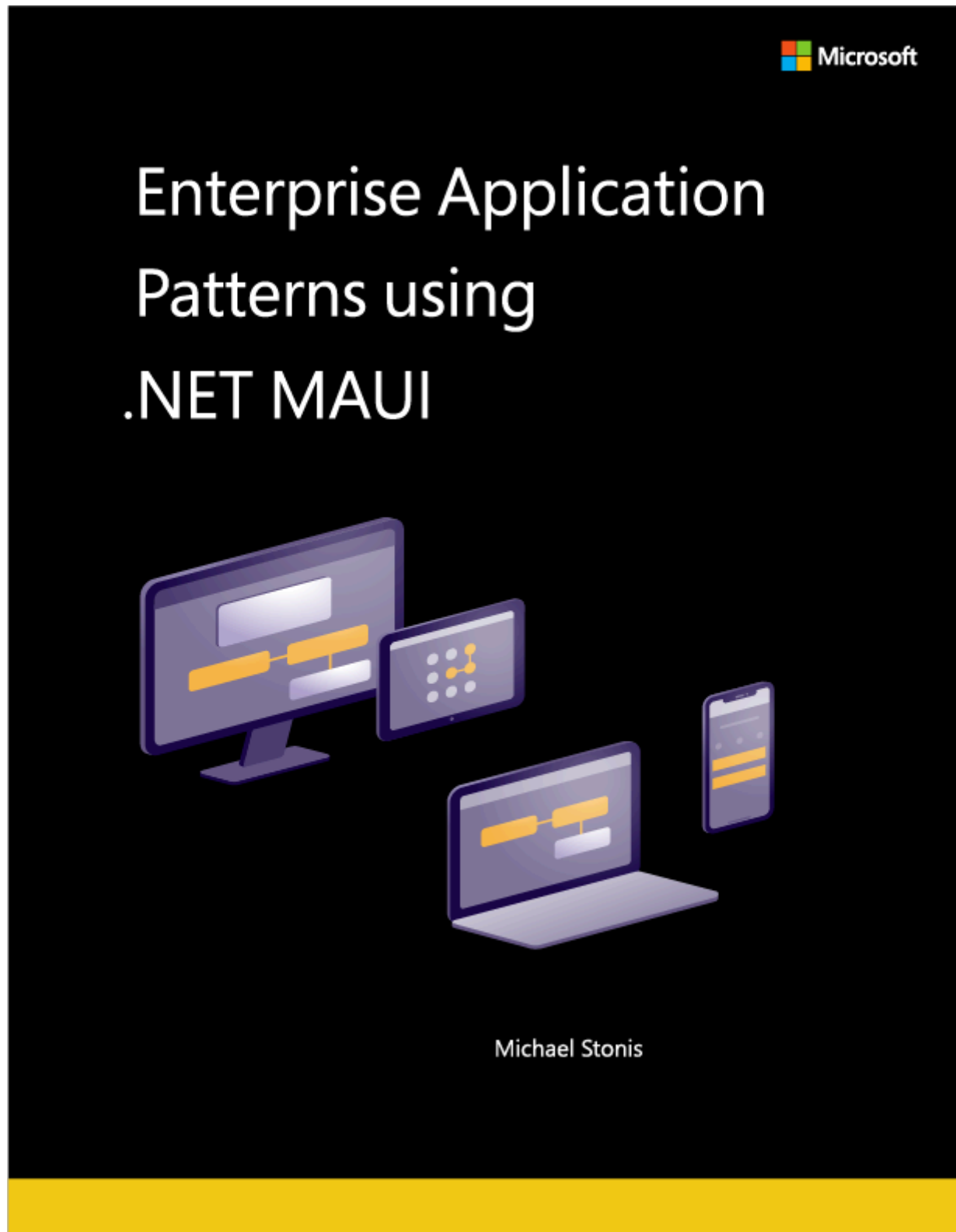


Patrones de aplicación empresarial mediante .NET MAUI

19/06/2025



DESCARGA disponible en: <https://aka.ms/maui-ebook> 

EDITION v2.0

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 © 2022 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 debe deducirse ninguna asociación o conexión real.

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

Mac y macOS son marcas comerciales de Apple Inc.

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

Autores:

Michael Stonis, arquitecto de software móvil, **Eight-Bot**

Revisores:

James Montemagno, director principal del programa, Microsoft Corp.

David Pine, Relaciones para desarrolladores, Microsoft Corp.

Reconocimientos

Este libro se originó a partir de los Excelentes Patrones de Aplicaciones Empresariales utilizando el eBook de Xamarin.Forms de **David Britch** y **Javier Suarez Ruiz**. Sin su trabajo duro, información detallada y excelentes ejemplos, este libro no sería posible.

Introducción

Las aplicaciones empresariales se enfrentan a una serie de problemas difíciles de resolver, incluyendo los requisitos empresariales siempre cambiantes, la necesidad de tiempos de respuesta rápidos, la compatibilidad con múltiples plataformas y la integración con varios sistemas. Debido a la naturaleza variable de estos problemas, es importante que la arquitectura de la aplicación le permita ser modular, modificable y extensible a lo largo del tiempo.

En este libro se proporcionan soluciones reales para solucionar estos problemas al compilar una aplicación empresarial mediante .NET MAUI. En este libro se usa una aplicación .NET MAUI pregenerada que actúa como front-end de una aplicación de comercio electrónico en línea como referencia y una guía para patrones de diseño empresariales comunes. En este libro se tratan temas como el patrón MVVM, la inserción de dependencias, la navegación, la configuración, el acoplamiento flexible de componentes y problemas empresariales adicionales. El contenido de este libro es útil para cualquier persona que quiera crear una nueva aplicación para este negocio o para resolver los problemas de las aplicaciones que evolucionan con el tiempo.

Quién debe usar el libro

Este libro es para desarrolladores de .NET MAUI que ya están familiarizados con el marco de trabajo, pero que buscan instrucciones sobre la arquitectura y la implementación al compilar aplicaciones empresariales. Este libro puede ayudar a los desarrolladores a resolver problemas comunes mediante patrones probados y verdaderos.

Cómo usar el libro

Este libro se centra en la creación de aplicaciones empresariales multiplataforma mediante .NET MAUI. Como tal, debe leerse en su totalidad para adquirir una base de conocimiento de estas aplicaciones y sus consideraciones técnicas. El libro, junto con su aplicación de ejemplo, también puede servir como punto de partida o referencia para crear una nueva aplicación empresarial. Use la aplicación de ejemplo asociada como plantilla para la nueva aplicación o para ver cómo organizar los elementos de componente de una aplicación. A continuación, vuelva a consultar esta guía para obtener indicaciones sobre la arquitectura. Puede encontrar la aplicación de ejemplo en [GitHub](#).

Lo que este libro no cubre

Este libro está dirigido a lectores que ya están familiarizados con .NET MAUI. Trata algunos conceptos de .NET MAUI para ayudar a ilustrar mejor el tema, pero no cubre la mayoría de los

controles y conceptos en detalle. Para obtener instrucciones generales sobre cómo compilar una nueva aplicación .NET MAUI , consulte la guía [Creación de la primera aplicación](#) en la documentación de .NET MAUI .

Recursos adicionales

Para obtener contenido oficial de .NETMAUI, consulte [la documentación de .NETMAUI](#). .NET MAUI se desarrolla como un proyecto de código abierto y está disponible en GitHub en [dotnet/maui](#) [↗]. Para obtener ejemplos de código desarrollados con .NET MAUI, consulte el repositorio [dotnet/maui-samples](#) [↗] .

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 está leyendo este libro en <learn.microsoft.com>, use la sección **Comentarios** en la parte inferior 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:** usando una plantilla de incidencia 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](#) [↗].

Próximo

Propósito

Artículo • 28/06/2024

💡 Sugerencia

Este contenido es un extracto del libro electrónico "Patrones de aplicaciones empresariales con .NET MAUI", disponible en [Documentación de .NET](#) o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF



Este libro electrónico proporciona indicaciones para la creación de aplicaciones empresariales multiplataforma con .NET MAUI. .NET MAUI es un kit de herramientas de interfaz de usuario multiplataforma que permite a los desarrolladores crear fácilmente diseños de interfaz de usuario nativa que se pueden compartir entre plataformas, como iOS, macOS, Android y Windows. Proporciona una solución completa para aplicaciones B2E (negocio a empleado), B2B (negocio a negocio) y B2C (negocio a consumidor), lo que aporta la capacidad de compartir código entre todas las plataformas de destino y ayudar a reducir el costo total de propiedad (TCO).

La guía proporciona instrucciones arquitectónicas para desarrollar aplicaciones empresariales con .NET MAUI que sean adaptables, fáciles de mantener y se puedan probar. Proporciona instrucciones para implementar MVVM, inserción de dependencias, navegación, validación y administración de la configuración, a la vez que mantiene un acoplamiento ligero. Además, incluye indicaciones para llevar a cabo la autenticación y la autorización con IdentityServer, acceder a datos de microservicios contenedorizados y realizar pruebas unitarias.

La guía incluye código fuente para la [aplicación multiplataforma eShop](#) y para la [aplicación de referencia eShop](#). La aplicación multiplataforma eShop es una aplicación empresarial multiplataforma desarrollada con .NET MAUI que se conecta a una serie de microservicios contenedorizados conocidos como la aplicación de referencia eShop. Sin

embargo, la aplicación multiplataforma eShop se puede configurar para consumir datos de servicios ficticios para aquellos que quieran evitar la implementación de los microservicios contenedorizados.

Qué está fuera del ámbito de esta guía

Esta guía está dirigida a lectores que ya están familiarizados con .NET MAUI. Para obtener una introducción detallada a .NET MAUI, consulte la [Documentación de .NET MAUI](#).

Destinatarios de esta guía

Esta guía está dirigida principalmente a desarrolladores y arquitectos que desean aprender a diseñar e implementar aplicaciones empresariales multiplataforma con .NET MAUI.

En segundo lugar, está dirigida a responsables de la toma de decisiones técnicas que desean obtener información general sobre la arquitectura y la tecnología antes de decidir qué enfoque adoptar para el desarrollo de aplicaciones empresariales multiplataforma con .NET MAUI.

Cómo utilizar esta guía

Esta guía se centra en la creación de aplicaciones empresariales multiplataforma con .NET MAUI. Como tal, debe leerse en su totalidad para adquirir una base de conocimiento de estas aplicaciones y sus consideraciones técnicas. La guía y la aplicación de ejemplo que incluye también pueden servir como punto de partida o de referencia para crear una nueva aplicación empresarial. Use la aplicación de ejemplo asociada como plantilla para la nueva aplicación o para ver cómo organizar los elementos que componen una aplicación. A continuación, vuelva a consultar esta guía para obtener indicaciones sobre la arquitectura.

No dude en reenviarles la guía a los miembros de su equipo para facilitar un conocimiento común del desarrollo de aplicaciones empresariales multiplataforma con .NET MAUI. El hecho de que todos trabajen con una terminología y unos principios subyacentes comunes ayudará a garantizar una aplicación coherente de las prácticas y los patrones de diseño.

Colaborar con nosotros en GitHub

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).

.NET

Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)

Introducción a .NET MAUI

Artículo • 30/05/2024

💡 Sugerencia

Este contenido es un extracto del libro electrónico "Patrones de aplicaciones empresariales con .NET MAUI", disponible en [Documentación de .NET](#) o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF



Sea cual sea la plataforma que usen, los desarrolladores de aplicaciones empresariales deben enfrentarse a diversas dificultades:

- Requisitos de aplicación que pueden cambiar con el tiempo
- Nuevas adversidades y oportunidades de negocio
- Comentarios constantes durante el desarrollo que pueden afectar significativamente al ámbito y los requisitos de la aplicación

Teniendo esto en cuenta, es importante crear aplicaciones que se puedan modificar o extender fácilmente con el tiempo. Diseñar esta capacidad de adaptación puede ser complicado, ya que requiere una arquitectura que permita desarrollar y probar partes concretas de la aplicación de manera independiente y aislada, sin que ello afecte al resto de la aplicación.

Muchas aplicaciones empresariales son lo suficientemente complejas como para requerir más de un desarrollador. Puede ser realmente arduo decidir cómo diseñar una aplicación para que varios desarrolladores puedan trabajar eficazmente en diferentes partes de ella de manera independiente y, al mismo tiempo, garantizar que las partes encajan sin problemas al integrarse en la aplicación.

El enfoque tradicional para diseñar y crear una aplicación da como resultado lo que se conoce como una aplicación *monolítica*, donde los componentes están estrechamente

acoplados, sin una clara distinción entre ellos. Normalmente, este enfoque monolítico da como resultado aplicaciones difíciles de mantener de forma eficaz, ya que puede ser difícil resolver errores sin interrumpir otros componentes de la aplicación, así como agregar nuevas características o reemplazar las ya existentes.

Una solución eficaz en estos casos consiste en dividir una aplicación en componentes discretos y acoplados de manera imprecisa que puedan integrarse fácilmente en una aplicación. Este enfoque ofrece varias ventajas:

- Permite que diferentes individuos o equipos desarrollen, prueben, amplíen y mantengan funcionalidades individuales.
- Fomenta la reutilización y establece una clara distinción de los problemas entre las funcionalidades horizontales de la aplicación (como la autenticación y el acceso a datos) y las funcionalidades verticales (como la funcionalidad empresarial específica de la aplicación). Esto permite que las dependencias y las interacciones entre los componentes de la aplicación sean más fáciles de administrar.
- Ayuda a mantener una separación de los roles, al permitir que diferentes individuos, o equipos, se centren en una tarea o parte específica de la funcionalidad según su experiencia. En concreto, proporciona una distinción más clara entre la interfaz de usuario y la lógica de negocios de la aplicación.

Sin embargo, existen muchos problemas que se deben resolver al crear particiones de una aplicación para dividirla en componentes discretos y acoplados de manera imprecisa. Entre ellas se incluyen las siguientes:

- Decidir cómo proporcionar una clara distinción de los problemas entre los controles de la interfaz de usuario y su lógica. Una de las decisiones más importantes al crear una aplicación empresarial .NET MAUI es si colocar la lógica de negocios en archivos de código subyacente o si establecer una clara distinción de los problemas entre los controles de la interfaz de usuario y su lógica, a fin de que la aplicación sea más fácil de mantener y probar. Para obtener más información, vea [Modelo-Vista-Modelo de vista](#).
- Decidir si se va a usar un contenedor de inserción de dependencias. Los contenedores de inserción de dependencias reducen el acoplamiento de dependencias entre objetos, proporcionando para ello una instalación donde crear instancias de clases con sus dependencias insertadas y administrar su duración en función de la configuración del contenedor. Para obtener más información, consulte [Inserción de dependencias](#).
- Elegir entre eventos proporcionados por la plataforma o la comunicación mediante mensajes entre componentes de acoplamiento impreciso que no conviene vincular por referencias de objeto y de tipo. Para obtener más información, vea [Comunicación entre componentes de acoplamiento impreciso](#).

- Decidir cómo navegar entre las páginas, incluido cómo invocar la navegación y dónde debe residir la lógica de navegación. Para obtener más información, consulte [Navigation](#) (Navegación).
- Decidir cómo validar las entradas de usuario para comprobar su exactitud. La decisión debe englobar cómo validar las entradas de usuario y cómo informar a los usuarios de los errores de validación. Para obtener más información, vea [Validación](#).
- Decidir cómo realizar la autenticación y cómo proteger los recursos mediante autorización. Para obtener más información, vea [Autenticación y autorización](#).
- Decidir cómo acceder a datos remotos desde servicios web, incluido cómo recuperar datos de forma confiable y cómo almacenarlos en caché. Para obtener más información, vea [Acceso a datos remotos](#).
- Decidir cómo probar la aplicación. Para más información, consulte [Prueba unitaria](#).

Esta guía incluye instrucciones sobre estos problemas y se centra en los patrones y la arquitectura principales para crear una aplicación empresarial multiplataforma mediante .NET MAUI. El objetivo es ayudar a generar un código adaptable y fácil de mantener y de probar, abordando para ello algunos escenarios comunes de desarrollo de aplicaciones empresariales .NET MAUI y distinguiendo entre problemas de presentación, de lógica de presentación y de entidades mediante la compatibilidad con el patrón Modelo-Vista-Modelo de vista (MVVM).

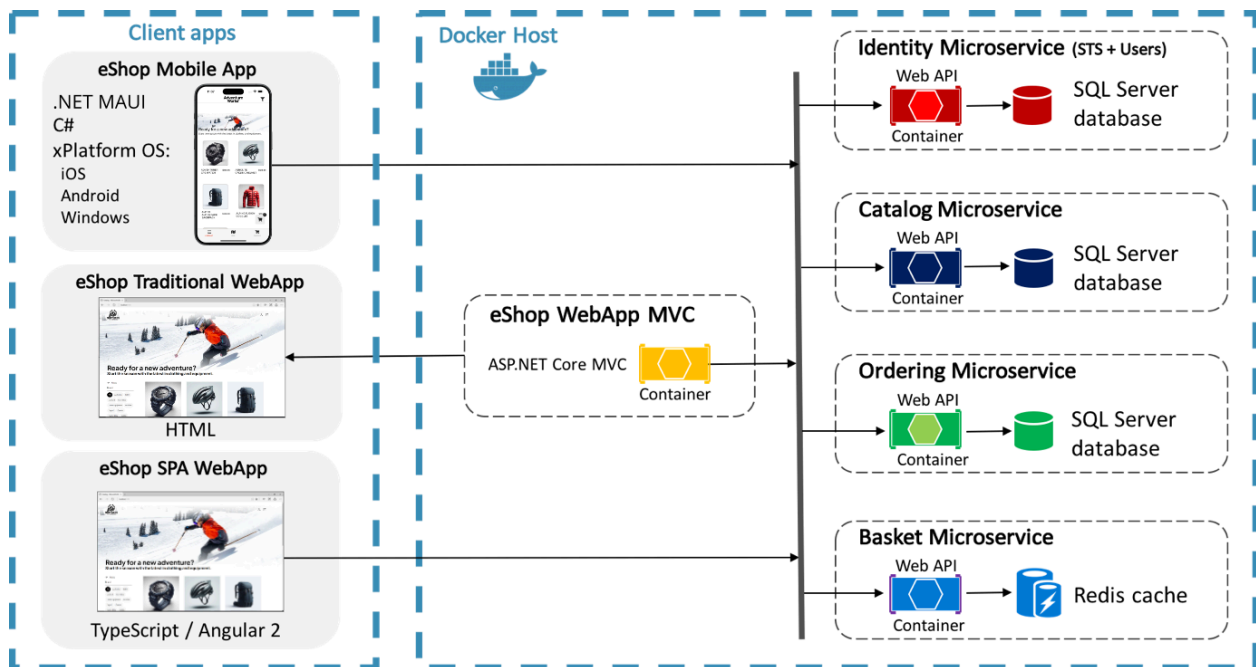
Aplicación de ejemplo

Esta guía incluye una aplicación de ejemplo, eShop, que es una tienda en línea que incluye las siguientes funcionalidades:

- Autenticación y autorización en un servicio back-end
- Examen de un catálogo de elementos.
- Filtrado del catálogo
- Ordenación de los artículos del catálogo
- Visualización del historial de pedidos del usuario
- Configuración de opciones

Arquitectura de una aplicación de ejemplo

Aquí se proporciona una descripción general de la arquitectura de una aplicación de ejemplo.



La aplicación de ejemplo viene con tres aplicaciones cliente:

- Una aplicación MVC desarrollada con ASP.NET Core.
- Una aplicación de página única (SPA) desarrollada con Angular 2 y Typescript. Este enfoque de aplicaciones web evita tener que realizar un recorrido de ida y vuelta al servidor con cada operación.
- Una aplicación multiplataforma desarrollada con .NET MAUI, que admite iOS, Android, macOS mediante Mac Catalyst y Windows 10/11.

Para obtener información sobre las aplicaciones web, vea [Diseño y desarrollo de aplicaciones web modernas con ASP.NET Core y Microsoft Azure](#).

La aplicación de ejemplo incluye los siguientes servicios back-end:

- Un microservicio de identidad que usa la identidad de ASP.NET Core e IdentityServer
- Un microservicio de catálogo, que es un servicio de creación, lectura, actualización, eliminación (CRUD) basado en datos que consume una base de datos de SQL Server mediante EntityFramework Core
- Un microservicio de ordenación, que es un servicio basado en dominio que usa patrones de diseño basados en dominio
- Un microservicio de cesta, que es un servicio CRUD basado en datos que usa Redis Cache

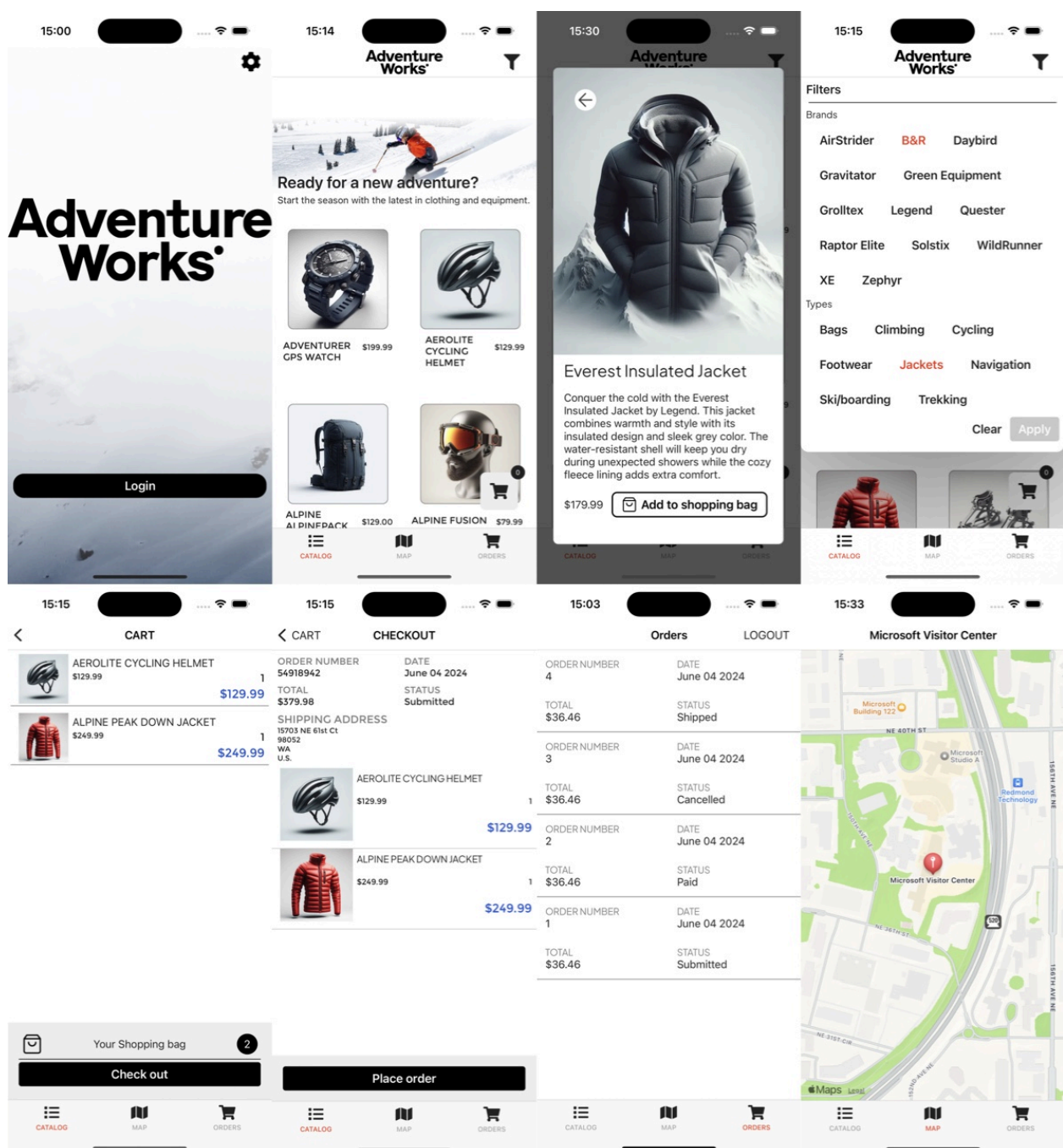
Estos servicios back-end se implementan como microservicios mediante ASP.NET Core MVC y se implementan como contenedores únicos dentro de un mismo host de Docker. En conjunto, estos servicios back-end se conocen como la aplicación de referencia eShop. Las aplicaciones cliente se comunican con los servicios back-end a través de una interfaz web de Transferencia de estado representacional (REST). Para

obtener más información sobre los microservicios y Docker, vea [Microservicios contenedorizados](#).

Para obtener información sobre la implementación de los servicios back-end, vea [Microservicios de .NET: Arquitectura de aplicaciones .NET contenedorizadas](#) [↗](#).

Aplicación multiplataforma

Esta guía versa sobre cómo crear aplicaciones empresariales multiplataforma mediante .NET MAUI, y en ella se usa la aplicación multiplataforma eShop como ejemplo. En la siguiente imagen se muestran las páginas de la aplicación multiplataforma eShop que proporcionan la funcionalidad descrita anteriormente.



La aplicación multiplataforma consume los servicios back-end proporcionados por la aplicación de referencia eShop. pero quienes no quieran implementar estos servicios back-end pueden configurarla para consumir datos de servicios ficticios.

La aplicación multiplataforma eShop pone en marcha la siguiente funcionalidad de .NET MAUI:

- XAML
- Controles
- Enlaces
- Convertidores
- Estilos
- Animaciones
- Comandos:
- Comportamientos
- Desencadenadores
- Efectos
- Controles personalizados

Para obtener más información sobre esta funcionalidad, vea la [documentación de .NET MAUI](#) en el Centro para desarrolladores de Microsoft y [Creación de aplicaciones multiplataforma con .NET MAUI](#) [↗](#).

Además, se proporcionan pruebas unitarias para algunas de las clases de la aplicación multiplataforma eShop.

Solución de aplicación multiplataforma

La solución de aplicación multiplataforma eShop organiza el código fuente y otros recursos en varios proyectos. Todos los componentes móviles principales están contenidos en un proyecto singular denominado eShopContainers. Se trata de una característica que apareció con .NET 6 y que permite que un proyecto tenga varias salidas como destino, lo que acaba con la necesidad de tener varios proyectos de plataforma, que habría sido el caso si hubiéramos usado Xamarin.Forms y versiones anteriores de .NET. Se incluye un proyecto más para las pruebas unitarias.

Aunque este proyecto tiene todos sus componentes almacenados en un proyecto singular, merece la pena estudiar la posibilidad de separarlo en varios proyectos en función de sus necesidades. Por ejemplo, si tiene varias implementaciones de proveedores de servicios basadas en un servicio con sus propias dependencias, quizá tenga sentido dividir esas implementaciones de proveedor de servicios en su proyecto propio correspondiente aparte. Algunos candidatos aptos para separar proyectos son

los modelos compartidos, las implementaciones de servicio, los componentes de cliente de API, las capas de base de datos o el almacenamiento en caché. Cualquier lugar donde crea que la empresa pueda reutilizar un componente en otro proyecto es teóricamente un buen candidato para establecer esta separación. Estos proyectos se pueden empaquetar a través de [NuGet](#) para facilitar la distribución y el control de versiones.

Todos los proyectos usan carpetas para organizar el código fuente y otros recursos en categorías. Las clases de la aplicación multiplataforma eShop se pueden reutilizar en cualquier otra aplicación .NET MAUI con apenas alguna (o ninguna) modificación.

Proyecto de eShop

El proyecto de eShop contiene las siguientes carpetas:

 Expandir tabla

Carpeta	Descripción
<i>Animaciones</i>	Contiene clases que permiten el consumo de animaciones en XAML.
<i>Comportamientos</i>	Contiene comportamientos que se exponen para ver clases.
<i>Controles</i>	Contiene controles personalizados usados por la aplicación.
<i>Convertidores</i>	Contiene convertidores de valores que aplican lógica personalizada a un enlace.
<i>Excepciones</i>	Contiene la excepción personalizada <code>ServiceAuthenticationException</code> .
<i>Extensiones</i>	Contiene los métodos de extensión de las clases <code>VisualElement</code> y <code>IEnumerable<T></code> .
<i>Asistentes</i>	Contiene clases auxiliares de la aplicación.
<i>Models</i>	Contiene las clases de modelo de la aplicación.
<i>Propiedades</i>	Contiene <code>AssemblyInfo.cs</code> , que es un archivo de metadatos de ensamblado .NET.
<i>Servicios</i>	Contiene las interfaces y clases que implementan servicios que se proporcionan a la aplicación.
<i>Desencadenadores</i>	Contiene el desencadenador <code>BeginAnimation</code> , que sirve para invocar una animación en XAML.
<i>Validaciones</i>	Contiene las clases que participan en la validación de entradas de datos.

Carpeta	Descripción
<i>ViewModels</i>	Contiene la lógica de la aplicación que se expone a las páginas.
<i>Vistas</i>	Contiene las páginas de la aplicación.

Resumen

Las plataformas y las herramientas de desarrollo de aplicaciones multiplataforma de Microsoft ofrecen una solución integral para aplicaciones cliente móviles B2E, B2B y B2C, lo que permite compartir código en todas las plataformas de destino (iOS, macOS, Android y Windows) y ayuda a reducir el costo total de propiedad. Las aplicaciones pueden compartir su interfaz de usuario y su código de lógica de la aplicación y, a la vez, conservar el aspecto de plataforma nativa.

Los desarrolladores de aplicaciones empresariales se enfrentan a diferentes dificultades que pueden alterar la arquitectura de la aplicación durante el desarrollo. Por eso es tan importante crear una aplicación que se pueda modificar o extender con el tiempo. Diseñar esta capacidad de adaptación puede ser complicado, y suele requerir la partición de una aplicación en componentes discretos y acoplados de manera imprecisa que puedan integrarse fácilmente en una aplicación.

Colaborar con nosotros en GitHub

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).



Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)

Modelo-Vista-Modelo de vista (MVVM)

Artículo • 10/09/2024

💡 Sugerencia

Este contenido es un extracto del libro electrónico "Patrones de aplicaciones empresariales con .NET MAUI", disponible en [Documentación de .NET](#) o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

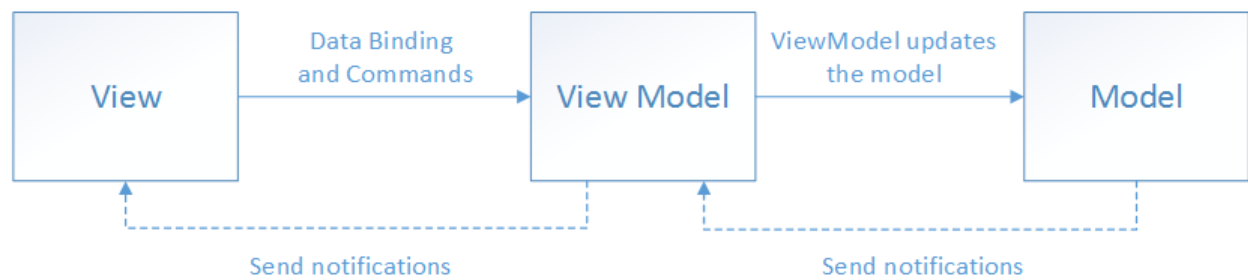


La experiencia del desarrollador de .NET MAUI normalmente supone crear una interfaz de usuario en XAML y, luego, agregar código subyacente que funcione en la interfaz de usuario. A medida que las aplicaciones se modifican y aumentan de tamaño y ámbito, pueden surgir problemas de mantenimiento complejos. Estos problemas incluyen el acoplamiento estricto entre los controles de interfaz de usuario y la lógica de negocios, lo que aumenta el costo de realizar modificaciones de la interfaz de usuario y la dificultad de realizar pruebas unitarias de este código.

El patrón MVVM ayuda a separar limpiamente la lógica de presentación y negocios de una aplicación de su interfaz de usuario (UI). Mantener una separación limpia entre la lógica de la aplicación y la interfaz de usuario ayuda a abordar numerosos problemas de desarrollo y facilita la prueba, el mantenimiento y la evolución de una aplicación. También puede mejorar considerablemente las oportunidades de reutilización del código y permite a los desarrolladores y a los diseñadores de la interfaz de usuario colaborar más fácilmente al desarrollar sus respectivas partes de una aplicación.

El patrón MVVM

Hay tres componentes principales en el patrón MVVM: el modelo, la vista y el modelo de vista. Cada uno de ellos sirve para un propósito diferente. En el diagrama siguiente se muestran las relaciones entre los tres componentes.



Además de comprender las responsabilidades de cada componente, también es importante comprender cómo interactúan. En general, la vista "conoce" el modelo de vista y el modelo de vista "conoce" el modelo, pero el modelo desconoce el modelo de vista y el modelo de vista desconoce la vista. Por lo tanto, el modelo de vista aísla la vista del modelo y permite que el modelo evolucione independientemente de la vista.

Las ventajas de usar el patrón MVVM son las siguientes:

- Si una implementación de modelo existente encapsula la lógica de negocios existente, puede ser difícil o arriesgado cambiarla. En este escenario, el modelo de vista actúa como adaptador para las clases de modelo y evita que realice cambios importantes en el código del modelo.
- Los desarrolladores pueden crear pruebas unitarias para el modelo de vista y el modelo, sin usar la vista. Las pruebas unitarias del modelo de vista pueden ejercer exactamente la misma funcionalidad que la vista.
- La interfaz de usuario de la aplicación se puede rediseñar sin tocar el modelo de vista y el código del modelo, siempre que la vista se implemente completamente en XAML o C#. Por lo tanto, una nueva versión de la vista debe funcionar con el modelo de vista existente.
- Los diseñadores y desarrolladores pueden trabajar de forma independiente y simultánea en sus componentes durante el desarrollo. Los diseñadores pueden centrarse en la vista, mientras que los desarrolladores pueden trabajar en el modelo de vista y los componentes del modelo.

La clave para usar MVVM de forma eficaz consiste en comprender cómo factorizar el código de la aplicación en las clases correctas y cómo interactúan las clases. En las secciones siguientes se describen las responsabilidades de cada una de las clases del patrón MVVM.

