

Diseño de aplicaciones web modernas con ASP.NET Core y Azure

Guía práctica para desarrollar, implementar y escalar aplicaciones web de nivel de producción en la nube



Arquitectura moderna en .NET



Implementación en Microsoft Azure



Contenedores y microservicios



Seguridad y buenas prácticas



Ejemplos y código de referencia



ASP.NET Core

```
12 var builder = WebApplication.CreateBuilder(args);
2   builder.Services.AddControllers();
4   builder.Services.AddEndpointsApiExplorer();
5   builder.Services.AddSwaggerGen();
6
19 var app = builder.Build();
17
18 app.MapControllers();
13 app.Run();
19
```

.NET 8

Steve “ardalis” Smith

Arquitecto de software y entrenador

Construyendo el futuro
con desarrolladores



.NET 8.0



Microsoft Azure



Código de referencia
en GitHub



Visual Studio

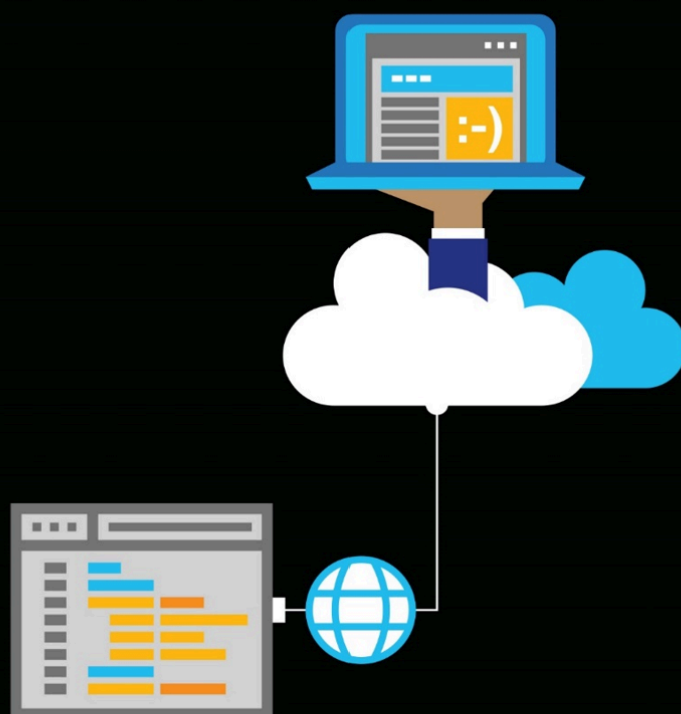
Diseño de aplicaciones web modernas con ASP.NET Core y Azure

Artículo • 22/01/2025

Antes de leer este artículo, se recomienda leer primero [patrón de aplicación web moderna para .NET](#). El artículo [Patrón de aplicaciones web modernas en .NET](#):

- Es la guía recomendada de Microsoft para los patrones de aplicaciones web modernas. Proporciona una arquitectura prescriptiva, código e instrucciones de configuración sobre cómo modernizar las aplicaciones web en la nube.
- Proporciona una [implementación de referencia de GitHub](#) [↗](#): una aplicación web de nivel de producción que usa procedimientos recomendados. Contiene ejemplos concretos para desarrollar aplicaciones web modernas en Azure.
- Está más actualizado que este artículo y se actualiza activamente.

Architecting Modern Web Applications with ASP.NET Core and Microsoft Azure



Steve “ardalis” Smith

EDITION v8.0: actualizado a ASP.NET Core 8.0

Consulte el [registro de cambios](#) para ver las modificaciones del libro y las colaboraciones para la comunidad.

PUBLICADO POR

Equipos de producto de Microsoft Developer Division, .NET y Visual Studio

Una división de Microsoft Corporation

One Microsoft Way

Redmond, Washington 98052-6399

Copyright © 2023 de Microsoft Corporation

Todos los derechos reservados. No se podrá reproducir ni transmitir ninguna parte del contenido de este libro en cualquier forma ni por ningún medio sin el permiso por escrito del editor.

Este libro se proporciona "tal cual" y expresa las opiniones del autor. Las vistas, opiniones e información expresadas en este libro, incluidas las direcciones URL y otras referencias a sitios web de Internet, pueden cambiar sin previo aviso.

Algunos ejemplos que se muestran aquí se proporcionan solo para la ilustración y son ficticios. No se pretende ni se debe deducir ninguna asociación o conexión real.

Microsoft y las marcas comerciales enumeradas en <https://www.microsoft.com> en la página web "Marcas comerciales" son marcas comerciales del grupo de empresas de Microsoft.

Mac y macOS son marcas comerciales de Apple Inc.

El logotipo de la ballena de Docker es una marca registrada de Docker, Inc. Usado por permiso.

Todas las demás marcas y logotipos son propiedad de sus respectivos propietarios.

Autor:

Steve "ardalis" Smith - Arquitecto de software y entrenador - [Ardalis.com](https://ardalis.com)

Editores:

Maira Wenzel

Vínculos de acción

- Este libro electrónico también está disponible en formato PDF (solo versión en inglés) [Descargar](#)
- Clona o haz un fork de la aplicación de referencia [eShopOnWeb en GitHub](#)

Introducción

.NET 8 y ASP.NET Core ofrecen varias ventajas sobre el desarrollo tradicional de .NET. Debe usar .NET 8 para las aplicaciones de servidor si algunas o todas las siguientes son importantes para el éxito de la aplicación:

- Compatibilidad multiplataforma.
- Uso de microservicios.
- Uso de contenedores de Docker.
- Requisitos de alto rendimiento y escalabilidad.
- Control de versiones en paralelo de versiones de .NET por aplicación en el mismo servidor.

Las aplicaciones tradicionales de .NET 4.x pueden y admiten muchos de estos requisitos, pero ASP.NET Core y .NET 8 se han optimizado para ofrecer compatibilidad mejorada con los escenarios anteriores.

Cada vez más organizaciones eligen hospedar sus aplicaciones web en la nube mediante servicios como Microsoft Azure. Debe considerar la posibilidad de hospedar la aplicación en la nube si lo siguiente es importante para la aplicación u organización:

- Reducción de la inversión en los costos del centro de datos (hardware, software, espacio, utilidades, administración de servidores, etc.)
- Precios flexibles (pago en función del uso, no para la capacidad inactiva).
- Confiabilidad extrema.
- Movilidad mejorada de la aplicación; Cambie fácilmente dónde y cómo se implementa la aplicación.
- Capacidad flexible; aumentar o disminuir según las necesidades reales.

La creación de aplicaciones web con ASP.NET Core, hospedada en Azure, ofrece muchas ventajas competitivas frente a las alternativas tradicionales. ASP.NET Core está optimizado para prácticas modernas de desarrollo de aplicaciones web y escenarios de hospedaje en la nube. En esta guía, aprenderá a diseñar sus aplicaciones ASP.NET Core para aprovechar mejor estas funcionalidades.

Versión

Esta guía se ha revisado para cubrir **versión de .NET 8.0** junto con muchas actualizaciones adicionales relacionadas con la misma "oleada" de tecnologías (es decir,

Azure y tecnologías de terceros adicionales) que coinciden a tiempo con la versión de .NET 8.0. Este es el motivo por el que la versión del libro se ha actualizado también a la versión 8.0.

Propósito

En esta guía se proporcionan instrucciones integrales sobre la creación de *aplicaciones web monolíticas* mediante ASP.NET Core y Azure. En este contexto, "monolítico" hace referencia al hecho de que estas aplicaciones se implementan como una sola unidad, no como una colección de servicios y aplicaciones interactivos. En algunos contextos, el término *monolito* se puede usar de manera peyorativa, pero en la gran mayoría de las situaciones, una sola aplicación es mucho más fácil de construir, implementar y depurar que una aplicación compuesta de muchos servicios diferentes, mientras sigue cumpliendo los requisitos empresariales.

Esta guía es complementaria a "[.NET Microservices. Arquitectura para aplicaciones .NET en contenedores](#)", que se centra más en Docker, microservicios y despliegue de contenedores para alojar aplicaciones empresariales.

Microservicios de .NET. Arquitectura para aplicaciones .NET en contenedor

- Libro electrónico
<https://aka.ms/MicroservicesEbook> ↗
- Aplicación de ejemplo
<https://aka.ms/microservicesarchitecture> ↗

Quién debe usar esta guía

El público de esta guía es principalmente desarrolladores, responsables de desarrollo y arquitectos interesados en crear aplicaciones web modernas mediante tecnologías y servicios de Microsoft en la nube.

Un público secundario es responsables de la toma de decisiones técnicas que ya están familiarizados ASP.NET o Azure y buscan información sobre si tiene sentido actualizar a ASP.NET Core para proyectos nuevos o existentes.

Cómo puede usar esta guía

Esta guía se ha condensado en un documento relativamente pequeño que se centra en la creación de aplicaciones web con tecnologías modernas de .NET y Azure. Por lo tanto, se puede leer en su totalidad para proporcionar una base de comprensión de estas aplicaciones y sus consideraciones técnicas. La guía, junto con su aplicación de ejemplo, también puede servir como punto de partida o referencia. Use la aplicación de ejemplo asociada como plantilla para sus propias aplicaciones o para ver cómo puede organizar los elementos de componente de la aplicación. Consulte los principios de la guía y la descripción de las opciones de arquitectura y tecnología, así como las consideraciones de decisión al evaluar estas opciones para su propia aplicación.

No dude en reenviar esta guía a su equipo para ayudar a garantizar una comprensión común de estas consideraciones y oportunidades. Tener a todos trabajando desde un conjunto común de terminología y principios subyacentes ayuda a garantizar una aplicación coherente de patrones y prácticas arquitectónicos.

Enviar sus comentarios

Este libro y ejemplos relacionados evolucionan constantemente, por lo que se agradecen sus comentarios. Si tiene comentarios sobre cómo se puede mejorar este libro y lo está leyendo en <learn.microsoft.com>, use la sección **Comentarios** que aparece al final de la página:

Feedback

Submit and view feedback for

This product

 This page

 [View all page feedback](#)

Como se ha resaltado en la captura de pantalla anterior, la sección de comentarios le permite enviar comentarios para:

- **Este producto:** mediante el formulario de comentarios del producto .NET.
- **Esta página:** mediante una plantilla de problemas de GitHub con los detalles de la página.

Si está leyendo este libro como un PDF, puede enviar comentarios creando un [problema de GitHub en Documentos de .NET](#) o mediante la [plantilla de problemas de GitHub en Libros electrónicos de arquitectura de .NET](#).

Referencias

- elegir entre .NET y .NET Framework para aplicaciones de servidor
<https://learn.microsoft.com/dotnet/standard/choosing-core-framework-server>

Siguiente

Características de las aplicaciones web modernas

18/06/2025

💡 Sugerencia

Este contenido es un extracto del libro electrónico, Arquitecto de aplicaciones web modernas con ASP.NET Core y Azure, disponible en [.NET Docs](#) o como un PDF descargable gratuito que se puede leer sin conexión.

Descargar PDF



"... con un diseño adecuado, las características resultan económicas. Este enfoque es difícil, pero sigue teniendo éxito".

- *Dennis Ritchie*

Las aplicaciones web modernas tienen mayores expectativas de usuario y mayores demandas que nunca. Se espera que las aplicaciones web de hoy estén disponibles 24/7 desde cualquier lugar del mundo y se puedan usar desde prácticamente cualquier dispositivo o tamaño de pantalla. Las aplicaciones web deben ser seguras, flexibles y escalables para satisfacer los picos de demanda. Cada vez más, los escenarios complejos deben controlarse mediante experiencias de usuario enriquecidas basadas en el cliente mediante JavaScript y la comunicación eficaz a través de las API web.

ASP.NET Core está optimizado para aplicaciones web modernas y escenarios de hospedaje basados en la nube. Su diseño modular permite a las aplicaciones depender solo de las características que realmente usan, lo que mejora la seguridad y el rendimiento de las aplicaciones, al tiempo que reduce los requisitos de recursos de hospedaje.

Aplicación de referencia: eShopOnWeb

Esta guía incluye una aplicación de referencia, *eShopOnWeb*, que muestra algunos de los principios y recomendaciones. La aplicación es una tienda en línea sencilla, que permite navegar por un catálogo de camisas, tazas de café y otros artículos de marketing. La aplicación de referencia es deliberadamente sencilla para facilitar la comprensión.

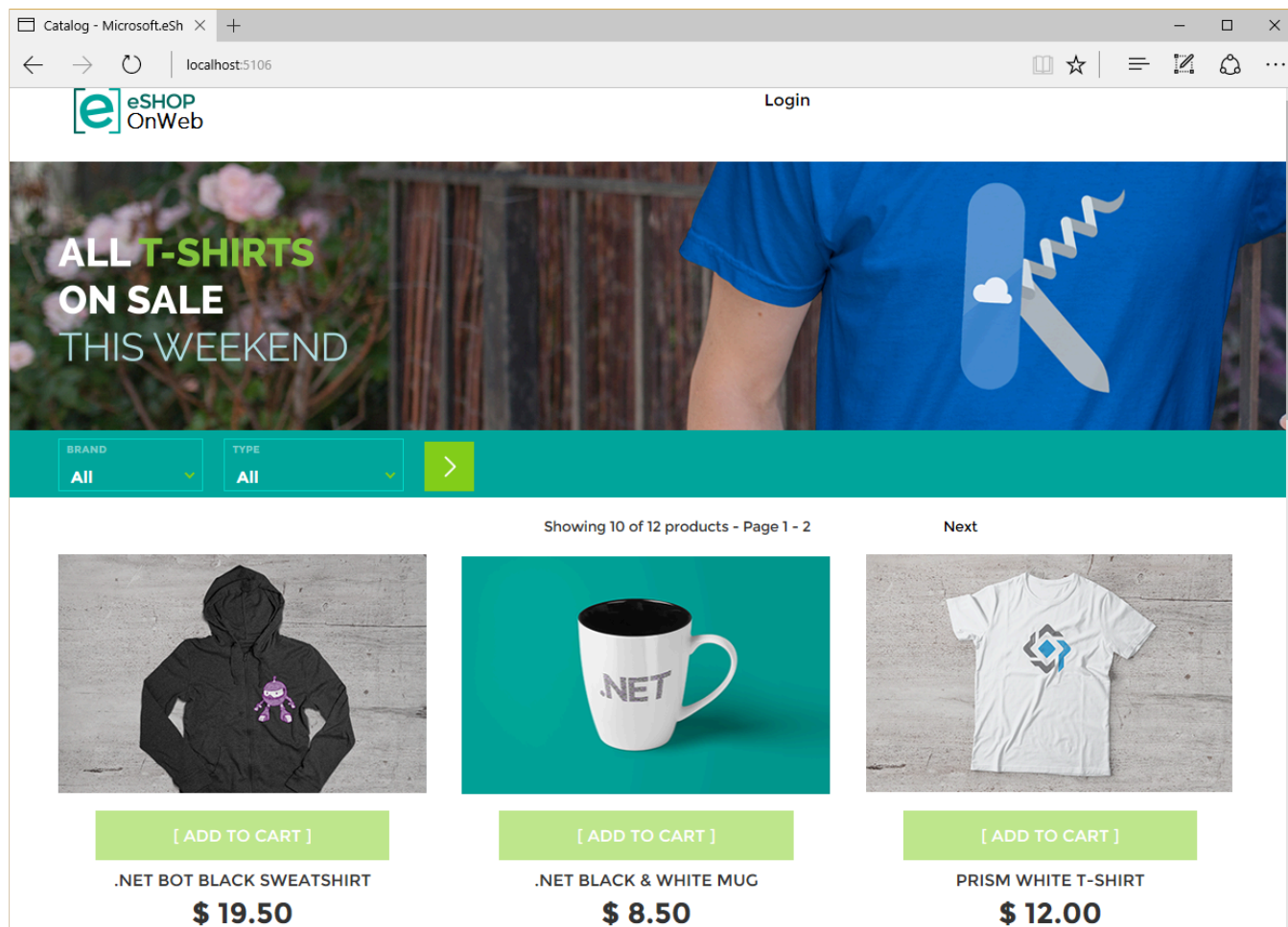


Figura 2-1. eShopOnWeb

Aplicación de referencia

- eShopOnWeb
<https://github.com/dotnet/eShopOnWeb>

Hospedado en la nube y escalable

ASP.NET Core está optimizado para la nube (nube pública, nube privada, cualquier nube) porque es de poca memoria y alto rendimiento. El menor tamaño de las aplicaciones de ASP.NET Core significa que puede hospedar más en el mismo hardware y pagar por menos recursos al usar servicios de hospedaje en la nube bajo el modelo de pago por uso. El mayor rendimiento significa que puede atender a más clientes de una aplicación dada el mismo hardware, lo que reduce aún más la necesidad de invertir en servidores y en la infraestructura de hospedaje.

Multiplataforma

ASP.NET Core es multiplataforma y se puede ejecutar en Linux, macOS y Windows. Esta funcionalidad abre muchas nuevas opciones para el desarrollo y la implementación de aplicaciones creadas con ASP.NET Core. Contenedores de Docker, tanto Linux como Windows, pueden hospedar aplicaciones ASP.NET Core, lo que les permite aprovechar las [ventajas de los contenedores y microservicios](#).

Modular y de acoplamiento flexible

Los paquetes NuGet son ciudadanos de primera clase en .NET Core y las aplicaciones de ASP.NET Core se componen de muchas bibliotecas a través de NuGet. Esta granularidad de la funcionalidad ayuda a garantizar que las aplicaciones solo dependan e implementen únicamente la funcionalidad que realmente requieren, reduciendo así su huella y la superficie de exposición a vulnerabilidades de seguridad.

ASP.NET Core también admite completamente la [inserción de dependencias](#) [↗](#), tanto internamente como en el nivel de aplicación. Las interfaces pueden tener varias implementaciones que se pueden intercambiar según sea necesario. La inserción de dependencias permite a las aplicaciones acoplarse flexiblemente a esas interfaces, en lugar de implementaciones específicas, lo que facilita la extensión, el mantenimiento y las pruebas.

Fácilmente comprobable con pruebas automatizadas

ASP.NET aplicaciones principales admiten pruebas unitarias y su acoplamiento flexible y compatibilidad con la inserción de dependencias facilita el intercambio de problemas de infraestructura con implementaciones falsas con fines de prueba. ASP.NET Core también se incluye con un TestServer que se puede usar para hospedar aplicaciones en memoria. Después, las pruebas funcionales pueden realizar solicitudes a este servidor en memoria, ejercer la pila de aplicaciones completa (incluido middleware, enrutamiento, enlace de modelos, filtros, etc.) y recibir una respuesta, todo en una fracción del tiempo que tardaría en hospedar la aplicación en un servidor real y realizar solicitudes a través de la capa de red. Estas pruebas son especialmente fáciles de escribir y valiosas para las API, que son cada vez más importantes en las aplicaciones web modernas.

Comportamientos tradicionales y de SPA admitidos

Las aplicaciones web tradicionales han implicado poco comportamiento del lado cliente, pero en su lugar se han basado en el servidor para toda la navegación, las consultas y las actualizaciones que la aplicación podría necesitar realizar. Cada nueva operación realizada por el usuario se traduciría en una nueva solicitud web, con el resultado de ser una recarga de página completa en el explorador del usuario final. Los marcos de controlador de vista de modelos (MVC) clásicos normalmente siguen este enfoque, en el que cada solicitud nueva se corresponde a otra acción de controlador, lo que a su vez podría funcionar con un modelo y devolver una vista. Algunas operaciones individuales en una página determinada podrían mejorarse con la funcionalidad AJAX (JavaScript asincrónico y XML), pero la arquitectura general de la aplicación usaba muchas vistas y puntos de conexión URL diferentes de MVC. Además, ASP.NET Core MVC también admite Razor Pages, una manera más sencilla de organizar páginas de estilo MVC.

Las aplicaciones de página única (SPA), por el contrario, implican muy pocas cargas de página del lado servidor generadas dinámicamente (si las hay). Muchas SPA se inicializan dentro de un archivo HTML estático que carga las bibliotecas de JavaScript necesarias para iniciar y ejecutar la aplicación. Estas aplicaciones hacen un uso intensivo de las API web para sus necesidades de datos y pueden proporcionar experiencias de usuario mucho más enriquecidas. BlazorWebAssembly proporciona un medio para compilar SPA mediante código .NET, que luego se ejecuta en el explorador del cliente.

Muchas aplicaciones web implican una combinación de comportamiento tradicional de aplicaciones web (normalmente para contenido) y SPA (para interactividad). ASP.NET Core admite MVC (vistas o basadas en páginas) y API web en la misma aplicación, con el mismo conjunto de herramientas y bibliotecas de marcos subyacentes.

Desarrollo e implementación sencillos

ASP.NET Core aplicaciones se pueden escribir usando editores de texto simples o mediante interfaces de línea de comandos, o entornos de desarrollo completos como Visual Studio. Normalmente, las aplicaciones monolíticas se implementan en un único punto de conexión. Las implementaciones se pueden automatizar fácilmente para que se produzcan como parte de una canalización de integración continua (CI) y entrega continua (CD). Además de las herramientas tradicionales de CI/CD, Microsoft Azure tiene compatibilidad integrada con repositorios de Git y puede implementar automáticamente las actualizaciones a medida que se realizan en una rama o etiqueta de Git especificada. Azure DevOps proporciona una canalización completa de construcción e implementación de CI/CD, y GitHub Actions ofrece otra opción para los proyectos alojados allí.

Formularios web y ASP.NET tradicionales

Además de ASP.NET Core, la ASP.NET 4.x tradicional sigue siendo una plataforma sólida y confiable para compilar aplicaciones web. ASP.NET admite modelos de desarrollo de MVC y API web, así como web Forms, que es muy adecuado para el desarrollo de aplicaciones basado en páginas enriquecidas y cuenta con un ecosistema enriquecido de componentes de terceros. Microsoft Azure tiene una gran compatibilidad con ASP.NET aplicaciones 4.x y muchos desarrolladores están familiarizados con esta plataforma.

Blazor

Blazor se incluye con ASP.NET Core 3.0 y versiones posteriores. Proporciona un nuevo mecanismo para crear aplicaciones cliente web interactivas enriquecidas con Razor, C# y ASP.NET Core. Ofrece otra solución que se debe tener en cuenta al desarrollar aplicaciones web modernas. Hay dos versiones de Blazor que se deben tener en cuenta: el lado servidor y el lado cliente.

El lado servidor Blazor se lanzó en 2019 con ASP.NET Core 3.0. Como su nombre implica, se ejecuta en el servidor y vuelve a representar los cambios en el documento cliente en el explorador a través de la red. El servidor Blazor proporciona una experiencia de cliente enriquecida sin necesidad de JavaScript en el lado del cliente y sin necesidad de cargar páginas por separado con cada interacción del cliente. Los cambios en la página cargada son solicitados y procesados por el servidor y, a continuación, se devuelven al cliente mediante SignalR.

Blazor del lado cliente se ha publicado en 2020 y elimina la necesidad de representar los cambios en el servidor. En su lugar, aprovecha WebAssembly para ejecutar código .NET dentro del cliente. El cliente todavía puede realizar llamadas API al servidor si es necesario para solicitar datos, pero todo el comportamiento del lado del cliente se ejecuta en el cliente a través de WebAssembly, que ya es compatible con todos los navegadores principales y es simplemente una biblioteca de JavaScript.

Referencias: aplicaciones web modernas

- **Introducción a ASP.NET Core**
<https://learn.microsoft.com/aspnet/core/>
- **Pruebas en ASP.NET Core**
<https://learn.microsoft.com/aspnet/core/testing/>
- **Blazor -Comenzar**
<https://blazor.net/docs/get-started.html> ↗

Elegir entre aplicaciones web tradicionales y aplicaciones de página única (SPA)

17/06/2025

💡 Sugerencia

Este contenido es un extracto del libro electrónico, Arquitecto de aplicaciones web modernas con ASP.NET Core y Azure, disponible en [.NET Docs](#) o como un PDF descargable gratuito que se puede leer sin conexión.

Descargar PDF



"Atwood's Law: cualquier aplicación que se pueda escribir en JavaScript, se escribirá finalmente en JavaScript".

- Jeff Atwood

Hay dos enfoques generales para compilar aplicaciones web hoy en día: aplicaciones web tradicionales que realizan la mayoría de la lógica de la aplicación en el servidor y aplicaciones de página única (SPA) que realizan la mayoría de la lógica de la interfaz de usuario en un explorador web, que se comunican con el servidor web principalmente mediante API web. Un enfoque híbrido también es posible, lo más sencillo es hospedar una o más subaplicaciones de tipo SPA enriquecidas dentro de una aplicación web tradicional más grande.

Use aplicaciones web tradicionales cuando:

- Los requisitos del lado cliente de la aplicación son sencillos o incluso de solo lectura.
- La aplicación debe funcionar en exploradores sin compatibilidad con JavaScript.
- La aplicación está orientada al público y se beneficia de la detección y las referencias del motor de búsqueda.

Use una SPA en los casos siguientes:

- La aplicación debe exponer una interfaz de usuario enriquecida con muchas características.
- Tu equipo está familiarizado con JavaScript, TypeScript o desarrollo de BlazorWebAssembly.
- La aplicación ya debe exponer una API para otros clientes (internos o públicos).

Además, los marcos de SPA requieren una mayor experiencia en arquitectura y seguridad. Experimentan una mayor renovación debido a actualizaciones frecuentes y marcos de cliente nuevos que las aplicaciones web tradicionales. La configuración de procesos automatizados de compilación e implementación y el uso de opciones de implementación como contenedores pueden ser más difíciles con las aplicaciones SPA que las aplicaciones web tradicionales.

Las mejoras en la experiencia del usuario que ha hecho posible el enfoque de SPA deben ser ponderadas con estas consideraciones.

Blazor

ASP.NET Core incluye un modelo para crear interfaces de usuario enriquecidas, interactivas y redactables denominadas Blazor. Blazor el lado servidor permite a los desarrolladores crear la interfaz de usuario con C# y Razor en el servidor y para que la interfaz de usuario se conecte interactivamente al explorador en tiempo real mediante una conexión de SignalR persistente. BlazorWebAssembly presenta otra opción para aplicaciones Blazor, lo que les permite ejecutarse en el explorador mediante WebAssembly. Dado que es código .NET real que se ejecuta en WebAssembly, puede reutilizar código y bibliotecas desde partes del lado servidor de la aplicación.

Blazor proporciona una nueva y tercera opción que se debe tener en cuenta al evaluar si se va a crear una aplicación web puramente representada por el servidor o una SPA. Puede crear comportamientos enriquecidos del lado cliente similares a SPA mediante Blazor, sin necesidad de un desarrollo significativo de JavaScript. Blazor Las aplicaciones pueden llamar a las API para solicitar datos o realizar operaciones del lado servidor. Pueden interoperar con JavaScript cuando sea necesario para aprovechar las ventajas de las bibliotecas y marcos de JavaScript.

Considere construir su aplicación web con Blazor cuando:

- La aplicación debe exponer una interfaz de usuario enriquecida
- El equipo está más cómodo con el desarrollo de .NET que el desarrollo de JavaScript o TypeScript

Si tiene una aplicación de formularios web existentes que está pensando en migrar a .NET Core o a .NET más reciente, puede que desee revisar el libro electrónico gratuito, [Blazor para que los](#)

[desarrolladores de Web Forms](#) vean si tiene sentido considerar la posibilidad de migrarlo a Blazor.

Para obtener más información sobre Blazor, vea [Introducción a Blazor](#).

Cuándo elegir aplicaciones web tradicionales

La siguiente sección es una explicación más detallada de las razones indicadas anteriormente para elegir aplicaciones web tradicionales.

La aplicación tiene requisitos simples del lado cliente, posiblemente de solo lectura

Muchas aplicaciones web se consumen principalmente de forma de solo lectura por la gran mayoría de sus usuarios. Las aplicaciones de solo lectura (o principalmente de solo lectura) tienden a ser mucho más sencillas que las que mantienen y manipulan una gran cantidad de elementos de estado. Por ejemplo, un motor de búsqueda puede constar de un único punto de entrada con un cuadro de texto y una segunda página para mostrar los resultados de la búsqueda. Los usuarios anónimos pueden realizar solicitudes fácilmente y hay poca necesidad de lógica del lado cliente. Del mismo modo, la aplicación orientada al público de un sistema de administración de contenido o blog normalmente consta principalmente de contenido con poco comportamiento del lado cliente. Estas aplicaciones se crean fácilmente como aplicaciones web tradicionales basadas en servidor, que realizan lógica en el servidor web y representan HTML para que se muestren en el explorador. El hecho de que cada página única del sitio tenga su propia dirección URL que los motores de búsqueda pueden marcar e indexar (de forma predeterminada, sin tener que agregar esta funcionalidad como una característica independiente de la aplicación) también es una ventaja clara en estos escenarios.

La aplicación debe funcionar en exploradores sin compatibilidad con JavaScript

Las aplicaciones web que necesitan funcionar en exploradores con compatibilidad limitada o sin JavaScript deben escribirse mediante flujos de trabajo de aplicaciones web tradicionales (o al menos poder revertir a este comportamiento). Las SPA requieren JavaScript del lado cliente para funcionar; si no está disponible, las SPA no son una buena opción.

El equipo no está familiarizado con las técnicas de desarrollo de JavaScript o TypeScript

Si el equipo no está familiarizado con JavaScript o TypeScript, pero está familiarizado con el desarrollo de aplicaciones web del lado servidor, es probable que puedan entregar una aplicación web tradicional más rápidamente que una SPA. A menos que el aprendizaje para programar SPA sea un objetivo o se requiera la experiencia del usuario que ofrece una SPA, las aplicaciones web tradicionales son una opción más productiva para los equipos que ya están familiarizados con su compilación.

Cuándo elegir SPA

La siguiente sección es una explicación más detallada de cuándo elegir un estilo de desarrollo de aplicaciones de página única para la aplicación web.

La aplicación debe exponer una interfaz de usuario enriquecida con muchas características

Los SPA pueden admitir una funcionalidad enriquecida del lado cliente que no requiere volver a cargar la página a medida que los usuarios realizan acciones o navegan entre áreas de la aplicación. Los SPA pueden cargarse más rápidamente, capturar datos en segundo plano y las acciones de usuario individuales son más sensibles, ya que las recargas de página completa son poco frecuentes. Los SPA pueden admitir actualizaciones incrementales, guardar formularios o documentos completados parcialmente sin que el usuario tenga que hacer clic en un botón para enviar un formulario. Los SPA pueden admitir comportamientos enriquecidos del lado cliente, como arrastrar y colocar, mucho más fácilmente que las aplicaciones tradicionales. Las SPA se pueden diseñar para ejecutarse en modo desconectado, realizando actualizaciones en un modelo del lado cliente que finalmente se sincronizan con el servidor una vez que se vuelve a establecer una conexión. Elija una aplicación de estilo SPA si los requisitos de la aplicación incluyen una funcionalidad enriquecida que va más allá de lo que ofrecen los formularios HTML típicos.

Con frecuencia, los SPA necesitan implementar características integradas en aplicaciones web tradicionales, como mostrar una dirección URL significativa en la barra de direcciones que refleje la operación actual (y permitir que los usuarios marquen o vinculen en profundidad esta dirección URL). Las SPA también deben permitir que los usuarios usen los botones atrás y adelante del explorador con resultados que no les sorprendan.

El equipo está familiarizado con el desarrollo de JavaScript o TypeScript

Para escribir SPA es necesario estar familiarizado con JavaScript o TypeScript y técnicas de programación y bibliotecas del lado cliente. El equipo debe ser competente para escribir JavaScript moderno mediante un marco spa como Angular.

Referencias: marcos SPA

- Angular: <https://angular.io> ↗
- React: <https://reactjs.org/> ↗
- Vue.js: <https://vuejs.org/> ↗

La aplicación ya debe exponer una API para otros clientes (internos o públicos)

Si ya admite una API web para su uso por otros clientes, crear una implementación de SPA que aproveche estas API puede requerir menos trabajo que reproducir la lógica en el lado de servidor. Las SPA usan ampliamente las API web para consultar y actualizar datos a medida que los usuarios interactúan con la aplicación.

Cuándo elegir Blazor

La siguiente sección es una explicación más detallada de cuándo elegir Blazor para la aplicación web.

La aplicación debe exponer una interfaz de usuario enriquecida

Al igual que las SPA basadas en JavaScript, Blazor las aplicaciones pueden admitir un comportamiento de cliente enriquecido sin recargas de páginas. Estas aplicaciones son más sensibles a los usuarios, capturando solo los datos (o HTML) necesarios para responder a una interacción de usuario determinada. Diseñadas correctamente, las aplicaciones del lado Blazor servidor se pueden configurar para que se ejecuten como aplicaciones del lado Blazor cliente con cambios mínimos una vez que se admita esta característica.

El equipo está más cómodo con el desarrollo de .NET que el desarrollo de JavaScript o TypeScript

Muchos desarrolladores son más productivos con .NET y Razor que con lenguajes del lado cliente como JavaScript o TypeScript. Dado que el lado servidor de la aplicación ya se está desarrollando con .NET, el uso Blazor de garantiza que todos los desarrolladores de .NET del equipo puedan comprender y crear potencialmente el comportamiento del front-end de la aplicación.

Tabla de decisiones

En la tabla de decisión siguiente se resumen algunos de los factores básicos que se deben tener en cuenta al elegir entre una aplicación web tradicional, una SPA o una Blazor aplicación.

 Expandir tabla

Factor	Aplicación web tradicional	Aplicación de página única	Blazor Aplicación
Conocimientos necesarios del equipo con JavaScript/TypeScript	Mínimo	Obligatorio	Mínimo
Compatibilidad con exploradores sin scripting	Soportado	No compatible	Soportado

Factor	Aplicación web tradicional	Aplicación de página única	Blazor Aplicación
Comportamiento mínimo de la aplicación Client-Side	Well-Suited	Exceso	Viable
Requisitos enriquecidos y complejos de la interfaz de usuario	Limitados	Well-Suited	Well-Suited

Otras consideraciones

Las aplicaciones web tradicionales tienden a ser más sencillas y tienen mejores criterios de optimización de motor de búsqueda (SEO) que las aplicaciones SPA. Los bots de los motores de búsqueda pueden consumir fácilmente contenido de aplicaciones tradicionales, mientras que el soporte para la indexación de SPAs puede ser insuficiente o limitado. Si la aplicación se beneficia de la detección pública por parte de los motores de búsqueda, esto suele ser una consideración importante.

Además, a menos que haya creado una herramienta de administración para el contenido del SPA, puede requerir que los desarrolladores realicen cambios. Las aplicaciones web tradicionales suelen ser más fáciles para que los no desarrolladores realicen cambios, ya que gran parte del contenido es simplemente HTML y puede que no requiera un proceso de compilación para obtener una vista previa o incluso implementar. Si es probable que los no desarrolladores de su organización necesiten mantener el contenido de la aplicación, una aplicación web tradicional puede ser una mejor opción.

Las SPA brillan cuando la aplicación tiene formularios interactivos complejos u otras características de interacción del usuario. En el caso de aplicaciones complejas que requieren autenticación para su uso y, por tanto, no son accesibles por arañas del motor de búsqueda pública, las SPA son una excelente opción en muchos casos.

[Siguiendo anterior](#)

Principios arquitectónicos

19/06/2025

💡 Sugerencia

Este contenido es un extracto del libro electrónico, Arquitecto de aplicaciones web modernas con ASP.NET Core y Azure, disponible en [.NET Docs](#) o como un PDF descargable gratuito que se puede leer sin conexión.

Descargar PDF



"Si los constructores construyeran edificios de la manera en que los programadores escribieran programas, entonces el primer pájaro carpintero que pasara destruiría la civilización."

- Gerald Weinberg

Debe planificar y diseñar soluciones de software con la capacidad de mantenimiento en mente. Los principios descritos en esta sección pueden ayudarle a guiarse hacia las decisiones arquitectónicas que darán lugar a aplicaciones limpias y fáciles de mantener. Por lo general, estos principios le guiarán hacia la creación de aplicaciones fuera de componentes discretos que no están estrechamente acoplados a otras partes de la aplicación, sino que se comunican a través de interfaces explícitas o sistemas de mensajería.

Principios de diseño comunes

Separación de intereses

Un principio rector al desarrollar es **Separación de responsabilidades**. Este principio afirma que el software se debe separar en función de los tipos de trabajo que realiza. Por ejemplo, considere una aplicación que incluye lógica para identificar elementos destacados que se muestran al usuario y qué formato da formato a dichos elementos de una manera concreta

para que sean más evidentes. El comportamiento responsable de elegir qué elementos dar formato se debe mantener separado del comportamiento responsable de dar formato a los elementos, ya que estos comportamientos son preocupaciones independientes que solo están relacionadas de forma coincidente entre sí.

De forma arquitectónica, las aplicaciones se pueden compilar lógicamente para seguir este principio separando el comportamiento empresarial principal de la lógica de la infraestructura y la interfaz de usuario. Lo ideal es que las reglas de negocios y la lógica residan en un proyecto independiente, que no debe depender de otros proyectos de la aplicación. Esta separación ayuda a garantizar que el modelo de negocio es fácil de probar y puede evolucionar sin estar estrechamente acoplado a detalles de implementación de bajo nivel (también ayuda si los problemas de infraestructura dependen de las abstracciones definidas en el nivel de negocio). La separación de preocupaciones es una consideración clave detrás del uso de capas en arquitecturas de aplicaciones.

Encapsulación

Las distintas partes de una aplicación deben usar **la encapsulación** para aislarlas de otras partes de la aplicación. Los componentes y capas de la aplicación deben ser capaces de ajustar su implementación interna sin interrumpir sus colaboradores siempre que no se infrinjan los contratos externos. El uso adecuado de la encapsulación ayuda a lograr acoplamiento flexible y modularidad en los diseños de aplicación, ya que los objetos y paquetes se pueden reemplazar por implementaciones alternativas siempre que se mantenga la misma interfaz.

En las clases, la encapsulación se logra limitando el acceso externo al estado interno de la clase. Si un actor externo quiere manipular el estado del objeto, debe hacerlo a través de una función bien definida (o establecedor de propiedades), en lugar de tener acceso directo al estado privado del objeto. Del mismo modo, los componentes de aplicación y las propias aplicaciones deben exponer interfaces bien definidas para que los colaboradores usen, en lugar de permitir que su estado se modifique directamente. Este enfoque libera el diseño interno de la aplicación para evolucionar con el tiempo sin preocuparse de que hacerlo interrumpirá a los colaboradores, siempre y cuando se mantengan los contratos públicos.

El estado global mutable es antitético a la encapsulación. No se puede confiar en un valor capturado del estado global mutable en una función para tener el mismo valor en otra función (o incluso más en la misma función). Comprender los problemas con el estado global mutable es una de las razones por las que los lenguajes de programación como C# admiten diferentes reglas de ámbito, que se usan en todas partes, desde instrucciones a métodos a clases. Es importante señalar que las arquitecturas impulsadas por datos, que dependen de una base de datos central para la integración dentro y entre aplicaciones, eligen, por su naturaleza, depender del estado global mutable representado por la base de datos. Una consideración clave en el diseño controlado por dominio y la arquitectura limpia es cómo encapsular el

acceso a los datos y cómo garantizar que el estado de la aplicación no se vuelva inválido mediante el acceso directo a su formato de persistencia.

Inversión de dependencias

La dirección de dependencia dentro de la aplicación debe estar en la dirección de la abstracción, no los detalles de implementación. La mayoría de las aplicaciones se escriben de manera que la dependencia durante la compilación fluya hacia la ejecución, produciendo un gráfico de dependencias directo. Es decir, si la clase A llama a un método de la clase B y la clase B llama a un método de la clase C, la clase A dependerá de la clase B y la clase B dependerá de la clase C, como se muestra en la figura 4-1.

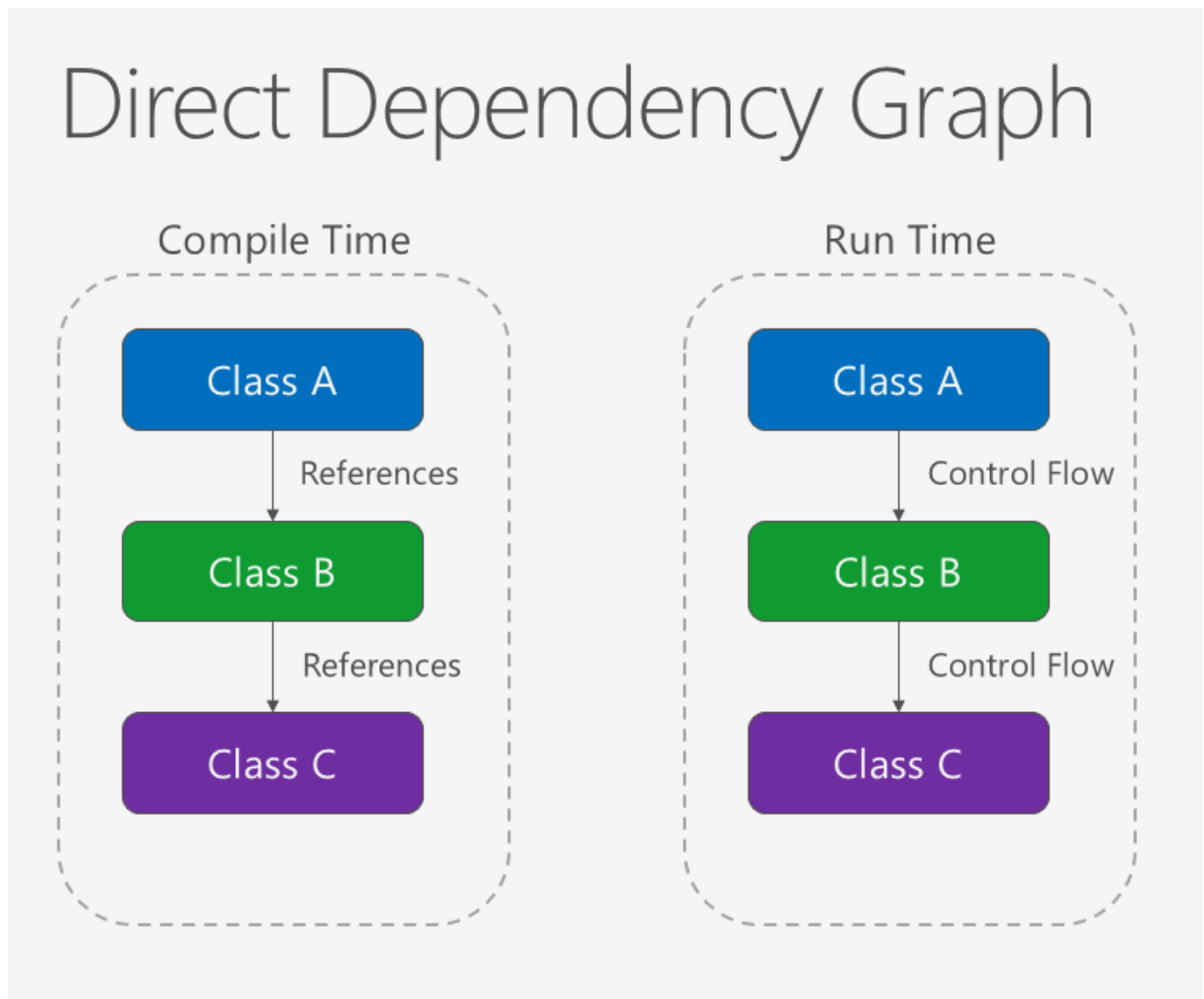


Figura 4-1. Gráfico de dependencias directa.

Aplicar el principio de inversión de dependencias permite que A llame a métodos en una abstracción que implementa B, lo que hace posible que A llame a B en tiempo de ejecución, pero que B dependa de una interfaz que controla A en tiempo de compilación (por tanto, se *invierte* la dependencia de tiempo de compilación normal). En tiempo de ejecución, el flujo de

ejecución del programa permanece sin cambios, pero la introducción de interfaces significa que se pueden conectar fácilmente diferentes implementaciones de estas interfaces.

