Contents

Patrones de aplicación empresarial utilizando el libro electrónico de Xamarin.Forms

13/11/2018 • 9 minutes to read • Edit Online

Guía de arquitectura para el desarrollo de Xamarin.Forms adaptables, fáciles de mantener y probar las aplicaciones empresariales

Este libro electrónico proporciona instrucciones sobre cómo implementar el patrón Model-View-ViewModel (MVVM), la inserción de dependencias, navegación, validación y la administración de configuración, manteniendo el acoplamiento flexible. Además, también hay una guía sobre cómo realizar la autenticación y autorización con IdentityServer, acceso a datos de microservicios en contenedores y las pruebas unitarias.

Prefacio

Este capítulo explica el propósito y el ámbito de la guía y quién lo está destinado a.

Introducción

Los desarrolladores de aplicaciones empresariales enfrentan a varios desafíos que pueden modificar la arquitectura de la aplicación durante el desarrollo. Por lo tanto, es importante compilar una aplicación para que se puede modificar o extendido con el tiempo. Diseño de la capacidad de dichos adaptación puede ser difícil, pero normalmente implica dividir una aplicación en componentes discretos y con acoplamiento flexible que pueden integrarse fácilmente entre sí en una aplicación.

MVVM

El patrón Model-View-ViewModel (MVVM) ayuda a separar la lógica de negocios y la presentación de una aplicación desde su interfaz de usuario (UI). Mantener una separación clara entre la interfaz de usuario y la lógica de aplicación ayuda a abordar numerosos problemas de desarrollo y hacer más fácil probar una aplicación, mantener y evolucionar. Se pueden mejorar enormemente las oportunidades de reutilización de código y permite a los desarrolladores y diseñadores de interfaz de usuario más colaboran fácilmente al desarrollar sus respectivos partes de una aplicación.

Inserción de dependencias

Inserción de dependencias permite la separación de tipos concretos desde el código que dependa de estos tipos. Normalmente usa un contenedor que contiene una lista de registros y las asignaciones entre tipos e interfaces abstractos y los tipos concretos que implementan o amplían estos tipos.

Contenedores de inserción de dependencia reducen el acoplamiento entre objetos por lo que proporciona una funcionalidad para crear instancias de clase y administra su duración en función de la configuración del contenedor. Durante la creación de objetos, el contenedor inserta las dependencias que requiere el objeto en él. Si aún no se

han creado esas dependencias, el contenedor crea y resuelve primero sus dependencias.

Comunicación entre componentes débilmente acoplados

Xamarin.Forms `MessagingCenter` clase implementa la publicación-patrón, que permite la comunicación basada en mensajes entre los componentes que no son convenientes para vincular mediante referencias de objeto y el tipo de suscripción. Este mecanismo permite a los publicadores y suscriptores transmitir sin tener una referencia entre sí, lo que ayuda a reducir las dependencias entre componentes, mientras que permite que los componentes que se desarrollan y prueban de forma independiente.

Navegación

Xamarin.Forms incluye compatibilidad con la navegación de página, que normalmente da como resultado de la interacción del usuario con la interfaz de usuario o de la aplicación, como resultado de los cambios de estado controlado por la lógica interna. Sin embargo, navegación puede ser difícil de implementar en las aplicaciones que usan el patrón MVVM.

Este capítulo presenta un `NavigationService` (clase), que se usa para realizar la navegación de model first de vista de modelos de vista. Colocar en la vista lógica de navegación clases de modelo significa que la lógica puede realizarse a través de pruebas automatizadas. Además, el modelo de vista, a continuación, puede implementar la lógica para controlar el desplazamiento para garantizar que se apliquen determinadas reglas empresariales.

Validación

Cualquier aplicación que acepta entradas de los usuarios debe asegurarse de que la entrada es válida. Sin validación, el usuario puede proporcionar datos a los que hará que la aplicación producirá un error. La validación aplica las reglas de negocios y evita que un atacante inserta datos malintencionados.

En el contexto de Model-View-ViewModel (MVVM) de patrón, un modelo de vista o modelo a menudo se requerirá para realizar la validación de datos y señalar los errores de validación a la vista para que el usuario puede corregirlos.

Administración de configuraciones

La configuración permite la separación de datos que se configura el comportamiento de una aplicación desde el código, lo que permite el comportamiento que se puede cambiar sin volver a generar la aplicación. Configuración de la aplicación es datos que una aplicación crea y administra y configuración de usuario es la configuración de una aplicación personalizable que afectan al comportamiento de la aplicación y no requiere el ajuste de volver a frecuentes.

Microservicios en contenedores

Los Microservicios ofrecen un enfoque al desarrollo de aplicaciones e implementación que se adapte a los requisitos de agilidad, escalabilidad y confiabilidad de aplicaciones modernas en la nube. Una de las principales ventajas de los microservicios es que pueden ser escaladas horizontalmente de forma independiente, lo que significa que se puede escalar un área funcional específica que requiere más procesamiento de energía o ancho de banda para admitir la demanda, sin escalado innecesariamente áreas de la aplicación que no está experimentando una mayor demanda.

Autenticación y autorización

Existen muchos enfoques para integrar la autenticación y autorización en una aplicación de Xamarin.Forms que se comunica con una aplicación web ASP.NET MVC. En este caso, la autenticación y autorización se realizan con un microservicio en contenedor de identidad que usa 4 IdentityServer. IdentityServer es un marco de código abierto

de OAuth 2.0 y OpenID Connect para ASP.NET Core que se integra con ASP.NET Core Identity para realizar la autenticación de token de portador.

Acceso a datos remotos

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

Pruebas unitarias

Probar los modelos y los modelos de vista de las aplicaciones MVVM es idéntica a las pruebas de otras clases, y se pueden usar las mismas herramientas y técnicas. Sin embargo, hay algunos patrones típicos de modelo y las clases de modelo de vista, que pueden beneficiarse de las técnicas de pruebas de unidad específica.

Comentarios

Este proyecto tiene un sitio de la Comunidad, en el que puede publicar preguntas y proporcionar comentarios. El sitio de la Comunidad se encuentra en GitHub. Como alternativa, se pueden enviar comentarios sobre el libro electrónico a dotnet-architecture-ebooks-feedback@service.microsoft.com .

Vínculos relacionados

- Descargar libro electrónico (PDF de 2Mb)
- eShopOnContainers (GitHub) (ejemplo)

Prefacio para desarrollo de aplicaciones de empresa

09/06/2018 • 5 minutes to read • Edit Online

Este libro electrónico proporciona una guía sobre la creación de aplicaciones de empresa entre plataformas con Xamarin.Forms. Xamarin.Forms es un Kit de herramientas de interfaz de usuario entre plataformas que permite a los desarrolladores crear fácilmente usuario nativos diseños de interfaz que se pueden compartir entre plataformas, como la plataforma Universal de Windows (UWP), iOS y Android. Proporciona una solución completa para la empresa empleado (B2E), negocio a negocio (B2B) y empresariales para las aplicaciones de consumidor (B2C), ofrece la capacidad de compartir código entre todas las plataformas de destino y para ayudar a reducir el costo total de propiedad (TCO).

La guía proporciona una guía de arquitectura para el desarrollo de aplicaciones de empresa de Xamarin.Forms adaptables, fácil de mantener y pueden someterse a prueba. Se proporciona orientación sobre cómo implementar MVVM, inyección de dependencia, navegación, validación y la administración de configuración, al tiempo que mantiene el acoplamiento flexible. Además, también hay instrucciones acerca de cómo realizar la autenticación y autorización con IdentityServer, acceso a datos desde microservicios en contenedores y las pruebas unitarias.

La guía incluye código fuente de la aplicación móvil eShopOnContainersy código de fuente la eShopOnContainers hacen referencia a aplicación. La aplicación móvil eShopOnContainers es una aplicación de enterprise multiplataforma desarrollada con Xamarin.Forms, que se conecta a una serie de microservicios en contenedores conocidos como aplicación hacen referencia a la eShopOnContainers. Sin embargo, la aplicación móvil eShopOnContainers puede configurarse para consumir datos de servicios ficticios para quienes desea evitar la implementación de la microservicios en contenedores.

Lo que queda fuera del ámbito de esta guía

Esta guía está dirigida a los lectores que ya están familiarizados con Xamarin.Forms. Para obtener una introducción detallada a Xamarin.Forms, consulte el Xamarin.Forms documentación, y crear aplicaciones móviles con Xamarin.Forms.

La guía es complementaria a Microservicios .NET: arquitectura de aplicaciones de .NET en contenedores, que se centra en desarrollar e implementar microservicios en contenedores. Incluyen otras guías de vale la pena lectura diseño de la arquitectura y desarrollo de aplicaciones de Web modernas con ASP.NET Core y Microsoft Azure, en contenedores Docker Application Lifecycle con Microsoft Platform y herramientas, y plataforma de Microsoft y las herramientas de desarrollo de aplicaciones móviles.

¿Quién debe usar esta guía

Los destinatarios de esta guía son principalmente los programadores y arquitectos de que le gustaría obtener información sobre cómo diseñar e implementar aplicaciones de empresa entre plataformas con Xamarin.Forms.

Una audiencia secundaria es decisiones técnicas que le gustaría recibir una visión general de arquitectura y tecnología antes de decidir qué método seleccionar para el desarrollo de aplicaciones de empresa multiplataforma usando Xamarin.Forms.

Cómo usar esta guía

Esta guía se centra en la creación de aplicaciones de empresa entre plataformas con Xamarin.Forms. Por lo tanto, se debe leer en su totalidad para proporcionar una base de descripción de esas aplicaciones y sus consideraciones técnicas. La guía, junto con su aplicación de ejemplo, también puede servir como punto de inicio o la referencia para crear una nueva aplicación empresarial. Usar la aplicación de ejemplo asociados como plantilla para la nueva

aplicación, o para ver cómo organizar los componentes de una aplicación. A continuación, hacen referencia a esta guía para una guía de arquitectura.

No dude en reenviar a esta guía para los miembros del equipo para ayudar a garantizar una comprensión común de desarrollo de aplicaciones de empresa multiplataforma usando Xamarin.Forms. Tener todos los usuarios trabajan en un conjunto común de terminología y subyacente principios le ayudará a garantizar una aplicación coherente de los patrones arquitectónicos y procedimientos recomendados.

Vínculos relacionados

- Descargar libros electrónicos (PDF de 2Mb)
- eShopOnContainers (GitHub) (ejemplo)

Introducción al desarrollo de aplicaciones de empresa

09/06/2018 • 17 minutes to read • Edit Online

Independientemente de la plataforma, los desarrolladores de aplicaciones empresariales enfrentan a varios desafíos:

- Requisitos de las aplicaciones que pueden cambiar con el tiempo.
- Desafíos y nuevas oportunidades de negocio.
- Comentarios en curso durante el desarrollo que puede afectar significativamente al ámbito y los requisitos de la aplicación.

Con esto en mente, es importante compilar aplicaciones que se pueden modificar fácilmente o extendidas con el tiempo. Diseñar para tal adaptabilidad puede ser difícil, ya que requiere una arquitectura que permite que las partes individuales de la aplicación para desarrollar y probar de forma aislada sin afectar al resto de la aplicación por separado.