Ver

La vista es responsable de definir la estructura, el diseño y la apariencia de lo que ve el usuario en la pantalla. Idealmente, cada vista se define en XAML, con un código subyacente limitado que no contiene lógica de negocios. Sin embargo, en algunos casos, el código subyacente podría contener lógica de interfaz de usuario que

implementa el comportamiento visual que es difícil de expresar en XAML, como es el caso de las animaciones.

En una aplicación .NET MAUI, una vista suele ser una clase `ContentPage` derivada o una clase `ContentView` derivada. Sin embargo, las vistas también se pueden representar mediante una plantilla de datos, que especifica los elementos de la interfaz de usuario que se usarán para representar visualmente un objeto cuando se muestra. Una plantilla de datos como vista no tiene código subyacente y está diseñada para enlazar a un tipo de modelo de vista específico.

Sugerencia

Evite habilitar y deshabilitar los elementos de la interfaz de usuario en el código subyacente.

Asegúrese de que los modelos de vista sean responsables de definir los cambios de estado lógico que afectan a algunos aspectos de la presentación de la vista, por ejemplo, si un comando está disponible, o una indicación de que una operación está pendiente. Por lo tanto, habilite y deshabilite los elementos de la interfaz de usuario enlazando a las propiedades del modelo de vista, en lugar de habilitarlos y deshabilitarlos en el código subyacente.

Hay varias opciones para ejecutar código en el modelo de vista en respuesta a las interacciones de la vista, como un clic de botón o una selección de elementos. Si un control admite comandos, la propiedad `Command` del control puede estar enlazada por datos a una propiedad `ICommand` del modelo de vista. Cuando se invoca el comando del control, se ejecuta el código del modelo de vista. Además de los comandos, los comportamientos pueden estar asociados a un objeto en la vista y pueden escuchar el comando que se va a invocar o el evento que se va a generar. En respuesta, el comportamiento puede invocar una propiedad `ICommand` en el modelo de vista o un método en el modelo de vista.

ViewModel

El modelo de vista implementa propiedades y comandos a los que la vista puede enlazar datos, y avisa a la vista los cambios de estado mediante eventos de notificación de cambios. Las propiedades y comandos que proporciona el modelo de vista definen la funcionalidad que ofrece la interfaz de usuario, pero la vista determina cómo se va a mostrar esa funcionalidad.

Sugerencia

Mantenga la capacidad de respuesta de la interfaz de usuario con operaciones asincrónicas.

Las aplicaciones multiplataforma deben mantener desbloqueado el subproceso de interfaz de usuario para mejorar la percepción del rendimiento del usuario. Por lo tanto, en el modelo de vista, use métodos asincrónicos para las operaciones de E/S y genere eventos para notificar asincrónicamente las vistas de los cambios de propiedad.

El modelo de vista también es responsable de coordinar las interacciones de la vista con las clases de modelo necesarias. Normalmente hay una relación uno a varios entre el modelo de vista y las clases de modelo. El modelo de vista puede optar por exponer clases de modelo directamente a la vista para que los controles de la vista puedan enlazar datos directamente a ellas. En este caso, las clases de modelo deberán diseñarse para admitir el enlace de datos y los eventos de notificación de cambios.

Cada modelo de vista proporciona datos de un modelo en un formato que la vista puede consumir fácilmente. Para ello, el modelo de vista a veces realiza la conversión de datos. Es una buena idea situar esta conversión de datos en el modelo de vista porque proporciona propiedades que la vista puede enlazar. Por ejemplo, el modelo de vista podría combinar los valores de dos propiedades para facilitar la visualización por parte de la vista.

Sugerencia

Centralice las conversiones de datos en una capa de conversión.

También es posible usar convertidores como una capa de conversión de datos independiente que se encuentra entre el modelo de vista y la vista. Esto puede ser necesario, por ejemplo, cuando los datos requieren un formato especial que el modelo de vista no proporciona.

Para que el modelo de vista participe en el enlace de datos bidireccional con la vista, sus propiedades deben generar el evento `PropertyChanged`. Los modelos de vista satisfacen este requisito al implementar la interfaz `INotifyPropertyChanged` y generar el evento `PropertyChanged` cuando se cambia una propiedad.

Para las colecciones, se proporciona la vista descriptiva `ObservableCollection<T>`. Esta colección implementa una notificación de cambios en la colección, que evita al desarrollador tener que implementar la interfaz `INotifyCollectionChanged` en las colecciones.

Modelo

Las clases de modelo son clases no visuales que encapsulan los datos de la aplicación. Por lo tanto, el modelo se puede considerar como que representa el modelo de dominio de la aplicación, que normalmente incluye un modelo de datos junto con la lógica de validación y de negocios. Algunos ejemplos de objetos de modelo son objetos de transferencia de datos (DTO), objetos CLR antiguos sin formato (POCO) y objetos de entidad y proxy generados.

Las clases de modelo se suelen usar junto con servicios o repositorios que encapsulan el acceso a datos y el almacenamiento en caché.

Conexión de modelos de vista a vistas

Los modelos de vista se pueden conectar a las vistas mediante las funcionalidades de enlace de datos de .NET MAUI. Hay muchos enfoques que se pueden usar para construir vistas y ver modelos y asociarlos en tiempo de ejecución. Estos enfoques se dividen en dos categorías, conocidas como composición inicial de vista y composición inicial de modelo de vista. Elegir entre la composición inicial de vista y la composición inicial del modelo de vista es un asunto de preferencia y complejidad. Sin embargo, todos los enfoques comparten el mismo objetivo, que es que la vista tenga asignado un modelo de vista a su propiedad `BindingContext`.

Con la composición inicial de vista, la aplicación se compone conceptualmente de vistas que se conectan a los modelos de vista de los que dependen. La principal ventaja de este enfoque es que facilita la construcción de aplicaciones de acoplamiento flexible y comprobables por unidades, ya que los modelos de vista no dependen de las propias vistas. También es fácil comprender la estructura de la aplicación siguiendo su estructura visual, en lugar de tener que realizar un seguimiento de la ejecución de código para comprender cómo se crean y asocian las clases. Además, la construcción de primera vista se alinea con el sistema de navegación de Microsoft Maui que es responsable de construir páginas cuando se produce la navegación, lo que hace que una composición de primer modelo de vista sea complejo y esté mal alineado con la plataforma.

Con la composición inicial del modelo de vista, la aplicación se compone conceptualmente de modelos de vista, con un servicio responsable de localizar la vista para un modelo de vista. Para algunos desarrolladores, la composición inicial del modelo de vista resulta más natural, ya que la creación de la vista puede abstraerse, lo que les permite centrarse en la estructura lógica sin interfaz de usuario de la aplicación. Además, permite crear modelos de vista mediante otros modelos de vista. Sin embargo,

este enfoque suele ser complejo y puede resultar difícil comprender cómo se crean y asocian las distintas partes de la aplicación.

Sugerencia

Mantenga los modelos de vista y las vistas independientes.

El enlace de vistas a una propiedad de un origen de datos debe ser la dependencia principal de la vista en su modelo de vista correspondiente. En concreto, no haga referencia a tipos de vista, como `Button` y `ListView`, desde modelos de vista. Siguiendo los principios descritos aquí, los modelos de vista se pueden probar de forma aislada, lo que reduce la probabilidad de defectos de software al limitar el ámbito.

En las secciones siguientes se describen los enfoques principales para conectar modelos de vista a vistas.

Creación de un modelo de vista mediante declaración

El enfoque más sencillo es que la vista cree una instancia declarativa de su modelo de vista correspondiente en XAML. Cuando se construye la vista, también se construirá el objeto de modelo de vista correspondiente. Este enfoque se muestra en el ejemplo de código siguiente:

XAML

```
<ContentPage xmlns:local="clr-namespace:eShop">
  <ContentPage.BindingContext>
    <local:LoginViewModel />
  </ContentPage.BindingContext>
  <!-- Omitted for brevity... -->
</ContentPage>
```

Cuando se crea `ContentPage`, se crea automáticamente una instancia de `LoginViewModel` y se establece como el elemento `BindingContext` de la vista.

Esta construcción declarativa y la asignación del modelo de vista por la vista tiene la ventaja de que es simple, pero la desventaja de que requiere un constructor predeterminado (sin parámetros) en el modelo de vista.

Creación de un modelo de vista mediante programación

Una vista puede tener código en el archivo de código subyacente, lo que da lugar a que el modelo de vista se asigne a su propiedad `BindingContext`. Esto suele realizarse en el constructor de la vista, como se muestra en el ejemplo de código siguiente:

C#

```
public LoginView()
{
    InitializeComponent();
    BindingContext = new LoginViewModel(navigationService);
}
```

La construcción y asignación mediante programación del modelo de vista dentro del código subyacente de la vista tiene la ventaja de que es simple. Sin embargo, la principal desventaja de este enfoque es que la vista debe proporcionar al modelo de vista las dependencias necesarias. El uso de un contenedor de inserción de dependencias puede ayudar a mantener el acoplamiento flexible entre la vista y el modelo de vista. Para obtener más información, consulte [Inserción de dependencias](#).

Actualización de las vistas en respuesta a los cambios en el modelo o el modelo de vista subyacente

Todas las clases de modelo y modelos de vistas que son accesibles para una vista deben implementar la interfaz `INotifyPropertyChanged`. La implementación de esta interfaz en un modelo de vista o una clase de modelo permite a la clase proporcionar notificaciones de cambios a los controles enlazados a datos en la vista cuando cambia el valor de propiedad subyacente.

La aplicación debe diseñarse para el uso correcto de la notificación de cambio de propiedad, de modo que se cumplan los siguientes requisitos:

- Generar siempre un evento `PropertyChanged` si cambia el valor de una propiedad pública. No suponer que se puede omitir la generación del evento `PropertyChanged` debido al conocimiento de cómo se produce el enlace XAML.
- Generar siempre un evento `PropertyChanged` para las propiedades calculadas cuyos valores usan otras propiedades en el modelo de vista o modelo.

- Generar siempre el evento `PropertyChanged` al final del método que realiza un cambio de propiedad o cuando se sabe que el objeto está en un estado seguro. La generación del evento interrumpe la operación invocando los controladores del evento de forma sincrónica. Si este hecho se produce en medio de una operación, podría exponer el objeto a las funciones de devolución de llamada cuando se encuentra en un estado no seguro y actualizado parcialmente. Además, es posible que los eventos `PropertyChanged` desencadenen cambios en cascada. Por lo general, los cambios en cascada requieren que se realicen actualizaciones antes de que el cambio en cascada se pueda ejecutar de forma segura.
- Nunca generar un evento `PropertyChanged` si la propiedad no cambia. Esto significa que debe comparar los valores antiguos y nuevos antes de generar el evento `PropertyChanged`.
- Nunca generar el evento `PropertyChanged` durante el constructor de un modelo de vista si va a inicializar una propiedad. Los controles enlazados a datos de la vista no se habrán suscrito para recibir notificaciones de cambios en este momento.
- Nunca generar más de un evento `PropertyChanged` con el mismo argumento de nombre de propiedad dentro de una única invocación sincrónica de un método público de una clase. Por ejemplo, dada una propiedad `NumberOfItems` cuya memoria auxiliar es el campo `_numberOfItems`, si un método incrementa `_numberOfItems` cincuenta veces durante la ejecución de un bucle, solo debe generar una notificación de cambio de propiedad en la propiedad `NumberOfItems` una vez, después de que se complete todo el trabajo. Para los métodos asíncronos, genere el evento `PropertyChanged` para un nombre de propiedad determinado en cada segmento sincrónico de una cadena de continuación asíncrona.

Una manera sencilla de proporcionar esta funcionalidad sería crear una extensión de la clase `BindableObject`. En este ejemplo, la clase `ExtendedBindableObject` proporciona notificaciones de cambio, que se muestran en el ejemplo de código siguiente:

C#

```
public abstract class ExtendedBindableObject : BindableObject
{
    public void RaisePropertyChanged<T>(Expression<Func<T>> property)
    {
        var name = GetMemberInfo(property).Name;
        OnPropertyChanged(name);
    }

    private MemberInfo GetMemberInfo(Expression expression)
    {
        // Omitted for brevity ...
    }
}
```

```
}  
}
```

La clase `BindableObject` de .NET MAUI implementa la interfaz `INotifyPropertyChanged` y proporciona un método `OnPropertyChanged`. La clase `ExtendedBindableObject` proporciona el método `RaisePropertyChanged` para invocar la notificación de cambio de propiedad y, al hacerlo, usa la funcionalidad proporcionada por la clase `BindableObject`.

Las clases de modelo de vista se pueden derivar de la clase `ExtendedBindableObject`. Por lo tanto, cada clase de modelo de vista usa el método `RaisePropertyChanged` de la clase `ExtendedBindableObject` para proporcionar una notificación de cambio de propiedad. En el ejemplo de código siguiente se muestra cómo la aplicación multiplataforma eShop invoca la notificación de cambio de propiedad mediante una expresión lambda:

C#

```
public bool IsLogin  
{  
    get => _isLogin;  
    set  
    {  
        _isLogin = value;  
        RaisePropertyChanged(() => IsLogin);  
    }  
}
```

El uso de una expresión lambda de esta manera supone un pequeño costo de rendimiento porque esta expresión debe evaluarse con cada llamada. Aunque el costo de rendimiento es pequeño y normalmente no afectaría a una aplicación, los costos pueden acumularse cuando hay muchas notificaciones de cambio. Sin embargo, la ventaja de este enfoque es que proporciona compatibilidad con la seguridad y la refactorización de tipos en tiempo de compilación al cambiar el nombre de las propiedades.

Marcos MVVM

El patrón MVVM está bien establecido en .NET y la comunidad ha creado muchos marcos que ayudan a facilitar este desarrollo. Aunque cada marco proporciona un conjunto diferente de características, la norma es proporcionar un modelo de vista común con una implementación de la interfaz `INotifyPropertyChanged`. Entre las características adicionales de los marcos MVVM se incluyen comandos personalizados, asistentes de navegación, componentes de inserción de dependencias o localizador de servicios e integración de la plataforma de interfaz de usuario. Aunque no es necesario

usar estos marcos, pueden acelerar y estandarizar el desarrollo. La aplicación multiplataforma eShop usa [el kit de herramientas de MVVM de la comunidad .NET](#). Al elegir un marco, debe tener en cuenta las necesidades de la aplicación y los puntos fuertes de su equipo. En la lista siguiente se incluyen algunos de los marcos MVVM más comunes para .NET MAUI.

- [Kit de herramientas de MVVM de la comunidad .NET](#)
- [ReactiveUI](#) 
- [Prism Library](#) 

Interacción de la interfaz de usuario mediante comandos y comportamientos

En las aplicaciones multiplataforma, las acciones se suelen invocar en respuesta a una acción del usuario, como un clic de botón, que se puede implementar mediante la creación de un controlador de eventos en el archivo de código subyacente. Sin embargo, en el patrón MVVM, la responsabilidad de implementar la acción radica en el modelo de vista y se debe evitar colocar código en el código subyacente.

Los comandos proporcionan una manera cómoda de representar acciones que se pueden enlazar a controles de la interfaz de usuario. Encapsulan el código que implementa la acción y ayudan a evitar que se desacople de su representación visual en la vista. De este modo, los modelos de vista se vuelven más portátiles para las nuevas plataformas, ya que no tienen una dependencia directa de los eventos proporcionados por el marco de interfaz de usuario de la plataforma. .NET MAUI incluye controles que se pueden conectar mediante declaración a un comando y estos controles invocan el comando cuando el usuario interactúa con el control.

Los comportamientos también permiten que los controles se conecten mediante declaración a un comando. Sin embargo, se pueden usar comportamientos para invocar una acción asociada a un intervalo de eventos generados por un control. Por lo tanto, los comportamientos solucionan muchos de los mismos escenarios que los controles habilitados para comandos, a la vez que proporcionan un mayor grado de flexibilidad y control. Además, también se pueden usar comportamientos para asociar objetos o métodos de comandos a controles que no se han diseñado para interactuar con los comandos.

Implementación de comandos

Los modelos de vista suelen exponer propiedades públicas, para enlazar desde la vista, que implementan la interfaz `ICommand`. Muchos controles y gestos de .NET MAUI proporcionan una propiedad `Command`, que pueden ser datos enlazados a un objeto `ICommand` proporcionado por el modelo de vista. El control de botón es uno de los controles más usados, y proporciona una propiedad de comando que se ejecuta cuando se hace clic en el botón.

❗ Nota

Aunque es posible exponer la implementación real de la interfaz `ICommand` que usa el modelo de vista (por ejemplo, `Command<T>` o `RelayCommand`), se recomienda exponer los comandos públicamente como `ICommand`. De este modo, si más adelante necesita cambiar la aplicación, podrá hacerlo fácilmente.

La interfaz `ICommand` define un método `Execute`, que encapsula la propia operación, un método `CanExecute`, que indica si se puede invocar el comando y un evento `CanExecuteChanged` que se produce cuando tienen lugar cambios que afectan a la ejecución del comando. En la mayoría de los casos, solo se proporcionará el método `Execute` para nuestros comandos. Para información general más detallada de `ICommand`, consulte [Commanding](#) en la documentación de .NET MAUI.

Con .NET MAUI se proporcionan las clases `Command` y `Command<T>` que implementan la interfaz `ICommand`, donde `T` es el tipo de los argumentos para `Execute` y `CanExecute`. `Command` y `Command<T>` son implementaciones básicas que proporcionan el conjunto mínimo de funcionalidades necesarias para la interfaz `ICommand`.

❗ Nota

Muchos marcos MVVM ofrecen implementaciones con más características de la interfaz `ICommand`.

El constructor `Command` o `Command<T>` requiere un objeto de devolución de llamada `Action` al que se llama cuando se invoca el método `ICommand.Execute`. El método `CanExecute` es un parámetro de constructor opcional y es una función que devuelve un valor booleano.

La aplicación multiplataforma eShop usa los objetos [RelayCommand](#) y [AsyncRelayCommand](#). La principal ventaja para las aplicaciones modernas es que

`AsyncRelayCommand` proporciona una mejor funcionalidad para las operaciones asincrónicas.

En el código siguiente se muestra cómo se construye una instancia de `Command`, que representa un comando register, mediante la especificación de un delegado para el método de modelo de vista `Register`:

C#

```
public ICommand RegisterCommand { get; }
```

El comando se expone a la vista mediante una propiedad que devuelve una referencia a un objeto `ICommand`. Cuando se llama al método `Execute` en el objeto `Command`, simplemente se reenvía la llamada al método en el modelo de vista mediante el delegado especificado en el constructor `Command`. Un comando puede invocar un método asincrónico mediante las palabras clave `async` y `await` al especificar el delegado del `Execute` comando. Esto indica que la devolución de llamada es un objeto `Task` y se debe esperar. Por ejemplo, el código siguiente muestra cómo se construye una instancia de `ICommand`, que representa un comando de inicio de sesión, mediante la especificación de un delegado para el método de modelo de vista `SignInAsync`:

C#

```
public ICommand SignInCommand { get; }  
...  
SignInCommand = new AsyncRelayCommand(async () => await SignInAsync());
```

Los parámetros se pueden pasar a las acciones `Execute` y `CanExecute` mediante la clase `AsyncRelayCommand<T>` para crear una instancia del comando. Por ejemplo, el código siguiente muestra cómo se usa una instancia de `AsyncRelayCommand<T>` para indicar que el método `NavigateAsync` requerirá un argumento de tipo cadena:

C#

```
public ICommand NavigateCommand { get; }  
...  
NavigateCommand = new AsyncRelayCommand<string>(NavigateAsync);
```

En las clases `RelayCommand` y `RelayCommand<T>`, el delegado al método `CanExecute` en cada constructor es opcional. Si no se especifica un delegado, `Command` devolverá `true` en `CanExecute`. Sin embargo, el modelo de vista puede indicar un cambio en el estado

de `CanExecute` del comando mediante la llamada al método `ChangeCanExecute` en el objeto `Command`. Esto hace que se genere el evento `CanExecuteChanged`. Los controles de interfaz de usuario enlazados al comando actualizarán entonces su estado habilitado para reflejar la disponibilidad del comando enlazado a datos.

Invocación de comandos desde una vista

En el ejemplo de código siguiente se muestra cómo un objeto `Grid` de `LoginView` enlaza con `RegisterCommand` en la clase `LoginViewModel` mediante una instancia de `TapGestureRecognizer`:

XML

```
<Grid Grid.Column="1" HorizontalOptions="Center">
    <Label Text="REGISTER" TextColor="Gray"/>
    <Grid.GestureRecognizers>
        <TapGestureRecognizer Command="
{Binding RegisterCommand}" NumberOfTapsRequired="1" />
    </Grid.GestureRecognizers>
</Grid>
```

Un parámetro de comando también se puede definir opcionalmente mediante la propiedad `CommandParameter`. El tipo del argumento esperado se especifica en los métodos de destino `Execute` y `CanExecute`. El objeto `TapGestureRecognizer` invocará automáticamente el comando de destino cuando el usuario interactúe con el control asociado. El objeto `CommandParameter`, si se proporciona, se pasará como argumento al delegado `Execute` del comando.

Implementación de comportamientos

Los comportamientos permiten agregar funcionalidad a los controles de interfaz de usuario sin tener que aplicarles subclases. En su lugar, la función se implementa en una clase de comportamiento y se asocia al control como si fuera parte de este. Los comportamientos permiten implementar código que normalmente tendría que escribirse como código subyacente, ya que interactúa directamente con la API del control, de manera que puede asociarse al control de manera concisa y empaquetarse para reutilizarlo en más de una aplicación. En el contexto de MVVM, los comportamientos son un método útil para conectar controles a comandos.

Un comportamiento asociado a un control mediante propiedades asociadas se conoce como *comportamiento asociado*. El comportamiento puede usar entonces la API

expuesta del elemento al que está asociado para agregar funcionalidad a ese control, o a otros controles, en el árbol visual de la vista.

Un comportamiento de .NET MAUI es una clase que se obtiene de la clase `Behavior` o `Behavior<T>`, donde T es el tipo del control al que se debe aplicar el comportamiento. Estas clases proporcionan métodos `OnAttachedTo` y `OnDetachingFrom`, que se deben invalidar para proporcionar lógica que se ejecutará cuando el comportamiento se asocie a los controles y se desasocie de estos.

En la aplicación multiplataforma eShop, la clase `BindableBehavior<T>` se obtiene de la clase `Behavior<T>`. El propósito de la clase `BindableBehavior<T>` es proporcionar una clase base para cualquier comportamiento de .NET MAUI que requiera que el objeto `BindingContext` del comportamiento se establezca en el control asociado.

La clase `BindableBehavior<T>` proporciona un método `OnAttachedTo` reemplazable que establece el objeto `BindingContext` del comportamiento, y un método `OnDetachingFrom` reemplazable que limpia el objeto `BindingContext`.

La aplicación multiplataforma eShop incluye una clase `EventToCommandBehavior` que se proporciona en el kit de herramientas de la comunidad MAUI.

`EventToCommandBehavior` ejecuta un comando en respuesta a un evento que se produce. Esta clase se obtiene de la clase `BaseBehavior<View>`, de modo que el comportamiento puede enlazar con un objeto `ICommand` especificado por una propiedad `Command`, y ejecutarlo, cuando se usa el comportamiento. En el ejemplo de código siguiente se muestra la clase `EventToCommandBehavior`:

C#

```
/// <summary>
/// The <see cref="EventToCommandBehavior"/> is a behavior that allows the
/// user to invoke a <see cref="ICommand"/> through an event. It is designed to
/// associate Commands to events exposed by controls that were not designed to
/// support Commands. It allows you to map any arbitrary event on a control to a
/// Command.
/// </summary>
public class EventToCommandBehavior : BaseBehavior<VisualElement>
{
    // Omitted for brevity...

    /// <inheritdoc/>
    protected override void OnAttachedTo(VisualElement bindable)
    {
        base.OnAttachedTo(bindable);
        RegisterEvent();
    }
}
```

```

/// <inheritdoc>
protected override void OnDetachingFrom(VisualElement bindable)
{
    UnregisterEvent();
    base.OnDetachingFrom(bindable);
}

static void OnEventNamePropertyChanged(BindableObject bindable, object
oldValue, object newValue)
    => ((EventToCommandBehavior)bindable).RegisterEvent();

void RegisterEvent()
{
    UnregisterEvent();

    var eventName = EventName;
    if (View is null || string.IsNullOrEmpty(eventName))
    {
        return;
    }

    eventInfo = View.GetType()?.GetRuntimeEvent(eventName) ??
        throw new ArgumentException($"{nameof(EventToCommandBehavior)}:
Couldn't resolve the event.", nameof(EventName));

    ArgumentNullException.ThrowIfNull(eventInfo.EventHandlerType);
    ArgumentNullException.ThrowIfNull(eventHandlerMethodInfo);

    eventHandler =
eventHandlerMethodInfo.CreateDelegate(eventInfo.EventHandlerType, this) ??
        throw new ArgumentException($"{nameof(EventToCommandBehavior)}:
Couldn't create event handler.", nameof(EventName));

    eventInfo.AddEventHandler(View, eventHandler);
}

void UnregisterEvent()
{
    if (eventInfo is not null && eventHandler is not null)
    {
        eventInfo.RemoveEventHandler(View, eventHandler);
    }

    eventInfo = null;
    eventHandler = null;
}

/// <summary>
/// Virtual method that executes when a Command is invoked
/// </summary>
/// <param name="sender"></param>
/// <param name="eventArgs"></param>
[Microsoft.Maui.Controls.Internals.Preserve(Conditional = true)]
protected virtual void OnTriggerHandled(object? sender = null, object?
eventArgs = null)

```

```

{
    var parameter = CommandParameter
        ?? EventArgsConverter?.Convert(eventArgs, typeof(object), null,
null);

    var command = Command;
    if (command?.CanExecute(parameter) ?? false)
    {
        command.Execute(parameter);
    }
}
}

```

Los métodos `OnAttachedTo` y `OnDetachingFrom` se usan para registrar y anular el registro de un controlador de eventos para el evento definido en la propiedad `EventName`. A continuación, cuando se desencadena el evento, se invoca el método `OnTriggerHandled`, que ejecuta el comando.

La ventaja de usar `EventToCommandBehavior` para ejecutar un comando cuando se desencadena un evento es que se pueden asociar comandos con controles que no se han diseñado para interactuar con comandos. Además, el código de control de eventos se traslada a los modelos de vista, donde se pueden realizar pruebas unitarias.

Invocación de comportamientos desde una vista

`EventToCommandBehavior` es especialmente útil para asociar un comando a un control que no admite comandos. Por ejemplo, `LoginView` usa `EventToCommandBehavior` para ejecutar `ValidateCommand` cuando el usuario cambia el valor de su contraseña, como se muestra en el código siguiente:

XAML

```

<Entry
    IsPassword="True"
    Text="{Binding Password.Value, Mode=TwoWay}">
    <!-- Omitted for brevity... -->
    <Entry.Behaviors>
        <mct:EventToCommandBehavior
            EventName="TextChanged"
            Command="{Binding ValidateCommand}" />
    </Entry.Behaviors>
    <!-- Omitted for brevity... -->
</Entry>

```

En tiempo de ejecución, `EventToCommandBehavior` responderá a la interacción con `Entry`. Cuando un usuario escribe en el campo `Entry`, se desencadena el evento `TextChanged`, que ejecutará `ValidateCommand` en `LoginViewModel`. De forma predeterminada, los argumentos del evento se pasan al comando. Si es necesario, la propiedad `EventArgsConverter` se puede usar para convertir el objeto `EventArgs` proporcionado por el evento en un valor que el comando espera como entrada.

Para más información sobre los comportamientos, consulte [Comportamientos](#) en el Centro para desarrolladores de .NET MAUI.

Resumen

El patrón Modelo-Vista-Modelo de vista (MVVM) ayuda a separar limpiamente la lógica de presentación y de negocios de una aplicación de su interfaz de usuario (UI). Mantener una separación limpia entre la lógica de la aplicación y la interfaz de usuario ayuda a abordar numerosos problemas de desarrollo y facilita la prueba, el mantenimiento y la evolución de una aplicación. También puede mejorar considerablemente las oportunidades de reutilización del código y permite a los desarrolladores y a los diseñadores de la interfaz de usuario colaborar más fácilmente al desarrollar sus respectivas partes de una aplicación.

Con el patrón MVVM, la interfaz de usuario de la aplicación y la lógica de presentación y negocios subyacente se separan en tres clases distintas: la vista, que encapsula la lógica de la interfaz de usuario y la interfaz de usuario; el modelo de vista, que encapsula la lógica y el estado de la presentación; y el modelo, que encapsula la lógica de negocios y los datos de la aplicación.

Inserción de dependencia

Artículo • 30/05/2024

💡 Sugerencia

Este contenido es un extracto del libro electrónico "Patrones de aplicaciones empresariales con .NET MAUI", disponible en [Documentación de .NET](#) o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF



Normalmente, se invoca un constructor de clase al crear una instancia de un objeto, y los valores que el objeto necesita se pasan como argumentos al constructor. Este es un ejemplo de inserción de dependencias conocida como *inserción de constructores*. Las dependencias que necesita el objeto se insertan en el constructor.

Al especificar dependencias como tipos de interfaz, la inserción de dependencias permite desacoplar los tipos concretos del código que depende de estos tipos. Por lo general, usa un contenedor que contiene una lista de registros y asignaciones entre interfaces y tipos abstractos, y los tipos concretos que implementan o extienden estos tipos.

También hay otros tipos de inserción de dependencias, como la *inserción de establecedor de propiedades* y la *inserción de llamadas de método*, pero son menos frecuentes. Por lo tanto, este capítulo se centrará únicamente en la realización de la inserción de constructores con un contenedor de inserción de dependencias.

Introducción a la inserción de dependencias

La inserción de dependencias es una versión especializada del patrón de inversión de control (IoC), donde la preocupación que se invierte es el proceso de obtener la dependencia necesaria. Con la inserción de dependencias, otra clase es responsable de

insertar dependencias en un objeto en tiempo de ejecución. En el ejemplo de código siguiente se muestra cómo se estructura la clase `ProfileViewModel` al usar la inserción de dependencias:

C#

```
private readonly ISettingsService _settingsService;
private readonly IAppEnvironmentService _appEnvironmentService;

public ProfileViewModel(
    IAppEnvironmentService appEnvironmentService,
    IDialogService dialogService,
    INavigationService navigationService,
    ISettingsService settingsService)
    : base(dialogService, navigationService, settingsService)
{
    _appEnvironmentService = appEnvironmentService;
    _settingsService = settingsService;

    // Omitted for brevity
}
```

El constructor `ProfileViewModel` recibe varias instancias de objeto de interfaz como argumentos insertados por otra clase. La única dependencia de la clase `ProfileViewModel` es de los tipos de interfaz. Por lo tanto, la clase `ProfileViewModel` no tiene ningún conocimiento de la clase responsable de crear instancias de los objetos de interfaz. La clase responsable de crear instancias de los objetos de interfaz y de insertarlas en la clase `ProfileViewModel` se conoce como *contenedor de inserción de dependencias*.

Los contenedores de inserción de dependencias reducen el acoplamiento entre objetos proporcionando un recurso para crear instancias de clase y administrar su duración en función de la configuración del contenedor. Durante la creación de objetos, el contenedor inserta las dependencias que el objeto requiere. Si esas dependencias aún no se han creado, el contenedor crea y resuelve primero sus dependencias.

El uso de un contenedor de inserción de dependencias ofrece varias ventajas:

- Un contenedor elimina la necesidad de que una clase localice sus dependencias y administre sus duraciones.
- Un contenedor permite la asignación de dependencias implementadas sin afectar a la clase.
- Un contenedor facilita la comprobación, ya que permite simular las dependencias.
- Un contenedor aumenta la capacidad de mantenimiento al permitir que las nuevas clases se agreguen fácilmente a la aplicación.

En el contexto de una aplicación MAUI de .NET que usa MVVM, normalmente se usará un contenedor de inserción de dependencias para registrar y resolver vistas, registrar y resolver modelos de vista, y para registrar servicios e insertarlos en modelos de vista.

Hay muchos contenedores de inserción de dependencias disponibles en .NET; la aplicación multiplataforma eShop usa `Microsoft.Extensions.DependencyInjection` para administrar la creación de instancias de vistas, modelos de vista y clases de servicio en la aplicación. `Microsoft.Extensions.DependencyInjection` facilita la creación de aplicaciones de acoplamiento flexible y proporciona todas las características que se encuentran normalmente en contenedores de inserción de dependencias, incluidos métodos para registrar asignaciones de tipos e instancias de objeto, resolver objetos, administrar duraciones de objetos e insertar objetos dependientes en constructores de objetos que resuelve. Para más información sobre `Microsoft.Extensions.DependencyInjection`, vea [Inserción de dependencias en .NET](#).

En .NET MAUI, la clase `MauiProgram` llamará al método `CreateMauiApp` para crear un objeto `MauiAppBuilder`. El objeto `MauiAppBuilder` tiene una propiedad `Services` de tipo `IServiceCollection`, que proporciona una ubicación para registrar nuestros componentes, como vistas, modelos de vista y servicios para la inserción de dependencias. Los componentes registrados con la propiedad `Services` se proporcionarán al contenedor de inserción de dependencias cuando se llame al método `MauiAppBuilder.Build`.

En tiempo de ejecución, el contenedor debe saber qué implementación de los servicios se solicita para crear instancias de ellos para los objetos solicitados. En la aplicación multiplataforma eShop, las interfaces `IAppEnvironmentService`, `IDialogService`, `INavigationService` y `ISettingsService` deben resolverse para poder crear instancias de un objeto `ProfileViewModel`. Esto implica que el contenedor realice las siguientes acciones:

- Decidir cómo crear una instancia de un objeto que implementa la interfaz. Esto se conoce como *registro*.
- Crear instancias del objeto que implementa la interfaz necesaria y el objeto `ProfileViewModel`. Esto se conoce como *resolución*.

Al final, la aplicación terminará usando el objeto `ProfileViewModel` y estará disponible para la recolección de elementos no utilizados. En este momento, el recolector de elementos no utilizados debe eliminar las implementaciones de interfaz de corta duración si otras clases no comparten la misma instancia.