Inverted Dependency Graph

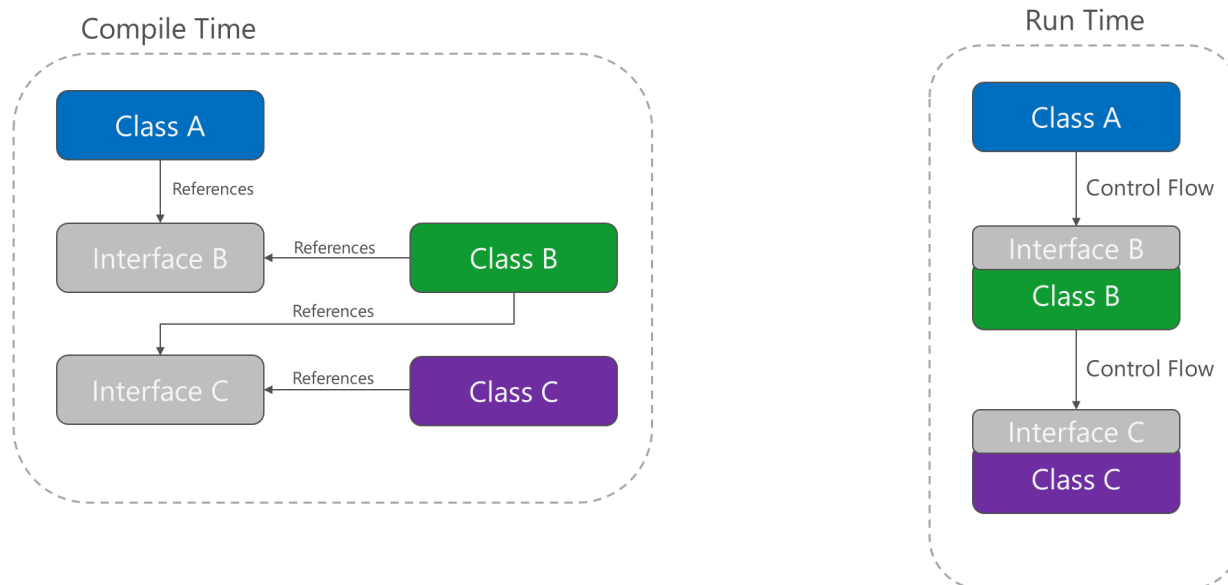


Figura 4-2. Gráfico de dependencias invertido.

La **inversión de dependencias** es una parte clave de la creación de aplicaciones acopladas de forma flexible, ya que los detalles de implementación se pueden escribir para depender e implementar abstracciones de nivel superior, en lugar de hacerlo de otra manera. Las aplicaciones resultantes son más fáciles de probar, modulares y fáciles de mantener como resultado. La práctica de inserción de *dependencias* es posible siguiendo el principio de inversión de dependencias.

Dependencias explícitas

Los métodos y clases deben requerir explícitamente los objetos de colaboración que necesiten para funcionar correctamente. Se denomina [principio de dependencias explícitas](#). Los constructores de clase proporcionan una oportunidad para que las clases identifiquen las cosas que necesitan para que estén en un estado válido y funcionen correctamente. Si define clases que se pueden construir y llamar, pero que solo funcionarán correctamente si determinados componentes globales o de infraestructura están en vigor, estas clases están siendo *deshonestas* con sus clientes. El contrato de constructor indica al cliente que solo necesita las cosas especificadas (posiblemente nada si la clase simplemente usa un constructor sin parámetros), pero en tiempo de ejecución resulta que el objeto realmente necesitaba algo más.

Siguiendo el principio de dependencias explícitas, las clases y los métodos son honestos con sus clientes sobre lo que necesitan para funcionar. Siguiendo el principio, el código es más

autodocumentado y los contratos de codificación son más fáciles de usar, ya que los usuarios confiarán en que siempre que proporcionen lo necesario en forma de parámetros de método o constructor, los objetos con los que trabajan se comportarán correctamente en tiempo de ejecución.

Responsabilidad única

El principio de responsabilidad única se aplica al diseño orientado a objetos, pero también se puede considerar como un principio arquitectónico similar a la separación de preocupaciones. Indica que los objetos deben tener solo una responsabilidad y que solo deben tener una razón para cambiar. En concreto, la única situación en la que el objeto debe cambiar es si se debe actualizar la manera en que realiza su responsabilidad. Seguir este principio ayuda a generar sistemas más flexibles y modulares, ya que muchos tipos de comportamiento nuevos se pueden implementar como nuevas clases, en lugar de agregar responsabilidad adicional a las clases existentes. Agregar nuevas clases siempre es más segura que cambiar las clases existentes, ya que aún no hay código que dependa de las nuevas clases.

En una aplicación monolítica, podemos aplicar el principio de responsabilidad única en un nivel alto a las capas de la aplicación. La responsabilidad de la presentación debe permanecer en el proyecto de interfaz de usuario, mientras que la responsabilidad de acceso a los datos debe mantenerse dentro de un proyecto de infraestructura. La lógica de negocios debe mantenerse en el proyecto principal de la aplicación, donde se puede probar fácilmente y puede evolucionar independientemente de otras responsabilidades.

Cuando este principio se aplica a la arquitectura de la aplicación y se lleva a su punto final lógico, se obtienen microservicios. Un microservicio determinado debe tener una sola responsabilidad. Si necesita ampliar el comportamiento de un sistema, normalmente es mejor hacerlo agregando microservicios adicionales, en lugar de agregar responsabilidad a uno existente.

[Más información sobre la arquitectura de microservicios](#) 

Una vez y solo una (DRY)

La aplicación debe evitar especificar el comportamiento relacionado con un concepto determinado en varios lugares, ya que esta práctica es una fuente frecuente de errores. En algún momento, un cambio en los requisitos requerirá cambiar este comportamiento. Es probable que al menos una instancia del comportamiento no se actualice y el sistema se comportará incoherentemente.

En lugar de duplicar la lógica, encapsularla en una construcción de programación. Haga que esta construcción sea la única autoridad sobre este comportamiento y que cualquier otra parte

de la aplicación que requiera este comportamiento use la nueva construcción.

⚠ Nota

Evite vincular comportamientos que solo coincidan por repetición. Por ejemplo, solo porque dos constantes diferentes tienen el mismo valor, eso no significa que solo debe tener una constante, si conceptualmente hacen referencia a cosas diferentes. La duplicación siempre es preferible a acoplarse a la abstracción equivocada.

Omisión de persistencia

La **ignorancia de persistencia** (PI) hace referencia a los tipos que deben conservarse, pero cuyo código no se ve afectado por la elección de la tecnología de persistencia. Estos tipos de .NET se conocen a veces como Objetos CLR Simples (POCOs), ya que no necesitan heredar de una clase base determinada ni implementar una interfaz determinada. El desconocimiento en materia de persistencia es valioso porque permite que el mismo modelo de negocio se implemente de varias maneras, ofreciendo así una mayor flexibilidad a la aplicación. Las opciones de persistencia pueden cambiar con el tiempo, desde una tecnología de base de datos a otra, o se podrían requerir formas adicionales de persistencia además de lo que se inicie la aplicación (por ejemplo, mediante una caché de Redis o Azure Cosmos DB además de una base de datos relacional).

Algunos ejemplos de infracciones de este principio son:

- Clase base necesaria.
- Una implementación de interfaz requerida.
- Clases responsables de guardarse a sí mismas (por ejemplo, el patrón de registro activo).
- Se requiere un constructor sin parámetros.
- Propiedades que requieren palabra clave virtual.
- Atributos específicos necesarios para la persistencia.

El requisito de que las clases tengan cualquiera de las características o comportamientos anteriores agrega acoplamiento entre los tipos que se van a conservar y la elección de la tecnología de persistencia, lo que dificulta la adopción de nuevas estrategias de acceso a datos en el futuro.

Contextos delimitados

Los contextos delimitados son un patrón central en Domain-Driven diseño. Proporcionan una manera de abordar la complejidad en aplicaciones o organizaciones de gran tamaño dividiéndose en módulos conceptuales independientes. A continuación, cada módulo conceptual representa un contexto que está separado de otros contextos (por lo tanto, delimitados) y puede evolucionar de forma independiente. Idealmente, cada contexto delimitado debe ser libre de elegir sus propios nombres para los conceptos que contiene y debe tener acceso exclusivo a su propio almacenamiento de persistencia.

Como mínimo, las aplicaciones web individuales deben esforzarse por ser su propio contexto limitado, con su propio almacén de persistencia para su modelo de negocio, en lugar de compartir una base de datos con otras aplicaciones. La comunicación entre contextos enlazados se produce a través de interfaces mediante programación, en lugar de a través de una base de datos compartida, lo que permite que la lógica de negocios y los eventos se produzcan en respuesta a los cambios que tienen lugar. Los contextos delimitados se asignan estrechamente a los microservicios que, idealmente, también se implementan como sus propios contextos delimitados individuales.

Recursos adicionales

- [Principios](#) ↗
- [Contexto delimitado](#) ↗

Siguiente anterior

Arquitecturas de aplicaciones web comunes

18/06/2025

💡 Sugerencia

Este contenido es un extracto del libro electrónico, Arquitecto de aplicaciones web modernas con ASP.NET Core y Azure, disponible en [.NET Docs](#) o como un PDF descargable gratuito que se puede leer sin conexión.

Descargar PDF



"Si cree que una buena arquitectura es costosa, pruebe una arquitectura incorrecta". -
Brian Foote y Joseph Yoder

La mayoría de las aplicaciones .NET tradicionales se implementan como unidades únicas correspondientes a un archivo ejecutable o a una sola aplicación web que se ejecuta dentro de un único appdomain de IIS. Este enfoque es el modelo de implementación más sencillo y sirve muy bien a muchas aplicaciones públicas internas y más pequeñas. Sin embargo, incluso dada esta sola unidad de implementación, la mayoría de las aplicaciones empresariales no triviales se benefician de cierta separación lógica en varias capas.

¿Qué es una aplicación monolítica?

Una aplicación monolítica es una que es totalmente independiente, en términos de su comportamiento. Puede interactuar con otros servicios o almacenes de datos en el transcurso de realizar sus operaciones, pero el núcleo de su comportamiento se ejecuta dentro de su propio proceso y toda la aplicación normalmente se implementa como una sola unidad. Si esta aplicación necesita escalar horizontalmente, normalmente toda la aplicación se duplica en varios servidores o máquinas virtuales.

Aplicaciones todo en uno

El menor número posible de proyectos para una arquitectura de aplicación es uno. En esta arquitectura, toda la lógica de la aplicación se incluye en un único proyecto, compilado en un único ensamblado e implementado como una sola unidad.

Un nuevo proyecto de ASP.NET Core, ya sea creado en Visual Studio o desde la línea de comandos, comienza como un monolito "todo en uno" simple. Contiene todo el comportamiento de la aplicación, incluida la lógica de presentación, negocio y acceso a datos. En la figura 5-1 se muestra la estructura de archivos de una aplicación de un solo proyecto.

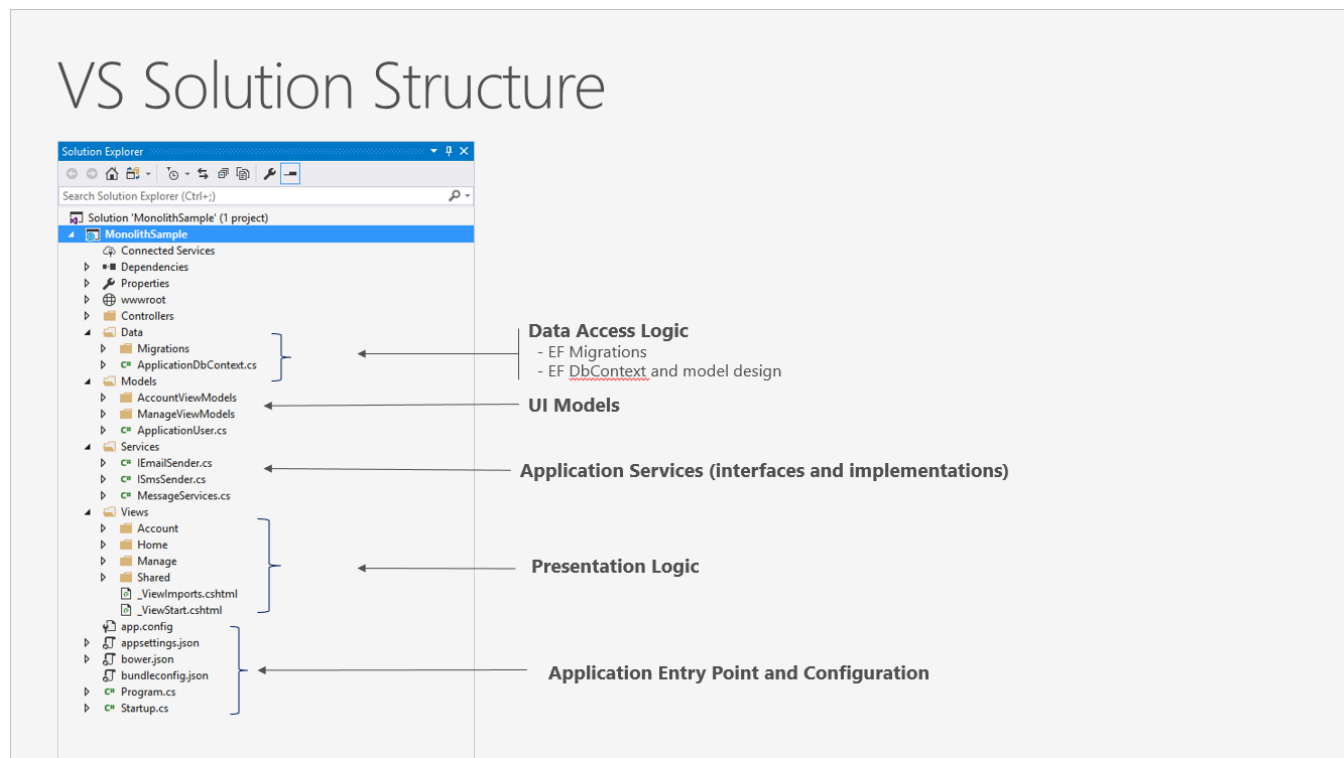


Figura 5-1. Un proyecto de aplicación ASP.NET Core único.

En un único escenario de proyecto, la separación de preocupaciones se logra mediante el uso de carpetas. La plantilla predeterminada incluye carpetas independientes para las responsabilidades del patrón MVC de modelos, vistas y controladores, así como carpetas adicionales para datos y servicios. En esta disposición, los detalles de la presentación deben limitarse tanto como sea posible a la carpeta Vistas y los detalles de implementación de acceso a datos deben limitarse a las clases que se mantienen en la carpeta Datos. La lógica de negocios debe residir en servicios y clases dentro de la carpeta Models.

Aunque es simple, la solución monolítica de un solo proyecto tiene algunas desventajas. A medida que crece el tamaño y la complejidad del proyecto, el número de archivos y carpetas seguirá creciendo también. Los problemas de la interfaz de usuario (modelos, vistas, controladores) residen en varias carpetas, que no se agrupan alfabéticamente. Este problema solo se empeora cuando se agregan construcciones de nivel de interfaz de usuario adicionales,

como Filtros o ModelBinders, en sus propias carpetas. La lógica de negocios está dispersa entre las carpetas Modelos y Servicios y no hay ninguna indicación clara de qué clases en qué carpetas deben depender de las demás. Esta falta de organización en el nivel del proyecto suele dar lugar a [código espagueti](#) [↗].

Para solucionar estos problemas, las aplicaciones suelen evolucionar en soluciones de varios proyectos, donde se considera que cada proyecto reside en una *capa* determinada de la aplicación.

¿Qué son las capas?

A medida que las aplicaciones crecen en complejidad, una manera de administrar esa complejidad es dividir la aplicación según sus responsabilidades o preocupaciones. Este enfoque sigue la separación del principio de preocupación y puede ayudar a mantener organizado un código base creciente para que los desarrolladores puedan encontrar fácilmente dónde se implementa cierta funcionalidad. Sin embargo, la arquitectura superpuesta ofrece una serie de ventajas más allá de la organización de código.

Al organizar el código en capas, se puede reutilizar la funcionalidad común de bajo nivel en toda la aplicación. Esta reutilización es beneficiosa porque significa que es necesario escribir menos código y porque puede permitir que la aplicación normalice una sola implementación, siguiendo el principio [de no repetirse \(DRY\)](#) [↗].

Con una arquitectura superpuesta, las aplicaciones pueden aplicar restricciones en las que las capas pueden comunicarse con otras capas. Esta arquitectura ayuda a lograr la encapsulación. Cuando se cambia o reemplaza una capa, solo deben verse afectadas las capas que funcionan con ella. Al limitar qué capas dependen de qué otras capas, se puede mitigar el impacto de los cambios para que un único cambio no afecte a toda la aplicación.

Las capas (y la encapsulación) facilitan mucho la sustitución de la funcionalidad dentro de la aplicación. Por ejemplo, una aplicación podría usar inicialmente su propia base de datos de SQL Server para la persistencia, pero más adelante podría optar por usar una estrategia de persistencia basada en la nube o una detrás de una API web. Si la aplicación ha encapsulado correctamente su implementación de persistencia dentro de una capa lógica, esa capa específica de SQL Server podría reemplazarse por una nueva que implemente la misma interfaz pública.

Además del potencial de intercambiar implementaciones en respuesta a cambios futuros en los requisitos, las capas de aplicación también pueden facilitar el intercambio de implementaciones con fines de prueba. En lugar de tener que escribir pruebas que funcionan con la capa de datos real o la capa de interfaz de usuario de la aplicación, estas capas se pueden reemplazar en tiempo de prueba por implementaciones falsas que proporcionan

respuestas conocidas a las solicitudes. Este enfoque normalmente hace que las pruebas sean mucho más fáciles de escribir y se ejecuten mucho más rápido en comparación con la ejecución de pruebas en la infraestructura real de la aplicación.

La capa lógica es una técnica común para mejorar la organización del código en aplicaciones de software empresarial y hay varias maneras en que el código se puede organizar en capas.

❗ Nota

Las capas representan la separación lógica dentro de la aplicación. En caso de que la lógica de la aplicación se distribuya físicamente a servidores o procesos independientes, estos destinos de implementación física independientes se conocen como *niveles*. Es posible, y bastante común, tener una aplicación de N niveles que se implementa en un solo nivel.

Aplicaciones tradicionales de arquitectura de "N niveles"

La organización más común de la lógica de aplicación en capas se muestra en la figura 5-2.

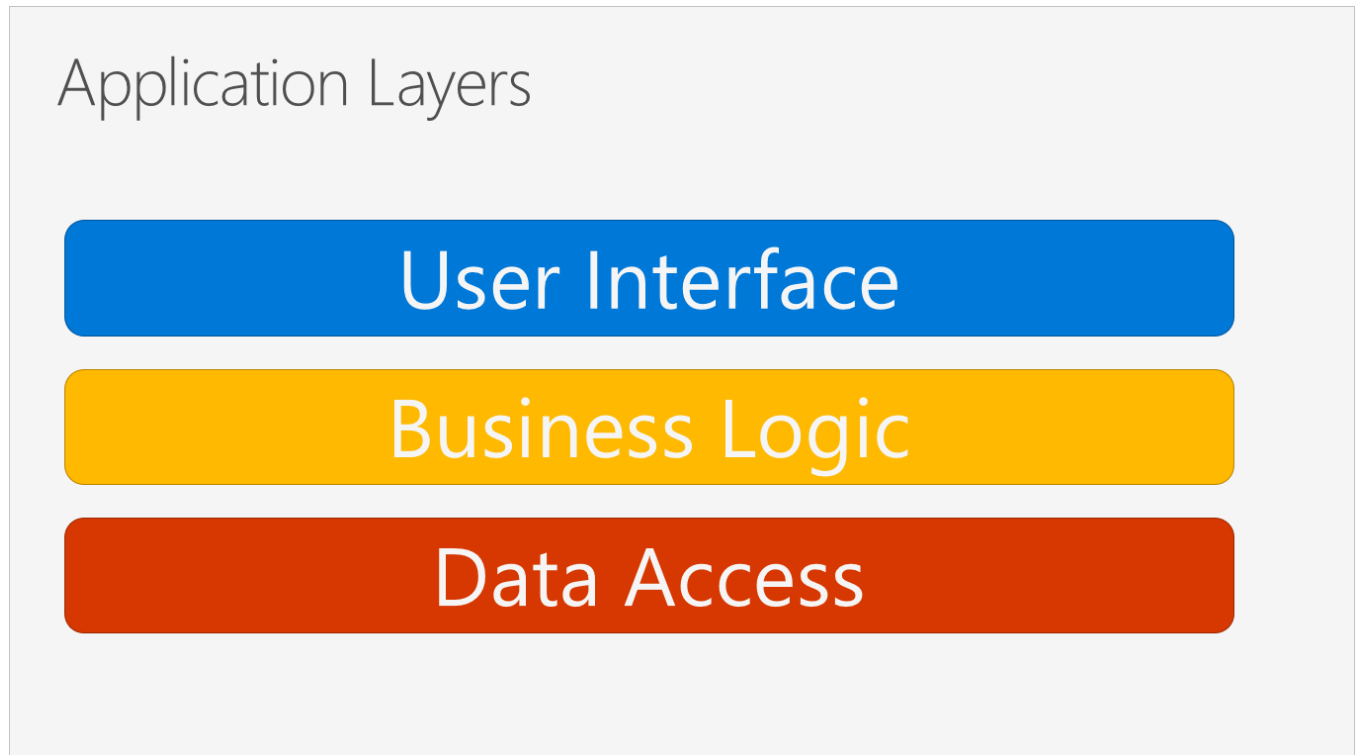


Figura 5-2. Capas de aplicación típicas.

Estas capas suelen abreviarse como UI, BLL (capa de lógica de negocios) y DAL (capa de acceso a datos). Con esta arquitectura, los usuarios realizan solicitudes a través de la capa de interfaz de usuario, que solo interactúa con el BLL. El BLL, a su vez, puede llamar al DAL para las

solicitudes de acceso a datos. La capa de interfaz de usuario no debe realizar ninguna solicitud directamente a la DAL ni debe interactuar con la persistencia directamente a través de otros medios. Del mismo modo, BLL solo debe interactuar con la persistencia a través de DAL. De esta manera, cada capa tiene su propia responsabilidad conocida.

Una desventaja de este enfoque tradicional de capas es que las dependencias en tiempo de compilación se ejecutan desde la parte superior a la inferior. Es decir, la capa de interfaz de usuario depende del BLL, que depende de la DAL. Esto significa que el BLL, que normalmente contiene la lógica más importante de la aplicación, depende de los detalles de implementación del acceso a datos (y, a menudo, de la existencia de una base de datos). La prueba de la lógica de negocios en una arquitectura de este tipo suele ser difícil, lo que requiere una base de datos de prueba. El principio de inversión de dependencias se puede usar para solucionar este problema, como verá en la sección siguiente.

En la figura 5-3 se muestra una solución de ejemplo que divide la aplicación en tres proyectos por responsabilidad (o capa).

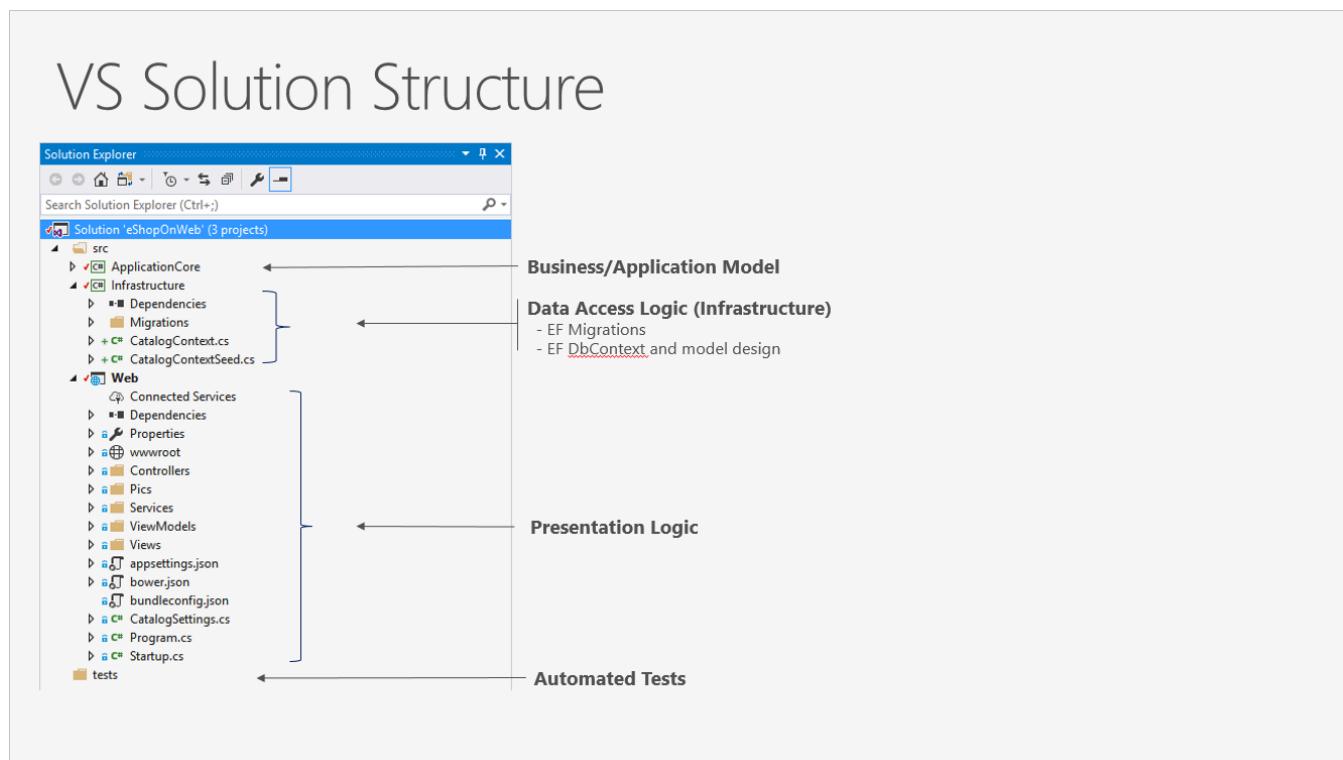


Figura 5-3. Una aplicación monolítica simple con tres proyectos.

Aunque esta aplicación usa varios proyectos para fines organizativos, todavía se implementa como una sola unidad y sus clientes interactuarán con ella como una sola aplicación web. Esto permite un proceso de implementación muy sencillo. En la figura 5-4 se muestra cómo se puede hospedar una aplicación de este tipo mediante Azure.

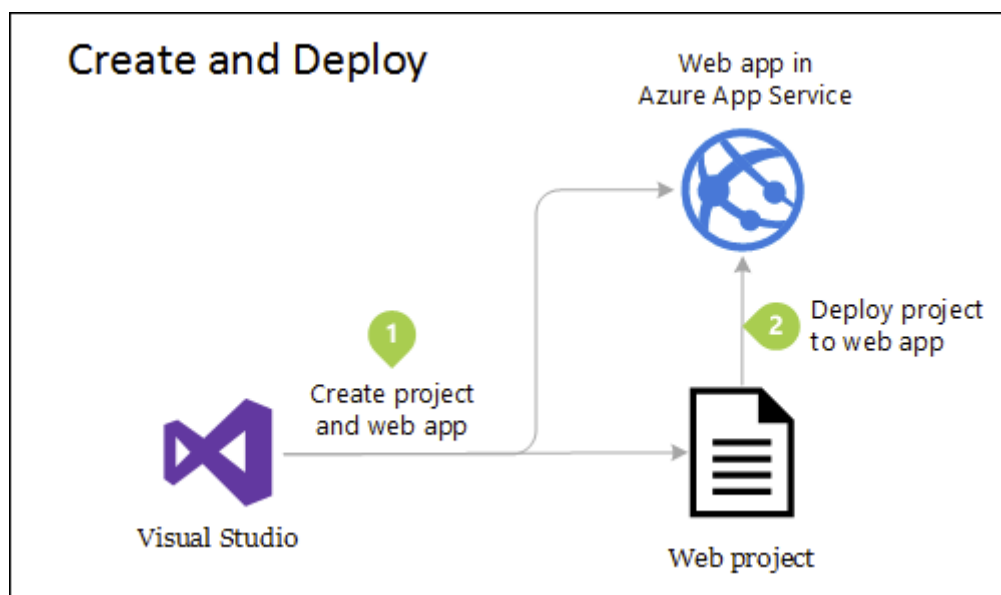


Figura 5-4. Implementación sencilla de Azure Web App

A medida que aumentan las necesidades de la aplicación, es posible que se requieran soluciones de implementación más complejas y sólidas. En la figura 5-5 se muestra un ejemplo de un plan de implementación más complejo que admite funcionalidades adicionales.

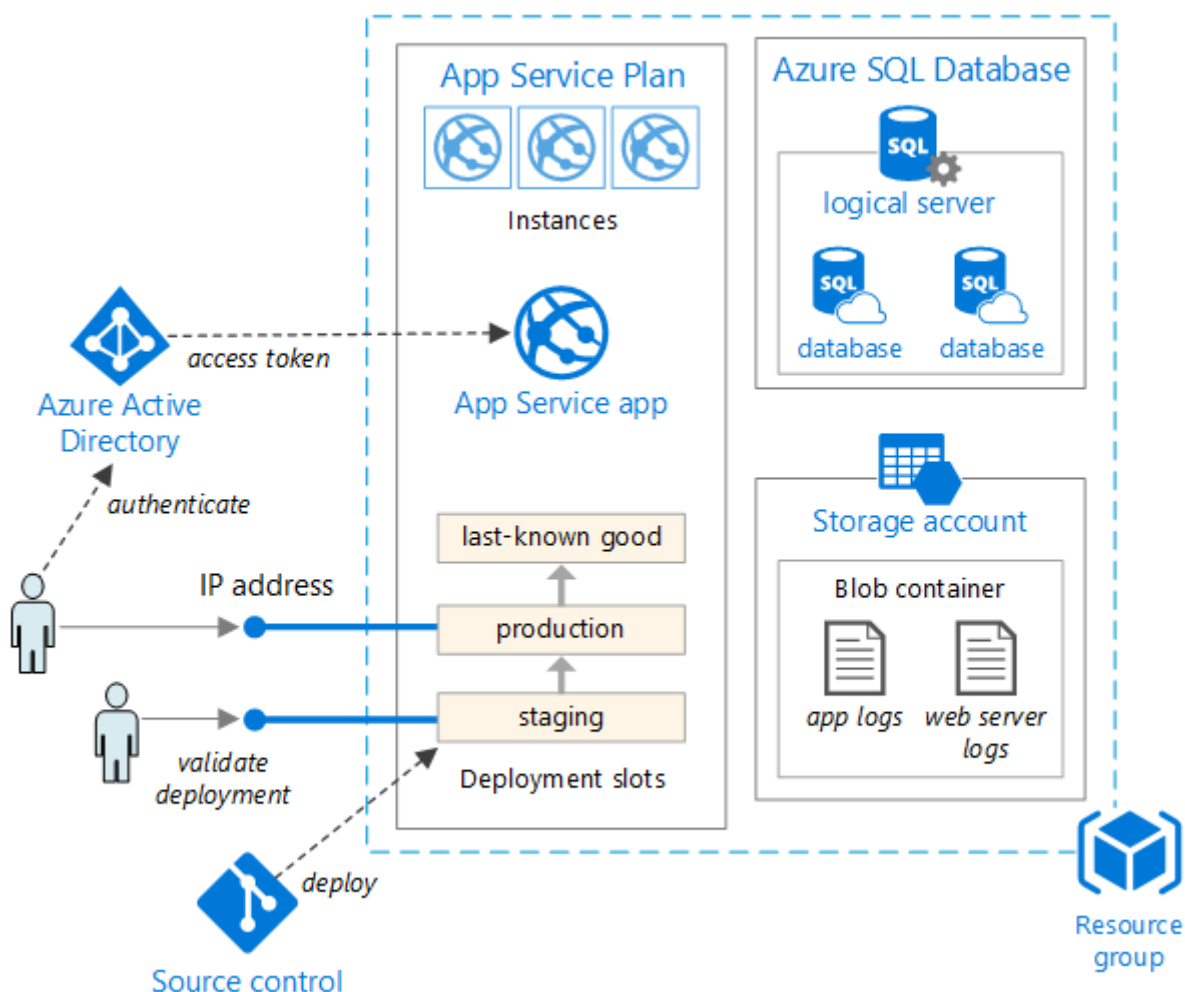


Figura 5-5. Implementación de una aplicación web en un Azure App Service

Internamente, la organización de este proyecto en varios proyectos en función de la responsabilidad mejora la capacidad de mantenimiento de la aplicación.

Esta unidad se puede escalar hacia arriba o hacia afuera para aprovechar la escalabilidad a demanda basada en la nube. El escalado vertical significa agregar cpu adicional, memoria, espacio en disco u otros recursos al servidor que hospeda la aplicación. El escalado horizontal significa agregar instancias adicionales de estos servidores, ya sean servidores físicos, máquinas virtuales o contenedores. Cuando la aplicación se hospeda en varias instancias, se usa un equilibrador de carga para asignar solicitudes a instancias de aplicación individuales.

El enfoque más sencillo para escalar una aplicación web en Azure es configurar el escalado manualmente en el plan de App Service de la aplicación. En la figura 5-6 se muestra la pantalla de panel de Azure adecuada para configurar el número de instancias que atienden una aplicación.

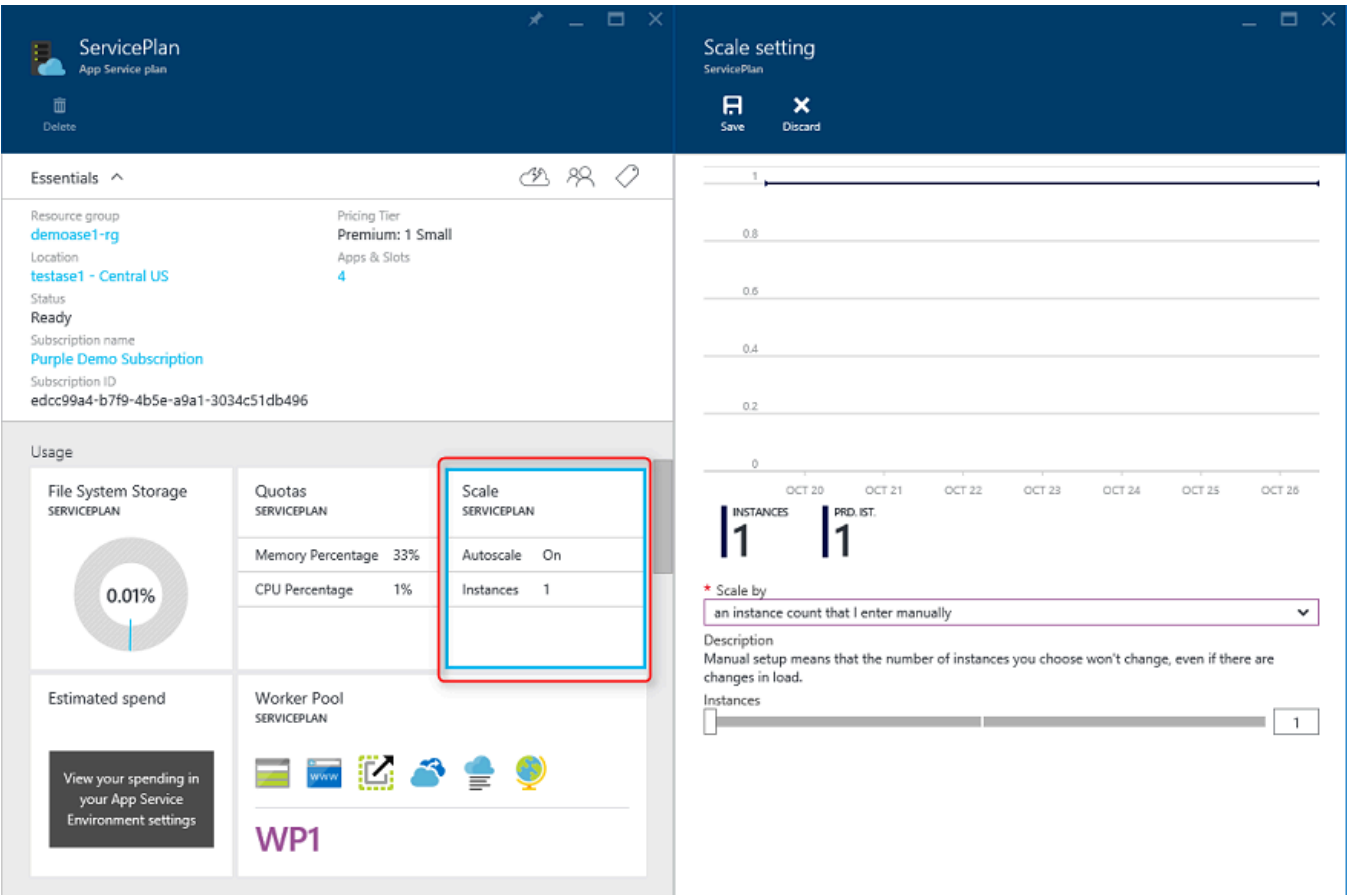


Figura 5-6. Escalamiento del plan de servicio de aplicaciones en Azure.

Arquitectura limpia

Las aplicaciones que siguen el principio de inversión de dependencias, así como los principios de diseño de Domain-Driven (DDD) tienden a llegar a una arquitectura similar. Esta arquitectura ha pasado por muchos nombres a lo largo de los años. Uno de los primeros nombres era Arquitectura hexagonal, seguido de Puertos y Adaptadores. Más recientemente,

se ha citado como [arquitectura cebolla](#) o [arquitectura limpia](#). Este último nombre, Clean Architecture, se usa como nombre para esta arquitectura en este libro electrónico.

La aplicación de referencia eShopOnWeb usa el enfoque de arquitectura limpia para organizar su código en proyectos. Puede encontrar una plantilla de solución que puede usar como punto de partida para sus propias soluciones de ASP.NET Core en el repositorio de GitHub [ardalis/cleanarchitecture](#) o [instalando la plantilla desde NuGet](#).

La arquitectura limpia coloca la lógica de negocios y el modelo de aplicación en el centro de la aplicación. En lugar de que la lógica de negocios dependa del acceso a los datos o de otras consideraciones de infraestructura, esta dependencia se invierte: ahora los detalles de la infraestructura y la implementación dependen del núcleo de la aplicación. Esta funcionalidad se logra mediante la definición de abstracciones, o interfaces, en Application Core, que luego se implementan mediante tipos definidos en el nivel de infraestructura. Una forma común de visualizar esta arquitectura es usar una serie de círculos concéntricos, similares a una cebolla. En la figura 5-7 se muestra un ejemplo de este estilo de representación arquitectónica.

Clean Architecture Layers (Onion view)

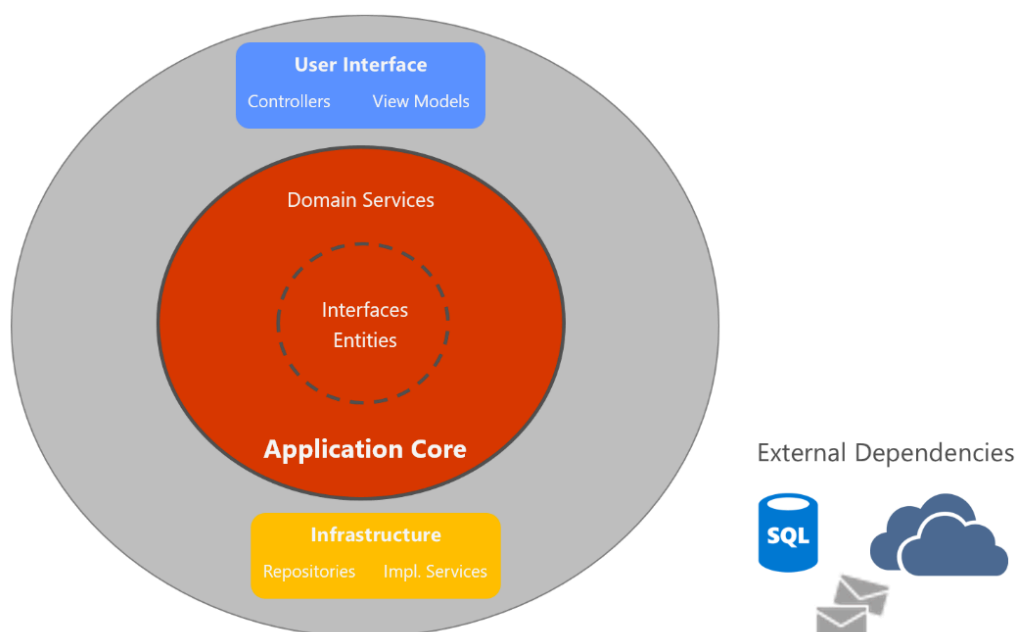


Figura 5-7. Arquitectura limpia; vista de cebolla

En este diagrama, las dependencias fluyen hacia el círculo más interno. Application Core toma su nombre de su posición en el núcleo de este diagrama. Y puede ver en el diagrama que Application Core no tiene dependencias en otras capas de aplicación. Las entidades e interfaces de la aplicación están en el centro. Justo fuera, pero aún dentro del "Application Core", se encuentran los servicios de dominio, que normalmente implementan interfaces definidas en el círculo interno. Fuera del núcleo de la aplicación, tanto la interfaz de usuario como las capas de infraestructura dependen del núcleo de la aplicación, pero no entre sí (necesariamente).

En la figura 5-8 se muestra un diagrama de capas horizontales más tradicional que refleja mejor la dependencia entre la interfaz de usuario y otras capas.

Clean Architecture Layers

-----> Optional Compile-Time Dependency
—————> Compile-Time Dependency

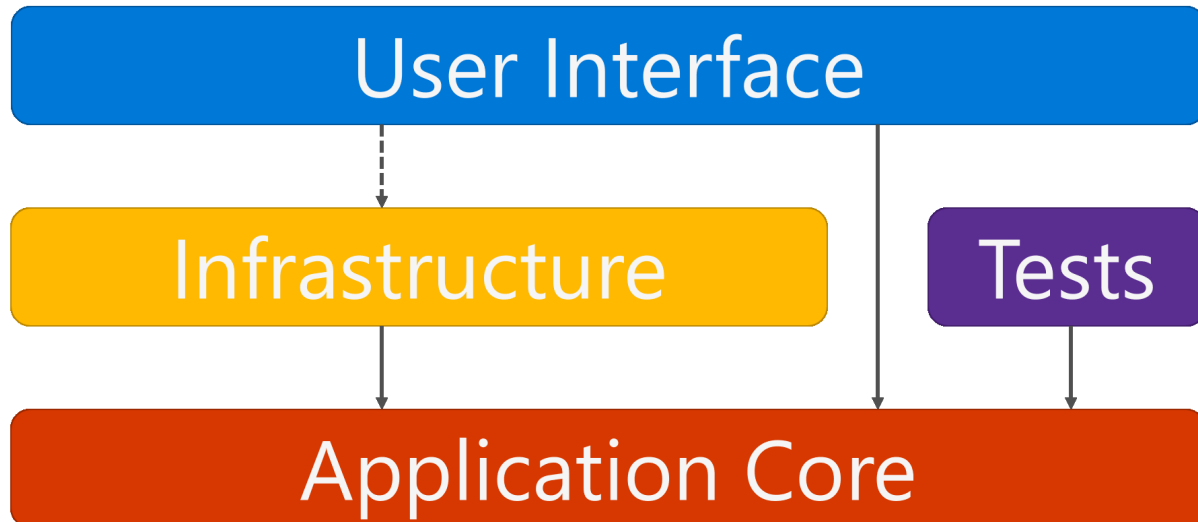


Figura 5-8. Arquitectura limpia; vista de capa horizontal

Tenga en cuenta que las flechas sólidas representan las dependencias de tiempo de compilación, mientras que la flecha discontinua representa una dependencia solo de tiempo de ejecución. Con la arquitectura limpia, la capa de interfaz de usuario funciona con interfaces definidas en Application Core en tiempo de compilación y, idealmente, no debe conocer los tipos de implementación definidos en el nivel de infraestructura. Sin embargo, en tiempo de ejecución, estos tipos de implementación son necesarios para que la aplicación se ejecute, por lo que deben estar presentes y conectados a las interfaces de Application Core a través de la inserción de dependencias.