Muchas aplicaciones empresariales son bastante complejas como para requerir más de un desarrollador de software. Puede ser un desafío importante para decidir cómo diseñar una aplicación para que varios desarrolladores puedan trabajar eficazmente en diferentes partes de la aplicación de forma independiente, asegurándose de que las partes unificación sin problemas cuando se integra en la aplicación.

El enfoque tradicional para diseñar y compilar una aplicación de resultados en lo que se conoce como un *monolítico* aplicación, donde los componentes están estrechamente con ninguna separación clara entre ellos. Normalmente, este enfoque monolítico da lugar a las aplicaciones que son difíciles y eficaz de mantener, dado que pueden ser difícil de resolver los errores sin que ello interrumpa otros componentes en la aplicación, y puede ser difícil para agregar nuevas características o reemplazar las características existentes.

Una solución efectiva para estos desafíos es dividir una aplicación en componentes discretos y con acoplamiento flexible que pueden integrarse fácilmente juntos en una aplicación. Este enfoque ofrece varias ventajas:

- Permite la funcionalidad individual desarrollado, probado, extendido y mantenida por distintas personas o equipos.
- Promueve la reutilización y una separación clara de intereses entre capacidades horizontal de la aplicación, como la autenticación y acceso a datos y las capacidades verticales, como las funciones de negocio específicas de aplicación. Esto permite que las dependencias e interacciones entre los componentes de aplicación para administrarlos más fácilmente.
- Ayuda a mantener una separación de roles proporcionando a distintas personas o equipos, centrarse en una tarea específica o parte de la funcionalidad según su experiencia. En particular, proporciona una separación más nítida entre la interfaz de usuario y la lógica de negocios de la aplicación.

Sin embargo, hay muchos problemas que deben resolverse al crear particiones en una aplicación en componentes discretos y con acoplamiento flexible. Se incluyen los siguientes:

- Decidir cómo proporcionar una separación clara de intereses entre los controles de interfaz de usuario y su lógica. Una de las decisiones más importantes al crear una aplicación de empresa de Xamarin.Forms es si se debe colocar la lógica de negocios en archivos de código subyacente, o si se crea una separación clara de intereses entre los controles de interfaz de usuario y la lógica para que la aplicación más fácil de mantener y pueden someterse a prueba. Para obtener más información, consulte Model-View-ViewModel.
- Determinar si se debe usar un contenedor de inyección de dependencia. Contenedores de inyección de dependencia reducen la dependencia de acoplamiento entre objetos por lo que proporciona una funcionalidad para construir instancias de clases con sus dependencias insertados y administran su duración en función de la

configuración del contenedor. Para obtener más información, consulte inyección de dependencia.

- Elegir entre los eventos de plataforma que proporciona y flexible de acoplamiento comunicación basada en mensajes entre los componentes que son poco práctico vincular por referencias de objeto y el tipo. Para obtener más información, consulte Introducción a comunicarse entre débilmente acoplados componentes.
- Decidir cómo navegar entre páginas, incluido cómo invocar la navegación, y donde debe residir la lógica de navegación. Para obtener más información, consulte Navigation (Navegación).
- Determinar cómo validar proporcionados por el usuario son correctos. Debe incluir la decisión de cómo validar proporcionados por el usuario y cómo notificar al usuario sobre los errores de validación. Para obtener más información, consulte validación.
- Decidir cómo realizar la autenticación y cómo proteger los recursos con autorización. Para obtener más información, consulte autenticación y autorización.
- Determinar cómo obtener acceso a datos remotos de web services, incluido cómo recuperar datos de forma confiable y cómo almacenar en caché los datos. Para obtener más información, consulte obtiene acceso a datos remotos.
- Decidir cómo probar la aplicación. Para obtener más información, consulte Unit Testing.

Esta guía proporciona instrucciones acerca de estos problemas y se centra en la arquitectura para la creación de una aplicación empresarial de multiplataforma usando Xamarin.Forms y patrones principales. Tiene como objetivo ayudar a generar código comprobable, adaptable y fácil de mantener, arreglando comunes escenarios de desarrollo de aplicaciones de empresa de Xamarin.Forms y mediante la separación de las preocupaciones de presentación, lógica de presentación y entidades a través de la compatibilidad con las instrucciones del Modelo de Model-View-ViewModel (MVVM).

Aplicación de ejemplo

Esta guía incluye una aplicación de ejemplo, eShopOnContainers, que es una tienda en línea que incluye la funcionalidad siguiente:

- Autenticación y autorización en un servicio back-end.
- Examinar un catálogo de camisetas, tazas de café y otros elementos de marketing.
- Filtrar el catálogo.
- Ordenar los elementos del catálogo.
- Ver el historial de pedidos del usuario.
- Configuración de valores.

Arquitectura de la aplicación de ejemplo

Figura 1-1 proporciona una descripción general de la arquitectura de la aplicación de ejemplo.

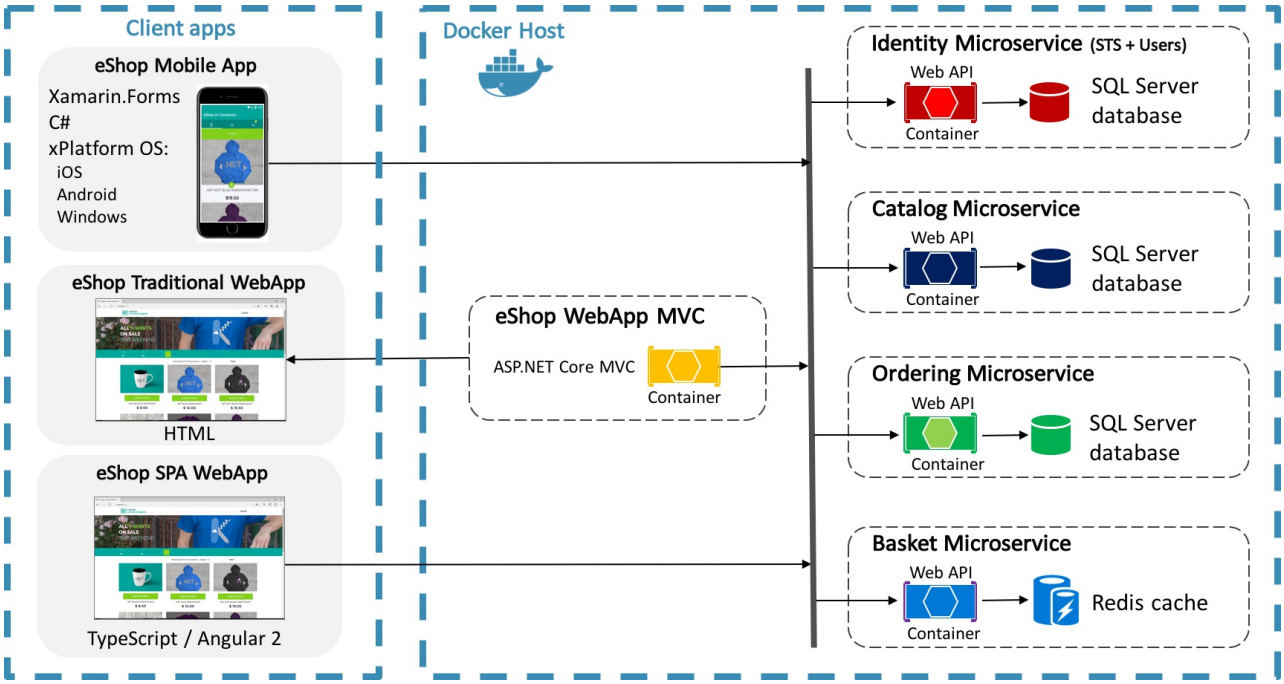

Figura 1-1: arquitectura de alto nivel eShopOnContainers

La aplicación de ejemplo incluye tres aplicaciones de cliente:

- Una aplicación de MVC desarrollado con ASP.NET Core.
- Una aplicación de página única (SPA) desarrollado con 2 Angular y Typescript. Este enfoque para las aplicaciones web evita realizar un envío y recepción en el servidor con cada operación.
- Una aplicación móvil se desarrolló con Xamarin.Forms, que es compatible con la plataforma Universal de Windows (UWP), iOS y Android.

Para obtener información acerca de las aplicaciones web, consulte diseño de la arquitectura y desarrollo de aplicaciones de Web modernas con ASP.NET Core y Microsoft Azure.

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

- Un microservicio de identidad, que usa ASP.NET Core Identity y IdentityServer.
- Un microservicio de catálogo, que es una creación controladas por datos, lee, actualizar y eliminar servicio (CRUD) que utiliza una base de datos de SQL Server mediante Entity Framework Core.
- Una ordenación microservicio, que es un servicio controlada por el dominio que usa patrones de diseño basado en dominio.
- Microservicio de la cesta de compra, que es un servicio CRUD controladas por datos que usa la caché en Redis.

Estos servicios back-end se implementan como microservicios con MVC de ASP.NET Core y se implementan como únicos contenedores dentro de un único host de Docker. Colectivamente, estos servicios back-end se denominan aplicaciones hacen referencia a la eShopOnContainers. Las aplicaciones cliente se comunican con los servicios back-end a través de una interfaz web de Representational State Transfer (REST). Para obtener más información acerca de microservicios y Docker, consulte Microservicios en contenedores.

Para obtener información acerca de la implementación de los servicios back-end, vea Microservicios. NET: arquitectura de aplicaciones de .NET en contenedores.

Aplicación móvil

Esta guía se centra en la creación de aplicaciones de empresa entre plataformas con Xamarin.Forms y utiliza la aplicación móvil eShopOnContainers como ejemplo. Figura 1-2 muestra las páginas de la aplicación móvil eShopOnContainers que proporcionan la funcionalidad que se describen anteriormente.

