndoable(id, context);
  }
  return rt;
}
```

Undo via delayed execution

Implementing rollback via a delayed execution looks slightly different. You can implement this option by queuing up the unsafe operation (create, update, or delete) with a delay. Then if needed, you can cancel the change *before* it is actually committed to the backend service.

In the next example, there is an underlying operation that commits changes to three sources: `customers`, `orders`, and `products`. Here's a simplified version of how that might work in code:

```
// external action
function scheduleUpdate(inputProperties) {
  // sort out properties to three sources:
  var customers = {...};
  var orders = {...}
  var products = {...}
  context = setTimeout(
    function(){updateSources(customers, orders, products);}, 5000
  );
  return context;
}

// external action
function cancelUpdate(context) {
  clearTimeout(context);
}

// internal function
function updateSources(customers, orders, products) {
  var rt = null;
  // commit all three sources in parallel
  rt = updateWith(customers, orders, products);
  return rt;
}
```

The HTTP exchange to support this use case might look like this (in HTML):

```
**** REQUEST ****
PUT /update HTTP/1.1
Host: api.example.org
```

```
Content-Type: x-www.form-urlencoded
Accept: text/html
...

customerId=q1w2e3&orderId=p0o9i8&productId=5t6y7u&...

**** RESPONSE ****
HTTP/1.1 202 Accepted
Content-Type text/html
...

<p>Order Updated. You can cancel this order in the next five seconds.</p>
<form name="rollback" method="put" action="http://api.example.org/update/r4t5">
  <input type="hidden" name="contextId" value="r4t5" />
  <input type="submit" value="Undo Order Update" />
</form>
```

Notice that the HTTP exchange uses `HTTP 202 Accepted` as the response. This is not a requirement for supporting rollbacks via delayed execution but can be helpful for API client applications since it makes the delay explicit in the exchange. You can also just return `200 OK` or `201 Created` and still support the undo pattern.

Discussion

In the case of the direct undo pattern, make sure your current running context (e.g., the logged-in user context) has rights to complete all the underlying actions needed for the undo. For example, if the logged-in user doesn't have access rights to HTTP `DELETE`, that logged-in user should not have access rights to the undo functionality.

Even when the underlying service(s) code supports simple CRUD actions, if your API code relies on unsafe actions (add, edit, or delete) against more than one underlying service (e.g., `updateCustomer` and `updateSalesLog`), you should consider using the delay method for supporting local undo.

This recipe is designed to support rollback within your own service interface. See Recipe 7.17 for details on how to handle rollbacks across multiple services.

It might seem that implementing your own undo functionality will cause problems for underlying services. This is not true as long as your API code sticks to implementing the exposed functionality of the service. In other words, if the service supports a remove operation, it should not matter if the remove is used on a record three weeks old created by someone else, as opposed to using the remove operation on a record your own API consumer created five seconds ago.

One of the downsides of using the delay approach is that you are adding wait states to your service interface. A couple of seconds may not seem like a big deal, but it can add up quickly. Consider the possibility that you add a wait of three seconds to your API, and there are three underlying services called by that API. Now assume that each of those three underlying services has three-second wait states, too. And so forth. Before you know it, we could be looking at 15 to 30 seconds of fixed delays in our workflow! That's fine if we're looking at overnight batch updates, but it would be a real mess if these APIs are used to provide instant updates to our stock trading platform.

See Also

- Recipe 3.6, "Designing Consistent Data Writes with Idempotent Actions"
- Recipe 3.9, "Designing for Reversible Actions"
- Recipe 4.3, "Coding Resilient Clients with Message-Centric Implementations"
- Recipe 5.16, "Providing Runtime Fallbacks for Dependent Services"
- Recipe 5.17, "Using Semantic Proxies to Access Noncompliant Services"
- Recipe 7.16, "Short-Term Fixes with Automatic Retries"
- Recipe 7.18, "Calling for Help"

7.18 Calling for Help

There are times when a workflow problem cannot be easily resolved by a machine at runtime. When that happens, it is a good idea to simply call for help by sending an alert to a person who can sort out the problem and then, if possible, restart the workflow or cancel it completely.

Problem

What do you do when a workflow error has occurred and there is no easy way to resolve it? What is the bare minimum information needed to send a call for help?

Solution

There will be times when your carefully crafted workflow request will fail badly. Sometimes you can fix the problem using an automatic retry (see Recipe 7.16), but that may not work. When that happens, it is best to simply call for help from a responsible person and let them sort it out.

For example, some network error may be causing requests to fail and there is nothing that can be done to fix that at the moment. When that happens, your workflow processor should send an alert (often an email along with an SMS message) to some predetermined person responsible for handling these kinds of problems.

The alert should include (or point to) the following:

- The workflow description (Recipes 7.3, 7.4, and 7.5)
- The workflow progress resource (Recipe 7.7)
- The workflow shared state (Recipe 7.1)
- Any other pertinent information, such as error reports, traces, etc.

This can be either shipped as a ZIP file collection attached to an email or as pointers (URLs) to the relevant documents (assuming the person has the access rights to all the resources).

The person in charge of the incident can then try to sort out the problem and, if it can be fixed, continue or restart the workflow request. If neither of those work, the person can cancel the job completely.

This should be written up as an incident and placed in a collection that is reviewed regularly. This incident collection can become the start of a bug fix or redesign to help reduce calls for help in the future.

Example

The first step in supporting this recipe is including a responsible party on the workflow job description. For document-driven workflow (Recipe 7.5), that would look like this (see the `contact` section of the HTML version of the workflow document):

```
<html>
  <head>
    <title>Shopping Checkout Workflow</title>
  </head>
  <body>
    <h1>Shopping Checkout Workflow</h1>
    <div class="job">
      <span class="jobID">q1w2e3r4t5</span>
      <span class="jobStatus">working</span>
      ...
      <div class="contact">
        <span name="person">Mook Maundy</span>
        <span name="email">mook@example.org</span>
        <span name="sms">1234567890</span>
        <span name="voice">9086574321</span>
      </div>
      <div class="tasks">
        <div class="task">
```

```
            <span class="taskID">p0o9i8u7y6</span>
            <span class="taskStatus">completed</span>
            ...
          </div>
          ...
        </div>
      </div>
    </body>
  </html>
```

Note that the `contact` block contains `name`, `email`, `voice`, and `sms` properties. These values might not all be supplied, but at least `email`, `voice`, or `sms` should appear as part of the `contact` block. These values may also be group values (e.g., `support@example.org`) or a general help line (e.g., `1-800-PLS-HELP`).

If a halting error occurs (one that cannot be automatically resolved), an alert should be automatically sent using `email` and/or `sms` with the related data or a URL pointing to that data. For example, this error might automatically generate the following incident message sent as an attachment to the `email` address:

```
{"collection": {
  "title": "CFH Incident Report q1w2e3r4",
  "links": [
    {"rel": "job", "href":"..."},
    {"rel": "progress", "href": "..."},
    {"rel": "sharedState", "href": "..."},
    {"rel": "errorReport", "href": "...}
  ]
}}
```

This example is a machine-readable version. The same data can be sent in a human-friendly format, as well.

Discussion

Implementing this recipe is an important part of a healthy workflow design. You can never be sure what might happen at runtime, and the ability to simply alert a human to sort out the problem is essential.

While not mentioned here, it may also help to set some type of priority codes for your workflow alerts. This goes beyond simple workflow design and is best handled by an existing incident management platform.

Generating a call for help incident is rather simple when you are using workflow documents (Recipe 7.5), but it is a bit more involved if you are using DSL or code to describe workflow. In the case of DSL workflow (Recipe 7.4), you probably need to generate a resource that holds a nonexecutable version of the workflow script. For

cases where the workflow is expressed as source code (Recipe 7.3), you will likely be unable to share that code and may need to generate some tracing or other internal information to share in the report.

Be sure that your call for help records *do not* publicly share any private data, proprietary information, or other security-related data that could be exploited.

You can use the output from the call for help system to publish on a dashboard to indicate the general health of your overall system. This can help you track the most common call for help incidents and target them for bug fixes or updates. It can also give you a sense of the ebb and flow of incidents (do they happen at common times of the workday? etc.). Finally, you can track the time it takes to resolve incidents as a measure of your response quality and adjust your teams, if needed.

Depending on your current environment, you may be able to use your existing bug or incident reporting system to automatically generate a high-priority ticket that kicks off its own alerting and routing workflow. This is not required but would likely be easier to support than setting your own internal email/sms/voice alerting process for alerts.

If you are generating your own alerts, you probably need to set up a `list` resource (Recipe 7.11) and maybe even a work-in-progress (Recipe 7.10) interface to track and manage your incidents. At this point, you are dangerously close to implementing your own workflow support platform and probably need to check into some off-the-shelf solutions for your call for help program.

See Also

- Recipe 3.3, "Sharing Domain Specifics via Published Vocabularies"
- Recipe 4.11, "Validating Data Properties at Runtime"
- Recipe 5.16, "Providing Runtime Fallbacks for Dependent Services"
- Recipe 6.5, "Returning Metadata for Query Responses"
- Recipe 7.7, "Exposing a Progress Resource for Your Workflows"
- Recipe 7.17, "Supporting Local Undo or Rollback"

7.19 Scaling Workflow with Queues and Clusters

One of the challenges of supporting workflow on the web is the possibility of high-traffic loads (either sporadic or constant). This recipe covers some internal programming you can do to help better handle times when request volume might cause a backup in processing.

Problem

How can you make sure that heavy traffic doesn't cause workflow services to crash or operate unreliably? What can be done behind the scenes to make it easier to handle large traffic volumes and/or flaky workflow processing without causing the service interface consumers to change their own behaviors?

Solution

If you are concerned that your workflow services might get overrun with high-volume traffic or jobs that run too long, you can use two programming or architecture techniques behind the interface to improve reliability and scalability: queues and clusters.

This recipe, like Recipe 7.16, deals with internal server-side programming details; all the elements here should be treated accordingly. This is not information that should be easily reachable on the web.

Use queues to scale request traffic

By adding a message queue between the service interface and the service itself, you can reduce the time it takes for a request to get *acknowledged* even if it will take some time before the request is *processed* (Figure 7-13).

This technique is especially easy if you are expressing workflow requests as documents (Recipe 7.5). By adding a queue between the interface and the service, you can quickly return an HTTP `202 Accepted` when the request arrives and then log the request into a persistent queue. The service can then read messages off the queue and process them as they are available.

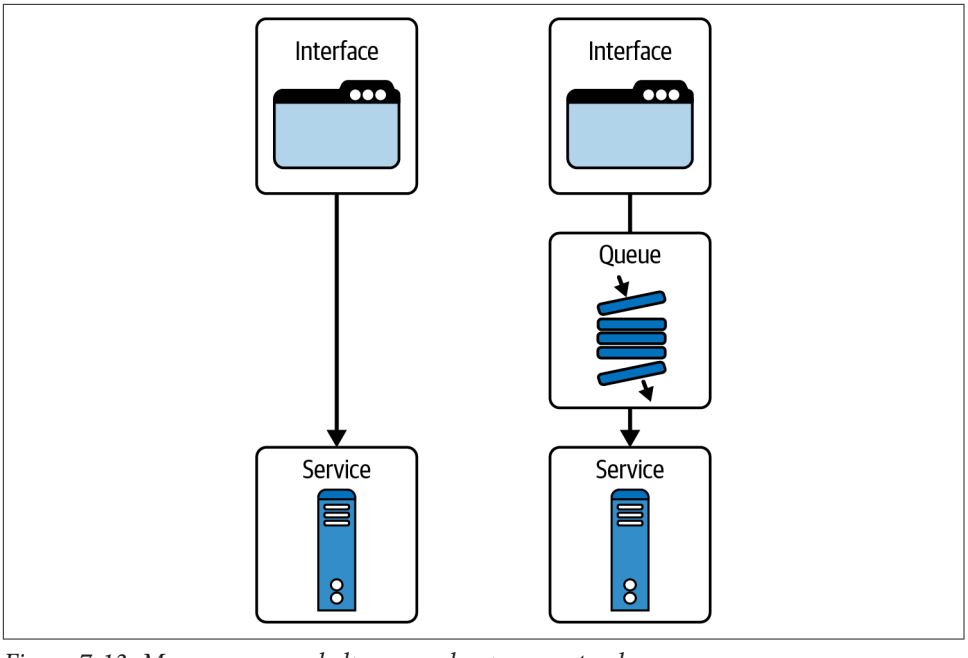

Figure 7-13. Message queues help you scale up request volume

If your messages are timing out in the queue (reaching `jobMaxTTL` or `taskMaxTTL`) before they are successfully processed, you may need to add more workflow processors to read the queues.

Use clusters to handle request processing

If your workflow processing is timing out, you likely need to add more processors—more workflow engines—to handle the added work. The easiest way to do this is to create a machine cluster (see Figure 7-14). Most platforms (including all the cloud platforms) have a feature that allows you to declare a single IP address as the destination for a service interface and then place several machines behind that destination. Any machine running in the cluster can accept workflow requests. That includes reading off the backlog queue.

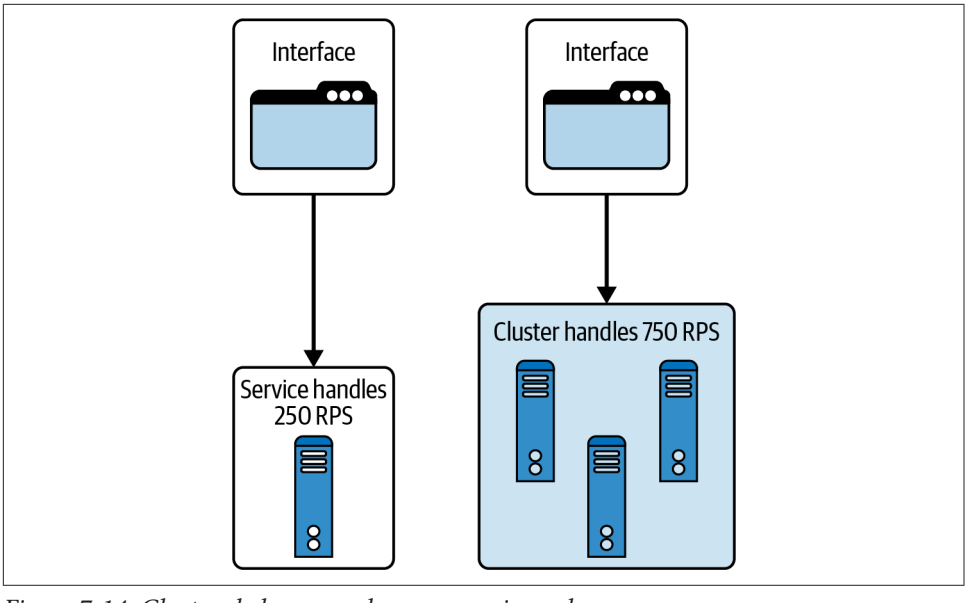

Figure 7-14. Clusters help you scale up processing volume

The details of setting up and managing machine clusters is beyond the scope of this book. Kubernetes is a great example of a product that handles cluster management. All the major cloud platforms (AWS, Google, Heroku, Microsoft, etc.) offer support for clustering, too.

Using machine clusters can also protect you against intermittent machine failures. If one machine in the cluster goes down, you may still have one or more machines up and running that can take the request load while the problem machine gets rebooted or replaced.

The good news is that both queues and clusters can be introduced without disrupting the public service interface.

Discussion

There are lots of solutions for implementing queues behind your interface. The simplest one is to create a disk folder somewhere (e.g., /requests/) and write all requests as a file in that folder. Then write code that monitors the folder and reads any files that appear there. The drawback of this option is that the folder is often tied to a specific machine, making it harder to implement shared queues for machine clusters.

All the major cloud vendors offer an auto-scaling feature. You can set parameters that control how large the traffic volume grows, automatically spin up new machines to handle the added requests, and then close out machines as the volume decreases. This can sometimes be fiddly to set up and possibly costly to operate, but it can get you through a short-term jam.

In both cases (queues and clusters), it is important that the messages are fully self-describing, the share states exist as independent resources, and the services can operate in parallel. These are all features of workflow-compliant services covered in Recipe 7.1.

See Also

- Recipe 3.7, "Enabling Interoperability with Inter-Service State Transfers"
- Recipe 4.11, "Validating Data Properties at Runtime"
- Recipe 6.1, "Hiding Your Data Storage Internals"
- Recipe 6.9, "Improving Performance with Caching Directives"
- Recipe 7.7, "Exposing a Progress Resource for Your Workflows"
- Recipe 7.15, "Synchronous Reply for Incomplete Work with 202 Accepted"

7.20 Using Workflow Proxies to Enlist Noncompliant Services

There are times when you want to use an existing noncompliant service interface as part of your workflow process. This recipe shows you how to do that safely and when to *not* do it at all.

Problem

What do you do when you have an existing service interface that is not a workflow-compliant implementation and you want to include that as a task in a workflow job? Under what cases can this be done safely, and when it is not advisable to try at all?

Solution

The simplest way to enlist a service interface that is not workflow-compliant into your workflow stream is to wrap it in a workflow-compliant *proxy*. This proxy can supply all the aspects of a workflow-compliant service (see Recipe 7.1), including all the actions, support for shared state, and passing correlation and request identifiers.

Your workflow proxy needs to support all the important actions listed in Recipe 7.1, including:

- Execute
- Repeat
- Revert

Typically, you can use the Execute action to point to the target service and perform the desired work. You will likely need to code routines that read and write content in the associated shared state resource, too.

However, support for the Repeat idempotent action and the Revert undo action can be more difficult. For read-only services like validating a postal code against a list, or compute-only services like calculating sales tax, there is no meaningful Revert to support and you can easily Repeat the actions as many times as you wish.

Note that if the target service changes state on the server side (e.g., modifies stored data, modifies a resource, etc.), you'll need to make sure to support both Repeat and Revert. If the existing service interface does this already, you can just wire up the proxy actions to the target service. If not, you'll need to see if you can code the proxy in a way that gives you the support you need.

For example, you might be able to support the Revert action for creating a resource by coding into the proxy the local knowledge of the resources the proxy has created, and upon request, executing the Revert action to delete that resource. However, you need to ensure that the resource you are attempting to delete has not created some other dependent state change (modified existing resources like a work ticket, log entry, etc.). This can be close to impossible if you or your team do not "own" the target service interface.

Here Be Dragons!

For reasons stated here, you should only proxy services that are read-only. The exception to this rule would be cases where you are in control of the noncompliant service and are confident you can safely support Revert and Repeat actions on that service.

In general, you should steer clear of relying on workflow proxies if at all possible. When you can't avoid it, try to stick to proxying read-only services unless you are absolutely sure the unsafe actions can be properly repeated and reverted.

Example

Assume you have a noncompliant service called `computeVAT` that you want to enlist in your shopping workflow. To do this, you need to create a new workflow-compliant proxy interface (let's call it `workflowVAT`) that supports following:

`readSharedState`
: This loads shared state into local memory, ready to use to supply any required data for execution service actions.

`inputVATData`
: This exposes a form for calling the execution operation and, if possible, is filled in by properties in the `sharedState` collection.

`executeVATComputation`
: This calls the target service's key functionality (e.g., `computeVAT`) using the previous form, filled in with data from the `sharedState` collection.

`writeSharedState`
: This writes the results of the execution to the `sharedState` collection.

`repeatVATComputation`
: This repeats the process of taking data from the `sharedState` collection, filling in the `inputVATData` form, and running the `executeVATComputation` action. This also results in a call to the `writeSharedState` action to update the `sharedState` collection.

`RevertVATComputation`
: This action removes any results (from the `executeVATComputation` action) from the `sharedState` collection (essentially reverting it back to its state *before* any execution took place) and performs a `writeSharedState` to commit the Revert action to the collection.

All through this process, the HTTP interface for the proxy actions supports both the `correlation-id` (job identifier) and `request-id` (task identifier) properties that are passed back and forth for any of the HTTP request or response messages.

You may notice that the proxy is doing quite a bit of heavy lifting to make it possible to use the `computeVAT` service. So much heavy lifting, that it might make sense to simply replace the `computeVAT` service with one that *is* workflow compliant. And you would be correct.

In other words, only use the workflow proxy recipe when you are unable to replace the noncompliant service with one that is workflow compliant.

Discussion