En la figura 5-9 se muestra una vista más detallada de la arquitectura de una aplicación de ASP.NET Core cuando se compilan siguiendo estas recomendaciones.

ASP.NET Core Architecture

---> Compile Time Dependency
 —> Run Time Dependency

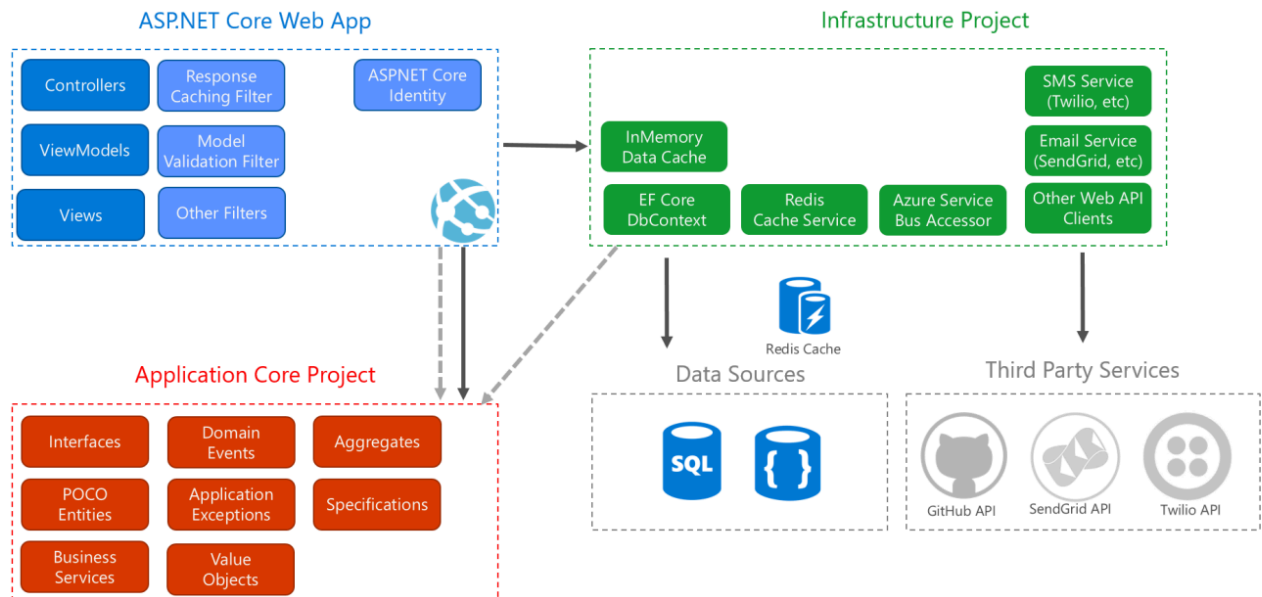


Figura 5-9. Diagrama de arquitectura de ASP.NET Core en el que se sigue la arquitectura limpia.

Dado que Application Core no depende de la infraestructura, es muy fácil escribir pruebas unitarias automatizadas para esta capa. Las figuras 5-10 y 5-11 muestran cómo encajan las pruebas en esta arquitectura.

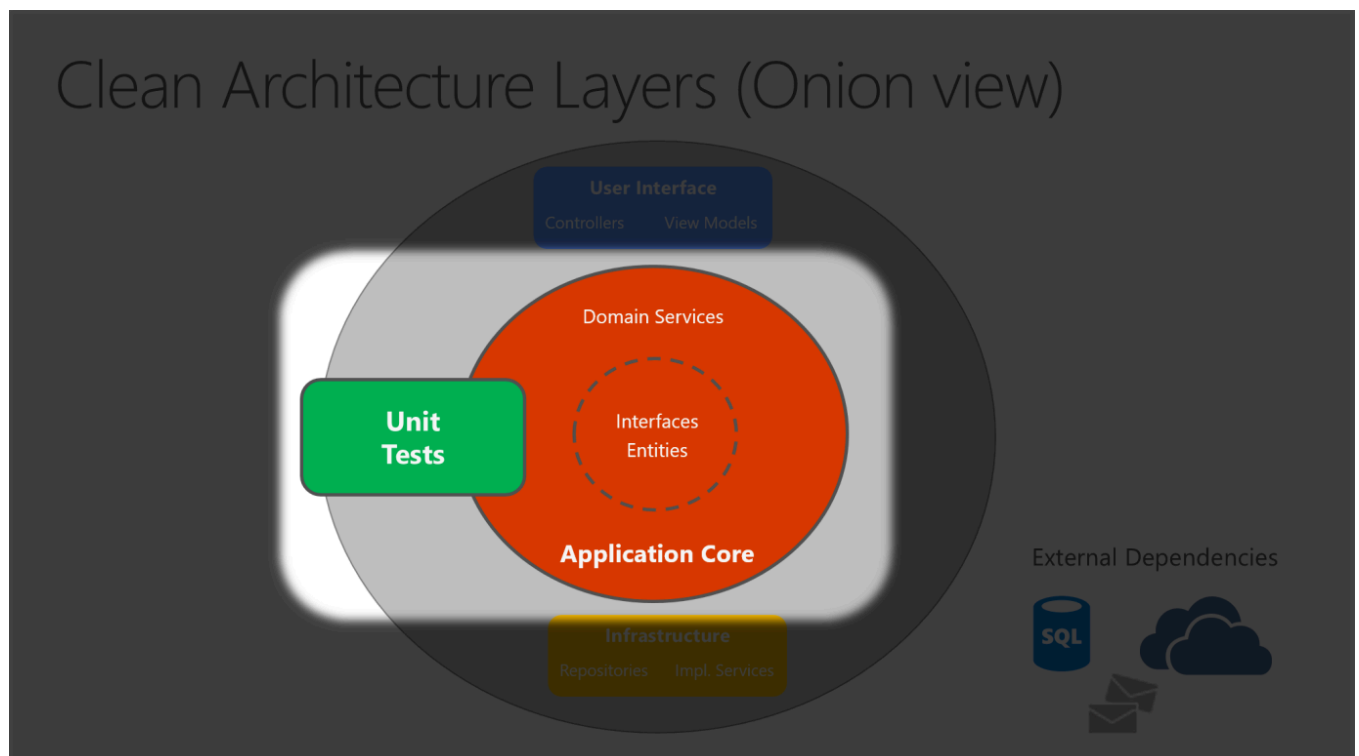


Figura 5-10. Prueba unitaria de Application Core de forma aislada.

Clean Architecture Layers (Onion view)

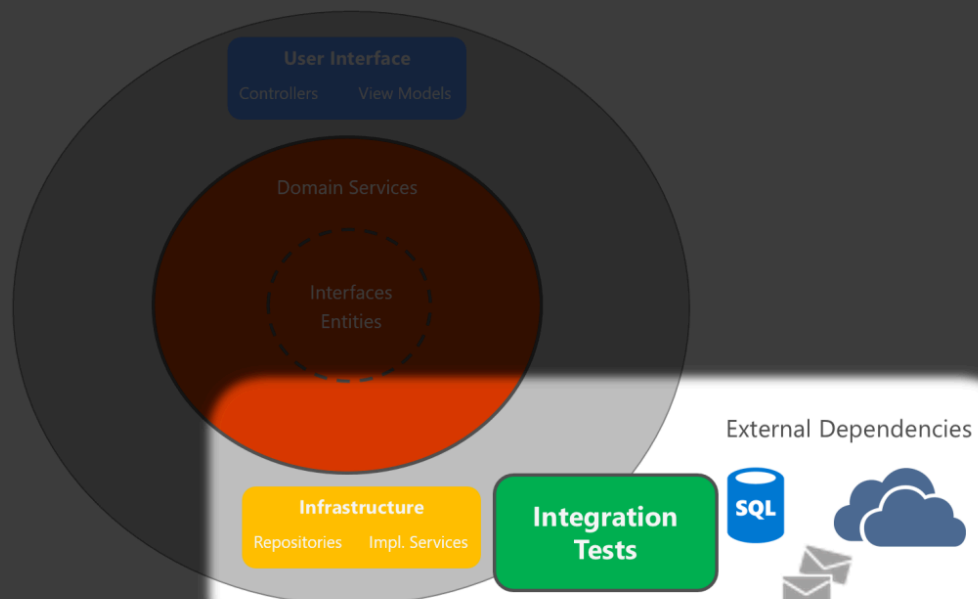


Figura 5-11. Pruebas de integración Implementaciones de infraestructura con dependencias externas.

Dado que la capa de interfaz de usuario no tiene ninguna dependencia directa de los tipos definidos en el proyecto de infraestructura, también es muy fácil intercambiar implementaciones, ya sea para facilitar las pruebas o en respuesta a los cambios en los requisitos de la aplicación. La integración incorporada en ASP.NET Core y el soporte para la inyección de dependencias hace que esta arquitectura sea la manera más adecuada de estructurar aplicaciones monolíticas no triviales.

En el caso de las aplicaciones monolíticas, los proyectos de núcleo de la aplicación, la infraestructura y la interfaz de usuario se ejecutan como una sola aplicación. La arquitectura de la aplicación en tiempo de ejecución podría tener un aspecto similar a la figura 5-12.

ASP.NET Core Architecture

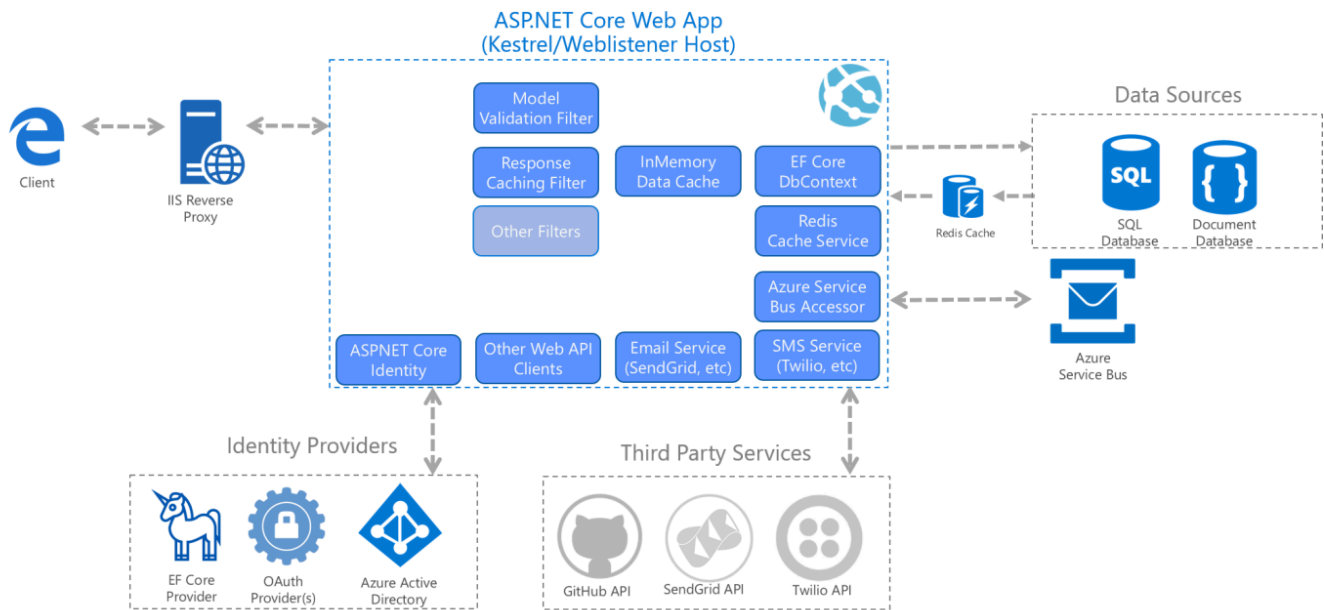


Figura 5-12. Arquitectura en tiempo de ejecución de una aplicación de ejemplo de ASP.NET Core.

Organización del código en arquitectura limpia

En una solución de arquitectura limpia, cada proyecto tiene responsabilidades claras. Por lo tanto, determinados tipos pertenecen a cada proyecto y con frecuencia encontrará carpetas correspondientes a estos tipos en el proyecto adecuado.

Núcleo de la Aplicación

Application Core contiene el modelo de negocio, que incluye entidades, servicios e interfaces. Estas interfaces incluyen abstracciones para las operaciones que se realizarán mediante infraestructura, como el acceso a datos, el acceso al sistema de archivos, las llamadas de red, etc. A veces, los servicios o interfaces definidos en esta capa tendrán que trabajar con tipos que no sean de entidad que no tengan dependencias en la interfaz de usuario o la infraestructura. Se pueden definir como objetos de transferencia de datos simples (DTO).

Tipos principales de la aplicación

- Entidades (clases de modelo de negocio que se conservan)
- Agregados (grupos de entidades)
- Interfaces
- Servicios de dominio
- Características técnicas

- Excepciones personalizadas y cláusulas de protección
- Controladores y eventos de dominio

Infraestructura

Normalmente, el proyecto de infraestructura incluye implementaciones de acceso a datos. En una aplicación web típica de ASP.NET Core, estas implementaciones incluyen Entity Framework (EF) DbContext, cualquier objeto de EF Core `Migration` que se haya definido y las clases de implementación de acceso a datos. La manera más común de abstraer el código de implementación de acceso a datos es mediante el uso del [patrón de diseño repositorio](#).

Además de las implementaciones de acceso a datos, el proyecto de infraestructura debe contener implementaciones de servicios que deben interactuar con problemas de infraestructura. Estos servicios deben implementar interfaces definidas en Application Core, por lo que la infraestructura debe tener una referencia al proyecto de Application Core.

Tipos de infraestructura

- Tipos de EF Core (`DbContext`, `Migration`)
- Tipos de implementación de acceso a datos (repositorios)
- Servicios específicos de la infraestructura (por ejemplo, `FileLogger` o `SmtpNotifier`)

Capa de interfaz de usuario

La capa de interfaz de usuario de una aplicación ASP.NET Core MVC es el punto de entrada de la aplicación. Este proyecto debe hacer referencia al proyecto de Application Core y sus tipos deben interactuar con la infraestructura estrictamente a través de interfaces definidas en Application Core. No se debe permitir la instanciación directa ni las llamadas estáticas a los tipos de la capa de infraestructura en la capa de interfaz de usuario.

Tipos de capa de interfaz de usuario

- Controladores
- Filtros personalizados
- Middleware personalizado
- Vistas
- ViewModels
- Empresa emergente

La `Startup` clase o *Program.cs* archivo es responsable de configurar la aplicación y de conectar tipos de implementación a interfaces. El lugar donde se realiza esta lógica se conoce como *raíz*

de composición de la aplicación y es lo que permite que la inserción de dependencias funcione correctamente en tiempo de ejecución.

ⓘ Nota

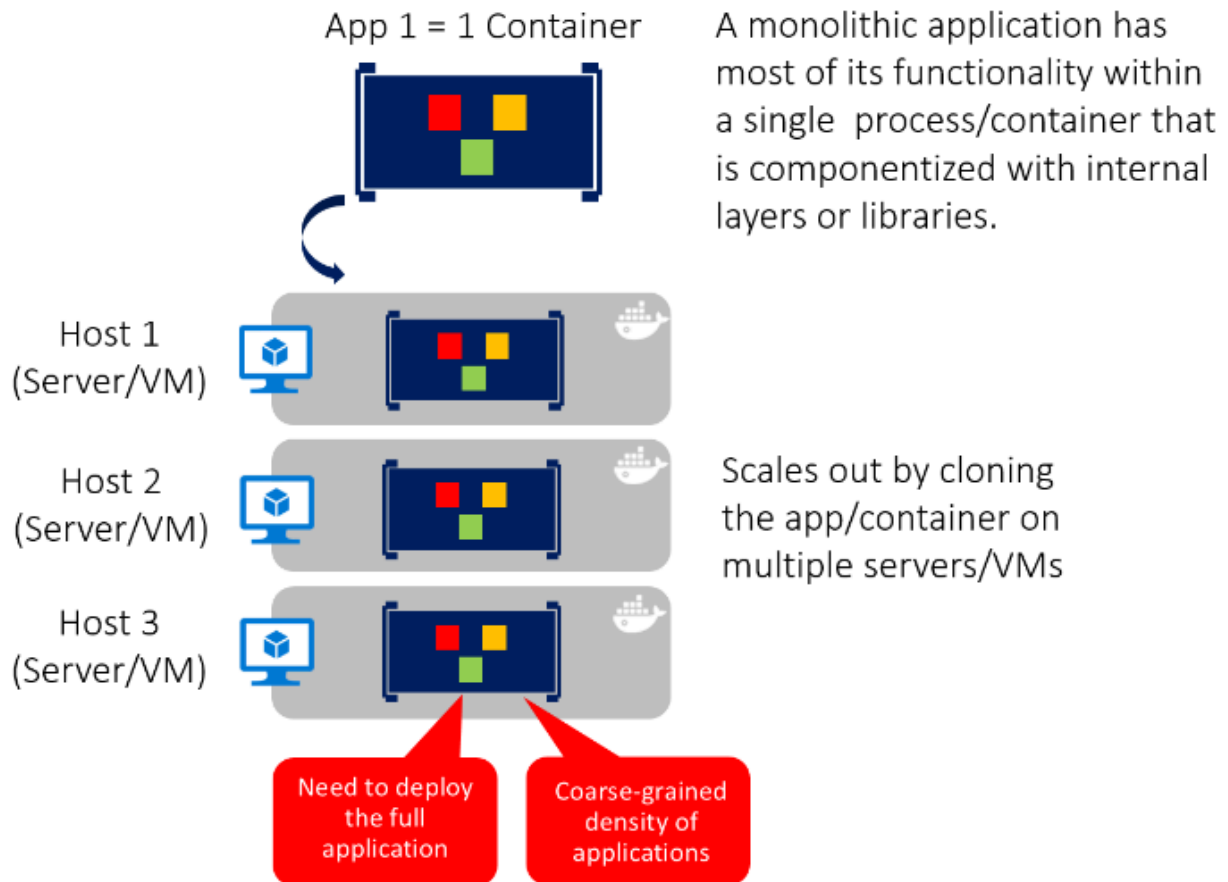
Para configurar la inyección de dependencias durante el inicio de la aplicación, es posible que el proyecto de la capa de interfaz de usuario tenga que hacer referencia al proyecto de infraestructura. Esta dependencia se puede eliminar más fácilmente mediante un contenedor de inserción de dependencias personalizado que tiene compatibilidad integrada para cargar tipos desde ensamblados. Para los fines de este ejemplo, el enfoque más sencillo es permitir que el proyecto de interfaz de usuario haga referencia al proyecto de infraestructura (pero los desarrolladores deben limitar las referencias reales a los tipos del proyecto de infraestructura a la raíz de composición de la aplicación).

Aplicaciones y contenedores monolíticos

Puede crear una aplicación web o un servicio basados en una implementación única y monolítica e implementarla como un contenedor. Dentro de la aplicación, es posible que no sea monolítico, sino organizado en varias bibliotecas, componentes o capas. Externamente, es un único contenedor con un único proceso, una sola aplicación web o un único servicio.

Para administrar este modelo, implemente un único contenedor para representar la aplicación. Para escalar, solo tiene que agregar más copias con un equilibrador de carga delante. La simplicidad procede de administrar una sola implementación en un único contenedor o máquina virtual.

Monolithic Containerized application



Puede incluir varios componentes o bibliotecas o capas internas dentro de cada contenedor, como se muestra en la figura 5-13. Sin embargo, siguiendo el principio de contenedor de *"un contenedor hace una cosa y lo hace en un proceso"*, el patrón monolítico podría ser un conflicto.

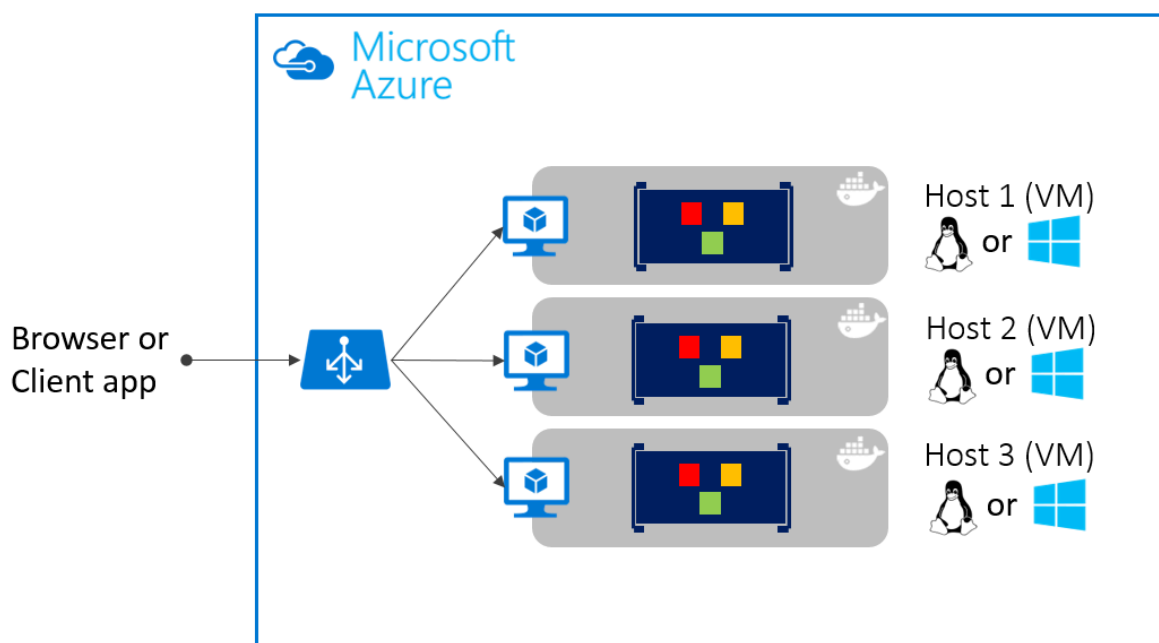
La desventaja de este enfoque viene si o cuando crece la aplicación, lo que requiere que se escale. Si la aplicación se escala completamente, no será realmente un problema. Pero en la mayoría de los casos, solo algunos elementos de la aplicación son los puntos de obstrucción que deben escalarse, mientras que otros componentes se usan menos.

Utilizando el ejemplo típico de comercio electrónico, lo que es probable que necesite escalar es el componente de información del producto. Muchos más clientes examinan productos que los compran. Más clientes usan su cesta que usar la canalización de pago. Menos clientes agregan comentarios o ven su historial de compras. Y es probable que solo tenga unos pocos empleados, en una sola región, que necesiten administrar las campañas de contenido y marketing. Al escalar el diseño monolítico, todo el código se implementa varias veces.

Además del problema de "escalar todo", los cambios en un único componente requieren una nueva prueba completa de toda la aplicación y una reimplementación completa de todas las instancias.

El enfoque monolítico es común y muchas organizaciones están desarrollando con este enfoque arquitectónico. Muchos están obteniendo resultados lo suficientemente buenos, mientras que otros se están encontrando con límites. Muchos diseñaron sus aplicaciones en este modelo, ya que las herramientas y la infraestructura eran demasiado difíciles de compilar arquitecturas orientadas a servicios (SOA) y no vieron la necesidad hasta que creció la aplicación. Si encuentra que alcanza los límites del enfoque monolítico, dividir la aplicación para que pueda aprovechar mejor los contenedores y microservicios puede ser el siguiente paso lógico.

Architecture in Docker infrastructure for monolithic applications



La implementación de aplicaciones monolíticas en Microsoft Azure se puede lograr mediante máquinas virtuales dedicadas para cada instancia. Con [Conjuntos de escalado](#) de máquinas virtuales de Azure, puede escalar fácilmente las máquinas virtuales. [Azure App Services](#) [↗] puede ejecutar aplicaciones monolíticas y escalar fácilmente instancias sin tener que administrar las máquinas virtuales. Azure App Services también puede ejecutar instancias únicas de contenedores de Docker, lo que simplifica la implementación. Con Docker, puede implementar una sola máquina virtual como host de Docker y ejecutar varias instancias. Con el equilibrador de Azure, como se muestra en la figura 5-14, puede administrar el escalado.

La implementación en los distintos hosts se puede administrar con técnicas de implementación tradicionales. Los hosts de Docker se pueden administrar con comandos como **docker** **ejecutados** manualmente o a través de la automatización, como canalizaciones de entrega continua (CD).

Aplicación monolítica implementada como contenedor

Existen ventajas de usar contenedores para administrar implementaciones de aplicaciones monolíticas. El escalado de las instancias de contenedores es mucho más rápido y sencillo que la implementación de máquinas virtuales adicionales. Incluso cuando se usan conjuntos de escalado de máquinas virtuales para escalar las máquinas virtuales, la creación tarda un tiempo. Cuando se implementa como instancias de aplicación, la configuración de la aplicación se administra como parte de la máquina virtual.

La implementación de actualizaciones como imágenes de Docker es mucho más rápida y eficiente en cuanto a la red. Normalmente, las imágenes de Docker se inician en segundos, lo que acelera las implementaciones. Eliminar una instancia de Docker es tan fácil como emitir un comando `docker stop`, que normalmente se completa en menos de un segundo.

Dado que los contenedores son intrínsecamente inmutables por diseño, nunca es necesario preocuparse por las máquinas virtuales dañadas, mientras que los scripts de actualización pueden olvidar tener en cuenta alguna configuración o archivo específicos que quedan en el disco.

Puede usar contenedores de Docker para una implementación monolítica de aplicaciones web más sencillas. Este enfoque mejora la integración continua y las canalizaciones de implementación continua y ayuda a lograr un éxito de implementación en producción. Ya no tendrá que pensar en por qué no funciona en producción, aunque sí que funcione en su equipo.

Una arquitectura basada en microservicios tiene muchas ventajas, pero estas ventajas conllevan un costo mayor de complejidad. En algunos casos, los costos superan las ventajas, por lo que una aplicación de implementación monolítica que se ejecuta en un solo contenedor o en solo unos pocos contenedores es una mejor opción.

Es posible que una aplicación monolítica no se pueda descomponer fácilmente en microservicios bien separados. Los microservicios deben funcionar independientemente entre sí para proporcionar una aplicación más resistente. Si no puede entregar segmentos de características independientes de la aplicación, separarlo solo agrega complejidad.

Es posible que una aplicación aún no necesite escalar características de forma independiente. Muchas aplicaciones, cuando necesitan escalar más allá de una sola instancia, pueden hacerlo a través del proceso relativamente simple de clonación de esa instancia completa. El trabajo adicional para separar la aplicación en servicios discretos proporciona una ventaja mínima cuando el escalado de instancias completas de la aplicación es sencillo y rentable.

Al principio del desarrollo de una aplicación, es posible que no tenga una idea clara de dónde están los límites funcionales naturales. A medida que desarrolle un producto mínimo viable, es

posible que la separación natural aún no haya surgido. Algunas de estas condiciones pueden ser temporales. Puede empezar por crear una aplicación monolítica y, posteriormente, separar algunas características que se van a desarrollar e implementar como microservicios. Es posible que otras condiciones sean esenciales para el espacio de problemas de la aplicación, lo que significa que la aplicación nunca se puede dividir en varios microservicios.

La separación de una aplicación en muchos procesos discretos también presenta sobrecarga. Hay más complejidad en separar las características en distintos procesos. Los protocolos de comunicación se vuelven más complejos. En lugar de llamadas a métodos, debe usar comunicaciones asincrónicas entre servicios. Cuando cambie a una arquitectura de microservicios, deberá agregar muchos de los bloques de creación que se implementan en la versión de microservicios de la aplicación eShopOnContainers: control de bus de eventos, reintentos y resistencia de mensajes, coherencia eventual y mucho más.

La aplicación de referencia eShopOnWeb, mucho más sencilla, ofrece soporte para el uso de un contenedor monolítico único. La aplicación incluye una aplicación web que incluye vistas de MVC tradicionales, API web y Páginas de Razor. Opcionalmente, puede ejecutar el componente de administración basado en Blazor de la aplicación, que requiere que también se ejecute un proyecto de API independiente.

La aplicación se puede iniciar desde la raíz de la solución mediante los `docker-compose build` comandos y `docker-compose up`. Este comando configura un contenedor para la instancia web, mediante el `Dockerfile` que se encuentra en la raíz del proyecto web y ejecuta el contenedor en un puerto especificado. Puede descargar el origen de esta aplicación desde GitHub y ejecutarlo localmente. Incluso esta aplicación monolítica se beneficia de la implementación en un entorno de contenedor.

Para una, la implementación en contenedores significa que cada instancia de la aplicación se ejecuta en el mismo entorno. Este enfoque incluye el entorno de desarrollador en el que se realizan las primeras pruebas y el desarrollo. El equipo de desarrollo puede ejecutar la aplicación en un entorno contenedorizado que coincida con el entorno de producción.

Además, las aplicaciones contenedorizadas se escalan horizontalmente con un costo menor. El uso de un entorno de contenedor permite un uso compartido de recursos mayor que los entornos de máquina virtual tradicionales.

Por último, la contenedorización de la aplicación fuerza una separación entre la lógica de negocios y el servidor de almacenamiento. A medida que la aplicación se expande, todos los contenedores dependerán de un solo medio físico de almacenamiento. Este medio de almacenamiento normalmente sería un servidor de alta disponibilidad que ejecuta una base de datos de SQL Server.

Compatibilidad con Docker

El `eShopOnWeb` proyecto se ejecuta en .NET. Por lo tanto, puede ejecutarse en contenedores basados en Linux o basados en Windows. Tenga en cuenta que para la implementación de Docker, quiere usar el mismo tipo de host para SQL Server. Los contenedores basados en Linux permiten una superficie más pequeña y se prefieren.

Puede usar Visual Studio 2017 o posterior para agregar compatibilidad con Docker a una aplicación existente; para ello, haga clic con el botón derecho en un proyecto en el **Explorador** de soluciones y elija **Agregar>compatibilidad con Docker**. En este paso se agregan los archivos necesarios y se modifica el proyecto para usarlos. El ejemplo actual `eShopOnWeb` ya tiene estos archivos en su lugar.

El archivo `docker-compose.yml` de nivel de solución contiene información sobre qué imágenes se van a compilar y qué contenedores se van a iniciar. El archivo permite usar el `docker-compose` comando para iniciar varias aplicaciones al mismo tiempo. En este caso, solo se lanza el proyecto web. También puede usarlo para configurar dependencias, como un contenedor de base de datos independiente.

yml

```
version: '3'

services:
  eshopwebmvc:
    image: eshopwebmvc
    build:
      context: .
      dockerfile: src/Web/Dockerfile
    environment:
      - ASPNETCORE_ENVIRONMENT=Development
    ports:
      - "5106:5106"

networks:
  default:
    external:
      name: nat
```

El archivo `docker-compose.yml` hace referencia a `Dockerfile` en el proyecto `Web`. `Dockerfile` se usa para especificar qué contenedor base se usará y cómo se configurará la aplicación en él. El `Dockerfile` de `Web`:

Dockerfile

```
FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build
WORKDIR /app

COPY *.sln .
COPY . .
WORKDIR /app/src/Web
RUN dotnet restore

RUN dotnet publish -c Release -o out

FROM mcr.microsoft.com/dotnet/aspnet:8.0 AS runtime
WORKDIR /app
COPY --from=build /app/src/Web/out ./

ENTRYPOINT ["dotnet", "Web.dll"]
```

Solución de problemas de Docker

Una vez que ejecute la aplicación en contenedor, seguirá ejecutándose hasta que se detenga. Puede ver qué contenedores se ejecutan con el `docker ps` comando. Puede detener un contenedor en ejecución mediante el `docker stop` comando y especificar el identificador del contenedor.

Tenga en cuenta que la ejecución de contenedores de Docker puede estar enlazada a los puertos que podría intentar usar en el entorno de desarrollo. Si intenta ejecutar o depurar una aplicación con el mismo puerto que un contenedor de Docker en ejecución, recibirá un error que indica que el servidor no se puede enlazar a ese puerto. Una vez más, detener el contenedor debe resolver el problema.

Si quiere agregar compatibilidad con Docker a la aplicación mediante Visual Studio, asegúrese de que Docker Desktop se está ejecutando al hacerlo. El asistente no se ejecutará correctamente si Docker Desktop no se está ejecutando al iniciar el asistente. Además, el asistente examina la opción de contenedor actual para agregar la compatibilidad correcta con Docker. Si quiere agregar compatibilidad con contenedores de Windows, debe ejecutar el asistente mientras tiene Docker Desktop en ejecución con contenedores de Windows configurados. Si quiere agregar compatibilidad con contenedores de Linux, ejecute el asistente mientras Docker se ejecuta con contenedores de Linux configurados.

Otros estilos de arquitectura de aplicaciones web

- **Web-Queue-Worker:** Los componentes principales de esta arquitectura son un front-end web que atiende solicitudes de cliente y un worker que realiza tareas que consumen

muchos recursos, flujos de trabajo de ejecución prolongada o trabajos por lotes. El front-end web se comunica con el trabajador a través de una cola de mensajes.

- **N niveles:** una arquitectura de N niveles divide una aplicación en capas lógicas y niveles físicos.
- **Microservicio:** una arquitectura de microservicios consta de una colección de servicios pequeños y autónomos. Cada uno de servicio es independiente y debe implementar una funcionalidad de negocio individual dentro de un contexto delimitado.

Referencias: arquitecturas web comunes

- **Arquitectura limpia**
<https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html> ↗
- **Arquitectura Onion**
<https://jeffreypalermo.com/blog/the-onion-architecture-part-1/> ↗
- **El patrón de repositorio**
<https://deviq.com/repository-pattern/> ↗
- **Plantilla de solución de arquitectura limpia**
<https://github.com/ardalis/cleanarchitecture> ↗
- **Libro electrónico sobre la arquitectura de microservicios**
<https://aka.ms/MicroservicesEbook> ↗
- **DDD (Domain-Driven Design)**
<https://learn.microsoft.com/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/>

Siguiente anterior

Tecnologías web comunes del lado del cliente

18/06/2025

💡 Sugerencia

Este contenido es un extracto del libro electrónico, Arquitecto de aplicaciones web modernas con ASP.NET Core y Azure, disponible en [.NET Docs](#) o como un PDF descargable gratuito que se puede leer sin conexión.

Descargar PDF



"Los sitios web deben ser buenos desde dentro y fuera." - *Paul Cookson*

ASP.NET aplicaciones principales son aplicaciones web y normalmente se basan en tecnologías web del lado cliente como HTML, CSS y JavaScript. Al separar el contenido de la página (html) de su diseño y estilo (CSS) y su comportamiento (a través de JavaScript), las aplicaciones web complejas pueden aprovechar el principio separación de preocupaciones. Los cambios futuros en la estructura, el diseño o el comportamiento de la aplicación se pueden realizar más fácilmente cuando estos problemas no están entrelazados.

Aunque HTML y CSS son relativamente estables, JavaScript, por medio de los marcos de aplicaciones y utilidades con los que los desarrolladores trabajan para crear aplicaciones basadas en web, está evolucionando a una velocidad vertiginosa. En este capítulo se examinan algunas maneras en que los desarrolladores web usan JavaScript y se proporciona información general de alto nivel de las bibliotecas del lado cliente de Angular y React.

⚠️ Nota

Blazor proporciona una alternativa a los marcos de JavaScript para crear interfaces de usuario cliente enriquecidas e interactivas.

HTML

HTML es el lenguaje de marcado estándar que se usa para crear páginas web y aplicaciones web. Sus elementos forman los bloques de creación de páginas, que representan texto con formato, imágenes, entradas de formulario y otras estructuras. Cuando un explorador realiza una solicitud a una dirección URL, tanto si captura una página como una aplicación, lo primero que se devuelve es un documento HTML. Este documento HTML puede hacer referencia o incluir información adicional sobre su apariencia y diseño en forma de CSS o comportamiento en forma de JavaScript.

CSS

CSS (Hojas de estilos en cascada) se usa para controlar la apariencia y el diseño de los elementos HTML. Los estilos CSS se pueden aplicar directamente a un elemento HTML, definido por separado en la misma página o definidos en un archivo independiente y al que hace referencia la página. Los estilos se aplican en cascada según cómo se utilizan para seleccionar un determinado elemento HTML. Por ejemplo, un estilo podría aplicarse a un documento completo, pero se reemplazaría por un estilo que se aplica a un elemento determinado. Del mismo modo, un estilo específico del elemento se reemplazaría por un estilo que se aplicaba a una clase CSS que se aplicó al elemento, lo que, a su vez, se invalidaría mediante un estilo destinado a una instancia específica de ese elemento (a través de su identificador). Figura 6-1

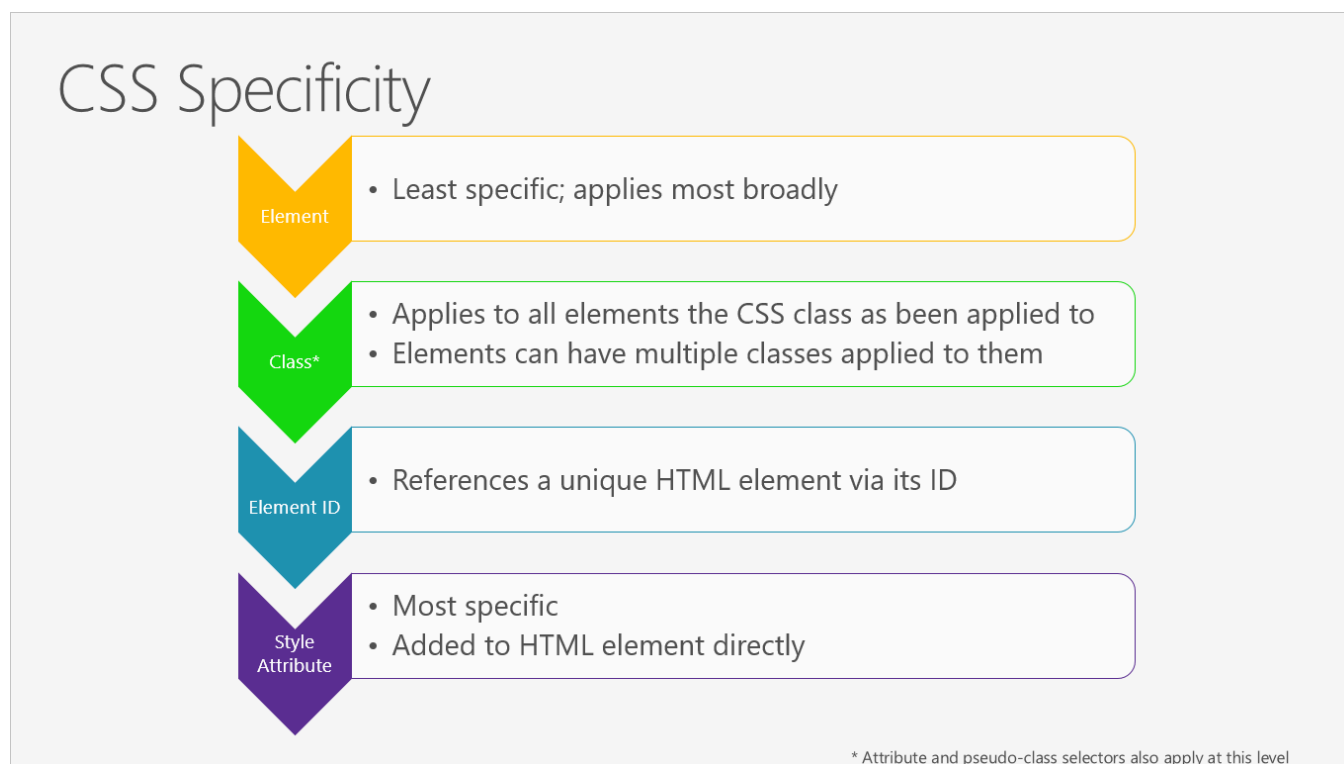


Figura 6-1. Reglas de especificidad de CSS, en orden.

Es mejor mantener los estilos en sus propios archivos de hoja de estilos independientes y usar la cascada basada en selección para implementar estilos coherentes y reutilizables dentro de la aplicación. La colocación de reglas de estilo en HTML debe evitarse y aplicar estilos a elementos individuales específicos (en lugar de clases completas de elementos o elementos que han tenido una clase CSS determinada aplicada) debe ser la excepción, no la regla.

Preprocesadores CSS

Las hojas de estilos CSS carecen de compatibilidad con la lógica condicional, las variables y otras características del lenguaje de programación. Por lo tanto, las hojas de estilos grandes suelen incluir un poco de repetición, ya que el mismo color, fuente u otro valor se aplica a muchas variaciones diferentes de elementos HTML y clases CSS. Los preprocesadores CSS pueden ayudar a las hojas de estilos a seguir el [principio DRY](#) [↗](#) agregando compatibilidad con variables y lógicas.

Los preprocesadores CSS más populares son Sass y LESS. Ambos amplían CSS y son compatibles con las versiones anteriores, lo que significa que un archivo CSS sin formato es un archivo Sass o LESS válido. Sass se basa en Ruby y LESS se basa en JavaScript y normalmente se ejecutan como parte del proceso de desarrollo local. Ambos tienen herramientas de línea de comandos disponibles, así como compatibilidad integrada en Visual Studio para ejecutarlas mediante tareas de Gulp o Grunt.

JavaScript

JavaScript es un lenguaje de programación dinámico e interpretado que se ha estandarizado en la especificación del lenguaje ECMAScript. Es el lenguaje de programación de la web. Al igual que CSS, JavaScript se puede definir como atributos dentro de elementos HTML, como bloques de script dentro de una página o en archivos independientes. Al igual que CSS, se recomienda organizar JavaScript en archivos independientes, lo que lo mantiene separado tanto como sea posible del HTML que se encuentra en páginas web individuales o vistas de aplicación.

Al trabajar con JavaScript en la aplicación web, normalmente es necesario realizar algunas tareas:

- Seleccionar un elemento HTML y recuperar o actualizar su valor.
- Consulta de una API web para obtener datos.
- Enviar un comando a una API web (y responder a una devolución de llamada con su resultado).

- Realización de la validación.

Puede realizar todas estas tareas solo con JavaScript, pero existen muchas bibliotecas para facilitar estas tareas. Una de las primeras y más exitosas de estas bibliotecas es jQuery, que sigue siendo una opción popular para simplificar estas tareas en páginas web. En el caso de las aplicaciones de página única (SPA), jQuery no proporciona muchas de las características deseadas que ofrece Angular y React.

Aplicaciones web antiguas con jQuery

A pesar de ser antiguo según los estándares de los frameworks de JavaScript, jQuery sigue siendo una biblioteca ampliamente utilizada para trabajar con HTML/CSS y desarrollar aplicaciones que realizan llamadas AJAX a las API web. Sin embargo, jQuery funciona en el nivel del modelo de objetos de documento del explorador (DOM) y, de forma predeterminada, solo ofrece un modelo imperativo, en lugar de declarativo.