Registro

Para que las dependencias se puedan insertar en un objeto, los tipos de las dependencias deben registrarse primero en el contenedor. El registro de un tipo implica pasar al contenedor una interfaz y un tipo concreto que implementa la interfaz.

Hay dos maneras de registrar tipos y objetos en el contenedor mediante código:

- Registre un tipo o una asignación en el contenedor. Esto se conoce como registro transitorio. Cuando sea necesario, el contenedor compilará una instancia del tipo especificado.
- Registre un objeto existente en el contenedor como singleton. Cuando sea necesario, el contenedor devolverá una referencia al objeto existente.

❗ Nota

Los contenedores de inserción de dependencias no siempre son adecuados. La inserción de dependencias presenta una complejidad adicional y requisitos que podrían no ser adecuados o útiles para aplicaciones pequeñas. Si una clase no tiene dependencias o no es una dependencia para otros tipos, es posible que no tenga sentido colocarla en el contenedor. Además, si una clase tiene un único conjunto de dependencias que son integrales del tipo y nunca cambiará, es posible que no tenga sentido colocarla en el contenedor.

El registro de tipos que requieren inserción de dependencias debe realizarse en un único método de una aplicación. Este método se debe invocar al principio del ciclo de vida de la aplicación para asegurarse de que conoce las dependencias entre sus clases. La aplicación multiplataforma eShop ejecuta este método `MauiProgram.CreateMauiApp`. En el siguiente ejemplo de código se muestra cómo la aplicación multiplataforma eShop declara `CreateMauiApp` en la clase `MauiProgram`:

C#

```
public static class MauiProgram
{
    public static MauiApp CreateMauiApp()
    => MauiApp.CreateBuilder()
        .UseMauiApp<App>()
        // Omitted for brevity
        .RegisterAppServices()
        .RegisterViewModels()
        .RegisterViews()
        .Build();
}
```

El método `MauiApp.CreateBuilder` crea un objeto `MauiAppBuilder` que podemos usar para registrar nuestras dependencias. Deben registrarse muchas dependencias de la aplicación multiplataforma eShop, por lo que los métodos de extensión `RegisterAppServices`, `RegisterViewModels` y `RegisterViews` se crearon para ayudar a proporcionar un flujo de trabajo de registro organizado y fácil de mantener. En el código siguiente se muestra el método `RegisterViewModels`:

C#

```
public static MauiAppBuilder RegisterViewModels(this MauiAppBuilder
mauiAppBuilder)
{
    mauiAppBuilder.Services.AddSingleton<ViewModels.MainViewModel>();
    mauiAppBuilder.Services.AddSingleton<ViewModels.LoginViewModel>();
    mauiAppBuilder.Services.AddSingleton<ViewModels.BasketViewModel>();
    mauiAppBuilder.Services.AddSingleton<ViewModels.CatalogViewModel>();
    mauiAppBuilder.Services.AddSingleton<ViewModels.ProfileViewModel>();

    mauiAppBuilder.Services.AddTransient<ViewModels.CheckoutViewModel>();
    mauiAppBuilder.Services.AddTransient<ViewModels.OrderDetailViewModel>();
    mauiAppBuilder.Services.AddTransient<ViewModels.SettingsViewModel>();
    mauiAppBuilder.Services.AddTransient<ViewModels.CampaignViewModel>();

    mauiAppBuilder.Services.AddTransient<ViewModels.CampaignDetailsViewModel>();

    return mauiAppBuilder;
}
```

Este método recibe una instancia de `MauiAppBuilder`, y podemos usar la propiedad `Services` para registrar los modelos de vista. En función de las necesidades de la aplicación, es posible que tenga que agregar servicios con diferentes duraciones. En la tabla siguiente se proporciona información sobre cuándo puede elegir estas diferentes duraciones de registro:

[Expandir tabla](#)

Method	Descripción
<code>AddSingleton<T></code>	Crearé una única instancia del objeto que permanecerá durante la vigencia de la aplicación.
<code>AddTransient<T></code>	Crearé una instancia del objeto cuando se solicite durante la resolución. Los objetos transitorios no tienen una duración predefinida, pero normalmente durarán lo mismo que su host.

❗ Nota

Los modelos de vista no heredan de una interfaz, por lo que solo necesitan su tipo concreto proporcionado a los métodos `AddSingleton<T>` y `AddTransient<T>`.

`CatalogViewModel` se usa cerca de la raíz de la aplicación y siempre debe estar disponible, por lo que es beneficioso registrarlo en `AddSingleton<T>`. Se navega a otros modelos de vista, como `CheckoutViewModel` y `OrderDetailViewModel`, en función de la situación, o bien se usan más tarde en la aplicación. Supongamos que sabe que tiene un componente que puede no usarse siempre. En ese caso, si requiere un uso intensivo de la memoria o de cálculo o requiere datos Just-In-Time, puede ser un mejor candidato para el registro de `AddTransient<T>`.

Otra manera común de agregar servicios es usar los métodos `AddSingleton<TService, TImplementation>` y `AddTransient<TService, TImplementation>`. Estos métodos usan dos tipos de entrada: la definición de interfaz y la implementación concreta. Este tipo de registro es más conveniente para los casos en los que se implementan servicios basados en interfaces. En el ejemplo de código siguiente, registramos nuestra interfaz `ISettingsService` mediante la implementación de `SettingsService`:

C#

```
public static MauiAppBuilder RegisterAppServices(this MauiAppBuilder
mauiAppBuilder)
{
    mauiAppBuilder.Services.AddSingleton<ISettingsService, SettingsService>
();
    // Omitted for brevity...
}
```

Una vez registrados todos los servicios, se debe llamar al método `MauiAppBuilder.Build` para crear nuestro `MauiApp` y rellenar nuestro contenedor de inserción de dependencias con todos los servicios registrados.

❗ Importante

Una vez que se ha llamado al método `Build`, el contenedor de inserción de dependencias es inmutable y ya no se puede actualizar ni modificar. Asegúrese de que todos los servicios que necesite dentro de la aplicación se hayan registrado antes de llamar a `Build`.

Solución

Una vez registrado un tipo, se puede resolver o insertar como una dependencia. Cuando se resuelve un tipo y el contenedor debe crear una instancia, inserta todas las dependencias en la instancia.

Por lo general, cuando se resuelve un tipo, sucede una de estas tres cosas:

1. Si el tipo no se ha registrado, el contenedor produce una excepción.
2. Si el tipo se ha registrado como singleton, el contenedor devuelve la instancia singleton. Si es la primera vez que se llama al tipo, el contenedor lo crea si es necesario y mantiene una referencia a él.
3. Si el tipo se ha registrado como transitorio, el contenedor devuelve una nueva instancia y no mantiene una referencia a él.

.NET MAUI ofrece varias maneras de resolver los componentes registrados en función de sus necesidades. La manera más directa de acceder al contenedor de inserción de dependencias es a partir de un elemento `Element` mediante

`Handler.MauiContext.Services`. A continuación se muestra un ejemplo:

C#

```
var settingsService =  
this.Handler.MauiContext.Services.GetService<ISettingsService>();
```

Esto puede resultar útil si necesita resolver un servicio desde dentro de `Element` o desde fuera del constructor de `Element`.

⊗ Precaución

Existe la posibilidad de que la propiedad `Handler` de su `Element` sea null, por lo que debe tener en cuenta que es posible que tenga que controlar esas situaciones. Para obtener más información, consulte [Ciclo de vida del controlador](#) en el Centro de documentación de Microsoft.

Si usa el control `Shell` para .NET MAUI, llamará implícitamente al contenedor de inserción de dependencias para crear los objetos durante la navegación. Al configurar nuestro control `Shell`, el método `Routing.RegisterRoute` vinculará una ruta de acceso a `View` como se muestra en el ejemplo siguiente:

C#

```
Routing.RegisterRoute("Filter", typeof(FiltersView));
```

Durante la navegación por `Shell`, buscará registros de `FiltersView` y, si se encuentra alguno, creará esa vista e insertará las dependencias en el constructor. Como se muestra en el ejemplo de código siguiente, `CatalogViewModel` se insertará en `FiltersView`:

C#

```
namespace eShop.Views;

public partial class FiltersView : ContentPage
{
    public FiltersView(CatalogViewModel viewModel)
    {
        BindingContext = viewModel;

        InitializeComponent();
    }
}
```

Sugerencia

El contenedor de inserción de dependencias es ideal para crear instancias de modelo de vista. Si un modelo de vista tiene dependencias, controlará la creación e inserción de todos los servicios necesarios. Solo tiene que asegurarse de registrar los modelos de vista y las dependencias que puedan tener con el método `CreateMauiApp` en la clase `MauiProgram`.

Resumen

La inserción de dependencias permite desacoplar tipos concretos del código que depende de estos tipos. Normalmente usa un contenedor que contiene una lista de registros y asignaciones entre interfaces y tipos abstractos, y los tipos concretos que implementan o extienden estos tipos.

`Microsoft.Extensions.DependencyInjection` facilita la creación de aplicaciones de acoplamiento flexible y proporciona todas las características que se encuentran normalmente en contenedores de inserción de dependencias, incluidos métodos para registrar asignaciones de tipos e instancias de objeto, resolver objetos, administrar duraciones de objetos e insertar objetos dependientes en los constructores de objetos que resuelve.

Colaborar con nosotros en GitHub

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).

.NET

Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)

Comunicación entre componentes acoplados de forma imprecisa

Artículo • 30/05/2024

💡 Sugerencia

Este contenido es un extracto del libro electrónico "Patrones de aplicaciones empresariales con .NET MAUI", disponible en [Documentación de .NET](#) o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF



El patrón de publicación y suscripción es un patrón de mensajería en el que los publicadores envían mensajes sin tener conocimiento de los destinatarios, que se conocen como suscriptores. Del mismo modo, los suscriptores escuchan mensajes específicos, sin tener conocimiento de ningún publicador.

Los eventos de .NET implementan el patrón de publicación y suscripción, y son el enfoque más sencillo para una capa de comunicación entre los componentes si no se requiere el acoplamiento flexible, como sucede con un control y la página que lo contiene. Pero las duraciones del publicador y del suscriptor están acopladas por referencias a objetos, y el tipo de suscriptor debe tener una referencia al tipo de publicador. Esto puede generar problemas de administración de memoria, en especial cuando hay objetos de corta duración que se suscriben a un evento de un objeto estático o de larga duración. Si no se quita el controlador de eventos, el suscriptor se mantendrá activo mediante la referencia a él en el publicador y esto impedirá o retrasará la recolección de elementos no utilizados del suscriptor.

Introducción a MVVM Toolkit Messenger

La interfaz de MVVM Toolkit `IMessenger` implementa el patrón de publicación y suscripción, lo que permite la comunicación basada en mensajes entre los componentes que no se recomienda vincular mediante referencias de tipo y objeto. Este mecanismo permite a los publicadores y suscriptores comunicarse sin que exista una referencia directa entre sí, lo que ayuda a reducir las dependencias entre los componentes, a la vez que permite desarrollar y probar esos componentes de forma independiente.

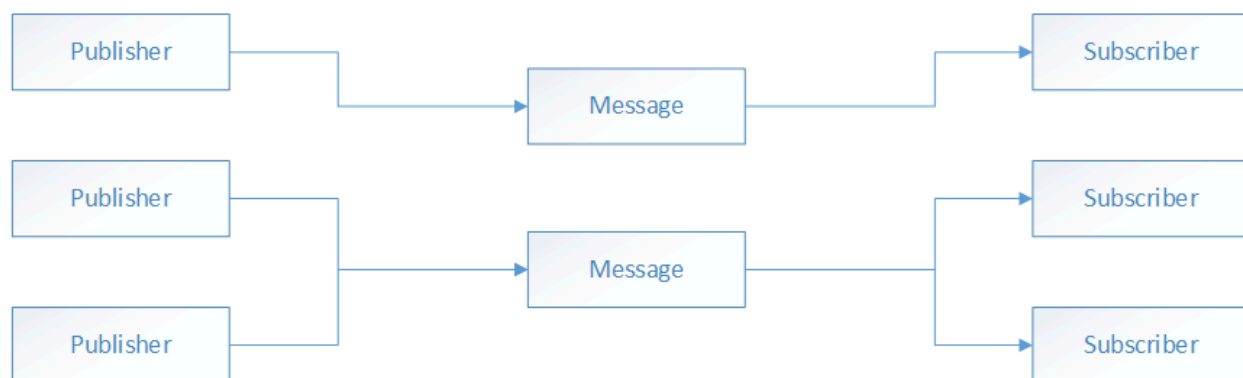
❗ Nota

MVVM Toolkit Messenger forma parte del paquete de `CommunityToolkit.Mvvm`. Para obtener información sobre cómo agregar el paquete al proyecto, consulte la [Introducción MVVM Toolkit](#) en el Centro para desarrolladores de Microsoft.

⚠ Advertencia

.NET MAUI contiene una clase de `MessagingCenter` integrada que ya no se recomienda para su uso. Use MVVM Toolkit Messenger en su lugar.

La interfaz de `IMessenger` permite la funcionalidad de publicación y suscripción de multidifusión. Esto significa que puede haber varios publicadores que publican un único mensaje, y puede haber varios suscriptores que escuchen el mismo mensaje. En la siguiente imagen se ilustra esta relación:



Hay dos implementaciones de la interfaz de `IMessenger` que vienen con el paquete de `CommunityToolkit.Mvvm`. `WeakReferenceMessenger` usa referencias débiles, que pueden dar lugar a una limpieza más sencilla para los suscriptores de mensajes. Es una buena opción si los suscriptores no tienen un ciclo de vida claramente definido.

`StrongReferenceMessenger` usa referencias seguras que pueden dar lugar a un mejor rendimiento y una duración más claramente controlada de la suscripción. Si tiene un flujo de trabajo con una duración muy controlada (por ejemplo, una suscripción enlazada a los métodos `OnAppearing` y `OnDisappearing` de una página),

`StrongReferenceManager` puede ser una opción mejor, si el rendimiento es un problema. Ambas implementaciones están disponibles con implementaciones predeterminadas listas para usar haciendo referencia a `WeakReferenceMessenger.Default` o `StrongReferenceMessenger.Default`.

❗ Nota

Aunque la interfaz `IMessenger` permite la comunicación entre clases de acoplamiento impreciso, no ofrece la única solución arquitectónica a este problema. Por ejemplo, la comunicación entre un modelo de vista y una vista también se puede lograr mediante el motor de enlace y a través de notificaciones de cambio de propiedad. Asimismo, la comunicación entre dos modelos de vista también se puede lograr pasando datos durante la navegación.

La aplicación multiplataforma eShop usa la clase `WeakReferenceMessenger` para comunicarse entre componentes de acoplamiento impreciso. La aplicación define un solo mensaje denominado `AddProductMessage`. La clase `CatalogViewModel` publica `AddProductMessage` cuando se agrega un elemento a la cesta de la compra. A cambio, la clase `CatalogView` se suscribe al mensaje y lo usa en respuesta para resaltar las adiciones de productos con una animación.

En la aplicación multiplataforma eShop, se usa `WeakReferenceMessenger` para actualizar en la interfaz de usuario en respuesta a una acción que se produce en otra clase. Por lo tanto, los mensajes se publican desde el subproceso en el que se ejecuta la clase, con suscriptores que reciben el mensaje en el mismo subproceso.

💡 Sugerencia

Serialice en la interfaz de usuario o en el subproceso principal al realizar actualizaciones de interfaz de usuario. Si las actualizaciones de interfaces de usuario no se realizan en este subproceso, puede suceder que la aplicación se bloquee o se vuelva inestable.

Si para actualizar la interfaz de usuario se requiere un mensaje enviado desde un subproceso en segundo plano, procese dicho mensaje en el subproceso de interfaz de usuario del suscriptor, invocando para ello el método `MainThread.BeginInvokeOnMainThread`.

Para obtener más información sobre `Messenger`, consulte [Messenger](#) en el Centro para desarrolladores de Microsoft.

Definición de un mensaje

Los mensajes de `IMessenger` son objetos personalizados que proporcionan cargas personalizadas. En el ejemplo de código siguiente se muestra el mensaje

`AddProductMessage` definido en la aplicación multiplataforma eShop:

C#

```
public class AddProductMessage : ValueChangedMessage<int>
{
    public AddProductMessage(int count) : base(count)
    {
    }
}
```

La clase base se define mediante `ValueChangedMessage<T>`, donde `T` puede ser de cualquier tipo necesario para pasar datos. Tanto los publicadores de mensajes como los suscriptores pueden esperar mensajes de un tipo específico (por ejemplo, `AddProductMessage`). Esto puede ayudar a garantizar que ambas partes hayan acordado un contrato de mensajería y que los datos proporcionados con ese contrato sean coherentes. Además, este enfoque proporciona compatibilidad con la seguridad y la refactorización de tipos en tiempo de compilación.

Publicación de un mensaje

Para publicar un mensaje, es necesario usar el método `IMessenger.Send`. Se puede acceder a esto con más frecuencia a través de `WeakReferenceMessenger.Default.Send` o `StrongReferenceMessenger.Default.Send`. El mensaje enviado puede ser de cualquier tipo de objeto. En el ejemplo de código siguiente se muestra cómo publicar el mensaje `AddProduct`:

C#

```
WeakReferenceMessenger.Default.Send(new
    Messages.AddProductMessage(BadgeCount));
```

En este ejemplo, el método `Send` especifica que se proporciona una nueva instancia del objeto `AddProductMessage` para la recepción de los suscriptores descendentes. Se puede agregar un segundo parámetro de token adicional para usarlo cuando varios suscriptores diferentes necesiten recibir mensajes del mismo tipo sin recibir el mensaje incorrecto.

El método `Send` publicará el mensaje y sus datos de carga mediante un enfoque del tipo "dispara y olvida". Por lo tanto, el mensaje se envía incluso aunque no haya ningún suscriptor registrado para recibir el mensaje. En esta situación, se omite el mensaje enviado.

Suscripción a un mensaje

Los suscriptores pueden registrarse para recibir un mensaje mediante una de las sobrecargas de `IMessenger.Register<T>`. En el ejemplo de código siguiente se describe cómo la aplicación multiplataforma eShop se suscribe al mensaje `AddProductMessage` y lo procesa:

C#

```
WeakReferenceMessenger.Default
    .Register<CatalogView, Messages.AddProductMessage>(
        this,
        async (recipient, message) =>
        {
            await recipient.Dispatcher.DispatchAsync(
                async () =>
                {
                    await recipient.badge.ScaleTo(1.2);
                    await recipient.badge.ScaleTo(1.0);
                });
        });
```

En el ejemplo anterior, el método `Register` se suscribe al mensaje `AddProductMessage` y ejecuta un delegado de devolución de llamada en respuesta a la recepción del mensaje. Este delegado de devolución de llamada, especificado como una expresión lambda, ejecuta código que actualiza la interfaz de usuario.

⚠ Nota

Evite el uso de `this` dentro del delegado de devolución de llamada para evitar capturar ese objeto dentro del delegado. Esto puede ayudar a mejorar el rendimiento. En su lugar, use el parámetro `recipient`.

Si se proporcionan datos de carga, no intente modificarlos desde dentro de un delegado de devolución de llamada, ya que varios subprocesos podrían estar accediendo a los datos recibidos simultáneamente. En este escenario, los datos de carga deben ser inmutables para evitar errores de simultaneidad.

Cancelación de la suscripción a un mensaje

Los suscriptores pueden cancelar la suscripción a los mensajes que ya no quieren recibir. Esto se logra con uno de las sobrecargas `IMessenger.Unregister`, como se muestra en el ejemplo de código siguiente:

C#

```
WeakReferenceMessenger.Default.Unregister<Messages.AddProductMessage>(this);
```

ⓘ Nota

En este ejemplo, no es totalmente necesario llamar a `Unregister`, ya que `WeakReferenceMessenger` permitirá la recolección residual de los objetos no utilizados. Si se usara `StrongReferenceMessenger`, se recomienda llamar a `Unregister` para cualquier suscripción que ya no esté en uso.

En este ejemplo, la sintaxis del método `Unsubscribe` especifica el argumento de tipo del mensaje y el objeto receptor que escucha los mensajes.

Resumen

La interfaz de MVVM Toolkit `IMessenger` implementa el patrón de publicación y suscripción, lo que permite la comunicación basada en mensajes entre los componentes que no se recomienda vincular mediante referencias de tipo y objeto. Este mecanismo permite a los publicadores y suscriptores comunicarse sin que exista una referencia entre sí, lo que ayuda a reducir las dependencias entre los componentes, a la vez que permite desarrollar y probar esos componentes de forma independiente.

Colaborar con nosotros en GitHub

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte

.NET

Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

 [Abrir incidencia con la documentación](#)

nuestra guía para
colaboradores.

 Proporcionar comentarios sobre
el producto

Navegación

Artículo • 30/05/2024

💡 Sugerencia

Este contenido es un extracto del libro electrónico "Patrones de aplicaciones empresariales con .NET MAUI", disponible en [Documentación de .NET](#) o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF



.NET MAUI incluye compatibilidad con la navegación por páginas, que normalmente surge a partir de la interacción del usuario con la UI o de la propia aplicación, como resultado de los cambios de estado internos causados por la lógica. Sin embargo, la navegación puede ser compleja de implementar en aplicaciones que usan el patrón Modelo-Vista-Modelo de vista (MVVM), ya que se deben superar los siguientes desafíos:

- Identificar la vista a la que se va a navegar mediante un enfoque que no introduce acoplamientos y dependencias estrictos entre las vistas.
- Coordinar el proceso por el que se crea y se inicializa una instancia de la vista a la que se va a navegar. Al usar MVVM, es necesario crear instancias del modelo de vista y de la vista y asociarlas entre sí mediante el contexto de enlace de la vista. Cuando una aplicación usa un contenedor de inserción de dependencias, la creación de instancias de vistas y modelos de vistas podría requerir un mecanismo de construcción específico.
- Saber si se va a realizar la navegación con prioridad para la vista o para el modelo de vista. Si se prioriza la vista, la página a la que se navega hace referencia al nombre del tipo de vista. Durante la navegación, se crea una instancia de la vista especificada, junto con su modelo de vista correspondiente y otros servicios dependientes. Un enfoque alternativo consiste en priorizar el modelo de vista. Así, la página a la que se navega hace referencia al nombre del tipo de modelo de vista.

- Determinar cómo separar limpiamente el comportamiento de navegación de la aplicación entre las vistas y los modelos de vistas. El patrón MVVM separa la UI de la aplicación, su presentación y su lógica de negocios, pero no proporciona un mecanismo directo para unirlos. Sin embargo, el comportamiento de navegación de una aplicación suele abarcar la UI y las partes de presentación de la aplicación. El usuario suele iniciar la navegación desde una vista y la vista se reemplazará como resultado de la navegación. Sin embargo, es posible que, con cierta asiduidad, la navegación también tenga que iniciarse o coordinarse desde dentro del modelo de vista.
- Determinar cómo pasar parámetros durante la navegación con fines de inicialización. Por ejemplo, si el usuario navega a una vista para actualizar los detalles de una orden, los datos de la orden tendrán que pasarse a la vista para que pueda mostrar los datos correctos.
- Coordinar la navegación para garantizar que se cumplen las reglas específicas del negocio. Por ejemplo, es posible que, antes de salir de una vista, se solicite a los usuarios que corrijan datos no válidos o que envíen o descarten los cambios de datos realizados en la vista.

En este capítulo se abordan estos desafíos mediante la presentación de una clase de servicio de navegación denominada `MauiNavigationService`, que se usa para realizar la navegación de la página con prioridad para el modelo de vista.

ⓘ Nota

La clase `MauiNavigationService` que utiliza la aplicación es simplista y no abarca todos los tipos de navegación posibles. Los tipos de navegación necesarios para la aplicación pueden requerir funcionalidades adicionales.

Navegación entre las páginas

La lógica de navegación puede residir en el código subyacente de una vista o en un modelo de vista enlazado a datos. Aunque colocar la lógica de navegación en una vista podría ser el enfoque más directo, no se puede probar fácilmente con pruebas unitarias. Si se coloca la lógica de navegación en las clases de modelo de vista, la lógica se puede comprobar mediante pruebas unitarias. Además, el modelo de vista puede implementar lógica para controlar la navegación y así asegurar que se aplican determinadas reglas del negocio. Por ejemplo, es posible que una aplicación no permita que el usuario salga de una página sin asegurarse primero de que los datos especificados sean válidos.

Normalmente, se invoca un servicio de navegación a partir de modelos de vistas, con el fin de promover la capacidad de realizar pruebas con ellos. Sin embargo, navegar a las vistas desde los modelos de vistas requeriría que los modelos de vistas hagan referencia a las vistas y, particularmente, a las vistas con las que el modelo de vista activo no está asociado, algo que no es recomendable. Por lo tanto, el elemento MauiNavigationService que se presenta aquí especifica el tipo de modelo de vista como destino al que navegar.

La aplicación multiplataforma eShop usa la clase MauiNavigationService para proporcionar navegación con prioridad para el modelo de vista. Esta clase implementa la interfaz INavigationService, que se muestra en el siguiente código de ejemplo:

```
C#

public interface INavigationService
{
    Task InitializeAsync();

    Task NavigateToAsync(string route, IDictionary<string, object>
routeParameters = null);

    Task PopAsync();
}
```

Esta interfaz especifica que una clase de implementación debe proporcionar los siguientes métodos:

 Expandir tabla

Método	Propósito
InitializeAsync	Realiza la navegación a una de las dos páginas cuando se inicia la aplicación.
NavigateToAsync(string route, IDictionary<string, object> routeParameters = null)	Realiza la navegación jerárquica a una página especificada mediante una ruta de navegación registrada. Opcionalmente, puede pasar parámetros de ruta con nombre, que se usarán para su procesamiento en la página de destino.
PopAsync	Quita la página actual de la pila de navegación.

 **Nota**

Normalmente, una interfaz `INavigationService` también especificaría un método `GoBackAsync`, que se usa para volver mediante programación a la página anterior de la pila de navegación. Sin embargo, este método no está en la aplicación multiplataforma eShop porque no es necesario.

Creación de la instancia de MauiNavigationService

La clase `MauiNavigationService`, que implementa la interfaz `INavigationService`, se registra como singleton con el contenedor de inserción de dependencias en el método `MauiProgram.CreateMauiApp()`, como se muestra en el siguiente código de ejemplo:

C#

```
mauiAppBuilder.Services.AddSingleton<INavigationService,>  
    MauiNavigationService>();;
```

La interfaz `INavigationService` se puede resolver agregándola al constructor de nuestras vistas y modelos de vistas, como se muestra en el siguiente código de ejemplo:

C#

```
public AppShell(INavigationService navigationService)
```

Esto devuelve una referencia al objeto `MauiNavigationService` almacenado en el contenedor de inserción de dependencias.

La clase `ViewModelBase` almacena la instancia `MauiNavigationService` en una propiedad `NavigationService`, de tipo `INavigationService`. Por lo tanto, todas las clases de modelos de vistas, que se derivan de la clase `ViewModelBase`, pueden usar la propiedad `NavigationService` para tener acceso a los métodos especificados por la interfaz `INavigationService`.

Control de solicitudes de navegación

.NET MAUI ofrece varias formas de navegar dentro de una aplicación. La forma tradicional de navegar es con la clase `NavigationPage`, que implementa una experiencia de navegación jerárquica en la que el usuario puede navegar por páginas, hacia delante y hacia atrás, según considere. La aplicación eShop usa el componente `Shell` como

contenedor raíz para la aplicación y como host de navegación. Para más información sobre la navegación de Shell, consulte [Navegación de Shell](#) en el Centro para desarrolladores de Microsoft.

La navegación se realiza dentro de las clases de modelos de vistas invocando uno de los métodos `NavigateToAsync`, especificando la ruta de acceso de la página a la que se navega, como se muestra en el siguiente código de ejemplo:

C#

```
await NavigationService.NavigateToAsync("//Main");
```

En el siguiente código de ejemplo se muestra el método `NavigateToAsync` proporcionado por la clase `MauiNavigationService`:

C#

```
public Task NavigateToAsync(string route, IDictionary<string, object>
routeParameters = null)
{
    return
        routeParameters != null
            ? Shell.Current.GoToAsync(route, routeParameters)
            : Shell.Current.GoToAsync(route);
}
```

El control .NET MAUI `Shell` ya está familiarizado con la navegación basada en rutas, por lo que el método `NavigateToAsync` funciona para enmascarar esta funcionalidad. El método `NavigateToAsync` permite especificar datos de navegación como un argumento que se pasa al modelo de vista al que se navega, donde normalmente se usa para realizar la inicialización. Para más información, consulte [Pasar parámetros durante la navegación](#).

Importante

Hay varias formas de realizar la navegación en .NET MAUI. `MauiNavigationService` se compila específicamente para trabajar con `Shell`. Si usa `NavigationPage`, `TabbedPage` o un mecanismo de navegación diferente, este servicio de enrutamiento tendría que actualizarse para que funcione con esos componentes.

Para registrar rutas para `MauiNavigationService` tenemos que proporcionar información de ruta desde XAML o en el código subyacente. En el ejemplo siguiente se muestra el registro de rutas a través de XAML.

XAML

```
<?xml version="1.0" encoding="UTF-8" ?>
<Shell
  xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
  xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
  xmlns:views="clr-namespace:eShop.Views"
  x:Class="eShop.AppShell">

  <!-- Omitted for brevity -->

  <FlyoutItem >
    <ShellContent x:Name="login" ContentTemplate="{DataTemplate
views:LoginView}" Route="Login" />
  </FlyoutItem>

  <TabBar x:Name="main" Route="Main">
    <ShellContent Title="CATALOG" Route="Catalog" Icon="{StaticResource
CatalogIconImageSource}" ContentTemplate="{DataTemplate views:CatalogView}"
/>
    <ShellContent Title="PROFILE" Route="Profile" Icon="{StaticResource
ProfileIconImageSource}" ContentTemplate="{DataTemplate views:ProfileView}"
/>
  </TabBar>
</Shell>
```

En este ejemplo, los objetos de interfaz de usuario `ShellContent` y `TabBar` establecen su propiedad `Route`. Este es el método preferido para registrar rutas para objetos de interfaz de usuario controlados por `Shell`.

Si tenemos objetos que se agregarán a la pila de navegación más adelante, deberá agregarlos mediante el código subyacente. En el ejemplo siguiente se muestra el registro de rutas en el código subyacente.

C#

```
Routing.RegisterRoute("Filter", typeof(FiltersView));
Routing.RegisterRoute("Basket", typeof(BasketView));
```

En el código subyacente, llamaremos al método `Routing.RegisterRoute`, que tiene un nombre de ruta como primer parámetro y un tipo de vista como segundo parámetro. Cuando un modelo de vista usa la propiedad `NavigationService` para navegar, el objeto `Shell` de la aplicación buscará rutas registradas y las insertará en la pila de navegación.

Una vez se haya creado la vista y se acceda a ella, se ejecutan los métodos `ApplyQueryAttributes` y `InitializeAsync` del modelo de vista asociado a la vista. Para más información, consulte [Pasar parámetros durante la navegación](#).

Navegación cuando se inicia la aplicación

Cuando se inicia la aplicación, se establece un objeto `Shell` como vista raíz de la aplicación. Una vez establecido, `Shell` se usará para controlar el registro de rutas y se mantendrá en la raíz de nuestra aplicación. Una vez creado, podemos esperar a que `Shell` se asocie a la aplicación mediante el método `OnParentSet` para inicializar nuestra ruta de navegación. El siguiente ejemplo de código muestra este método:

C#

```
protected override async void OnParentSet()
{
    base.OnParentSet();

    if (Parent is not null)
    {
        await _navigationService.InitializeAsync();
    }
}
```

El método usa una instancia de `INavigationService` proporcionada por el constructor a partir de la inserción de dependencias e invoca su método `InitializeAsync`.

En el siguiente código de ejemplo se muestra la implementación del método `MauiNavigationService.InitializeAsync`:

C#

```
public Task InitializeAsync()
{
    return
        NavigateToAsync(string.IsNullOrEmpty(_settingsService.AuthAccessToken)
            ? "//Login"
            : "//Main/Catalog");
}
```

Se navega a la ruta `//Main/Catalog` si la aplicación tiene un token de acceso almacenado en caché, que se usa para la autenticación. De lo contrario, se navega a la ruta `//Login`.

Pasar parámetros durante la navegación

El método `NavigateToAsync`, especificado por la interfaz `INavigationService`, permite especificar datos de navegación como un argumento `IDictionary<string, object>` de

datos que se pasan al modelo de vista al que se navega, donde normalmente se usa para realizar la inicialización.

Por ejemplo, la clase `ProfileViewModel` contiene un `OrderDetailCommand` que se ejecuta cuando el usuario selecciona una orden en la página `ProfileView`. Después, esto sirve para ejecutar el método `OrderDetailAsync`, que se muestra en el siguiente código de ejemplo:

C#

```
private async Task OrderDetailAsync(Order order)
{
    if (order is null)
    {
        return;
    }

    await NavigationService.NavigateToAsync(
        "OrderDetail",
        new Dictionary<string, object>{ { "OrderNumber", order.OrderNumber }
    });
}
```

Este método invoca la navegación a la ruta `OrderDetail`, pasando la información del número de la orden a la orden seleccionada por el usuario. Cuando el marco de inserción de dependencias crea `OrderDetailView` para la ruta `OrderDetail` junto con la clase `OrderDetailViewModel` que se asigna al `BindingContext` de la vista.

`OrderDetailViewModel` tiene un atributo agregado que le permite recibir datos del servicio de navegación, como se muestra en el siguiente código de ejemplo.

C#

```
[QueryProperty(nameof(OrderNumber), "OrderNumber")]
public class OrderDetailViewModel : ViewModelBase
{
    public int OrderNumber { get; set; }
}
```

El atributo `QueryProperty` nos permite proporcionar un parámetro para que una propiedad asigne valores y una clave para buscar valores del diccionario de parámetros de consulta. En este ejemplo, se proporcionaron la clave "OrderNumber" y el valor del número de la orden durante la llamada de `NavigateToAsync`. El modelo de vista encontró la clave "OrderNumber" y asignó el valor a la propiedad `OrderNumber`. Así, la propiedad `OrderNumber` se puede usar más adelante para recuperar los detalles completos de la orden desde la instancia `OrderService`.