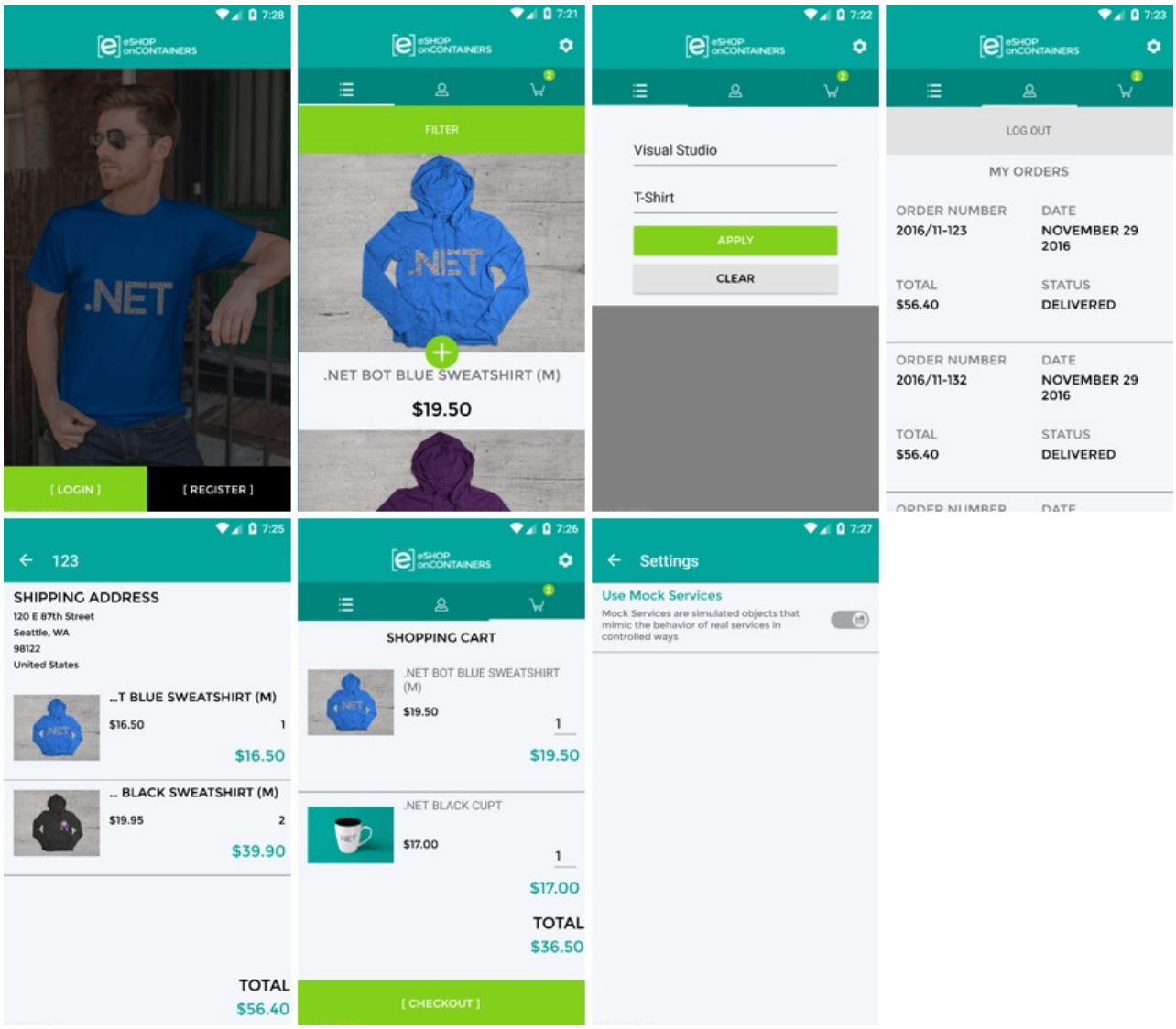

Figura 1-2: la aplicación móvil eShopOnContainers

La aplicación móvil consume los servicios back-end proporcionados por la aplicación de referencia de eShopOnContainers. Sin embargo, puede configurarse para consumir datos desde los servicios ficticios para quienes desea evitar la implementación de los servicios back-end.

La aplicación móvil eShopOnContainers ejercita la funcionalidad de Xamarin.Forms siguiente:

- XAML
- Controles
- Enlaces
- Convertidores de
- Estilos
- Animaciones
- Comandos
- comportamientos
- Desencadenadores
- Efectos
- Representadores personalizados
- MessagingCenter
- Controles personalizados

Para obtener más información acerca de esta funcionalidad, consulte la Xamarin.Forms documentación, y crear

aplicaciones móviles con Xamarin.Forms.

Además, las pruebas unitarias se proporcionan para que algunas de las clases en la aplicación móvil eShopOnContainers.

Solución de aplicación móvil

La solución de aplicación móvil eShopOnContainers organiza el código fuente y otros recursos en proyectos. Todos los proyectos usar carpetas para organizar el código fuente y otros recursos en categorías. En la tabla siguiente se describe los proyectos que forman la aplicación móvil eShopOnContainers:

PROYECTO	DESCRIPCIÓN
eShopOnContainers.Core	Este proyecto es el proyecto de biblioteca (PCL) de clases portable que contiene el código compartido y la interfaz de usuario compartida.
eShopOnContainers.Droid	Este proyecto contiene código específico de Android y es el punto de entrada para la aplicación de Android.
eShopOnContainers.iOS	Este proyecto contiene código específico de iOS y es el punto de entrada para la aplicación de iOS.
eShopOnContainers.UWP	Este proyecto contiene código específico de plataforma Universal de Windows (UWP) y es el punto de entrada para la aplicación de Windows.
eShopOnContainers.TestRunner.Droid	Este proyecto es el ejecutor de pruebas de Android para el proyecto eShopOnContainers.UnitTests.
eShopOnContainers.TestRunner.iOS	Este proyecto es el ejecutor de pruebas de iOS para el proyecto eShopOnContainers.UnitTests.
eShopOnContainers.TestRunner.Windows	Este proyecto es el ejecutor de pruebas de la plataforma Universal de Windows para el proyecto eShopOnContainers.UnitTests.
eShopOnContainers.UnitTests	Este proyecto contiene pruebas unitarias para el proyecto eShopOnContainers.Core.

Las clases de la aplicación móvil eShopOnContainers puede volver a usar en cualquier aplicación de Xamarin.Forms con poca o ninguna modificación.

eShopOnContainers.Core proyecto

El proyecto PCL eShopOnContainers.Core contiene las siguientes carpetas:

CARPETA	DESCRIPCIÓN
Animaciones	Contiene clases que habilitan las animaciones a ser consumidos en XAML.
comportamientos	Contiene los comportamientos que se exponen para ver las clases.
Controles	Contiene controles personalizados usados por la aplicación.
Convertidores de	Contiene los convertidores de valores que aplican la lógica personalizada a un enlace.

CARPETA	DESCRIPCIÓN
Efectos	Contiene el `EntryLineColorEffect` (clase), que se utiliza para cambiar el color del borde específicas de `Entry` controles.
Excepciones	Contiene la custom `ServiceAuthenticationException`.
Extensiones	Contiene métodos de extensión para la `VisualElement` y `IEnumerable` clases.
Aplicaciones auxiliares	Contiene clases de aplicación auxiliar para la aplicación.
Modelos	Contiene las clases del modelo de la aplicación.
Propiedades	Contiene `AssemblyInfo.cs`, un archivo de metadatos de ensamblado. NET.
Servicios	Contiene interfaces y clases que implementan los servicios que se proporcionan a la aplicación.
Desencadenadores	Contiene el `BeginAnimation` desencadenador, que se utiliza para invocar una animación en XAML.
Validaciones	Contiene las clases implicadas en la validación de entrada de datos.
ViewModels	Contiene la lógica de aplicación que se expone a las páginas.
Vistas	Contiene las páginas de la aplicación.

Proyectos de plataforma

Los proyectos de plataforma contienen implementaciones de efecto, las implementaciones de representador personalizado y otros recursos específicos de la plataforma.

Resumen

Plataformas y herramientas de desarrollo de aplicaciones móviles multiplataforma de Xamarin proporcionan una solución completa para clientes móviles B2E, B2B y B2C aplicaciones, ofrece la capacidad de compartir código entre todas las plataformas de destino (iOS, Android y Windows) y le ayuda a reducir el costo total de propiedad. Las aplicaciones pueden compartir su código de lógica usuario interfaz y la aplicación, conservando la apariencia y funcionamiento de la plataforma nativa.

Los desarrolladores de aplicaciones empresariales enfrentan a varios desafíos que pueden modificar la arquitectura de la aplicación durante el desarrollo. Por lo tanto, es importante compilar una aplicación para que se puede modificar o extender con el tiempo. Diseñar para tal adaptabilidad puede ser difícil, pero normalmente implica crear particiones de una aplicación en componentes discretos y con acoplamiento flexible que pueden integrarse fácilmente juntos en una aplicación.

Vínculos relacionados

- Descargar libros electrónicos (PDF de 2Mb)
- eShopOnContainers (GitHub) (ejemplo)

El patrón Model-View-ViewModel

13/07/2018 • 45 minutes to read • Edit Online

La experiencia del desarrollador Xamarin.Forms normalmente implica la creación de una interfaz de usuario en XAML y, a continuación, agregar código subyacente que opera en la interfaz de usuario. Como las aplicaciones se modifican y aumentan de tamaño y ámbito, pueden surgir problemas de mantenimiento complejas. Estos problemas incluyen el estrecho acoplamiento entre los controles de interfaz de usuario y la lógica de negocios, lo que aumenta el costo de realizar las modificaciones de la interfaz de usuario y la dificultad de este tipo de código de pruebas unitarias.

El patrón Model-View-ViewModel (MVVM) ayuda a separar la lógica de negocios y la presentación de una aplicación desde su interfaz de usuario (UI). Mantener una separación clara entre la interfaz de usuario y la lógica de aplicación ayuda a abordar numerosos problemas de desarrollo y hacer más fácil probar una aplicación, mantener y evolucionar. Se pueden mejorar enormemente las oportunidades de reutilización de código y permite a los desarrolladores y diseñadores de interfaz de usuario más colaboran fácilmente al desarrollar sus respectivos 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 una tiene una finalidad distinta. Figura 2-1 muestra las relaciones entre los tres componentes.

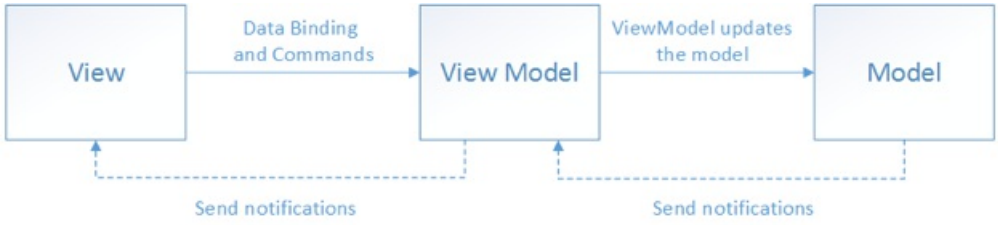

Figura 2-1: el patrón MVVM

Además de comprender las responsabilidades de cada componentes, también es importante entender cómo interactúan entre sí. En un nivel alto, la vista "conoce" el modelo de vista y el modelo de vista "conoce" el modelo, pero el modelo no es consciente de que el modelo de vista y el modelo de vista no es consciente de la vista. Por lo tanto, el modelo de vista aísla la vista del modelo y permite que el modelo evolucionar independientemente de la vista.

Las ventajas de usar el patrón MVVM son los siguientes:

- Si hay una implementación existente de modelo que encapsula la lógica de negocios existente, puede ser difícil o riesgo para cambiarlo. En este escenario, el modelo de vista actúa como un adaptador para las clases del modelo y le permite evitar realizar cambios importantes en el código de modelo.
- Los desarrolladores pueden crear pruebas unitarias para el modelo de vista y el modelo, sin usar la vista. Las pruebas unitarias para el modelo de vista pueden ejercer exactamente la misma funcionalidad que se usa la vista.
- La interfaz de usuario de la aplicación podrá modificarse sin tocar el código, siempre que la vista se implementa completamente en XAML. Por lo tanto, una nueva versión de la vista debe trabajar con el modelo de vista existente.
- Los diseñadores y desarrolladores pueden trabajar al mismo tiempo y de forma independiente en sus componentes durante el proceso de 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 uso eficaz de MVVM reside en la comprensión de cómo incluir código de aplicación en las clases correctas y comprender cómo interactúan las clases. Las siguientes secciones describen las responsabilidades de cada una de las clases en el 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 limitado código subyacente que no contienen lógica de negocios. Sin embargo, en algunos casos, el código subyacente podría contener lógica de la interfaz de usuario que implementa el comportamiento visual que es difícil de expresar en XAML, como animaciones.