Por ejemplo, imagine que si el valor de un cuadro de texto supera 10, se debe hacer visible un elemento de la página. En jQuery, esta funcionalidad normalmente se implementaría escribiendo un controlador de eventos con código que inspeccionaría el valor del cuadro de texto y establecería la visibilidad del elemento de destino en función de ese valor. Este proceso es un enfoque imperativo basado en código. En su lugar, otro marco podría usar databinding para enlazar la visibilidad del elemento al valor del cuadro de texto mediante declaración. Este enfoque no requeriría escribir ningún código, sino que solo requiere decorar los elementos implicados en los atributos de enlace de datos. A medida que los comportamientos del lado cliente crecen más complejos, los enfoques de enlace de datos suelen dar lugar a soluciones más sencillas con menos código y complejidad condicional.

jQuery frente a un marco de SPA

[🔗 Expandir tabla](#)

Factor	jQuery	Angular
Abstrae el DOM	Sí	Sí
Compatibilidad con AJAX	Sí	Sí
Enlace de datos declarativo	No	Sí
Enrutamiento de estilo MVC	No	Sí
Plantillas	No	Sí
Enrutamiento de vínculo profundo	No	Sí

La mayoría de las características que jQuery carece intrínsecamente se puede agregar con la adición de otras bibliotecas. Sin embargo, un marco spa como Angular proporciona estas características de forma más integrada, ya que se ha diseñado con todos ellos en mente desde el principio. Además, jQuery es una biblioteca imperativa, lo que significa que debe llamar a funciones de jQuery para poder hacer cualquier cosa con jQuery. Gran parte del trabajo y la funcionalidad que proporcionan los marcos de SPA se pueden realizar mediante declaración, lo que requiere que no se escriba código real.

El enlace de datos es un excelente ejemplo de esta funcionalidad. En jQuery, normalmente solo toma una línea de código para obtener el valor de un elemento DOM o para establecer el valor de un elemento. Sin embargo, debe escribir este código en cualquier momento que necesite cambiar el valor del elemento y, a veces, esto se producirá en varias funciones de una página. Otro ejemplo común es la visibilidad de elementos. En jQuery, puede haber muchos lugares diferentes en los que escribiría código para controlar si determinados elementos estaban visibles. En cada uno de estos casos, al usar el enlace de datos, no es necesario escribir código. Simplemente enlazaría el valor o la visibilidad de los elementos en cuestión a un *modelo de vista* de la página y los cambios en ese modelo de vista se reflejarían automáticamente en los elementos enlazados.

SPA de Angular

Angular sigue siendo uno de los marcos de JavaScript más populares del mundo. Desde Angular 2, el equipo recompiló el marco desde cero (con [TypeScript](#)) y cambió el nombre del nombre original de AngularJS a Angular. Ahora, tras varios años desde su rediseño, Angular sigue siendo un sólido marco para el desarrollo de aplicaciones de una sola página.

Las aplicaciones angulares se compilan a partir de componentes. Los componentes combinan plantillas HTML con objetos especiales y controlan una parte de la página. Aquí se muestra un componente sencillo de los documentos de Angular:

JavaScript

```
import { Component } from '@angular/core';

@Component({
  selector: 'my-app',
  template: `<h1>Hello {{name}}</h1>`
})

export class AppComponent { name = 'Angular'; }
```

Los componentes se definen mediante la `@Component` función de decorador, que toma metadatos sobre el componente. La propiedad `selector` identifica el identificador del elemento

en la página donde se mostrará este componente. La propiedad `template` es una plantilla HTML sencilla que incluye un marcador de posición que corresponde a la propiedad `name` del componente, definida en la última línea.

Al trabajar con componentes y plantillas, en lugar de elementos DOM, las aplicaciones de Angular pueden funcionar en un nivel superior de abstracción y con menos código general que las aplicaciones escritas con solo JavaScript (también denominado "vanilla JS") o con jQuery. Angular también impone algún orden sobre cómo organizar los archivos de script del lado cliente. Por convención, las aplicaciones de Angular usan una estructura de carpetas común, con archivos de script de módulos y componentes ubicados en una carpeta de la aplicación. Los scripts angulares relacionados con la compilación, implementación y prueba de la aplicación se encuentran normalmente en una carpeta de nivel superior.

Puede desarrollar aplicaciones de Angular mediante una CLI. Introducción al desarrollo de Angular localmente (suponiendo que ya tiene git y npm instalado) consiste en simplemente clonar un repositorio desde GitHub y ejecutar `npm install` y `npm start`. Además de esto, Angular incluye su propia CLI, que puede crear proyectos, agregar archivos y ayudar con las tareas de prueba, agrupación e implementación. Esta facilidad de uso de la CLI hace que Angular sea especialmente compatible con ASP.NET Core, que también ofrece una gran compatibilidad con la CLI.

Microsoft ha desarrollado una aplicación de referencia, `eShopOnContainers`, que incluye una implementación de SPA de Angular. Esta aplicación incluye módulos de Angular para administrar la cesta de la compra de la tienda en línea, cargar y mostrar elementos desde su catálogo y controlar la creación de pedidos. Puede ver y descargar la aplicación de ejemplo desde [GitHub](#).

Reaccionar

A diferencia de Angular, que ofrece una implementación completa del modelo View-Controller, React solo se preocupa por las vistas. No es un marco, solo una biblioteca, por lo que para crear una SPA, deberá aprovechar bibliotecas adicionales. Hay una serie de bibliotecas diseñadas para usarse con React para generar aplicaciones enriquecidas de página única.

Una de las características más importantes de React es su uso de un DOM virtual. El DOM virtual proporciona a React varias ventajas, incluido el rendimiento (el DOM virtual puede optimizar qué partes del DOM real deben actualizarse) y la capacidad de prueba (no es necesario tener un explorador para probar React y sus interacciones con su DOM virtual).

React también es inusual en cómo funciona con HTML. En lugar de tener una separación estricta entre código y marcado (con referencias a JavaScript que aparecen en atributos HTML

quizás), React agrega HTML directamente dentro de su código JavaScript como JSX. JSX es una sintaxis similar a HTML que se puede compilar hasta JavaScript puro. Por ejemplo:

JavaScript

```
<ul>
{ authors.map(author =>
  <li key={author.id}>{author.name}</li>
)}
</ul>
```

Si ya conoce JavaScript, el aprendizaje de React debe ser fácil. No hay casi tanta curva de aprendizaje ni sintaxis especial implicada como con Angular u otras bibliotecas populares.

Dado que React no es un marco completo, normalmente querrá que otras bibliotecas controle aspectos como el enrutamiento, las llamadas api web y la administración de dependencias. Lo bueno es que puede elegir la mejor biblioteca para cada uno de estos, pero la desventaja es que necesita tomar todas estas decisiones y comprobar que todas las bibliotecas elegidas funcionan bien cuando haya terminado. Si desea un buen punto de partida, puede usar un kit de inicio como React Slingshot, que empaqueta previamente un conjunto de bibliotecas compatibles junto con React.

Vue

En su guía de introducción, "Vue es un marco progresivo para crear interfaces de usuario. A diferencia de otros marcos monolíticos, Vue está diseñado desde el principio para que se pueda adoptar de forma incremental. La biblioteca principal se centra solo en la capa de vista y es fácil de seleccionar e integrar con otras bibliotecas o proyectos existentes. Por otro lado, Vue es perfectamente capaz de impulsar sofisticadas aplicaciones Single-Page cuando se usan en combinación con herramientas modernas y bibliotecas auxiliares".

La introducción a Vue simplemente requiere incluir su script dentro de un archivo HTML:

HTML

```
<!-- development version, includes helpful console warnings -->
<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
```

Con el marco agregado, podrá representar datos declarativamente en el DOM mediante la sintaxis de plantillas sencilla de Vue:

HTML

```
<div id="app">
  {{ message }}
</div>
```

y, a continuación, agregue el siguiente script:

JavaScript

```
var app = new Vue({
  el: '#app',
  data: {
    message: 'Hello Vue!'
  }
})
```

Esto es suficiente para representar "Hello Vue!" en la página. Sin embargo, tenga en cuenta que Vue no representa simplemente el mensaje incluido en el elemento "div" una vez. Admite el enlace de datos y las actualizaciones dinámicas de manera que si el valor de `message` cambia, el valor de `<div>` se actualiza inmediatamente para reflejarlo.

Por supuesto, esto apenas revela la esencia de lo que Vue es capaz de. Ha ganado una gran popularidad en los últimos años y tiene una gran comunidad. Hay una [enorme y creciente lista de componentes y bibliotecas auxiliares](#) que funcionan con Vue para ampliarlo también. Si desea agregar el comportamiento del lado cliente a la aplicación web o considerar la posibilidad de crear una SPA completa, Vue merece la pena investigar.

Blazor WebAssembly

A diferencia de otros marcos de JavaScript, `Blazor WebAssembly` es un marco de aplicación de página única (SPA) para crear aplicaciones web interactivas del lado cliente con .NET. Blazor WebAssembly usa estándares web abiertos sin complementos ni volver a compilar código en otros lenguajes. Blazor WebAssembly funciona en todos los exploradores web modernos, incluidos los exploradores móviles.

La ejecución de código .NET dentro de exploradores web es posible mediante WebAssembly (abreviado `wasm`). WebAssembly es un formato de código de bytes compacto optimizado para descargas rápidas y una velocidad de ejecución máxima. WebAssembly es un estándar web abierto y se admite en exploradores web sin complementos.

El código WebAssembly puede acceder a la funcionalidad completa del explorador a través de JavaScript, denominada interoperabilidad de JavaScript, a menudo acortada a la interoperabilidad de JavaScript o a la interoperabilidad de JS. El código de .NET que se ejecuta a través de WebAssembly en el explorador se ejecuta a su vez en el espacio aislado de

JavaScript del explorador con las protecciones que proporciona dicho espacio aislado contra acciones malintencionadas en la máquina cliente.

Para obtener más información, vea [Introducción a ASP.NET Core Blazor](#).

Elección de un marco de SPA

Al valorar qué opción funcionará mejor para admitir la SPA, tenga en cuenta las consideraciones siguientes:

- ¿Su equipo está familiarizado con el marco y sus dependencias (incluido TypeScript en algunos casos)?
- ¿Hasta qué punto es opinable el marco, y usted está de acuerdo con la manera predeterminada de hacer las cosas?
- ¿Incluye (o una biblioteca complementaria) todas las características que requiere la aplicación?
- ¿Está bien documentado?
- ¿Cómo está activa su comunidad? ¿Se están construyendo nuevos proyectos con ello?
- ¿Cómo está activo su equipo principal? ¿Se resuelven los problemas y se envían periódicamente las nuevas versiones?

Los marcos siguen evolucionando con velocidad vertiginosa. Use las consideraciones enumeradas anteriormente para ayudar a mitigar el riesgo de elegir un marco en el que más adelante lamentará que dependa. Si es particularmente aversante al riesgo, considere un marco que ofrezca soporte técnico comercial o que esté desarrollando una gran empresa.

Referencias: tecnologías web cliente

- **HTML y CSS**
<https://www.w3.org/standards/webdesign/htmlcss> ↗
- **Sass frente a LESS**
<https://www.keycdn.com/blog/sass-vs-less/> ↗
- **Estilización de aplicaciones ASP.NET Core con LESS, Sass y Font Awesome**
<https://learn.microsoft.com/aspnet/core/client-side/less-sass-fa>
- **desarrollo deClient-Side en ASP.NET Core**
<https://learn.microsoft.com/aspnet/core/client-side/>
- **jQuery**
<https://jquery.com/> ↗

- **Angular**
<https://angular.io/> ↗
- **Reaccionar**
<https://reactjs.org/> ↗
- **Vue**
<https://vuejs.org/> ↗
- **Angular frente a React frente a Vue: qué marco elegir en 2020**
<https://www.codeinwp.com/blog/angular-vs-vue-vs-react/> ↗
- **Los principales marcos de JavaScript para el desarrollo de Front-End en 2020**
<https://www.freecodecamp.org/news/complete-guide-for-front-end-developers-javascript-frameworks-2019/> ↗

Siguiente anterior

Desarrollo de aplicaciones ASP.NET Core MVC

18/06/2025

💡 Sugerencia

Este contenido es un extracto del libro electrónico, Arquitecto de aplicaciones web modernas con ASP.NET Core y Azure, disponible en [.NET Docs](#) o como un PDF descargable gratuito que se puede leer sin conexión.

Descargar PDF



"No es importante hacerlo bien la primera vez. Es vital hacerlo bien por última vez". -
Andrew Hunt y David Thomas

ASP.NET Core es un marco de código abierto multiplataforma para crear aplicaciones web modernas optimizadas para la nube. ASP.NET Las aplicaciones Core son ligeras y modulares, con compatibilidad integrada para la inserción de dependencias, lo que permite una mayor capacidad de prueba y mantenimiento. Combinado con MVC, que admite la creación de API web modernas además de aplicaciones basadas en vistas, ASP.NET Core es un marco eficaz con el que crear aplicaciones web empresariales.

MVC y Razor Pages

ASP.NET Core MVC ofrece muchas características que son útiles para crear aplicaciones y API basadas en web. El término MVC significa "Model-View-Controller", un patrón de interfaz de usuario que divide las responsabilidades de responder a las solicitudes del usuario en varias partes. Además de seguir este patrón, también puede implementar características en las aplicaciones de ASP.NET Core como Razor Pages.

Las páginas de Razor están integradas en ASP.NET Core MVC y usan las mismas características para el enrutamiento, el enlace de modelos, los filtros, la autorización, etc. Sin embargo, en lugar de tener carpetas y archivos independientes para Controladores, Modelos, Vistas, etc. y mediante el enrutamiento basado en atributos, Razor Pages se coloca en una sola carpeta ("/Pages"), ruta basada en su ubicación relativa en esta carpeta y controla las solicitudes con controladores en lugar de acciones de controlador. Como resultado, al trabajar con Razor Pages, todos los archivos y clases que necesita se colocan normalmente, no se distribuyen por todo el proyecto web.

Obtenga más información sobre [cómo se aplican MVC, Razor Pages y patrones relacionados en la aplicación de ejemplo eShopOnWeb](#) [↗](#).

Al crear una nueva aplicación ASP.NET Core, debe tener un plan en mente para el tipo de aplicación que desea desarrollar. Al crear un nuevo proyecto, en el IDE o mediante el comando de la `dotnet new` CLI, elegirá entre varias plantillas. Las plantillas de proyecto más comunes son Empty, Web API, Web App y Web App (Model-View-Controller). Aunque solo puede tomar esta decisión cuando cree por primera vez un proyecto, no es una decisión irreversible. El proyecto de API web usa controladores estándar de modeloView-Controller: solo carece de vistas de forma predeterminada. Del mismo modo, la plantilla de aplicación web predeterminada usa Razor Pages, por lo que también carece de una carpeta Views. Puede agregar una carpeta Views a estos proyectos más adelante para admitir el comportamiento basado en vistas. Los proyectos de API web y ModeloView-Controller no incluyen una carpeta Páginas de forma predeterminada, pero puede agregar una más adelante para admitir el comportamiento basado en Razor Pages. Puede considerar estas tres plantillas como compatibles con tres tipos diferentes de interacción de usuario predeterminada: datos (API web), basados en páginas y basados en vistas. Sin embargo, puede mezclar y hacer coincidir cualquiera o todas estas plantillas dentro de un solo proyecto si lo desea.

¿Por qué Razor Pages?

Razor Pages es el enfoque predeterminado para las nuevas aplicaciones web en Visual Studio. Razor Pages ofrece una manera más sencilla de crear características de aplicación basadas en páginas, como formularios que no son SPA. Con controladores y vistas, era habitual que las aplicaciones tuvieran controladores muy grandes que funcionaban con muchas dependencias y modelos de vista diferentes y devolvieron muchas vistas diferentes. Esto dio lugar a una mayor complejidad y, a menudo, a los controladores que no seguían el principio de responsabilidad única o los principios abiertos y cerrados de forma eficaz. Razor Pages soluciona este problema mediante la encapsulación de la lógica del lado servidor para una "página lógica" específica en una aplicación web mediante su marcado de Razor. Una página de Razor que no tiene lógica del lado servidor solo puede constar de un archivo de Razor (por ejemplo, "Index.cshtml"). Sin embargo, la mayoría de las páginas de Razor no triviales tendrán una clase de modelo de

página asociada, que por convención se denomina igual que el archivo de Razor con una extensión ".cs" (por ejemplo, "Index.cshtml.cs").

Un modelo de página de Razor Page combina las responsabilidades de un controlador MVC y un modelo de vista. En lugar de controlar las solicitudes con métodos de acción del controlador, se ejecutan controladores de modelo de página como "OnGet()", que representan su página asociada de forma predeterminada. Razor Pages simplifica el proceso de creación de páginas individuales en una aplicación ASP.NET Core, a la vez que proporciona todas las características arquitectónicas de ASP.NET Core MVC. Son una buena opción predeterminada para la nueva funcionalidad basada en páginas.

Cuándo usar MVC

Si estás desarrollando APIs web, el patrón MVC tiene más sentido que intentar usar Razor Pages. Si el proyecto solo expondrá puntos de conexión de API web, lo ideal es empezar desde la plantilla de proyecto de API web. De lo contrario, es fácil agregar controladores y puntos de conexión de API asociados a cualquier aplicación de ASP.NET Core. Use el enfoque de MVC basado en vistas si va a migrar una aplicación existente de ASP.NET MVC 5 o anterior a ASP.NET Core MVC y desea hacerlo con la menor cantidad de esfuerzo. Una vez que haya realizado la migración inicial, puede evaluar si tiene sentido adoptar Razor Pages para nuevas características o incluso como una migración mayorista. Para obtener más información sobre cómo migrar aplicaciones de .NET 4.x a .NET 8, consulte [Migración de aplicaciones de ASP.NET existentes a ASP.NET Core eBook](#).

Tanto si decide compilar la aplicación web mediante páginas de Razor o vistas de MVC, la aplicación tendrá un rendimiento similar e incluirá compatibilidad con la inserción de dependencias, filtros, enlace de modelos, validación, etc.

Asignación de solicitudes a respuestas

En su corazón, las aplicaciones ASP.NET Core asignan las solicitudes entrantes a las respuestas salientes. A nivel básico, este mapeo se realiza con middleware, y las aplicaciones simples de ASP.NET Core y los microservicios pueden estar compuestos únicamente de middleware específico. Al usar ASP.NET Core MVC, puede trabajar en un nivel algo más alto, pensando en las *rutas*, *controladores* y *acciones*. Cada solicitud entrante se compara con la tabla de enrutamiento de la aplicación y, si se encuentra una ruta coincidente, se llama al método de acción asociado (que pertenece a un controlador) para controlar la solicitud. Si no se encuentra ninguna ruta que coincida, se llama a un controlador de errores (en este caso, se devuelve un resultado de NotFound).

ASP.NET aplicaciones de Core MVC pueden usar rutas convencionales, rutas de atributo o ambas. Las rutas convencionales se definen en el código, especificando convenciones de enrutamiento mediante *sintaxis* como en el ejemplo siguiente:

C#

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(name: "default", pattern: "
{controller=Home}/{action=Index}/{id?}");
});
```

En este ejemplo, se ha agregado una ruta denominada "default" a la tabla de enrutamiento. Define una plantilla de ruta con marcadores de posición para `controller`, `action` y `id`. Los marcadores de posición `controller` y `action` tienen el valor predeterminado especificado (`Home` y `Index`, respectivamente), y el marcador de posición `id` es opcional por la aplicación de un "?" a este. La convención definida aquí indica que la primera parte de una solicitud debe corresponder al nombre del controlador, la segunda parte a la acción y, si es necesario, una tercera parte representará un parámetro ID. Las rutas convencionales normalmente se definen en un solo lugar para la aplicación, como en *Program.cs* donde se configura la canalización de middleware de solicitud.

Las rutas de atributo se aplican directamente a controladores y acciones, en lugar de especificarse globalmente. Este enfoque tiene la ventaja de hacerlas mucho más reconocibles cuando se examina un método determinado, pero significa que la información de enrutamiento no se mantiene en un solo lugar de la aplicación. Con las rutas de atributo, puede especificar fácilmente varias rutas para una acción determinada, así como combinar rutas entre controladores y acciones. Por ejemplo:

C#

```
[Route("Home")]
public class HomeController : Controller
{
    [Route("")] // Combines to define the route template "Home"
    [Route("Index")] // Combines to define route template "Home/Index"
    [Route("/")] // Does not combine, defines the route template ""
    public IActionResult Index() {}
}
```

Las rutas se pueden especificar en `[HttpGet]` y atributos similares, lo que evita la necesidad de agregar atributos `[Route]` independientes. Las rutas de atributo también pueden usar tokens para reducir la necesidad de repetir nombres de controlador o acción, como se muestra a continuación:

C#

```
[Route("[controller]")]
public class ProductsController : Controller
{
    [Route("")] // Matches 'Products'
    [Route("Index")] // Matches 'Products/Index'
    public IActionResult Index() {}
}
```

Razor Pages no usa el enrutamiento de atributos. Puede especificar información de la plantilla de ruta adicional para una Razor Page como parte de su directiva de `@page`:

C#

```
@page "{id:int}"
```

En el ejemplo anterior, la página en cuestión coincidiría con una ruta con un parámetro entero `id`. Por ejemplo, la página *Products.cshtml* ubicada en la raíz de `/Pages` respondería a solicitudes como esta:

HTTP

```
/Products/123
```

Una vez que una solicitud determinada se ha combinado con una ruta, pero antes de llamar al método de acción, ASP.NET Core MVC realizará el [enlace de modelos](#) y la [validación de modelos](#) en la solicitud. El enlace de modelos es responsable de convertir los datos HTTP entrantes en los tipos de .NET especificados como parámetros del método de acción al que se va a llamar. Por ejemplo, si el método de acción espera un `int id` parámetro, el enlace de modelos intentará proporcionar este parámetro a partir de un valor proporcionado como parte de la solicitud. Para ello, la vinculación de modelos busca valores en un formulario publicado, valores en la propia ruta y valores en la cadena de consulta. Suponiendo que se encuentra un `id` valor, se convertirá en un entero antes de pasarse al método de acción.

Después de enlazar el modelo, pero antes de llamar al método de acción, se produce la validación del modelo. La validación de modelos usa atributos opcionales en el tipo de modelo y puede ayudar a garantizar que el objeto de modelo proporcionado se ajusta a determinados requisitos de datos. Algunos valores pueden especificarse según sea necesario o limitarse a una longitud determinada o un intervalo numérico, etc. Si se especifican atributos de validación pero el modelo no cumple sus requisitos, la propiedad `ModelState.IsValid` será `false` y el conjunto de reglas de validación con errores estará disponible para enviar al cliente que realiza la solicitud.

Si usa la validación del modelo, debe asegurarse de comprobar siempre que el modelo es válido antes de realizar cualquier comando de modificación de estado, para asegurarse de que la aplicación no está dañada por datos no válidos. Puede usar un [filtro](#) para evitar la necesidad de agregar código para esta validación en cada acción. Filtros de ASP.NET Core MVC ofrecen una manera de interceptar grupos de solicitudes, de modo que las directivas comunes y las preocupaciones transversales se puedan aplicar en un modo específico. Los filtros se pueden aplicar a acciones individuales, controladores completos o globalmente para una aplicación.

En el caso de las API web, ASP.NET Core MVC admite [la negociación de contenido](#), lo que permite a las solicitudes especificar cómo se deben dar formato a las respuestas. En función de los encabezados proporcionados en la solicitud, las acciones que devuelven datos darán formato a la respuesta en XML, JSON u otro formato compatible. Esta característica permite que varios clientes usen la misma API con distintos requisitos de formato de datos.

Los proyectos de API web deben considerar el uso del `[ApiController]` atributo, que se puede aplicar a controladores individuales, a una clase de controlador base o a todo el ensamblado. Este atributo agrega la comprobación automática de validación de modelos y cualquier acción con un modelo no válido devolverá un `BadRequest` con los detalles de los errores de validación. El atributo también requiere que todas las acciones tengan una ruta de atributo, en lugar de usar una ruta convencional, y devuelva información de `ProblemDetails` más detallada en respuesta a errores.

Mantener los controladores bajo control

En el caso de las aplicaciones basadas en páginas, Razor Pages realiza un gran trabajo para evitar que los controladores sean demasiado grandes. Cada página individual recibe sus propios archivos y clases dedicados solo a sus controladores. Antes de la introducción de Razor Pages, muchas aplicaciones centradas en vistas tendrían clases de controlador grandes responsables de muchas acciones y vistas diferentes. Estas clases crecerían naturalmente para tener muchas responsabilidades y dependencias, lo que hace que sean más difíciles de mantener. Si sus controladores basados en vistas están creciendo demasiado, considere la posibilidad de refactorizarlos para usar Razor Pages o de incluir un patrón como mediador.

El patrón de diseño de mediador se usa para reducir el acoplamiento entre clases, al tiempo que permite la comunicación entre ellas. En las aplicaciones ASP.NET Core MVC, este patrón se emplea con frecuencia para dividir los controladores en partes más pequeñas mediante el uso de *manejadores* para realizar el trabajo de los métodos de acción. El [popular paquete NuGet `mediatR`](#) se usa a menudo para lograr esto. Normalmente, los controladores incluyen muchos métodos de acción diferentes, cada uno de los cuales puede requerir determinadas dependencias. El conjunto de todas las dependencias requeridas por cualquier acción debe pasarse al constructor del controlador. Al usar MediatR, la única dependencia que tendrá un

controlador suele ser una instancia del mediador. A continuación, cada acción usa la instancia de mediador para enviar un mensaje, que un controlador procesa. El controlador es específico de una sola acción y, por tanto, solo necesita las dependencias requeridas por esa acción. Aquí se muestra un ejemplo de un controlador mediante Mediatr:

C#

```
public class OrderController : Controller
{
    private readonly IMediator _mediator;

    public OrderController(IMediator mediator)
    {
        _mediator = mediator;
    }

    [HttpGet]
    public async Task<IActionResult> MyOrders()
    {
        var viewModel = await _mediator.Send(new GetMyOrders(User.Identity.Name));
        return View(viewModel);
    }
    // other actions implemented similarly
}
```

En la `MyOrders` acción, esta clase controla la llamada a `Send` un `GetMyOrders` mensaje:

C#

```
public class GetMyOrdersHandler : IRequestHandler<GetMyOrders,
IEnumerable<OrderViewModel>>
{
    private readonly IOrderRepository _orderRepository;
    public GetMyOrdersHandler(IOrderRepository orderRepository)
    {
        _orderRepository = orderRepository;
    }

    public async Task<IEnumerable<OrderViewModel>> Handle(GetMyOrders request,
CancellationTokentoken cancellationTokentoken)
    {
        var specification = new
CustomerOrdersWithItemsSpecification(request.UserName);
        var orders = await _orderRepository.ListAsync(specification);
        return orders.Select(o => new OrderViewModel
        {
            OrderDate = o.OrderDate,
            OrderItems = o.OrderItems?.Select(oi => new OrderItemViewModel()
            {
                PictureUrl = oi.ItemOrdered.PictureUri,
                ProductId = oi.ItemOrdered.CatalogItemId,
```

```

        ProductName = oi.ItemOrdered.ProductName,
        UnitPrice = oi.UnitPrice,
        Units = oi.Units
    }).ToList(),
    OrderNumber = o.Id,
    ShippingAddress = o.ShipToAddress,
    Total = o.Total()
});
}
}

```

El resultado final de este enfoque es que los controladores sean mucho más pequeños y se centren principalmente en el enrutamiento y el enlace del modelo, mientras que los controladores individuales son responsables de las tareas específicas que necesita un punto de conexión determinado. Este enfoque también se puede lograr sin MediatR mediante el [paquete NuGet ApiEndpoints](#), que intenta traer a los controladores de API las mismas ventajas que Razor Pages aporta a los controladores basados en vistas.

Referencias: asignación de solicitudes a respuestas

- Enrutamiento a acciones del controlador
<https://learn.microsoft.com/aspnet/core/mvc/controllers/routing>
- Vinculación de Modelos
<https://learn.microsoft.com/aspnet/core/mvc/models/model-binding>
- Validación de modelos
<https://learn.microsoft.com/aspnet/core/mvc/models/validation>
- Filtros
<https://learn.microsoft.com/aspnet/core/mvc/controllers/filters>
- Atributo ApiController
<https://learn.microsoft.com/aspnet/core/web-api/>

Trabajar con dependencias

ASP.NET Core tiene compatibilidad integrada con e internamente hace uso de una técnica conocida como [inserción de dependencias](#). La inserción de dependencias es una técnica que permite el acoplamiento flexible entre los distintos elementos de una aplicación. El acoplamiento flexible es deseable porque facilita el aislamiento de partes de la aplicación, lo que permite realizar pruebas o reemplazos. También hace menos probable que un cambio en una parte de la aplicación tenga un impacto inesperado en otro lugar de la aplicación. La inserción de dependencias se basa en el principio de inversión de dependencias y suele ser clave para lograr el principio abierto y cerrado. Al evaluar cómo funciona la aplicación con sus

dependencias, tenga cuidado con el olor a código rígido y recuerde el aforismo "el nuevo es pegamento".

La estática se produce cuando las clases realizan llamadas a métodos estáticos, o bien tienen acceso a propiedades estáticas, que tienen efectos secundarios o dependencias en la infraestructura. Por ejemplo, si tiene un método que llama a un método estático, que a su vez escribe en una base de datos, el método está estrechamente acoplado a la base de datos. Todo lo que interrumpa esa llamada de base de datos interrumpirá el método. Probar estos métodos es notoriamente difícil, ya que estas pruebas requieren bibliotecas de simulación comercial para simular las llamadas estáticas, o solo se pueden probar con una base de datos de prueba en su lugar. Las llamadas estáticas que no tienen ninguna dependencia de la infraestructura, especialmente aquellas que son completamente sin estado, son adecuadas para llamarse y no tienen ningún impacto en el acoplamiento o la capacidad de prueba (más allá del acoplamiento del código a la propia llamada estática).

Muchos desarrolladores entienden los riesgos del acoplamiento estático y el estado global. A pesar de ello, seguirán acoplando estrechamente su código a implementaciones específicas mediante la instanciación directa. "Lo nuevo se pega" está pensado para ser un aviso de este acoplamiento y no un rechazo general del uso de la palabra clave `new`. Al igual que con las llamadas a métodos estáticos, las nuevas instancias de tipos que no tienen dependencias externas normalmente no acoplan estrechamente el código a los detalles de implementación o hacen que las pruebas sean más difíciles. Pero cada vez que se crea una instancia de una clase, dedique un breve momento a tener en cuenta si tiene sentido codificar esa instancia específica en esa ubicación concreta o si sería un mejor diseño solicitar esa instancia como dependencia.

Declaración de las dependencias

ASP.NET Core se basa en la creación de métodos y clases que declaran sus dependencias, solicitándolas como argumentos. Las aplicaciones de ASP.NET generalmente se configuran en *Program.cs* o en una clase `Startup`.

ⓘ Nota

La configuración de aplicaciones completamente en *Program.cs* es el enfoque predeterminado para las aplicaciones de .NET 6 (y posteriores) y Visual Studio 2022. Las plantillas de proyecto se han actualizado para ayudarle a empezar a trabajar con este nuevo enfoque. Los proyectos de ASP.NET Core aún pueden usar una clase `Startup`, si se desea.

Configuración de servicios en *Program.cs*

Para aplicaciones muy sencillas, puede conectar las dependencias directamente en el archivo *Program.cs* usando un `WebApplicationBuilder`. Una vez agregados todos los servicios necesarios, el generador se usa para crear la aplicación.

C#

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddRazorPages();

var app = builder.Build();
```

Configuración de servicios en *Startup.cs*

La *Startup.cs* está configurada para admitir la inserción de dependencias en varios puntos. Si usa una clase `Startup`, puede darle un constructor y puede pedir dependencias a través de ella, de la siguiente manera:

C#

```
public class Startup
{
    public Startup(IHostingEnvironment env)
    {
        var builder = new ConfigurationBuilder()
            .SetBasePath(env.ContentRootPath)
            .AddJsonFile("appsettings.json", optional: false, reloadOnChange:
true)
            .AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional:
true);
    }
}
```

La `Startup` clase es interesante en que no hay requisitos de tipo explícitos para ella. No hereda de una clase base especial `Startup` ni implementa ninguna interfaz determinada. Puede darle un constructor, o no, y puede especificar tantos parámetros en el constructor como desee. Cuando se inicie el host web que ha configurado para la aplicación, llamará a la `Startup` clase (si le ha dicho que use una) y usará la inserción de dependencias para rellenar las dependencias que requiere la `Startup` clase. Por supuesto, si solicita parámetros que no están configurados en el contenedor de servicios usado por ASP.NET Core, obtendrá una excepción, pero siempre que se ajuste a las dependencias que conoce el contenedor, puede solicitar todo lo que desee.

La inserción de dependencias se integra en las aplicaciones de ASP.NET Core directamente desde el principio, al crear la instancia de inicio. La clase Startup no termina aquí. También puede solicitar dependencias en el `Configure` método :

```
C#  
  
public void Configure(IApplicationBuilder app,  
    IHostingEnvironment env,  
    ILoggerFactory loggerFactory)  
{  
  
}
```

El método `ConfigureServices` es la excepción a este comportamiento; debe tomar solo un parámetro de tipo `IServiceCollection`. Realmente no es necesario admitir la inserción de dependencias, ya que, por un lado, es responsable de agregar objetos al contenedor de servicios y, por otro, tiene acceso a todos los servicios configurados actualmente a través del `IServiceCollection` parámetro. Por lo tanto, puede trabajar con dependencias definidas en la colección de servicios principales de ASP.NET en cada parte de la `Startup` clase, ya sea solicitando el servicio necesario como parámetro o trabajando con `IServiceCollection` en `ConfigureServices`.

⚠ Nota

Si necesita asegurarse de que determinados servicios están disponibles para su `Startup` clase, puede configurarlos mediante un `IWebHostBuilder` y su `ConfigureServices` método dentro de la `CreateDefaultBuilder` llamada.

La clase `Startup` es un modelo para cómo debe estructurar otras partes de la aplicación ASP.NET Core, desde Controladores a Middleware a Filtros a sus propios servicios. En cada caso, debe seguir el [principio de dependencias explícitas](#), solicitar las dependencias en lugar de crearlas directamente y aprovechar la inserción de dependencias en toda la aplicación. Tenga cuidado de dónde y cómo crea instancias directas de implementaciones, especialmente servicios y objetos que funcionan con infraestructura o tienen efectos secundarios. Prefiere trabajar con abstracciones definidas en el núcleo de tu aplicación y pásalas como argumentos, en lugar de codificar rígidamente referencias a tipos de implementación específicos.

Estructuración de la aplicación

Normalmente, las aplicaciones monolíticas tienen un único punto de entrada. En el caso de una aplicación web ASP.NET Core, el punto de entrada será el proyecto web ASP.NET Core. Sin

embargo, eso no significa que la solución debe constar de un solo proyecto. Resulta útil dividir la aplicación en diferentes capas con el fin de seguir la separación de problemas. Una vez divididas en capas, resulta útil ir más allá de carpetas para separar proyectos, lo que puede ayudar a lograr una mejor encapsulación. El mejor enfoque para lograr estos objetivos con una aplicación ASP.NET Core es una variación de la arquitectura limpia que se describe en el capítulo 5. Después de este enfoque, la solución de la aplicación incluirá bibliotecas independientes para la interfaz de usuario, la infraestructura y `ApplicationCore`.

Además de estos proyectos, también se incluyen proyectos de prueba independientes (las pruebas se tratan en el capítulo 9).

El modelo de objetos y las interfaces de la aplicación deben colocarse en el proyecto `ApplicationCore`. Este proyecto tendrá tan pocas dependencias como sea posible (y ninguna en problemas de infraestructura específicos), y los demás proyectos de la solución harán referencia a él. Las entidades empresariales que deben conservarse se definen en el proyecto `ApplicationCore`, ya que son servicios que no dependen directamente de la infraestructura.

Detalles de implementación, como cómo se realiza la persistencia o cómo se pueden enviar notificaciones a un usuario, se conservan en el proyecto de infraestructura. Este proyecto hará referencia a paquetes específicos de implementación, como `Entity Framework Core`, pero no debe exponer detalles sobre estas implementaciones fuera del proyecto. Los servicios y repositorios de infraestructura deben implementar interfaces definidas en el proyecto `ApplicationCore` y sus implementaciones de persistencia son responsables de recuperar y almacenar entidades definidas en `ApplicationCore`.

El proyecto de interfaz de usuario principal de ASP.NET es responsable de cualquier problema de nivel de interfaz de usuario, pero no debe incluir la lógica de negocios ni los detalles de la infraestructura. De hecho, lo ideal es que ni siquiera tenga una dependencia en el proyecto de infraestructura, lo que ayudará a garantizar que no se introduzca ninguna dependencia entre los dos proyectos accidentalmente. Esto se puede lograr mediante un contenedor de DI de terceros, como `Autofac`, que permite definir reglas de DI en clases del módulo en cada proyecto.

Otro enfoque para desacoplar la aplicación de los detalles de implementación es tener los microservicios de llamada a la aplicación, quizás implementados en contenedores individuales de `Docker`. Esto proporciona una separación de intereses y desacoplamiento incluso mayor que aprovechar la DI entre dos proyectos, pero tiene una complejidad adicional.

Organización de funcionalidades

De forma predeterminada, las aplicaciones ASP.NET Core organizan su estructura de carpetas para que incluya controladores y vistas, y con frecuencia `ViewModels`. El código del lado cliente

para admitir estas estructuras del lado servidor se almacena normalmente por separado en la carpeta `wwwroot`. Sin embargo, las aplicaciones grandes pueden encontrar problemas con esta organización, ya que trabajar en cualquier característica determinada a menudo requiere saltar entre estas carpetas. Esto resulta más y más difícil a medida que crece el número de archivos y subcarpetas de cada carpeta, lo que da lugar a una gran cantidad de desplazamiento por el Explorador de soluciones. Una solución a este problema es organizar el código de aplicación por *característica* en lugar de por tipo de archivo. Este estilo organizativo se conoce normalmente como carpetas de características o [segmentos de características](#) (consulte también: [Segmentos verticales](#) [↗](#)).

ASP.NET Core MVC admite áreas para este propósito. Usando áreas, puede crear conjuntos independientes de carpetas de Controladores y Vistas (así como cualquier modelo asociado) en cada carpeta Área. En la figura 7-1 se muestra una estructura de carpetas de ejemplo, mediante Áreas.