Introducing workflow proxies can reduce the reliability of your system. You are adding another service in the chain, and that exposes the possibility of network disconnects as well as service failures at either the proxy or target location. Be sure to include fixes for these things in your implementation (see Recipe 5.16).

It can be especially dangerous to attempt to proxy an unsafe noncompliant service (one that writes data) if you do not control the target service. For example, you don't know just what is happening behind the interface you are shown. Are there other services running? Is there a chain of dependent writes going on that you do not see? Even if you are confident of the current actions of the target service, what happens when that target is updated or replaced by its owner?

If you own the noncompliant service, you might need to use the workflow proxy pattern as an interim solution, one that will help you build up your workflow service practice while you spend time replacing noncompliant services with ones that meet all the requirements laid out in Recipe 7.1. This is a common digital transformation pattern that I see many organizations go through. A kind of "fake it until you make it" approach. See "Transforming Existing Services" on page 410 for more on this point.

See Also

- Recipe 3.4, "Describing Problem Spaces with Semantic Profiles"
- Recipe 3.6, "Designing Consistent Data Writes with Idempotent Actions"
- Recipe 4.4, "Coding Effective Clients to Understand Vocabulary Profiles"
- Recipe 5.13, "Improving Service Discoverability with a Runtime Service Registry"
- Recipe 5.17, "Using Semantic Proxies to Access Noncompliant Services"
- Recipe 6.13, "Using Pass-Through Proxies for Data Exchange"
- Recipe 7.1, "Designing Workflow-Compliant Services"
- Recipe 7.15, "Synchronous Reply for Incomplete Work with 202 Accepted"
- Recipe 7.17, "Supporting Local Undo or Rollback"

CHAPTER 8
Closing Remarks

Every day is a journey, and the journey itself is home.

—Matsuo Basho

Now we come to the final stop on this journey—the closing. We've covered quite a bit of ground in this book and it's worth taking a few moments for some final review and reflection.

We started this journey with an introduction (Chapter 1) to the basic ideas behind this book, including exploring the meaning of RESTful web APIs and how hypermedia fits into the picture. We also identified some "Shared Principles for Scalable Services on the Web" on page 16 to help guide us in selecting and defining the recipes for this book.

I also included some background (Chapter 2) on some of the key technologies and design concepts that drive the web and provide inspiration for the recipes listed here. Hopefully, this historical detour provided some valuable context for the kinds of design and implementations that animate the recipes here.

Of course, the bulk of the book is in Part II, where all the recipes are found.

Finally, I've included some pointers to additional material outside the pages of this book that might be helpful as well as a few parting thoughts as you take the content of this book to the next step.

Applying These Recipes

The recipes in this book cover a wide field, including design (Chapter 3), clients (Chapter 4), services (Chapter 5), data (Chapter 6), and workflow (Chapter 7). While they are all laid out in a similar way, some recipes are definitely more challenging than others. Even though there is a meaning to the order in which I present the

chapters, that order might not be the way you and your team need to tackle the work of applying these recipes to the service interfaces you publish and/or consume within your organization.

Also, some of these recipes are mostly advice (e.g., Recipe 3.8) and some are detailed implementation guides (e.g., Recipe 7.6). Some rely heavily on cooperation between both API consumers and producers (e.g., Recipe 7.13). And some can be implemented without the need for any direct client-server interactions at design or build time (e.g., Recipe 5.2). Sorting out which recipes can be applied independently and which require close cooperation between parties will help you determine which recipes offer the biggest returns on your initial investment of time and resources.

It is worth pointing out that one of the advantages of using these recipes is they give us common names to use for shared things. The recipes are a language that we can use to talk about our designs. In my experience, even before I introduce these recipes to a client, I see examples of them in some of their service interfaces. Usually, while the teams know about a common way to handle paging through lists, for example, they haven't really shared that as a reuseable recipe and don't have a common name for it throughout the company. Sometimes just using the *names* of these recipes can bring additional coherence to your API implementations.

Design First

In my experience, you can use the design recipes as a starting place for any cases where you need to build new service interfaces from scratch. This will work whether you are using APIs to access existing services or are creating both the interface and the service code at the same time. The advantage here is that you get to use the design recipes to set new expectations for both API producers and consumers without the struggle of making new interface rules work for well-established, noncompliant existing services.

But even when you are starting out fresh with your services, it is important to resist the temptation to "make the whole world new again." Take small steps to improve the design of your APIs. For example, setting a standard that leads new APIs to be limited to a fixed set of representation formats (see Recipe 3.1) is a good early step. Supporting things like repeatability, reversibility, extensibility, and modifiability are more challenging since they require both the producer and consumer to understand the implications of the recipe at runtime.

Clients and Servers Unite!

The chapters covering client recipes (Chapter 4) and services (Chapter 5) were written as standalone chapters. However, the reader is likely to find some level of repetition and interconnection between them. It is not much fun to write service interfaces when there are no client applications to consume them. For this reason, even if your

current role in the team is to focus on just one of these two ends of the interaction, it would be a good idea to read through both chapters to get the full picture.

One of the key messages in the client recipes is to assume a level of independence from any particular service. Instead of designing and building "captive" clients that are expected to work with just one service interface, I've tried to include recipes that help you build client applications that can more easily traverse multiple services in order to solve a single problem (see Recipes 4.15 and 4.16). Even if you still end up creating service-specific client applications, keeping the spirit of independence in your frontend apps can improve the stability and flexibility of your codebase.

Similarly, Chapter 5 contains lots of recipes that emphasize publishing meta-information about the service interface itself. Client preferences (Recipe 5.5), service definition files (Recipe 5.9), API metadata documents (Recipe 5.10), and vocabulary listings (Recipe 5.7) are all examples of the importance of sharing metadata about the service interface. Even the recipes on errors (Recipe 5.12), health (Recipe 5.11), and content negotiation (Recipe 5.6) illustrate how additional information about the service requests themselves can be helpful in creating robust service interfaces.

Starting with Data Is Challenging

I know many organizations base much of their architecture on existing data models. Sometimes these data models have been refined over decades of use and modification, and are an incredibly powerful way to capture key business rules and practices. For these companies, applying the data recipes in Chapter 6 can present some challenges. The approach to data in this book is one that assumes your services have their own nonshared data storage and that service interaction is enabled through passing properties at runtime (e.g., Recipe 7.2), instead of multiple services accessing the same database tables. In these cases, the data recipes that can be the most valuable are those that cover how to represent query requests (Recipe 6.4) and responses (Recipe 6.5), as well as the recipes that focus on hiding data models (Recipe 6.1) and caching (Recipe 6.9) response results.

However, as I already mentioned, if you are designing and implementing new data storage services, Chapter 6 has a wealth of recipes that can inform your choices of storage technologies, search libraries, and integrity management.

Going with the Flow

Finally, the chapter on programming workflows (Chapter 7) brings many of the previous recipes together to solve problems using multiple services in a collection. This chapter takes a different approach than most of the books I've seen on the subject, too. Instead of focusing on the notions of orchestration versus choreography, the recipes here lean heavily on the hypermedia features already covered in previous

chapters in order to build a more declarative, shared, distributed way to enlist services into a workflow.

Because I'm sharing a different approach, implementing these recipes using typical workflow platforms as tools can be a challenge. However, it has been my experience that as you learn to "think in hypermedia" (see Chapter 2), these recipes can become less opaque and easier to translate into working code.

It is worth pointing out that the workflow recipes are grouped together to make them a bit easier to digest. From Recipes 7.1 through 7.5, you'll find a general discussion of the foundation of a robust workflow system. The collection from Recipes 7.8 to 7.13 provides a set of straightforward recipes for solving common interactions like list navigation, form submission, work-in-progress interactions, and more. There are a few other recipes focused on particular challenges in defining and managing workflow solutions, but one of the most important is Recipe 7.6, which maps out the basics of an extensible foundation for a general "RESTful job control" language to handle just about any workflow challenge.

Like the other chapters, it is best to start small and stay simple for as long as you possibly can when implementing these workflow recipes. Feel free to skip over the ones that seem too complicated, and don't hesitate to modify the ones outlined here or even create ones of your own that better fit your organization's needs.

Transforming Existing Services

> *Whenever you do a transition, do the smallest thing that teaches you the most and do that over and over again.*
>
> *—Adrian Cockcroft,*
> *former Netflix Cloud CTO*

This collection of recipes, along with the initial guidance in Chapters 1 and 2, provide a solid basis for transforming existing services to be more in line with the kind of hypermedia approach advocated here. But just modifying existing service interfaces without any clear plan or payback can be not only frustrating, but also dangerous.

Like any transformation work, it only pays off if there are clear advantages to changing an already existing system. The best strategy is to make a number of independent, small changes over time instead of trying to package up a lot of important modifications in a single production release.

You can use the recipes in this book to identify a single problem (e.g., "How can we improve the performance of our data queries?"), and then work through one or more changes to your services and/or interface APIs that will result in measurable improvements. Focusing on making changes that result in measurable improvements has a kind of "flywheel" effect in the company. As each change brings better performance, a more stable runtime experience, and less resource-intensive updates, this adds

momentum to your work and makes it easier to get support throughout the organization for other improvement initiatives in the future. Good work leads to more good work.

This book includes a handful of recipes designed to help ease transformation work, too. These are labeled as "pass-through" recipes (Recipe 5.17, Recipe 6.13, and Recipe 7.20) and are designed to make it possible (in some cases) to *wrap* existing services in a thin proxy interface that more closely follows the other recipes in this book. This can make it possible to "modernize" some existing services, provide additional customization for external services you don't control, and help you bring some consistency to your API ecosystem; all without having to recode or replace existing services that are still cost-effective and functional. As mentioned in these pass-through recipes, there are downsides to using these proxy recipes, and you need to use caution when implementing them.