En una aplicación de Xamarin.Forms, una vista es normalmente un `Page` -derivados o `ContentView` -clase derivada. Sin embargo, las vistas también pueden representarse mediante una plantilla de datos, que especifica los elementos de interfaz de usuario que se usará para representar visualmente un objeto cuando se muestre. Una plantilla de datos como una vista no tiene ningún código subyacente y está diseñada para enlazar a un tipo de modelo de vista concreta.

> **TIP**
> Evite habilitar y deshabilitar los elementos de interfaz de usuario en el código subyacente. Asegúrese de que los modelos de vista son responsables de definir los cambios de estado lógico que afectan a algunos aspectos de presentación de la vista, por ejemplo, si un comando está disponible, o una indicación de que una operación está pendiente. Por lo tanto, habilitar y deshabilitar los elementos de interfaz de usuario mediante un enlace a ver las propiedades del modelo, en lugar de habilitación y deshabilitación de código subyacente.

Hay varias opciones para ejecutar código en el modelo de vista en respuesta a interacciones en la vista, como un clic del botón o la selección de elementos. Si un control es compatible con los comandos, el control `Command` propiedad puede ser datos enlazados a un `ICommand` propiedad en el modelo de vista. Cuando se invoca el comando del control, se ejecutará el código en el modelo de vista. Además de los comandos, comportamientos pueden asociarse a un objeto en la vista y pueden escuchar para que un comando que se debe invocar o el evento. En respuesta, el comportamiento, a continuación, puede invocar un `ICommand` en el modelo de vista o un método en el modelo de vista.

ViewModel

El modelo de vista implementa propiedades y comandos a la que la vista puede enlazar datos a y notifica a la vista de cualquier cambio de estado a través de los eventos de notificación de cambio. Las propiedades y los comandos que proporciona el modelo de vista definen la funcionalidad que se les ofrezca la interfaz de usuario, pero la vista determina cómo se mostrará esa funcionalidad.

> **TIP**
> Que la interfaz de usuario siga respondiendo con operaciones asincrónicas. Aplicaciones móviles deben mantener el subproceso de interfaz de usuario desbloqueado para mejorar la percepción de rendimiento. Por lo tanto, en el modelo de vista, use métodos asincrónicos para operaciones de E/S y generar eventos para notificar de forma asincrónica 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 que son necesarias. Normalmente, hay una relación de uno a varios entre el modelo de vista y las clases del modelo. El modelo de vista puede optar por exponer directamente a la vista de clases del modelo para que los controles en la vista pueden enlazar los datos directamente a ellos. En este caso, las clases del modelo debe estar diseñado para admitir el enlace de datos y los eventos de notificación de cambio.

Cada modelo de vista proporciona los datos de un modelo en un formulario que puede consumir fácilmente la vista. Para lograr esto, el modelo de vista a veces realiza la conversión de datos. Colocar esta conversión de datos

en el modelo de vista es una buena idea, ya que proporciona propiedades que se puede enlazar la vista. Por ejemplo, el modelo de vista podría combinar los valores de dos propiedades que resulte más fácil para mostrar la vista.

> **TIP**
> Centralizar las conversiones de datos en una capa de conversión. También es posible usar convertidores de tipos 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 no proporciona el modelo de vista.

En orden para el modelo de vista participar en el enlace de datos bidireccional con la vista, sus propiedades deben generar el `PropertyChanged` eventos. Ver modelos cumplen este requisito mediante la implementación de la `INotifyPropertyChanged` interfaz y generar el `PropertyChanged` evento cuando se cambia una propiedad.

Para las colecciones, la vista fáciles `ObservableCollection<T>` se proporciona. Esta colección implementa la notificación de colección cambia, lo cual elimina el desarrollador tenga que implementar el `INotifyCollectionChanged` interfaz en colecciones.

Modelo

Clases de modelo son no visuales que encapsulan los datos de la aplicación. Por lo tanto, el modelo puede considerarse que representa el modelo de dominio de la aplicación, que normalmente incluye un modelo de datos junto con la lógica empresarial y la validación. Objetos de transferencia de datos (dto), objetos CLR estándar (poco) y entidad generada y objetos proxy son ejemplos de los objetos del modelo.

Clases de modelo se utilizan normalmente junto con los servicios o los repositorios que encapsulan el acceso a datos y almacenamiento en caché.

Conectar los modelos de vista a las vistas

Los modelos de vista se pueden conectar a las vistas mediante el uso de las capacidades de enlace de datos de Xamarin.Forms. Existen varios enfoques que pueden usarse para construir las vistas y ver modelos y las asocia en tiempo de ejecución. Estos enfoques se dividen en dos categorías, conocidas como composición de la primera vista y la primera composición del modelo de vista. Elección entre la composición de la primera vista y la composición del primer modelo es una cuestión de preferencias y la complejidad de la vista. No obstante, todos los enfoques comparten el mismo objetivo, que es para que la vista tiene un modelo de vista asignado a su propiedad BindingContext.

Con la vista de composición primero la aplicación conceptualmente se compone de las vistas que se conectan a los modelos de vista que dependen. La principal ventaja de este enfoque es que resulta fácil de construir el acoplamiento flexible, aplicaciones apta para las pruebas unitarias porque los modelos de vista no tienen ninguna dependencia en las vistas a sí mismos. También es fácil de entender la estructura de la aplicación siguiendo su estructura visual, en lugar de tener que realizar un seguimiento de ejecución del código para comprender cómo las clases se crean y asociadas. Además, construcción primera vista se alinea con el sistema de navegación de Xamarin.Forms que es responsable de construir páginas cuando se produce la navegación, lo que hace que una composición primer modelo de vista complejo y mal alineados con la plataforma.

Con la vista de composición del modelo primero la aplicación conceptualmente se compone de modelos de vista, con un servicio que se va a responsable de localizar la vista de un modelo de vista. Primera composición del modelo de vista se siente más natural para algunos desarrolladores, ya que la creación de vistas se puede resumir inmediatamente, lo que les permite centrarse en la estructura lógica que no son de interfaz de usuario de la aplicación. Además, permite modelos de vista que va a crear otros modelos de vista. Sin embargo, este enfoque suele ser complejo y puede ser difícil de entender cómo se crea y se asocian las distintas partes de la aplicación.

> **TIP**
> Mantener los modelos de vista y vistas independientes. El enlace de vistas para una propiedad en un origen de datos debe tener dependencias de entidad de seguridad de la vista en su modelo de vista correspondiente. En concreto, no hacen referencia a tipos de vistas, 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 que los defectos de software limitando el ámbito.

Las secciones siguientes tratan los enfoques principales para conectar los modelos de vista a las vistas.

Crear un modelo de vista de forma declarativa

Es el enfoque más sencillo para que la vista crear instancias mediante declaración a su modelo de vista correspondiente en XAML. Cuando se crea la vista, también se construirá el objeto de modelo de vista correspondiente. Este método se muestra en el ejemplo de código siguiente:

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

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

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

Crear un modelo de vista mediante programación

Una vista puede tener código en el archivo de código subyacente que da como resultado el modelo de vista que se asigna a su `BindingContext` propiedad. A menudo, esto se logra en el constructor de la vista, tal como se muestra en el ejemplo de código siguiente:

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

La construcción de programación y la asignación del modelo de vista de código subyacente de la vista tiene la ventaja que es sencillo. Sin embargo, la principal desventaja de este enfoque es que se debe proporcionar el modelo de vista con las dependencias necesarias. Uso de un contenedor de inserción de dependencia puede ayudar a mantener un acoplamiento flexible entre la vista y el modelo de vista. Para obtener más información, consulte inserción de dependencias.

Creación de una vista que se define como una plantilla de datos

Una vista puede definirse como una plantilla de datos y asociada con un tipo de modelo de vista. Plantillas de datos se pueden definir como recursos o se pueden definir en línea dentro del control que se mostrará el modelo de vista. El contenido del control es la instancia de modelo de vista y la plantilla de datos se usa para representar visualmente en él. Esta técnica es un ejemplo de una situación en la que el modelo de vista se crea una instancia en primer lugar, seguida de la creación de la vista.

Creación automática de un modelo de vista con un localizador de modelo de vista

Un localizador de modelo de vista es una clase personalizada que administra la creación de instancias de los

modelos de vista y su asociación a vistas. En la aplicación móvil de eShopOnContainers, el `ViewModelLocator` clase tiene una propiedad adjunta, `AutoWireViewModel`, que se utiliza para asociar los modelos de vista con las vistas. En XAML la vista, esta propiedad adjunta se establece en true para indicar que el modelo de vista se debe conectar automáticamente a la vista, tal como se muestra en el ejemplo de código siguiente:

```
viewModelBase:ViewModelLocator.AutoWireViewModel="true"
```

El `AutoWireViewModel` es una propiedad enlazable que se inicializa en false y, cuando se cambia su valor el `OnAutoWireViewModelChanged` se llama al controlador de eventos. Este método resuelve el modelo de vista para la vista. En el ejemplo de código siguiente se muestra la forma de conseguirlo:

```
private static void OnAutoWireViewModelChanged(BindableObject bindable, object oldValue, object newValue)
{
    var view = bindable as Element;
    if (view == null)
    {
        return;
    }

    var viewType = view.GetType();
    var viewName = viewType.FullName.Replace(".Views.", ".ViewModels.");
    var viewAssemblyName = viewType.GetTypeInfo().Assembly.FullName;
    var viewModelName = string.Format(
        CultureInfo.InvariantCulture, "{0}Model, {1}", viewName, viewAssemblyName);

    var viewModelType = Type.GetType(viewModelName);
    if (viewModelType == null)
    {
        return;
    }
    var viewModel = _container.Resolve(viewModelType);
    view.BindingContext = viewModel;
}
```

El `OnAutoWireViewModelChanged` método intenta resolver el modelo de vista con un enfoque basado en convenciones. Esta convención supone que:

- Modelos de vista están en el mismo ensamblado como tipos de vista.
- Las vistas están en una. Espacio de nombres de las vistas secundarias.
- Los modelos de vista están en una. Espacio de nombres secundario ViewModels.
- Los nombres de modelo de vista se corresponden con los nombres de vista y terminan con "ViewModel".

Por último, el `OnAutoWireViewModelChanged` método establece el `BindingContext` del tipo de vista para el tipo de modelo de vista resuelto. Para obtener más información acerca de cómo resolver el tipo de modelo de vista, consulte resolución.

Este enfoque tiene la ventaja de que una aplicación tiene una única clase que es responsable de la creación de instancias de los modelos de vista y su conexión a las vistas.