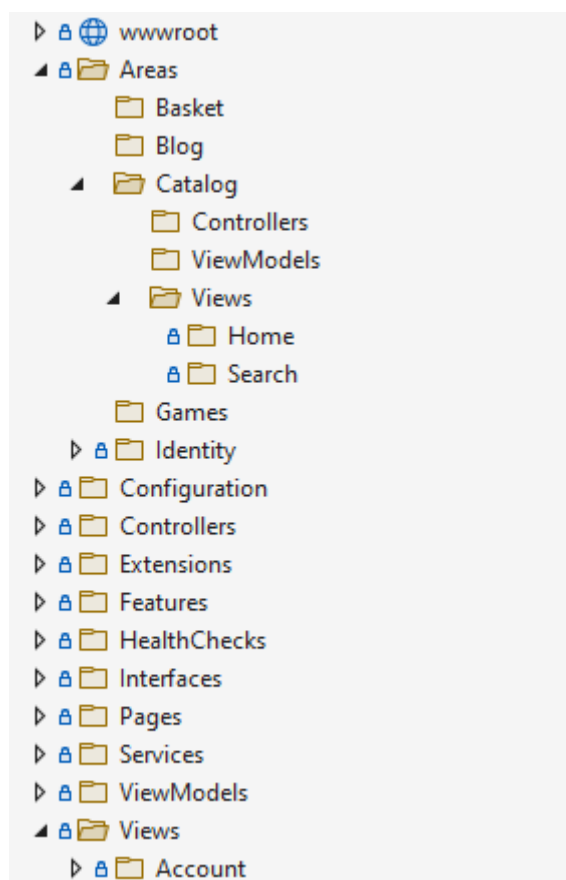


Figura 7-1. Organización de áreas de ejemplo

Al usar Áreas, debe usar atributos para decorar los controladores con el nombre del área a la que pertenecen:

C#

```
[Area("Catalog")]  
public class HomeController
```

```
{}
```

También se debe agregar compatibilidad con las áreas a las rutas:

C#

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(name: "areaRoute", pattern: "
{area:exists}/{controller=Home}/{action=Index}/{id?}");
    endpoints.MapControllerRoute(name: "default", pattern: "
{controller=Home}/{action=Index}/{id?}");
});
```

Además de la compatibilidad integrada con áreas, también puede usar su propia estructura de carpetas y convenciones en lugar de atributos y rutas personalizadas. Esto le permitiría tener carpetas de características que no incluyeran carpetas independientes para vistas, controladores, etc., mantener la jerarquía más plana y facilitar la visualización de todos los archivos relacionados en un solo lugar para cada característica. En el caso de las API, las carpetas se pueden usar para reemplazar controladores y cada carpeta puede contener todos los puntos de conexión de API y sus DTO asociados.

ASP.NET Core usa tipos de convención integrados para controlar su comportamiento. Puede modificar o reemplazar estas convenciones. Por ejemplo, puede crear una convención que obtendrá automáticamente el nombre de la característica de un controlador determinado en función de su espacio de nombres (que normalmente se correlaciona con la carpeta en la que se encuentra el controlador):

C#

```
public class FeatureConvention : IControllerModelConvention
{
    public void Apply(ControllerModel controller)
    {
        controller.Properties.Add("feature",
            GetFeatureName(controller.ControllerType));
    }

    private string GetFeatureName(TypeInfo controllerType)
    {
        string[] tokens = controllerType.FullName.Split('.');
        if (!tokens.Any(t => t == "Features")) return "";
        string featureName = tokens
            .SkipWhile(t => !t.Equals("features",
                StringComparison.CurrentCultureIgnoreCase))
            .Skip(1)
            .Take(1)
            .FirstOrDefault();
    }
}
```

```
        return featureName;
    }
}
```

A continuación, especifique esta convención como opción al agregar compatibilidad con MVC a la aplicación en `ConfigureServices` (o en `Program.cs`):

```
C#

// ConfigureServices
services.AddMvc(o => o.Conventions.Add(new FeatureConvention()));

// Program.cs
builder.Services.AddMvc(o => o.Conventions.Add(new FeatureConvention()));
```

ASP.NET Core MVC también usa una convención para buscar vistas. Se puede reemplazar con una convención personalizada para que las vistas se ubiquen en las carpetas de características (mediante el nombre de la característica proporcionado por `FeatureConvention`, anteriormente). Puede obtener más información sobre este enfoque y descargar un ejemplo de trabajo del artículo de MSDN Magazine , [Segmentos de características para ASP.NET Core MVC](#).

API y aplicaciones Blazor

Si la aplicación incluye un conjunto de API web, que deben protegerse, estas API deben configurarse idealmente como un proyecto independiente de la aplicación View o Razor Pages. Separar las API, especialmente las API públicas, de la aplicación web del lado servidor tiene una serie de ventajas. Estas aplicaciones a menudo tendrán características únicas de implementación y carga. También es muy probable que adopten diferentes mecanismos para la seguridad, con aplicaciones estándar basadas en formularios que aprovechan la autenticación basada en cookies y las API más probables mediante la autenticación basada en tokens.

Además, las aplicaciones Blazor, ya sea que usen Blazor Server o BlazorWebAssembly, deben desarrollarse como proyectos independientes. Las aplicaciones tienen diferentes características en tiempo de ejecución, así como modelos de seguridad. Es probable que compartan tipos comunes con la aplicación web del lado servidor (o proyecto de API) y estos tipos deben definirse en un proyecto compartido común.

La adición de una BlazorWebAssembly interfaz de administración a eShopOnWeb requiere agregar varios proyectos nuevos. El propio proyecto de BlazorWebAssembly, `BlazorAdmin`; En el proyecto se define un nuevo conjunto de puntos de conexión de API públicos, usados por `BlazorAdmin` y configurados para usar la `PublicApi` autenticación basada en tokens. Ciertos

tipos compartidos utilizados por estos proyectos se conservan en un nuevo proyecto `BlazorShared`.

Podría preguntarse, ¿por qué agregar un proyecto independiente `BlazorShared` cuando ya hay un proyecto común `ApplicationCore` que podría usarse para compartir cualquier tipo requerido por ambos `PublicApi` y `BlazorAdmin`? La respuesta es que este proyecto incluye toda la lógica de negocios de la aplicación y, por tanto, es mucho mayor que lo necesario y, además, es mucho más probable que tenga que mantenerse protegido en el servidor. Recuerde que cualquier biblioteca a la que hace referencia `BlazorAdmin` se descargará en los exploradores de los usuarios cuando carguen la aplicación Blazor.

En función de si se usa el [patrón backends-For-Frontends \(BFF\)](#), es posible que las API consumidas por la BlazorWebAssembly aplicación no compartan sus tipos 100% con Blazor. En concreto, una API pública que está pensada para ser consumida por muchos clientes diferentes puede definir sus propios tipos de solicitud y resultados, en lugar de compartirlos en un proyecto compartido específico del cliente. En el ejemplo de eShopOnWeb, se supone que el proyecto `PublicApi` está, de hecho, hospedando una API pública, por lo que no todos sus tipos de solicitud y respuesta proceden del proyecto `BlazorShared`.

Intereses transversales

A medida que crecen las aplicaciones, es cada vez más importante tener en cuenta los problemas transversales para eliminar la duplicación y mantener la coherencia. Algunos ejemplos de problemas transversales en ASP.NET Aplicaciones principales son la autenticación, las reglas de validación de modelos, el almacenamiento en caché de salida y el control de errores, aunque hay muchos otros. ASP.NET [filtros](#) de Core MVC permiten ejecutar código antes o después de determinados pasos en la canalización de procesamiento de solicitudes. Por ejemplo, un filtro puede ejecutarse antes y después del enlace del modelo, antes y después de una acción, o antes y después del resultado de una acción. También puede usar un filtro de autorización para controlar el acceso al resto de la canalización. Las figuras 7-2 muestran cómo fluye la ejecución de solicitudes a través de filtros, si están configurados.

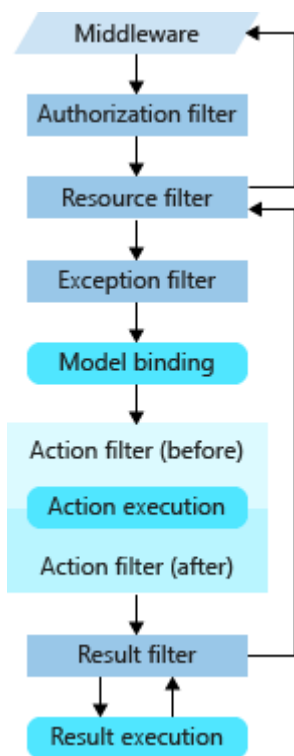


Figura 7-2. Ejecución de solicitudes a través de filtros y canalización de solicitudes.

Normalmente, los filtros se implementan como atributos, por lo que puede aplicarlos a controladores o acciones (o incluso globalmente). Cuando se agregan de esta manera, los filtros especificados en el nivel de acción anulan o se construyen sobre los filtros especificados en el nivel de controlador, que a su vez reemplazan los filtros globales. Por ejemplo, el `[Route]` atributo se puede usar para crear rutas entre controladores y acciones. Del mismo modo, la autorización se puede configurar en el nivel de controlador y, a continuación, invalidarla mediante acciones individuales, como se muestra en el ejemplo siguiente:

C#

```
[Authorize]
public class AccountController : Controller
{
    [AllowAnonymous] // overrides the Authorize attribute
    public async Task<IActionResult> Login() {}
    public async Task<IActionResult> ForgotPassword() {}
}
```

El primer método, `Login`, usa el `[AllowAnonymous]` filtro (atributo) para invalidar el conjunto de filtros `Authorize` en el nivel de controlador. La `ForgotPassword` acción (y cualquier otra acción de la clase que no tenga un atributo `AllowAnonymous`) requerirá una solicitud autenticada.

Los filtros se pueden usar para eliminar la duplicación en forma de directivas comunes de control de errores para las API. Por ejemplo, una directiva de API típica consiste en devolver una respuesta `NotFound` a las solicitudes que hacen referencia a claves que no existen y una

respuesta si se produce un `BadRequest` error en la validación del modelo. En el ejemplo siguiente se muestran estas dos directivas en acción:

C#

```
[HttpPut("{id}")]
public async Task<IActionResult> Put(int id, [FromBody]Author author)
{
    if ((await _authorRepository.ListAsync()).All(a => a.Id != id))
    {
        return NotFound(id);
    }
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }
    author.Id = id;
    await _authorRepository.UpdateAsync(author);
    return Ok();
}
```

No permita que los métodos de acción se vuelvan desordenados con código condicional como este. En su lugar, organice las directivas en filtros que se pueden aplicar según las necesidades. En este ejemplo, la comprobación de validación del modelo, que debe producirse cada vez que se envía un comando a la API, se puede reemplazar por el atributo siguiente:

C#

```
public class ValidateModelAttribute : ActionFilterAttribute
{
    public override void OnActionExecuting(ActionExecutingContext context)
    {
        if (!context.ModelState.IsValid)
        {
            context.Result = new BadRequestObjectResult(context.ModelState);
        }
    }
}
```

Puede agregar el `ValidateModelAttribute` a su proyecto como una dependencia de NuGet mediante la inclusión del paquete [Ardalis.ValidateModel](#). En el caso de las API, puede usar el `ApiController` atributo para aplicar este comportamiento sin necesidad de un filtro independiente `ValidateModel`.

Del mismo modo, se puede usar un filtro para comprobar si existe un registro y devolver un 404 antes de ejecutar la acción, lo que elimina la necesidad de realizar estas comprobaciones en la acción. Una vez que haya extraído las convenciones comunes y organizado la solución

para separar el código de infraestructura y la lógica empresarial de la interfaz de usuario, los métodos de acción de MVC deben ser extremadamente finos:

C#

```
[HttpPut("{id}")]
[ValidateAuthorExists]
public async Task<IActionResult> Put(int id, [FromBody]Author author)
{
    await _authorRepository.UpdateAsync(author);
    return Ok();
}
```

Puede obtener más información sobre la implementación de filtros y descargar un ejemplo de trabajo en el artículo de MSDN Magazine, [Real-World ASP.NET Filtros de MVC Core](#).

Si encuentra que tiene una serie de respuestas comunes de las API basadas en escenarios comunes, como errores de validación (solicitud incorrecta), errores de recurso no encontrados y errores de servidor, puede considerar la posibilidad de usar una abstracción de *resultados*. Los servicios consumidos por los puntos de conexión de API devolverían la abstracción de resultados, y la acción o el punto de conexión del controlador usarían un filtro para traducirlos en `IActionResults`.

Referencias: estructuración de aplicaciones

- **Áreas**
<https://learn.microsoft.com/aspnet/core/mvc/controllers/areas>
- **MSDN Magazine: Funcionalidades para ASP.NET Core MVC**
<https://learn.microsoft.com/archive/msdn-magazine/2016/september/asp-net-core-feature-slices-for-asp-net-core-mvc>
- **Filtros**
<https://learn.microsoft.com/aspnet/core/mvc/controllers/filters>
- **MSDN Magazine: Filtros Real World ASP.NET Core MVC**
<https://learn.microsoft.com/archive/msdn-magazine/2016/august/asp-net-core-real-world-asp-net-core-mvc-filters>
- **Resultado en eShopOnWeb**
<https://github.com/dotnet-architecture/eShopOnWeb/wiki/Patterns#result> [↗](#)

Seguridad

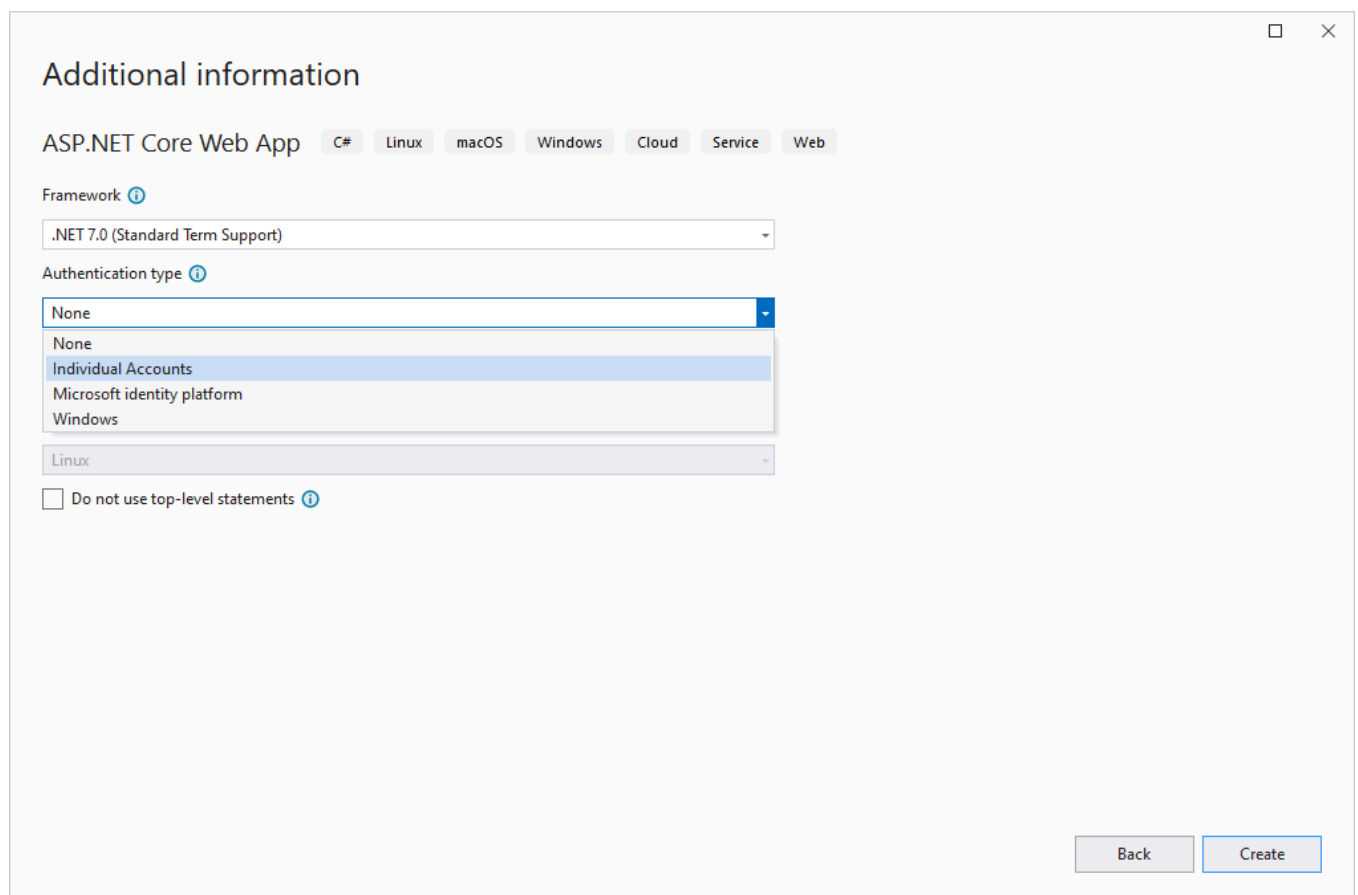
La protección de aplicaciones web es un tema grande, con muchas consideraciones. En su nivel más básico, la seguridad implica asegurarse de saber a quién procede una solicitud

determinada y, a continuación, asegurarse de que la solicitud solo tiene acceso a los recursos que debe. La autenticación es el proceso de comparar las credenciales proporcionadas con una solicitud a las de un almacén de datos de confianza para ver si la solicitud debe tratarse como procedente de una entidad conocida. La autorización es el proceso de restringir el acceso a determinados recursos en función de la identidad del usuario. Un tercer problema de seguridad es proteger las solicitudes de interceptación por parte de terceros, para las que debe asegurarse al menos de [que la aplicación use SSL](#).

identidad

ASP.NET Core Identity es un sistema de pertenencia que puede usar para admitir la funcionalidad de inicio de sesión de la aplicación. Tiene compatibilidad con cuentas de usuario locales, así como compatibilidad con proveedores de inicio de sesión externos, como cuentas de Microsoft, Twitter, Facebook, Google, etc. Además de ASP.NET Core Identity, la aplicación puede usar la autenticación de Windows o un proveedor de identidades de terceros, como [Identity Server](#).

ASP.NET Core Identity está incluido en las nuevas plantillas de proyecto si se selecciona la opción de Cuentas de usuario individuales. Esta plantilla incluye compatibilidad con el registro, el inicio de sesión, los inicios de sesión externos, las contraseñas olvidadas y la funcionalidad adicional.



Additional information

ASP.NET Core Web App C# Linux macOS Windows Cloud Service Web

Framework ⓘ
.NET 7.0 (Standard Term Support)

Authentication type ⓘ
None
Individual Accounts
Microsoft identity platform
Windows

Linux

☐ Do not use top-level statements ⓘ

Back Create

Figura 7-3. Seleccione Cuentas de usuario individuales para que la identidad esté preconfigurada.

La configuración del soporte de identidades se realiza en *Program.cs* o *Startup*, e incluye la configuración de servicios y middleware.

Configuración de la identidad en *Program.cs*

En *Program.cs*, configurará los servicios desde la *WebApplicationBuilder* instancia y, después, una vez creada la aplicación, configurará su middleware. Los puntos clave que se deben tener en cuenta son la llamada a *AddDefaultIdentity* para los servicios necesarios y las llamadas a *UseAuthentication* y *UseAuthorization* que agregan el middleware necesario.

C#

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
var connectionString =
builder.Configuration.GetConnectionString("DefaultConnection");
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(connectionString));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();

builder.Services.AddDefaultIdentity<IdentityUser>(options =>
options.SignIn.RequireConfirmedAccount = true)
    .AddEntityFrameworkStores<ApplicationDbContext>();
builder.Services.AddRazorPages();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    app.UseMigrationsEndPoint();
}
else
{
    app.UseExceptionHandler("/Error");
    // The default HSTS value is 30 days. You may want to change this for production
    scenarios, see https://aka.ms/aspnetcore-hsts.
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthentication();
```

```
app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

Configuración de identidad en el inicio de la aplicación

C#

```
// Add framework services.
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));
builder.Services.AddIdentity<ApplicationUser, IdentityRole>()
    .AddEntityFrameworkStores<ApplicationDbContext>()
    .AddDefaultTokenProviders();
builder.Services.AddMvc();

var app = builder.Build();

if (app.Environment.IsDevelopment())
{
    app.UseMigrationsEndPoint();
}
else
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthentication();
app.UseAuthorization();

app.MapRazorPages();
```

Es importante que `UseAuthentication` y `UseAuthorization` aparezcan antes de `MapRazorPages`. Al configurar servicios de identidad, notarás una llamada a `AddDefaultTokenProviders`. Esto no tiene nada que ver con tokens que se pueden usar para proteger las comunicaciones web, pero en su lugar hace referencia a proveedores que crean avisos que se pueden enviar a los usuarios a través de SMS o correo electrónico para que confirmen su identidad.

Puede obtener más información sobre [cómo configurar la autenticación en dos fases](#) y [habilitar proveedores de inicio de sesión externos](#) desde los documentos oficiales de ASP.NET Core.

Autenticación

La autenticación es el proceso de determinar quién tiene acceso al sistema. Si usa ASP.NET Core Identity y los métodos de configuración que se muestran en la sección anterior, configurará automáticamente algunos valores predeterminados de autenticación en la aplicación. Sin embargo, también puede configurar estos valores predeterminados manualmente o invalidar los establecidos por AddIdentity. Si usa Identity, configura la autenticación basada en cookies como *esquema* predeterminado.

En la autenticación basada en web, normalmente hay hasta cinco acciones que se pueden realizar durante la autenticación de un cliente de un sistema. Estos son:

- Autenticar: Use la información proporcionada por el cliente para crear una identidad para que la usen en la aplicación.
- Desafío Esta acción se usa para requerir que el cliente se identifique a sí mismo.
- Prohibir. Informe al cliente de que está prohibido realizar una acción.
- Iniciar sesión: Conserve el cliente existente de alguna manera.
- Cerrar sesión: quite el cliente de la persistencia.

Hay varias técnicas comunes para realizar la autenticación en aplicaciones web. Estos se conocen como esquemas. Un esquema determinado definirá acciones para algunas o todas las opciones anteriores. Algunos esquemas solo admiten un subconjunto de acciones y pueden necesitar un esquema separado para realizar aquellas que no admiten. Por ejemplo, el esquema de OpenId-Connect (OIDC) no admite el inicio de sesión ni el cierre de sesión, pero normalmente está configurado para usar la autenticación de cookies para esta persistencia.

En su aplicación ASP.NET Core, puede configurar un `DefaultAuthenticateScheme` así como esquemas específicos opcionales para cada una de las acciones descritas anteriormente. Por ejemplo, `DefaultChallengeScheme` y `DefaultForbidScheme`. La llamada a [AddIdentity](#) configura varios aspectos de la aplicación y añade muchos servicios necesarios. También incluye esta llamada para configurar el esquema de autenticación:

C#

```
builder.Services.AddAuthentication(options =>
{
    options.DefaultAuthenticateScheme = IdentityConstants.ApplicationScheme;
    options.DefaultChallengeScheme = IdentityConstants.ApplicationScheme;
    options.DefaultSignInScheme = IdentityConstants.ExternalScheme;
});
```

Estos esquemas usan cookies para la persistencia y el redireccionamiento a las páginas de inicio de sesión para la autenticación de forma predeterminada. Estos esquemas son adecuados para las aplicaciones web que interactúan con los usuarios a través de exploradores

web, pero no se recomiendan para las API. En su lugar, las API suelen usar otra forma de autenticación, como tokens de portador JWT.

Las API web se consumen mediante código, como `HttpClient` en aplicaciones .NET y tipos equivalentes en otros marcos de trabajo. Estos clientes esperan una respuesta utilizable de una llamada API o un código de estado que indica qué, si existe, problema. Estos clientes no interactúan a través de un explorador y no representan ni interactúan con ningún CÓDIGO HTML que pueda devolver una API. Por lo tanto, no es adecuado que los puntos de conexión de API redirijan a sus clientes a páginas de inicio de sesión si no se autentican. Otro esquema es más adecuado.

Para configurar la autenticación para las API, puede configurar la autenticación como la siguiente, que usa el `PublicApi` proyecto en la aplicación de referencia `eShopOnWeb`:

C#

```
builder.Services
    .AddAuthentication(config =>
    {
        config.DefaultScheme = JwtBearerDefaults.AuthenticationScheme;
    })
    .AddJwtBearer(config =>
    {
        config.RequireHttpsMetadata = false;
        config.SaveToken = true;
        config.TokenValidationParameters = new TokenValidationParameters
        {
            ValidateIssuerSigningKey = true,
            IssuerSigningKey = new SymmetricSecurityKey(key),
            ValidateIssuer = false,
            ValidateAudience = false
        };
    });
```

Aunque es posible configurar varios esquemas de autenticación diferentes dentro de un solo proyecto, es mucho más sencillo configurar un único esquema predeterminado. Por este motivo, entre otros, la aplicación de referencia `eShopOnWeb` separa sus API en su propio proyecto, `PublicApi`, independiente del proyecto principal `Web`, que incluye las vistas de la aplicación y las páginas Razor.

Autenticación en Blazor aplicaciones

Blazor Las aplicaciones de servidor pueden aprovechar las mismas características de autenticación que cualquier otra aplicación de ASP.NET Core. `BlazorWebAssembly` Sin embargo, las aplicaciones no pueden usar los proveedores de identidades y autenticación

integrados, ya que se ejecutan en el explorador. BlazorWebAssembly Las aplicaciones pueden almacenar el estado de autenticación de usuario localmente y pueden acceder a las notificaciones para determinar qué acciones deben poder realizar los usuarios. Sin embargo, todas las comprobaciones de autenticación y autorización deben realizarse en el servidor, independientemente de cualquier lógica implementada dentro de la BlazorWebAssembly aplicación, ya que los usuarios pueden omitir fácilmente la aplicación e interactuar directamente con las API.

Referencias: autenticación

- Acciones de autenticación y valores predeterminados
<https://stackoverflow.com/a/52493428> ↗
- Autenticación y autorización para SPA
<https://learn.microsoft.com/aspnet/core/security/authentication/identity-api-authorization>
- autenticación y autorización principales Blazor de ASP.NET
<https://learn.microsoft.com/aspnet/core/blazor/security/>
- Seguridad: autenticación y autorización en formularios web de ASP.NET y Blazor
<https://learn.microsoft.com/dotnet/architecture/blazor-for-web-forms-developers/security-authentication-authorization>

Autorización

La forma más sencilla de autorización implica restringir el acceso a los usuarios anónimos. Esta funcionalidad se puede lograr aplicando el `[Authorize]` atributo a determinados controladores o acciones. Si se usan roles, el atributo se puede ampliar aún más para restringir el acceso a los usuarios que pertenecen a determinados roles, como se muestra a continuación:

C#

```
[Authorize(Roles = "HRManager,Finance")]  
public class SalaryController : Controller  
{  
  
}
```

En este caso, los usuarios que pertenecen a los `HRManager` roles o `Finance` (o ambos) tendrán acceso a `SalaryController`. Para requerir que un usuario pertenezca a varios roles (no solo uno de varios), puede aplicar el atributo varias veces, especificando un rol necesario cada vez.

Especificar determinados conjuntos de roles como cadenas en muchos controladores y acciones diferentes puede provocar una repetición no deseada. Como mínimo, defina

constantes para estos literales de cadena y use las constantes en cualquier lugar donde necesite especificar la cadena. También puede configurar directivas de autorización, que encapsulan las reglas de autorización y, a continuación, especificar la directiva en lugar de roles individuales al aplicar el `[Authorize]` atributo :

C#

```
[Authorize(Policy = "CanViewPrivateReport")]
public IActionResult ExecutiveSalaryReport()
{
    return View();
}
```

Al usar las directivas de esta manera, se pueden separar los tipos de acciones que se restringen de las reglas o roles específicos a los que se aplican. Más adelante, si crea un nuevo rol que necesita tener acceso a determinados recursos, solo puede actualizar una directiva, en lugar de actualizar cada lista de roles en cada `[Authorize]` atributo.

Reclamaciones

Las declaraciones son pares nombre-valor que representan las propiedades de un usuario autenticado. Por ejemplo, es posible almacenar el número de empleado de los usuarios como una notificación. Después, las notificaciones se pueden usar como parte de las directivas de autorización. Podría crear una directiva denominada "EmployeeOnly" que requiera la existencia de una notificación denominada "EmployeeNumber", como se muestra en este ejemplo:

C#

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();
    services.AddAuthorization(options =>
    {
        options.AddPolicy("EmployeeOnly", policy =>
        policy.RequireClaim("EmployeeNumber"));
    });
}
```

Después, esta directiva podría usarse con el `[Authorize]` atributo para proteger cualquier controlador o acción, como se ha descrito anteriormente.

Protección de las API web

La mayoría de las API web deben implementar un sistema de autenticación basado en tokens. La autenticación de tokens no tiene estado y está diseñada para ser escalable. En un sistema de autenticación basado en tokens, el cliente primero debe autenticarse con el proveedor de autenticación. Si se ejecuta correctamente, el cliente emite un token, que es simplemente una cadena criptográficamente significativa de caracteres. El formato más común para los tokens es JSON Web Token o JWT (a menudo se pronuncia "jot"). Cuando el cliente necesita emitir una solicitud a una API, agrega este token como encabezado en la solicitud. A continuación, el servidor valida el token que se encuentra en el encabezado de solicitud antes de completar la solicitud. En la figura 7-4 se muestra este proceso.

Token-Based Authentication

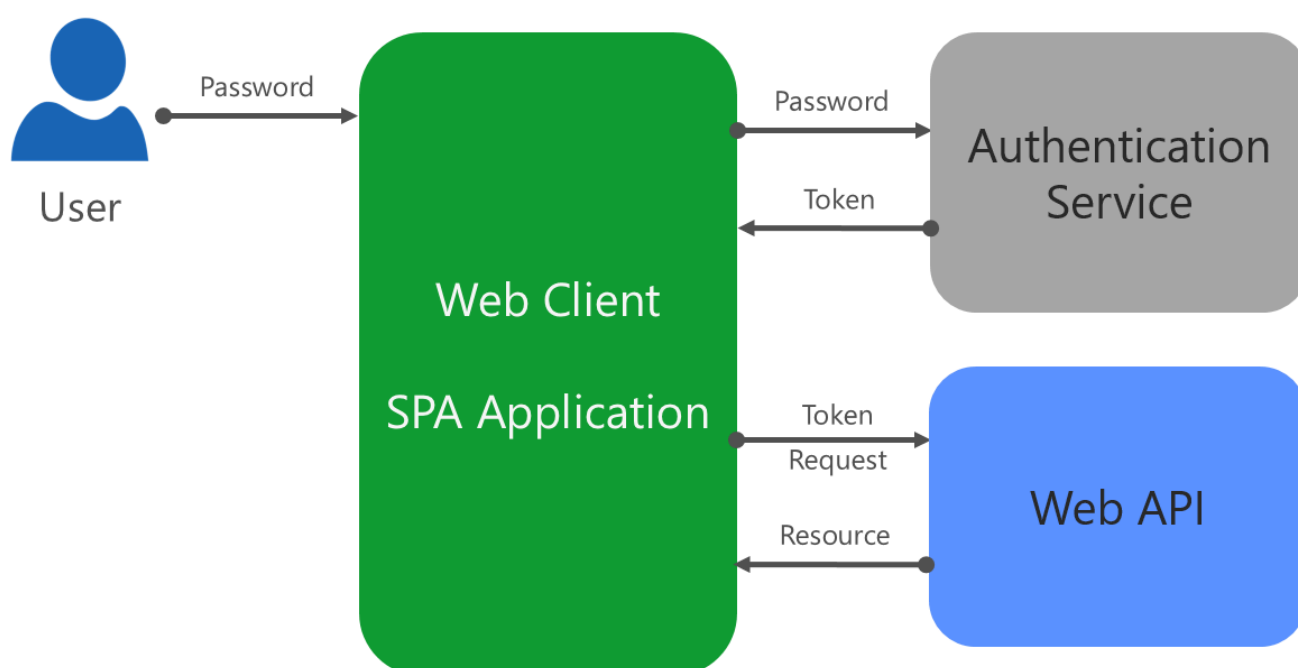


Figura 7-4. Autenticación basada en tokens para las API web.

Puede crear su propio servicio de autenticación, integrarlo con Azure AD y OAuth, o implementar un servicio mediante una herramienta de código abierto como [IdentityServer](#).

Los tokens JWT pueden incluir declaraciones sobre el usuario, que se pueden leer en el cliente o el servidor. Puede usar una herramienta como [jwt.io](#) para ver el contenido de un token JWT. No almacene datos confidenciales como contraseñas o claves en tokens JWT, ya que su contenido se lee fácilmente.

Al usar tokens JWT con SPA o BlazorWebAssembly aplicaciones, debe almacenar el token en algún lugar del cliente y, a continuación, agregarlo a cada llamada API. Normalmente, esta actividad se realiza como encabezado, como se muestra en el código siguiente:

```
// AuthService.cs in BlazorAdmin project of eShopOnWeb
private async Task SetAuthorizationHeader()
{
    var token = await GetToken();
    _httpClient.DefaultRequestHeaders.Authorization = new
AuthenticationHeaderValue("Bearer", token);
}
```

Después de llamar al método anterior, las solicitudes realizadas con `_httpClient` tendrán el token incrustado en los encabezados de la solicitud, lo que permite que la API del lado servidor se autentique y autorice la solicitud.

Seguridad personalizada

⊗ Precaución

Como regla general, evite implementar sus propias implementaciones de seguridad personalizadas.

Tenga especial cuidado al crear una implementación de criptografía, una pertenencia de usuarios o un sistema de generación de tokens propios. Hay muchas alternativas comerciales y de código abierto disponibles, que seguramente tendrán una mejor seguridad que una implementación personalizada.

Referencias: seguridad

- Información general sobre la documentación de seguridad
<https://learn.microsoft.com/aspnet/core/security/>
- Aplicación de SSL en una aplicación de ASP.NET Core
<https://learn.microsoft.com/aspnet/core/security/enforcing-ssl>
- Introducción a la identidad
<https://learn.microsoft.com/aspnet/core/security/authentication/identity>
- Introducción a la autorización
<https://learn.microsoft.com/aspnet/core/security/authorization/introduction>
- Autenticación y autorización para aplicaciones de API en Azure App Service
<https://learn.microsoft.com/azure/app-service-api/app-service-api-authentication>
- Identity Server
<https://github.com/IdentityServer> ↗

Comunicación de cliente

Además de atender páginas y responder a solicitudes de datos a través de API web, las aplicaciones de ASP.NET Core pueden comunicarse directamente con los clientes conectados. Esta comunicación saliente puede usar una variedad de tecnologías de transporte, siendo WebSockets la más común. ASP.NET Core SignalR es una biblioteca que facilita la adición de funcionalidades de comunicación de servidor a cliente en tiempo real a las aplicaciones. SignalR admite una variedad de tecnologías de transporte, como WebSockets, y abstrae muchos de los detalles de implementación del desarrollador.

La comunicación de cliente en tiempo real, ya sea mediante WebSockets directamente u otras técnicas, es útil en una variedad de escenarios de aplicación. Algunos ejemplos son:

- Aplicaciones de sala de chat en directo
- Supervisión de aplicaciones
- Actualizaciones del progreso del trabajo
- Notificaciones
- Aplicaciones de formularios interactivos

Al integrar la comunicación con los clientes en las aplicaciones, normalmente hay dos componentes:

- Administrador de conexiones del lado servidor (SignalR Hub, WebSocketManager, WebSocketHandler)
- Biblioteca del lado cliente

Los clientes no están limitados a exploradores: las aplicaciones móviles, las aplicaciones de consola y otras aplicaciones nativas también pueden comunicarse mediante SignalR/WebSockets. El siguiente programa simple devuelve todo el contenido enviado a una aplicación de chat a la consola, como parte de una aplicación de ejemplo de WebSocketManager:

C#

```
public class Program
{
    private static Connection _connection;
    public static void Main(string[] args)
    {
        StartConnectionAsync();
        _connection.On("receiveMessage", (arguments) =>
        {
            Console.WriteLine($"{arguments[0]} said: {arguments[1]}");
        });
        Console.ReadLine();
    }
}
```

```

        StopConnectionAsync();
    }

    public static async Task StartConnectionAsync()
    {
        _connection = new Connection();
        await _connection.StartConnectionAsync("ws://localhost:65110/chat");
    }

    public static async Task StopConnectionAsync()
    {
        await _connection.StopConnectionAsync();
    }
}

```

Considere las formas en que las aplicaciones se comunican directamente con las aplicaciones cliente y considere si la comunicación en tiempo real mejoraría la experiencia del usuario de la aplicación.

Comunicación con clientes – Referencias

- SignalR de ASP.NET Core
<https://github.com/dotnet/aspnetcore/tree/main/src/SignalR>
- Administrador de WebSocket
<https://github.com/radu-matei/websocket-manager>

Diseño controlado por dominio: ¿Debe aplicarlo?

Domain-Driven Diseño (DDD) es un enfoque ágil para crear software que enfatiza centrarse en el *dominio empresarial*. Hace hincapié en la comunicación y la interacción con expertos en dominios empresariales, quienes pueden explicar a los desarrolladores cómo funciona el sistema en el mundo real. Por ejemplo, si va a crear un sistema que controle las operaciones de acciones, es posible que su experto en dominio sea un agente de acciones experimentado. DDD está diseñado para abordar problemas empresariales grandes y complejos, y a menudo no es adecuado para aplicaciones más pequeñas y sencillas, ya que la inversión en comprender y modelar el dominio no vale la pena.

Al desarrollar software siguiendo un enfoque de DDD, el equipo (incluidas las partes interesadas y colaboradores no técnicos) debe desarrollar un *lenguaje omnipresente* para el área problemática. Es decir, se debe usar la misma terminología para el concepto real que se modela, el equivalente de software y cualquier estructura que pueda existir para conservar el concepto (por ejemplo, tablas de base de datos). Por lo tanto, los conceptos descritos en el lenguaje omnipresente deben formar la base del *modelo de dominio*.

El modelo de dominio consta de objetos que interactúan entre sí para representar el comportamiento del sistema. Estos objetos pueden estar incluidos en las siguientes categorías:

- Entidades, que representan objetos con un hilo de identidad. Normalmente, las entidades se almacenan en almacenamiento persistente con una clave por la que se pueden recuperar más adelante.
- [Agregados](#), que representan grupos de objetos que se deben conservar como una unidad.
- [Objetos value](#), que representan conceptos que se pueden comparar basándose en la suma de sus valores de propiedad. Por ejemplo, `DateRange` consta de una fecha de inicio y finalización.
- [Eventos de dominio](#), que representan cosas que ocurren dentro del sistema que son de interés para otras partes del sistema.

Un modelo de dominio DDD debe encapsular el comportamiento complejo dentro del modelo. Las entidades, en particular, no deben ser simplemente colecciones de propiedades. Cuando el modelo de dominio carece de comportamiento y simplemente representa el estado del sistema, se dice que es un [modelo anémico](#), que no es deseable en DDD.

Además de estos tipos de modelo, DDD normalmente emplea una variedad de patrones:

- [Repositorio](#), para abstraer detalles relacionados con la persistencia.
- [Fábrica](#), para encapsular la creación de objetos complejos.
- [Servicios](#), para encapsular detalles de implementación de infraestructura o comportamiento complejo.
- [Comando](#), para desacoplar la emisión de comandos y ejecutar el comando en sí.
- [Especificación](#), para encapsular los detalles de la consulta.

DDD también recomienda el uso de la arquitectura limpia descrita anteriormente, lo que permite acoplamiento flexible, encapsulación y código que se puede comprobar fácilmente mediante pruebas unitarias.

¿Cuándo debe aplicar el DDD?

DDD es adecuado para aplicaciones grandes con una complejidad empresarial significativa (no solo técnica). La aplicación debe requerir el conocimiento de los expertos en dominio. Debe haber un comportamiento significativo en el propio modelo de dominio, que representa las

reglas de negocio y las interacciones más allá de simplemente almacenar y recuperar el estado actual de varios registros de almacenes de datos.

¿Cuándo no debes aplicar DDD?

DDD implica inversiones en modelado, arquitectura y comunicación que pueden no estar justificadas para aplicaciones más pequeñas que esencialmente solo realizan operaciones CRUD (crear/leer/actualizar/borrar). Si decide abordar la aplicación después de DDD, pero encuentra que el dominio tiene un modelo anémico sin comportamiento, es posible que tenga que replantearse el enfoque. Es posible que la aplicación no necesite DDD o que necesite ayuda para refactorizar la aplicación para encapsular la lógica de negocios en el modelo de dominio, en lugar de en la base de datos o la interfaz de usuario.

Un enfoque híbrido sería utilizar DDD solo para las áreas transaccionales o más complejas de la aplicación, pero no para las partes más sencillas como las de creación, lectura, actualización y eliminación (CRUD) o de solo lectura de la aplicación. Por ejemplo, no necesita las restricciones de un agregado si está consultando datos para mostrar un informe o para visualizar los datos de un panel. Es perfectamente aceptable tener un modelo de lectura independiente y más sencillo para estos requisitos.

Referencias – Diseño Domain-Driven

- DDD en lenguaje sencillo (respuesta de StackOverflow)
<https://stackoverflow.com/questions/1222392/can-someone-explain-domain-driven-design-ddd-in-plain-english-please/1222488#1222488> ↗

Despliegue

Hay algunos pasos implicados en el proceso de implementación de la aplicación ASP.NET Core, independientemente de dónde se hospede. El primer paso es publicar la aplicación, que se puede realizar mediante el comando de la `dotnet publish` CLI. Este paso compilará la aplicación y colocará todos los archivos necesarios para ejecutar la aplicación en una carpeta designada. Al implementar desde Visual Studio, este paso se realiza automáticamente. La carpeta publish contiene archivos .exe y .dll para la aplicación y sus dependencias. Una aplicación independiente también incluirá una versión del entorno de ejecución de .NET. ASP.NET aplicaciones principales también incluirán archivos de configuración, recursos de cliente estáticos y vistas de MVC.

Las aplicaciones ASP.NET Core son aplicaciones de consola que deben iniciarse cuando el servidor arranca y deben reiniciarse si la aplicación (o el servidor) se bloquea. Un administrador de procesos se puede usar para automatizar este proceso. Los administradores de procesos

más comunes para ASP.NET Core son Nginx y Apache en Linux e IIS o servicio de Windows en Windows.

Además de un administrador de procesos, las aplicaciones ASP.NET Core pueden usar un servidor proxy inverso. Un servidor proxy inverso recibe solicitudes HTTP de Internet y las reenvía a Kestrel después de un control preliminar. Los servidores proxy inversos proporcionan una capa de seguridad para la aplicación. Kestrel tampoco admite el hospedaje de varias aplicaciones en el mismo puerto, por lo que no se pueden usar técnicas como encabezados de host para habilitar el hospedaje de varias aplicaciones en el mismo puerto y dirección IP.

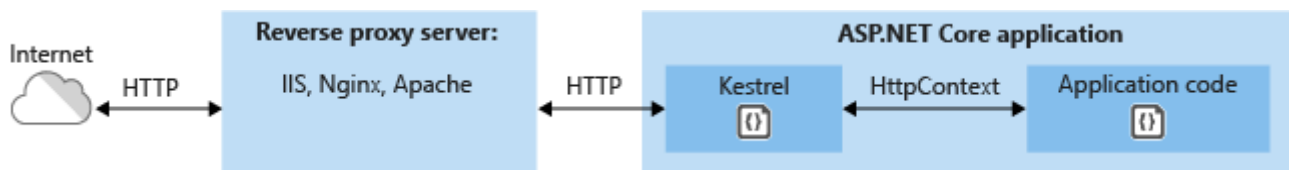


Figura 7-5. ASP.NET hospedado en Kestrel detrás de un servidor proxy inverso

Otro escenario en el que un proxy inverso puede resultar útil es proteger varias aplicaciones mediante SSL/HTTPS. En este caso, solo el proxy inverso tendría que tener SSL configurado. La comunicación entre el servidor proxy inverso y Kestrel podría tener lugar a través de HTTP, como se muestra en la figura 7-6.

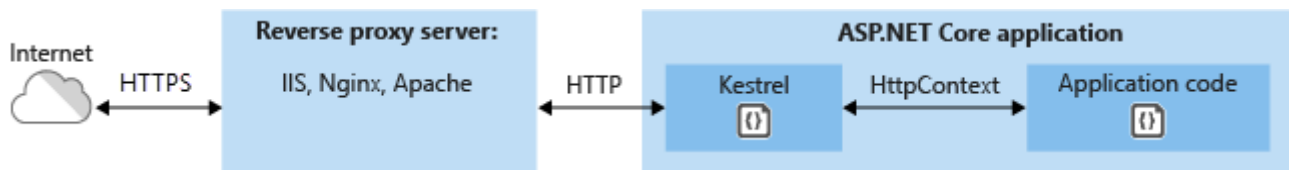


Figura 7-6. ASP.NET hospedado detrás de un servidor proxy inverso protegido por HTTPS

Un enfoque cada vez más popular es hospedar la aplicación ASP.NET Core en un contenedor de Docker, que luego se puede hospedar localmente o implementar en Azure para el hospedaje basado en la nube. El contenedor de Docker podría contener el código de la aplicación, que se ejecuta en Kestrel y se implementaría detrás de un servidor proxy inverso, como se muestra anteriormente.

Si hospeda la aplicación en Azure, puede usar Microsoft Azure Application Gateway como una aplicación virtual dedicada para proporcionar varios servicios. Además de actuar como proxy inverso para aplicaciones individuales, Application Gateway también puede ofrecer las siguientes características:

- Equilibrio de carga HTTP
- Descarga SSL (SSL solo para Internet)
- SSL de extremo a extremo

- Enrutamiento de varios sitios (consolidar hasta 20 sitios en una sola instancia de Application Gateway)
- Firewall de aplicaciones web
- Compatibilidad con Websocket
- Diagnósticos avanzados

Obtenga más información sobre las opciones de implementación de Azure en el [capítulo 10](#).

Referencias: implementación

- Información general sobre hospedaje e implementación
<https://learn.microsoft.com/aspnet/core/publishing/>
- Cuándo usar Kestrel con un proxy inverso
<https://learn.microsoft.com/aspnet/core/fundamentals/servers/kestrel#when-to-use-kestrel-with-a-reverse-proxy>
- Hospedaje de aplicaciones ASP.NET Core en Docker
<https://learn.microsoft.com/aspnet/core/publishing/docker>
- Introducción a Azure Application Gateway
<https://learn.microsoft.com/azure/application-gateway/application-gateway-introduction>

Siguiente anterior

Trabajar con datos en aplicaciones ASP.NET Core

Artículo • 25/04/2023

💡 Sugerencia

Este contenido es un extracto del libro electrónico "Architect Modern Web Applications with ASP.NET Core and Azure" (Diseño de la arquitectura de aplicaciones web modernas con ASP.NET Core y Azure), disponible en [Documentación de .NET](#) o como un PDF descargable y gratuito para leerlo sin conexión.

Descargar PDF



"Los datos son algo muy valioso y durarán más que los propios sistemas".

Tim Berners-Lee

El acceso a datos es una parte importante de la mayoría de las aplicaciones de software. ASP.NET Core admite varias opciones de acceso a datos, incluido Entity Framework Core (así como Entity Framework 6) y puede funcionar con cualquier marco de acceso a datos de .NET. La elección del marco de acceso a datos que se va a usar depende de las necesidades de la aplicación. La abstracción de estas opciones de los proyectos ApplicationCore y UI, y la encapsulación de los detalles de implementación en la infraestructura, ayuda a crear software de acoplamiento flexible y que se pueda probar.

Entity Framework Core (para bases de datos relacionales)

Si va a escribir una aplicación ASP.NET Core nueva que tenga que trabajar con datos relacionales, Entity Framework Core (EF Core) es la manera recomendada para que la aplicación tenga acceso a sus datos. EF Core es un asignador relacional de objetos (O/RM) que permite a los desarrolladores de .NET persistir objetos en y desde un origen de datos. Elimina la necesidad de la mayor parte del código de acceso a datos que los desarrolladores normalmente tendrían que escribir. Al igual que ASP.NET Core, EF Core se ha vuelto a escribir desde el principio para admitir aplicaciones multiplataforma modulares. Se agrega a la aplicación como un paquete NuGet, se configura durante el inicio de la aplicación y se solicita a través de la inserción de dependencias siempre que se necesite.

Para usar EF Core con una base de datos de SQL Server, ejecute el comando siguiente de la CLI de DotNet:

CLI de .NET

```
dotnet add package Microsoft.EntityFrameworkCore.SqlServer
```

Para agregar compatibilidad para un origen de datos InMemory, para las pruebas:

CLI de .NET

```
dotnet add package Microsoft.EntityFrameworkCore.InMemory
```

DbContext

Para trabajar con EF Core, necesita una subclase de [DbContext](#). Esta clase contiene propiedades que representan las colecciones de las entidades con las que la aplicación va a trabajar. En el ejemplo eShopOnWeb, se incluye un elemento `CatalogContext` con colecciones de productos, marcas y tipos:

C#

```
public class CatalogContext : DbContext
{
    public CatalogContext(DbContextOptions<CatalogContext> options) :
    base(options)
    {
    }

    public DbSet<CatalogItem> CatalogItems { get; set; }
    public DbSet<CatalogBrand> CatalogBrands { get; set; }
    public DbSet<CatalogType> CatalogTypes { get; set; }
}
```

El elemento DbContext debe tener un constructor que acepte DbContextOptions y pase este argumento al constructor DbContext base. Si solo tiene un elemento DbContext en la aplicación, puede pasar una instancia de DbContextOptions<T>, pero, si tiene más, tendrá que usar el tipo genérico DbContextOptions y pasar el tipo de DbContext como el parámetro genérico.