As your company applies more and more of these recipes to your existing and new APIs, you may also notice changes in nontechnical areas, including team dynamics and individual empowerment. Because the selected recipes were designed to support a common set of shared principles (see "Shared Principles for Scalable Services on the Web" on page 16), they can also have the effect of freeing teams from the limits of waterfall practices as well as unlocking creativity within your organization. By creating and publishing services that make it easier for teams to work at their own pace and safely explore creative solutions without worrying about breaking other parts of the ecosystem, your company can begin to reflect some of the same virtues embodied in the recipes in this book.

If you engage in a concerted effort to transform technical elements of your IT departments, be prepared to experience transformations in company culture, too.

Additional Resources

This book is a big project and, frankly, not everything that was planned for has ended up between these pages. Time, resources, and priorities can work against an author sometimes, and cuts need to be made. Also, there are related projects like samples, online documentation for recipes, videos, and other elements, that live outside of these pages. So I wanted to share a few pointers to additional related materials you can use to enrich your *RESTful Web API Patterns and Practices Cookbook* experience.

Here are some other resources you can look forward to now that the book is released:

WebAPICookbook.com
: This is the landing page for this book and all related materials. If you're looking for additional resources, pointers to new materials, or just want to check on the status of the cookbook project, check out the website first.

@WebAPICookbook
: This is the Twitter account associated with this book. Be sure to follow this account for any announcements about content updates, other resources, presentations, etc.

Videos
: I'll be releasing a set of videos that cover some of the same material in this book along with interviews with API designers, architects, and programmers. The initial home for these videos will be on my YouTube channel. You can also monitor the Twitter account for video releases.

More recipes
: I'll be posting additional recipes (ones that didn't make the cut in the book) online as time goes on. Check out the Twitter account for details. You can also visit the GitHub repo for this book.

ALPS catalog
: There is an extensive online catalog of the Application-Level Profile Semantics (ALPS) documents for many recipes in this book. You'll find additional ALPS documents there for related recipes and sample implementations of the recipes, too.

Additional content
: Over time there may be other material available, including presentations, podcasts, etc. that crop up. I'll do my best to maintain an up-to-date reference list of these resources, too.

Finally, if you post anything online or see references to the recipe collection anywhere, please ping me via the Twitter account and I'll be sure to add the pointer to my collection.

Next Steps

So, now that you've made it to the end of this part of the journey, what's next? Hopefully, you have some ideas on how to "think in hypermedia" and how to apply these recipes to your own API efforts. You may even have additional recipes in mind that can solve important problems for your community. Now is a great time to start applying the ideas here to your own work. This is the best way to figure out which of these recipes will work for you, and which recipes are missing and need to be designed and documented.

Keeping in mind the guiding principle of this book is another way to think about your next steps. How can you and your team "leverage the global reach of the web to solve problems you haven't thought of for people you haven't met"? Even if you are not using these recipes directly, following the principle of helping people you haven't

met solve new problems is an excellent way to reach out to your professional and personal communities and make things just a little bit better.

I hope this book has inspired you to think creatively about how hypermedia and the internet can be used to solve problems in a new and imaginative way. No one knows what the future holds but, to quote Alan Kay (one more time):

> The best way to predict the future is to invent it.

Best of luck in your future, however you make it.

APPENDIX A

Guiding Principles of RESTful Web APIs

The guiding principle driving this recipe collection is:

> Leverage global reach to solve problems you haven't thought of for people you have never met.

The supporting principles are illustrated in Figure A-1 and described further in the following list.

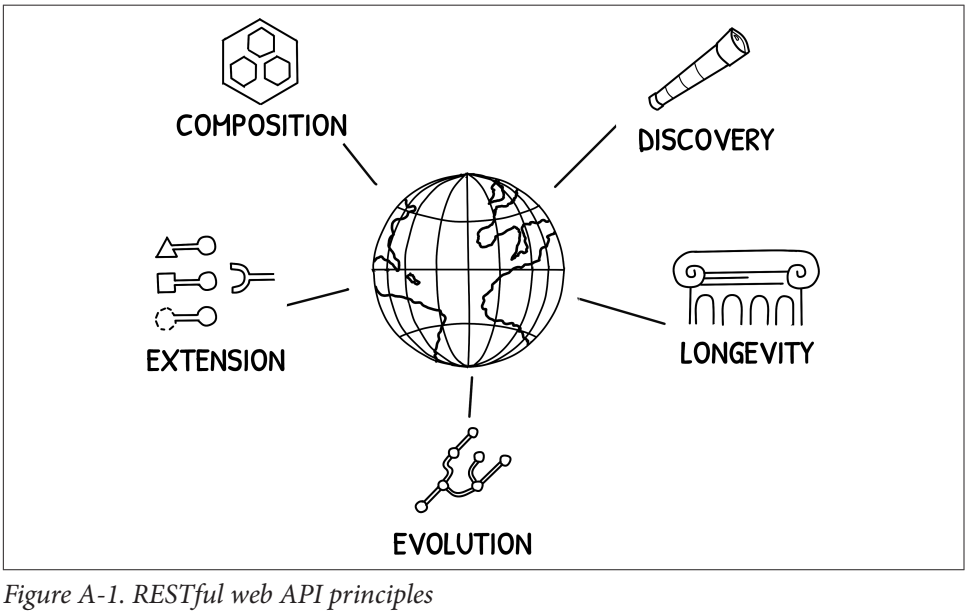

Figure A-1. RESTful web API principles

Discovery
: Good recipes increase our global reach—the ability to share our solutions and to find and use the solutions of others.

Extension
: Good recipes make well-designed services available for others to use in ways we haven't thought of yet.

Composition
: Good recipes make it possible for "strangers" (services and/or people) to safely and successfully interact with each other to solve a problem.

Evolution
: Good recipes promote longevity and independent evolution on a scale of decades.

Longevity
: Good recipes recognize that nothing is permanent and things will always change over time.

APPENDIX B
Additional Reading

Here is a list of reading materials that I keep handy on my (virtual) bookshelf. Most of the ones listed here were mentioned somewhere in the book:

- Alexander, Christopher. *A Pattern Language*. New York: Oxford University Press, 1977.
- Alexander, Christopher. *The Timeless Way of Building*. New York: Oxford University Press, 1979.
- Amundsen, Mike. *Building Hypermedia Clients with HTML5 and Node*. Sebastapol, CA: O'Reilly Media, Inc., 2011.
- Amundsen, Mike. *Design and Build Great Web APIs*. Raleigh, NC: Pragmatic Bookshelf, 2020.
- Amundsen, Mike. *RESTful Web Clients*. Sebastapol, CA: O'Reilly Media, Inc., 2017.
- Amundsen, Mike. *What Is Serverless?* Sebastapol, CA: O'Reilly Media, Inc., 2020.
- Amundsen, Mike, and Derric Gillig. *API Strategy for Decision-Makers*. Sebastapol, CA: O'Reilly Media, Inc., 2022.
- Bardini, Thierry. *Bootstrapping*. Stanford, CA: Stanford University Press, 2000.
- Bass, Len, et al. *Software Architecture in Practice, 4th Edition*. Boston: Addison-Wesley, 2021.
- Berners-Lee, Tim, and Mark Fishetti. *Weaving the Web*. New York: HarperCollins, 2008.
- Evans, Eric. *Domain-Driven Design*. Sebastapol, CA: O'Reilly Media, Inc., 2003.
- Gibson, James J. *The Ecological Approach to Visual Perception*. New York: Psychology Press, 1979.

- Gibson, James J. *The Senses Considered as Perceptual Systems*. Boston: Houghton Mifflin, 1966.
- Hall, Wendy, et al. *Rethinking Hypermedia*. New York: Springer US, 1996.
- Higginbotham, James. *Principles of Web API Design*. Boston: Addison-Wesley, 2021.
- Hohpe, Gregor, and Bobby Woolf. *Enterprise Integration Patterns*. Boston: Addison-Wesley, 2003.
- Lauret, Arnaud. *The Design of Web APIs*. Shelter Island, NY: Manning Publications, 2019.
- Meadows, Donella. *Thinking in Systems*. White River Junction, VT: Chelsea Green Publishing, 2008.
- Medjaoui, Mehdi, et al. *Continuous API Management*, 1st ed. Sebastapol, CA: O'Reilly Media, Inc., 2018.
- Medjaoui, Mehdi, et al. *Continuous API Management*, 2nd ed. Sebastapol, CA: O'Reilly Media, Inc., 2021.
- Nadareishvili, Irakli, et al. *Microservice Architecture*. Sebastapol, CA: O'Reilly Media, Inc., 2016.
- Nadareishvili, Irakli, and Ronnie Mitra. *Microservices: Up and Running*. Sebastapol, CA: O'Reilly Media, Inc., 2020.
- Nelson, Ted. *Computer Lib / Dream Machines*. Tempus Books, 1974.
- Newman, Sam. *Building Microservices*. Sebastapol, CA: O'Reilly Media, Inc., 2021.
- Norman, Donald. *The Design of Everyday Things*. New York, NY: Basic Books, 2013
- Richardson, Chris. *Microservices Patterns*. Shelter Island: Manning Publications, 2018.
- Richardson, Leonard, et al. *RESTful Web APIs*. Sebastapol, CA: O'Reilly Media, Inc., 2013.
- Rosenfeld, Louis, et al. *Information Architecture*, 4th edition. Sebastapol, CA: O'Reilly Media, Inc., 2015.
- Russell, Stuart, and Peter Norvig. *Artificial Intelligence*. Engelwood Cliffs, NJ: Prentice-Hall, Inc., 1995.
- Shiflett, Chris. *HTTP Developer's Handbook*. Indianapolis, IN: Sams Publishing, 2003.
- Taylor, Richard N., et al. *Software Architecture*. Hoboken, NJ: John Wiley & Sons, 2008.
- Wolff, Eberhard. *Microservices*. Boston: Addison-Wesley, 2016.

- Wright, Alex. *Cataloging the World*. New York: Oxford University Press, 2014.
- Yourdon, Edward, and Larry E. Constantine. *Structured Design*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1975.

APPENDIX C

Related Standards

Throughout the book I referred to standards documents. Most of them are public standards, but some are proprietary or private standards efforts. Here is a comprehensive list of standards I mentioned in these pages.

Viable Registered Media Types for RESTful Web APIs

This section contains a list (as of the release of this book) of media types that should be considered when you create your own RWAs. Most of them are already registered with the Internet Assigned Numbers Authority (IANA). Some are still going through the process of registration, and some new ones have been created.

By the time you read this book, some new media types may have been added, and some listed here may have fallen out of favor. It's a good idea to keep a look out for changes in the media type registry over time.

I've only listed formats I've had some experience with in the past. There are likely formats you know well that do not appear here or ones that appear here, you haven't seen before. Just focus on the ones that work best for you, and feel free to modify this list to fit your own experience and needs. See Recipe 3.2 for details on structured and unstructured types.

Structured Media Types

- HTML
- Collection+JSON
- UBER
- HAL
- SIREN
- HAL-FORMS
- Atom

Unstructured Media Types

- XML
- JSON
- CSV
- JSON-LD

Unregistered Media Types

- YAML
- JSON API

All hypermedia formats are not equal. For example, the Hypertext Application Language (HAL) format standardizes link elements but does not standardize forms or data bodies. The Structured Interface for Representing Entities (SIREN) format standardizes links and forms, but not data bodies. HTML, Collection+JSON, and Uniform Basis for Exchanging Representations (UBER) standardize all three message elements. I've created some "hypermedia polyfill" formats to supplement SIREN (Profile Object Description) and HAL (HAL-FORMS) to help bridge the gaps, but these additions can complicate implementations.

API Definition Formats

The following is a list of the common API definition formats (in alphabetical order) in use as of this writing. You can use these formats to define the exact details of your service's API (URIs, request methods, response codes, bodies, metadata, etc.). Some of these formats are designed to support classic HTTP request/response interactions, and others are for asynchronous messaging. It is likely that you'll use more than one of these formats in the implementation of your service APIs:

- AsyncAPI
- OpenAPI
- Protobuf
- RAML
- Schema Definition Language (tutorial)
- WSDL
- WADL

The versions listed here might be outdated by the time you are reading this book. Be sure to confirm the versions and release dates of the formats you choose to support.

Semantic Profile Document Formats

As of this writing, there are two SPDs in common use, along with one "legacy" format that I still see sometimes (XMDP):

- Dublin Core Application Profiles (DCAP)
- Application-Level Profile Semantics (ALPS)
- XHTML Meta Data Profiles (XMDP)

Hypermedia Supporting Types

These are formats designed to help fill in gaps in some hypermedia formats (POD and HAL-FORMS) as well as express other hypermedia-related metadata at design or build time (WESTL):

- Profiles for Object Description (POD)
- HAL-FORMS Media Type
- Web Service Transition Language (WESTL)

APPENDIX D

Using the HyperCLI

The `hyper` utility is a simple command-line style shell/REPL for interacting with online services/APIs. While a fully functional HTTP client, `hyper` is especially good at dealing with hypermedia services, including Collection+JSON, SIREN, and HAL. There are plans to add support for PRAG+JSON, MASH+JSON, and possibly UBER in the future. Other media type support can be added via plug-ins, too.

Along with HTTP and media type-aware commands, `hyper` also supports some convenience functionality like shell commands, configuration file management, and a LIFO stack to handle local memory variables.

You can access tutorials, examples, the source code, and the latest release of the HyperCLI tool via the HyperCLI landing page.

Hello, Hyper!

Here's a super simple HyperLANG script:

```
GOTO https://company-atk.herokuapp.com
```

And here's how you do it:

1. Make sure you have installed the latest update of the HyperCLI:
   ```
   $> npm install -g @mamund/hyper
   ```
2. Launch the HyperCLI REPL:
   ```
   $> hyper
   ```
3. Type the following command and press Enter:

```
> GOTO https://company-atk.herokuapp.com
```

It may take a few seconds for the sample service to fire up, but eventually you should see the following response:

```
STATUS 200
https://company-atk.herokuapp.com
application/forms+json; charset=utf-8
```

You can also view the request/response details with these commands:

```
SHOW REQUEST
SHOW RESPONSE
SHOW METADATA
SHOW ALL
SHOW URL
SHOW STATUS
SHOW CONTENT-TYPE
SHOW HEADERS
```

Finally, you can place all the HyperLANG commands in a text file and then pipe that file into the HyperCLI, like this:

```
$> hyper <hello-hyper.script
```

Congratulations! You've just created your first HyperLANG program.

Other Information

Some other details on the HyperCLI project:

Examples
: See the */scripts/* folder in the GitHub repo for lots of working examples.

Plug-in support
: You can author your own `hyper` plug-in and place it in the */plugins/* folder of the project. It will be automatically loaded at runtime. See the "Plug-In Authoring" section in the GitHub repo for details.

NPM package support
: You can download and track the latest `npm` package updates for HyperCLI, too.

Source code
: You'll find the source code for this utility in the */src/* folder of the GitHub repo.

HyperCLI Commands

Table D-1 lists the HyperCLI command keywords (as of the release of this book).

Be sure to check the latest documentation at the GitHub repo for the most recent changes.

Table D-1. HyperCLI command keywords and descriptions

#	Comment lines
VERSION	Returns version info
EXIT|STOP	Halt and exit with 0
EXIT-ERR	Halt and exit with 1
EXIT-IF	Halt and exit with 1 if simple condition is met
.. INVALID-URL <*url/#$*>	Returns TRUE if the string is NOT a valid URL
.. STACK-EMPTY	Returns TRUE if there is nothing on the internal stack
CLEAR	Clears the console
SHELL	Simple shell (bash/dos) support
.. LS|DIR *[folder/path]*	
PLUGINS	Returns list of loaded external plug-ins
CONFIG (READ)	Returns NVP of saved config data
.. FILE|LOAD *[filename]*	Loads config file (defaults to "hyper.config")
.. SAVE|WRITE *[filename]*	Loads config file (defaults to "hyper.config")
.. SET <*{ n:v,… }*>	Shared config file write
.. CLEAR	Removes all settings
.. RESET	Resets to default settings
.. REMOVE <*string*>	Removes the named item
STACK	JSON object LIFO stack
.. CLEAR|FLUSH	Clears all the items from the stack
.. PEEK	Displays the JSON object at the top of the stack
.. PUSH <*{ n:v,… }*>	Adds a new JSON object to the stack
.. PUSH WITH-RESPONSE	Adds a new item on the stack from the top of the *response* stack
.. PUSH WITH-PATH <*path-string/$#*>	Adds a new item on the stack that is the result of the JSONPATH
.. EXPAND-ARRAY	Expands the array on the top of the stack into n-items on the stack
.. POP	Removes the top item from the stack
.. LEN|LENGTH	Returns depth of the stack
.. SET <*{"n":"v",…}*>	Updates the JSON object on the top of the stack
.. LOAD|FILE *[filename]*	Reads a single JSON object from disk onto the stack (defaults to *hyper.stack*)

#	Comment lines
.. SAVE|WRITE *[filename]*	Writes the top item on the stack to disk (defaults to *hyper.stack*)
.. DUMP *[filename]*	Writes the full stack to disk (defaults to *hyper.dump*)
.. FILL *[filename]*	Replaces the current stack with contents in disk file (defaults to *hyper.dump*)
OAUTH OAuth 2.0 support	
.. LOAD *[filename]*	Loads OAuth config file (defaults to *oauth.env*)
.. SAVE *[filename]*	Saves OAuth config file (defaults to *oauth.env*)
.. DEFINE *<string> <{ n:v,… }>*	Creates an entry in the OAuth configuration
.. UPDATE *<string> <{ n:v,… }>*	Modifies settings in an existing OAuth configuration
.. GENERATE|GEN *<string>*	Generates an access token using the configuration data
.. REMOVE *<string>*	Removes the named item
ACTIVATE|CALL|GOTO|GO	Makes an HTTP request
.. WITH-URL *<url /$#>*	Uses URL to make the request
.. WITH-REL *<string /$#>*	Uses HREF value on the associated in-doc element
.. WITH-ID *<string /$#>*	Uses HREF value on the associated in-doc element
.. WITH-NAME *<string /$#>*	Uses HREF value on the associated in-doc element
.. WITH-PATH *<json-path-string /$#>*	Uses value from JSONPath result as the URL
.. WITH-ACCEPT *<string /$#>*	Sets the accept header directly
.. WITH-HEADERS *<{ n:v,… } /$#>*	Requests headers
.. WITH-OAUTH *<string /$#>*	Sets the `authorization` header using the named OAUTH token
.. WITH-BASIC *<string /$#>*	Sets the `authorization` header using the named BASIC config
.. WITH-QUERY *<{ n:v,… } /$#>*	Query string args as JSON nvps
.. WITH-BODY *<name=value&… /$#>*	For POST/PUT/PATCH (defaults to *app/form-urlencoded*)
.. WITH-METHOD *<string} /$#>*	To set HTTP method (defaults to GET)
.. WITH-ENCODING *<media-type /$#>*	To set custom encoding for POST/PUT/PATCH
.. WITH-FORMAT	Sets `accept` header with config value
.. WITH-PROFILE	Sets `link` profile header with config value
.. WITH-FORM *<name} /$#>*	Uses the metadata of the named form (URL, METHOD, ENCODING, FIELDS) to construct an HTTP request (SIREN-only)
.. WITH-STACK	Uses the top-level stack item as a set of vars for other operations (e.g., to fill in forms, supply query string values, headers, etc.)
.. WITH-DATA *<name=value&… /$#>*	Fills in forms with ad hoc data
DISPLAY|SHOW	Shows saved response (from top of the LIFO stack)
.. ALL	Returns the complete interaction (request, response metadata, response body)
.. REQUEST	Returns request info (URL, method, query string, body, and headers)
.. METADATA|META	Returns the response metadata (URL, status, and headers)
.. RESPONSE|PEEK	Returns response body from the top of the stack
.. URL|HREF	Returns actual URL of the response
.. STATUS|STATUS-CODE	Returns HTTP status of the response

#	Comment lines
.. CONTENT-TYPE	Returns HTTP content-type of the response
.. HEADERS	Returns the complete HTTP header collection of the response
.. POP	Removes response from top of the stack
.. CLEAR|FLUSH	Removes all responses from the stack
.. LEN|LENGTH	Returns length of saved stack
.. PATH *<JSONPath|XMLPath|$#>*	Returns results of a query (XML or JSON) from top-of-stack response
.. JPATH *<JSONPath|$#>*	Returns results of a JSONPath query from top-of-stack response
.. XPATH *<XMLPath|$#>*	Returns results of an XPath query from top-of-stack response
CJ	Returns a strong-typed version of response from top of the stack (`vnd.collection+json`)
.. METADATA	Returns metadata array from a Collection+JSON response
.. LINKS	Returns links array from a Collection+JSON response
.. ITEMS	Returns items array from a Collection+JSON response
.. QUERIES	Returns queries array from a Collection+JSON response
.. TEMPLATE	Returns template collection from a Collection+JSON response
.. ERROR|ERRORS	Returns `error` object from a Collection+JSON response
.. RELATED	Returns the `related` object from a Collection+JSON response
.. ID|NAME|REL|TAG *<string|$#>*	Returns a single node
.. IDS|RELS|NAMES|TAGS	Returns a simple list
.. PATH *<JSONPath|$#>*	Returns results of a JSONPath query from a Collection+JSON response
HAL	Returns a strong-typed version of response from top of the stack (`vnd.hal+json`)
.. LINKS	Returns links array from a HAL response
.. EMBEDDED	Returns items array from a HAL response
.. ID|REL|KEY|NAME|TAG *<string|$#>*	Returns a single node
.. IDS|RELS|KEYSTAGS	Returns a simple list
.. PATH *<JSONPath|$#>*	Returns results of a JSONPath query from a HAL response
SIREN	Returns a strong-typed version of response from top of the stack (`vnd.siren+json`)
.. LINKS	Returns links array from a SIREN response
.. ACTIONS|FORMS	Returns actions array from a SIREN response
.. ENTITIES	Returns entities array from a SIREN response
.. PROPERTIES	Returns properties array from a SIREN response
.. TAG|CLASS *<string|$#>*	Returns nodes associated with the CLASS value
.. ID|ENTITY *<string|$#>*	Returns an entity associated with the ID
.. REL|LINK *<string|$#>*	Returns a link associated with the REL
.. NAME|FORM|ACTION *<string|$#>*	Returns an action associated with the NAME
.. IDS|RELS|NAMES|FORMS|TAGS|CLASSES	Returns a simple list

#	Comment lines
.. PATH <*JSONPath|$#*>	Returns results of a JSONPath query from a SIREN response
WSTL	Returns a strong-typed version of response from top of the stack (`vnd.wstl+json`)
.. TITLE	Returns title string from a WSTL response
.. ACTIONS	Returns actions array from a WSTL response
.. DATA	Returns entities array from a WSTL response
.. RELATED	Returns `related` object from a WSTL response
.. CONTENT	Returns `content` object from a WSTL response
.. ID|REL|NAME|FORM|TAG|TARGET <*string|$#*>	Returns a single node
.. IDS|RELS|NAMES|FORMS|TAGS|TARGETS	Returns a simple list
.. PATH <*JSONPath|$#*>	Returns results of a JSONPath query from a WSTL response
FJ	Returns a strong-typed version of JSON+FORMS response from top of the stack (`forms+json`)
.. METADATA	Returns metadata array from a response
.. LINKS	Returns links array from a response
.. ITEMS	Returns items array from a response
.. ID <*string|$#*>	Returns an element (metadata, link, item) associated with the ID
.. TAG <*string|$#*>	Returns matching nodes
.. REL <*string|$#*>	Returns a link associated with the REL
.. NAME <*string|$#*>	Returns an element (metadata, link, property) associated with the NAME
.. IDS|NAMES|RELS|FORMS|TAGS	Returns a simple list
.. PATH <*JSONPath|$#*>	Returns results of a JSONPath query from a response
MASH	Returns a strong-typed version of response from top of the stack (`vnd.mash+json`)
.. METADATA	Returns metadata array from a response
.. LINKS	Returns links array from a response
.. ITEMS	Returns items array from a response
.. TAG <*string|$#*>	Returns matching nodes
.. ID <*string|$#*>	Returns an element (metadata, link, item) associated with the ID