Invocación de la navegación mediante comportamientos

Normalmente, la interacción del usuario desencadena la navegación desde una vista. Por ejemplo, `LoginView` realiza la navegación después de autenticarse correctamente. En el siguiente código de ejemplo se muestra la forma en que un comportamiento invoca la navegación:

XML

```
<WebView>
  <WebView.Behaviors>
    <behaviors:EventToCommandBehavior
      EventName="Navigating"
      EventArgsConverter="
{StaticResource WebNavigatingEventArgsConverter}"
      Command="{Binding NavigateCommand}" />
  </WebView.Behaviors>
</WebView>
```

En tiempo de ejecución, `EventToCommandBehavior` responderá a la interacción con `WebView`. Cuando `WebView` navega a una página web, se activará el evento `Navigating`, que ejecutará `NavigateCommand` en `LoginViewMode`. De manera predeterminada, los argumentos de evento para el evento se pasarán al comando. Estos datos se convierten a medida que se pasan entre el origen y el destino mediante el convertidor especificado en la propiedad `EventArgsConverter`, que devuelve `Url` desde `WebNavigatingEventArgs`. Por lo tanto, cuando `NavigationCommand` se ejecuta, la propiedad `Url` de la página web se pasa como un parámetro a la Acción registrada.

Después, `NavigationCommand` ejecuta el método `NavigateAsync`, que se muestra en el siguiente código de ejemplo:

C#

```
private async Task NavigateAsync(string url)
{
    // Omitted for brevity.
    if (!string.IsNullOrEmpty(accessToken))
    {
        _settingsService.AuthAccessToken = accessToken;
        _settingsService.AuthIdToken = authResponse.IdentityToken;
        await NavigationService.NavigateToAsync("//Main/Catalog");
    }
}
```

Este método invoca la ruta `NavigationService` de la aplicación a la ruta `//Main/Catalog`.

Confirmación o cancelación de la navegación

Es posible que una aplicación tenga que interactuar con el usuario durante una operación de navegación para que el usuario pueda confirmar o cancelar la navegación. Esto puede ser necesario, por ejemplo, cuando el usuario intenta navegar antes de haber completado del todo una página de entrada de datos. En esta situación, una aplicación debe enviar una notificación que permita al usuario salir de la página o cancelar la operación de navegación antes de que se produzca. Esto se puede lograr en una clase de modelo de vista, usando la respuesta de una notificación para controlar si se invoca o no la navegación.

Resumen

.NET MAUI incluye compatibilidad con la navegación por páginas, que normalmente surge a partir de la interacción del usuario con la UI o de la propia aplicación, como resultado de los cambios de estado internos causados por la lógica. Sin embargo, la navegación puede ser compleja de implementar en aplicaciones que usan el patrón MVVM.

En este capítulo se ha descrito una clase `NavigationService`, que se usa para realizar la navegación de la página con prioridad para el modelo de vista desde modelos de vistas. Si se coloca la lógica de navegación en las clases de modelo de vista, la lógica se puede revisar mediante pruebas unitarias. Además, el modelo de vista puede implementar lógica para controlar la navegación y así asegurar que se aplican determinadas reglas del negocio.

Colaborar con nosotros en GitHub

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).



Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)

Validación

Artículo • 30/05/2024

💡 Sugerencia

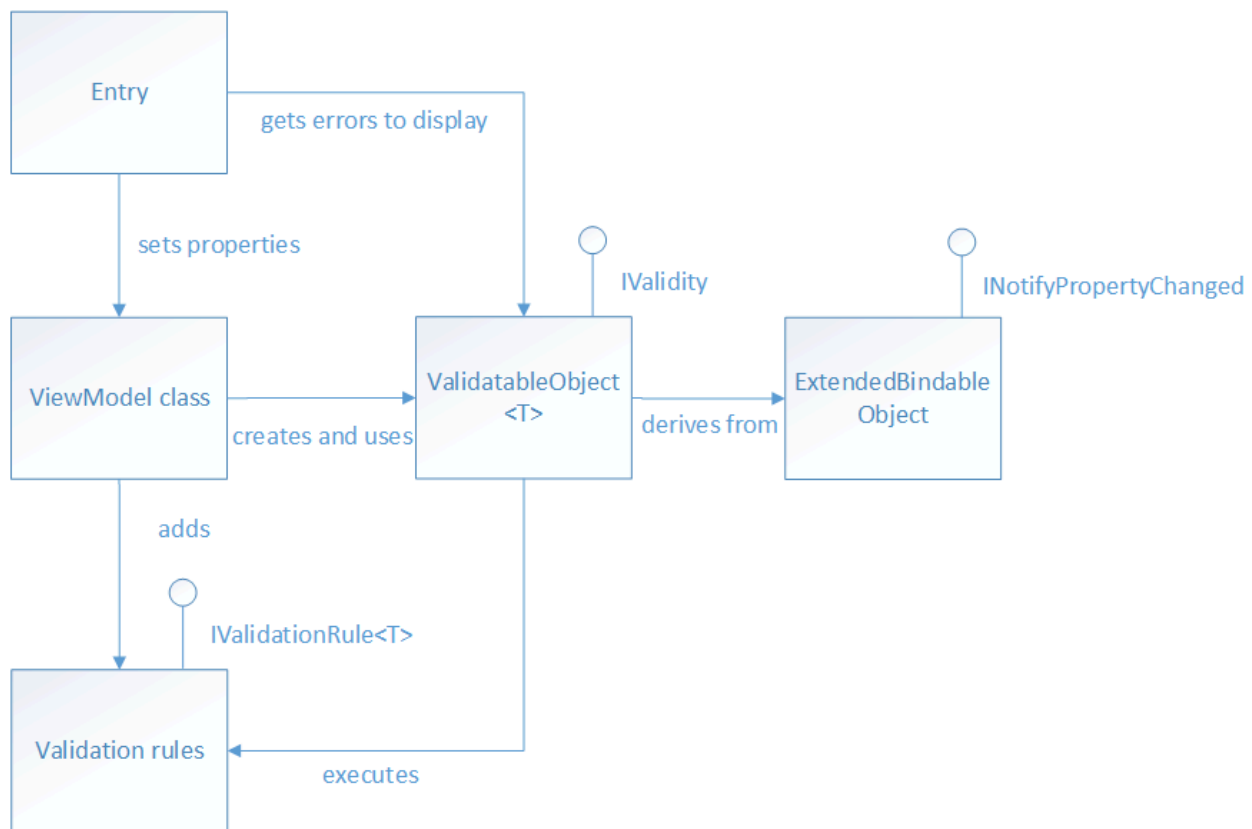
Este contenido es un extracto del libro electrónico "Patrones de aplicaciones empresariales con .NET MAUI", disponible en [Documentación de .NET](#) o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF



Cualquier aplicación que acepte la entrada de los usuarios debe asegurarse de que la entrada sea válida. Una aplicación podría, por ejemplo, comprobar si hay entradas que contengan solo caracteres en un intervalo determinado, es de una longitud determinada o coincide con un formato determinado. Sin validación, un usuario puede proporcionar datos que hacen que se produzca un error en la aplicación. La validación adecuada aplica reglas de negocio y podría ayudar a evitar que un atacante inserte datos malintencionados.

En el contexto del patrón Modelo-Vista-Modelo de vista (MVVM), a menudo se necesitará un modelo o un modelo de vista para realizar la validación de los datos e indicar cualquier error de validación en la vista para que el usuario pueda corregirlo. La aplicación multiplataforma eShop realiza la validación sincrónica del lado cliente de las propiedades del modelo de vista y notifica al usuario los errores de validación resaltando el control que contiene los datos no válidos y mostrando mensajes de error que informan al usuario de por qué los datos no son válidos. En la imagen siguiente se muestran las clases implicadas en la validación en la aplicación multiplataforma eShop.



Las propiedades del modelo de vista que requieren validación son de tipo `ValidatableObject<T>` y cada instancia de `ValidatableObject<T>` tiene reglas de validación agregadas a su propiedad `Validations`. La validación se invoca desde el modelo de vista llamando al método `Validate` de la instancia `ValidatableObject<T>`, que recupera las reglas de validación y las ejecuta en la propiedad `ValidatableObject<T>.Value`. Los errores de validación se colocan en la propiedad `Errors` de la instancia de `ValidatableObject<T>` y la propiedad `IsValid` de la instancia de `ValidatableObject<T>` se actualiza para indicar si la validación se realizó correctamente o no. En el siguiente código se muestra la implementación de `ValidatableObject<T>`.

C#

```

using CommunityToolkit.Mvvm.ComponentModel;
namespace eShop.Validations;
public class ValidatableObject<T> : ObservableObject, IValidity
{
    private IEnumerable<string> _errors;
    private bool _isValid;
    private T _value;
    public List<IValidationRule<T>> Validations { get; } = new();
    public IEnumerable<string> Errors
    {
        get => _errors;
        private set => SetProperty(ref _errors, value);
    }
    public bool IsValid

```

```

{
    get => _isValid;
    private set => SetProperty(ref _isValid, value);
}
public T Value
{
    get => _value;
    set => SetProperty(ref _value, value);
}
public ValidatableObject()
{
    _isValid = true;
    _errors = Enumerable.Empty<string>();
}
public bool Validate()
{
    Errors = Validations
        ?.Where(v => !v.Check(Value))
        ?.Select(v => v.ValidationMessage)
        ?.ToArray()
        ?? Enumerable.Empty<string>();
    IsValid = !Errors.Any();
    return IsValid;
}
}

```

La clase `ObservableObject` proporciona la notificación de los cambios de propiedad, por lo que un control `Entry` puede enlazar a la propiedad `IsValid` de la instancia de `ValidatableObject<T>` en la clase del modelo de vista para que se le notifique si los datos especificados son válidos o no.

Especificación de reglas de validación

Las reglas de validación se especifican mediante la creación de una clase que deriva de la interfaz `IValidationRule<T>`, que se muestra en el ejemplo de código siguiente:

C#

```

public interface IValidationRule<T>
{
    string ValidationMessage { get; set; }
    bool Check(T value);
}

```

Esta interfaz especifica que una clase de regla de validación debe proporcionar un método booleano `Check` que se usa para realizar la validación necesaria y una

propiedad `ValidationMessage` cuyo valor es el mensaje de error de validación que se mostrará si se produce un error en la validación.

En el ejemplo de código siguiente se muestra la regla de validación

`IsNotNullOrEmptyRule<T>`, que se usa para realizar la validación del nombre de usuario y la contraseña especificados por el usuario en `LoginView` cuando se usan servicios ficticios en la aplicación multiplataforma eShop:

C#

```
public class IsNotNullOrEmptyRule<T> : IValidationRule<T>
{
    public string ValidationMessage { get; set; }

    public bool Check(T value) =>
        value is string str && !string.IsNullOrEmpty(str);
}
```

El método `Check` devuelve un valor booleano que indica si el argumento de valor es nulo, está vacío o solo consta de caracteres de espacio en blanco.

Aunque la aplicación multiplataforma eShop no la usa, en el ejemplo de código siguiente se muestra una regla de validación para validar las direcciones de correo electrónico:

C#

```
public class EmailRule<T> : IValidationRule<T>
{
    private readonly Regex _regex = new(@"^([w.-]+)@([w-]+)((.(w){2,3})+)$");

    public string ValidationMessage { get; set; }

    public bool Check(T value) =>
        value is string str && _regex.IsMatch(str);
}
```

El método `Check` devuelve un valor booleano que indica si el argumento de valor es una dirección de correo electrónico válida. Esto se logra buscando en el argumento de valor la primera aparición del patrón de expresión regular especificado en el constructor `Regex`. Que el patrón de expresión regular se haya encontrado en la cadena de entrada se puede determinar comparando `value` con `Regex.IsMatch`.

❗ Nota

La validación de propiedades a veces puede implicar propiedades dependientes. Un ejemplo de propiedades dependientes es cuando el conjunto de valores válidos de la propiedad A depende del valor determinado que se ha establecido en la propiedad B. Para comprobar que el valor de la propiedad A es uno de los valores permitidos, habría que recuperar el valor de la propiedad B. Además, cuando cambia el valor de la propiedad B, la propiedad A tendría que volver a validarse.

Adición de reglas de validación a una propiedad

En la aplicación multiplataforma eShop, las propiedades del modelo de vista que requieren validación se declaran de tipo `ValidatableObject<T>`, donde `T` es el tipo de los datos que se van a validar. En el ejemplo de código siguiente se muestra un ejemplo de dos de estas propiedades:

C#

```
public ValidatableObject<string> UserName { get; private set; }  
public ValidatableObject<string> Password { get; private set; }
```

Para que se produzca la validación, se deben agregar reglas de validación a la colección `Validations` de cada instancia de `ValidatableObject<T>`, como se muestra en el ejemplo de código siguiente:

C#

```
private void AddValidations()  
{  
    UserName.Validations.Add(new IsNotNullOrEmptyRule<string>  
    {  
        ValidationMessage = "A username is required."  
    });  
  
    Password.Validations.Add(new IsNotNullOrEmptyRule<string>  
    {  
        ValidationMessage = "A password is required."  
    });  
}
```

Este método agrega la regla de validación `IsNotNullOrEmptyRule<T>` a la colección `Validations` de cada instancia de `ValidatableObject<T>`, y se especifican valores para la propiedad `ValidationMessage` de la regla de validación. Esta propiedad determina el mensaje de error de validación que se mostrará si se produce un error en la validación.

Desencadenamiento de la validación

El enfoque de validación usado en la aplicación multiplataforma eShop puede desencadenar la validación de una propiedad manualmente y, luego, automáticamente, cuando la propiedad cambia.

Desencadenamiento de la validación manualmente

La validación se puede desencadenar manualmente para una propiedad de modelo de vista. Por ejemplo, esto ocurre en la aplicación multiplataforma eShop cuando el usuario pulsa el botón `Login` en `LoginView`, cuando se usan servicios ficticios. El delegado de comandos llama al método `MockSignInAsync` en `LoginViewModel`, que invoca la validación mediante la ejecución del método `Validate`, que se muestra en el ejemplo de código siguiente:

C#

```
private bool Validate()
{
    bool isValidUser = ValidateUserName();
    bool isValidPassword = ValidatePassword();
    return isValidUser && isValidPassword;
}

private bool ValidateUserName()
{
    return _userName.Validate();
}

private bool ValidatePassword()
{
    return _password.Validate();
}
```

El método `Validate` realiza la validación del nombre de usuario y la contraseña especificados por el usuario en `LoginView`, invocando el método `Validate` en cada instancia de `ValidatableObject<T>`. En el ejemplo de código siguiente se muestra el método `Validate` de la clase `ValidatableObject<T>`:

C#

```
public bool Validate()
{
    Errors = _validations
        ?.Where(v => !v.Check(Value))
        ?.Select(v => v.ValidationMessage)
}
```

```

        ?.ToArray()
        ?? Enumerable.Empty<string>());

    IsValid = !Errors.Any();

    return IsValid;
}

```

Este método recupera las reglas de validación que se agregaron a la colección `Validations` del objeto. El método `Check` de cada regla de validación recuperada se ejecuta y el valor de propiedad `ValidationMessage` de cualquier regla de validación que no valide los datos se agrega a la colección `Errors` de la instancia de `ValidatableObject<T>`. Por último, se establece la propiedad `IsValid` y su valor se devuelve al método que realiza la llamada, que indica si la validación se realizó correctamente o no.

Desencadenamiento de la validación cuando cambian las propiedades

La validación también se desencadena automáticamente cada vez que cambia una propiedad enlazada. Por ejemplo, cuando un enlace bidireccional de `LoginView` establece la propiedad `UserName` o `Password`, se desencadena la validación. En el ejemplo de código siguiente se muestra cómo se produce esto:

XML

```

<Entry Text="{Binding UserName.Value, Mode=TwoWay}">
    <Entry.Behaviors>
        <behaviors:EventToCommandBehavior
            EventName="TextChanged"
            Command="{Binding ValidateUserNameCommand}" />
    </Entry.Behaviors>
</Entry>

```

El control `Entry` enlaza a la propiedad `UserName.Value` de la instancia de `ValidatableObject<T>` y la colección `Behaviors` del control tiene una instancia de `EventToCommandBehavior` agregada. Este comportamiento ejecuta `ValidateUserNameCommand` en respuesta al evento `TextChanged` que se desencadena en `Entry`, que se genera cuando cambia el texto de `Entry`. A su vez, el delegado `ValidateUserNameCommand` ejecuta el método `ValidateUserName`, que ejecuta el método `Validate` en la instancia de `ValidatableObject<T>`. Por lo tanto, cada vez que el usuario escribe un carácter en el control `Entry` para el nombre de usuario, se realiza la validación de los datos especificados.

Visualización de los errores de validación

Para notificar al usuario los errores de validación, la aplicación multiplataforma eShop resalta el control que contiene los datos no válidos con un fondo rojo y muestra debajo un mensaje de error que informa al usuario de por qué los datos no son válidos. Cuando se corrigen los datos no válidos, el fondo vuelve al estado predeterminado y se quita el mensaje de error. La imagen siguiente muestra `LoginView` en la aplicación multiplataforma eShop cuando hay errores de validación.

User name or email

A username is required.

Password

A password is required.

Resaltado de un control que contiene datos no válidos

.NET MAUI ofrece varias maneras de presentar información de validación a los usuarios finales, pero una de las formas más sencillas es mediante el uso de `Triggers`. Los elementos `Triggers` proporcionan una manera de cambiar el estado de nuestros controles, normalmente por su apariencia, en función de un evento o de un cambio de datos que se produce para un control. Para la validación, usaremos un elemento `DataTrigger` que escuchará los cambios generados desde una propiedad enlazada y responderá a ellos. Los controles `Entry` de `LoginView` se configuran mediante el código siguiente:

XML

```
<Entry Text="{Binding UserName.Value, Mode=TwoWay}">
  <Entry.Style>
    <OnPlatform x:TypeArguments="Style">
      <On Platform="iOS, Android" Value="{StaticResource EntryStyle}"
    />
    <On Platform="WinUI" Value="{StaticResource WinUIEntryStyle}" />
    </OnPlatform>
  </Entry.Style>
  <Entry.Behaviors>
    <mct:EventToCommandBehavior
      EventName="TextChanged"
      Command="{Binding ValidateCommand}" />
  </Entry.Behaviors>
</Entry>
```

```

</Entry.Behaviors>
<Entry.Triggers>
  <DataTrigger
    TargetType="Entry"
    Binding="{Binding UserName.IsValid}"
    Value="False">
    <Setter Property="BackgroundColor" Value="{StaticResource
ErrorColor}" />
  </DataTrigger>
</Entry.Triggers>
</Entry>

```

`DataTrigger` especifica las siguientes propiedades:

 Expandir tabla

Propiedad	Descripción
<code>TargetType</code>	Tipo de control al que pertenece el desencadenador.
<code>Binding</code>	Marcado de datos <code>Binding</code> que proporcionará notificaciones de los cambios y el valor de la condición del desencadenador.
<code>Value</code>	Valor de datos que se va a especificar cuando se ha cumplido la condición del desencadenador.

En `Entry`, se escucharán los cambios en la propiedad `LoginViewModel.UserName.IsValid`. Cada vez que esta propiedad genera un cambio, el valor se comparará con la propiedad `Value` establecida en `DataTrigger`. Si los valores son iguales, se cumplirá la condición del desencadenador y se ejecutarán los objetos `Setter` proporcionados a `DataTrigger`. Este control tiene un único objeto `Setter` que actualiza la propiedad `BackgroundColor` a un color personalizado definido mediante el marcado `StaticResource`. Cuando ya no se cumpla una condición `Trigger`, el control revertirá las propiedades establecidas por el objeto `Setter` a su estado anterior. Para más información sobre `Triggers`, consulte [Documentación de .NET MAUI: Desencadenadores](#).

Visualización de mensajes de error

La interfaz de usuario muestra los mensajes de error de validación en los controles `Label` debajo de cada control cuyos datos no se pudieron validar. En el ejemplo de código siguiente se muestra el elemento `Label` que muestra un mensaje de error de validación, si el usuario no ha escrito un nombre de usuario válido:

XML

```
<Label
    Text="{Binding UserName.Errors, Converter=
    {StaticResource FirstValidationErrorConverter}}"
    Style="{StaticResource ValidationErrorLabelStyle}" />
```

Cada etiqueta enlaza a la propiedad `Errors` del objeto de modelo de vista que se está validando. La propiedad `Errors` se proporciona mediante la clase `ValidatableObject<T>`, y es de tipo `IEnumerable<string>`. Dado que la propiedad `Errors` puede contener varios errores de validación, la instancia de `FirstValidationErrorConverter` se usa para recuperar el primer error de la colección y mostrarlo.

Resumen

La aplicación multiplataforma eShop realiza la validación sincrónica del lado cliente de las propiedades del modelo de vista y notifica al usuario los errores de validación resaltando el control que contiene los datos no válidos y mostrando mensajes de error que informan al usuario de por qué los datos no son válidos.

Las propiedades del modelo de vista que requieren validación son de tipo `ValidatableObject<T>` y cada instancia de `ValidatableObject<T>` tiene reglas de validación agregadas a su propiedad `Validations`. La validación se invoca desde el modelo de vista llamando al método `Validate` de la instancia de `ValidatableObject<T>`, que recupera las reglas de validación y las ejecuta en la propiedad de valor `ValidatableObject<T>`. Los errores de validación se colocan en la propiedad `Errors` de la instancia de `ValidatableObject<T>` y la propiedad `IsValid` de la instancia de `ValidatableObject<T>` se actualiza para indicar si la validación se realizó correctamente o no.

Colaborar con nosotros en GitHub

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).

 .NET

Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)

Administración de la configuración de la aplicación

Artículo • 08/06/2024

💡 Sugerencia

Este contenido es un extracto del libro electrónico "Patrones de aplicaciones empresariales con .NET MAUI", disponible en [Documentación de .NET](#) o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF



La configuración permite la separación de datos que configura el comportamiento de una aplicación desde el código, lo que permite cambiar el comportamiento sin volver a generar la aplicación. Hay dos tipos de configuración: la configuración de la aplicación y la configuración de usuario.

La configuración de la aplicación son datos que crea y administra una aplicación. Puede incluir datos como puntos de conexión fijos de servicio web, claves de API y estado en tiempo de ejecución. La configuración de la aplicación está vinculada a la funcionalidad principal y solo es significativa para esa aplicación.

La configuración del usuario es la configuración personalizable de una aplicación que afecta al comportamiento de la aplicación y no requiere un reajuste frecuente. Por ejemplo, una aplicación podría permitir al usuario especificar dónde recuperar datos y cómo mostrarlos en la pantalla.

Crear una interfaz de configuración

Aunque el administrador de preferencias se puede usar directamente en la aplicación, tiene el inconveniente de que hace que la aplicación esté estrechamente acoplada a la

implementación del administrador de preferencias. Este acoplamiento significa que la creación de pruebas unitarias o la ampliación de la funcionalidad de administración de preferencias será limitada, ya que la aplicación no tendrá una manera directa de interceptar el comportamiento. Para solucionar este problema, se puede crear una interfaz para que funcione como proxy para la administración de preferencias. La interfaz nos permitirá proporcionar una implementación que se adapte a nuestras necesidades. Por ejemplo, al escribir una prueba unitaria, es posible que deseemos establecer una configuración específica y la interfaz nos proporcionará una manera fácil de establecer de forma coherente estos datos para la prueba. En el ejemplo de código siguiente, se muestra la interfaz `ISettingsService` de la aplicación multiplataforma eShop:

C#

```
namespace eShop.Services.Settings;

public interface ISettingsService
{
    string AuthAccessToken { get; set; }
    string AuthIdToken { get; set; }
    bool UseMocks { get; set; }
    string IdentityEndpointBase { get; set; }
    string GatewayShoppingEndpointBase { get; set; }
    string GatewayMarketingEndpointBase { get; set; }
    bool UseFakeLocation { get; set; }
    string Latitude { get; set; }
    string Longitude { get; set; }
    bool AllowGpsLocation { get; set; }
}
```

Adición de valores

.NET MAUI incluye un administrador de preferencias que permite almacenar la configuración del entorno de ejecución de un usuario. Se puede acceder a esta característica desde cualquier lugar de la aplicación mediante la clase `Microsoft.Maui.Storage.Preferences`. El administrador de preferencias proporciona un enfoque coherente, con seguridad de tipos y multiplataforma para conservar y recuperar la configuración de las aplicaciones y los usuarios, al tiempo que utiliza la administración de configuración nativa de cada plataforma. Además, es sencillo usar el enlace de datos para acceder a los datos de configuración expuestos por la biblioteca. Para obtener más información, consulte [Preferencias](#) en el Centro para desarrolladores de Microsoft.

💡 Sugerencia

Las preferencias están pensadas para almacenar datos relativamente pequeños. Si necesita almacenar datos más grandes o más complejos, considere la posibilidad de usar una base de datos local o un sistema de archivos para almacenar los datos.

Nuestra aplicación usará la clase `Preferences` necesaria para implementar la interfaz `ISettingsService`. En el código siguiente, se muestra cómo el valor de `SettingsService` de la aplicación multiplataforma eShop implementa las propiedades `AuthTokenAccess` y `UseMocks`:

C#

```
public sealed class SettingsService : ISettingsService
{
    private const string AccessToken = "access_token";
    private const string AccessTokenDefault = string.Empty;

    private const string IdUseMocks = "use_mocks";
    private const bool UseMocksDefault = true;

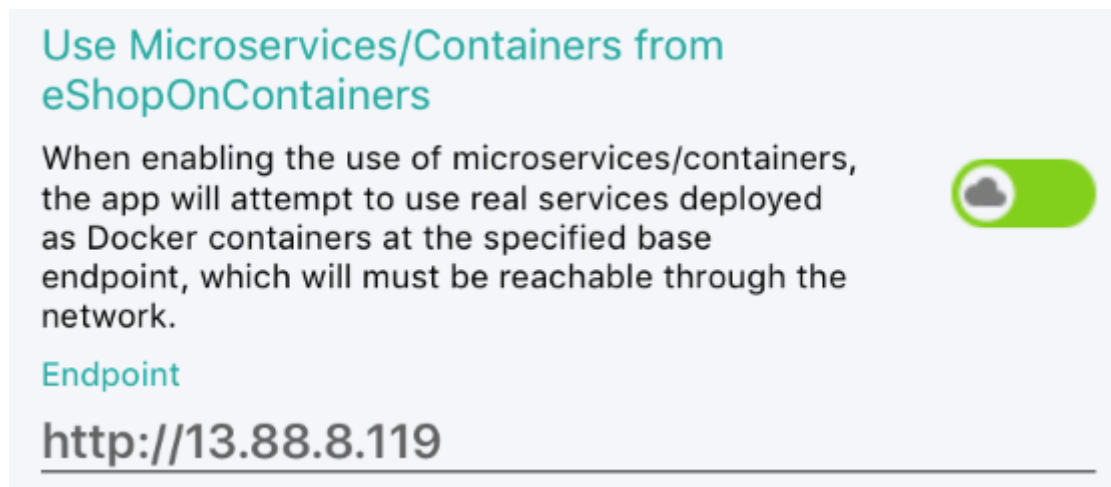
    public string AuthAccessToken
    {
        get => Preferences.Get(AccessToken, AccessTokenDefault);
        set => Preferences.Set(AccessToken, value);
    }

    public bool UseMocks
    {
        get => Preferences.Get(IdUseMocks, UseMocksDefault);
        set => Preferences.Set(IdUseMocks, value);
    }
}
```

Cada configuración consta de una clave privada, un valor predeterminado privado y una propiedad pública. La clave siempre es una cadena const que define un nombre único, siendo el valor predeterminado para la configuración un valor estático de solo lectura o constante del tipo requerido. Proporcionar un valor predeterminado garantiza que un valor válido esté disponible si se recupera un ajuste no establecido. Esta implementación del servicio se puede proporcionar a través de la inserción de dependencias en nuestra aplicación para su uso en modelos de vista u otros servicios en toda la aplicación.

Enlace de datos a la configuración del usuario

En la aplicación multiplataforma eShop, el elemento `SettingsView` expone varios valores que el usuario puede configurar en tiempo de ejecución. Estas opciones incluyen permitir la configuración de si la aplicación debe recuperar datos de microservicios implementados como contenedores Docker o si la aplicación debe recuperar datos de servicios ficticios que no requieren conexión a Internet. Al recuperar datos de microservicios en contenedores, se debe especificar una dirección URL de punto de conexión base para los microservicios. En la imagen siguiente se muestra `settingsView` cuando el usuario ha elegido recuperar datos de microservicios en contenedores.



El enlace de datos se puede usar para recuperar y establecer la configuración expuesta por la interfaz `ISettingService`. Esto se logra mediante controles en el enlace de vista para ver las propiedades del modelo que, a su vez, acceden a las propiedades de la interfaz `ISettingService` y generan una notificación de cambio de propiedad si el valor ha cambiado.

En el ejemplo de código siguiente se muestra el control `Entry` del valor `SettingsView` que permite al usuario escribir una dirección URL de punto de conexión de identidad base para los microservicios en contenedores:

XML

```
<Entry Text="{Binding IdentityEndpoint, Mode=TwoWay}" />
```

Este control `Entry` se enlaza a la propiedad `IdentityEndpoint` de la clase `SettingsViewModel` utilizando un enlace bidireccional. En el código de ejemplo siguiente se muestra la propiedad `IdentityEndpoint`:

C#

```
private readonly ISettingsService _settingsService;  
  
private string _identityEndpoint;
```

```

public SettingsViewModel(
    ILocationService locationService, IAppEnvironmentService
appEnvironmentService,
    IDialogService dialogService, INavigationService navigationService,
ISettingsService settingsService)
    : base(dialogService, navigationService, settingsService)
{
    _settingsService = settingsService;

    _identityEndpoint = _settingsService.IdentityEndpointBase;
}

public string IdentityEndpoint
{
    get => _identityEndpoint;
    set
    {
        SetProperty(ref _identityEndpoint, value);

        if (!string.IsNullOrEmpty(value))
        {
            UpdateIdentityEndpoint();
        }
    }
}

```

Cuando se establece la propiedad `IdentityEndpoint`, se llama al método `UpdateIdentityEndpoint`, siempre que el valor proporcionado sea válido. El siguiente ejemplo de código muestra el método `UpdateIdentityEndpoint`:

C#

```

private void UpdateIdentityEndpoint()
{
    _settingsService.IdentityEndpointBase = _identityEndpoint;
}

```

Este método actualiza la propiedad `IdentityEndpointBase` en la implementación de la interfaz `ISettingService` con el valor de dirección URL del punto de conexión base especificado por el usuario. Si la clase `SettingsService` se proporciona como implementación de `_settingsService`, el valor se conservará en el almacenamiento específico de la plataforma.

Resumen

La configuración permite la separación de datos que configura el comportamiento de una aplicación desde el código, lo que permite cambiar el comportamiento sin volver a generar la aplicación. La configuración de la aplicación son datos que una aplicación crea y administra, y la configuración del usuario es la configuración personalizable de una aplicación que afecta al comportamiento de la aplicación y no requiere un reajuste frecuente.

La clase `Microsoft.Maui.Storage.Preferences` proporciona un enfoque coherente, con seguridad de tipos y multiplataforma para conservar y recuperar la configuración de la aplicación y del usuario.

Microservicios en contenedores

Artículo • 30/05/2024

💡 Sugerencia

Este contenido es un extracto del libro electrónico "Patrones de aplicaciones empresariales con .NET MAUI", disponible en [Documentación de .NET](#) o como un PDF descargable y gratuito que se puede leer sin conexión.

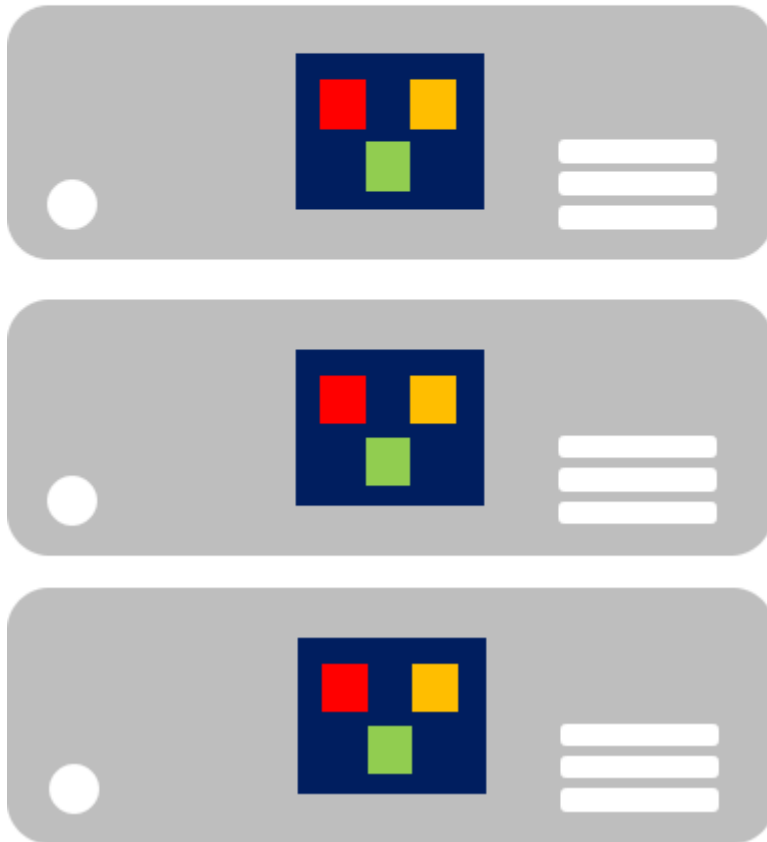
Descargar PDF



El desarrollo de aplicaciones cliente-servidor ha dado lugar a un enfoque en la creación de aplicaciones en capas que usan tecnologías específicas en cada capa. Estas aplicaciones se conocen a menudo como *monolíticas* y se empaquetan en hardware preescalado para cargas máximas. Los principales inconvenientes de este enfoque de desarrollo son el acoplamiento estricto entre los componentes de cada capa, que los componentes individuales no se pueden escalar fácilmente y el costo de las pruebas. Una actualización sencilla puede tener efectos imprevistos en el resto de la capa, por lo que un cambio en un componente de la aplicación requiere volver a probar e implementar toda la capa.