Configuración de EF Core

En la aplicación de ASP.NET Core, normalmente configurará EF Core en *Program.cs* con otras dependencias de la aplicación. EF Core usa un elemento DbContextOptionsBuilder, que admite varios métodos de extensión útiles para optimizar su configuración. Para configurar CatalogContext para usar una base de datos de SQL Server con una cadena de conexión definida en la configuración, se debe agregar el código siguiente:

C#

```
builder.Services.AddDbContext<CatalogContext>(
    options => options.UseSqlServer(
        builder.Configuration.GetConnectionString("DefaultConnection")));
```

Para usar la base de datos en memoria:

C#

```
builder.Services.AddDbContext<CatalogContext>(options =>
    options.UseInMemoryDatabase());
```

Después de instalar EF Core, crear un tipo secundario de DbContext y agregar el tipo a los servicios de la aplicación, ya tiene todo listo para usar EF Core. Puede solicitar una instancia del tipo de DbContext en cualquier servicio que lo necesite y empezar a trabajar con las entidades persistentes con LINQ como si simplemente estuvieran en una colección. EF Core realiza el trabajo de traducir las expresiones de LINQ a consultas SQL para almacenar y recuperar los datos.

Puede ver las consultas que ejecuta EF Core si configura un registrador y se asegura de que su nivel se establece al menos en Información, como se muestra en la figura 8-1.

```
C:\Program Files\dotnet\dotnet.exe
Hosting environment: Development
Content root path: C:\dev\github\eShopOnWeb\src\Web
Now listening on: http://0.0.0.0:5106
Application started. Press Ctrl+C to shut down.
Info: Microsoft.AspNetCore.Hosting.Internal.WebHost[1]
      Request starting HTTP/1.1 GET http://localhost:5106/
Info: Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker[1]
      Executing action method Microsoft.eShopWeb.Controllers.CatalogController.Index (Web) with arguments (, , ) - Model
State is Valid
Info: Microsoft.EntityFrameworkCore.Storage.IRelationalCommandBuilderFactory[1]
      Executed DbCommand (20ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      SELECT COUNT_BIG(*)
      FROM [Catalog] AS [c]
Info: Microsoft.EntityFrameworkCore.Storage.IRelationalCommandBuilderFactory[1]
      Executed DbCommand (53ms) [Parameters=[@__p_0='?', @__p_1='?'], CommandType='Text', CommandTimeout='30']
      SELECT [c].[Id], [c].[CatalogBrandId], [c].[CatalogTypeId], [c].[Description], [c].[Name], [c].[PictureUri], [c].[
Price]
      FROM [Catalog] AS [c]
      ORDER BY (SELECT 1)
      OFFSET @__p_0 ROWS FETCH NEXT @__p_1 ROWS ONLY
Info: Microsoft.EntityFrameworkCore.Storage.IRelationalCommandBuilderFactory[1]
      Executed DbCommand (0ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      SELECT [c].[Id], [c].[Brand]
      FROM [CatalogBrand] AS [c]
Info: Microsoft.EntityFrameworkCore.Storage.IRelationalCommandBuilderFactory[1]
      Executed DbCommand (0ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      SELECT [c].[Id], [c].[Type]
      FROM [CatalogType] AS [c]
Info: Microsoft.AspNetCore.Mvc.ViewFeatures.Internal.ViewResultExecutor[1]
      Executing ViewResult, running view at path /Views/Catalog/Index.cshtml.
```

Figura 8-1. Registro de consultas de EF Core en la consola

Recuperación y almacenamiento de los datos

Para recuperar datos de EF Core, se obtiene acceso a la propiedad adecuada y se usa LINQ para filtrar el resultado. También se puede usar LINQ para realizar la proyección, y transformar el resultado de un tipo a otro. En el ejemplo siguiente, se recuperaría `CatalogBrands`, ordenadas por nombre, filtradas por su propiedad `Enabled` y proyectadas en un tipo `SelectListItem`:

```
C#

var brandItems = await _context.CatalogBrands
    .Where(b => b.Enabled)
    .OrderBy(b => b.Name)
    .Select(b => new SelectListItem {
        Value = b.Id, Text = b.Name })
    .ToListAsync();
```

En el ejemplo anterior, es importante agregar la llamada a `ToToListAsync` para ejecutar la consulta inmediatamente. En caso contrario, la instrucción asignará un elemento `IQueryable<SelectListItem>` a `brandItems`, que no se ejecutará hasta que se enumere. Devolver los resultados de `IQueryable` desde los métodos tiene sus ventajas y desventajas. Permite modificar más la consulta que EF Core va a construir, pero también se pueden generar errores que solo se producen en tiempo de ejecución, si se agregan a la consulta operaciones que EF Core no puede traducir. Generalmente, es más seguro pasar los filtros al método que realiza el acceso a datos y devolver una colección en memoria (por ejemplo, `List<T>`) como resultado.

EF Core realiza el seguimiento de los cambios en las entidades que recupera de la persistencia. Para guardar los cambios en una entidad de la que se realiza el seguimiento, simplemente llame al método `SaveChangesAsync` en `DbContext`, asegurándose de que sea la misma instancia de `DbContext` que se usó para recuperar la entidad. La adición y eliminación de entidades se realiza directamente en la propiedad `DbSet` adecuada, de nuevo con una llamada a `SaveChangesAsync` para ejecutar los comandos de base de datos. En el ejemplo siguiente se muestra cómo agregar, actualizar y quitar entidades de la persistencia.

C#

```
// create
var newBrand = new CatalogBrand() { Brand = "Acme" };
_context.Add(newBrand);
await _context.SaveChangesAsync();

// read and update
var existingBrand = _context.CatalogBrands.Find(1);
existingBrand.Brand = "Updated Brand";
await _context.SaveChangesAsync();

// read and delete (alternate Find syntax)
var brandToDelete = _context.Find<CatalogBrand>(2);
_context.CatalogBrands.Remove(brandToDelete);
await _context.SaveChangesAsync();
```

EF Core es compatible con métodos sincrónicos y asincrónicos para recuperar y guardar. En las aplicaciones web, se recomienda usar el patrón `async/await` con los métodos asincrónicos, para que los subprocesos de servidor web no se bloqueen mientras se espera a que finalicen las operaciones de acceso a datos.

Para más información, consulte [Almacenamiento en búfer y streaming](#).

Recuperación de datos relacionados

Cuando EF Core recupera entidades, rellena todas las propiedades que se almacenan directamente con esa entidad en la base de datos. Las propiedades de navegación, como las listas de entidades relacionadas, no se rellenan y pueden tener su valor establecido en `NULL`. Este proceso garantiza que EF Core no recupere cambios de más datos de los necesarios, lo que es especialmente importante para las aplicaciones web, que deben procesar las solicitudes rápidamente y devolver respuestas de manera eficiente. Para incluir las relaciones con una entidad mediante la *carga diligente*, especifique la propiedad con el método de extensión `Include` en la consulta, como se muestra aquí:

C#

```
// .Include requires using Microsoft.EntityFrameworkCore
var brandsWithItems = await _context.CatalogBrands
    .Include(b => b.Items)
    .ToListAsync();
```

Puede incluir varias relaciones y también relaciones secundarias mediante `ThenInclude`. EF Core ejecutará una sola consulta para recuperar el conjunto resultante de entidades. Como alternativa, puede incluir propiedades de navegación de las propiedades de navegación; para ello, pase una cadena separada por "." al método de extensión `.Include()`. Por ejemplo:

C#

```
.Include("Items.Products")
```

Además de encapsular la lógica de filtro, puede especificar la forma de los datos que se van a devolver, incluidas las propiedades que se van a rellenar. En el ejemplo de `eShopOnWeb` se incluyen varias especificaciones que demuestran cómo encapsular información de carga diligente dentro de la especificación. Puede ver cómo se usa la especificación como parte de una consulta aquí:

C#

```
// Includes all expression-based includes
query = specification.Includes.Aggregate(query,
    (current, include) => current.Include(include));

// Include any string-based include statements
query = specification.IncludeStrings.Aggregate(query,
    (current, include) => current.Include(include));
```

Otra opción para cargar los datos relacionados consiste en usar la *carga explícita*. La carga explícita permite cargar datos adicionales en una entidad que ya se ha recuperado. Como este enfoque implica una solicitud independiente a la base de datos, no se recomienda para las aplicaciones web, que deben minimizar el número de recorridos de ida y vuelta a la base de datos realizados por cada solicitud.

La *carga diferida* es una característica que carga automáticamente los datos relacionados tal y como la aplicación hace referencia a ellos. EF Core ha agregado compatibilidad para la carga diferida en la versión 2.1. La carga diferida no está habilitada de forma predeterminada y requiere que se instale

`Microsoft.EntityFrameworkCore.Proxies`. Como sucede con la carga explícita,

normalmente la carga diferida se debe deshabilitar para las aplicaciones web, ya que su uso dará como resultado que se realicen consultas de base de datos adicionales dentro de cada solicitud web. Desafortunadamente, la sobrecarga que produce la carga diferida a menudo pasa desapercibida en tiempo de desarrollo, cuando la latencia es reducida y los conjuntos de datos que se usan para las pruebas normalmente son pequeños. Pero en producción, con más usuarios, datos y latencia, las solicitudes de base de datos adicionales pueden causar un bajo rendimiento para las aplicaciones web que hacen un uso intensivo de la carga diferida.

[Avoid Lazy Loading Entities in ASPNET Applications](#) (Evitar la carga diferida de entidades en aplicaciones ASP.NET)

Es recomendable probar la aplicación para examinar las consultas reales de base de datos que realiza. En determinadas circunstancias, EF Core puede realizar muchas más consultas o una consulta que consume más recursos de lo que es óptimo para la aplicación. Este problema se conoce como [explosión cartesiana](#). El equipo de EF Core pone a disposición el [método AsSplitQuery](#) como una de las maneras para optimizar el comportamiento en tiempo de ejecución.

Encapsulación de datos

EF Core admite varias características que permiten que el modelo encapsule correctamente su estado. Un problema habitual de los modelos de dominio es que exponen propiedades de navegación de colecciones como tipos de lista públicamente accesibles. Este problema permite que cualquier colaborador manipule el contenido de estos tipos de colecciones, con lo que se pueden omitir importantes reglas de negocio relacionadas con la colección, lo que podría dejar el objeto en un estado no válido. La solución a este problema es conceder acceso de solo lectura a las colecciones relacionadas y proporcionar explícitamente métodos que definan formas para que los clientes las manipulen, como en el ejemplo siguiente:

C#

```
public class Basket : BaseEntity
{
    public string BuyerId { get; set; }
    private readonly List<BasketItem> _items = new List<BasketItem>();
    public IReadOnlyCollection<BasketItem> Items => _items.AsReadOnly();

    public void AddItem(int catalogItemId, decimal unitPrice, int quantity =
1)
    {
        var existingItem = Items.FirstOrDefault(i => i.CatalogItemId ==
catalogItemId);
        if (existingItem == null)
```

```

    {
        _items.Add(new BasketItem()
        {
            CatalogItemId = catalogItemId,
            Quantity = quantity,
            UnitPrice = unitPrice
        });
    }
    else existingItem.Quantity += quantity;
}
}

```

Este tipo de entidad no expone ninguna propiedad `List` o `ICollection` pública, sino que expone un tipo `ICollection` que encapsula el tipo de lista subyacente. Al usar este patrón, puede indicar a Entity Framework Core que use el campo de respaldo de esta manera:

C#

```

private void ConfigureBasket(EntityTypeBuilder<Basket> builder)
{
    var navigation = builder.Metadata.FindNavigation(nameof(Basket.Items));

    navigation.SetPropertyAccessMode(PropertyAccessMode.Field);
}

```

Otra manera de mejorar el modelo de dominio es usar objetos de valor para los tipos que carecen de identidad y solo se distinguen por sus propiedades. Usar estos tipos como propiedades de sus entidades puede ayudar a mantener la lógica específica del objeto de valor al que pertenece y evitar que se duplique la lógica en varias entidades que usen el mismo concepto. En Entity Framework Core, puede conservar los objetos de valor en la misma tabla que su entidad en propiedad configurando el tipo como entidad en propiedad. Por ejemplo:

C#

```

private void ConfigureOrder(EntityTypeBuilder<Order> builder)
{
    builder.OwnsOne(o => o.ShipToAddress);
}

```

En este ejemplo, la propiedad `ShipToAddress` es del tipo `Address`. `Address` es un objeto de valor con varias propiedades, como `Street` y `City`. EF Core asigna el objeto `Order` a su tabla con una columna por propiedad `Address` y agrega el nombre de la propiedad como prefijo a cada nombre de columna. En este ejemplo, la tabla `Order` incluye

columnas como `ShipToAddress_Street` y `ShipToAddress_City`. También es posible almacenar los tipos de propiedad en tablas independientes, si es lo que se busca hacer.

Obtenga más información sobre la [compatibilidad con entidades en EF Core](#).

Conexiones resistentes

En ocasiones, los recursos externos como las bases de datos SQL pueden no estar disponibles. En casos de indisponibilidad temporal, las aplicaciones pueden usar lógica de reintento para evitar que se genere una excepción. Esta técnica se conoce normalmente como *resistencia de la conexión*. Se puede implementar una técnica de [reintento con retroceso exponencial propia](#) intentando el reintento con un tiempo de espera que aumenta exponencialmente, hasta que se alcance un número máximo de reintentos. Esta técnica se basa en el hecho de que es posible que los recursos en la nube no estén disponibles de forma intermitente durante breves períodos, lo que produciría un error en algunas solicitudes.

Para Azure SQL DB, Entity Framework Core ya proporciona la lógica de reintento y resistencia de conexión de base de datos interna. Pero debe habilitar la estrategia de ejecución de Entity Framework para cada conexión de DbContext si quiere tener conexiones resistentes de EF Core.

Por ejemplo, el código siguiente en el nivel de conexión de EF Core permite conexiones resistentes de SQL que se vuelven a intentar si se produce un error en la conexión.

C#

```
builder.Services.AddDbContext<OrderingContext>(options =>
{
    options.UseSqlServer(builder.Configuration["ConnectionString"],
        sqlServerOptionsAction: sqlOptions =>
        {
            sqlOptions.EnableRetryOnFailure(
                maxRetryCount: 5,
                maxRetryDelay: TimeSpan.FromSeconds(30),
                errorNumbersToAdd: null);
        }
    );
});
```

Estrategias de ejecución y transacciones explícitas mediante BeginTransaction y varios DbContexts

Cuando se habilitan los reintentos en las conexiones de EF Core, cada operación que se realiza mediante EF Core se convierte en su propia operación que se puede reintentar. Cada consulta y cada llamada a `SaveChangesAsync` se reintentará como una unidad si se produce un error transitorio.

Pero si el código inicia una transacción con `BeginTransaction`, va a definir un grupo de operaciones propio que se deben tratar como una unidad; todo dentro de la transacción se debe revertir si se produce un error. Verá una excepción similar a la siguiente si intenta ejecutar esa transacción cuando se usa una estrategia de ejecución de EF (directiva de reintentos) y se incluyen varias llamadas a `SaveChangesAsync` desde varios elementos `DbContext` en la transacción.

`System.InvalidOperationException`: la estrategia de ejecución configurada `SqlServerRetryingExecutionStrategy` no es compatible con las transacciones iniciadas por el usuario. Use la estrategia de ejecución que devuelve `DbContext.Database.CreateExecutionStrategy()` para ejecutar todas las operaciones en la transacción como una unidad que se puede reintentar.

La solución consiste en invocar manualmente la estrategia de ejecución de EF con un delegado que representa a todos los elementos que se deben ejecutar. Si se produce un error transitorio, la estrategia de ejecución vuelve a invocar al delegado. En el código siguiente se muestra cómo implementar este enfoque:

C#

```
// Use of an EF Core resiliency strategy when using multiple DbContexts
// within an explicit transaction
// See:
// https://learn.microsoft.com/ef/core/miscellaneous/connection-resiliency
var strategy = _catalogContext.Database.CreateExecutionStrategy();
await strategy.ExecuteAsync(async () =>
{
    // Achieving atomicity between original Catalog database operation and the
    // IntegrationEventLog thanks to a local transaction
    using (var transaction = _catalogContext.Database.BeginTransaction())
    {
        _catalogContext.CatalogItems.Update(catalogItem);
        await _catalogContext.SaveChangesAsync();

        // Save to EventLog only if product price changed
        if (raiseProductPriceChangedEvent)
        {
            await _integrationEventLogService.SaveEventAsync(priceChangedEvent);
            transaction.Commit();
        }
    }
});
```

El primer elemento DbContext es `_catalogContext` y el segundo elemento DbContext está dentro del objeto `_integrationEventLogService`. Por último, la acción Commit se realizará en varios DbContext mediante una estrategia de ejecución de EF.

Referencias: Entity Framework Core

- Documentación de EF Core <https://learn.microsoft.com/ef/>
- EF Core: datos relacionados <https://learn.microsoft.com/ef/core/querying/related-data>
- Evitar la carga diferida de entidades en aplicaciones ASP.NET <https://ardalis.com/avoid-lazy-loading-entities-in-asp-net-applications> ↗

¿EF Core o micro-ORM?

Aunque EF Core es una opción excelente para administrar la persistencia, y en su mayor parte encapsula los detalles de base de datos de los desarrolladores de aplicaciones, no es la única opción. Otra alternativa de código abierto popular es [Dapper](#) ↗, lo que se denomina un micro-ORM. Un micro-ORM es una herramienta ligera y menos completa para la asignación de objetos a estructuras de datos. En el caso de Dapper, sus objetivos de diseño se centran en el rendimiento, en lugar de encapsular totalmente las consultas subyacentes que usa para recuperar y actualizar los datos. Como no abstrae SQL del desarrollador, Dapper es "más cercano al sistema operativo" y permite a los desarrolladores escribir las consultas exactas que quieren usar para una operación de acceso a datos determinada.

EF Core proporciona dos características importantes que lo diferencian de Dapper, pero que también se agregan a su sobrecarga de rendimiento. La primera es la traducción de expresiones LINQ a SQL. Estas traducciones se almacenan en caché, pero aun así hay sobrecarga la primera vez que se realizan. La segunda es el seguimiento de los cambios en las entidades (para poder generar instrucciones de actualización eficaces). Este comportamiento se puede desactivar para consultas específicas mediante la extensión [AsNoTracking](#). EF Core también genera consultas SQL que suelen ser muy eficaces y en cualquier caso perfectamente aceptables desde la perspectiva del rendimiento, pero si se necesita un control preciso sobre la consulta específica que se va a ejecutar, también se puede pasar código SQL personalizado (o ejecutar un procedimiento almacenado) con EF Core. En este caso, Dapper también supera a EF Core, pero solo muy ligeramente. En [el sitio de Dapper](#) ↗ se pueden encontrar datos de pruebas comparativas de rendimiento actuales para una amplia variedad de métodos de acceso a datos.

Para ver cómo varía la sintaxis de Dapper con respecto a EF Core, tenga en cuenta estas dos versiones del mismo método para recuperar una lista de elementos:

C#

```
// EF Core
private readonly CatalogContext _context;
public async Task<IEnumerable<CatalogType>> GetCatalogTypes()
{
    return await _context.CatalogTypes.ToListAsync();
}

// Dapper
private readonly SqlConnection _conn;
public async Task<IEnumerable<CatalogType>> GetCatalogTypesWithDapper()
{
    return await _conn.QueryAsync<CatalogType>("SELECT * FROM CatalogType");
}
```

Si tiene que generar gráficos de objetos más complejos con Dapper, tendrá que escribir personalmente las consultas asociadas (en lugar de agregar un archivo de inclusión como haría en EF Core). Esta función se admite por medio de diversas sintaxis, incluida una característica denominada Asignación múltiple, que permite asignar filas individuales a varios objetos asignados. Por ejemplo, dada una clase Post con una propiedad Owner de tipo User, el código SQL siguiente devolvería todos los datos necesarios:

SQL

```
select * from #Posts p
left join #Users u on u.Id = p.OwnerId
Order by p.Id
```

Cada fila devuelta incluye los datos de User y Post. Como los datos de User se deben asociar a los datos de Post a través de su propiedad Owner, se usa la función siguiente:

C#

```
(post, user) => { post.Owner = user; return post; }
```

La lista de código completa para devolver una colección de entradas con su propiedad Owner rellena con los datos de usuario asociados sería esta:

C#

```
var sql = @"select * from #Posts p
left join #Users u on u.Id = p.OwnerId
Order by p.Id";
var data = connection.Query<Post, User, Post>(sql,
(post, user) => { post.Owner = user; return post;});
```

Como ofrece menos encapsulación, Dapper requiere que los desarrolladores tengan más información sobre cómo se almacenan los datos, cómo consultarlos de forma eficaz y escribir más código para recuperarlos. Cuando el modelo cambia, en lugar de crear simplemente una migración (otra característica de EF Core) o actualizar la información de asignación en una posición en un DbContext, se deben actualizar todas las consultas que se van a ver afectadas. Estas consultas no tienen garantías en tiempo de compilación, por lo que se pueden interrumpir en tiempo de ejecución en respuesta a cambios en el modelo o la base de datos, lo que dificulta más la detección rápida de los errores. A cambio de estos inconvenientes, Dapper ofrece un rendimiento extremadamente rápido.

Para la mayoría de las aplicaciones y la mayoría de los elementos de casi todas las aplicaciones, EF Core ofrece un rendimiento aceptable. Por tanto, sus ventajas de productividad para los desarrolladores suelen superar su sobrecarga de rendimiento. Para las consultas que se pueden beneficiar del almacenamiento en caché, es posible que solo se pueda ejecutar la consulta real en un pequeño porcentaje de los casos, lo que hace que las diferencias de rendimiento de las consultas de pequeño tamaño sean irrelevantes.

SQL o NoSQL

Tradicionalmente, las bases de datos relacionales como SQL Server han dominado el mercado del almacenamiento de datos persistentes, pero no son la única solución disponible. Las bases de datos NoSQL como [MongoDB](#) ofrecen un enfoque diferente para el almacenamiento de objetos. En lugar de asignar objetos a tablas y filas, otra opción consiste en serializar el gráfico de objetos completo y almacenar el resultado. Las ventajas de este enfoque, al menos inicialmente, son la simplicidad y el rendimiento. Es más simple almacenar un único objeto serializado con una clave que descomponerlo en muchas tablas con relaciones, actualizaciones y filas que pueden haber cambiado desde la última vez que se recuperó el objeto de la base de datos. Del mismo modo, la recuperación y deserialización de un único objeto de un almacén basado en claves suele ser mucho más rápida y sencilla que combinaciones complejas o varias consultas de base de datos necesarias para crear completamente el mismo objeto de una base de datos relacional. La falta de bloqueos o transacciones, o de un esquema fijo, también

hace que las bases de datos NoSQL sean sensibles al escalado entre varios equipos, admitiendo grandes conjuntos de datos.

Por otro lado, las bases de datos NoSQL (como normalmente se denominan) tienen sus desventajas. Las bases de datos relacionales usan la normalización para aplicar la coherencia y evitar la duplicación de datos. Este enfoque reduce el tamaño total de la base de datos y garantiza que las actualizaciones de los datos compartidos estén disponibles inmediatamente en toda la base de datos. En una base de datos relacional, es posible que una tabla de direcciones haga referencia a una tabla de país por el id., de forma que, si se cambiara el nombre de un país o región, los registros de direcciones se beneficiarían de la actualización sin tener que actualizarlos. Pero, en una base de datos NoSQL, es posible que la dirección y su región o país asociado se serialicen como parte de muchos objetos almacenados. Una actualización en un nombre de país o región requeriría actualizar todos esos objetos, en lugar de una sola fila. Las bases de datos relacionales también pueden asegurar la integridad relacional mediante la aplicación de reglas como claves externas. Normalmente, las bases de datos NoSQL no ofrecen estas restricciones en sus datos.

Otra complejidad que las bases de datos NoSQL deben superar es el control de versiones. Cuando cambian las propiedades de un objeto, es posible que no se pueda deserializar de las versiones anteriores que se almacenaron. Por tanto, todos los objetos existentes que tengan una versión serializada (anterior) del objeto tendrán que actualizarse para que se ajusten a su esquema nuevo. Conceptualmente, este enfoque no es diferente a una base de datos relacional, en la que los cambios del esquema a veces requieren scripts de actualización o asignación de actualizaciones. Pero el número de entradas que se deben modificar suele ser mucho mayor en el enfoque de NoSQL, porque hay más duplicación de los datos.

En las bases de datos NoSQL se pueden almacenar varias versiones de los objetos, algo que las bases de datos relacionales de esquema fijo normalmente no admiten. Pero, en este caso, el código de la aplicación deberá tener en cuenta la existencia de versiones anteriores de los objetos, lo que agrega complejidad adicional.

Las bases de datos NoSQL normalmente no aplican [ACID](#) , lo que significa que tienen ventajas de rendimiento y escalabilidad con respecto a las bases de datos relacionales. Son adecuadas para conjuntos de datos y objetos muy grandes que no son adecuados para el almacenamiento en estructuras de tabla normalizadas. No hay ninguna razón para que una aplicación no pueda aprovechar las ventajas de las bases de datos relacionales y NoSQL, y usar cada una cuando sea más adecuado.

Azure Cosmos DB

Azure Cosmos DB es un servicio de base de datos NoSQL completamente administrado que ofrece almacenamiento de datos sin esquema basado en la nube. Azure Cosmos DB se ha creado para rendimiento rápido y predecible, alta disponibilidad, escalado elástico y distribución global. A pesar de ser una base de datos NoSQL, los desarrolladores pueden usar funciones de consulta SQL enriquecidas y conocidas en los datos JSON. Todos los recursos de Azure Cosmos DB se almacenan como documentos JSON. Los recursos se administran como *elementos*, que son documentos que contienen metadatos, y *fuentes*, que son colecciones de elementos. En la figura 8-2 se muestra la relación entre los diferentes recursos de Azure Cosmos DB.

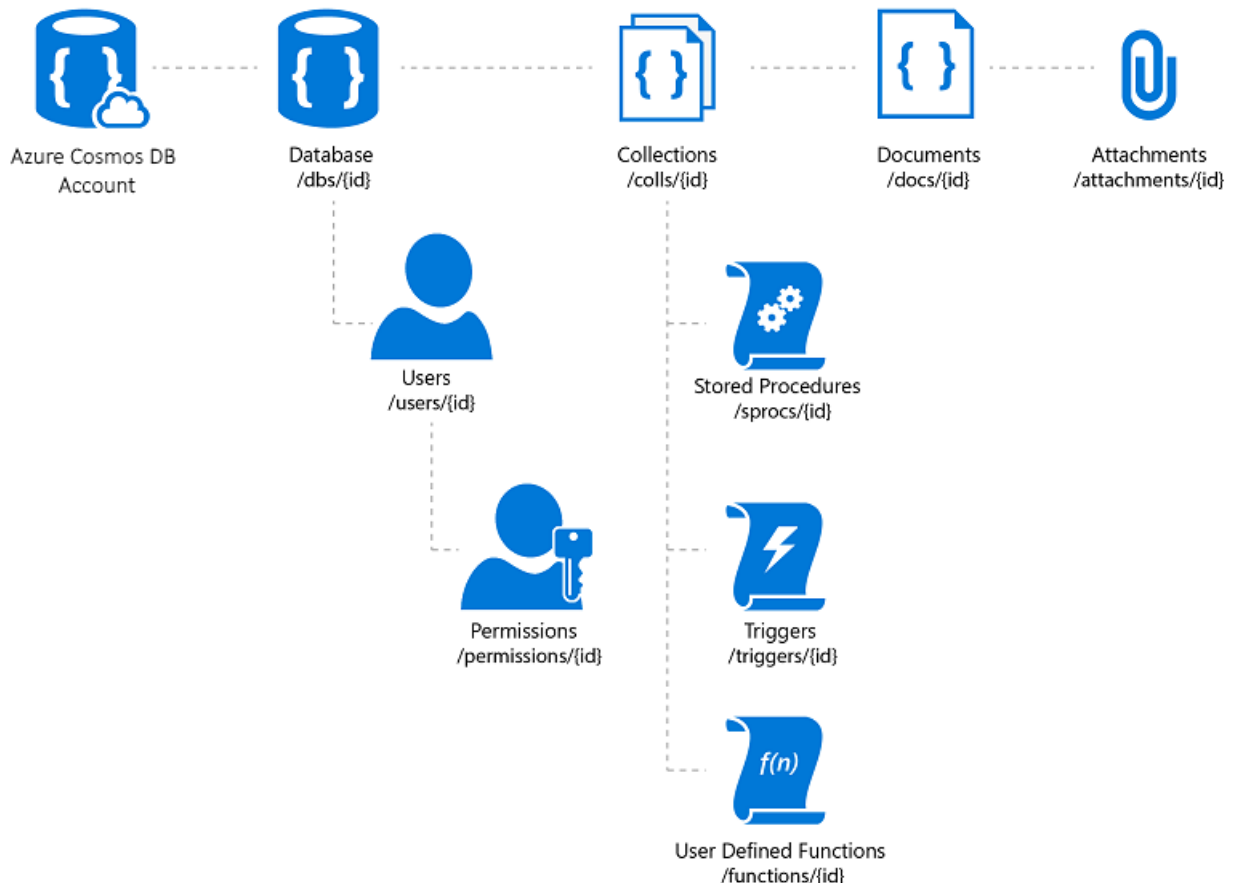


Figura 8-2. Organización de recursos de Azure Cosmos DB.

El lenguaje de consulta de Azure Cosmos DB es una interfaz sencilla, pero eficaz, para consultar documentos JSON. El lenguaje admite un subconjunto de la gramática SQL ANSI y agrega integración profunda de matrices de objetos JavaScript, construcción de objetos e invocación de funciones.

Referencias: Azure Cosmos DB

- Introducción a Azure Cosmos DB <https://learn.microsoft.com/azure/cosmos-db/introduction>

Otras opciones de persistencia

Además de las opciones de almacenamiento relacionales y NoSQL, las aplicaciones ASP.NET Core pueden usar Azure Storage para almacenar varios formatos de datos y archivos de forma escalable y basada en la nube. Azure Storage es escalable de forma masiva, para que pueda empezar a almacenar pequeñas cantidades de datos y escalar verticalmente hasta almacenar cientos o terabytes si así lo requiere la aplicación. Azure Storage admite cuatro tipos de datos:

- Blob Storage para almacenamiento de texto binario no estructurado, que también se denomina almacenamiento de objetos.
- Table Storage para conjuntos de datos estructurados, accesible a través de claves de fila.
- Queue Storage para la mensajería confiable basada en colas.
- File Storage para el acceso a archivos compartido entre máquinas virtuales de Azure y aplicaciones locales.

Referencias: Azure Storage

- Introducción a Azure Storage <https://learn.microsoft.com/azure/storage/common/storage-introduction>

Almacenamiento en memoria caché

En las aplicaciones web, cada solicitud web se debe completar en el menor tiempo posible. Una manera de lograr esta función consiste en limitar el número de llamadas externas que el servidor debe realizar para completar la solicitud. El almacenamiento en caché implica almacenar una copia de los datos en el servidor (u otro almacén de datos que sea más fácil de consultar que el origen de los datos). Las aplicaciones web y especialmente las aplicaciones web tradicionales que no son de SPA, necesitan generar la interfaz de usuario completa con cada solicitud. Con frecuencia, este enfoque implica realizar muchas de las mismas consultas de base de datos varias veces de una solicitud de usuario a la siguiente. En la mayoría de los casos, estos datos apenas cambian, de modo que no hay motivos para solicitarlos constantemente de la base de datos. ASP.NET Core admite el almacenamiento de respuestas en caché, para almacenar en caché páginas completas y el almacenamiento de datos en caché, que admite un comportamiento de almacenamiento en caché más granular.

Al implementar el almacenamiento en caché, es importante tener en cuenta la separación de intereses. Evite implementar la lógica de almacenamiento en caché en la lógica de acceso a datos o en la interfaz de usuario. En su lugar, encapsule el

almacenamiento en caché en sus propias clases y use la configuración para administrar su comportamiento. Este enfoque sigue los principios de Abierto o cerrado y Responsabilidad única, y facilitará la administración del uso del almacenamiento en caché en la aplicación a medida que crezca.

Almacenamiento en caché de respuestas de ASP.NET Core

ASP.NET Core admite dos niveles de almacenamiento en caché de respuestas. El primer nivel no almacena en caché nada en el servidor, pero agrega encabezados HTTP que indican a los clientes y servidores proxy que almacenen las respuestas en caché. Esta función se implementa mediante la incorporación del atributo `ResponseCache` a controladores o acciones individuales:

C#

```
[ResponseCache(Duration = 60)]
public IActionResult Contact()
{
    ViewData["Message"] = "Your contact page.";
    return View();
}
```

Como resultado del ejemplo anterior, el encabezado siguiente se agrega a la respuesta, indicando a los clientes que almacenen el resultado en caché hasta 60 segundos.

HTML

```
Cache-Control: public,max-age=60
```

Con el fin de agregar almacenamiento en caché en memoria del lado servidor a la aplicación, debe hacer referencia al paquete NuGet

`Microsoft.AspNetCore.ResponseCaching` y, después, agregar el middleware de almacenamiento en caché de las respuestas. Este middleware se configura con servicios y middleware durante el inicio de la aplicación:

C#

```
builder.Services.AddResponseCaching();

// other code omitted, including building the app

app.UseResponseCaching();
```

El software intermedio de almacenamiento de las respuestas en caché almacenará automáticamente las respuestas en caché en función de un conjunto de condiciones que se pueden personalizar. De forma predeterminada, solo se almacenan en caché las respuestas 200 (OK) solicitadas a través de métodos GET o HEAD. Además, las solicitudes deben tener una respuesta con un encabezado público Cache-Control: y no pueden incluir encabezados Authorization o Set-Cookie. Vea una [lista completa de las condiciones de almacenamiento en caché que usa el software intermedio de almacenamiento en caché de las respuestas](#).

Almacenamiento de datos en caché

En lugar (o además de) almacenar en caché las respuestas web completas, se pueden almacenar en caché los resultados de consultas de datos individuales. Para esta función, se puede usar el almacenamiento en caché en memoria en el servidor web o [una caché distribuida](#). En esta sección se muestra cómo implementar el almacenamiento en caché en memoria.

Agregue compatibilidad para el almacenamiento en caché en memoria (o distribuido) con el código siguiente:

C#

```
builder.Services.AddMemoryCache();  
builder.Services.AddMvc();
```

Asegúrese de agregar también el paquete NuGet `Microsoft.Extensions.Caching.Memory`.

Una vez agregado el servicio, se solicita `IMemoryCache` a través de la inserción de dependencias siempre que haya que obtener acceso a la caché. En este ejemplo, `CachedCatalogService` usa el patrón de diseño de Proxy, o Decorator, proporcionando una implementación alternativa de `ICatalogService` que controla el acceso a la implementación de `CatalogService` subyacente, o bien le agrega comportamiento.

C#

```
public class CachedCatalogService : ICatalogService  
{  
    private readonly IMemoryCache _cache;  
    private readonly CatalogService _catalogService;  
    private static readonly string _brandsKey = "brands";  
    private static readonly string _typesKey = "types";  
    private static readonly TimeSpan _defaultCacheDuration =  
        TimeSpan.FromSeconds(30);
```

```

public CachedCatalogService(
    IMemoryCache cache,
    CatalogService catalogService)
{
    _cache = cache;
    _catalogService = catalogService;
}

public async Task<IEnumerable<SelectListItem>> GetBrands()
{
    return await _cache.GetOrCreateAsync(_brandsKey, async entry =>
    {
        entry.SlidingExpiration = _defaultCacheDuration;
        return await _catalogService.GetBrands();
    });
}

public async Task<Catalog> GetCatalogItems(int pageIndex, int itemsPage,
int? brandID, int? typeId)
{
    string cacheKey = $"items-{pageIndex}-{itemsPage}-{brandID}-{typeId}";
    return await _cache.GetOrCreateAsync(cacheKey, async entry =>
    {
        entry.SlidingExpiration = _defaultCacheDuration;
        return await _catalogService.GetCatalogItems(pageIndex, itemsPage,
brandID, typeId);
    });
}

public async Task<IEnumerable<SelectListItem>> GetTypes()
{
    return await _cache.GetOrCreateAsync(_typesKey, async entry =>
    {
        entry.SlidingExpiration = _defaultCacheDuration;
        return await _catalogService.GetTypes();
    });
}
}

```

Para configurar la aplicación para usar la versión en caché del servicio, pero seguir permitiendo que el servicio obtenga la instancia de `CatalogService` que necesita en su constructor, se deben agregar las líneas siguientes en *Program.cs*:

C#

```

builder.Services.AddMemoryCache();
builder.Services.AddScoped<ICatalogService, CachedCatalogService>();
builder.Services.AddScoped<CatalogService>();

```

Después de agregar este código, las llamadas de base de datos para recuperar los cambios de los datos del catálogo solo se realizarán una vez por minuto, en lugar de en

cada solicitud. Según el tráfico al sitio, esto puede tener un impacto significativo en el número de consultas realizadas a la base de datos y el tiempo medio de carga de la página principal que depende actualmente de las tres consultas expuestas por este servicio.

Un problema que surge cuando se implementa el almacenamiento en caché es el de los *datos obsoletos*, es decir, datos que han cambiado en el origen, pero de los que permanece en caché una versión obsoleta. Una manera sencilla de mitigar esta incidencia consiste en usar duraciones de caché pequeñas, ya que, para una aplicación ocupada, la ventaja adicional de extender la longitud de los datos en caché es limitada. Por ejemplo, considere una página que realiza una única consulta de base de datos y se solicita 10 veces por segundo. Si esta página se almacena en caché durante un minuto, el número de consultas de base de datos realizadas por minuto descenderá 600 a 1, una reducción del 99,8 %. Si, en su lugar, la duración de la caché fuera de una hora, la reducción general sería del 99,997 %, pero ahora la probabilidad y la antigüedad posible de los datos obsoletos aumentan considerablemente.

Otro enfoque consiste en quitar proactivamente las entradas de caché cuando se actualizan los datos que contienen. Cualquier entrada individual se puede quitar si se conoce su clave:

C#

```
_cache.Remove(cacheKey);
```

Si la aplicación expone funcionalidad para actualizar las entradas que almacena en caché, puede quitar las entradas de caché correspondientes en el código que realiza las actualizaciones. En ocasiones puede haber muchas otras entradas que dependen de un conjunto de datos determinado. En ese caso, puede ser útil crear dependencias entre las entradas de caché, mediante el uso de un `CancellationToken`. Con un `CancellationToken` se pueden expirar varias entradas de caché a la vez si se cancela el token.

C#

```
// configure CancellationToken and add entry to cache
var cts = new CancellationTokenSource();
_cache.Set("cts", cts);
_cache.Set(cacheKey, itemToCache, new CancellationToken(cts.Token));

// elsewhere, expire the cache by cancelling the token\
_cache.Get<CancellationTokenSource>("cts").Cancel();
```

El almacenamiento en caché puede mejorar considerablemente el rendimiento de las páginas web que solicitan repetidamente los mismos valores de la base de datos. Asegúrese de medir el acceso de datos y el rendimiento de la página antes de aplicar el almacenamiento en caché y aplíquelo solo donde vea una necesidad de mejora. El almacenamiento en caché consume recursos de memoria de servidor web y aumenta la complejidad de la aplicación, por lo que es importante que no optimice de forma prematura con esta técnica.

Obtención de datos para aplicaciones de BlazorWebAssembly

Si va a compilar aplicaciones que usan Blazor Server, puede usar Entity Framework y otras tecnologías de acceso directo a datos como se han analizado hasta ahora en este capítulo. Sin embargo, al compilar aplicaciones de BlazorWebAssembly, al igual que otros marcos de SPA, necesitará una estrategia diferente para acceder a los datos. Normalmente, estas aplicaciones acceden a los datos e interactúan con el servidor a través de puntos de conexión de API web.

Si los datos o las operaciones que se realizan son confidenciales, asegúrese de revisar la sección sobre seguridad del [capítulo anterior](#) y de proteger las API frente al acceso no autorizado.

Encontrará un ejemplo de una aplicación de BlazorWebAssembly en la [aplicación de referencia eShopOnWeb](#) en el proyecto BlazorAdmin. Este proyecto se hospeda dentro del proyecto web eShopOnWeb y permite a los usuarios del grupo de administradores administrar los elementos del almacén. Puede ver una captura de pantalla de la aplicación en la figura 8-3.

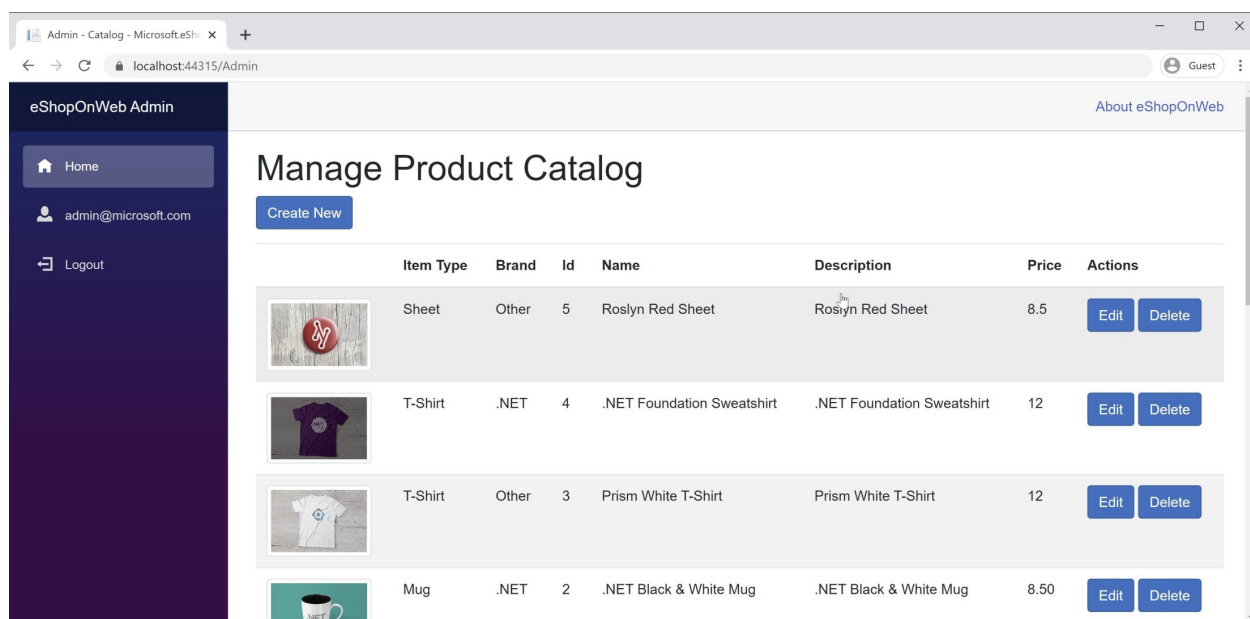


Figura 8-3. Captura de pantalla de administrador del catálogo de eShopOnWeb.