.. REL <*string|$#*>	Returns a link associated with the REL
.. NAME <*string|$#*>	Returns an element (metadata, link, property) associated with the NAME
.. IDS|NAMES|RELS|FORMS|TAGS	Returns a simple list
.. PATH <*JSONPath|$#*>	Returns results of a JSONPath query from a SIREN response
PRAG	Returns a strong-typed version of response from top of the stack (`vnd.prag+json`)
.. METADATA	Returns metadata array from a PRAG response
.. LINKS	Returns links array from a PRAG response

#	Comment lines
.. ITEMS	Returns items array from a PRAG response
.. ID *<string|$#>*	Returns an element (metadata, link, item) associated with the ID
.. TAG *<string|$#>*	Returns matching nodes
.. REL *<string|$#>*	Returns a link associated with the REL
.. NAME *<string|$#>*	Returns an element (metadata, link, property) associated with the NAME
.. IDS|NAMES|RELS|FORMS|TAGS	Returns a simple list
.. PATH *<JSONPath|$#>*	Returns results of a JSONPath query from a SIREN response

Index

Symbols

*/http media types, 42
200 OK HTTP
  about, 270
  example, 271
201 CREATED, 77, 90
202 Accepted
  synchronous reply for incomplete work with, 381-386
300 Multiple Choices, 187
301 Moved Permanently, 165
404 Not Found
  about, 271
  example, 272
  temporarily missing data, 301
410 Gone, 256
4xx Bad Request
  about, 271
  example, 272
502 Bad Gateway, 313
5xx Server Error
  about, 271
  example, 273

A

a priori design, 25
accept-profile request, 117-119
action, data as evidence of, 41
actions, external, 171-176
additional reading, 417
Advanced Research Projects Agency (ARPA), 22
Advanced Research Projects Agency Network (ARPANET), 22
affordance (term), 13
AI (artificial intelligence), 157
alerts, 396-399
ALPS (see Application-Level Profile Semantics)
API client applications (see hypermedia clients)
API definition formats, 422
APIs.json, 203, 206-210
Application-Level Profile Semantics (ALPS)
  coding effective clients to understand vocabulary profiles, 113-116
  describing problem spaces with, 69-72
  supporting shared vocabularies in standard formats for service interfaces, 197-200
"Architectural Styles and the Design of Network-based Software Architectures" (Fielding), 5
ARPA (Advanced Research Projects Agency), 22
ARPANET (Advanced Research Projects Agency Network), 22
artificial intelligence (AI), 157
"As We May Think" (Bush), 11

B

Basho, Matsuo
  on journeys, 407
Berners-Lee, Tim
  goal of World Wide Web, 8
  HTTP and HTML, 13
  information architecture, 22
  RDF initiatives, 14
  Rule of Least Power, 246
  universal linked information system, 46
  universality of Web, 37

World Wide Web origins, 6-8, 12
binding targets, 27
binding, of client to responses, 27
Bossert, William, 14
bounded context, 15
breaking changes, 96
Bush, Vannevar, 11

C

caching directives
immutable caching, 287
improving performance with, 285-291
caching metadata
for API consumers, 287
consumer request with, 289
extended example, 289
provider response with, 288
for service providers, 286
capsulation, 42
choreography, 14, 32, 48
Clements, Paul
on good software architecture, 33
on software architecture, 163
client-driven workflow, 363
cloud services
service choreography, 49
service orchestration, 48
clusters, 400-403
Cockcroft, Adrian
on transitions, 410
Collection+JSON, 74, 151, 166, 175, 178, 257, 262, 340, 348
"Complex Information Processing" (Nelson), 9
composable (workflow-compliant) services, 317-321
content negotiation
converting internal models to external messages, 171-176
supporting HTTP content negotiation, 186-189
content-profile response header, 117-119
Content-Type header, 125
control language, 23
COP (see custom one-off proxy)
correlation-id header, 319
Cross-Origin Resource Sharing (CORS), 83
custom one-off proxy (COP)
about, 239
example, 240
when to use, 242

D

data exchange, pass-through proxies for, 307-314
data fields, ignoring unknown, 280-285
data languages
information retrieval engines and, 45
robust, 44-46
data model changes
adding a new property, 293
modifying in production, 291-297
reverting models, 296
single-access function, 294
support for multiple schema versions, 296
data model relationships, hiding from external actions, 256-260
data pass-through service, data source service versus, 313
data properties, validating, 139-141
data query resource, 277
data query support, using media types for, 274-279
data recipes (see distributed data)
data results resource, 277
data source service, data pass-through service versus, 313
data stores, extending, 298-302
data writes, 76-79
data, as evidence of action, 41
data-centric queries, returning HTTP 200 versus HTTP 400 for, 270-274
data-centric service interfaces (see distributed data)
database query language (DQL), 45
DCAP (Dublin Core Application Profile), 69
declarative documents, 331-333
dedicated operations, 81
Defined Exit Goal (DEG), 157-159
Defined State Goal (DSG), 159, 363
delayed response (DR), 381-386
delayed undo
about, 392
drawbacks of, 396
example, 394
dependency chain, 33
dependent services, runtime fallbacks for, 232-237
design (see hypermedia design)

Design of Everyday Things, The (Norman), 13
direct undo
  about, 392
  example, 393
  rights and, 395
discoverability, 220-223
distributed data, 245-314
  applying recipes, 409
  basics, 245-246
  data as evidence of action, 41
  extending remote data stores, 298-302
  hiding data relationships from external actions, 256-260
  hiding your data storage internals, 247-250
  hypermedia design and, 40-46
  ignoring unknown data fields, 280-285
  improving performance with caching directives, 285-291
  leveraging HTTP URLs to support contains…AND queries, 260-264
  limiting large-scale responses, 302-306
  making all changes idempotent, 251-256
  modifying data models in production, 291-297
  outside versus inside, 42
  reading versus writing, 43
  returning HTTP 200 versus HTTP 400 for data-centric queries, 270-274
  returning metadata for query responses, 264-269
  robust data languages, 44-46
  using media types for data queries, 274-279
  using pass-through proxies for data exchange, 307-314
document model, 51
document queries, validating incoming messages with, 147-149
documents, describing workflow as, 331-333
domain actions, expressing at runtime with embedded hypermedia, 73-76
domain specifics, sharing via published vocabularies, 64-67
Domain-Driven Design (Evans), 15
domain-specific code, message-centric code versus, 111-112
domain-specific language (DSL), 329-330
DQL (database query language), 45
DR (delayed response), 381-386
Dream Machines (Nelson), 12
DSG (Defined State Goal), 159, 363
DSL (domain-specific language), 329-330
Dublin Core Application Profile (DCAP), 69
Dynabook, 12

E

EBO (exponential backoff), 388, 391
ELP (see enterprise-level proxy)
embedded hypermedia messages, 73-76
empty collections, returning HTTP 200 versus HTTP 400, 270-274
Engelbart, Douglas, 12
enterprise-level proxy (ELP)
  about, 239
  example, 241
  when to use, 242
entity relationships, hiding from external actions, 256-260
error handling
  calling for help, 396-399
  workflow errors, 52
error reporting, 216-219
Evans, Eric, 15
evidence of action, data as, 41
evolvability, hypermedia controls and, 38
"Explaining Information Architecture" (video), 25
exponential backoff (EBO), 388, 391
extensible messages, 91-94
external messages, converting internal models to, 171-176
extreme late-binding, 8, 20, 25

F

failed HTTP requests, 387-391
fallbacks, for dependent services, 232-237
Fielding, Roy T.
  and architectural properties of key interest, 5
  and Hypermedia Constraint, 132
  and hypermedia design, 20
  and information architecture, 22
  and REST concept, 5-6
  and RESTful APIs, 190
  and temporal thinking, 37
  and timescales of REST software design, 18
find and bind, 39
forking the interface, 96
formats, binding to, 27

forms
hypermedia and, 10
nonhypermedia services, 135-138
FORMS
for transferring state between services, 80, 81
for updating existing records, 248
validating with JSON Schema, 144
functions, internal, expressing as external actions, 171-176

G

generality of interfaces, 20
Gibson, James J., 13
goal-oriented clients, 156-161
guiding principles of RESTful web APIs, 415

H

H-Factors (hypermedia factors), 132-135
HAL, 139, 262, 268
HAL-FORMS, 139
HCI (Human–Computer Interaction) movement, 13
Health Check Response Format for HTTP APIs, 211
information in OPTIONS responses, 214
information in service metadata documents, 214
health monitoring, of dependent services, 210-216
help, calling for, 396-399
hiding data storage internals, 247-250
Hippocratic Oath of APIs, 35-37, 95
HTTP (Hypertext Transfer Protocol)
advertising support for client response preferences, 181-186
coding clients to be HTTP aware, 107-110
messages and, 13
supporting HTTP content negotiation, 186-189
HTTP API, 260-264
HTTP DELETE, 88, 256, 395
HTTP metadata resource, 338
HTTP OPTIONS
and supported client preferences, 183-186
SDDs in HTTP OPTIONS responses, 204
HTTP PATCH, 251
HTTP POST
HTTP PUT versus, 76-79, 87
lost response problem and, 228-229
HTTP PUT
creating new resource with, 252
designing consistent data writes with idempotent actions, 76-79
handling a 410 Gone response, 256
handling a failed update, 254
HTTP POST versus, 87
improving reliability of data writes on the web, 251-256
lost response problem and, 229-231
rolling back changes with, 89
updating existing resource with, 253
HTTP requests
runtime resolution with metadata, 28
short-term fixes with automatic retries, 387-391
HTTP response, 270-274
hyper utility, 425
HyperCLI, 133, 154, 425-427
simple HyperLANG script, 425
hypermedia (generally)
a priori design, 25
affordances, 13
defined, 9
development/evolution, 10-12
impact on technology, xii
power of vocabularies, 14
reasons to use, 9-16
thinking/designing in, 19-52
value of messages, 13
hypermedia clients, 101-161
applying recipes, 408
basics, 101-102
coding clients to be HTTP aware, 107-110
coding effective clients to understand vocabulary profiles, 113-116
coding more resilient clients with message-centric implementations, 110-112