En la era de la nube, es especialmente preocupante que los componentes individuales no se puedan escalar fácilmente. Una aplicación monolítica contiene funcionalidad específica del dominio y suele estar dividida en capas funcionales, como front-end, lógica de negocios y almacenamiento de datos. En la imagen siguiente se muestra que una aplicación monolítica se escala clonando toda la aplicación en varias máquinas.

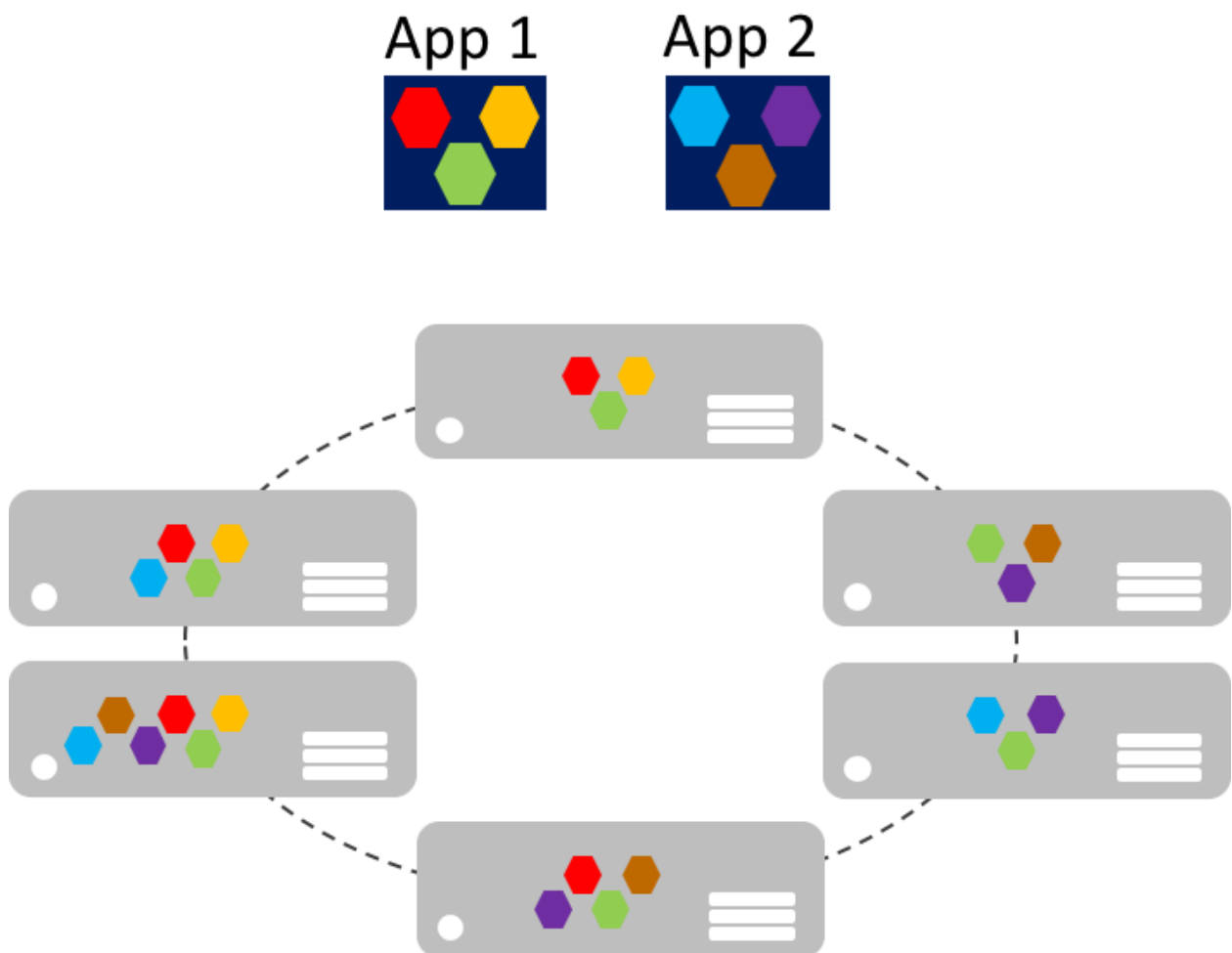
App



Microservicios

Los microservicios ofrecen un enfoque diferente para el desarrollo y la implementación de aplicaciones, un enfoque adecuado para los requisitos de agilidad, escala y confiabilidad de las aplicaciones en la nube modernas. Una aplicación de microservicios se divide en componentes independientes que funcionan conjuntamente para ofrecer la funcionalidad general de la aplicación. El término microservicio hace hincapié en que las aplicaciones deben estar compuestas de servicios lo suficientemente pequeños como para reflejar puntos concretos que son motivo de preocupación, por lo que cada microservicio implementa una sola función. Además, cada microservicio tiene contratos bien definidos con los que otros microservicios se comunican y comparten datos. Entre los ejemplos típicos de microservicios se incluyen los carros de la compra, el procesamiento de inventarios, los subsistemas de compra y el procesamiento de pagos.

Los microservicios se pueden escalar de forma independiente en comparación con las aplicaciones monolíticas gigantes que se escalan enteras. Esto significa que un área funcional específica que requiere más capacidad de procesamiento o ancho de banda de red para atender la demanda se puede escalar en lugar de escalar horizontalmente otras áreas de la aplicación innecesariamente. En la imagen siguiente se muestra este enfoque, donde los microservicios se implementan y escalan de forma independiente y crean instancias de servicios entre máquinas.



El escalado horizontal de microservicios puede ser casi instantáneo, lo que permite a una aplicación adaptarse a las cargas cambiantes. Por ejemplo, un único microservicio para la funcionalidad orientada a la web de una aplicación puede ser el único microservicio que sea necesario escalar horizontalmente para controlar el tráfico entrante adicional.

El modelo clásico para la escalabilidad de las aplicaciones es tener una capa sin estado y con equilibrio de carga con un almacén de datos externo compartido para almacenar datos persistentes. Los microservicios con estado administran sus propios datos persistentes, normalmente los almacenan localmente en los servidores en los que se colocan, para evitar la sobrecarga en el acceso a la red y la complejidad en las operaciones entre servicios. Esto permite el procesamiento más rápido posible de los datos y puede eliminar la necesidad de sistemas de almacenamiento en caché. Además,

los microservicios con estado escalables suelen particionar los datos entre sus instancias, con el fin de administrar el tamaño de los datos y transferir el procesamiento más allá de lo que un único servidor puede admitir.

Los microservicios también admiten actualizaciones independientes. Este acoplamiento ligero entre microservicios proporciona una evolución rápida y confiable de las aplicaciones. Su naturaleza distribuida e independiente facilita la implementación gradual de actualizaciones, donde solo un subconjunto de instancias de un único microservicio se actualiza en un momento dado. Por tanto, si se detecta un problema, se puede revertir una actualización errónea antes de que todas las instancias se actualicen con el código o la configuración defectuosos. De forma similar, los microservicios suelen usar el control de versiones de esquema, de modo que los clientes ven una versión coherente cuando se aplican las actualizaciones, independientemente de la instancia del microservicio con la que se comuniquen.

Por tanto, las aplicaciones de microservicios tienen muchas ventajas con comparación con las aplicaciones monolíticas:

- Cada microservicio es relativamente pequeño, fácil de administrar y desarrollar.
- Cada microservicio se puede implementar y actualizar independientemente de otros servicios.
- Cada microservicio se puede escalar horizontalmente de forma independiente. Por ejemplo, es posible que un servicio de catálogo o un servicio de cesta de la compra deba escalarse horizontalmente más que un servicio de pedidos. Por tanto, la infraestructura resultante consumirá los recursos de una forma más eficaz al escalar horizontalmente la capacidad.
- Cada microservicio aísla cualquier problema. Por ejemplo, si hay un problema en un servicio, solo afecta a ese servicio. Los demás servicios pueden seguir atendiendo solicitudes.
- Cada microservicio puede usar las tecnologías más recientes. Dado que los microservicios son autónomos y se ejecutan en paralelo, se pueden usar las tecnologías y los marcos más recientes, en lugar de estar obligado a usar un marco anterior que podría usar una aplicación monolítica.

No obstante, una solución basada en microservicios también tiene posibles inconvenientes:

- Elegir cómo particionar una aplicación en microservicios puede resultar complicado, ya que cada microservicio debe ser completamente autónomo y completo, incluida la responsabilidad de sus orígenes de datos.
- Los desarrolladores deben implementar comunicación entre los servicios, lo que agrega complejidad y latencia a la aplicación.

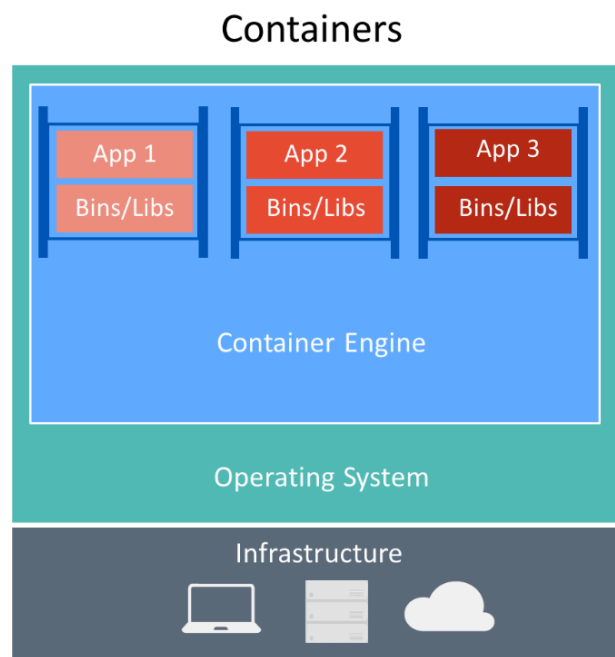
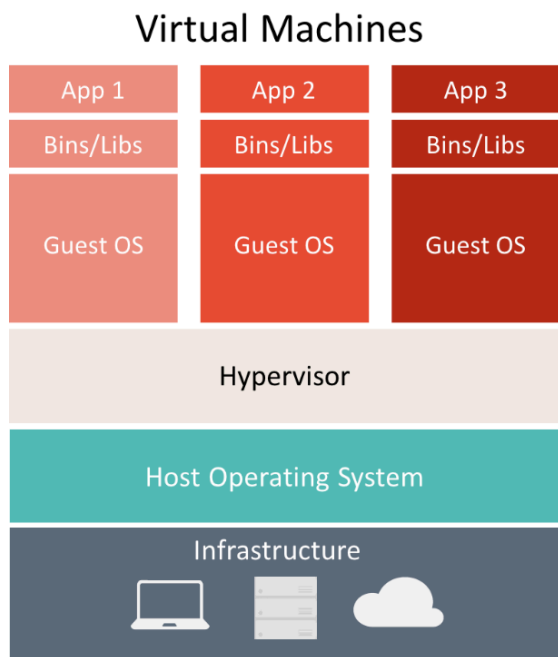
- Normalmente, no se pueden realizar transacciones atómicas entre varios microservicios. Por tanto, los requisitos empresariales deben adoptar la coherencia final entre los microservicios.
- En producción, la implementación y administración de un sistema compuesto por muchos servicios independientes supone una complejidad operativa.
- La comunicación directa de cliente a microservicio puede dificultar la refactorización de los contratos de los microservicios. Por ejemplo, con el tiempo es posible que deba cambiarse la forma en la que la aplicación está particionada en servicios. Un único servicio puede dividirse en dos o más servicios y dos servicios pueden combinarse. Cuando los clientes se comunican directamente con microservicios, este trabajo de refactorización puede interrumpir la compatibilidad con las aplicaciones cliente.

Inclusión en contenedores

La contenedorización es un enfoque para el desarrollo de software en el que una aplicación y su conjunto de dependencias con control de versiones, además de su configuración de entorno extraída en archivos de manifiesto de implementación, se empaquetan juntos como una imagen de contenedor, se prueban como una unidad y se implementan en un sistema operativo host.

Un contenedor es un entorno operativo aislado, controlado por recursos y portable, donde una aplicación se puede ejecutar sin tocar los recursos de otros contenedores ni el host. Un contenedor es y actúa como una máquina virtual o un equipo físico recién instalado.

Hay muchas similitudes entre los contenedores y las máquinas virtuales, como se muestra a continuación.



Un contenedor ejecuta un sistema operativo, tiene un sistema de archivos y se puede acceder a él a través de una red, al igual que una máquina física o virtual. Sin embargo, la tecnología y los conceptos que usan los contenedores son muy diferentes de los de las máquinas virtuales. Las máquinas virtuales incluyen las aplicaciones, las dependencias necesarias y un sistema operativo invitado completo. Los contenedores incluyen la aplicación y sus dependencias, pero comparten el sistema operativo con otros contenedores, que se ejecutan como procesos aislados en el sistema operativo host (excepto los contenedores Hyper-V, que se ejecutan dentro de una máquina virtual especial por contenedor). Por tanto, los contenedores comparten recursos y normalmente requieren menos recursos que las máquinas virtuales.

La ventaja de un enfoque de desarrollo e implementación orientado a contenedores es que elimina la mayoría de los problemas que surgen por configuraciones de entorno incoherentes. Además, los contenedores permiten una funcionalidad rápida de escalado vertical de las aplicaciones mediante la creación de nuevos contenedores según sea necesario.

Los conceptos clave para crear y trabajar con contenedores son:

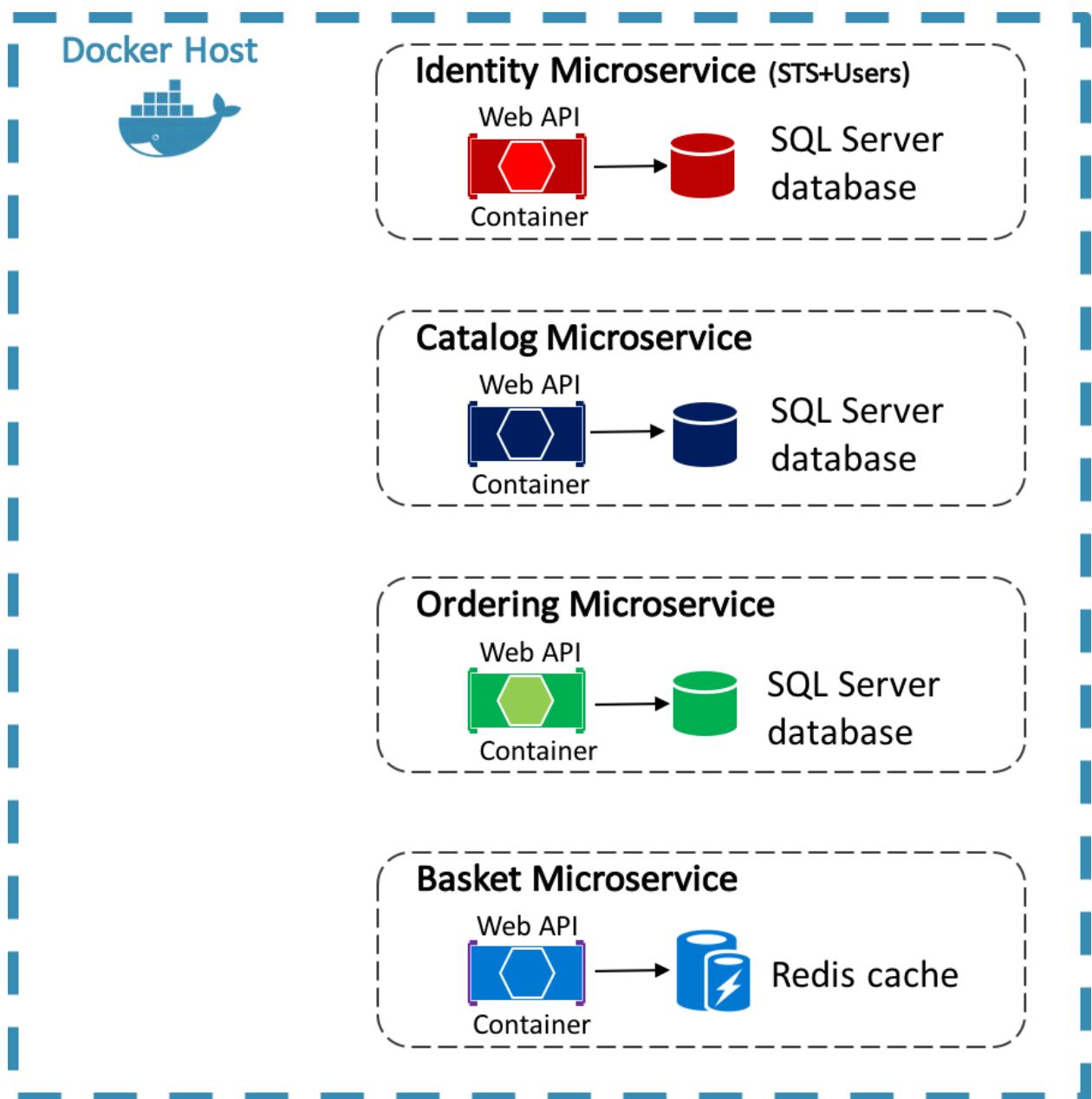
[Expandir tabla](#)

Concepto	Descripción
Host de contenedor	Máquina física o virtual configurada para hospedar contenedores. El host ejecutará uno o varios contenedores.
Imagen de contenedor	Una imagen consta de una unión de sistemas de archivos por capas apilados y es la base de un contenedor. Una imagen no tiene estado y nunca cambia a medida que se implementa en entornos diferentes.

Concepto	Descripción
Contenedor	Un contenedor es una instancia de una imagen en tiempo de ejecución.
Imagen de sistema operativo del contenedor	Los contenedores se implementan a partir de imágenes. La imagen del sistema operativo de contenedor es la primera de (quizá) muchas capas de imagen que componen un contenedor. Un sistema operativo de contenedor es inmutable y no se puede modificar.
Repositorio de contenedores	Cada vez que se crea una imagen de contenedor, esta y sus dependencias se almacenan en un repositorio local. Estas imágenes se pueden reutilizar muchas veces en el host de contenedor. Las imágenes de contenedor también pueden almacenarse en un registro público o privado, como Docker Hub , de forma que se puedan usar en varios hosts de contenedor diferentes.

Las empresas adoptan cada vez más contenedores cuando implementan aplicaciones basadas en microservicios y Docker se ha convertido en la implementación de contenedor estándar que la mayoría de las plataformas de software y proveedores de nube han adoptado.

La aplicación de referencia eShop usa Docker para hospedar cuatro microservicios de back-end contenedorizados, como se muestra en el diagrama siguiente.



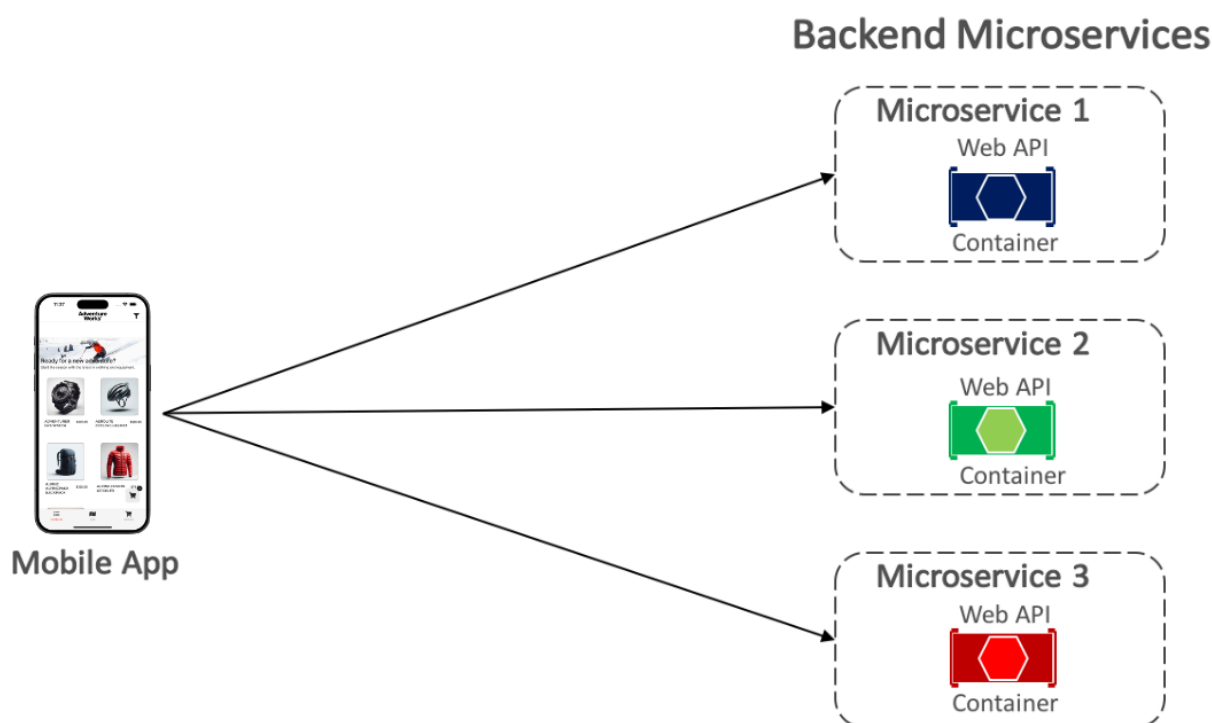
La arquitectura de los servicios de back-end de la aplicación de referencia se descompone en varios subsistemas autónomos en forma de microservicios y contenedores que colaboran. Cada microservicio proporciona una única área de funcionalidad: un servicio de identidad, un servicio de catálogo, un servicio de pedidos y un servicio de cesta.

Cada microservicio tiene su propia base de datos, lo que permite separarlo totalmente de otros microservicios. Cuando es necesario, la coherencia entre las bases de datos de diferentes microservicios se logra mediante eventos de nivel de aplicación. Para obtener más información, consulte [Comunicación entre microservicios](#).

Para obtener más información sobre la arquitectura de microservicios, consulte [Microservicios de .NET: arquitectura de aplicaciones de .NET contenedorizadas](#).

Comunicación entre cliente y microservicios

La aplicación multiplataforma eShop se comunica con los microservicios de back-end contenedorizados por medio de la comunicación *directa de cliente a microservicio*, como se muestra a continuación.



Con la comunicación directa de cliente a microservicio, la aplicación multiplataforma realiza solicitudes a cada microservicio directamente a través de su punto de conexión público, con un puerto TCP diferente por microservicio. En producción, el punto de conexión se suele asignar al equilibrador de carga del microservicio, que distribuye las solicitudes entre las instancias disponibles.

💡 Sugerencia

Considere la posibilidad de usar la comunicación de puerta de enlace de API.

La comunicación directa de cliente a microservicio puede tener inconvenientes cuando se crea una aplicación basada en microservicios grande y compleja, pero está más que indicada para una aplicación pequeña. Considere la posibilidad de usar la comunicación de puerta de enlace de API cuando diseñe una aplicación basada en microservicios de gran tamaño con decenas de microservicios. Para obtener más información, consulte [Microservicios de .NET: arquitectura de aplicaciones de .NET contenedorizadas](#).

Comunicación entre microservicios

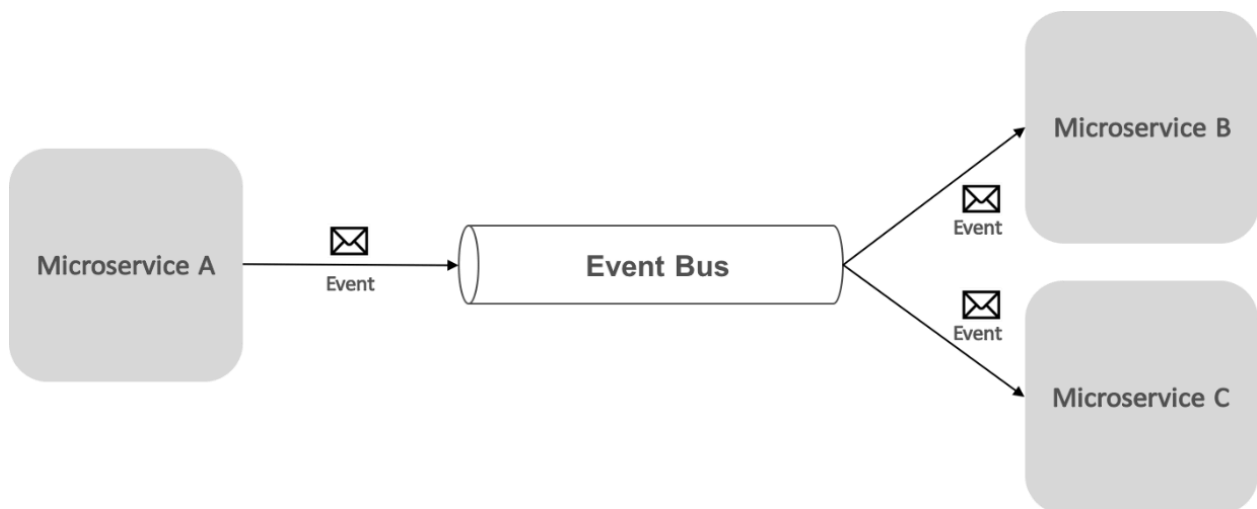
Una aplicación basada en microservicios es un sistema distribuido, que se puede ejecutar en varias máquinas. Lo habitual es que cada instancia de servicio sea un

proceso. Por tanto, los servicios deben interactuar usando un protocolo de comunicación entre procesos, como HTTP, TCP y Advanced Message Queuing Protocol (AMQP), o protocolos binarios, en función de la naturaleza de cada servicio.

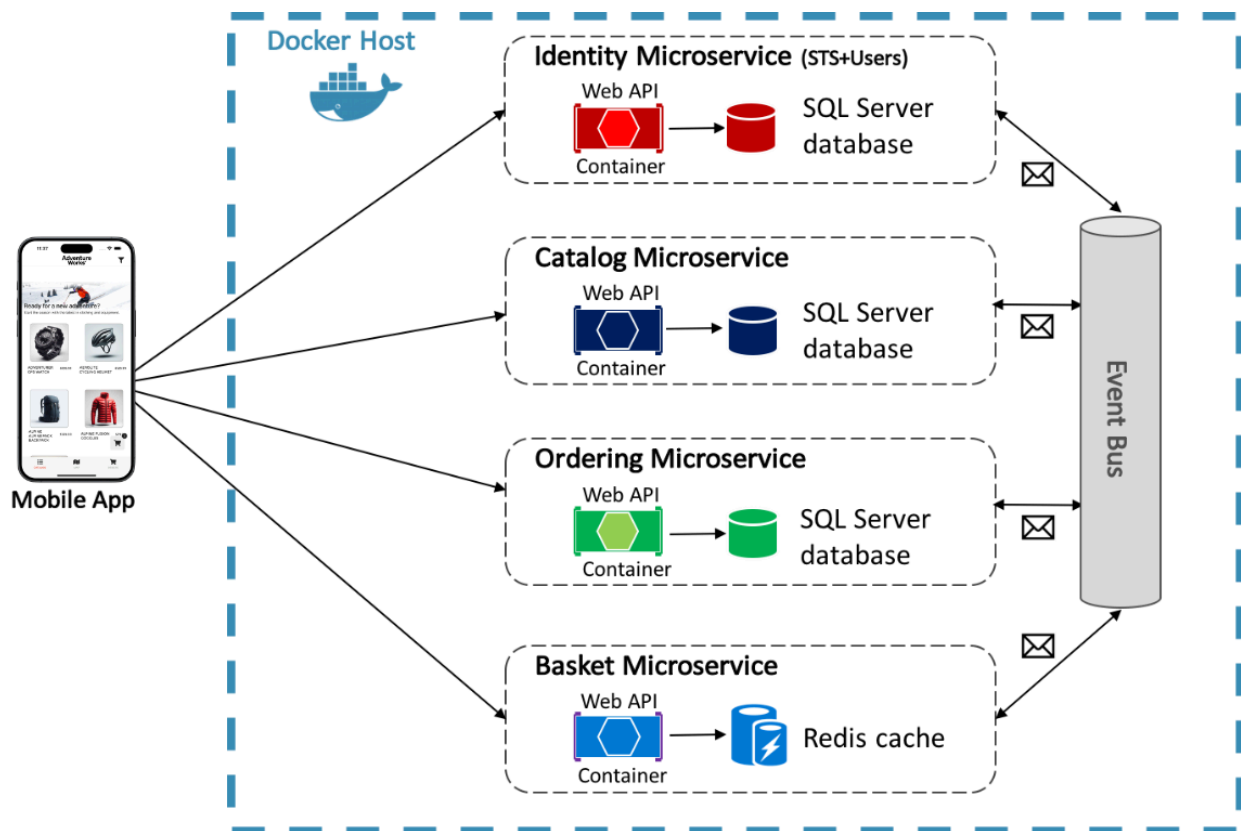
Los dos enfoques comunes para la comunicación entre microservicios son la comunicación REST basada en HTTP para consultar datos y la mensajería asíncrona ligera para comunicar actualizaciones en varios microservicios.

La comunicación controlada por eventos basada en la mensajería asíncrona es fundamental para propagar los cambios en varios microservicios. Con este enfoque, un microservicio publica un evento cuando sucede algo importante, por ejemplo, cuando actualiza una entidad empresarial. Otros microservicios se suscriben a estos eventos. Así, cuando un microservicio recibe un evento, actualiza sus propias entidades empresariales, lo que a su vez puede suponer que se publiquen más eventos. Esta funcionalidad de publicación-suscripción se suele lograr con un bus de eventos.

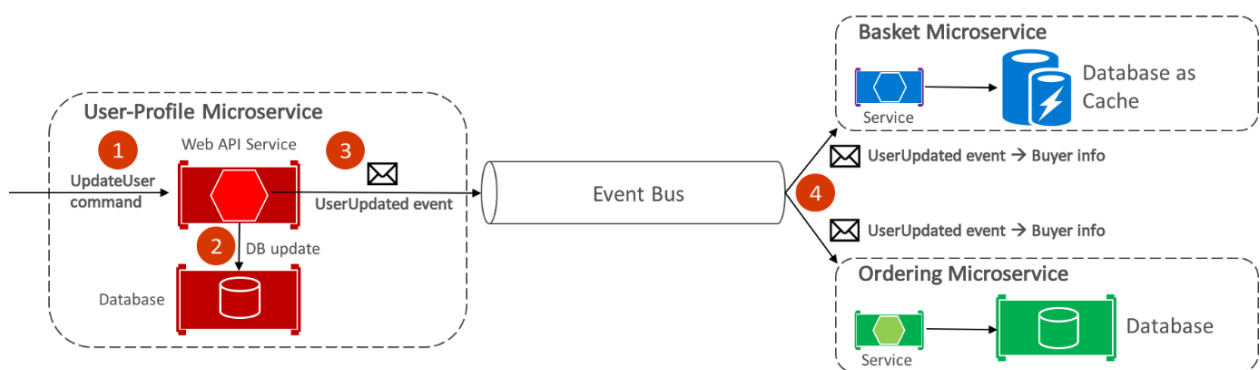
Un bus de eventos permite la comunicación de tipo suscripción-publicación entre microservicios, sin necesidad de que los componentes se reconozcan entre sí, como se muestra a continuación.



Desde la perspectiva de una aplicación, el bus de eventos es simplemente un canal de publicación-suscripción expuesto a través de una interfaz. Sin embargo, la forma en la que se implementa el bus de eventos puede variar. Por ejemplo, la implementación de un bus de eventos podría usar RabbitMQ, Azure Service Bus u otros buses de servicio, como NServiceBus y MassTransit. En el diagrama siguiente se muestra cómo se usa un bus de eventos en la aplicación de referencia eShop.



El bus de eventos de eShop, implementado con RabbitMQ, proporciona funcionalidad de publicación-suscripción de uno a varios. Esto significa que, después de publicar un evento, puede haber varios suscriptores escuchando el mismo evento. En el diagrama siguiente se muestra esta relación.



Este enfoque de comunicación de uno a varios usa eventos para implementar transacciones empresariales que abarcan varios servicios, lo que garantiza la coherencia final entre los servicios. Una transacción coherente consta de una serie de pasos distribuidos. Por tanto, cuando el microservicio de perfil de usuario recibe el comando **UpdateUser**, actualiza los detalles del usuario en su base de datos y publica el evento **UserUpdated** en el bus de eventos. Tanto el microservicio de cesta como el microservicio de pedidos se han suscrito para recibir este evento y, en respuesta, actualizan la información del comprador en sus respectivas bases de datos.

El bus de eventos de eShop, implementado con RabbitMQ, está pensado para usarse solo como prueba de concepto. Para los sistemas de producción, se deben tener en cuenta implementaciones alternativas de bus de eventos.

Para obtener más información sobre la implementación del bus de eventos, consulte [Microservicios de .NET: arquitectura de aplicaciones de .NET contenedorizadas](#).

Resumen

Los microservicios ofrecen un enfoque para el desarrollo y la implementación de aplicaciones que es adecuado para los requisitos de agilidad, escala y confiabilidad de las aplicaciones en la nube modernas. Una de las principales ventajas de los microservicios es que se pueden escalar horizontalmente de forma independiente, lo que significa que se puede escalar un área funcional específica que requiere más capacidad de procesamiento o ancho de banda de red para atender la demanda sin escalar innecesariamente áreas de la aplicación que no experimentan una mayor demanda.

Un contenedor es un entorno operativo aislado, controlado por recursos y portátil en el que una aplicación se puede ejecutar sin tocar los recursos de otros contenedores ni el host. Las empresas adoptan cada vez más contenedores cuando implementan aplicaciones basadas en microservicios y Docker se ha convertido en la implementación de contenedor estándar que la mayoría de las plataformas de software y proveedores de nube han adoptado.

Colaborar con nosotros en GitHub

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).



Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)

Acceso a datos remotos

Artículo • 30/05/2024

💡 Sugerencia

Este contenido es un extracto del libro electrónico "Patrones de aplicaciones empresariales con .NET MAUI", disponible en [Documentación de .NET](#) o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF



Muchas soluciones modernas basadas en web hacen uso de servicios web, hospedados en servidores web, para proporcionar funcionalidad a las aplicaciones cliente remotas. Las operaciones que expone un servicio web constituyen una API web.