A la hora de obtener datos de las API web dentro de una aplicación de BlazorWebAssembly, simplemente use una instancia de `HttpClient` como haría en cualquier aplicación .NET. Los pasos básicos que hay que seguir son crear la solicitud de envío (si es necesario, normalmente para las solicitudes POST o PUT), esperar a que se realice la solicitud, comprobar el código de estado y deserializar la respuesta. Si va a hacer muchas solicitudes a un determinado conjunto de API, es aconsejable encapsular las API y configurar la dirección base de `HttpClient` de forma centralizada. De este modo, si necesita ajustar cualquiera de estas configuraciones en varios entornos, puede realizar los cambios en un solo lugar. Debe agregar compatibilidad con este servicio en `Program.Main`:

C#

```
builder.Services.AddScoped(sp => new HttpClient
{
    BaseAddress = new Uri(builder.HostEnvironment.BaseAddress)
});
```

Si necesita acceder a los servicios de forma segura, acceda a un token seguro y configure `HttpClient` para pasar este token como un encabezado de autenticación con cada solicitud:

C#

```
_httpClient.DefaultRequestHeaders.Authorization =
    new AuthenticationHeaderValue("Bearer", token);
```

Esta actividad se puede hacer en cualquier componente que tenga `HttpClient` insertado, siempre que `HttpClient` no se haya agregado a los servicios de la aplicación con una duración `Transient`. Todas las referencias a `HttpClient` en la aplicación hacen referencia a la misma instancia, por lo que los cambios en un componente se aplican en toda la aplicación. Un buen lugar para realizar esta comprobación de autenticación (y después, la especificación del token) es en un componente compartido, como la navegación principal del sitio. Obtenga más información sobre este enfoque en el proyecto `BlazorAdmin` en la [aplicación de referencia eShopOnWeb](#).

Uno de los beneficios de BlazorWebAssembly frente a las SPA tradicionales de JavaScript es que no es necesario conservar sincronizadas las copias de los objetos de transferencia de datos (DTO). Su proyecto de BlazorWebAssembly y de API web pueden compartir el mismo objeto de transferencia de datos en un proyecto común

compartido. Este enfoque elimina parte de la fricción que conlleva el desarrollo de aplicaciones de página única.

Para obtener datos rápidamente de un punto de conexión de API, puede usar el método auxiliar integrado `GetFromJsonAsync`. Hay métodos similares para POST, PUT, etc. A continuación, se muestra cómo obtener una clase `CatalogItem` de un punto de conexión de API mediante un elemento `HttpClient` configurado en una aplicación de BlazorWebAssembly:

C#

```
var item = await _httpClient.GetFromJsonAsync<CatalogItem>($"catalog-items/{id}");
```

Una vez que tenga los datos que necesita, normalmente hará un seguimiento de los cambios de forma local. Para realizar actualizaciones en el almacén de datos de back-end, llamará a API web adicionales.

Referencias: datos de Blazor

- Llamada a una API web desde Blazor <https://learn.microsoft.com/aspnet/core/blazor/call-web-api> de ASP.NET Core

[Anterior](#)[Siguiente](#)

Prueba de aplicaciones ASP.NET Core MVC

18/06/2025

💡 Sugerencia

Este contenido es un extracto del libro electrónico, Arquitecto de aplicaciones web modernas con ASP.NET Core y Azure, disponible en [.NET Docs](#) o como un PDF descargable gratuito que se puede leer sin conexión.

Descargar PDF



"Si no le gustan las pruebas unitarias del producto, lo más probable es que a los clientes no les guste probarlo, tampoco". _Anónimo-

El software de cualquier complejidad puede producir errores de maneras inesperadas en respuesta a los cambios. Por lo tanto, las pruebas después de realizar cambios son necesarias para todas las aplicaciones más triviales (o menos críticas). Las pruebas manuales son la manera más lenta, menos confiable y costosa de probar software. Desafortunadamente, si las aplicaciones no están diseñadas para ser testeables, puede ser el único medio de prueba disponible. Las aplicaciones escritas para seguir los principios arquitectónicos establecidos en el [capítulo 4](#) deben ser en gran medida susceptibles de pruebas unitarias. ASP.NET aplicaciones core admiten la integración automatizada y las pruebas funcionales.

Tipos de pruebas automatizadas

Hay muchos tipos de pruebas automatizadas para aplicaciones de software. La prueba de nivel más simple y inferior es la prueba unitaria. En un nivel ligeramente superior, hay pruebas de integración y pruebas funcionales. Otros tipos de pruebas, como pruebas de IU, pruebas de carga, pruebas de esfuerzo y pruebas de humo, están fuera del ámbito de este documento.

Pruebas unitarias

Una prueba unitaria prueba una sola parte de la lógica de la aplicación. Uno puede describirlo más enumerando algunas de las cosas que no es. Una prueba unitaria no prueba cómo funciona el código con dependencias o infraestructura, es decir, qué son las pruebas de integración. Una prueba unitaria no prueba el marco en el que se escribe el código: debe suponer que funciona o, si lo encuentra, registre un error y codifique una solución alternativa. Una prueba unitaria se ejecuta completamente en memoria y en proceso. No se comunica con el sistema de archivos, la red ni una base de datos. Las pruebas unitarias solo deben probar el código.

Las pruebas unitarias, en virtud del hecho de que solo prueban una sola unidad del código, sin dependencias externas, deben ejecutarse muy rápido. Por lo tanto, debería poder ejecutar conjuntos de pruebas de cientos de pruebas unitarias en unos segundos. Ejecútelos con frecuencia, idealmente antes de cada inserción en un repositorio de control de código fuente compartido y, ciertamente, con cada compilación automatizada en el servidor de compilación.

Pruebas de integración

Aunque es una buena idea encapsular el código que interactúa con infraestructura como bases de datos y sistemas de archivos, seguirá teniendo parte de ese código y probablemente quiera probarlo. Además, debe comprobar que las capas del código interactúan según lo esperado cuando las dependencias de la aplicación se resuelven por completo. Esta funcionalidad es responsabilidad de las pruebas de integración. Las pruebas de integración tienden a ser más lentas y más difíciles de configurar que las pruebas unitarias, ya que a menudo dependen de dependencias externas e infraestructura. Por lo tanto, debe evitar probar las cosas que se podrían probar con pruebas unitarias en las pruebas de integración. Si puede probar un escenario determinado con una prueba unitaria, debe probarlo con una prueba unitaria. Si no es posible, considere la posibilidad de usar una prueba de integración.

Las pruebas de integración a menudo tendrán procedimientos de configuración y desmontaje más complejos que las pruebas unitarias. Por ejemplo, una prueba de integración que va en contra de una base de datos real necesitará una manera de devolver la base de datos a un estado conocido antes de cada ejecución de prueba. A medida que se agregan nuevas pruebas y el esquema de base de datos de producción evoluciona, estos scripts de prueba tienden a crecer en tamaño y complejidad. En muchos sistemas grandes, no es práctico ejecutar conjuntos completos de pruebas de integración en estaciones de trabajo de desarrollador antes de comprobar los cambios en el control de código fuente compartido. En estos casos, las pruebas de integración se pueden ejecutar en un servidor de compilación.

Pruebas funcionales

Las pruebas de integración se escriben desde la perspectiva del desarrollador para comprobar que algunos componentes del sistema funcionan correctamente juntos. Las pruebas funcionales se escriben desde la perspectiva del usuario y comprueban la exactitud del sistema en función de sus requisitos. El siguiente extracto ofrece una analogía útil para pensar en las pruebas funcionales, en comparación con las pruebas unitarias:

"Muchas veces el desarrollo de un sistema es similar al edificio de una casa. Aunque esta analogía no es bastante correcta, podemos ampliarla para comprender la diferencia entre las pruebas unitarias y funcionales. Las pruebas unitarias son análogas a un inspector de edificio que visita el sitio de construcción de una casa. Está enfocado en los diversos sistemas internos de la casa: los cimientos, la estructura, la electricidad, la fontanería, etc. Asegura (pruebas) que las partes de la casa funcionarán correctamente y de forma segura, es decir, cumplan con el código de construcción. Las pruebas funcionales en este escenario son análogas al propietario que visita este mismo sitio de construcción. Asume que los sistemas internos se comportarán adecuadamente, que el inspector de edificio está realizando su tarea. El propietario se centra en lo que será vivir en esta casa. Está preocupado por cómo se ve la casa, si las distintas habitaciones tienen un tamaño cómodo, si la casa se ajusta a las necesidades de la familia, y si las ventanas están en un buen lugar para atrapar el sol de la mañana. El propietario está realizando pruebas funcionales en la casa. Tiene la perspectiva del usuario. El inspector municipal está realizando pruebas unitarias en la casa. Tiene la perspectiva del constructor".

Origen: [Pruebas unitarias frente a pruebas funcionales](#) 

Me encanta decir "Como desarrolladores, fallamos de dos maneras: creamos lo malo o creamos lo incorrecto". Las pruebas unitarias garantizan que está construyendo lo correcto; Las pruebas funcionales garantizan que está creando lo correcto.

Dado que las pruebas funcionales funcionan en el nivel del sistema, pueden requerir cierto grado de automatización de la interfaz de usuario. Al igual que las pruebas de integración, normalmente funcionan con algún tipo de infraestructura de prueba. Esta actividad hace que sean más lentas y frágiles que las pruebas unitarias y de integración. Solo debe tener tantas pruebas funcionales como necesite para estar seguro de que el sistema se comporta según lo previsto por los usuarios.

Pirámide de pruebas

Martin Fowler escribió sobre la pirámide de pruebas, un ejemplo del cual se muestra en la figura 9-1.

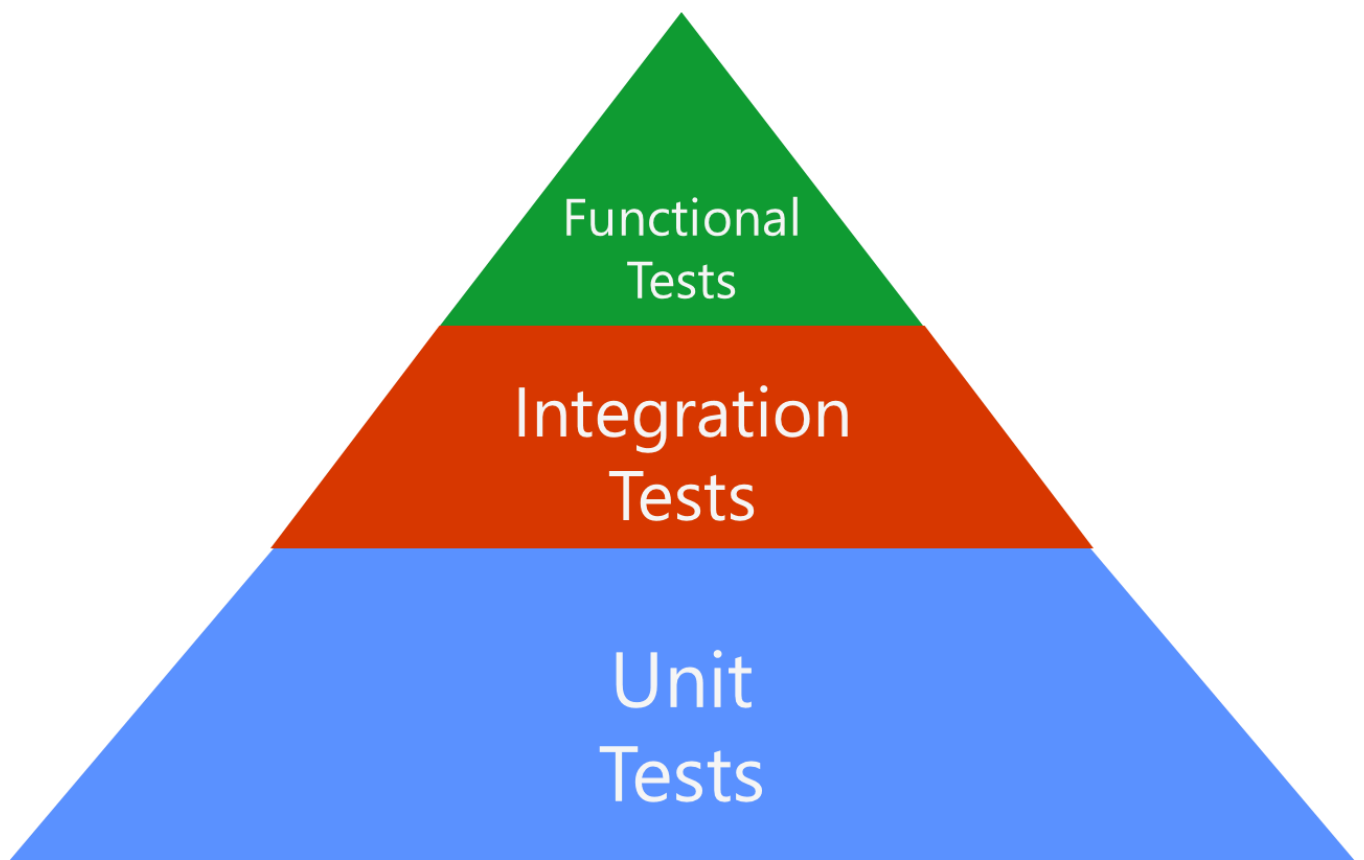


Figura 9-1. Pirámide de Pruebas

Las diferentes capas de la pirámide, y sus tamaños relativos, representan diferentes tipos de pruebas y cuántos debe escribir para la aplicación. Como puede ver, la recomendación es tener una gran base de pruebas unitarias, compatible con una capa más pequeña de pruebas de integración, con una capa aún más pequeña de pruebas funcionales. Idealmente, cada capa solo debe tener pruebas en ella que no se puedan realizar adecuadamente en una capa inferior. Tenga en cuenta la pirámide de pruebas cuando intente decidir qué tipo de prueba necesita para un escenario determinado.

Qué probar

Un problema común para los desarrolladores que no tienen experiencia al escribir pruebas automatizadas es decidir qué probar. Un buen punto de partida es probar la lógica condicional. En cualquier lugar que tenga un método con comportamiento que cambie en función de una instrucción condicional (if-else, switch, etc.), debería poder presentar al menos un par de pruebas que confirmen el comportamiento correcto para determinadas condiciones. Si el código tiene condiciones de error, es conveniente escribir al menos una prueba para la "ruta de acceso feliz" a través del código (sin errores) y al menos una prueba para la "ruta de acceso triste" (con errores o resultados inusuales) para confirmar que la aplicación se comporta como se esperaba en el caso de errores. Por último, intente centrarse en las pruebas que pueden producir errores, en lugar de centrarse en métricas como la cobertura de código. Por lo general, es mejor tener más cobertura de código que menos. Sin embargo, escribir algunas

pruebas más de un método complejo y crítico para la empresa suele ser un mejor uso del tiempo que escribir pruebas para propiedades automáticas solo para mejorar las métricas de cobertura de código de prueba.

Organización de proyectos de prueba

Los proyectos de prueba se pueden organizar de la manera que mejor funcione para usted. Es recomendable separar las pruebas por tipo (prueba unitaria, prueba de integración) y por lo que están probando (por proyecto, por espacio de nombres). Si esta separación consta de carpetas dentro de un único proyecto de prueba o de varios proyectos de prueba, es una decisión de diseño. Un proyecto es más sencillo, pero para proyectos grandes con muchas pruebas o para ejecutar más fácilmente diferentes conjuntos de pruebas, es posible que desee tener varios proyectos de prueba diferentes. Muchos equipos organizan proyectos de prueba basados en el proyecto que están probando, que para las aplicaciones con más de algunos proyectos pueden dar lugar a un gran número de proyectos de prueba, especialmente si sigue desglosando estos de acuerdo con qué tipo de pruebas se encuentran en cada proyecto. Un enfoque de compromiso es tener un proyecto por tipo de prueba, por aplicación, con carpetas dentro de los proyectos de prueba para indicar el proyecto (y la clase) que se está probando.

Un enfoque común consiste en organizar los proyectos de aplicación en una carpeta "src" y los proyectos de prueba de la aplicación en una carpeta "tests" paralela. Puede crear carpetas de soluciones coincidentes en Visual Studio si encuentra útil esta organización.

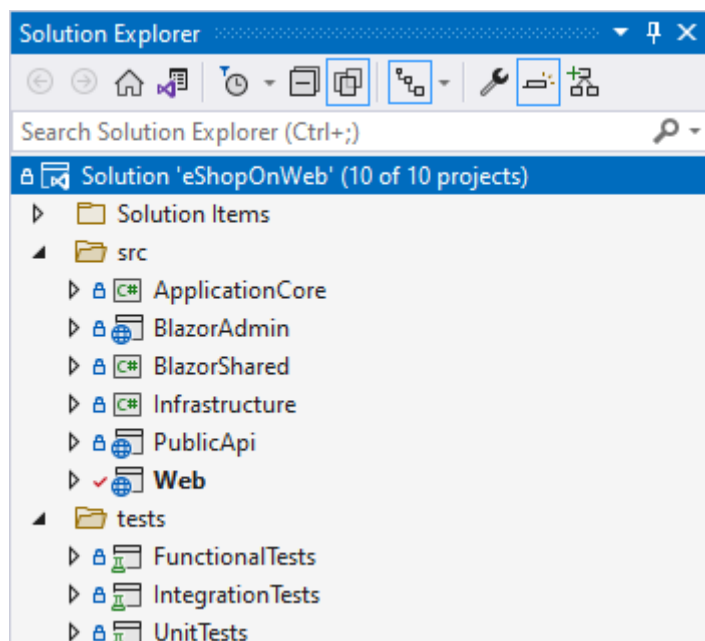


Figura 9-2. Prueba de la organización en la solución

Puede usar el marco de pruebas que prefiera. El marco de trabajo xUnit funciona bien y es en el que están escritas todas las pruebas de ASP.NET Core y EF Core. Puede agregar un proyecto

de prueba de xUnit en Visual Studio mediante la plantilla que se muestra en la figura 9-3 o desde la CLI mediante `dotnet new xunit`.

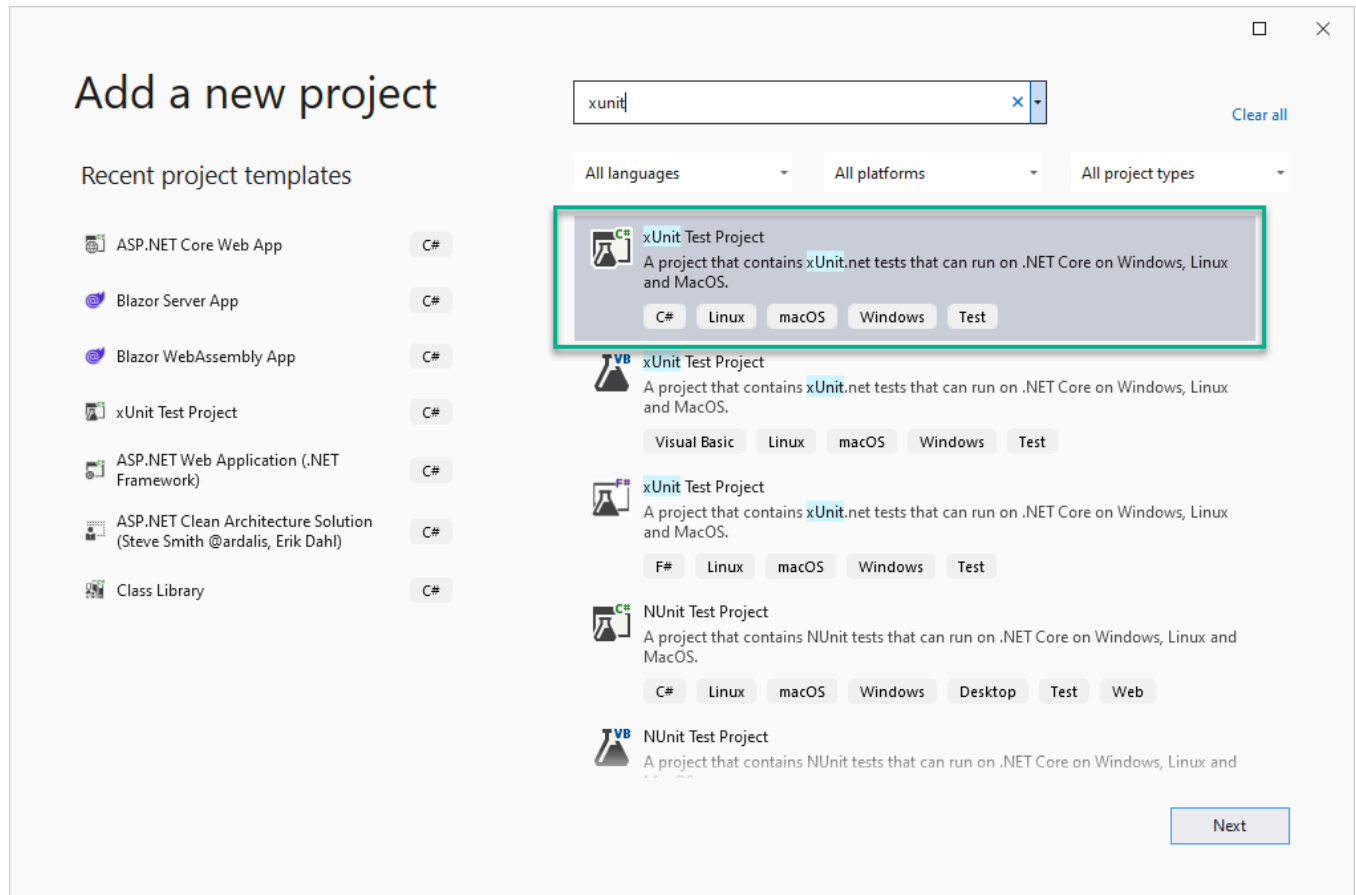


Figura 9-3. Agregar un proyecto de prueba de xUnit en Visual Studio

Nombres de pruebas

Asigne un nombre a las pruebas de forma coherente, con nombres que indiquen lo que hace cada prueba. Un enfoque con el que he tenido gran éxito es asignar un nombre a las clases de prueba según la clase y el método que están probando. Este enfoque resulta en muchas clases de prueba pequeñas, pero hace que quede extremadamente claro de qué es responsable cada prueba. Con el nombre de clase de prueba configurado, para identificar la clase y el método que se va a probar, se puede usar el nombre del método de prueba para especificar el comportamiento que se está probando. Este nombre debe incluir el comportamiento esperado y las entradas o suposiciones que deben producir este comportamiento. Algunos nombres de prueba de ejemplo:

- `CatalogControllerGetImage.CallsImageServiceWithId`
- `CatalogControllerGetImage.LogsWarningGivenImageMissingException`
- `CatalogControllerGetImage.ReturnsFileResultWithBytesGivenSuccess`
- `CatalogControllerGetImage.ReturnsNotFoundResultGivenImageMissingException`

Una variante de este enfoque finaliza cada nombre de clase de prueba con "Should" (Debería) y modifica ligeramente el tiempo:

- `CatalogControllerGetImage` **Should** . **Call** `ImageServiceWithId`
- `CatalogControllerGetImage` **Debería** . **Registrar** `WarningGivenImageMissingException`

Para algunos equipos el segundo método de nomenclatura es más claro, aunque un poco más detallado. En cualquier caso, intente usar una convención de nomenclatura que proporcione información sobre el comportamiento de las pruebas, de modo que cuando se produzca un error en una o varias pruebas, es obvio de sus nombres qué casos han fallado. Evite asignar nombres a las pruebas vagamente, como `ControllerTests.Test1`, ya que estos nombres no ofrecen ningún valor cuando los ve en los resultados de la prueba.

Si sigue una convención de nomenclatura como la anterior que genera muchas clases de prueba pequeñas, es recomendable organizar aún más las pruebas mediante carpetas y espacios de nombres. En la figura 9-4 se muestra un enfoque para organizar las pruebas por carpeta dentro de varios proyectos de prueba.

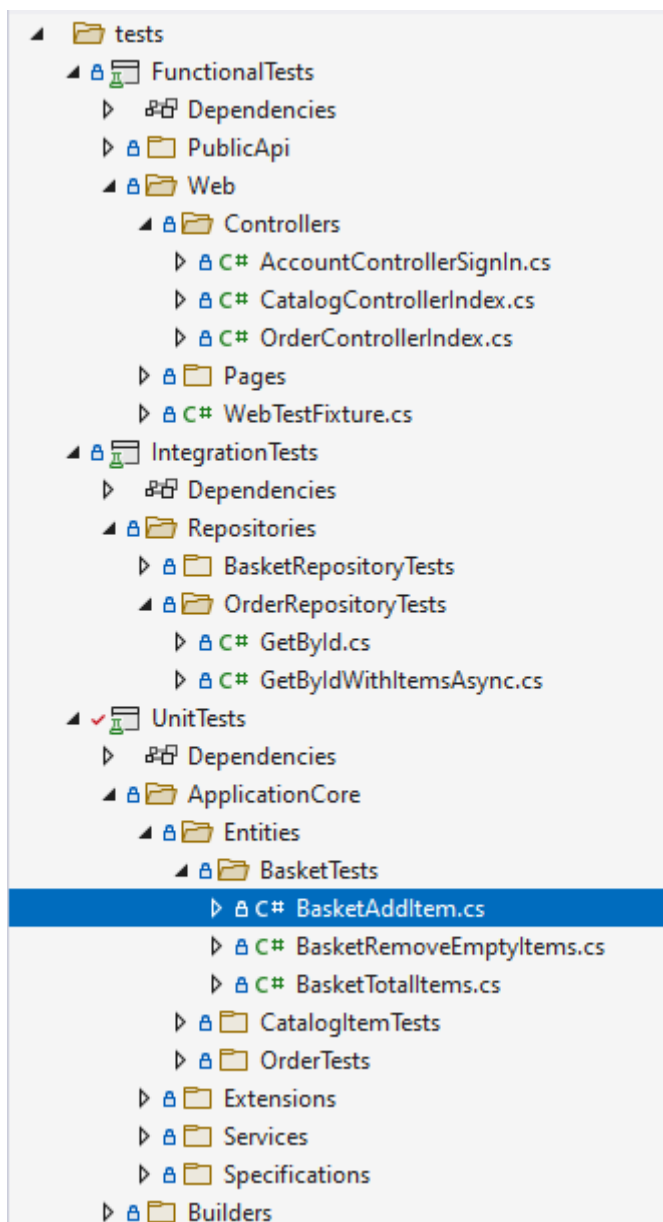


Figura 9-4. Organización de clases de prueba por carpeta en función de la clase que se está probando.

Si una clase de aplicación determinada tiene muchos métodos que se están probando (y, por tanto, muchas clases de prueba), puede tener sentido colocar estas clases en una carpeta correspondiente a la clase de aplicación. Esta organización no es diferente de cómo puede organizar archivos en carpetas en otro lugar. Si tiene más de tres o cuatro archivos relacionados en una carpeta que contiene muchos otros archivos, a menudo resulta útil moverlos a su propia subcarpeta.

Pruebas unitarias de aplicaciones de ASP.NET Core

En una aplicación ASP.NET Core bien diseñada, la mayor parte de la complejidad y la lógica de negocios se encapsularán en entidades empresariales y una variedad de servicios. La aplicación ASP.NET Core MVC, con sus controladores, filtros, modelos de vista y vistas, debe requerir pocas pruebas unitarias. Gran parte de la funcionalidad de una acción determinada se

encuentra fuera del propio método de acción. Las pruebas de si el enrutamiento o el control global de errores funcionan correctamente no se pueden realizar de forma eficaz con una prueba unitaria. Del mismo modo, los filtros, incluidos los filtros de validación del modelo y de autenticación y autorización, no se pueden probar de manera unitaria con una prueba dirigida al método de acción de un controlador. Sin estos orígenes de comportamiento, la mayoría de los métodos de acción deben ser trivialmente pequeños y delegar la mayor parte de su trabajo en los servicios que se pueden probar independientemente del controlador que los use.

En ocasiones tendrá que refactorizar el código para poder realizar pruebas unitarias en él. Con frecuencia, esta actividad implica identificar abstracciones y usar la inserción de dependencias para acceder a la abstracción en el código que desea probar, en lugar de codificar directamente en la infraestructura. Por ejemplo, considere este método de acción fácil para mostrar imágenes:

C#

```
[HttpGet("[controller]/pic/{id}")]
public IActionResult GetImage(int id)
{
    var contentRoot = _env.ContentRootPath + "//Pics";
    var path = Path.Combine(contentRoot, id + ".png");
    Byte[] b = System.IO.File.ReadAllBytes(path);
    return File(b, "image/png");
}
```

La prueba unitaria de este método es difícil por su dependencia directa de `System.IO.File`, que usa para leer desde el sistema de archivos. Puede probar este comportamiento para asegurarse de que funciona según lo previsto, pero hacerlo con archivos reales es una prueba de integración. Vale la pena tener en cuenta que no se puede probar unitariamente la ruta de este método; verá cómo realizar esta prueba con una prueba funcional en breve.

Si no puede probar directamente el comportamiento del sistema de archivos y no puede probar la ruta, ¿qué hay que probar? Bueno, después de refactorizar para que las pruebas unitarias sean posibles, puede detectar algunos casos de prueba y el comportamiento que falta, como el control de errores. ¿Qué hace el método cuando no se encuentra un archivo? ¿Qué debería hacer? En este ejemplo, el método refactorizado tiene este aspecto:

C#

```
[HttpGet("[controller]/pic/{id}")]
public IActionResult GetImage(int id)
{
    byte[] imageBytes;
    try
    {
        imageBytes = _imageService.GetImageBytesById(id);
    }
}
```

```
}  
catch (CatalogImageMissingException ex)  
{  
    _logger.LogWarning($"No image found for id: {id}");  
    return NotFound();  
}  
return File(imageBytes, "image/png");  
}
```

`_logger` y `_imageService` se insertan como dependencias. Ahora puede probar que el mismo identificador que se pasa al método de acción se pasa a `_imageService` y que los bytes resultantes se devuelven como parte de `FileResult`. También puede probar que el registro de errores se está produciendo según lo previsto y que se devuelve un `NotFound` resultado si falta la imagen, suponiendo que este comportamiento es importante para la aplicación (es decir, no solo código temporal que el desarrollador agregó para diagnosticar un problema). La lógica real del archivo se ha movido a un servicio de implementación independiente y se ha mejorado para devolver una excepción específica de la aplicación en el caso de que falte un archivo. Puede probar esta implementación de forma independiente mediante una prueba de integración.

En la mayoría de los casos, querrá usar controladores de excepciones globales en los controladores, por lo que la cantidad de lógica en ellos debe ser mínima y probablemente no valga la pena realizar pruebas unitarias. Realice la mayoría de las pruebas de acciones del controlador mediante pruebas funcionales y la `TestServer` clase que se describe a continuación.

Pruebas de integración de aplicaciones ASP.NET Core

La mayoría de las pruebas de integración de las aplicaciones de ASP.NET Core deben ser servicios de prueba y otros tipos de implementación definidos en el proyecto de infraestructura. Por ejemplo, podría [probar que EF Core estaba actualizando y recuperando correctamente los datos que espera](#) de las clases de acceso a datos que residen en el proyecto de infraestructura. La mejor manera de probar que el proyecto de ASP.NET Core MVC se comporta correctamente es con pruebas funcionales que se ejecutan en la aplicación que se ejecuta en un host de prueba.

Pruebas funcionales de aplicaciones ASP.NET Core

Para las aplicaciones ASP.NET Core, la clase hace que las `TestServer` pruebas funcionales sean bastante fáciles de escribir. Configura un `TestServer` usando un `WebApplicationBuilder` (o

`HostBuilder`) directamente (como es habitual para su aplicación) o con el tipo `WebApplicationFactory` (disponible desde la versión 2.1). Intente hacer coincidir su host de prueba con el host de producción lo más estrechamente posible, de modo que sus pruebas ejecuten un comportamiento similar al que mostrará la aplicación en producción. La clase `WebApplicationFactory` es útil para configurar el `ContentRoot` de `TestServer`, que es utilizado por ASP.NET Core para localizar recursos estáticos como las vistas.

Puede crear pruebas funcionales sencillas mediante la creación de una clase de prueba que implemente `IClassFixture<WebApplicationFactory<TEntryPoint>>`, donde `TEntryPoint` es la clase `Startup` de su aplicación web. Después de incorporar esta interfaz, el accesorio de prueba puede crear un cliente con el método `CreateClient` de la fábrica:

C#

```
public class BasicWebTests : IClassFixture<WebApplicationFactory<Program>>
{
    protected readonly HttpClient _client;

    public BasicWebTests(WebApplicationFactory<Program> factory)
    {
        _client = factory.CreateClient();
    }

    // write tests that use _client
}
```

💡 Sugerencia

Si usa la configuración mínima de API en el archivo *de Program.cs*, de forma predeterminada, la clase se declarará interna y no será accesible desde el proyecto de prueba. En su lugar, puede elegir cualquier otra clase de instancia en el proyecto web o agregarla al archivo *Program.cs*:

C#

```
// Make the implicit Program class public so test projects can access it
public partial class Program { }
```

Con frecuencia, querrá realizar alguna configuración adicional del sitio antes de que se ejecute cada prueba, como configurar la aplicación para usar un almacén de datos en memoria y, a continuación, inicializar la aplicación con datos de prueba. Para lograr esta funcionalidad, cree su propia subclase de `WebApplicationFactory<TEntryPoint>` e invalide su `ConfigureWebHost`

método. El ejemplo siguiente es del proyecto eShopOnWeb FunctionalTests y se usa como parte de las pruebas en la aplicación web principal.

C#

```
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc.Testing;
using Microsoft.EntityFrameworkCore;
using Microsoft.eShopWeb.Infrastructure.Data;
using Microsoft.eShopWeb.Infrastructure.Identity;
using Microsoft.eShopWeb.Web;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;
using System;

namespace Microsoft.eShopWeb.FunctionalTests.Web;
public class WebTestFixture : WebApplicationFactory<Startup>
{
    protected override void ConfigureWebHost(IWebHostBuilder builder)
    {
        builder.UseEnvironment("Testing");

        builder.ConfigureServices(services =>
        {
            services.AddEntityFrameworkInMemoryDatabase();

            // Create a new service provider.
            var provider = services
                .AddEntityFrameworkInMemoryDatabase()
                .BuildServiceProvider();

            // Add a database context (ApplicationDbContext) using an in-memory
            // database for testing.
            services.AddDbContext<CatalogContext>(options =>
            {
                options.UseInMemoryDatabase("InMemoryDbForTesting");
                options.UseInternalServiceProvider(provider);
            });

            services.AddDbContext<AppIdentityDbContext>(options =>
            {
                options.UseInMemoryDatabase("Identity");
                options.UseInternalServiceProvider(provider);
            });

            // Build the service provider.
            var sp = services.BuildServiceProvider();

            // Create a scope to obtain a reference to the database
            // context (ApplicationDbContext).
            using (var scope = sp.CreateScope())
            {
                var scopedServices = scope.ServiceProvider;
```

```

var db = scopedServices.GetRequiredService<CatalogContext>();
var loggerFactory = scopedServices.GetRequiredService<ILoggerFactory>();

var logger = scopedServices
    .GetRequiredService<ILogger<WebTestFixture>>();

// Ensure the database is created.
db.Database.EnsureCreated();

try
{
    // Seed the database with test data.
    CatalogContextSeed.SeedAsync(db, loggerFactory).Wait();

    // seed sample user data
    var userManager =
scopedServices.GetRequiredService<userManager<ApplicationUser>>();
    var roleManager =
scopedServices.GetRequiredService<RoleManager<IdentityRole>>();
    AppIdentityDbContextSeed.SeedAsync(userManager, roleManager).Wait();
}
catch (Exception ex)
{
    logger.LogError(ex, $"An error occurred seeding the " +
        "database with test messages. Error: {ex.Message}");
}
}
});
}
}

```

Las pruebas pueden hacer uso de este WebApplicationFactory personalizado mediante él para crear un cliente y, a continuación, realizar solicitudes a la aplicación mediante esta instancia de cliente. La aplicación tendrá datos propagados que se pueden usar como parte de las aserciones de la prueba. La siguiente prueba comprueba que la página principal de la aplicación eShopOnWeb se carga correctamente e incluye una lista de productos que se agregó a la aplicación como parte de los datos de inicialización.

C#

```

using Microsoft.eShopWeb.FunctionalTests.Web;
using System.Net.Http;
using System.Threading.Tasks;
using Xunit;

namespace Microsoft.eShopWeb.FunctionalTests.WebRazorPages;
[Collection("Sequential")]
public class HomePageOnGet : IClassFixture<WebTestFixture>
{
    public HomePageOnGet(WebTestFixture factory)
    {
        Client = factory.CreateClient();
    }
}

```

```

}

public HttpClient Client { get; }

[Fact]
public async Task ReturnsHomePageWithProductListing()
{
    // Arrange & Act
    var response = await Client.GetAsync("/");
    response.EnsureSuccessStatusCode();
    var stringResponse = await response.Content.ReadAsStringAsync();

    // Assert
    Assert.Contains(".NET Bot Black Sweatshirt", stringResponse);
}
}

```

Esta prueba funcional evalúa el conjunto completo de aplicaciones ASP.NET Core MVC / Razor Pages, incluyendo todos los middleware, filtros y enlazadores que puedan estar presentes. Comprueba que una ruta determinada ("/") devuelve el código de estado correcto esperado y la salida HTML. Lo hace sin configurar un servidor web real y evita gran parte de la fragilidad que puede experimentar el uso de un servidor web real para las pruebas (por ejemplo, problemas con la configuración del firewall). Las pruebas funcionales que se ejecutan en TestServer suelen ser más lentas que las pruebas unitarias y de integración, pero son mucho más rápidas que las pruebas que se ejecutarían a través de la red en un servidor web de prueba. Utilice pruebas funcionales para garantizar que la interfaz de la aplicación funcione según lo previsto. Estas pruebas son especialmente útiles cuando se encuentra duplicación en los controladores o páginas y se aborda la duplicación mediante la adición de filtros. Lo ideal es que esta refactorización no cambie el comportamiento de la aplicación y un conjunto de pruebas funcionales comprobará que este es el caso.

Referencias: prueba de aplicaciones ASP.NET Core MVC

- Pruebas en ASP.NET Core
<https://learn.microsoft.com/aspnet/core/testing/>
- Convención de nomenclatura de pruebas unitarias
<https://ardalis.com/unit-test-naming-convention> [↗](#)
- Prueba de EF Core
<https://learn.microsoft.com/ef/core/miscellaneous/testing/>
- Pruebas de integración en ASP.NET Core
<https://learn.microsoft.com/aspnet/core/test/integration-tests>

Proceso de desarrollo para Azure

18/06/2025

💡 Sugerencia

Este contenido es un extracto del libro electrónico, Arquitecto de aplicaciones web modernas con ASP.NET Core y Azure, disponible en [.NET Docs](#) o como un PDF descargable gratuito que se puede leer sin conexión.

Descargar PDF



"Con la nube, los individuos y las pequeñas empresas pueden ajustar sus dedos y configurar instantáneamente los servicios de clase empresarial".

- Roy Stephan

Visión

Desarrolle aplicaciones ASP.NET Core bien diseñadas como quiera, mediante Visual Studio o la CLI de dotnet y Visual Studio Code o el editor que prefiera.

Entorno de desarrollo para aplicaciones de ASP.NET Core

Opciones de herramientas de desarrollo: IDE o editor

Tanto si prefiere un IDE completo y potente como un editor ligero y ágil, Microsoft tiene soluciones para usted al desarrollar aplicaciones ASP.NET Core.

Visual Studio 2022. Visual Studio 2022 es el IDE de mejor clase para desarrollar aplicaciones para ASP.NET Core. Ofrece una serie de características que aumentan la productividad del

desarrollador. Puede usarlo para desarrollar la aplicación y, a continuación, analizar su rendimiento y otras características. El depurador integrado le permite pausar la ejecución del código y recorrer el código sobre la marcha mientras se ejecuta. Su compatibilidad con la recarga en caliente le permite continuar trabajando en la aplicación desde donde la dejó, incluso después de realizar cambios de código, sin tener que reiniciar la aplicación. El ejecutor de pruebas integrado le permite organizar las pruebas y sus resultados e incluso realizar pruebas unitarias dinámicas mientras está codificando. Con Live Share, puede colaborar en tiempo real con otros desarrolladores y compartir la sesión de código sin problemas a través de la red. Y cuando esté listo, Visual Studio incluye todo lo que necesita para publicar la aplicación en Azure o donde quiera que lo hospede.

[Descargar Visual Studio 2022](#) ↗

Visual Studio Code y la CLI de dotnet (Herramientas multiplataforma para Mac, Linux y Windows). Si prefiere un editor ligero y multiplataforma que admita cualquier lenguaje de desarrollo, puede usar Microsoft Visual Studio Code y la CLI de dotnet. Estos productos proporcionan una experiencia sencilla y sólida que simplifica el flujo de trabajo del desarrollador. Además, Visual Studio Code admite extensiones para C# y desarrollo web, lo que proporciona IntelliSense y tareas de acceso directo en el editor.

[Descargar el SDK de .NET](#) ↗

[Descargar Visual Studio Code](#) ↗

Flujo de trabajo de desarrollo para aplicaciones ASP.NET Core hospedadas en Azure

El ciclo de vida de desarrollo de aplicaciones comienza desde la máquina de cada desarrollador, codificando la aplicación con su lenguaje preferido y probándola localmente. Los desarrolladores pueden elegir su sistema de control de código fuente preferido y pueden configurar la integración continua (CI) o la entrega/implementación continua (CD) mediante un servidor de compilación o en función de las características integradas de Azure.

Para empezar a desarrollar una aplicación de ASP.NET Core mediante CI/CD, puede usar Azure DevOps Services o el propio Team Foundation Server (TFS) de su organización. Acciones de GitHub proporcionan otra opción para compilar e implementar fácilmente aplicaciones en Azure, para aplicaciones cuyo código se hospeda en GitHub.

Configuración inicial

Para crear una canalización de versión de la aplicación, debe tener el código de la aplicación en el control de código fuente. Configure un repositorio local y conéctelo a un repositorio remoto

en un proyecto de equipo. Siga estas instrucciones:

- [Uso compartido del código con Git y Visual Studio](#) o
- [Comparte tu código con TFVC y Visual Studio](#)

Cree una instancia de Azure App Service donde implementará la aplicación. Para crear una aplicación web, vaya a la hoja App Services en Azure Portal. Haga clic en +Agregar, seleccione la plantilla Aplicación web, haga clic en Crear y proporcione un nombre y otros detalles. Se podrá acceder a la aplicación web desde {name}.azurewebsites.net.