having a goal in mind, 156-161
including identifiers for important elements within a response, 128-131
increasing resilience with, 26-32
limiting the use of hardcoded URLs, 103-107
M2M challenges, 29
maintaining your own state, 153-156
managing representation formats at runtime, 119-123

negotiating for profile support at runtime, 116-119
relying on hypermedia controls in response, 132-135
runtime resolution with metadata, 28
semantic vocabularies, 30
supporting client-centric workflows, 31-32
supporting links and forms for nonhypermedia services, 135-138
using document queries to validate incoming messages, 147-149
using document schemas to validate outgoing messages, 141-147
using schema documents as a source of message metadata, 123-128
validating data properties at runtime, 139-141
validating incoming data, 150-153
Hypermedia Constraint, 132
hypermedia controls, evolvability and, 38
hypermedia design, 57-59
applying recipes, 408
basics, 57-98
creating interoperability with registered media types, 60-61
describing problem spaces with semantic profiles, 68-72
designing consistent data writes with idempotent actions, 76-79
designing for extensible messages, 91-94
designing for modifiable interfaces, 95-98
designing for repeatable actions, 84-87
designing for reversible actions, 87-91
empowering extensibility with hypermedia workflow, 46-52
enabling interoperability with inter-service state transfers, 79-83
enduring future compatibility with structured media types, 62-63
establishing a foundation, 21-25
expressing domain actions at runtime with embedded hypermedia, 73-76
ideas behind, 19-52
increasing resilience with hypermedia clients, 26-32
on scale of decades, 18
promoting stability/modifiability with hypermedia services, 33-40
sharing domain specifics via published vocabularies, 64-67
supporting distributed data, 40-46
hypermedia services, 163-243
advertising support for client response preferences, 181-186
applying recipes, 408
converting internal models to external messages, 171-176
hypermedia support for stability/evolvability, 37-39
improving reliability with idempotent create, 228-231
improving service discoverability with Runtime Service Registry, 220-223
increasing throughput with client-supplied identifiers, 224-227
modifiability problem, 34-37
preventing internal model leaks, 167-170
promoting stability/modifiability, 33-40
providing runtime fallbacks for dependent services, 232-237
publishing API metadata, 206-210
publishing at least one stable URL, 165-167
publishing complete vocabularies for machine clients, 190-196
publishing service definition documents, 202-205
from self-servicing to find and bind, 39
standardizing error reporting, 216-219
supporting HTTP content negotiation, 186-189
supporting service health monitoring, 210-216
supporting shared vocabularies in standard formats, 196-201
transforming existing services, 410
using semantic proxies to access noncompliant services, 238-243
workflow, 46
hypermedia signature, 132-135
hypermedia workflow, 46-52, 315-406
about, 49-51
applying recipes, 409
basics, 315-317
calling for help, 396-399
challenges for implementing, 51
choreography and orchestration, 47-51
constrained workflow, 52

describing workflow as code, 325-328
describing workflow as documents, 331-333
describing workflow as DSL, 329-330
designing workflow-compliant services, 317-321
empowering extensibility with, 46-52
enabling standard list navigation, 358-362
error handling, 52
exposing a progress resource for workflows, 338-342
jazz analogy, 50
observable workflow, 52
optimizing queries with stored replays, 375-380
returning all related actions, 342-346
returning most recently used resources, 346-350
scaling with queues and clusters, 400-403
sharing state, not data models, 51
short-term fixes with automatic retries, 387-391
supporting client-centric workflows, 31-32
supporting local undo or rollback, 391-396
supporting partial form submit, 363-366
supporting RESTful job control language, 334-338
supporting shared state for workflows, 322-325
supporting stateful work in progress, 350-357
synchronous reply for incomplete work with 202 Accepted, 381-386
time as workflow element, 52
using state-watch to enable client-driven workflow, 363
using workflow proxies to enlist noncompliant services, 400-403
hypertext, origin of term, 7, 10

I

IA (see information architecture)
IANA (Internet Assigned Numbers Authority) Media Types Registry, 38
IBO (incremental backoff), 388
idempotence
designing consistent data writes with idempotent actions, 76-79
designing for repeatable actions, 84-87
making all changes idempotent, 251-256
identifiers
for every important element within a response, 128-131
increasing throughput with client-supplied identifiers, 224-227
RDF and, 15
IETF (Internet Engineering Task Force), 23
immediate retry (IR), 388
immutable caching, 287
import/export operations, 80
incoming messages, validating, 147-149, 150-153
incomplete work, synchronous reply with 202 Accepted response status, 381-386
incremental backoff (IBO), 388
Information Architects (Wurman), 24
information architecture (IA)
origins, 22
Peter Morville and, 24
Information Architecture (Morville and Rosenfeld), 14, 24
"Information Management: A Proposal" (Berners-Lee), 6
information retrieval engines, 45
information retrieval query language (IRQL)
DQLs versus, 45
leveraging HTTP URLs to support contains…AND queries, 260-264
returning metadata for query responses, 264-269
inter-service state transfers, 79-83
interfaces, modifiable, 95-98
internal functions, expressing as external actions, 171-176
Internet Assigned Numbers Authority (IANA) Media Types Registry, 38
Internet Engineering Task Force (IETF), 23
interoperability
creating with registered media types, 60-61
enabling inter-service state transfers, 79-83
IR (immediate retry), 388
IRQL (see information retrieval query language)
IRQL-like data engines, 45
Isakowitz, Tomas, 10

J

jazz, as hypermedia workflow analogy, 50
job control language (JCL), 334-338

executing workflow with, 335
managing RESTful JCL records, 336
shared state, correlation IDs, and progress, 337
JSON, 123-128
JSON Path queries, 149
JSON Schema
validating FORMS with, 144
validation of sample document, 127
XML Schema versus, 146

K

Kay, Alan
and extreme late-binding, 8
and hypermedia design, 20
and messages in object-oriented programming, 13
and OOP, 8
Klyn, Dan
"Explaining Information Architecture" (video), 25
on elements of information architecture, 14

L

late binding (see extreme late-binding)
leaks
hiding your data storage internals, 247-250
preventing internal model leaks in services, 167-170
learning resources, 411, 417
Licklider Transmission Protocol (LTP), 23
Licklider, J.C.R. "Lick"
and ARPANET, 22-24
on communication, 57
links, for nonhypermedia services, 135-138
list navigation, 358-362
lost response problem
improving reliability of services with idempotent create, 228-231
network failures, 255
LTP (Licklider Transmission Protocol), 23

M

M2M (machine-to-machine) interactions
hiding from data relationships from external actions, 260
hypermedia client design challenges, 29
publishing complete vocabularies for machine clients, 190-196
response delays, 44
semantic vocabularies, 30
SMTs and, 62-63
using document queries to validate incoming messages, 147-149
Machine Accessible Semantic Hypermedia (MASH), 128
magic strings, 15, 66, 190
Manhattan Project, 11
master data management (MDM), 40
Meadows, Donella H.
on boundaries of systems, 19
media types, for data query support, 274-279
message format
as binding target, 27
managing representation formats for client applications at runtime, 119-123
stability and, 38
message-centric code, domain-specific code versus, 111-112
message/http media type, 42
messages
converting internal models to external messages, 171-176
designing for extensible messages, 91-94
expressing domain actions at runtime with embedded hypermedia messages, 73-76
HTTP and, 13
unknown properties with incoming messages, 127
using document queries to validate incoming messages, 147-149
validating incoming messages, 150-153
validating outgoing messages with schema documents, 141-147
value of, 13
meta-preferences, 183-185
metadata
publishing API metadata about service interface, 206-210
returning for query responses, 264-269
runtime resolution with, 28
schema documents as source of message metadata for client applications, 123-128
synchronous reply for incomplete work with 202 Accepted response status, 381-386

metamessages, 23, 25
Meyer, Paul J.
  on productivity, 315
model leaks, preventing, 167-170
modifiability
  Hippocratic Oath of APIs, 35-37
  hypermedia services, 34-37
modifiability, as architectural property, 6
Morville, Peter, 14, 24
most recently used (MRU) resources, 346-350
"Mother of All Demos" (Engelbart), 12
Must Ignore rule, 281, 284

N

Nadareishvili, Irakli
  on breaking the data-centric habit, 245
  on treating all data as remote, 40
name-value pair (NVP) collection, 92
Nelson, Ted, 12, 26
  and hypermedia design, 20
  and hypertext, 7, 9
  on telling computers what to do, 101
network failure
  handling, 255
  short-term fixes with automatic retries, 387-391
network idempotence, 84
network-level language, 23
nodes, 10
noncompliant services, 400-403
nonhypermedia services, 135-138
Norman, Donald
  on affordance, 13
  on affordances in well-designed objects, 17
  on user actions, 363
NVP (name-value pair) collection, 92

O

object-oriented programming (OOP), 8
observability, 52
ontology, 14, 15
operation idempotence, 84
orchestration, 47
Otlet, Paul, 11
outgoing messages, validating, 141-147

P

PAGE (percepts, actions, goals, environment) model, 157
pagination, 358-362
parallel properties, 92
Partial Form Submit (PFS), 363-366
pass-through recipes, 411
  using pass-through proxies for data exchange, 307-314
  using semantic proxies to access noncompliant services, 238-243
pass-through service, 308, 313
PCN (see proactive content negotiation)