Las aplicaciones cliente deben tener la capacidad de usar la API web sin saber cómo se implementan los datos o las operaciones que expone la API. Para ello, se requiere que la API se rija por las normas comunes que permiten que una aplicación cliente y un servicio web acuerden qué formatos de datos se usan y la estructura de los datos que se intercambian entre las aplicaciones cliente y el servicio web.

Introducción a la transferencia de estado representacional

La Transferencia de estado representacional (REST) es un estilo de arquitectura para la creación de sistemas distribuidos basados en hipermedia. Una de las principales ventajas del modelo de REST es que se basa en estándares abiertos y no vincula la implementación del modelo o las aplicaciones cliente que tienen acceso a él con ninguna implementación específica. Por tanto, un servicio web REST podría implementarse mediante [Microsoft ASP.NET Core](#), y las aplicaciones cliente podrían

desarrollarse usando cualquier lenguaje y conjunto de herramientas que pueda generar solicitudes HTTP y analizar las respuestas HTTP.

El modelo REST usa un esquema de navegación para representar objetos y servicios en una red, denominados recursos. Los sistemas que implementan REST normalmente usan el protocolo HTTP para transmitir solicitudes de acceso a estos recursos. En estos sistemas, una aplicación cliente envía una solicitud en forma de un URI que identifica un recurso y un método HTTP (como GET, POST, PUT o DELETE) que indica la operación que se realizará en ese recurso. El cuerpo de la solicitud HTTP contiene todos datos necesarios para realizar la operación.

ⓘ Nota

REST define un modelo de solicitud sin estado. Por lo tanto, las solicitudes HTTP deben ser independientes y pueden producirse en cualquier orden.

La respuesta de una solicitud REST hace uso de códigos de estado HTTP estándar. Por ejemplo, una solicitud que devuelve datos válidos debe incluir el código de respuesta HTTP 200 (`OK`), mientras que una solicitud que no encuentra o elimina un recurso especificado debe devolver una respuesta que incluya el código de estado HTTP 404 (`Not Found`).

Una API web RESTful expone un conjunto de recursos conectados y proporciona las operaciones básicas que permiten a una aplicación manipular esos recursos y navegar fácilmente entre ellos. Por este motivo, los URI que constituyen una API web RESTful típica están orientados a los datos que dicha API expone y usan las funciones proporcionadas por HTTP para operar con estos datos.

Los datos incluidos por una aplicación cliente en una solicitud HTTP, y los mensajes de respuesta correspondientes del servidor web, podrían presentarse en una variedad de formatos, conocidos como tipos de medios. Cuando una aplicación cliente envía una solicitud que devuelve datos en el cuerpo de un mensaje, puede especificar los tipos de medios que puede controlar en el encabezado `Accept` de la solicitud. Si el servidor web admite este tipo de medio, puede responder con una respuesta que incluye el encabezado `Content-Type` que especifica el formato de los datos en el cuerpo del mensaje. Es responsabilidad de la aplicación cliente analizar el mensaje de respuesta e interpretar correctamente los resultados en el cuerpo del mensaje.

Para más información sobre REST, consulte [Diseño de API](#) e [Implementación de API](#) en Microsoft Docs.

Consumo de API de RESTful

La aplicación multiplataforma eShop usa el patrón Modelo-Vista-Modelo de vista (MVVM), y los elementos del modelo del patrón representan las entidades de dominio usadas en la aplicación. Las clases de controlador y repositorio de la aplicación de referencia eShop aceptan y devuelven muchos de estos objetos de modelo. Por lo tanto, se usan como objetos de transferencia de datos (DTO) que contienen todos los datos que se pasan entre la aplicación y los microservicios contenedorizados. La principal ventaja de usar DTO para pasar y recibir datos a y desde un servicio web es que, al transmitir más datos en una sola llamada remota, la aplicación puede reducir el número de llamadas remotas que se deben realizar.

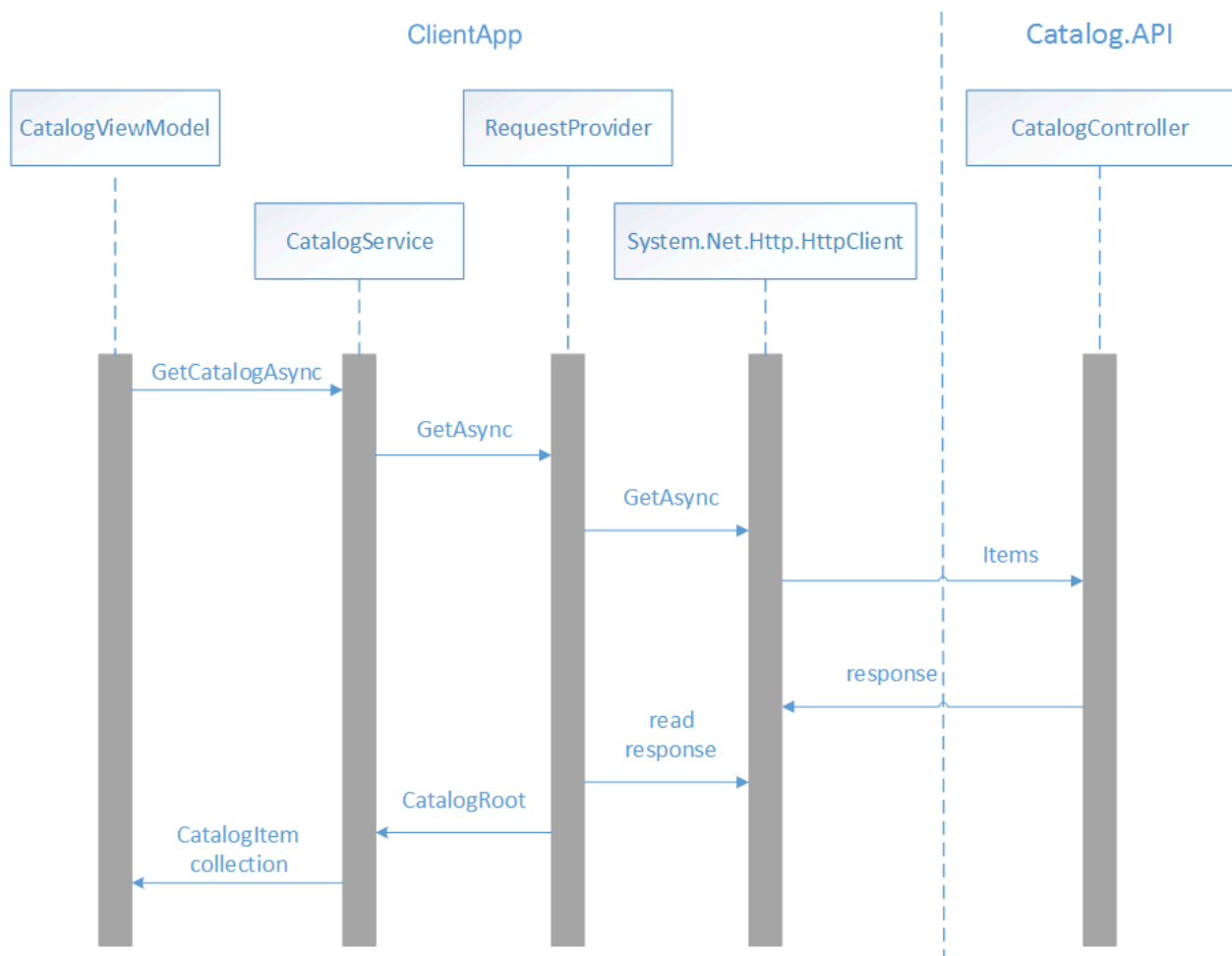
Realización de las solicitudes web

La aplicación multiplataforma eShop usa la clase `HttpClient` para realizar solicitudes a través de HTTP, con la utilización de JSON como tipo de medio. Esta clase proporciona funcionalidad para enviar solicitudes HTTP de forma asíncrona y recibir respuestas HTTP de un recurso identificado por URI. La clase `HttpResponseMessage` representa un mensaje de respuesta HTTP recibido de una API de REST una vez realizada una solicitud HTTP. Contiene información sobre la respuesta, incluido el código de estado, los encabezados y cualquier cuerpo. La clase `HttpContent` representa el cuerpo HTTP y los encabezados de contenido, como `Content-Type` y `Content-Encoding`. El contenido se puede leer mediante cualquiera de los métodos `ReadAs`, como `ReadAsStringAsync` y `ReadAsByteArrayAsync`, según el formato de los datos.

Realización de una solicitud GET

La clase `CatalogService` se usa para administrar el proceso de recuperación de datos desde el microservicio de catálogo. En el método `RegisterViewModels` de la clase `MauiProgram`, la clase `CatalogService` se registra como una asignación de tipos con el tipo `ICatalogService` con el contenedor de inserción de dependencias. A continuación, cuando se crea una instancia de la clase `CatalogViewModel1`, su constructor acepta un objeto `ICatalogService type`, que el contenedor de inserción de dependencias resuelve, y devuelve una instancia de la clase `CatalogService`. Para más información sobre la inserción de dependencias, vea [Inserción de dependencias](#).

En la imagen siguiente se muestra la interacción de las clases que leen datos de catálogo del microservicio de catálogo para su visualización mediante `CatalogView`.



Cuando se navega hasta `CatalogView`, se llama al método `OnInitialize` de la clase `CatalogViewModel`. Este método recupera datos de catálogo del microservicio de catálogo, como se muestra en el ejemplo de código siguiente:

C#

```

public override async Task InitializeAsync()
{
    Products = await _productsService.GetCatalogAsync();
}

```

Este método llama al método `GetCatalogAsync` de la instancia `CatalogService` insertada en `CatalogViewModel` por el contenedor de inserción de dependencias. El siguiente ejemplo de código muestra el método `GetCatalogAsync`:

C#

```

public async Task<ObservableCollection<CatalogItem>> GetCatalogAsync()
{
    UriBuilder builder = new
    UriBuilder(GlobalSetting.Instance.CatalogEndpoint);
    builder.Path = "api/v1/catalog/items";
    string uri = builder.ToString();
}

```

```

        CatalogRoot? catalog = await _requestProvider.GetAsync<CatalogRoot>
(uri);

        return catalog?.Data;
    }

```

Este método compila el URI que identifica el recurso al que se enviará la solicitud y usa la clase `RequestProvider` para invocar el método HTTP GET en el recurso, antes de devolver los resultados a `CatalogViewModel`. La clase `RequestProvider` contiene una funcionalidad que envía una solicitud en forma de URI que identifica un recurso, un método HTTP que indica la operación que se va a realizar en ese recurso y un cuerpo que contiene los datos necesarios para realizar la operación. Para obtener información sobre cómo se inserta la clase `RequestProvider` en la clase `CatalogService`, vea [Inserción de dependencias](#).

En el ejemplo de código siguiente se muestra el método `GetAsync` de la clase `RequestProvider`:

C#

```

public async Task<TResult> GetAsync<TResult>(string uri, string token = "")
{
    HttpClient httpClient = GetOrCreateHttpClient(token);
    HttpResponseMessage response = await httpClient.GetAsync(uri);

    await HandleResponse(response);
    TResult result = await response.Content.ReadFromJsonAsync<TResult>();

    return result;
}

```

Este método llama al método `GetOrCreateHttpClient`, que devuelve una instancia de la clase `HttpClient` con los encabezados adecuados establecidos. A continuación, envía una solicitud asincrónica `GET` al recurso identificado por el URI, con la respuesta almacenada en la instancia `HttpResponseMessage`. A continuación, se invoca el método `HandleResponse`, que produce una excepción si la respuesta no incluye un código de estado HTTP correcto. A continuación, la respuesta se lee como una cadena, se convierte de JSON en un objeto `CatalogRoot` y se devuelve a `CatalogService`.

El método `GetOrCreateHttpClient` se muestra en el ejemplo de código siguiente:

C#

```

private readonly Lazy<HttpClient> _httpClient =
    new Lazy<HttpClient>()

```

```

        () =>
        {
            var httpClient = new HttpClient();
            httpClient.DefaultRequestHeaders.Accept.Add(new
MediaTyewithQualityHeaderValue("application/json"));
            return httpClient;
        },
        LazyThreadSafetyMode.ExecutionAndPublication);

private HttpClient GetOrCreateHttpClient(string token = "")
{
    var httpClient = _httpClient.Value;

    if (!string.IsNullOrEmpty(token))
    {
        httpClient.DefaultRequestHeaders.Authorization = new
AuthenticationHeaderValue("Bearer", token);
    }
    else
    {
        httpClient.DefaultRequestHeaders.Authorization = null;
    }

    return httpClient;
}

```

Este método crea una instancia o recupera una instancia almacenada en caché de la clase `HttpClient` y establece el encabezado `Accept` de las solicitudes realizadas por la instancia `HttpClient` a `application/json`, lo que indica que espera que el contenido de cualquier respuesta tenga el formato JSON. A continuación, si se pasó un token de acceso como un argumento al método `GetOrCreateHttpClient`, se agrega al encabezado `Authorization` de las solicitudes realizadas por la instancia `HttpClient`, cuyo prefijo es la cadena `Bearer`. Para obtener más información acerca de la autorización, vea [Autorización](#).

Sugerencia

Es muy recomendable almacenar las instancias de `HttpClient` en la caché y reutilizarlas para mejorar el rendimiento de la aplicación. La creación de un elemento `HttpClient` para cada operación puede provocar un problema con el agotamiento de sockets. Para obtener más información, consulte [Instanciación de HttpClient](#) en el Centro para desarrolladores de Microsoft.

Cuando el método `GetAsync` de la clase `RequestProvider` llama a `HttpClient.GetAsync`, se invoca el método `Items` de la clase `CatalogController` del proyecto `Catalog.API`, que se muestra en el ejemplo de código siguiente:

C#

```
[HttpGet]
[Route("[action]")]
public async Task<IActionResult> Items(
    [FromQuery]int pageSize = 10, [FromQuery]int pageIndex = 0)
{
    var totalItems = await _catalogContext.CatalogItems
        .LongCountAsync();

    var itemsOnPage = await _catalogContext.CatalogItems
        .OrderBy(c => c.Name)
        .Skip(pageSize * pageIndex)
        .Take(pageSize)
        .ToListAsync();

    itemsOnPage = ComposePicUri(itemsOnPage);
    var model = new PaginatedItemsViewModel<CatalogItem>(
        pageIndex, pageSize, totalItems, itemsOnPage);

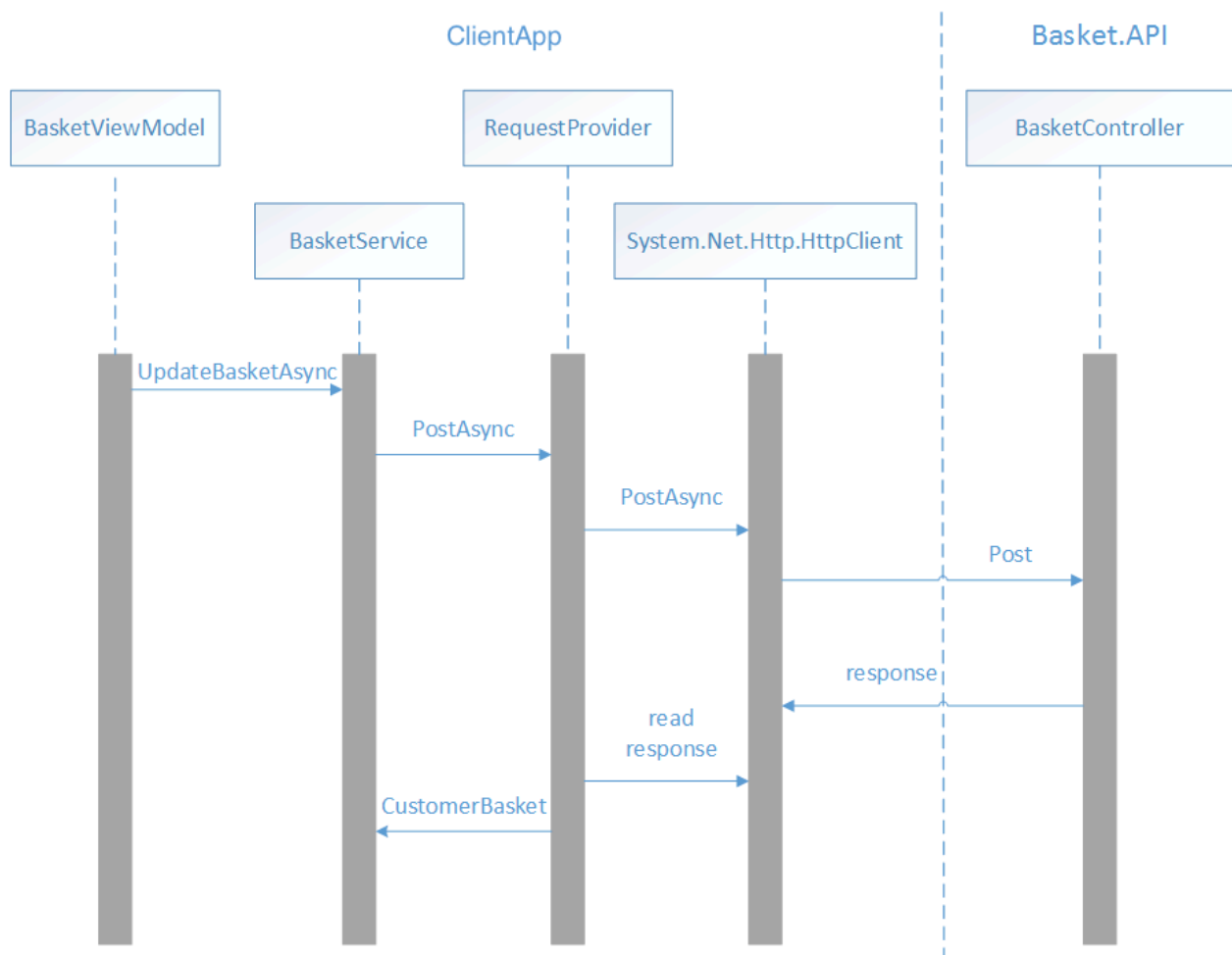
    return Ok(model);
}
```

Este método recupera los datos del catálogo de la base de datos SQL mediante [EntityFramework](#) y lo devuelve como un mensaje de respuesta que incluye un código de estado HTTP correcto y una colección de instancias `CatalogItem` con formato JSON.

Realización de una solicitud POST

La clase `BasketService` se usa para administrar el proceso de recuperación y actualización de datos con el microservicio de la cesta de la compra. En el método `RegisterAppServices` de la clase `MauiProgram`, la clase `BasketService` se registra como una asignación de tipos con el tipo `IBasketService` con el contenedor de inserción de dependencias. A continuación, cuando se crea una instancia de la clase `BasketViewModel`, su constructor acepta un tipo `IBasketService`, que el contenedor de inserción de dependencias resuelve, y devuelve una instancia de la clase `BasketService`. Para más información sobre la inserción de dependencias, vea [Inserción de dependencias](#).

En la imagen siguiente se muestra la interacción de las clases que envían los datos de la cesta de la compra mostrados por `BasketView` al microservicio de cesta de la compra.



Cuando se agrega un elemento a la cesta de la compra, se llama al método `ReCalculateTotalAsync` de la clase `BasketViewModel`. Este método actualiza el valor total de los elementos de la cesta de la compra y envía los datos de la cesta al microservicio de cesta de la compra, como se muestra en el ejemplo de código siguiente:

C#

```
private async Task ReCalculateTotalAsync()
{
    // Omitted for brevity...

    await _basketService.UpdateBasketAsync(
        new CustomerBasket
        {
            BuyerId = userInfo.UserId,
            Items = BasketItems.ToList()
        },
        authToken);
}
```

Este método llama al método `UpdateBasketAsync` de la instancia `BasketService` insertada en `BasketViewModel` por el contenedor de inserción de dependencias. El método siguiente muestra el método `UpdateBasketAsync`:

C#

```
public async Task<CustomerBasket> UpdateBasketAsync(
    CustomerBasket customerBasket, string token)
{
    UriBuilder builder = new
UriBuilder(GlobalSetting.Instance.BasketEndpoint);
    string uri = builder.ToString();
    var result = await _requestProvider.PostAsync(uri, customerBasket,
token);
    return result;
}
```

Este método compila el URI que identifica el recurso al que se enviará la solicitud y usa la clase `RequestProvider` para invocar el método HTTP POST en el recurso, antes de devolver los resultados a `BasketViewModel`. Tenga en cuenta que se requiere un token de acceso obtenido de `IdentityServer` durante el proceso de autenticación para autorizar las solicitudes al microservicio de cesta de la compra. Para obtener más información acerca de la autorización, vea [Autorización](#).

En el ejemplo de código siguiente se muestra uno de los métodos `PostAsync` de la clase `RequestProvider`:

C#

```
public async Task<TResult> PostAsync<TResult>(
    string uri, TResult data, string token = "", string header = "")
{
    HttpClient httpClient = GetOrCreateHttpClient(token);

    var content = new StringContent(JsonSerializer.Serialize(data));
    content.Headers.ContentType = new
MediaTypeHeaderValue("application/json");
    HttpResponseMessage response = await httpClient.PostAsync(uri, content);

    await HandleResponse(response);
    TResult result = await response.Content.ReadFromJsonAsync<TResult>();

    return result;
}
```

Este método llama al método `GetOrCreateHttpClient`, que devuelve una instancia de la clase `HttpClient` con los encabezados adecuados establecidos. A continuación, envía una solicitud POST asincrónica al recurso identificado por el URI, con los datos de la cesta de la compra serializados que se envían en formato JSON y la respuesta que se almacena en la instancia `HttpResponseMessage`. A continuación, se invoca el método `HandleResponse`, que produce una excepción si la respuesta no incluye un código de

estado HTTP correcto. A continuación, la respuesta se lee como una cadena, se convierte de JSON en un objeto `CustomerBasket` y se devuelve a `BasketService`. Para obtener más información sobre el método `GetOrCreateHttpClient`, vea [Realización de una solicitud GET](#).

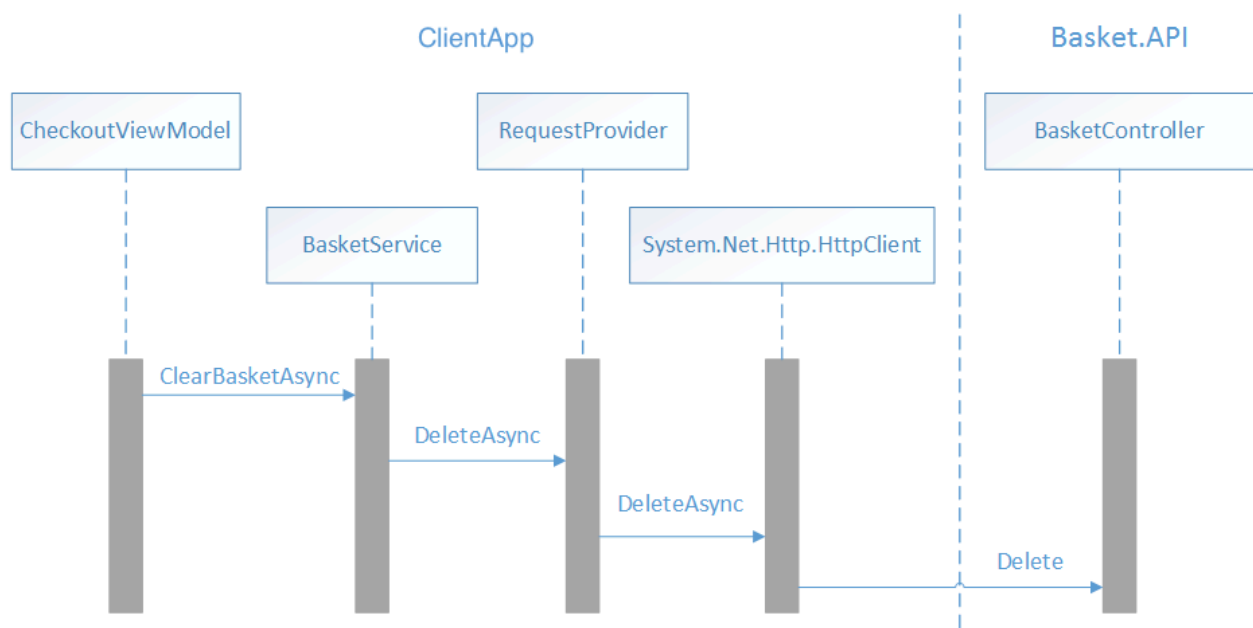
Cuando el método `PostAsync` de la clase `RequestProvider` llama a `HttpClient.PostAsync`, se invoca el método `Post` de la clase `BasketController` del proyecto `Basket.API`, que se muestra en el ejemplo de código siguiente:

```
C#  
  
[HttpPost]  
public async Task<IActionResult> Post([FromBody] CustomerBasket value)  
{  
    var basket = await _repository.UpdateBasketAsync(value);  
    return Ok(basket);  
}
```

Este método usa una instancia de la clase `RedisBasketRepository` para conservar los datos de la cesta de la compra en la caché de Redis y los devuelve como un mensaje de respuesta que incluye un código de estado HTTP correcto y una instancia `CustomerBasket` con formato JSON.

Realización de una solicitud DELETE

En la imagen siguiente se muestran las interacciones de las clases que eliminan los datos de la cesta de la compra del microservicio de cesta de la compra, para `CheckoutView`.



Cuando se invoca el proceso de finalización de la compra, se llama al método `CheckoutAsync` de la clase `CheckoutViewModel`. Este método crea un pedido, antes de vaciar la cesta de la compra, como se muestra en el ejemplo de código siguiente:

C#

```
private async Task CheckoutAsync()
{
    // Omitted for brevity...

    await _basketService.ClearBasketAsync(
        _shippingAddress.Id.ToString(), authToken);
}
```

Este método llama al método `ClearBasketAsync` de la instancia `BasketService` insertada en `CheckoutViewModel` por el contenedor de inserción de dependencias. El método siguiente muestra el método `ClearBasketAsync`:

C#

```
public async Task ClearBasketAsync(string guidUser, string token)
{
    UriBuilder builder = new(GlobalSetting.Instance.BasketEndpoint);
    builder.Path = guidUser;
    string uri = builder.ToString();
    await _requestProvider.DeleteAsync(uri, token);
}
```

Este método compila el URI que identifica el recurso al que se enviará la solicitud y usa la clase `RequestProvider` para invocar el método HTTP `DELETE` en el recurso. Tenga en cuenta que se requiere un token de acceso obtenido de `IdentityServer` durante el proceso de autenticación para autorizar las solicitudes al microservicio de cesta de la compra. Para obtener más información acerca de la autorización, vea [Autorización](#).

En el ejemplo de código siguiente se muestra el método `DeleteAsync` de la clase `RequestProvider`:

C#

```
public async Task DeleteAsync(string uri, string token = "")
{
    HttpClient httpClient = GetOrCreateHttpClient(token);
    await httpClient.DeleteAsync(uri);
}
```

Este método llama al método `GetOrCreateHttpClient`, que devuelve una instancia de la clase `HttpClient` con los encabezados adecuados establecidos. A continuación, envía una solicitud asíncrona `DELETE` al recurso identificado por el URI. Para obtener más información sobre el método `GetOrCreateHttpClient`, vea [Realización de una solicitud GET](#).

Cuando el método `DeleteAsync` de la clase `RequestProvider` llama a `HttpClient.DeleteAsync`, se invoca el método `Delete` de la clase `BasketController` del proyecto `Basket.API`, que se muestra en el ejemplo de código siguiente:

C#

```
[HttpDelete("{id}")]  
public void Delete(string id) =>  
    _repository.DeleteBasketAsync(id);
```

Este método usa una instancia de la clase `RedisBasketRepository` para eliminar los datos de la cesta de la compra de la caché de Redis.

Almacenamiento de datos en caché

El rendimiento de una aplicación se puede mejorar almacenando en la caché los datos a los que se accede con frecuencia a un almacenamiento rápido que se encuentra cerca de la aplicación. Si el almacenamiento rápido se encuentra más cerca de la aplicación que la fuente original, el almacenamiento en caché puede mejorar significativamente los tiempos de respuesta al recuperar datos.

La forma más común de almacenamiento en caché es el almacenamiento en caché de lectura, donde una aplicación recupera datos haciendo referencia a la memoria caché. Si los datos no se están en la caché, se recuperan del almacén de datos y se agregan a la caché. Las aplicaciones pueden implementar el almacenamiento en caché de lectura con el patrón cache-aside. Este patrón determina si el elemento está actualmente en la memoria caché. Si los datos no se están en la caché, se recuperan del almacén de datos y se agregan a la caché. Para obtener más información, consulte el patrón [Cache-Aside](#) en Microsoft Docs.

Sugerencia

Almacene en caché los datos que se leen con frecuencia y que cambian con poca frecuencia.

Estos datos pueden agregarse a la caché a petición la primera vez que los recupera una aplicación. Esto significa que la aplicación debe capturar los datos una sola vez del almacén de datos y que el posterior acceso se puede satisfacer mediante la caché.

Las aplicaciones distribuidas, como la aplicación de referencia eShop, deben proporcionar una de las siguientes memorias caché o ambas:

- Una caché compartida a la que pueden acceder varios procesos o máquinas.
- Una caché privada, donde los datos se mantienen localmente en el dispositivo que ejecuta la aplicación.

La aplicación multiplataforma eShop usa una caché privada, donde los datos se mantienen localmente en el dispositivo que ejecuta una instancia de la aplicación. Para obtener información sobre la memoria caché usada por la aplicación de referencia eShop, vea [Microservicios de .NET: Arquitectura para aplicaciones .NET en contenedor](#) .

Sugerencia

Considere la caché como un almacén de datos transitorios que podrían desaparecer en cualquier momento.

Asegúrese de que los datos se mantienen tanto en el almacén de datos original como en la memoria caché. Las posibilidades de perder datos se minimizan si la memoria caché deja de estar disponible.

Administración de la expiración de datos

No es práctico esperar que los datos almacenados en la caché siempre sean coherentes con los datos originales. Los datos del almacén de datos original pueden cambiar después de haberse almacenado en caché, lo que hace que los datos almacenados en la memoria caché se vuelvan obsoletos. Por tanto, las aplicaciones deben implementar una estrategia que ayude a garantizar que los datos de la caché estén lo más actualizados posible, pero que también pueda detectar y tratar situaciones que surgen cuando los datos de la caché se han vuelto obsoletos. La mayoría de los mecanismos de almacenamiento en caché permiten configurar la memoria caché para que expiren los datos y, por tanto, reducir el período durante el que los datos podrían quedar obsoletos.

Sugerencia

Establezca una hora de expiración predeterminada al configurar una memoria caché.

Muchas de las memorias caché implementan una expiración que invalida los datos y los quita de la caché si no se accede a ellos durante un período especificado. Sin embargo, se debe tener cuidado al elegir el período de expiración. Si es demasiado corto, los datos expirarán demasiado rápido y se reducirán las ventajas del almacenamiento en caché. Si es demasiado largo, existe el riesgo de que los datos se queden obsoletos. Por lo tanto, la hora de expiración debe coincidir con el patrón de acceso a las aplicaciones que usan los datos.

Cuando los datos almacenados en caché expiran, se deben quitar de la memoria caché, y la aplicación debe recuperar los datos del almacén de datos original y volver a almacenarlos en la memoria caché.

También es posible que una caché se rellene si se permite que los datos permanezcan almacenados durante mucho tiempo. Por lo tanto, es posible que se requieran solicitudes para agregar nuevos elementos a la memoria caché para quitar algunos elementos en un proceso conocido como *expulsión*. Normalmente, los servicios de almacenamiento en caché expulsan los datos menos usados recientemente. Sin embargo, hay otras directivas de expulsión, como los menos usados recientemente o el primero en llegar, el primero en salir. Para obtener más información, vea [Guía sobre el almacenamiento en caché](#) en Microsoft Docs.

Almacenamiento en caché de imágenes

La aplicación multiplataforma eShop consume imágenes de productos remotas que se benefician de almacenarse en caché. Estas imágenes se muestran mediante el control Image. El control Image MAUI de .NET admite el almacenamiento en caché de imágenes descargadas que tiene habilitado el almacenamiento en caché de forma predeterminada y almacenará la imagen localmente durante 24 horas. Además, la hora de expiración se puede configurar con la propiedad CacheValidity. Para obtener más información, vea [Almacenamiento en caché de imágenes descargadas](#) en el Centro para desarrolladores de Microsoft.

Aumento de la resistencia

Todas las aplicaciones que se comunican con los recursos y servicios remotos deben ser sensibles a errores transitorios. Los errores transitorios incluyen la pérdida momentánea de conectividad de red a los servicios, la indisponibilidad temporal de un servicio o

tiempos de espera que surgen cuando un servicio está ocupado. Estos errores suelen ser de corrección automática y, si la acción se repite después de un intervalo adecuado, es probable que se realice correctamente.

Los errores transitorios pueden tener un gran impacto en la calidad percibida de una aplicación, incluso si se ha probado exhaustivamente en todas las circunstancias previsibles. Para asegurarse de que una aplicación que se comunica con servicios remotos funciona de forma confiable, debe poder hacer todo lo siguiente:

- Detectar errores cuando se produzcan y determinar si es probable que los errores sean transitorios.
- Reintentar la operación si se determina que es posible que el error sea transitorio y realizar un seguimiento de la cantidad de veces que se vuelve a intentar la operación.
- Usar una estrategia de reintento apropiada, que especifica el número de reintentos, el intervalo entre cada intento y las acciones a realizar después de un intento fallido.

Este control de errores transitorios se puede lograr ajustando todos los intentos de acceder a un servicio remoto en el código que implementa el patrón de reintento.

Patrón Retry

Si una aplicación detecta un error al intentar enviar una solicitud a un servicio remoto, puede tratar el error de alguna de las siguientes formas:

- Reintentar la operación. La aplicación podría reintentar la solicitud con errores inmediatamente.
- Reintentar la operación después de un retraso. La aplicación debe esperar el tiempo adecuado antes de reintentar la solicitud.
- Cancelar la operación. La aplicación debe cancelar la operación e informar de una excepción.