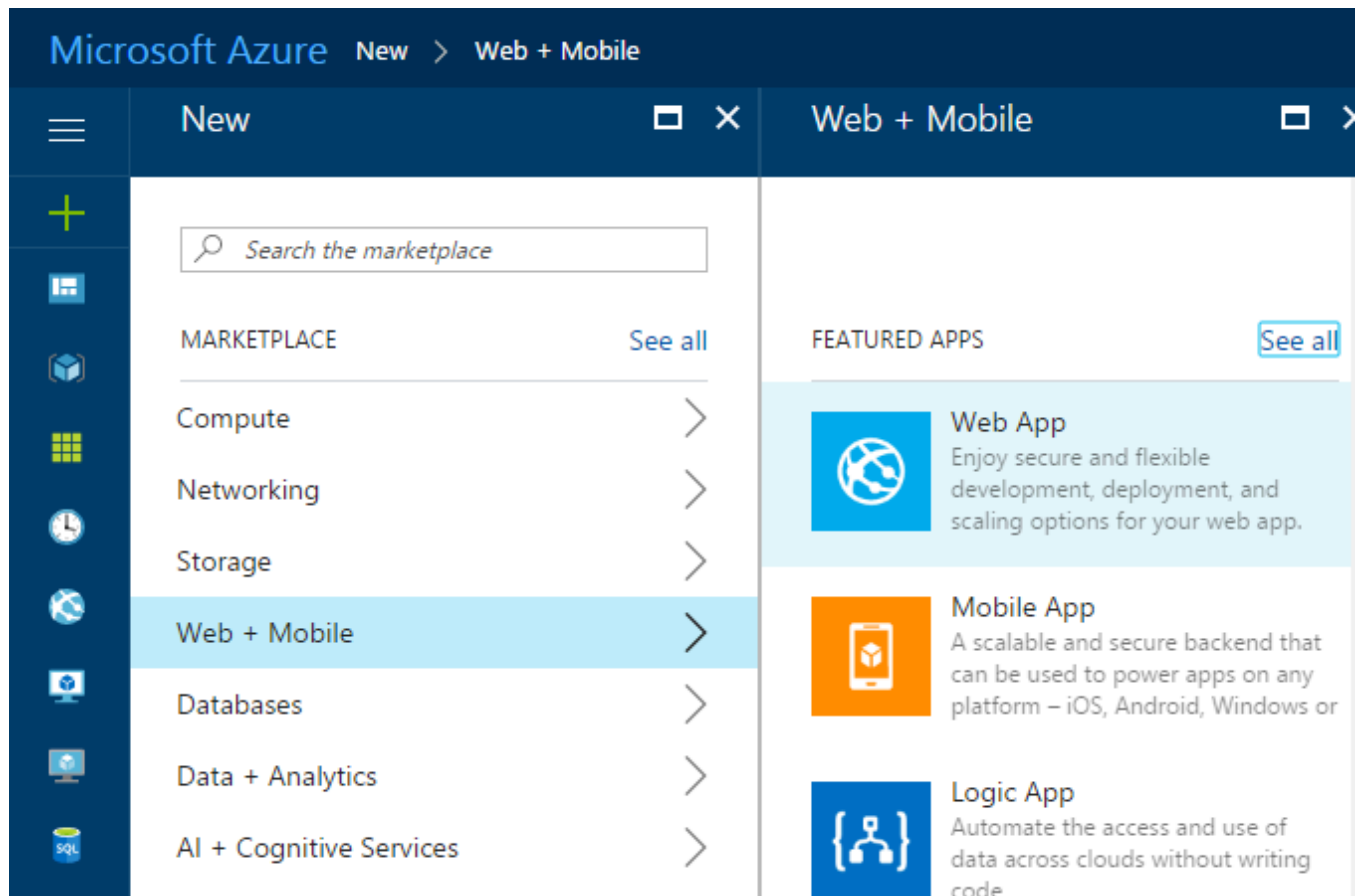


Figura 10-1. Creación de una aplicación web de Azure App Service en Azure Portal.

El proceso de compilación de CI realizará una compilación automatizada siempre que se confirme código nuevo en el repositorio de control de código fuente del proyecto. Este proceso ofrece información inmediata de que el código se compila (y, de forma ideal, que pasa las pruebas automatizadas) y que potencialmente se puede implementar. Esta compilación de CI generará un artefacto de paquete de implementación web y lo publicará para su uso por el proceso de CD.

[Definición del proceso de compilación de CI](#)

Asegúrese de habilitar la integración continua para que el sistema pondrá en cola una compilación cada vez que alguien del equipo confirme código nuevo. Pruebe la compilación y

compruebe que está produciendo un paquete de implementación web como uno de sus elementos.

Cuando una compilación se realiza correctamente, el proceso de CD implementará los resultados de la compilación de CI en la aplicación web de Azure. Para configurar este paso, cree y configure un *release*, que se implementará en Azure App Service.

Implementación de una aplicación web de Azure

Una vez configurada la canalización de CI/CD, puede realizar fácilmente actualizaciones en la aplicación web y confirmarlas en el control de código fuente para que se implementen.

Flujo de trabajo para desarrollar aplicaciones ASP.NET Core hospedadas en Azure

Una vez que haya configurado la cuenta de Azure y el proceso de CI/CD, el desarrollo de aplicaciones ASP.NET Core hospedadas en Azure es sencilla. A continuación se muestran los pasos básicos que normalmente se realizan al compilar una aplicación ASP.NET Core hospedada en Azure App Service como aplicación web, como se muestra en la figura 10-2.

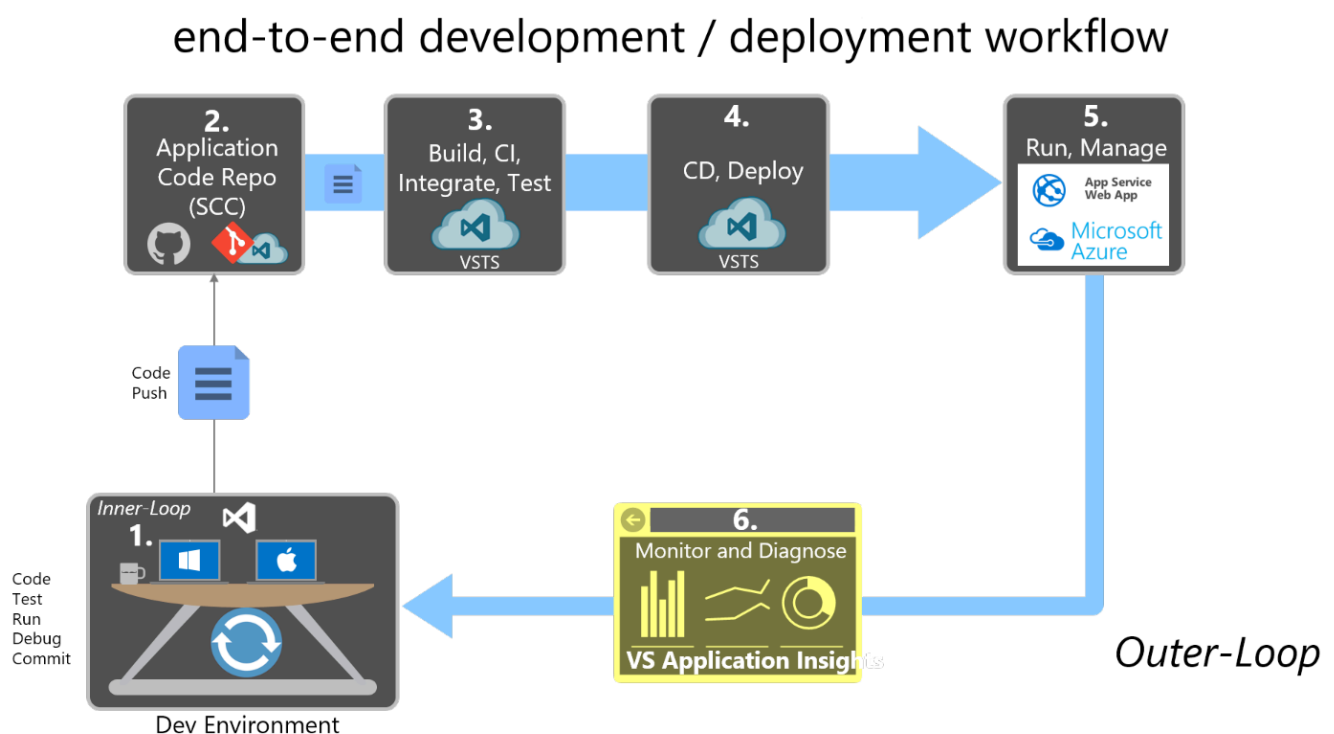


Figura 10-2. Flujo de trabajo paso a paso para compilar aplicaciones ASP.NET Core y hospedarlas en Azure

Paso 1. Bucle interno del entorno de desarrollo local

El desarrollo de la aplicación ASP.NET Core para la implementación en Azure no es diferente del desarrollo de la aplicación en caso contrario. Use el entorno de desarrollo local con el que se sienta cómodo, ya sea con Visual Studio 2019 o con la CLI de dotnet y Visual Studio Code o con el editor preferido. Puede escribir código, ejecutar y depurar los cambios, ejecutar pruebas automatizadas y realizar confirmaciones locales en el control de código fuente hasta que esté listo para insertar los cambios en el repositorio de control de código fuente compartido.

Paso 2. Repositorio de código de aplicación

Siempre que esté listo para compartir el código con su equipo, debe insertar los cambios del repositorio de origen local en el repositorio de origen compartido del equipo. Si ha estado trabajando en una rama personalizada, este paso normalmente implica fusionar su código en una rama compartida (quizás mediante un [pull request](#)).

Paso 3. Servidor de compilación: integración continua. compilación, prueba, paquete

Se desencadena una nueva compilación en el servidor de compilación cada vez que se realiza una nueva confirmación en el repositorio de código de la aplicación compartida. Como parte del proceso de CI, esta compilación debe compilar completamente la aplicación y ejecutar pruebas automatizadas para confirmar que todo funciona según lo previsto. El resultado final del proceso de CI debe ser una versión empaquetada de la aplicación web, lista para la implementación.

Paso 4. Servidor de compilación: entrega continua

Una vez realizada correctamente la compilación, el proceso de CD recogerá los artefactos de compilación generados. Este proceso incluirá un paquete de implementación web. El servidor de compilación implementará este paquete en Azure App Service, reemplazando cualquier servicio existente por el recién creado. Normalmente, este paso tiene como destino un entorno de ensayo, pero algunas aplicaciones se implementan directamente en producción a través de un proceso de CD.

Paso 5. Aplicación web de Azure App Service

Una vez implementada, la aplicación ASP.NET Core se ejecuta dentro del contexto de una aplicación web de Azure App Service. Esta aplicación web se puede supervisar y configurar aún más mediante Azure Portal.

Paso 6. Supervisión y diagnósticos de producción

Mientras se ejecuta la aplicación web, puede supervisar el estado de la aplicación y recopilar datos de comportamiento de usuario y diagnósticos. Application Insights se incluye en Visual Studio y ofrece instrumentación automática para aplicaciones ASP.NET. Puede proporcionarle información sobre el uso, las excepciones, las solicitudes, el rendimiento y los registros.

Referencias

Compilación e implementación de la aplicación ASP.NET Core en Azure

<https://learn.microsoft.com/azure/devops/build-release/apps/aspnet/build-aspnet-core>

Siguiente anterior

Recomendaciones de hospedaje de Azure para aplicaciones web de ASP.NET Core

18/06/2025

💡 Sugerencia

Este contenido es un extracto del libro electrónico, Arquitecto de aplicaciones web modernas con ASP.NET Core y Azure, disponible en [.NET Docs](#) o como un PDF descargable gratuito que se puede leer sin conexión.

Descargar PDF



Los líderes de línea de negocio en todas partes están omitiendo los departamentos de TI para obtener aplicaciones de la nube (también conocidas como SaaS) y pagarlas como si fueran una suscripción a una revista. Y cuando el servicio ya no sea necesario, pueden cancelar la suscripción sin que el equipo quede sin usar en la esquina".

- Daryl Plummer, analista de Gartner

Independientemente de las necesidades y la arquitectura de la aplicación, Microsoft Azure puede admitirla. Sus necesidades de hospedaje pueden ser tan simples como un sitio web estático o una aplicación sofisticada formada por docenas de servicios. Para ASP.NET aplicaciones web monolíticas principales y servicios auxiliares, hay varias configuraciones conocidas que se recomiendan. Las recomendaciones de este artículo se agrupan en función del tipo de recurso que se va a hospedar, ya sean aplicaciones completas, procesos individuales o datos.

Aplicaciones web

Las aplicaciones web se pueden hospedar con:

- Aplicaciones web de App Service

- Contenedores (varias opciones)
- Virtual Machines (VM)

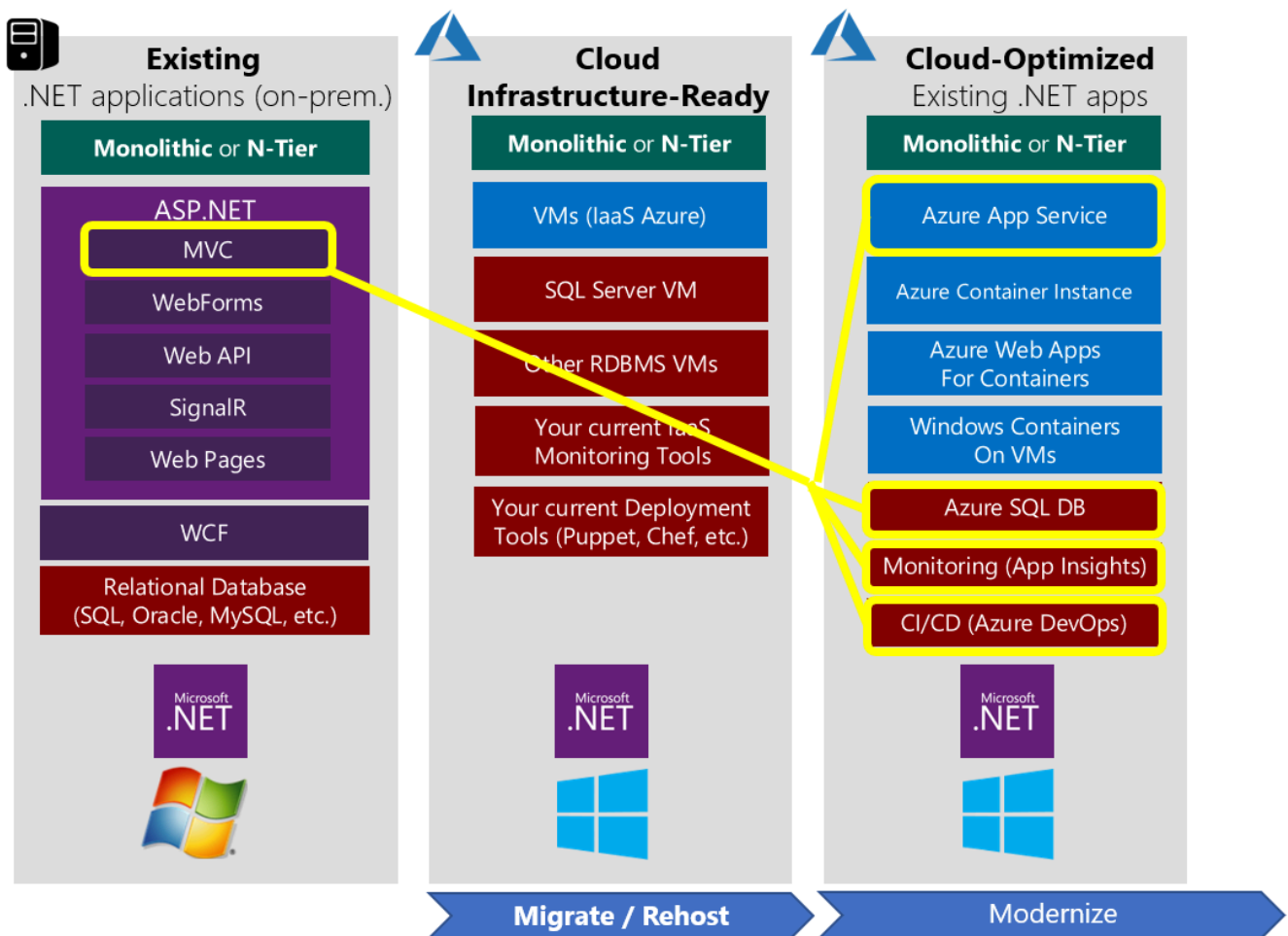
De estas opciones, App Service Web Apps es el enfoque recomendado para la mayoría de los escenarios, incluidas las aplicaciones sencillas basadas en contenedores. En el caso de las arquitecturas de microservicios, considere un enfoque basado en contenedores. Si necesita más control sobre las máquinas que ejecutan la aplicación, considere la posibilidad de usar Azure Virtual Machines.

Aplicaciones Web de App Service

App Service Web Apps ofrece una plataforma totalmente administrada optimizada para hospedar aplicaciones web. Es una oferta de plataforma como servicio (PaaS) que le permite centrarse en la lógica de negocios, mientras que Azure se encarga de la infraestructura necesaria para ejecutar y escalar la aplicación. Algunas características clave de App Service Web Apps:

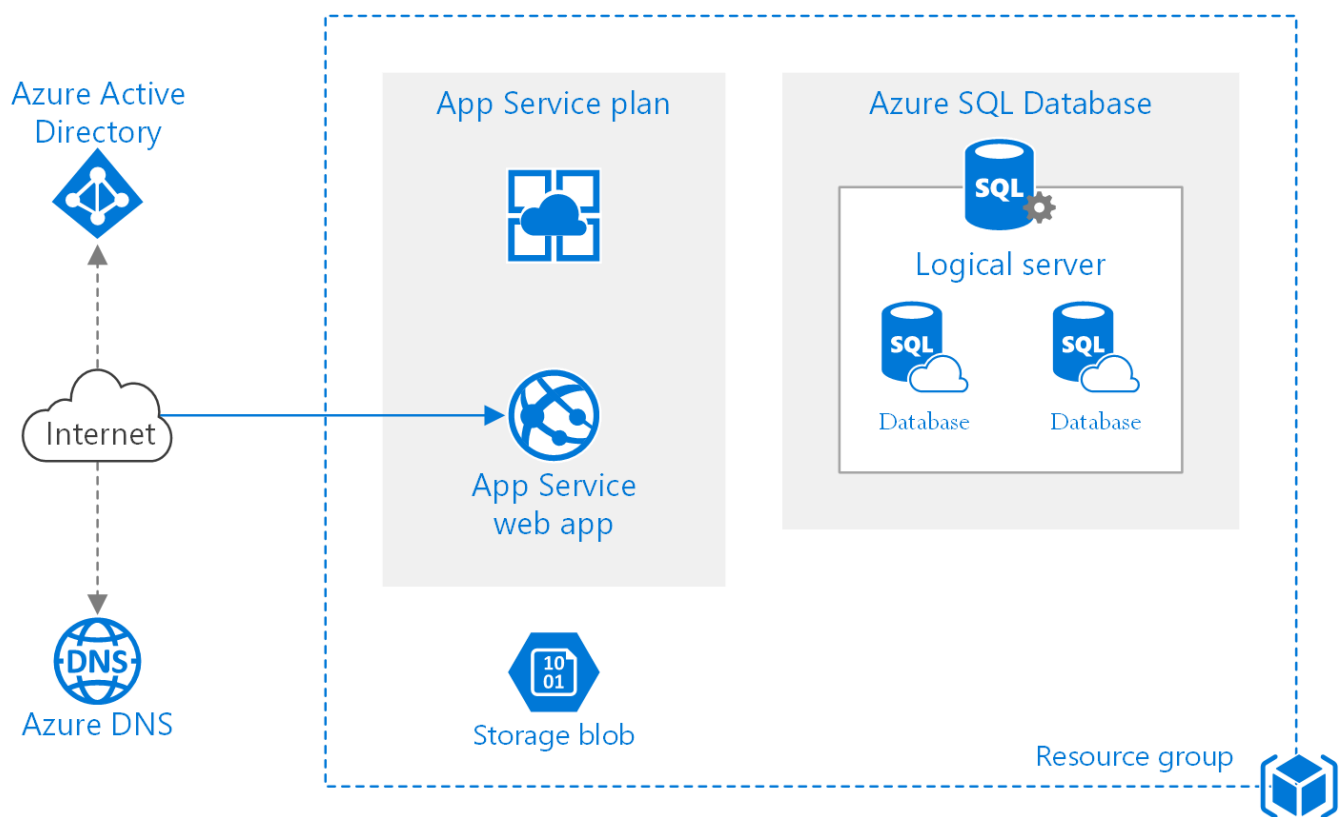
- Optimización de DevOps (integración y entrega continuas, varios entornos, pruebas A/B, compatibilidad con scripting).
- Escala global y alta disponibilidad.
- Conexiones a plataformas SaaS y a los datos locales.
- Seguridad y cumplimiento.
- Integración de Visual Studio.

Azure App Service es la mejor opción para la mayoría de las aplicaciones web. La implementación y la administración se integran en la plataforma, los sitios se pueden escalar rápidamente para controlar cargas de tráfico elevadas y el equilibrio de carga integrado y el administrador de tráfico proporcionan alta disponibilidad. Puede mover fácilmente sitios existentes a Azure App Service con una herramienta de migración en línea. Puede usar una aplicación de código abierto desde la Galería de aplicaciones web o crear un sitio con el marco y las herramientas que prefiera. La característica WebJobs facilita la adición del procesamiento de trabajos en segundo plano a la aplicación web de App Service. Si tiene una aplicación de ASP.NET existente hospedada en el entorno local mediante una base de datos local, hay una ruta de acceso clara para migrar. Puede usar App Service Web App con una instancia de Azure SQL Database (o proteger el acceso al servidor de bases de datos local, si lo prefiere).



En la mayoría de los casos, pasar de una aplicación de ASP.NET hospedada localmente a una aplicación web de App Service es un proceso sencillo. Poco o ninguna modificación debe ser necesaria de la propia aplicación y puede empezar a aprovechar rápidamente las muchas características que ofrece Azure App Service Web Apps.

Además de las aplicaciones que no están optimizadas para la nube, Azure App Service Web Apps es una solución excelente para muchas aplicaciones monolíticas sencillas (no distribuidas), como muchas aplicaciones ASP.NET Core. En este enfoque, la arquitectura es básica y sencilla de entender y administrar:

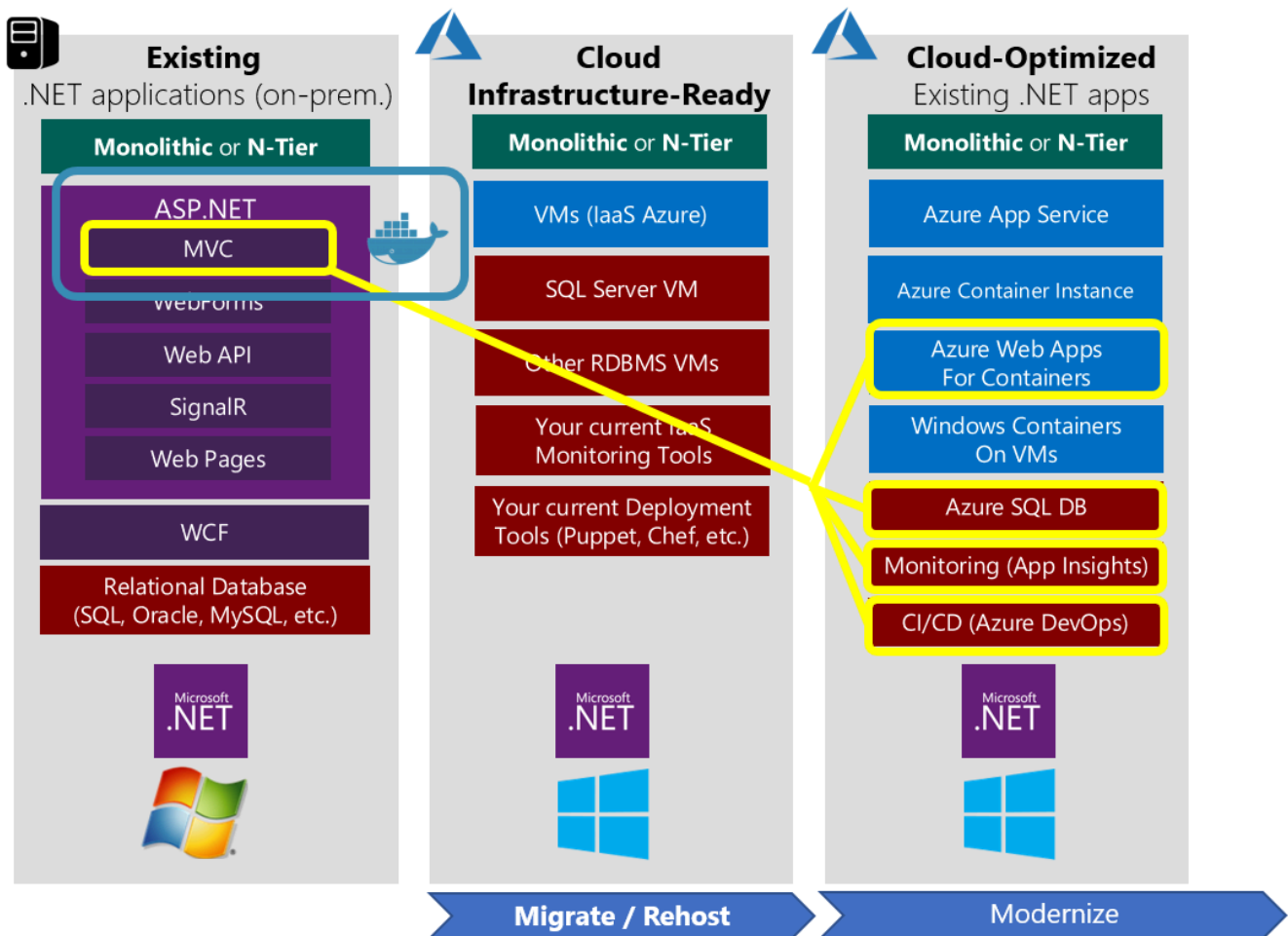


Normalmente, un pequeño número de recursos de un único grupo de recursos es suficiente para administrar dicha aplicación. Las aplicaciones que normalmente se implementan como una sola unidad, en lugar de las aplicaciones que se componen de muchos procesos independientes, son buenos candidatos para este [enfoque arquitectónico básico](#). Aunque es arquitectónicamente sencillo, este enfoque todavía permite que la aplicación hospedada escale verticalmente (más recursos por nodo) y fuera (más nodos hospedados) para satisfacer cualquier aumento de la demanda. Con el escalado automático, la aplicación se puede configurar para ajustar automáticamente el número de nodos que hospedan la aplicación en función de la demanda y la carga media entre los nodos.

Aplicaciones Web de App Service para Contenedores

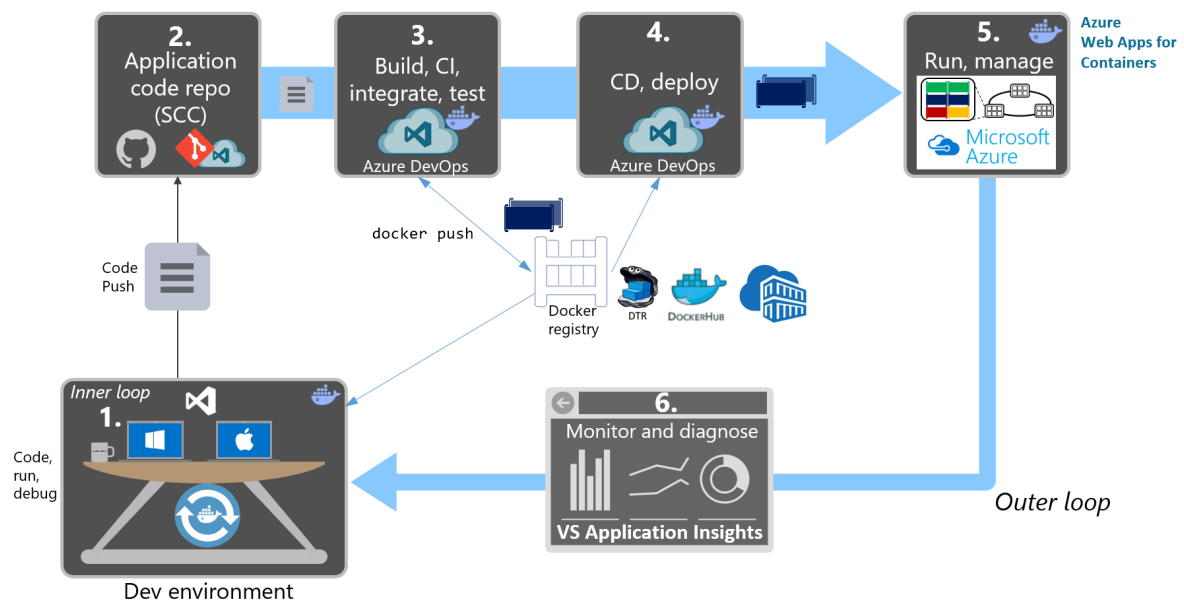
Además de la compatibilidad con el hospedaje de aplicaciones web directamente, [App Service Web Apps for Containers](#) se puede usar para ejecutar aplicaciones en contenedores en Windows y Linux. Con este servicio, puede implementar y ejecutar fácilmente aplicaciones en contenedores que se pueden escalar con su negocio. Las aplicaciones tienen todas las características de App Service Web Apps enumeradas anteriormente. Además, Web Apps for Containers admite CI/CD simplificados con Docker Hub, Azure Container Registry y GitHub. Puede usar Azure DevOps para definir canalizaciones de compilación e implementación que publican cambios en un registro. Estos cambios se pueden probar en un entorno de ensayo y implementarlos automáticamente en producción mediante ranuras de implementación, lo que permite actualizaciones sin tiempo de inactividad. Revertir a versiones anteriores se puede realizar con la misma facilidad.

Hay algunos escenarios en los que Web Apps for Containers tiene más sentido. Si tiene aplicaciones existentes que puede incluir en contenedores, ya sea en contenedores de Windows o Linux, puede hospedarlas fácilmente mediante este conjunto de herramientas. Solo tiene que publicar el contenedor y, a continuación, configurar Web Apps for Containers para extraer la versión más reciente de esa imagen del registro que prefiera. Este es un enfoque de "lift and shift" para migrar desde modelos clásicos de alojamiento de aplicaciones a un modelo optimizado en la nube.



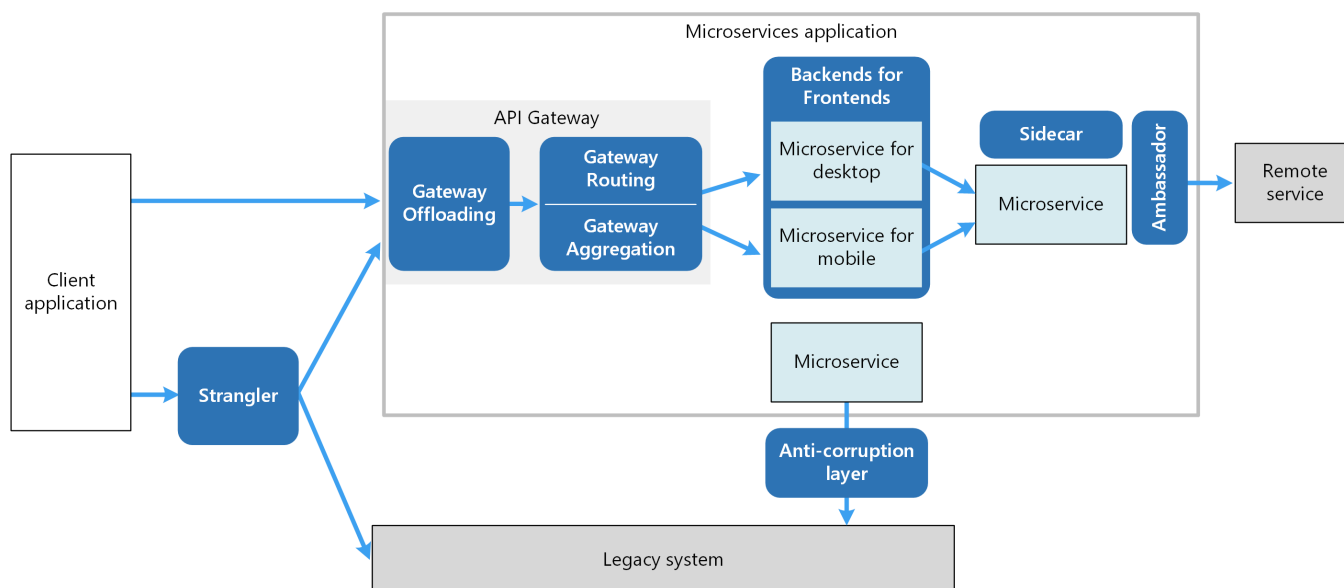
Este enfoque también funciona bien si el equipo de desarrollo puede pasar a un proceso de desarrollo basado en contenedores. El "bucle interno" del desarrollo de aplicaciones con contenedores incluye la compilación de la aplicación con contenedores. Los cambios realizados en el código, así como en la configuración del contenedor se insertan en el control de código fuente y una compilación automatizada es responsable de publicar nuevas imágenes de contenedor en un registro como Docker Hub o Azure Container Registry. Estas imágenes se usan como base para el desarrollo adicional, así como para las implementaciones en producción, como se muestra en el diagrama siguiente:

End to End Docker DevOps Lifecycle Workflow



El desarrollo con contenedores ofrece muchas ventajas, especialmente cuando se usan contenedores en producción. La misma configuración de contenedor se usa para hospedar la aplicación en cada entorno en el que se ejecuta, desde la máquina de desarrollo local para compilar y probar sistemas en producción. Este enfoque reduce considerablemente la probabilidad de defectos resultantes de diferencias en la configuración de la máquina o las versiones de software. Los desarrolladores también pueden usar las herramientas con las que son más productivos, incluido el sistema operativo, ya que los contenedores se pueden ejecutar en cualquier sistema operativo. En algunos casos, las aplicaciones distribuidas que implican muchos contenedores pueden ser muy intensivas en recursos para ejecutarse en una sola máquina de desarrollo. En este escenario, puede tener sentido actualizar a mediante Kubernetes y Azure Dev Spaces, que se tratan en la sección siguiente.

A medida que las partes de aplicaciones más grandes se dividen en sus propios *microservicios más pequeños* e independientes, se pueden usar patrones de diseño adicionales para mejorar el comportamiento de la aplicación. En lugar de trabajar directamente con servicios individuales, una *puerta de enlace de API* puede simplificar el acceso y desacoplar el cliente desde su back-end. Tener back-ends de servicio independientes para distintos servidores front-end también permite que los servicios evolucionen de forma conjunta con sus consumidores. Se puede acceder a los servicios comunes a través de un contenedor *sidecar* independiente, que puede incluir bibliotecas comunes de conectividad de cliente mediante el patrón *ambassador*.



Obtenga más información sobre los patrones de diseño que se deben tener en cuenta al crear sistemas basados en microservicios.

Azure Kubernetes Service

Azure Kubernetes Service (AKS) administra el entorno de Kubernetes hospedado, lo que facilita la implementación y administración rápida y sencilla de aplicaciones en contenedores sin experiencia en orquestación de contenedores. También elimina la carga de las operaciones y el mantenimiento continuos mediante el aprovisionamiento, la actualización y el escalado de recursos a petición, sin desconectar las aplicaciones.

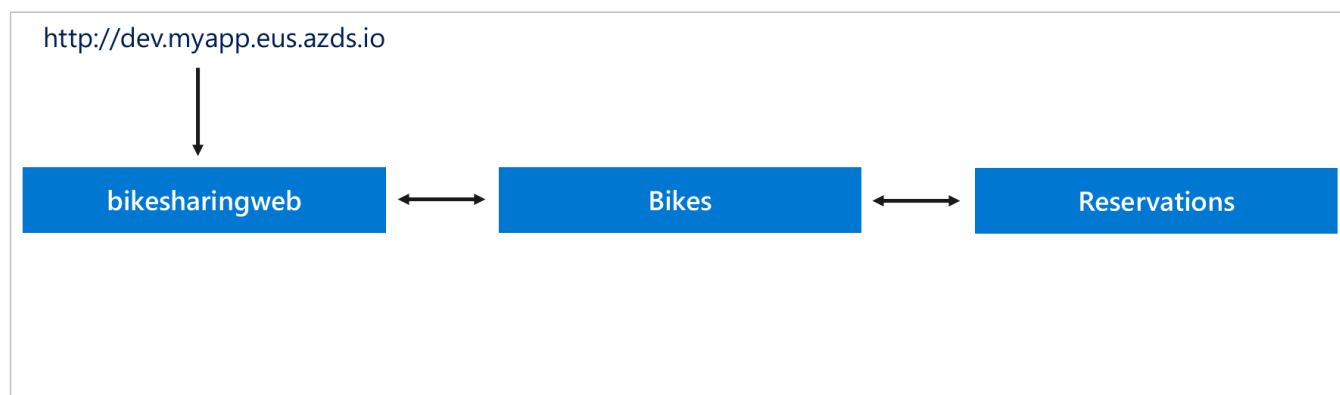
AKS reduce la complejidad y la sobrecarga operativa de administrar un clúster de Kubernetes descargando gran parte de esa responsabilidad en Azure. Como servicio de Kubernetes hospedado, Azure controla tareas críticas, como la supervisión de estado y el mantenimiento. Además, solo se paga por los nodos de agente en los clústeres, no por los nodos principales. Como servicio de Kubernetes administrado, AKS proporciona:

- Actualizaciones y revisiones automatizadas de la versión de Kubernetes.
- Escalado sencillo de clústeres.
- Plano de control hospedado de recuperación automática (nodos principales).
- Ahorro de costos: pague solo por la operación de nodos del grupo de agentes.

Con Azure que controla la administración de los nodos del clúster de AKS, ya no es necesario realizar muchas tareas manualmente, como las actualizaciones del clúster. Dado que Azure controla estas tareas de mantenimiento críticas, AKS no proporciona acceso directo (por ejemplo, con SSH) al clúster.

Los equipos que aprovechan AKS también pueden aprovechar Azure Dev Spaces. Azure Dev Spaces ayuda a los equipos a centrarse en el desarrollo y la iteración rápida de su aplicación de microservicios al permitir que los equipos trabajen directamente con toda su arquitectura o

aplicación de microservicios que se ejecutan en AKS. Azure Dev Spaces también proporciona una manera de actualizar de forma independiente partes de la arquitectura de microservicios de forma aislada sin afectar al resto del clúster de AKS u otros desarrolladores.



Azure Dev Spaces:

- Minimizar el tiempo de instalación de la máquina local y los requisitos de recursos
- Permitir que los equipos iteren de manera más rápida
- Reducir el número de entornos de integración requeridos por un equipo
- Eliminación de la necesidad de simular determinados servicios en un sistema distribuido al desarrollar o probar

[Más información sobre Azure Dev Spaces](#)

Azure Virtual Machines

Si tiene una aplicación existente que requeriría modificaciones sustanciales para ejecutarse en App Service, puede elegir Máquinas virtuales para simplificar la migración a la nube. Sin embargo, la configuración, la protección y el mantenimiento correctos de las máquinas virtuales requieren mucho más tiempo y experiencia en TI en comparación con Azure App Service. Si está pensando en Azure Virtual Machines, asegúrese de tener en cuenta el esfuerzo de mantenimiento continuo necesario para aplicar revisiones, actualizar y administrar el entorno de máquina virtual. Azure Virtual Machines es una infraestructura como servicio (IaaS), mientras que App Service es PaaS. También debe tener en cuenta si la implementación de la aplicación como contenedor de Windows en Web App for Containers podría ser una opción viable para su escenario.

Procesos lógicos

Los procesos lógicos individuales que se pueden desacoplar del resto de la aplicación se pueden implementar de forma independiente en Azure Functions de forma "sin servidor". Azure Functions le permite escribir el código que necesita para un problema determinado, sin preocuparse por la aplicación o la infraestructura para ejecutarlo. Puede elegir entre una

variedad de lenguajes de programación, como C#, F#, Node.js, Python y PHP, lo que le permite elegir el lenguaje más productivo para la tarea a mano. Al igual que la mayoría de las soluciones basadas en la nube, solo paga por la cantidad de tiempo que usa y puede confiar en Azure Functions para escalar verticalmente según sea necesario.

Datos

Azure ofrece una amplia variedad de opciones de almacenamiento de datos para que la aplicación pueda usar el proveedor de datos adecuado para los datos en cuestión.

Para los datos transaccionales, relacionales, las bases de datos de Azure SQL son la mejor opción. En el caso de datos de lectura de alto rendimiento principalmente, una caché de Redis respaldada por una base de datos de Azure SQL es una buena solución.

Los datos JSON no estructurados se pueden almacenar de varias maneras, desde columnas de SQL Database a blobs o tablas de Azure Storage, a Azure Cosmos DB. De estas opciones, Azure Cosmos DB ofrece la mejor funcionalidad de consulta y es la opción recomendada para un gran número de documentos basados en JSON que deben admitir consultas.

Los datos efímeros basados en comandos o eventos usados para orquestar el comportamiento de la aplicación pueden utilizar Azure Service Bus o Azure Storage Queues. Azure Service Bus ofrece más flexibilidad y es el servicio recomendado para la mensajería no trivial dentro y entre aplicaciones.

Recomendaciones de arquitectura

Los requisitos de la aplicación deben dictar su arquitectura. Hay muchos servicios de Azure diferentes disponibles. Elegir el adecuado es una decisión importante. Microsoft ofrece una galería de arquitecturas de referencia para ayudar a identificar arquitecturas típicas optimizadas para escenarios comunes. Puede encontrar una arquitectura de referencia que se asigne estrechamente a los requisitos de la aplicación o al menos ofrezca un punto de partida.

En la figura 11-1 se muestra una arquitectura de referencia de ejemplo. En este diagrama se describe un enfoque de arquitectura recomendado para un sitio web del sistema de administración de contenido de Sitecore optimizado para marketing.

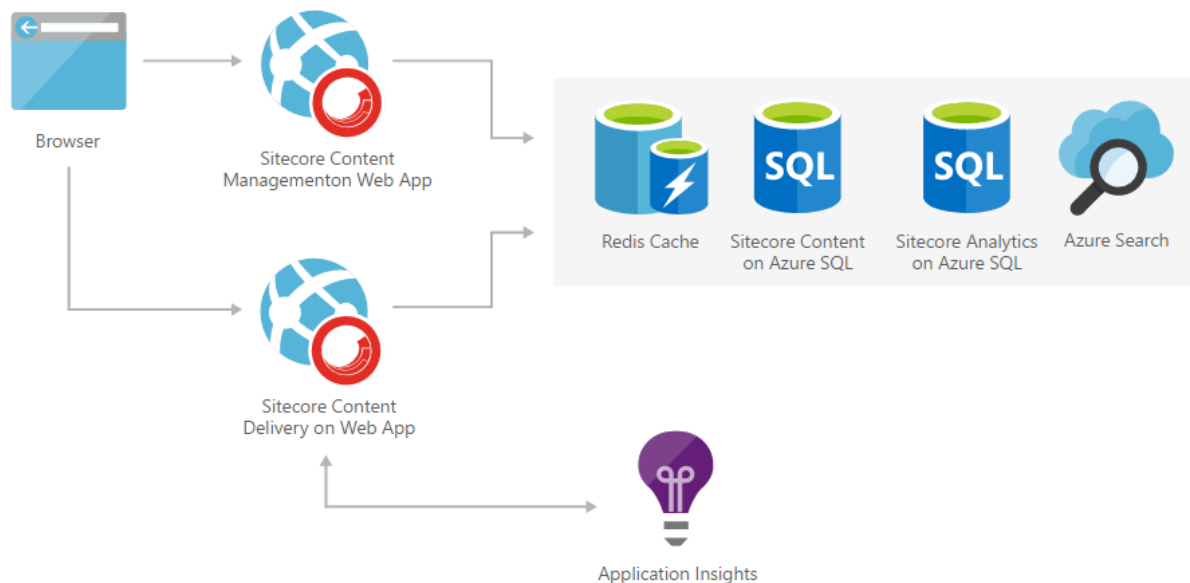


Figura 11-1. Arquitectura de referencia del sitio web de marketing de sitecore.

Referencias: recomendaciones de hospedaje de Azure

- Arquitecturas de soluciones de Azure
<https://azure.microsoft.com/solutions/architecture/>
- Arquitectura básica de aplicaciones web de Azure
<https://learn.microsoft.com/azure/architecture/reference-architectures/app-service-web-app/basic-web-app>
- Patrones de diseño para microservicios
<https://learn.microsoft.com/azure/architecture/microservices/design/patterns>
- Guía para desarrolladores de Azure
<https://azure.microsoft.com/campaigns/developer-guide/>
- Introducción a Web Apps
<https://learn.microsoft.com/azure/app-service/app-service-web-overview>
- Aplicación web para contenedores
<https://azure.microsoft.com/services/app-service/containers/>
- Introducción a Azure Kubernetes Service (AKS)
<https://learn.microsoft.com/azure/aks/intro-kubernetes>