performance, as architectural property, 6
personal computing, 12
PFS (Partial Form Submit), 363-366
Poe, Curtis, 8
portability, as architectural property, 6
POST (see HTTP POST)
Postel's Law
  client security and, 151
  schema documents and, 124
proactive content negotiation (PCN)
  about, 186
  drawbacks, 189
  example, 187
problem spaces, describing with semantic profiles, 68-72
production, modifying data models in, 291-297
progress resource, 338-342
protocol-aware clients, 107-110
protocols, binding to, 27
proxies
  accessing noncompliant services with, 238-243
  using workflow proxies to enlist noncompliant services, 400-403
PUT (see HTTP PUT)

Q

queries
  database versus information retrieval, 45
  optimizing with stored replays, 375-380
  using document queries to validate incoming messages, 147-149
query metadata, 264-269
query resources
  managing stored resources, 379

optimizing queries with stored replays, 375-380
query responses
implementing direct limits, 304
implementing truncated limits, 305
limiting large-scale responses, 302-306
query limits versus page sizes, 306
queues, scaling workflow with, 400-403

R

randomized retry (RR), 388
RCN (see reactive content negotiation)
RDF (see Resource Description Framework)
reactive content negotiation (RCN)
about, 187
drawbacks, 189
example, 188
reading data, rules for, 43
registered media type (RMT), 60-61, 421-422
regular interval retries (RIR), 388
related IANA link, 343-346
related standards documents, 421-423
reliability
as architectural property, 6
improving, 228-231
remote data store, 298-302
repeatable actions, designing for, 84-87
message repeatability, 85
network repeatability, 85
representation formats, managing at runtime, 119-123
Representational State Transfer (REST), 5-6
request metadata, runtime resolution with, 28
request-id header, 319
resilience
binding to protocols/formats, 27
coding more resilient clients with message-centric implementations, 110-112
increasing resilience with hypermedia clients, 26-32
M2M challenges, 29
runtime resolution with metadata, 28
Resource Description Framework (RDF), 14
identifiers and, 15
supporting shared vocabularies in standard formats for service interfaces, 197-201
resources, hypermedia and, 10
response preferences, advertising support for, 181-186
REST (Representational State Transfer), 5-6
RESTful web API (RWA) (generally)
basics, 4-18
change and, 18
guiding principles, 415
leveraging global reach, 16
Licklider Transmission Protocol, 23
shared principles for scalable services on the Web, 16-18
solving novel problems, 17
RESTful Web APIs (Richardson, Amundsen, and Ruby), 15
retries
options for handling transient network problems, 388
short-term fixes with automatic retries, 387-391
reversible actions, designing for, 87-91
rich input description (RID), 139-141
Richardson, Leonard, 4, 15
RID (rich input description), 139-141
RIR (regular interval retries), 388
RMT (registered media type), 60-61, 421-422
rollbacks
designing for reversible actions, 87-91
supporting local undo or rollback, 391-396
Rosenfeld, Louis, 14
RR (randomized retry), 388
RSR (Runtime Service Registry), improving service discoverability with, 220-223
Rule of Least Power, 7, 246
runtime
managing representation formats for client applications at runtime, 119-123
negotiating for profile support at runtime, 116-119
providing runtime fallbacks for dependent services, 232-237
resolution with metadata, 28
validating data properties, 139-141
runtime errors, 216-219
Runtime Service Registry (RSR), improving service discoverability with, 220-223
RWA (see RESTful web API)

S

scalability, as architectural property, 6
schema documents

problems inherent in validating incoming messages with, 149
source of message metadata for client applications, 123-128
validating outgoing messages with, 141-147
schema negotiation, 125
SDD (Service Definition Document)
HTTP OPTIONS responses, 204
HTTP response header, 203
publishing, 202-205
service metadata responses, 203
security
including unknown properties with incoming messages, 127
validating incoming data, 150-153
semantic gap, 15
semantic profile document (SPD), 68-72, 423
semantic profile proxy (SPP)
about, 239
example, 241
when to use, 242
semantic profile support, 116-119
semantic proxies, 238-243
semantic validation, 150
semantic vocabularies, 30
semantics, shared vocabularies and, 196-201
Senses Considered as Perceptual Systems, The (Gibson), 13
Service Definition Document (see SDD)
service interfaces
designing for modifiable interfaces, 95-98
three rules for modifying, 96
using media types for data queries, 274-279
service orchestration (see orchestration)
service proxies, 238-243
services (see hypermedia services)
services, transforming, 410
shared principles for scalable services on the Web, 16-18
shared references, 80, 82
shared state, 322-325
shared state resource, 322-325
simplicity, as architectural property, 6
single source of truth (SSOT), 40
SIREN, 36, 133, 140, 184
SMT (structured media type), 62-63
SOR (systems of record), 40
source code, describing workflow as, 325-328
source service, pass-through service versus, 313
space race (1960s), 23
SPD (semantic profile document), 68-72, 423
SPP (see semantic profile proxy)
SQL-like data engines, 45
SSOT (single source of truth), 40
stability
hypermedia support for, 37-39
message formats and, 38
standards documents, 421-423
API definition formats, 422
hypermedia supporting types, 423
registered media types, 421-422
semantic profile documents (SPDs), 423
state transfers, 79-83
state transitions, 342-346
state, maintaining your own, 153-156
state-watch
defined, 367
enabling client-driven workflow with, 363
typical interaction, 370
stateless choreography (see choreography)
storage service, pass-through service versus, 308
stored queries, 375-380
streaming data engines, 45
structured media type (SMT), 62-63
supplemental reading materials, 417
syntactic validation, 150
systems of record (SOR), 40

T

taxonomy, 14, 15
time, as workflow element, 52
timescales of systems, 17, 33-40
ToDo domain application, 111-112
transformation of existing services, 410
transient state, 153-156

U

undo, supporting local, 391-396
Uniform Resource Locator (see URL)
universal resource identifier (URI), 10
Universally Unique Identifier (UUID), 224-225
unknown data fields, ignoring, 280-285
unstructured media types, 123-128
updates to hypermedia services, 34-37
URI (universal resource identifier), 10
URL (Uniform Resource Locator)
as binding targets, 27

limiting memorized client URLs to one with hypermedia, 105
loading named URL variables from configuration, 105
publishing at least one stable URL, 165-167
reducing chances of client failure when service URLs are changed, 103-107
using named URL variables, 104
US Department of Defense
ARPANET, 22
J.C.R. Licklider and, 22-24
UUID (Universally Unique Identifier), 224-225

V

validation
of FORMS with JSON Schema, 144
of outgoing messages with schema documents, 141-147
visibility, as architectural property, 6
vocabularies, 17
coding effective clients to understand vocabulary profiles, 113-116
publishing complete vocabularies for machine clients, 190-196
semantic vocabularies, 30
sharing domain specifics via published vocabularies, 64-67
supporting shared vocabularies in standard formats for service interfaces, 196-201

W

Wilson, E. O., 14
work-in-progress (WIP)
single-page WIP documents, 356
stateful, 350-357
workflow (see hypermedia workflow)
workflow-compliant proxies, 400-403
workflow-compliant services, 317-321
correlation IDs for tasks and jobs, 319
shared state, 318
workflow actions, 317
workflow-progress resource, 338-342
World Wide Network, 11
World Wide Web (WWW)
basics, 6-8
goal of, 8
writing data
designing consistent data writes with idempotent actions, 76-79
rules for, 43
Wurman, Richard Saul, 24
WWW (see World Wide Web)

X

XML messages, validating with XML Schema, 143
XML Path queries, 149
XML Schema
JSON Schema versus, 146
validating XML messages with, 143
XML schema documents, as source of message metadata for client applications, 123-128

About the Author

An internationally known author and speaker, **Mike Amundsen** consults with organizations around the world on network architecture, web development, and the intersection of technology and society. He works with companies large and small to help them capitalize on the opportunities APIs, microservices, and digital transformation present for both consumers and the enterprise.

Colophon

The animal on the cover of *RESTful Web API Patterns and Practices Cookbook* is a Grandidier's mongoose (*Galidictis grandidieri*), named after the French naturalist and explorer Alfred Grandidier who visited Madagascar several times in the 1860s. There are six known species of mongoose in Madagascar; Grandider's mongoose lives in only a very small, remote, subtropical area around Lac Tsimanampetsotsa in southwestern Madagascar that researchers struggle to access.

Grandidier's mongoose, also known as the giant-striped mongoose, is 32 to 40 cm long with a 28 to 30 cm tail and has eight wide, dark stripes on its light brown fur. To avoid the heat of the region, this species has evolved to be nocturnal. The full diet of these carnivores is unknown, but they do eat invertebrates. We also don't know the typical lifespan of Grandidier's mongoose, but they have been documented breeding year-round in monogamous pairs.

Grandidier's mongoose is endangered, and their habitat itself is endangered as well. Many of the animals on O'Reilly's covers are endangered; all of them are important to the world.

The cover illustration is by Karen Montgomery, based on an antique line engraving from *Histoire Naturelle*. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.

O'REILLY®

Learn from experts.
Become one yourself.

Books | Live online courses
Instant Answers | Virtual events
Videos | Interactive learning

Get started at oreilly.com.

©2022 O'Reilly Media, Inc. O'Reilly is a registered trademark of O'Reilly Media, Inc. | 175