La estrategia de reintento se debe ajustar para que coincida con los requisitos empresariales de la aplicación. Por ejemplo, es importante optimizar el número y el intervalo de reintentos en la operación que se está intentando. Si la operación forma parte de una interacción del usuario, el intervalo de reintento debe ser corto, y solo unos pocos reintentos intentan evitar que los usuarios tengan que esperar una respuesta. Si la operación forma parte de un flujo de trabajo de larga duración, donde cancelar y reiniciar el flujo de trabajo resulta lento o costoso, conviene esperar más tiempo entre intentos y efectuar más reintentos.

ⓘ Nota

Una estrategia de reintentos agresiva con un retraso mínimo entre intentos, y un gran número de reintentos, podría degradar un servicio remoto que se ejecuta en su capacidad o próximo a esta. Además, dicha estrategia de reintento también podría afectar a la capacidad de respuesta de la aplicación si continuamente intenta realizar una operación con errores.

Si una solicitud sigue sin funcionar después de una serie de reintentos, lo mejor para la aplicación es impedir que las solicitudes adicionales vayan al mismo recurso y notificar un error. A continuación, después de un período establecido, la aplicación puede realizar una o varias solicitudes al recurso para ver si se realizan correctamente. Para más información, vea [Patrón de disyuntor](#).

💡 Sugerencia

Nunca implemente un mecanismo de reintento infinito. En su lugar, prefiera un retroceso exponencial.

Use un número finito de reintentos o implemente el patrón de [Disyuntor](#) para permitir que un servicio se recupere.

La aplicación de referencia eShop implementa el patrón de reintento. Para obtener más información, incluida una explicación de cómo combinar el patrón de reintento con la clase HttpClient, vea [Microservicios de .NET: arquitectura para aplicaciones .NET en contenedor](#) [↗].

Para obtener más información sobre el patrón de reintento, consulte el patrón [Retry](#) en Microsoft Docs.

Patrón de disyuntor

En algunas situaciones, pueden producirse errores debido a eventos anticipados que tardan más tiempo en corregirse. Estos errores pueden ir desde una pérdida parcial de conectividad hasta el fallo total del servicio. En estas situaciones, no tiene sentido que una aplicación reintente una operación que es poco probable que se realice correctamente y, en su lugar, debe aceptar que la operación ha provocado errores y tratarlos en consecuencia.

El patrón de disyuntor puede impedir que una aplicación intente ejecutar repetidamente una operación que es probable que produzca un error, al tiempo que permite a la

aplicación detectar si se ha resuelto el error.

ⓘ Nota

El propósito del patrón de disyuntor es diferente del patrón de reintento. El patrón Retry permite a una aplicación reintentar una operación esperando que se podrá ejecutar correctamente. El patrón de disyuntor impide que una aplicación realice una operación que es probable que tenga errores.

Un disyuntor actúa como un proxy para las operaciones que podrían producir errores. El proxy debe supervisar el número de errores recientes que se han producido y utilizar esta información para decidir si permitir que continúe la operación, o devolver de inmediato una excepción.

La aplicación multiplataforma eShop no implementa actualmente el patrón de disyuntor. Sin embargo, eShop sí. Para más información, consulte [Microservicios de .NET: arquitectura para aplicaciones .NET en contenedor](#).

💡 Sugerencia

Combine los patrones de reintento y disyuntor.

Una aplicación puede combinar estos dos patrones utilizando el patrón de reintento para invocar una operación mediante un disyuntor. Sin embargo, la lógica de reintento debe tener en cuenta las excepciones devueltas por el disyuntor y dejar de reintentar la operación si este indica que un error no es transitorio.

Para obtener más información sobre el patrón de disyuntor, consulte el patrón [Disyuntor](#) en Microsoft Docs.

Resumen

Muchas soluciones modernas basadas en web hacen uso de servicios web, hospedados en servidores web, para proporcionar funcionalidad a las aplicaciones cliente remotas. Las operaciones que expone un servicio web constituyen una API web, y las aplicaciones cliente deben poder usar la API web sin saber cómo se implementan los datos o las operaciones que la API expone.

El rendimiento de una aplicación se puede mejorar almacenando en la caché los datos a los que se accede con frecuencia a un almacenamiento rápido que se encuentra cerca de la aplicación. Las aplicaciones pueden implementar el almacenamiento en caché de

lectura con el patrón cache-aside. Este patrón determina si el elemento está actualmente en la memoria caché. Si los datos no se están en la caché, se recuperan del almacén de datos y se agregan a la caché.

Al comunicarse con las API web, las aplicaciones deben ser sensibles a errores transitorios. Los errores transitorios incluyen la pérdida momentánea de conectividad de red a los servicios, la indisponibilidad temporal de un servicio o tiempos de espera que surgen cuando un servicio está ocupado. Estos errores suelen ser de corrección automática y, si la acción se repite después de un intervalo adecuado, es probable que se realice correctamente. Por lo tanto, las aplicaciones deben ajustar todos los intentos de acceder a una API web en el código que implementa un mecanismo de control de errores transitorios.

Colaborar con nosotros en GitHub

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).



Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)

Autenticación y autorización

Artículo • 10/09/2024

💡 Sugerencia

Este contenido es un extracto del libro electrónico "Patrones de aplicaciones empresariales con .NET MAUI", disponible en [Documentación de .NET](#) o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF



La autenticación es el proceso de obtener credenciales de identificación, como el nombre y la contraseña de un usuario, y validar esas credenciales en una autoridad. La entidad que envió las credenciales se considera una identidad autenticada si las credenciales son válidas. Una vez establecida una identidad, un proceso de autorización determina si esa identidad tiene acceso a un recurso determinado.

Hay muchos enfoques para integrar la autenticación y la autorización en una aplicación .NET MAUI que se comunica con una aplicación web de ASP.NET, incluido el uso de ASP.NET Core Identity, proveedores de autenticación externos, como Microsoft, Google, Facebook o Twitter, y middleware de autenticación. La aplicación multiplataforma eShop realiza la autenticación y autorización con un microservicio de identidad contenedorizado que usa IdentityServer. La aplicación solicita tokens de seguridad de IdentityServer para autenticar a un usuario o acceder a un recurso. Para que IdentityServer emita tokens en nombre de un usuario, el usuario debe iniciar sesión en IdentityServer. Sin embargo, IdentityServer no proporciona una interfaz de usuario ni una base de datos para la autenticación. Por lo tanto, en la aplicación de referencia eShop, ASP.NET Core Identity se usa para este propósito.

Autenticación

Se requiere autenticación cuando una aplicación necesita conocer la identidad del usuario actual. El mecanismo principal de ASP.NET Core para identificar a los usuarios es el sistema de pertenencia a ASP.NET Core Identity, que almacena la información del usuario en un almacén de datos configurado por el desarrollador. Normalmente, este almacén de datos será un almacén de EntityFramework, aunque se pueden usar almacenes personalizados o paquetes de terceros para almacenar información de identidad en Azure Storage, DocumentDB u otras ubicaciones.

ASP.NET Core Identity es una solución adecuada para los escenarios de autenticación que usan un almacén de datos de usuario local y que conservan la información de la identidad entre las solicitudes mediante el uso de cookies (como es habitual en las aplicaciones web de ASP.NET). En cambio, las cookies no son una manera natural de conservar y transmitir datos. Por ejemplo, una aplicación web de ASP.NET Core que expone puntos de conexión de RESTful a los que se accede desde una aplicación normalmente tendrá que usar la autenticación por token de portador, ya que las cookies no se pueden usar en este escenario. Sin embargo, los tokens de portador se pueden recuperar e incluir fácilmente en el encabezado de autorización de las solicitudes web realizadas desde la aplicación.

Emisión de tokens de portador mediante IdentityServer

[IdentityServer](#) es un marco de código abierto de OpenID Connect y OAuth 2.0 para ASP.NET Core, que se puede usar para muchos escenarios de autenticación y autorización, incluida la emisión de tokens de seguridad para usuarios locales de ASP.NET Core Identity.

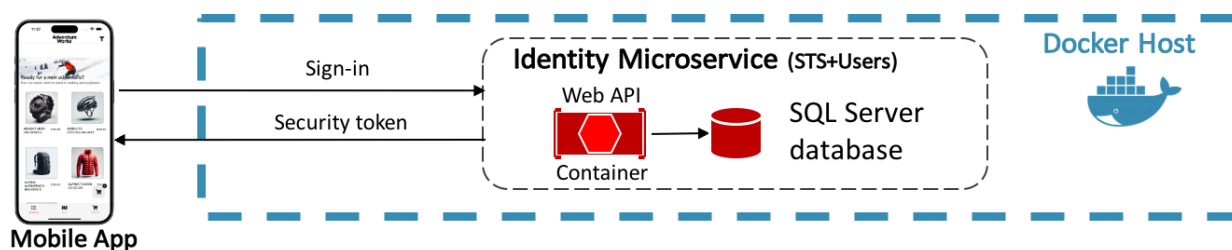
ⓘ Nota

OpenID Connect y OAuth 2.0 son muy similares, aunque tienen diferentes responsabilidades.

OpenID Connect es una capa de identidad basada en el protocolo OAuth 2.0. OAuth 2 es un protocolo que permite a las aplicaciones solicitar tokens de acceso desde un servicio de token de seguridad y usarlos para comunicarse con las API. Esta delegación reduce la complejidad tanto en las aplicaciones cliente como en las API, ya que la autenticación y la autorización se pueden centralizar.

OpenID Connect y OAuth 2.0 combinan los dos problemas de seguridad fundamentales de la autenticación y el acceso a la API, y IdentityServer es una implementación de estos protocolos.

En las aplicaciones que usan la comunicación directa de cliente a microservicio, como la aplicación de referencia eShop, se puede usar un microservicio de autenticación dedicado que actúa como servicio de token de seguridad (STS) para autenticar a los usuarios, como se muestra en el diagrama siguiente. Para obtener más información sobre la comunicación directa de cliente a microservicio, consulte [Microservicios](#).



La aplicación multiplataforma eShop se comunica con el microservicio de identidad, que usa IdentityServer para realizar la autenticación y el control de acceso para las API. Por lo tanto, la aplicación multiplataforma solicita tokens de IdentityServer, ya sea para autenticar a un usuario o para acceder a un recurso:

- La autenticación de usuarios con IdentityServer se logra mediante la aplicación multiplataforma que solicita un token de *identidad*, que representa el resultado de un proceso de autenticación. Como mínimo, contiene un identificador para el usuario e información sobre cómo y cuándo se autentica el usuario. También puede incluir datos de identidad adicionales.
- El acceso a un recurso con IdentityServer se logra mediante la aplicación multiplataforma que solicita un token de *acceso*, lo que permite acceder a un recurso de API. Los clientes solicitan tokens de acceso y los reenvían a la API. Los tokens de acceso contienen información sobre el cliente y el usuario, si están presentes. A continuación, las API usan esa información para autorizar el acceso a sus datos.

⚠ Nota

Un cliente debe registrarse con IdentityServer para poder solicitar tokens correctamente. Para obtener más información sobre cómo agregar clientes, vea [Definición de clientes](#).

Adición de IdentityServer a una aplicación web

Para que una aplicación web de ASP.NET Core use IdentityServer, debe agregarse a la solución de Visual Studio de la aplicación web. Para obtener más información, consulte [Configuración e información general](#) en la documentación de IdentityServer. Una vez que IdentityServer está incluido en la solución de Visual Studio de la aplicación web,

debe agregarse a su canalización de procesamiento de solicitudes HTTP para atender solicitudes a los puntos de conexión de OpenID Connect y OAuth 2.0. Esto se configura en el `Identity.API Program.cs` *del proyecto*, tal y como se muestra en el siguiente ejemplo de código:

C#

...

```
app.UseIdentityServer();
```

El orden es importante en la canalización de procesamiento de solicitudes HTTP de la aplicación web. Por lo tanto, `IdentityServer` debe agregarse a la canalización antes del marco de la interfaz de usuario que implementa la pantalla de inicio de sesión.

Configuración de IdentityServer

`IdentityServer` se configura en el `Identity.API Program.cs` *del proyecto* llamando al `AddIdentityServer` método, como se muestra en el ejemplo de código siguiente de la aplicación de referencia de eShop:

C#

```
builder.Services.AddIdentityServer(options =>
{
    options.Authentication.CookieLifetime = TimeSpan.FromHours(2);

    options.Events.RaiseErrorEvents = true;
    options.Events.RaiseInformationEvents = true;
    options.Events.RaiseFailureEvents = true;
    options.Events.RaiseSuccessEvents = true;

    // TODO: Remove this line in production.
    options.KeyManagement.Enabled = false;
})
.AddInMemoryIdentityResources(Config.GetResources())
.AddInMemoryApiScopes(Config.GetApiScopes())
.AddInMemoryApiResources(Config.GetApis())
.AddInMemoryClients(Config.GetClients(builder.Configuration))
.AddAspNetIdentity<ApplicationUser>()
// TODO: Not recommended for production - you need to store your key
material somewhere secure
.AddDeveloperSigningCredential();
```

Después de llamar al método `services.AddIdentityServer`, se llama a otras API fluidas para configurar lo siguiente:

- Credenciales usadas para firmar.
- Los recursos de identidad y de API a los que los usuarios podrían solicitar acceso.
- Los clientes que se conectarán para solicitar tokens.
- ASP.NET Core Identity.

Sugerencia

Cargue dinámicamente la configuración de IdentityServer. Las API de IdentityServer permiten configurar IdentityServer desde una lista en memoria de objetos de configuración. En la aplicación de referencia eShop, estas colecciones en memoria se codifican de forma rígida en la aplicación. Sin embargo, en escenarios de producción, se pueden cargar dinámicamente desde un archivo de configuración o desde una base de datos.

Para obtener información sobre cómo configurar IdentityServer para usar ASP.NET Core Identity, consulte [Uso de ASP.NET Core Identity](#) en la documentación de IdentityServer.

Configuración de recursos de API

Al configurar recursos de API, el método `AddInMemoryApiResources` espera una colección `IEnumerable<ApiResource>`. En el ejemplo de código siguiente se muestra el método `GetApis` que proporciona esta colección en la aplicación de referencia eShop:

```
C#

public static IEnumerable<ApiResource> GetApis()
{
    return new List<ApiResource>
    {
        new ApiScope("orders", "Orders Service"),
        new ApiScope("basket", "Basket Service"),
        new ApiScope("webhooks", "Webhooks registration Service"),
    };
}
```

Este método especifica que IdentityServer debe proteger las API de cesta de la compra y pedidos. Por lo tanto, se necesitarán tokens de acceso administrados por IdentityServer al realizar llamadas a estas API. Para obtener más información sobre el tipo `ApiResource`, consulte [Recurso de API](#) en la documentación de IdentityServer.

Configuración de recursos de identidad

Al configurar recursos de API, el método `AddInMemoryIdentityResources` espera una colección `IEnumerable<IdentityResource>`. Los recursos de identidad son datos como el identificador de usuario, el nombre o la dirección de correo electrónico. Cada recurso de identidad tiene un nombre único, y se le pueden asignar tipos de notificación arbitrarios, que se incluirán en el token de identidad del usuario. En el ejemplo de código siguiente se muestra el método `GetResources` que proporciona esta colección en la aplicación de referencia eShop:

C#

```
public static IEnumerable<IdentityResource> GetResources()
{
    return new List<IdentityResource>
    {
        new IdentityResources.OpenId(),
        new IdentityResources.Profile()
    };
}
```

La especificación de OpenID Connect especifica algunos [recursos de identidad estándar](#). El requisito mínimo es que se proporcione compatibilidad para emitir un identificador único para los usuarios. Esto se logra mediante la exposición del recurso de identidad `IdentityResources.OpenId`.

ⓘ Nota

La clase `IdentityResources` admite todos los ámbitos definidos en la especificación de OpenID Connect (OpenID, correo electrónico, perfil, teléfono y dirección).

`IdentityServer` también admite la definición de recursos de identidad personalizados. Para obtener más información, consulte [Definición de recursos de identidad personalizados](#) en la documentación de `IdentityServer`. Para obtener más información sobre el tipo `IdentityResource`, consulte [Recurso de identidad](#) en la documentación de `IdentityServer`.

Configuración de clientes

Los clientes son aplicaciones que pueden solicitar tokens de `IdentityServer`. Normalmente, se debe definir la siguiente configuración para cada cliente como mínimo:

- Un identificador de cliente único.

- Las interacciones permitidas con el servicio de token (conocido como tipo de concesión).
- La ubicación a la que se envían los tokens de identidad y acceso (conocida como URI de redireccionamiento).
- Una lista de recursos a los que el cliente puede acceder (conocidos como ámbitos).

Al configurar clientes, el método `AddInMemoryClients` espera una colección `IEnumerable<Client>`. En el ejemplo de código siguiente se muestra la configuración de la aplicación multiplataforma eShop en el método `GetClients` que proporciona esta colección en la aplicación de referencia eShop:

C#

```
public static IEnumerable<Client> GetClients(Dictionary<string,string> clientsUrl)
{
    return new List<Client>
    {
        // Omitted for brevity
        new Client
        {
            ClientId = "maui",
            ClientName = "eShop MAUI OpenId Client",
            AllowedGrantTypes = GrantTypes.Code,
            //Used to retrieve the access token on the back channel.
            ClientSecrets =
            {
                new Secret("secret".Sha256())
            },
            RedirectUri = { configuration["MauiCallback"] },
            RequireConsent = false,
            RequirePkce = true,
            PostLogoutRedirectUri = { $"
{configuration["MauiCallback"]}Account/Redirecting" },
            AllowedScopes = new List<string>
            {
                IdentityServerConstants.StandardScopes.OpenId,
                IdentityServerConstants.StandardScopes.Profile,
                IdentityServerConstants.StandardScopes.OfflineAccess,
                "orders",
                "basket",
                "mobileshoppingagg",
                "webhooks"
            },
            //Allow requesting refresh tokens for long lived API access
            AllowOfflineAccess = true,
            AllowAccessTokensViaBrowser = true,
            AlwaysIncludeUserClaimsInIdToken = true,
            AccessTokenLifetime = 60 * 60 * 2, // 2 hours
            IdentityTokenLifetime = 60 * 60 * 2 // 2 hours
        }
    }
}
```

```
};  
}
```

Esta configuración especifica los datos de las siguientes propiedades:

 Expandir tabla

Propiedad	Descripción
<code>ClientId</code>	Un identificador único para el cliente.
<code>ClientName</code>	El nombre para mostrar del cliente, que se usa para el registro y la pantalla de consentimiento.
<code>AllowedGrantTypes</code>	Especifica cómo un cliente quiere interactuar con IdentityServer. Para obtener más información, consulte Configuración del flujo de autenticación .
<code>ClientSecrets</code>	Especifica las credenciales del secreto de cliente que se usan al solicitar tokens desde el punto de conexión del token.
<code>RedirectUri</code>	Especifica los URI permitidos a los que se van a devolver tokens o códigos de autorización.
<code>RequireConsent</code>	Especifica si se requiere una pantalla de consentimiento.
<code>RequirePkce</code>	Especifica si los clientes que usan un código de autorización deben enviar una clave de prueba.
<code>PostLogoutRedirectUri</code>	Especifica los URI permitidos a los que se realizará el redireccionamiento después de cerrar la sesión.
<code>AllowedCorsOrigins</code>	Especifica el origen del cliente para que IdentityServer pueda permitir llamadas entre orígenes desde el origen.
<code>AllowedScopes</code>	Especifica los recursos a los que tiene acceso el cliente. De forma predeterminada, un cliente no tiene acceso a ningún recurso.
<code>AllowOfflineAccess</code>	Especifica si el cliente puede solicitar tokens de actualización.
<code>AllowAccessTokensViaBrowser</code>	Especifica si el cliente puede recibir tokens de acceso desde una ventana del explorador.
<code>AlwaysIncludeUserClaimsInIdToken</code>	Especifica que las notificaciones de usuario siempre se agregarán al token de identificador. De forma predeterminada, estos tendrían que recuperarse mediante el punto de conexión <code>userinfo</code> .
<code>AccessTokenLifetime</code>	Especifica la duración del token de acceso en segundos.

Propiedad	Descripción
<code>IdentityTokenLifetime</code>	Especifica la duración del token de identidad en segundos.

Configuración del flujo de autenticación

El flujo de autenticación entre un cliente y IdentityServer se puede configurar especificando los tipos de concesión en la propiedad `Client.AllowedGrantTypes`. Las especificaciones de OpenID Connect y OAuth 2.0 definen varios flujos de autenticación, entre los que se incluyen:

 Expandir tabla

Flujo de autenticación	Descripción
Implícita	Este flujo está optimizado para aplicaciones basadas en el explorador y se debe usar para las solicitudes de solo autenticación de usuario o de autenticación y token de acceso. Todos los tokens se transmiten a través del explorador y, por tanto, no se admiten características avanzadas como los tokens de actualización.
Código de autorización	Este flujo proporciona la capacidad de recuperar tokens en un canal posterior, en lugar del canal frontal del explorador, a la vez que admite la autenticación de cliente.
Híbrido	Este flujo es una combinación de los tipos de concesión de código implícito y de autorización. El token de identidad se transmite a través del canal del explorador y contiene la respuesta del protocolo firmado y otros artefactos, como el código de autorización. Después de validar correctamente la respuesta, se debe usar el canal posterior para recuperar el token de acceso y actualización.

Sugerencia

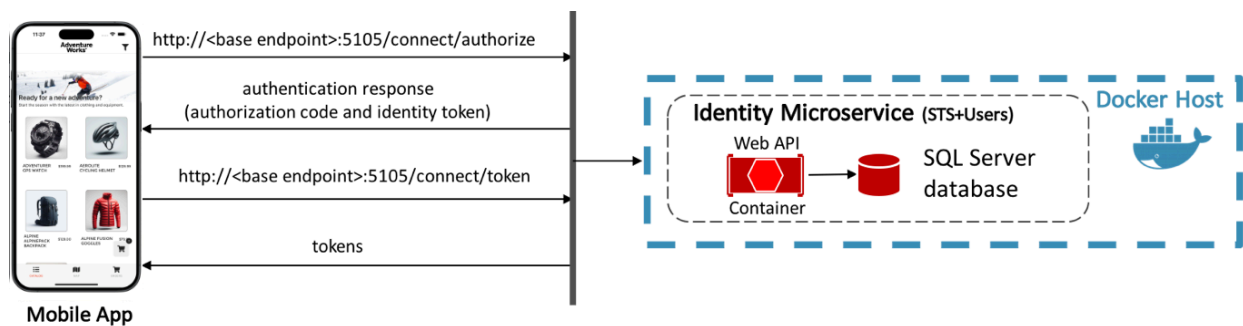
Considere la posibilidad de usar el flujo de autenticación híbrida. El flujo de autenticación híbrida mitiga varios ataques que se aplican al canal del explorador y es el flujo recomendado para las aplicaciones nativas que desean recuperar tokens de acceso, y posiblemente actualizar tokens.

Para obtener más información sobre los flujos de autenticación, consulte [Tipos de concesión](#)  en la documentación de IdentityServer.

Realización de la autenticación

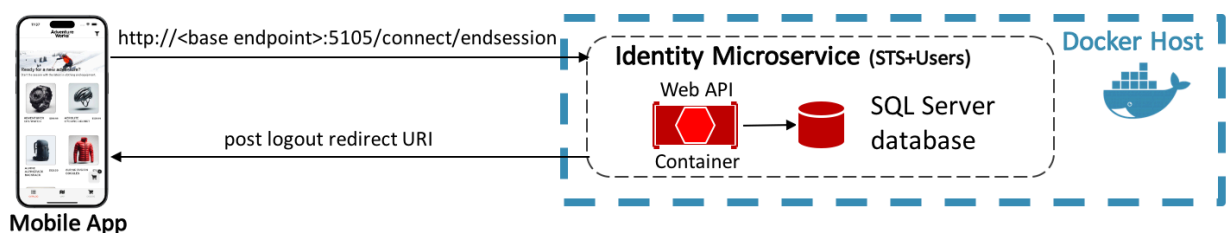
Para que IdentityServer emita tokens en nombre de un usuario, el usuario debe iniciar sesión en IdentityServer. Sin embargo, IdentityServer no proporciona una interfaz de usuario ni una base de datos para la autenticación. Por lo tanto, en la aplicación de referencia eShop, ASP.NET Core Identity se usa para este propósito.

La aplicación multiplataforma eShop se autentica con IdentityServer mediante el flujo de autenticación híbrida, que se ilustra en el diagrama siguiente.



Se realiza una solicitud de inicio de sesión a `<base endpoint>:5105/connect/authorize`. Después de una autenticación correcta, IdentityServer devuelve una respuesta de autenticación que contiene un código de autorización y un token de identidad. El código de autorización se envía a `<base endpoint>:5105/connect/token`, que responde con tokens de acceso, identidad y actualización.

La aplicación multiplataforma eShop cierra la sesión de IdentityServer mediante el envío de una solicitud a `<base endpoint>:5105/connect/endsession` con parámetros adicionales. Después de cerrar sesión, IdentityServer responde enviando un URI de redireccionamiento posterior al cierre de sesión a la aplicación multiplataforma. En el diagrama siguiente se ilustra este proceso.



En la aplicación multiplataforma eShop, la clase `IdentityService` realiza la comunicación con IdentityServer, que implementa la interfaz `IIdentityService`. La interfaz específica que la clase que realiza la implementación debe proporcionar los métodos `SignInAsync`, `SignOutAsync`, `GetUserInfoAsync` y `GetAuthTokenAsync`.

Inicio de sesión

Cuando el usuario pulsa el botón `LOGIN` en `LoginView`, se ejecuta `SignInCommand` en la clase `LoginViewModel`, que, a su vez, ejecuta el método `SignInAsync`. El siguiente ejemplo de código muestra este método:

C#

```
[RelayCommand]
private async Task SignInAsync()
{
    await IsBusyFor(
        async () =>
        {
            var loginSuccess = await
_appEnvironmentService.IdentityService.SignInAsync();

            if (loginSuccess)
            {
                await NavigationService.NavigateToAsync("//Main/Catalog");
            }
        });
}
```

Este método invoca el método `SignInAsync` en la clase `IdentityService`, como se muestra en el ejemplo de código siguiente:

C#

```
public async Task<bool> SignInAsync()
{
    var response = await GetClient().LoginAsync(new
LoginRequest()).ConfigureAwait(false);

    if (response.IsError)
    {
        return false;
    }

    await _settingsService
.SetUserTokenAsync(
    new UserToken
    {
        AccessToken = response.AccessToken,
        IdToken = response.IdentityToken,
        RefreshToken = response.RefreshToken,
        ExpiresAt = response.AccessTokenExpiration
    })
.ConfigureAwait(false);

    return !response.IsError;
}
```

El `IdentityService` usa el `oidcClient` proporcionado con el paquete NuGet `IdentityModel.OidcClient`. Este cliente muestra la vista web de autenticación al usuario en la aplicación y captura el resultado de la autenticación. El cliente se conecta al URI para el [punto de conexión de autorización](#) de IdentityServer con los parámetros necesarios. El punto de conexión de autorización está en `/connect/authorize` en el puerto 5105 del punto de conexión de base expuesto como una configuración del usuario. Para obtener más información sobre la configuración del usuario, consulte [Administración de configuración](#).

ⓘ Nota

La superficie expuesta a ataques de la aplicación multiplataforma eShop se reduce mediante la implementación de la clave de prueba para el intercambio de código (PKCE) en OAuth. PKCE protege el código de autorización para que no se use en caso de que se intercepte. Esto se logra mediante el cliente que genera un comprobador de secretos, cuyo hash se pasa en la solicitud de autorización, y que se presenta sin hash al canjear el código de autorización. Para obtener más información sobre PKCE, vea [Clave de prueba para intercambio de código por clientes públicos de OAuth](#) en el sitio web de Internet Engineering Task Force.

Login

Username

Password

☐ Remember me

The default users are alice/bob, password:
Pass123\$

Login

Si el punto de conexión del token recibe información de autenticación válida, código de autorización válido y un comprobador de secretos PKCE, responde con un token de acceso, un token de identidad y un token de actualización. El token de acceso (que permite el acceso a los recursos de API) y el token de identidad se almacenan como configuración de la aplicación y se realiza la navegación por páginas. Por lo tanto, el efecto general en la aplicación multiplataforma eShop es el siguiente: siempre que los usuarios puedan autenticarse correctamente con IdentityServer, navegan hasta la ruta `//Main/Catalog`, que es un elemento `TabbedPage` que muestra `CatalogView` como su pestaña seleccionada.

Para obtener información sobre la navegación por páginas, consulte [Navegación](#). Para obtener información sobre cómo la navegación de WebView hace que se ejecute un método de modelo de vista, consulte [Invocación de navegación mediante](#)

[comportamientos](#). Para obtener información sobre la configuración de la aplicación, consulte [Administración de configuración](#).

❗ Nota

eShop también permite un inicio de sesión ficticio cuando la aplicación está configurada para usar servicios ficticios en `SettingsView`. En este modo, la aplicación no se comunica con IdentityServer, sino que permite al usuario iniciar sesión con las credenciales.

Cierre de sesión

Cuando el usuario pulsa el botón `LOG OUT` en `ProfileView`, se ejecuta `LogoutCommand` en la clase `ProfileViewModel`, que ejecuta el método `LogoutAsync`. Este método realiza la navegación de página a la página `LoginView`, pasando un parámetro de consulta `Logout` establecido en `true`.

Ese parámetro se evalúa en el método `ApplyQueryAttributes`. Si el parámetro `Logout` está presente con un valor de `true`, se ejecuta el método `PerformLogoutAsync` de la clase `LoginViewModel`, que se muestra en el ejemplo de código siguiente:

C#

```
private async Task PerformLogoutAsync()
{
    await _appEnvironmentService.IdentityService.SignOutAsync();

    _settingsService.UseFakeLocation = false;

    UserName.Value = string.Empty;
    Password.Value = string.Empty;
}
```

Este método invoca el método `SignOutAsync` en la clase `IdentityService`, que invoca el `OidcClient` para finalizar la sesión del usuario y borra los tokens de usuario guardados. Para más información sobre la configuración de la aplicación, consulte [Administración de configuración](#). El siguiente ejemplo de código muestra el método `SignOutAsync`:

C#

```
public async Task<bool> SignOutAsync()
{
    var response = await GetClient().LogoutAsync(new
```

```
LogoutRequest()).ConfigureAwait(false);

    if (response.IsError)
    {
        return false;
    }

    await _settingsService.SetUserTokenAsync(default);

    return !response.IsError;
}
```

Este método usa el `OidcClient` para llamar al URI al [punto de conexión d finalización de la sesión](#) de IdentityServer con los parámetros necesarios. El punto de conexión de finalización de la sesión está en `/connect/endsession` en el puerto 5105 del punto de conexión de base expuesto como una configuración del usuario. Una vez que el usuario haya cerrado la sesión correctamente, se mostrará `LoginView` y se borrará cualquier información guardada del usuario.

Para obtener información sobre la navegación por páginas, consulte [Navegación](#). Para obtener información sobre cómo la navegación de `WebView` hace que se ejecute un método de modelo de vista, consulte [Invocación de navegación mediante comportamientos](#). Para obtener información sobre la configuración de la aplicación, consulte [Administración de configuración](#).

ⓘ Nota

eShop también permite un cierre de sesión ficticio cuando la aplicación está configurada para usar servicios ficticios en `SettingsView`. En este modo, la aplicación no se comunica con IdentityServer y, en su lugar, borra todos los tokens almacenados de la configuración de la aplicación.

Authorization

Después de la autenticación, las API web de ASP.NET Core a menudo necesitan autorizar el acceso, lo que permite que un servicio haga que las API estén disponibles para algunos usuarios autenticados, pero no para todos.

La restricción del acceso a una ruta de ASP.NET Core se puede lograr mediante la aplicación de un atributo `Authorize` a un controlador o una acción, que limita el acceso al controlador o la acción a los usuarios autenticados, como se muestra en el ejemplo de código siguiente:

C#

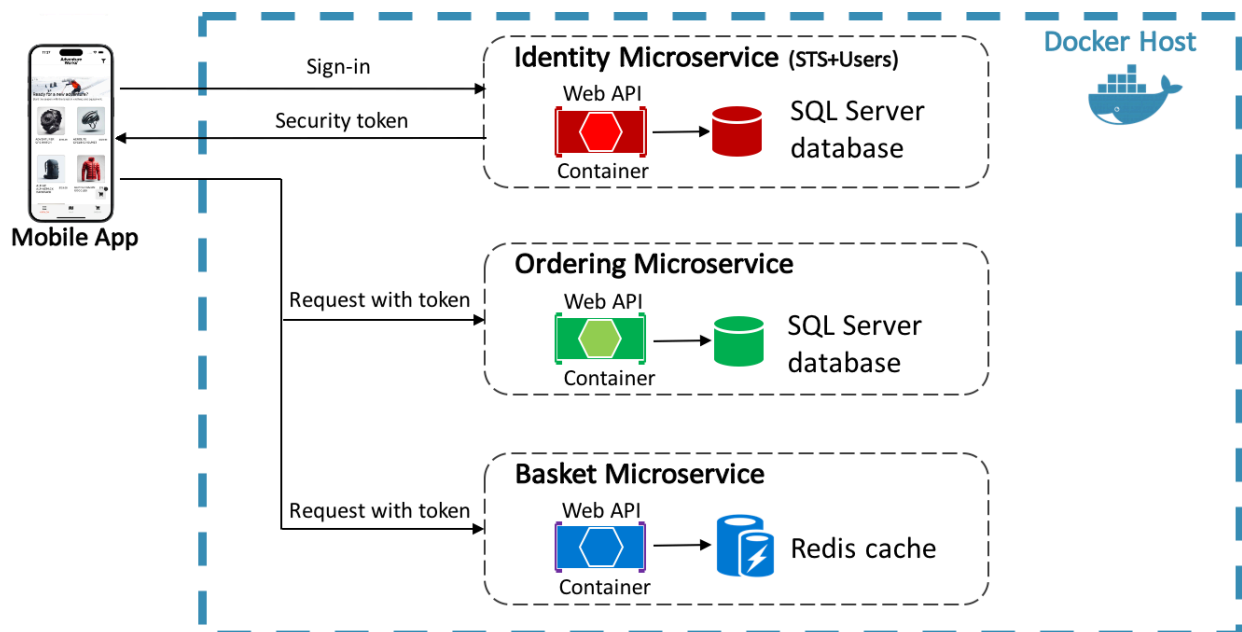
```
[Authorize]
public sealed class BasketController : Controller
{
    // Omitted for brevity
}
```

Si un usuario no autorizado intenta acceder a un controlador o una acción marcados con el atributo `Authorize`, el marco API devuelve un código de estado HTTP `401` (`unauthorized`).

❗ Nota

Los parámetros se pueden especificar en el atributo `Authorize` para restringir una API a usuarios específicos. Para obtener más información, consulte [Documentación de ASP.NET Core: Autorización](#).

IdentityServer se puede integrar en el flujo de trabajo de autorización para que los tokens de acceso proporcionen autorización de control. Este enfoque se muestra en el diagrama siguiente.



La aplicación multiplataforma eShop se comunica con el microservicio de identidad y solicita un token de acceso como parte del proceso de autenticación. A continuación, el token de acceso se reenvía a las API expuestas por los microservicios de pedidos y cesta de la compra como parte de las solicitudes de acceso. Los tokens de acceso contienen información sobre el cliente y el usuario. A continuación, las API usan esa información para autorizar el acceso a sus datos. Para obtener información sobre cómo configurar IdentityServer para proteger las API, consulte [Configuración de recursos de API](#).

Configuración de IdentityServer para realizar la autorización

Para realizar la autorización con IdentityServer, se debe agregar su middleware de autorización a la canalización de solicitudes HTTP de la aplicación web. El middleware se agrega en el método de extensión `AddDefaultAuthentication`, que se invoca desde el método `AddApplicationServices` en la clase `Program` y se muestra en el siguiente ejemplo de código de la aplicación de referencia de eShop:

C#

```
public static IServiceCollection AddDefaultAuthentication(this
IHostApplicationBuilder builder)
{
    var services = builder.Services;
    var configuration = builder.Configuration;

    var identitySection = configuration.GetSection("Identity");

    if (!identitySection.Exists())
    {
        // No identity section, so no authentication
        return services;
    }

    // prevent from mapping "sub" claim to nameidentifier.
    JwtTokenHandler.DefaultInboundClaimTypeMap.Remove("sub");

    services.AddAuthentication().AddJwtBearer(options =>
    {
        var identityUrl = identitySection.GetRequiredValue("Url");
        var audience = identitySection.GetRequiredValue("Audience");

        options.Authority = identityUrl;
        options.RequireHttpsMetadata = false;
        options.Audience = audience;
        options.TokenValidationParameters.ValidIssuers = [identityUrl];
        options.TokenValidationParameters.ValidateAudience = false;
    });

    services.AddAuthorization();

    return services;
}
```

Este método garantiza que solo se pueda acceder a la API con un token de acceso válido. El middleware valida el token entrante para garantizar que se envía desde un emisor de confianza y valida que el token es válido para usarse con la API que lo recibe.

Por lo tanto, la navegación al controlador de pedidos o de la cesta de la compra devolverá un código de estado HTTP `401 (unauthorized)`, que indica que se requiere un token de acceso.

Realización de solicitudes de acceso a las API

Al realizar solicitudes a los microservicios de pedidos y de cesta de la compra, el token de acceso obtenido de IdentityServer durante el proceso de autenticación debe incluirse en la solicitud, como se muestra en el ejemplo de código siguiente:

C#

```
public async Task CreateOrderAsync(models.Orders.Order newOrder)
{
    var authToken = await
        _identityService.GetAuthTokenAsync().ConfigureAwait(false);

    if (string.IsNullOrEmpty(authToken))
    {
        return;
    }

    var uri = $"
        {UriHelper.CombineUri(_settingsService.GatewayOrdersEndpointBase,
        ApiUrlBase)}?api-version=1.0";

    var success = await _requestProvider.PostAsync(uri, newOrder, authToken,
        "x-requestid").ConfigureAwait(false);
}
```

El token de acceso se almacena con la implementación de `IIdentityService` y se puede recuperar mediante el método `GetAuthTokenAsync`.

Del mismo modo, el token de acceso debe incluirse al enviar datos a una API protegida de IdentityServer, como se muestra en el ejemplo de código siguiente:

C#

```
public async Task ClearBasketAsync()
{
    var authToken = await
        _identityService.GetAuthTokenAsync().ConfigureAwait(false);

    if (string.IsNullOrEmpty(authToken))
    {
        return;
    }
}
```

```
await GetBasketClient().DeleteBasketAsync(new DeleteBasketRequest(),
CreateAuthenticationHeaders(authToken))
    .ConfigureAwait(false);
}
```

El token de acceso se recupera del `IIdentityService` y se incluye en la llamada al método `ClearBasketAsync` de la clase `BasketService`.

La clase `RequestProvider` de la aplicación multiplataforma eShop usa la clase `HttpClient` para realizar solicitudes a las API de RESTful expuestas por la aplicación de referencia eShop. Al realizar solicitudes a las API de pedidos y cesta de la compra, que requieren autorización, se debe incluir un token de acceso válido con la solicitud. Esto se logra agregando el token de acceso a los encabezados de la instancia `httpClient`, como se muestra en el ejemplo de código siguiente:

C#

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

La propiedad `DefaultRequestHeaders` de la clase `HttpClient` expone los encabezados que se envían con cada solicitud, y el token de acceso se agrega al encabezado `Authorization` cuyo prefijo es la cadena `Bearer`. Cuando la solicitud se envía a una API de RESTful, el valor del encabezado `Authorization` se extrae y se valida para garantizar que se envía desde un emisor de confianza y se usa para determinar si el usuario tiene permiso para invocar la API que lo recibe.

Para obtener más información sobre cómo la aplicación multiplataforma eShop realiza solicitudes web, consulte [Acceso a datos remotos](#).

Resumen

Hay muchos enfoques para integrar la autenticación y la autorización en una aplicación de .NET MAUI que se comunica con una aplicación web de ASP.NET. La aplicación multiplataforma eShop realiza la autenticación y autorización con un microservicio de identidad contenedorizado que usa IdentityServer. IdentityServer es un marco de código abierto de OpenID Connect y OAuth 2.0 para ASP.NET Core que se integra con ASP.NET Core Identity para realizar la autenticación de tokens de portador.

La aplicación multiplataforma solicita tokens de seguridad de IdentityServer para autenticar a un usuario o acceder a un recurso. Al acceder a un recurso, se debe incluir un token de acceso en la solicitud a las API que requieren autorización. El middleware

de IdentityServer valida los tokens de acceso entrantes para garantizar que se envían desde un emisor de confianza y que son válidos para usarlos con la API que los recibe.

Características del kit de herramientas de MVVM

Artículo • 30/05/2024

💡 Sugerencia

Este contenido es un extracto del libro electrónico "Patrones de aplicaciones empresariales con .NET MAUI", disponible en [Documentación de .NET](#) o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF



MVVM Toolkit

El patrón [Modelo-Vista-Modelo de vista \(MVVM\)](#) es una excelente base estructural para crear aplicaciones. En este patrón, el elemento Modelo de vista se convierte en la red troncal de nuestra aplicación, ya que proporciona la comunicación con nuestra interfaz de usuario de front-end y los componentes de respaldo. Para proporcionar integración con la interfaz de usuario, nos basaremos en las propiedades y comandos del Modelo de vista. Como se detalla en [Actualización de vistas en respuesta a los cambios en el modelo o el modelo de vista subyacentes](#), la interfaz `INotifyPropertyChanged` de nuestro Modelo de vista permite que se notifiquen los cambios en las propiedades cuándo se cambia el valor. La implementación de todas estas características significa que el Modelo de vista puede acabar convirtiéndose en muy detallado. Por ejemplo, el código siguiente muestra un Modelo de vista simple con propiedades que generan cambios:

C#

```
public class SampleViewModel : INotifyPropertyChanged
{
    private string _name;
    private int _value;
```

```

public event PropertyChangedEventHandler PropertyChanged;

public string Name
{
    get => _name;
    set => SetPropertyValue(ref _name, value);
}

public int Value
{
    get => _value;
    set => SetPropertyValue(ref _value, value);
}

protected void SetPropertyValue<T>(ref T storageField, T newValue,
[CallerMemberName] string propertyName = "")
{
    if (Equals(storageField, newValue))
        return;

    storageField = newValue;
    RaisePropertyChanged(propertyName);
}

protected virtual void RaisePropertyChanged([CallerMemberName] string
propertyName = "")
{
    PropertyChanged?.Invoke(this, new
PropertyChangedEventArgs(propertyName));
}
}

```

Aunque se podrían realizar algunas optimizaciones con el tiempo, terminaremos con un conjunto de código bastante detallado para definir nuestro Modelo de vista. Este código puede ser difícil de mantener y se vuelve propenso a errores.

Se puede usar el [paquete NuGet CommunityToolkit.Mvvm](#) (también conocido como kit de herramientas de MVVM) para ayudar a abordar y simplificar estos patrones MVVM comunes. El kit de herramientas de MVVM, junto con las características más recientes del lenguaje .NET, permite una lógica simplificada, una adopción sencilla en un proyecto e independencia del entorno de ejecución. En el ejemplo siguiente, se muestra el mismo Modelo de vista con los componentes que vienen con el kit de herramientas de MVVM:

C#

```

public partial class SampleViewModel : ObservableObject
{
    [ObservableProperty]
    private string _name;
}

```

```
[ObservableProperty]
private int _value;
}
```

❗ Nota

El kit de herramientas de MVVM se proporciona con el paquete `CommunityToolkit.Mvvm`. Para obtener información sobre cómo agregar el paquete al proyecto, consulte la [Introducción MVVM Toolkit](#) en el Centro para desarrolladores de Microsoft.

En comparación con el ejemplo original, se ha podido reducir drásticamente la complejidad general y simplificar la capacidad de mantenimiento del Modelo de vista. El kit de herramientas de MVVM incluye muchos componentes y características comunes generados previamente, como el elemento `ObservableObject` mostrado anteriormente, que simplifica y estandariza el código que tenemos en toda la aplicación.

ObservableObject

El kit de herramientas de MVVM proporciona el elemento `ObservableObject`, que está pensado para su uso como base de los objetos `ViewModel` o de cualquier objeto que necesite generar notificaciones de cambios. Implementa `INotifyPropertyChanged` e `INotifyPropertyChanging`, junto con métodos auxiliares para establecer propiedades y generar cambios. A continuación, se muestra un ejemplo de un Modelo de vista estándar que utiliza `ObservableObject`:

C#

```
public class SampleViewModel : ObservableObject
{
    private string _name;
    private int _value;

    public string Name
    {
        get => _name;
        set => SetProperty(ref _name, value);
    }

    public int Value
    {
        get => _value;
        set => SetProperty(ref _value, value);
    }
}
```

```
}  
}
```

`ObservableObject` controla toda la lógica necesaria para generar notificaciones de cambios mediante el método `SetProperty` en el establecedor de propiedades. Si tiene una propiedad que devuelve un elemento `Task<T>`, se puede usar el método `SetPropertyAndNotifyOnCompletion` para retrasar la publicación de un cambio de propiedad hasta que se haya completado la tarea. También se pueden usar los métodos `OnPropertyChanged` y `OnPropertyChanging` para generar cambios de propiedad cuando sea necesario en el objeto.

Para obtener información más detallada sobre `ObservableObject`, consulte [ObservableObject](#) en el Centro para desarrolladores del kit de herramientas de MVVM.

RelayCommand y AsyncRelayCommand

La interacción entre los controles de .NET MAUI (por ejemplo, pulsar un botón o seleccionar un elemento de una colección) y el Modelo de vista se realiza con la interfaz `ICommand`. .NET MAUI incluye una implementación predeterminada de `ICommand` con el objeto `Command`. El objeto `Command` de .NET MAUI es bastante básico y carece de compatibilidad con características más avanzadas, como la compatibilidad con el trabajo asíncrono y el estado de ejecución de los comandos.

El kit de herramientas de MVVM incluye dos comandos, `RelayCommand` y `AsyncRelayCommand`. `RelayCommand` está pensado para situaciones en las que haya código síncrono para ejecutar y tiene una implementación bastante similar al objeto `Command` de .NET MAUI.

⚠ Nota

Aunque el objeto `Command` de .NET MAUI y `RelayCommand` son similares, el uso de `RelayCommand` permite desacoplar el modelo de vista de cualquier referencia directa de .NET MAUI. Esto significa que el modelo de vista es más portátil, lo que facilita la reutilización entre proyectos.

`AsyncRelayCommand` proporciona muchas características adicionales a la hora de trabajar con flujos de trabajo asíncronos. Esto es bastante común en un modelo de vista, ya que normalmente nos comunicamos con repositorios, API, bases de datos y otros sistemas que usan `async/await`. El constructor de `AsyncRelayCommand` toma una tarea de ejecución definida como un elemento `Func<Task>` o un delegado que devuelva `Task`

como parte del constructor. Mientras se ejecuta la tarea de ejecución, `AsyncRelayCommand` supervisará el estado de la tarea y proporcionará actualizaciones mediante la propiedad `IsRunning`. La propiedad `IsRunning` se puede enlazar a la interfaz de usuario, lo que ayuda a administrar los estados de control, como mostrar la carga con un elemento `ActivityIndicator` o deshabilitar y habilitar un control. Mientras se ejecuta la tarea de ejecución, se puede llamar al método `Cancel` para intentar cancelar la tarea de ejecución, si se admite.

De manera predeterminada, `AsyncRelayCommand` no permite la ejecución simultánea. Esto es muy útil en situaciones en las que un usuario podría pulsar involuntariamente un control varias veces para ejecutar una operación de larga duración o costosa. Durante la ejecución de la tarea, `AsyncRelayCommand` llamará automáticamente al evento `CanExecuteChanged`. En .NET MAUI, los controles que admiten las propiedades `Command` y `CommandParameter`, como `Button`, escucharán este evento y lo habilitarán o deshabilitarán automáticamente durante la ejecución. Esta funcionalidad se puede invalidar mediante un parámetro `canExecute` personalizado o estableciendo la marca `AsyncRelayCommandOptions.AllowConcurrentExecutions` en el constructor.

Para obtener información más detallada sobre la implementación de los comandos, consulte la sección [Implementación de comandos](#) en el capítulo de MVVM. La información detallada sobre `RelayCommand` y `AsyncRelayCommand` está disponible en el documento [RelayCommand y AsyncRelayCommand](#) del Centro para desarrolladores del kit de herramientas de MVVM.

Generadores de origen

El uso de los componentes de serie del kit de herramientas de MVVM le permite simplificar considerablemente el modelo de vista. El kit de herramientas de MVVM permite simplificar aún más los casos de uso de código comunes mediante el uso de [generadores de código fuente](#). Los generadores de código fuente del kit de herramientas de MVVM buscan atributos específicos en el código y pueden generar contenedores para propiedades y comandos.

Importante

Los generadores de código fuente del kit de herramientas de MVVM generan código que se suma a nuestros objetos existentes. Por este motivo, cualquier objeto que aproveche un generador de código fuente se deberá marcar como `partial`.

El atributo `ObservableProperty` del kit de herramientas de MVVM se puede aplicar a campos de objetos que heredan de `ObservableObject` y encapsulará un campo privado con una propiedad que genera cambios. En el código siguiente, se muestra un ejemplo del uso del atributo `ObservableObject` en el campo `_name`:

C#

```
public partial class SampleViewModel : ObservableObject
{
    [ObservableProperty]
    private string _name;
}
```

Con el atributo `ObservableProperty` aplicado al campo `_name`, se ejecutará el generador de código fuente y generará otra clase parcial con el código siguiente:

C#

```
partial class SampleViewModel
{
    public string Name
    {
        get => _name;
        set
        {
            if (!EqualityComparer<string>.Default.Equals(_name, value))
            {
                OnNameChanging(value);
                OnPropertyChanged("Name");
                _name = value;
                OnNameChanged(value);
                OnPropertyChanged("Name");
            }
        }
    }
}
```

El elemento `SampleViewModel` generado ha utilizado el campo privado `_name` y ha generado una nueva propiedad `Name` que implementa toda la lógica necesaria para generar notificaciones de cambios.

El atributo `RelayCommand` del kit de herramientas de MVVM se puede aplicar a los métodos de un elemento `ObservableObject` y creará un elemento `RelayCommand` o `AsyncRelayCommand` correspondiente. En el código siguiente, se muestran ejemplos de uso del atributo `RelayCommand`:

C#

```

public partial class SampleViewModel : ObservableObject
{
    public INavigationService NavigationService { get; set; }

    [ObservableProperty]
    private string _name;

    [ObservableProperty]
    bool _isValid;

    [RelayCommand]
    private Task SettingsAsync()
    {
        return NavigationService.NavigateToAsync("Settings");
    }

    [RelayCommand]
    private void Validate()
    {
        IsValid = !string.IsNullOrEmpty(Name);
    }
}

```

El elemento `RelayCommand` aplicado al método `Validate` generará un elemento `RelayCommand` que valida `ValidateCommand` porque tiene un valor devuelto `void` y el método `SettingsAsync` generará un elemento `AsyncRelayCommand` llamado `SettingsCommand`. El generador de código fuente generará el código siguiente en otras clases parciales:

C#

```

partial class SampleViewModel
{
    private AsyncRelayCommand? settingsCommand;

    SettingsCommand => settingsCommand ??= new
    AsyncRelayCommand(SettingsAsync);
}

partial class SampleViewModel
{
    private RelayCommand? validateCommand;

    public IRelayCommand ValidateCommand => validateCommand ??= new
    RelayCommand(Validate);
}

```

El generador de código fuente se ha ocupado de toda la complejidad de encapsular los métodos del modelo de vista con una implementación de `ICommand`.

Para obtener más información sobre los generadores de código fuente del kit de herramientas de MVVM, consulte [Generadores de código fuente de MVVM](#) en el Centro para desarrolladores del kit de herramientas de MVVM.

Resumen

El kit de herramientas de MVVM es una excelente manera de estandarizar y simplificar el código del modelo de vista. El kit de herramientas de MVVM ofrece excelentes implementaciones de los componentes estándar de MVVM, como `ObservableObject` y `Async/RelayCommand`. Los generadores de código fuente ayudan a simplificar las propiedades y comandos del modelo de vista mediante la generación de todo el código reutilizable necesario para las interacciones de la interfaz de usuario. El kit de herramientas de MVVM ofrece aún más características al margen de lo que se ha mostrado en este capítulo. Para obtener más información sobre el kit de herramientas de MVVM, consulte [Introducción al kit de herramientas de MVVM](#) en el Centro para desarrolladores del kit de herramientas de MVVM.

Pruebas unitarias

Artículo • 30/05/2024

💡 Sugerencia

Este contenido es un extracto del libro electrónico "Patrones de aplicaciones empresariales con .NET MAUI", disponible en [Documentación de .NET](#) o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF



Las aplicaciones multiplataforma experimentan problemas similares a las aplicaciones basadas en web y las de escritorio. Los usuarios móviles variarán en función de sus dispositivos, conectividad de red, disponibilidad de servicios y otros diversos factores. Por lo tanto, las aplicaciones multiplataforma se deben probar tal y como se usarían en el mundo real para mejorar su calidad, confiabilidad y rendimiento. Se deben realizar muchos tipos de pruebas en una aplicación, incluidas las pruebas unitarias, las pruebas de integración y las pruebas de interfaz de usuario. Las pruebas unitarias son la forma más común y esencial para crear aplicaciones de alta calidad.

Una prueba unitaria toma una pequeña unidad de la aplicación, normalmente un método, lo aísla del resto del código y comprueba que se comporta según lo previsto. Su objetivo es comprobar que cada unidad de funcionalidad funciona según lo previsto, para que los errores no se propaguen en toda la aplicación. Detectar un error donde se produce es más eficaz que observar el efecto de un error indirectamente en un punto de error secundario.

Las pruebas unitarias tienen un mayor efecto en la calidad del código cuando son parte integral del flujo de trabajo de desarrollo de software. Las pruebas unitarias pueden actuar como documentación de diseño y especificaciones funcionales para una aplicación. En cuanto se haya escrito un método, se deben escribir pruebas unitarias que comprueben el comportamiento del método en respuesta a casos de datos de entrada

estándar, límites e incorrectos, y que comprueben las suposiciones explícitas o implícitas realizadas por el código. Como alternativa, con el desarrollo controlado por pruebas, las pruebas unitarias se escriben antes que el código. Para obtener más información sobre el desarrollo controlado por pruebas y cómo implementarlo, consulte [Tutorial: Desarrollo controlado por pruebas con el Explorador de pruebas](#).

ⓘ Nota

Las pruebas unitarias son muy eficaces frente a la regresión. Es decir, una funcionalidad que solía funcionar, pero que ha sido perturbada por una actualización incorrecta.

Las pruebas unitarias suelen usar el patrón organizar-actuar-afirmar:

[Expandir tabla](#)

Paso	Descripción
Organizar	Inicializa los objetos y establece el valor de los datos que se pasan al método objeto de las pruebas.
Actuar	Invoca al método objeto de las pruebas con los argumentos necesarios.
Assert	Comprueba si la acción del método objeto de las pruebas se comporta de la forma prevista.

Este patrón garantiza que las pruebas unitarias sean legibles, autodescriptivas y coherentes.

Inserción de dependencias y pruebas unitarias

Una de las motivaciones para adoptar una arquitectura de acoplamiento flexible es que facilita las pruebas unitarias. Uno de los tipos registrados con el servicio de inserción de dependencias es la interfaz `IAppEnvironmentService`. En el ejemplo de código siguiente, se muestra un esquema de esta clase:

C#

```
public class OrderDetailViewModel : ViewModelBase
{
    private IAppEnvironmentService _appEnvironmentService;

    public OrderDetailViewModel(
        IAppEnvironmentService appEnvironmentService,
```

```
IDialogService dialogService, INavigationService navigationService,
ISettingsService settingsService)
    : base(dialogService, navigationService, settingsService)
{
    _appEnvironmentService = appEnvironmentService;
}
}
```

La clase `OrderDetailViewModel` tiene una dependencia del tipo `IAppEnvironmentService`, que el contenedor de inserción de dependencias resuelve cuando crea una instancia de un objeto `OrderDetailViewModel`. Sin embargo, en lugar de crear un objeto `IAppEnvironmentService` que use servidores, dispositivos y configuraciones reales para hacer las pruebas unitarias de la clase `OrderDetailViewModel`, reemplaza el objeto `IAppEnvironmentService` por un objeto ficticio con fines de prueba. Un objeto ficticio es uno que tiene la misma firma que un objeto o una interfaz, pero que se crea de forma específica para ayudar con las pruebas unitarias. Se usa a menudo con la inserción de dependencias para proporcionar implementaciones específicas de interfaces para probar diferentes escenarios de datos y flujos de trabajo.

Este enfoque permite pasar el objeto `IAppEnvironmentService` a la clase `OrderDetailViewModel` en tiempo de ejecución y, en interés de la capacidad de prueba, permite pasar una clase ficticia a la clase `OrderDetailViewModel` en tiempo de pruebas. La principal ventaja de este enfoque es que permite ejecutar pruebas unitarias sin necesidad de recursos pesados, como las características de la plataforma en tiempo de ejecución, los servicios web o las bases de datos.

Prueba de aplicaciones MVVM

Probar modelos y ver modelos de aplicaciones MVVM es idéntico a probar cualquier otra clase y se usan las mismas herramientas y técnicas; esto incluye características como las pruebas unitarias y la simulación. Sin embargo, algunos patrones típicos para modelar y ver las clases del modelo se pueden beneficiar de técnicas específicas de las pruebas unitarias.

Sugerencia

Pruebe una cosa con cada prueba unitaria. A medida que se expande la complejidad de una prueba, hace que la comprobación de esa prueba sea más difícil. Al limitar una prueba unitaria a un único problema, podemos garantizar que nuestras pruebas sean más repetibles, aisladas y tengan un tiempo de ejecución

menor. Consulte [Procedimientos recomendados para pruebas unitarias con .NET](#) para conocer más procedimientos recomendados.

No se sienta tentado a hacer que prueba unitaria ejercite más de un aspecto del comportamiento de la unidad. Si lo hace, las pruebas son difíciles de leer y actualizar. También puede provocar confusión al interpretar un error.

La aplicación multiplataforma eShop usa [MSTest](#) para realizar las pruebas unitarias, que admite dos tipos diferentes de pruebas unitarias:

 Expandir tabla

Tipo de prueba	Atributo	Descripción
TestMethod	<code>TestMethod</code>	Define el método de prueba real que se va a ejecutar.
DataSource	<code>DataSource</code>	Pruebas que solo son verdaderas para un conjunto determinado de datos.

Las pruebas unitarias incluidas con la aplicación multiplataforma eShop son `TestMethod`, por lo que cada método de la prueba unitaria está decorado con el atributo `TestMethod`. Además de [MSTest](#) hay otros marcos de pruebas disponibles, como [NUnit](#) y [xUnit](#).

Prueba de la funcionalidad asincrónica

Al implementar el patrón MVVM, los modelos de vista suelen invocar operaciones en servicios, a menudo de forma asincrónica. Las pruebas del código que invoca estas operaciones suelen usar simulaciones como sustitución de los servicios reales. En el ejemplo de código siguiente, se muestra cómo probar la funcionalidad asincrónica pasando un servicio ficticio a un modelo de vista:

C#

```
[TestMethod]
public async Task OrderPropertyIsNotNullAfterViewModelInitializationTest()
{
    // Arrange
    var orderService = new OrderMockService();
    var orderViewModel = new OrderDetailViewModel(orderService);

    // Act
    var order = await orderService.GetOrderAsync(1,
GlobalSetting.Instance.AuthToken);
    await orderViewModel.InitializeAsync(order);
}
```

```
// Assert
Assert.IsNotNull(orderViewModel.Order);
}
```

Esta prueba unitaria comprueba que la propiedad `Order` de la instancia de `OrderDetailViewModel` tenga un valor después de invocar al método `InitializeAsync`. Se invoca al método `InitializeAsync` cuando se navega a la vista correspondiente del modelo de vista. Para más información sobre la navegación, consulte [Navegación](#).

Cuando se crea la instancia de `OrderDetailViewModel`, se espera que se especifique una instancia de `IOrderService` como argumento. Sin embargo, `OrderService` recupera datos de un servicio web. Por lo tanto, se especifica una instancia de `OrderMockService`, una versión ficticia de la clase `OrderService`, como argumento para el constructor `OrderDetailViewModel`. A continuación, se recuperan datos ficticios en lugar de comunicarse con un servicio web cuando se invoca al método `InitializeAsync` del modelo de vista, que usa operaciones de `IOrderService`.

Prueba de implementaciones de `INotifyPropertyChanged`

La implementación de la interfaz `INotifyPropertyChanged` permite que las vistas reaccionen ante los cambios que se originan en los modelos de vista y los modelos. Estos cambios no se limitan a los datos que se muestran en los controles; también se usan para controlar la vista, como los estados del modelo de vista que hacen que se inicien las animaciones o se deshabiliten los controles.

Las propiedades que se pueden actualizar directamente mediante la prueba unitaria se pueden probar adjuntando un controlador de eventos al evento `PropertyChanged` y comprobando si se genera el evento después de establecer un nuevo valor para la propiedad. En el ejemplo de código siguiente, se muestra una prueba de este tipo:

C#

```
[TestMethod]
public async Task SettingOrderPropertyShouldRaisePropertyChanged()
{
    var invoked = false;
    var orderService = new OrderMockService();
    var orderViewModel = new OrderDetailViewModel(orderService);

    orderViewModel.PropertyChanged += (sender, e) =>
    {
```

```

        if (e.PropertyName.Equals("Order"))
            invoked = true;
    };
    var order = await orderService.GetOrderAsync(1, GlobalSetting.Instance.AuthToken);
    await orderViewModel.InitializeAsync(order);

    Assert.IsTrue(invoked);
}

```

Esta prueba unitaria invoca al método `InitializeAsync` de la clase `OrderViewModel`, lo que hace que se actualice su propiedad `Order`. Se superará la prueba unitaria, siempre que se genere el evento `PropertyChanged` para la propiedad `Order`.

Prueba de la comunicación basada en mensajes

Los modelos de vista que usan la clase `MessagingCenter` para comunicarse entre clases de acoplamiento flexible se pueden probar de forma unitaria mediante la suscripción al mensaje enviado por el código objeto de las pruebas, como se muestra en el ejemplo de código siguiente:

C#

```

[TestMethod]
public void AddCatalogItemCommandSendsAddProductMessageTest()
{
    var messageReceived = false;
    var catalogService = new CatalogMockService();
    var catalogViewModel = new CatalogViewModel(catalogService);

    MessagingCenter.Subscribe<CatalogViewModel, CatalogItem>(
        this, MessageKeys.AddProduct, (sender, arg) =>
        {
            messageReceived = true;
        });
    catalogViewModel.AddCatalogItemCommand.Execute(null);

    Assert.IsTrue(messageReceived);
}

```

Esta prueba unitaria comprueba que `CatalogViewModel` publique el mensaje `AddProduct` en respuesta a la ejecución de su método `AddCatalogItemCommand`. Dado que la clase `MessagingCenter` admite suscripciones de mensajes de multidifusión, la prueba unitaria puede suscribirse al mensaje `AddProduct` y ejecutar un delegado de devolución de llamada en respuesta a su recepción. Este delegado de devolución de llamada,

especificado como una expresión lambda, establece un campo booleano utilizado por la instrucción `Assert` para comprobar el comportamiento de la prueba.

Prueba del control de excepciones

También se pueden escribir pruebas unitarias que comprueben que se generen excepciones específicas para acciones o entradas no válidas, como se muestra en el ejemplo de código siguiente:

C#

```
[TestMethod]
public void InvalidEventNameShouldThrowArgumentException()
{
    var behavior = new MockEventToCommandBehavior
    {
        EventName = "OnItemTapped"
    };
    var listView = new ListView();

    Assert.Throws<ArgumentException>
    (() => listView.Behaviors.Add(behavior));
}
```

Esta prueba unitaria producirá una excepción porque el control `ListView` no tiene un evento llamado `OnItemTapped`. El método `Assert.Throws<T>` es un método genérico donde `T` es el tipo de la excepción esperada. El argumento pasado al método `Assert.Throws<T>` es una expresión lambda que producirá la excepción. Por lo tanto, se superará la prueba unitaria siempre que la expresión lambda produzca una excepción `ArgumentException`.

Sugerencia

Evite escribir pruebas unitarias que examinen las cadenas de los mensajes de excepción. Las cadenas de los mensajes de excepción pueden cambiar con el tiempo, por lo que las pruebas unitarias que dependen de su presencia se consideran frágiles.

Prueba de la validación

Hay dos aspectos para probar la implementación de la validación: probar que las reglas de validación se hayan implementado correctamente y probar que la clase

`ValidatableObject<T>` se comporta según lo previsto.

La lógica de validación suele ser sencilla de probar, ya que normalmente es un proceso autocontenido en el que la salida depende de la entrada. Debe haber pruebas de los resultados de invocar al método `Validate` en cada propiedad que tenga al menos una regla de validación asociada, como se muestra en el ejemplo de código siguiente:

C#

```
[TestMethod]
public void CheckValidationPassesWhenBothPropertiesHaveDataTest()
{
    var mockViewModel = new MockViewModel();
    mockViewModel.Forename.Value = "John";
    mockViewModel.Surname.Value = "Smith";

    var isValid = mockViewModel.Validate();

    Assert.IsTrue(isValid);
}
```

Esta prueba unitaria comprueba que la validación se realice correctamente cuando las dos propiedades `ValidatableObject<T>` de la instancia de `MockViewModel` tienen datos.

Además de comprobar que la validación se realice correctamente, las pruebas unitarias de la validación también deben comprobar los valores de las propiedades `Value`, `IsValid` y `Errors` de cada instancia de `ValidatableObject<T>` para comprobar que la clase funciona según lo previsto. En el ejemplo de código siguiente, se muestra una prueba unitaria que hace esto:

C#

```
[TestMethod]
public void CheckValidationFailsWhenOnlyForenameHasDataTest()
{
    var mockViewModel = new MockViewModel();
    mockViewModel.Forename.Value = "John";

    bool isValid = mockViewModel.Validate();

    Assert.IsFalse(isValid);
    Assert.IsNotNull(mockViewModel.Forename.Value);
    Assert.IsNull(mockViewModel.Surname.Value);
    Assert.IsTrue(mockViewModel.Forename.IsValid);
    Assert.IsFalse(mockViewModel.Surname.IsValid);
    Assert.AreEqual(mockViewModel.Forename.Errors.Count(), 0);
    Assert.AreNotEqual(mockViewModel.Surname.Errors.Count(), 0);
}
```

Esta prueba unitaria comprueba que se produce un error en la validación cuando la propiedad `Surname` de `MockViewModel` no tiene ningún dato y las propiedades `Value`, `IsValid` y `Errors` de cada instancia de `ValidatableObject<T>` se han establecido correctamente.

Resumen

Una prueba unitaria toma una pequeña unidad de la aplicación, normalmente un método, lo aísla del resto del código y comprueba que se comporta según lo previsto. Su objetivo es comprobar que cada unidad de funcionalidad funciona según lo previsto, para que los errores no se propaguen en toda la aplicación.

Se puede aislar el comportamiento de un objeto sometido a prueba reemplazando los objetos dependientes por objetos ficticios que simulan el comportamiento de los objetos dependientes. Esto permite ejecutar pruebas unitarias sin necesidad de recursos pesados, como las características de la plataforma en tiempo de ejecución, los servicios web o las bases de datos.

Probar modelos y modelos de vista de aplicaciones MVVM es idéntico a probar cualquier otra clase y se pueden usar las mismas herramientas y técnicas.

Colaborar con nosotros en GitHub

El origen de este contenido se puede encontrar en GitHub, donde también puede crear y revisar problemas y solicitudes de incorporación de cambios. Para más información, consulte [nuestra guía para colaboradores](#).



Comentarios de .NET

.NET es un proyecto de código abierto. Seleccione un vínculo para proporcionar comentarios:

 [Abrir incidencia con la documentación](#)

 [Proporcionar comentarios sobre el producto](#)