> **TIP**
> Utilice un localizador de modelo de vista para facilitar la sustitución. Un localizador de modelo de vista también sirve como punto de sustitución para implementaciones alternativas de dependencias, como para los datos en tiempo de diseño o las pruebas unitarias.

Actualizables vistas en respuesta a cambios en la base de ver el modelo

o el modelo

Todos los modelo de vista y las clases de modelo que se puede acceder a una vista deben implementar la `INotifyPropertyChanged` interfaz. Implementa esta interfaz en un modelo de vista o una clase de modelo permite a la clase proporcionar notificaciones de cambio a los controles enlazados a datos en la vista cuando cambia el valor de propiedad subyacente.

Las aplicaciones deben llevar a cabo para el uso correcto de notificación de cambio de propiedad, cumplir los requisitos siguientes:

- Generar siempre un `PropertyChanged` eventos si cambia el valor de una propiedad pública. No suponga que genera el `PropertyChanged` eventos pueden omitirse debido a conocimiento de cómo se realiza el enlace XAML.
- Generar siempre un `PropertyChanged` eventos para cualquier calculan las propiedades cuyos valores se utilizan por otras propiedades en la vista de modelo o el modelo.
- Generar siempre la `PropertyChanged` eventos al final del método que hace que una propiedad cambia, o cuando se conoce el objeto esté en un estado seguro. Provoca el evento interrumpe la operación mediante la invocación de forma sincrónica los controladores del evento. Si esto ocurre en el medio de una operación, que podría exponer el objeto a funciones de devolución de llamada cuando se encuentra en un estado parcialmente actualizado no seguro. Además, es posible que los cambios en cascada que va a desencadenar `PropertyChanged` eventos. Cambios en cascada normalmente necesitan actualizaciones que se complete antes de que es seguro ejecutar el cambio en cascada.
- No generar nunca una `PropertyChanged` evento si la propiedad no cambia. Esto significa que deben comparar los valores antiguos y nuevos antes de generar el `PropertyChanged` eventos.
- No elevar nunca el `PropertyChanged` eventos durante el constructor de un modelo vista si se está inicializando una propiedad. Los controles enlazados a datos en la vista no habrá suscrito para recibir notificaciones de cambio en este momento.
- No elevar nunca más de uno `PropertyChanged` eventos con el mismo argumento de nombre de propiedad dentro de una sola invocación sincrónica de un método público de una clase. Por ejemplo, dada una `NumberOfItems` propiedad cuya memoria auxiliar es la `_numberOfItems` campo, si un método incrementa `_numberOfItems` cincuenta veces durante la ejecución de un bucle, debería solo desencadenar la notificación de cambio de propiedad en el `NumberOfItems` una vez, propiedad una vez completado todo el trabajo. Para los métodos asincrónicos, elevar el `PropertyChanged` eventos para un nombre de la propiedad especificada en cada segmento sincrónica de una cadena de continuación asincrónica.

La aplicación móvil de eShopOnContainers utiliza la `ExtendedBindableObject` clase para proporcionar notificaciones de cambio, que se muestra en el ejemplo de código siguiente:

```
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)
    {
        ...
    }
}
```

Del Xamarin.Form `BindableObject` la clase implementa la `INotifyPropertyChanged` interfaz y proporciona un `OnPropertyChanged` método. El `ExtendedBindableObject` clase proporciona el `RaisePropertyChanged` método que se invoca la propiedad notificación de cambios y al hacerlo, usa la funcionalidad proporcionada por el

`BindableObject` clase.

Cada clase de modelo de vista en la aplicación móvil de eShopOnContainers se deriva de la `ViewModelBase` (clase), que a su vez se deriva de la `ExtendedBindableObject` clase. Por lo tanto, cada clase de modelo de vista usa la `RaisePropertyChanged` método en el `ExtendedBindableObject` clase para proporcionar una notificación de cambio de propiedad. En el ejemplo de código siguiente se muestra cómo la aplicación móvil de eShopOnContainers invoca la notificación de cambio de propiedad mediante el uso de una expresión lambda:

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

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

Interacción de la interfaz de usuario mediante los comandos y comportamientos

En aplicaciones móviles, las acciones se invocan normalmente 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, responsable de aplicar la acción está relacionado con el modelo de vista, y se debe evitar colocar código en el código subyacente.

Comandos proporcionan una manera cómoda para representar las acciones que se pueden enlazar a controles en la interfaz de usuario. Se encapsula el código que implementa la acción y ayudar a mantenerlo separa de su representación visual en la vista. Xamarin.Forms 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 controles mediante declaración se conecten a un comando. Sin embargo, los comportamientos pueden usarse para invocar una acción que está asociado con un intervalo de los eventos generados por un control. Por lo tanto, los comportamientos abordan muchos de los mismos escenarios que los controles de comando habilitado, al tiempo que proporciona un mayor grado de flexibilidad y control. Además, los comportamientos también pueden utilizarse para asociar los objetos de comando o métodos a los controles que no se diseñaron específicamente para interactuar con los comandos.

Implementación de comandos

Los modelos de vista normalmente exponen propiedades de comando, para el enlace de la vista, que son instancias de objetos que implementan la `ICommand` interfaz. Proporcione un número de controles de Xamarin.Forms un `Command` propiedad, que puede ser datos enlazados a un `ICommand` objeto proporcionado por el modelo de vista. El `ICommand` interfaz define un `Execute` método, que encapsula la operación en Sí, un `CanExecute` método, que indica si se puede invocar el comando y un `CanExecuteChanged` evento que tiene lugar cuando producen cambios que afectan a si debe ejecutar el comando. El `Command` y `Command<T>` implementan clases, que proporciona Xamarin.Forms, el `ICommand` interfaz, donde `T` es el tipo de los argumentos `Execute` y

`CanExecute`.

Dentro de un modelo de vista, debe ser un objeto de tipo `Command` o `Command<T>` para cada propiedad pública en el modelo de vista del tipo `ICommand`. El `Command` o `Command<T>` constructor requiere un `Action` objeto de devolución de llamada que se llama cuando el `ICommand.Execute` se invoca el método. El `CanExecute` método es un parámetro de constructor opcional y es un `Func` que devuelve un `bool`.

El siguiente código muestra cómo un `Command` se construye la instancia, que representa un comando register, especificando un delegado para el `Register` ver el método de modelo:

```
public ICommand RegisterCommand => new Command(Register);
```

El comando se expone a la vista a través de una propiedad que devuelve una referencia a un `ICommand`. Cuando el `Execute` se llama al método en el `Command` objeto, simplemente reenvía la llamada al método en el modelo de vista a través del delegado que se especificó en el `Command` constructor.

Un método asincrónico se puede invocar un comando mediante el uso de la `async` y `await` palabras clave al especificar el comando `Execute` delegar. Esto indica que la devolución de llamada es un `Task` y se debe esperar. Por ejemplo, el siguiente código muestra cómo un `Command` instancia, que representa un comando de inicio de sesión, se construye especificando un delegado para el `SignInAsync` ver el método de modelo:

```
public ICommand SignInCommand => new Command(async () => await SignInAsync());
```

Se pueden pasar parámetros a la `Execute` y `CanExecute` acciones mediante la `Command<T>` clase para crear instancias del comando. Por ejemplo, el siguiente código muestra cómo un `Command<T>` instancia se utiliza para indicar que el `NavigateAsync` método requerirá un argumento de tipo `string`:

```
public ICommand NavigateCommand => new Command<string>(NavigateAsync);
```

Tanto en el `Command` y `Command<T>` clases, el delegado para el `CanExecute` método en cada constructor es opcional. Si no se especifica un delegado, el `Command` devolverá `true` para `CanExecute`. Sin embargo, el modelo de vista puede indicar un cambio en el comando `CanExecute` estado mediante una llamada a la `ChangeCanExecute` método en el `Command` objeto. Esto hace que el `CanExecuteChanged` evento. Los controles en la interfaz de usuario que están enlazados al comando, a continuación, actualizará su estado habilitado para reflejar la disponibilidad de los comandos enlazados a datos.

Invocar comandos desde una vista

El siguiente ejemplo de código muestra cómo un `Grid` en el `LoginView` enlaza a la `RegisterCommand` en el `LoginViewModel` clase mediante el uso de un `TapGestureRecognizer` instancia:

```
<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, opcionalmente, definir mediante el `CommandParameter` propiedad. El tipo del argumento esperado se especifica en el `Execute` y `CanExecute` métodos de destino. El `TapGestureRecognizer` invocará automáticamente el comando de destino cuando el usuario interactúa con el control adjunto. El parámetro de comando, si se proporciona, se pasarán como argumento para el comando `Execute` delegar.

Implementación de comportamientos

Comportamientos permiten la funcionalidad que se agregará a los controles de interfaz de usuario sin tener que subclase ellos. En su lugar, la funcionalidad se implementa en una clase de comportamiento y se adjunta al control como si fuera parte del propio control. Los comportamientos le permiten implementar el código que normalmente tendría que escribir como código subyacente, ya que interactúa directamente con la API de control, de tal manera que puede ser concisa adjunta al control y empaqueta para su reutilización en más de una vista o la aplicación. En el contexto de MVVM, los comportamientos son un enfoque útil para conectar los controles a los comandos.

Un comportamiento que se adjunta a un control a través de las propiedades adjuntas se conoce como un *adjunta comportamiento*. El comportamiento, a continuación, puede usar la API expuesta del elemento al que se adjunta para agregar funcionalidad a ese control u otros controles, en el árbol visual de la vista. La aplicación móvil de eShopOnContainers contiene el `LineColorBehavior` (clase), que es un comportamiento asociado. Para obtener más información acerca de este comportamiento, consulte mostrar errores de validación.

Un comportamiento de Xamarin.Forms es una clase que deriva el `Behavior` o `Behavior<T>` (clase), donde `T` es el tipo del control al que debe aplicarse el comportamiento. Estas clases proporcionan `OnAttachedTo` y `OnDetachingFrom` métodos, que se deben invalidar para proporcionar la lógica que se ejecutará cuando el comportamiento se asocian y desasocian de controles.

En la aplicación móvil de eShopOnContainers, el `BindableBehavior<T>` clase se deriva de la `Behavior<T>` clase. El propósito de la `BindableBehavior<T>` clase es proporcionar una clase base para los comportamientos de Xamarin.Forms que requieren la `BindingContext` del comportamiento para establecerse en el control adjunto.

El `BindableBehavior<T>` clase proporciona un reemplazable `OnAttachedTo` método que establece el `BindingContext` del comportamiento y un reemplazable `OnDetachingFrom` método que limpia la `BindingContext`. Además, la clase almacena una referencia al control adjunto en el `AssociatedObject` propiedad.

La aplicación móvil de eShopOnContainers incluye un `EventToCommandBehavior` (clase), que se ejecuta un comando como respuesta a la que se produzca un evento. Esta clase se deriva de la `BindableBehavior<T>` clase para que el comportamiento se puede enlazar a y ejecutar un `ICommand` especificado por un `Command` propiedad cuando se consume el comportamiento. En el ejemplo de código siguiente se muestra la clase `EventToCommandBehavior`:

```
public class EventToCommandBehavior : BindableBehavior<View>
{
    ...
    protected override void OnAttachedTo(View visualElement)
    {
        base.OnAttachedTo(visualElement);

        var events = AssociatedObject.GetType().GetRuntimeEvents().ToArray();
        if (events.Any())
        {
            _eventInfo = events.FirstOrDefault(e => e.Name == EventName);
            if (_eventInfo == null)
                throw new ArgumentException(string.Format(
                        "EventToCommand: Can't find any event named '{0}' on attached type",
                        EventName));

            AddEventHandler(_eventInfo, AssociatedObject, OnFired);
        }
    }

    protected override void OnDetachingFrom(View view)
    {
        if (_handler != null)
            _eventInfo.RemoveEventHandler(AssociatedObject, _handler);

        base.OnDetachingFrom(view);
    }

    private void AddEventHandler(
            EventInfo eventInfo, object item, Action<object, EventArgs> action)
    {
        ...
    }

    private void OnFired(object sender, EventArgs eventArgs)
    {
        ...
    }
}
```

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

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

Invocar comportamientos de una vista

El `EventToCommandBehavior` es especialmente útil para asociar un comando a un control que no es compatible con los comandos. Por ejemplo, el `ProfileView` usa el `EventToCommandBehavior` para ejecutar el `OrderDetailCommand` cuando el `ItemTapped` evento se desencadena en el `ListView` que muestra los pedidos del usuario, como se muestra en el código siguiente:

```
<ListView>
    <ListView.Behaviors>
        <behaviors:EventToCommandBehavior
            EventName="ItemTapped"
            Command="{Binding OrderDetailCommand}"
            EventArgsConverter="{StaticResource ItemTappedEventArgsConverter}" />
    </ListView.Behaviors>
    ...
</ListView>
```

En tiempo de ejecución, el `EventToCommandBehavior` responderá a la interacción con el `ListView` . Cuando se selecciona un elemento en el `ListView` , `ItemTapped` se desencadenará el evento, que se ejecutará el `OrderDetailCommand` en el `ProfileViewModel` . De forma predeterminada, los argumentos de evento para el evento se pasan al comando. Estos datos se convierten como se pasa entre el origen y destino por el convertidor especificado en el `EventArgsConverter` propiedad, que devuelve el `Item` de la `ListView` desde el `ItemTappedEventArgs` . Por lo tanto, cuando el `OrderDetailCommand` se ejecuta, seleccionado `Order` se pasa como parámetro a la acción registrada.

Para obtener más información acerca de los comportamientos, consulte comportamientos.

Resumen

El patrón Model-View-ViewModel (MVVM) ayuda a separar la lógica de negocios y la presentación de una aplicación desde su interfaz de usuario (UI). Mantener una separación clara entre la interfaz de usuario y la lógica de aplicación ayuda a abordar numerosos problemas de desarrollo y hacer más fácil probar una aplicación, mantener y evolucionar. Se pueden mejorar enormemente las oportunidades de reutilización de código y permite a los desarrolladores y diseñadores de interfaz de usuario más colaboran fácilmente al desarrollar sus respectivos partes de una aplicación.

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

Vínculos relacionados

- Descargar libro electrónico (PDF de 2Mb)
- eShopOnContainers (GitHub) (ejemplo)

Inserción de dependencias

09/06/2018 • 21 minutes to read • Edit Online

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

Al especificar dependencias como tipos de interfaz, inyección de dependencia permite la separación de los tipos concretos desde el código que dependa de estos tipos. Por lo general, usa un contenedor que contiene una lista de registros y las asignaciones entre tipos abstractos e interfaces y los tipos concretos que implementan o extienden estos tipos.

También hay otros tipos de inserción de dependencias, como *inyección de establecedor de propiedad*, y *por inyección de código de llamada de método*, pero que se ven menos frecuente. Por lo tanto, en este capítulo se centrará únicamente en realizar la inyección de constructor con un contenedor de inyección de dependencia.

Introducción a la inyección de dependencia

Inserción de dependencias es una versión especializada del patrón de inversión de Control (IoC), donde la preocupación de que se invierte es el proceso de obtención de la dependencia necesaria. Con la inserción de dependencias, otra clase es responsable de insertar las dependencias en un objeto en tiempo de ejecución. El siguiente ejemplo de código muestra cómo el `ProfileViewModel` clase está estructurada al usar la inserción de dependencias:

```
public class ProfileViewModel : ViewModelBase
{
    private IOrderService _orderService;

    public ProfileViewModel(IOrderService orderService)
    {
        _orderService = orderService;
    }
    ...
}
```

El `ProfileViewModel` constructor recibe un `IOrderService` instancia como un argumento, insertado por otra clase. La única dependencia en el `ProfileViewModel` clase está en el tipo de interfaz. Por lo tanto, la `ProfileViewModel` clase no tiene ningún conocimiento de la clase que es responsable de crear instancias de la `IOrderService` objeto. La clase que es responsable de crear instancias de la `IOrderService` objeto e insertarlos en la `ProfileViewModel` de clases, se conoce como el *contenedor inyección de dependencia*.

Contenedores de inyección de dependencia reducen el acoplamiento entre objetos por lo que proporciona una funcionalidad 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 requiere el objeto en él. Si aún no se han creado esas dependencias, el contenedor crea y sus dependencias resuelve en primer lugar.

> **NOTE**
>
> Inyección de dependencia también puede implementarse manualmente mediante generadores. Sin embargo, mediante un contenedor proporciona funcionalidades adicionales como la administración de la duración y el registro a través de análisis de ensamblado.

Hay varias ventajas con respecto al uso de un contenedor de inyección de dependencia:

- Un contenedor elimina la necesidad de una clase encontrar sus dependencias y administrar sus duraciones.
- Un contenedor permite la asignación de dependencias implementadas sin que afecte a la clase.
- Un contenedor facilita la capacidad de prueba al permitir que las dependencias ser simulada.
- Un contenedor aumenta la facilidad de mantenimiento permitiendo nuevas clases agregar fácilmente a la aplicación.

En el contexto de una aplicación de Xamarin.Forms que usa MVVM, un contenedor de inyección de dependencia normalmente se usará para registrar y resolver ver modelos y de registro de los servicios y a insertar en modelos de vista.

Existen muchos contenedores de inyección de dependencia, con la aplicación móvil eShopOnContainers con Autofac para administrar la creación de instancias del modelo de vista y las clases en la aplicación del servicio. Autofac facilita la creación de aplicaciones de acoplamiento flexible y proporciona todas las características que se suelen encontrar en contenedores de inyección de dependencia, incluidos los métodos para registrar asignaciones de tipos e instancias de objeto, resuelva los objetos, administración la duración de los objetos e insertar objetos dependientes en constructores de objetos que resuelve. Para obtener más información sobre Autofac, consulte Autofac en readthedocs.io.

En Autofac, la `IContainer` interfaz proporciona el contenedor de inyección de dependencia. Figura 3-1 se muestran las dependencias cuando se usa este contenedor, que se crea una instancia de un `IOrderService` objeto y lo inserta en la `ProfileViewModel` clase.

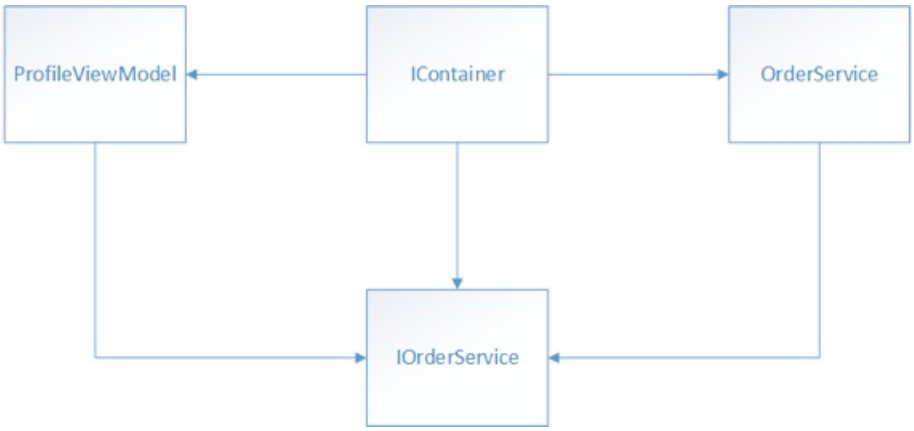

Figura 3-1: dependencias cuando se usa la inyección de dependencia

En tiempo de ejecución, el contenedor debe saber qué implementación de la `IOrderService` interfaz debe crear una instancia, antes de que puede crear una instancia de un `ProfileViewModel` objeto. Esto implica:

- El contenedor de la decisión sobre cómo crear una instancia de un objeto que implementa el `IOrderService` interfaz. Esto se conoce como *registro*.
- El contenedor de crear instancias del objeto que implementa el `IOrderService` interfaz y la `ProfileViewModel` objeto. Esto se conoce como *resolución*.

Finalmente, la aplicación finalizará utilizando la `ProfileViewModel` objeto y estará disponibles para la recolección. En este punto, el recolector de elementos no utilizados debe desechar el `IOrderService` si otras clases no comparten la misma instancia de la instancia.

> **TIP**
> Escribir código independiente del contenedor. Intente escribir siempre código independiente del contenedor para desacoplar la aplicación desde el contenedor de dependencia concreta que se va a usar.

Registro

Antes de que las dependencias se pueden insertar en un objeto, los tipos de las dependencias en primer lugar deben registrarse con el contenedor. Registrar un tipo normalmente implica pasar el contenedor de una interfaz y un tipo concreto que implementa la interfaz.

Hay dos formas de registrar los tipos y objetos en el contenedor a través de código:

- Registrar un tipo o una asignación con el contenedor. Cuando sea necesario, el contenedor creará una instancia del tipo especificado.
- Registrar un objeto existente en el contenedor como singleton. Cuando sea necesario, el contenedor devolverá una referencia al objeto existente.

> **TIP**
> Contenedores de inyección de dependencia no son siempre adecuados. Inyección de dependencia introduce una complejidad adicional y requisitos que podrían no ser útil o adecuado para pequeñas aplicaciones. Si una clase no tiene ninguna dependencia o no es una dependencia de otros tipos, no podría no sentido para colocarla en el contenedor. Además, si una clase tiene un único conjunto de dependencias que son fundamentales para el tipo y no cambia nunca, no tendría sentido lo coloque en el contenedor.

El registro de tipos que requieren la inserción de dependencias debe realizarse en un único método en una aplicación, y este método se debe invocar al principio de ciclo de vida de la aplicación para asegurarse de que la aplicación está al tanto de las dependencias entre sus clases. En la aplicación móvil eShopOnContainers Esto se realiza mediante la `ViewModelLocator` de la clase, qué compilaciones la `IContainer` del objeto y es la única clase en la aplicación que contiene una referencia a ese objeto. En el ejemplo de código siguiente se muestra cómo se declara la aplicación móvil eShopOnContainers el `IContainer` objeto en el `ViewModelLocator` clase:

```
private static IContainer _container;
```

Tipos e instancias están registrados en el `RegisterDependencies` método en la `ViewModelLocator` clase. Esto se logra creando primero una `ContainerBuilder` instancia, que se muestra en el ejemplo de código siguiente:

```
var builder = new ContainerBuilder();
```

Tipos e instancias, a continuación, se registran con el `ContainerBuilder` objeto y en el ejemplo de código siguiente se muestra la forma más común de registro del tipo:

```
builder.RegisterType<RequestProvider>().As<IRequestProvider>();
```

El `RegisterType` método se muestra a continuación, asigna un tipo de interfaz a un tipo concreto. Indica el contenedor para crear instancias de un `RequestProvider` objeto cuando se crea una instancia de un objeto que requiere una inyección de una `IRequestProvider` a través de un constructor.

También se pueden registrar tipos concretos directamente sin una asignación de un tipo de interfaz, como se muestra en el ejemplo de código siguiente:

```
builder.RegisterType<ProfileViewModel>();
```

Cuando el `ProfileViewModel` tipo se resuelve, el contenedor insertan sus dependencias necesarias.

Autofac también permite el registro de instancia, donde el contenedor es responsable de mantener una referencia

a una instancia de un tipo singleton. Por ejemplo, en el ejemplo de código siguiente se muestra cómo la aplicación móvil eShopOnContainers registra el tipo concreto que se utilizará cuando un `ProfileViewModel` instancia requiere un `IOrderService` instancia:

```
builder.RegisterType<OrderService>().As<IOrderService>().SingleInstance();
```

El `RegisterType` método se muestra a continuación, asigna un tipo de interfaz a un tipo concreto. El `SingleInstance` método configura el registro para que todos los objetos dependientes reciba la misma instancia compartida. Por lo tanto, solo una `OrderService` instancia existirá en el contenedor, que es compartido por los objetos que requieren una inyección de una `IOrderService` a través de un constructor.

Registro de la instancia también pueden realizarse con el `RegisterInstance` método, que se muestra en el ejemplo de código siguiente:

```
builder.RegisterInstance(new OrderMockService()).As<IOrderService>();
```

El `RegisterInstance` crea un nuevo método que se muestra aquí `OrderMockService` instancia y lo registra con el contenedor. Por lo tanto, solo una `OrderMockService` instancia existe en el contenedor, que es compartido por los objetos que requieren una inyección de una `IOrderService` a través de un constructor.

Tras el registro de tipo y la instancia, la `IContainer` objeto debe compilarse, que se muestra en el ejemplo de código siguiente:

```
_container = builder.Build();
```

Invocar la `Build` método en el `ContainerBuilder` instancia genera un nuevo contenedor de inyección de dependencia que contiene los registros que se han realizado.

> **Sugerencia**: considere la posibilidad de un `IContainer` como inmutable. Si bien proporciona Autofac un `Update` método para actualizar los registros de un contenedor existente, llamar a este método debería evitarse siempre que sea posible. Existen riesgos para modificar un contenedor después de que se ha compilado, especialmente si se ha utilizado el contenedor. Para obtener más información, consulte considere la posibilidad de un contenedor como inmutables en readthedocs.io.

Resolución

Después de registra un tipo, puede resolver o insertado como una dependencia. Cuando se resuelve un tipo y el contenedor debe crear una nueva instancia, todas las dependencias inserta en la instancia.

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

1. Si el tipo no se ha registrado, el contenedor inicia una excepción.
2. Si el tipo se ha registrado como un singleton, el contenedor devuelve la instancia de singleton. Si se trata de la primera vez que se llame al tipo de, el contenedor crea si es necesario y mantiene una referencia a él.
3. Si el tipo no se ha registrado como un singleton, el contenedor devuelve una nueva instancia y no mantiene una referencia a él.

El siguiente ejemplo de código muestra cómo el `RequestProvider` se puede resolver el tipo que se registraron anteriormente con Autofac:

```
var requestProvider = _container.Resolve<IRequestProvider>();
```

En este ejemplo, se solicita Autofac para resolver el tipo concreto para la `IRequestProvider` tipo, junto con todas las dependencias. Normalmente, el `Resolve` método se llama cuando se necesita una instancia de un tipo específico. Para obtener información sobre cómo controlar la duración de objetos resueltos, consulte administrar la duración de objetos resolver.

En el ejemplo de código siguiente se muestra cómo la aplicación móvil eShopOnContainers crea una instancia de la vista tipos de modelo y sus dependencias:

```
var viewModel = _container.Resolve(viewModelType);
```

En este ejemplo, se solicita Autofac para resolver el tipo de modelo de vista para un modelo de vista solicitada y el contenedor también resolverá las dependencias. Al resolver el `ProfileViewModel` es de tipo, la dependencia para resolver un `IOrderService` objeto. Por lo tanto, primero crea Autofac una `OrderService` objeto y, a continuación, se pasa al constructor de la `ProfileViewModel` clase. Para obtener más información acerca de cómo la aplicación móvil eShopOnContainers construye ver modelos y las asocia a vistas, vea creación automática de un modelo de vista con un localizador de modelo de vista.

> **NOTE**
> Registrar y resolver los tipos con un contenedor afectará al rendimiento debido a uso del contenedor de la reflexión para crear cada tipo, especialmente si se está regenerando dependencias para cada navegación de una página en la aplicación. Si hay muchos o profundas dependencias, puede aumentar significativamente el costo de la creación.

Administrar la vigencia de objetos resueltos

Después de registrar un tipo, el comportamiento predeterminado para Autofac es crear una nueva instancia del tipo registrado cada vez que el tipo se resuelve, o cuando el mecanismo de dependencia inserta instancias en otras clases. En este escenario, el contenedor no contiene una referencia al objeto resuelto. Sin embargo, al registrar una instancia, el comportamiento predeterminado para Autofac es administrar la vigencia del objeto como un singleton. Por lo tanto, la instancia permanece en el ámbito mientras el contenedor está en ámbito y se elimina cuando el contenedor se sale del ámbito y se recolectan, o cuando el código elimina explícitamente el contenedor.

Un ámbito de la instancia de Autofac puede utilizarse para especificar el comportamiento de singleton para un objeto que Autofac crea a partir de un tipo registrado. Ámbitos de la instancia de Autofac administran la duración de los objetos crea una instancia del contenedor. El ámbito de la instancia predeterminada de la `RegisterType` método es el `InstancePerDependency` ámbito. Sin embargo, el `SingleInstance` ámbito se puede utilizar con la `RegisterType` método, para que el contenedor se crea o se devuelve una instancia de un tipo singleton al llamar a la `Resolve` método. En el ejemplo de código siguiente se muestra cómo se indica Autofac para crear una instancia de singleton de la `NavigationService` clase:

```
builder.RegisterType<NavigationService>().As<INavigationService>().SingleInstance();
```

La primera vez que la `INavigationService` interfaz se resuelve, el contenedor crea un nuevo `NavigationService` de objetos y mantiene una referencia a él. En las siguientes resoluciones de la `INavigationService` interfaz, el contenedor devuelve una referencia a la `NavigationService` objeto que se ha creado previamente.

> **NOTE**
> El ámbito de SingleInstance elimina los objetos creados cuando se elimina el contenedor.

Autofac incluye ámbitos de instancia adicional. Para obtener más información, consulte ámbito de la instancia en readthedocs.io.

Resumen

Inyección de dependencia permite la separación de tipos concretos desde el código que dependa de estos tipos. Normalmente usa un contenedor que contiene una lista de registros y las asignaciones entre tipos abstractos e interfaces y los tipos concretos que implementan o extienden estos tipos.

Autofac facilita la creación de aplicaciones de acoplamiento flexible y proporciona todas las características que se suelen encontrar en contenedores de inyección de dependencia, incluidos los métodos para registrar asignaciones de tipos e instancias de objeto, resuelva los objetos, administración la duración de los objetos e insertar objetos dependientes en constructores de objetos que se resuelve.

Vínculos relacionados

- Descargar libros electrónicos (PDF de 2Mb)
- eShopOnContainers (GitHub) (ejemplo)

Comunicarse libremente entre los componentes acoplados

13/07/2018 • 12 minutes to read • Edit Online

Publish-suscribirse patrón es un patrón de mensajería en la que los editores envían mensajes sin tener conocimiento de los destinatarios, conocido como suscriptores. De forma similar, los suscriptores escuchan mensajes concretos, sin tener conocimiento de los publicadores.

Los eventos de .NET implementan la publicación-patrón de suscripción y es más simple y sencillo enfoque para un nivel de comunicación entre componentes si un acoplamiento no es necesario, como un control y la página que lo contiene. Sin embargo, las duraciones de publicador y suscriptor están acopladas por referencias a objetos entre sí y el tipo de suscriptor debe tener una referencia al tipo de publicador. Esto puede crear la forma en memoria problemas de administración, especialmente cuando hay objetos de corta duración que se suscriben a un evento de un objeto estático o de larga duración. Si no se elimina el controlador de eventos, el suscriptor se mantendrá conectado por la referencia a él en el publicador, y esto se impedir o retrasar la recolección de elementos del suscriptor.

Introducción a MessagingCenter

Xamarin.Forms `MessagingCenter` clase implementa la publicación-patrón, que permite la comunicación basada en mensajes entre los componentes que no son convenientes para vincular mediante referencias de objeto y el tipo de suscripción. Este mecanismo permite a los publicadores y suscriptores transmitir sin tener una referencia entre sí, lo que ayuda a reducir las dependencias entre componentes, mientras que permite que los componentes que se desarrollan y prueban de forma independiente.

El `MessagingCenter` clase proporciona multidifusión funcionalidad de publicación y suscripción. Esto significa que puede haber varios publicadores que publican un único mensaje y puede haber varios suscriptores a la escucha para el mismo mensaje. Figura 4-1 se ilustra esta relación:

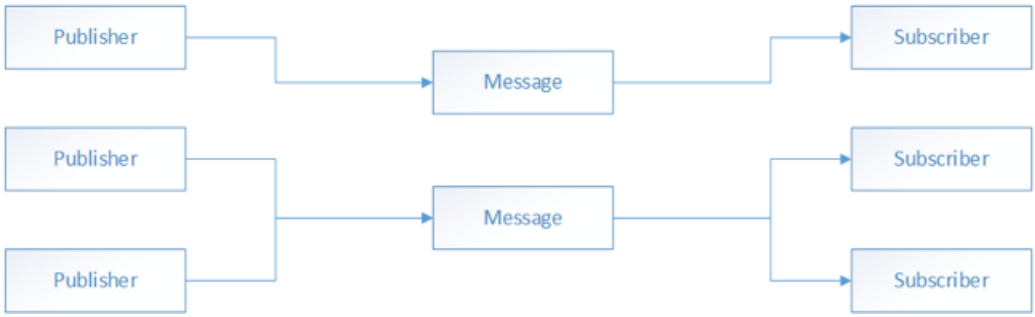

Figura 4-1: multidifusión funcionalidad de publicación y suscripción

Los editores envían mensajes mediante el `MessagingCenter.Send` método, mientras que los suscriptores realizar escuchas de mensajes mediante el `MessagingCenter.Subscribe` método. Además, los suscriptores pueden también cancelar la suscripción a suscripciones de mensajes, si es necesario, con el `MessagingCenter.Unsubscribe` método.

Internamente, el `MessagingCenter` clase usa referencias débiles. Esto significa que no mantendrá objetos activa y les permitirá sea recolectado. Por lo tanto, sólo debería ser necesario cancelar la suscripción a un mensaje cuando una clase ya no desea recibir el mensaje.

La aplicación móvil de eShopOnContainers utiliza la `MessagingCenter` componentes de clase para la comunicación entre débilmente acoplados. La aplicación define tres mensajes:

- El `AddProduct` mensaje publicado por el `CatalogViewModel` clase cuando se agrega un elemento a la cesta de compra. En cambio, el `BasketViewModel` clase se suscribe al mensaje e incrementa el número de elementos de la cesta de compra en la respuesta. Además, el `BasketViewModel` clase también cancela la suscripción de este mensaje.
- El `Filter` mensaje publicado por el `CatalogViewModel` clase cuando el usuario aplica un filtro de marca o el tipo a los elementos mostrados en el catálogo. En cambio, el `CatalogView` clase se suscribe al mensaje y actualiza la interfaz de usuario para que se muestren solo los elementos que coinciden con los criterios de filtro.
- El `ChangeTab` mensaje publicado por el `MainViewModel` clase cuando la `CheckoutViewModel` navega a la `MainViewModel` siguiendo la creación correcta y el envío de un nuevo pedido. En cambio, el `MainView` clase suscribe al mensaje y las actualizaciones de la interfaz de usuario hasta que el **mi perfil** pestaña está activa, para mostrar los pedidos del usuario.

> **NOTE**
> Mientras el `MessagingCenter` clase permite la comunicación entre las clases de acoplamiento flexible, no ofrece la solución de arquitectura solo a este problema. Por ejemplo, la comunicación entre un modelo de vista y una vista también puede lograrse mediante el motor de enlace y a través de las notificaciones de cambio de propiedad. Además, también puede lograrse la comunicación entre los dos modelos de vista pasando los datos durante la navegación.

En la aplicación móvil de eShopOnContainers, `MessagingCenter` se usa para actualizar en la interfaz de usuario en respuesta a una acción que se producen en otra clase. Por lo tanto, los mensajes se publican en el subproceso de interfaz de usuario, con los suscriptores que reciben el mensaje en el mismo subproceso.

> **TIP**
> Calcular las referencias al subproceso de interfaz de usuario cuando la interfaz de usuario de realizar actualizaciones. Si es necesario actualizar la interfaz de usuario un mensaje que se envía desde un subproceso en segundo plano, procesar el mensaje en el subproceso de interfaz de usuario en el suscriptor mediante la invocación del `Device.BeginInvokeOnMainThread` método.

Para obtener más información acerca de `MessagingCenter` , consulte MessagingCenter.

Definir un mensaje

`MessagingCenter` los mensajes son cadenas que se usan para identificar los mensajes. El ejemplo de código siguiente muestra los mensajes definidos dentro de la aplicación móvil de eShopOnContainers:

```
public class MessengerKeys
{
    // Add product to basket
    public const string AddProduct = "AddProduct";

    // Filter
    public const string Filter = "Filter";

    // Change selected Tab programmatically
    public const string ChangeTab = "ChangeTab";
}
```

En este ejemplo, los mensajes se definen mediante las constantes. La ventaja de este enfoque es que proporciona seguridad de tipos de tiempo de compilación y la compatibilidad de refactorización.

Publicar un mensaje

Editores de notificar a los suscriptores de un mensaje con uno de los `MessagingCenter.Send` sobrecargas. En el ejemplo de código siguiente se muestra cómo publicar el `AddProduct` mensaje:

```
MessagingCenter.Send(this, MessengerKeys.AddProduct, catalogItem);
```

En este ejemplo, el `Send` método especifica tres argumentos:

- El primer argumento especifica la clase del remitente. La clase del remitente debe especificarse ningún suscriptor que deseen recibir el mensaje.
- El segundo argumento especifica el mensaje.
- El tercer argumento especifica los datos de carga para enviarse al suscriptor. En este caso los datos de carga están un `CatalogItem` instancia.

El `Send` método va a publicar el mensaje y sus datos de carga, con un enfoque de fire and forget. Por lo tanto, se envía el mensaje incluso si no hay ningún suscriptor registrado para recibir el mensaje. En esta situación, se omite el mensaje enviado.

> **NOTE**
> El `MessagingCenter.Send` método puede usar los parámetros genéricos para controlar cómo se entregan los mensajes. Por lo tanto, diferentes suscriptores pueden recibir varios mensajes que comparten una identidad de mensaje pero enviar carga útil diferentes tipos de datos.

Suscribirse a un mensaje

Los suscriptores pueden registrar para recibir un mensaje mediante uno de los `MessagingCenter.Subscribe` sobrecargas. En el ejemplo de código siguiente se muestra cómo se suscribe a la aplicación móvil de eShopOnContainers y procesa, el `AddProduct` mensaje:

```
MessagingCenter.Subscribe<CatalogViewModel, CatalogItem>(
    this, MessageKeys.AddProduct, async (sender, arg) =>
{
    BadgeCount++;

    await AddCatalogItemAsync(arg);
});
```

En este ejemplo, el `Subscribe` método se suscribe a la `AddProduct` del mensaje y un delegado de devolución de llamada se ejecuta en respuesta a recibir el mensaje. Este delegado de devolución de llamada, especificado como una expresión lambda, ejecuta el código que actualiza la interfaz de usuario.

> **TIP**
> Considere el uso de datos de carga inmutable. No intente modificar los datos de carga desde dentro de un delegado de devolución de llamada porque varios subprocesos podrían tener acceso a los datos recibidos al mismo tiempo. En este escenario, los datos de carga deben ser inmutables para evitar errores de simultaneidad.

Un suscriptor no es posible que deba controlar todas las instancias de un mensaje publicado, y esto puede controlarse mediante los argumentos de tipo genérico que se especifican en el `Subscribe` método. En este ejemplo, el suscriptor sólo recibirá `AddProduct` los mensajes que se envían desde el `CatalogViewModel` (clase), cuyos datos de carga están un `CatalogItem` instancia.

Cancelación de un mensaje de la suscripción

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

```
MessagingCenter.Unsubscribe<CatalogViewModel, CatalogItem>(this, MessengerKeys.AddProduct);
```

En este ejemplo, el `Unsubscribe` sintaxis de método refleja los argumentos de tipo especificados al suscribirse para recibir el `AddProduct` mensaje.

Resumen

Xamarin.Forms `MessagingCenter` clase implementa la publicación-patrón, que permite la comunicación basada en mensajes entre los componentes que no son convenientes para vincular mediante referencias de objeto y el tipo de suscripción. Este mecanismo permite a los publicadores y suscriptores transmitir sin tener una referencia entre sí, lo que ayuda a reducir las dependencias entre componentes, mientras que permite que los componentes que se desarrollan y prueban de forma independiente.

Vínculos relacionados

- Descargar libro electrónico (PDF de 2Mb)
- eShopOnContainers (GitHub) (ejemplo)

Navegación de la aplicación de empresa

13/07/2018 • 24 minutes to read • Edit Online

Xamarin.Forms incluye compatibilidad con la navegación de página, que normalmente da como resultado de la interacción del usuario con la interfaz de usuario o de la aplicación como resultado de los cambios de estado controlado por la lógica interna. Sin embargo, navegación puede ser difícil de implementar en las aplicaciones que usan el patrón Model-View-ViewModel (MVVM), tal como se deben cumplir los siguientes desafíos:

- Cómo identificar la vista para navegar, mediante un enfoque que no introduce acoplamiento estrecho y las dependencias entre las vistas.
- Describe cómo coordinar el proceso por el que se crea una instancia e inicializa la vista para navegar. Al usar MVVM, la vista y el modelo de vista deben crear una instancia y asociados entre sí a través del contexto de enlace de la vista. Cuando una aplicación usa un contenedor de inserción de dependencia, la creación de instancias de las vistas y los modelos de vista podría requerir un mecanismo específico de construcción.
- Si se debe realizar la navegación a primera vista, o ver navegación model first. Con la navegación de la vista en primer lugar, para navegar a la página de hace referencia al nombre del tipo de vista. Durante la navegación, se crea la vista especificada, junto con su modelo de vista correspondiente y otros servicios dependientes. Un enfoque alternativo es usar el panel de navegación view model first, donde la página para ir a hace referencia al nombre del tipo de modelo de vista.
- ¿Cómo a claramente separar el comportamiento de navegación de la aplicación a través de las vistas y los modelos de vista. El patrón MVVM proporciona una separación entre la interfaz de usuario de la aplicación y su presentación y lógica de negocios. Sin embargo, el comportamiento de navegación de una aplicación a menudo abarcará los elementos de interfaz de usuario y presentaciones de la aplicación. El usuario a menudo iniciará la exploración de una vista, y se reemplazará la vista como resultado de la barra de navegación. Sin embargo, navegación a menudo también podría necesitar que se inició o coordinadas desde dentro del modelo de vista.
- Cómo pasar parámetros durante la navegación por motivos de inicialización. Por ejemplo, si el usuario navega a una vista para actualizar los detalles de pedido, los datos del pedido tendrá que pasarse a la vista para que pueda mostrar los datos correctos.
- Cómo navegación coordinarse, para asegurarse de que determinadas reglas empresariales son obedecer. Por ejemplo, es posible que se le pida a los usuarios antes de salir de una vista para que se pueden corregir los datos no válidos o se le pida que envíe o descarte los cambios de datos que se realizaron en la vista.

Este capítulo aborda estos retos presentando un `NavigationService` clase que se usa para realizar la navegación de páginas de model first de vista.

> **NOTE**
>
> El `NavigationService` utilizado por la aplicación está diseñada únicamente para realizar la navegación jerárquica entre instancias de ContentPage. Uso del servicio para navegar entre otros tipos de páginas podría producir un comportamiento inesperado.

Navegar entre páginas

Lógica de navegación puede residir en el código subyacente de una vista o en un datos enlaza el modelo de vista. Aunque colocar lógica de navegación en una vista puede ser el enfoque más sencillo, no es probar fácilmente a través de las pruebas unitarias. Colocar en la vista lógica de navegación clases de modelo significa que la lógica puede realizarse a través de las pruebas unitarias. Además, el modelo de vista, a continuación, puede implementar

la lógica para controlar el desplazamiento para garantizar que se apliquen determinadas reglas empresariales. Por ejemplo, una aplicación no es posible que permite al usuario a salir de una página sin primero lo que garantiza que los datos introducidos son válidos.

Un `NavigationService` clase normalmente se invoca desde modelos de vista, para promover la capacidad de prueba. Sin embargo, navegar a vistas de los modelos de vista requeriría los modelos de vista a las vistas de referencia y especialmente las vistas que no está asociado, lo que no se recomienda el modelo de vista activa. Por lo tanto, el `NavigationService` presentado aquí se especifica el tipo de modelo de vista como destino para navegar a.

La aplicación móvil de eShopOnContainers utiliza la `NavigationService` clase para proporcionar navegación model first de vista. Esta clase implementa la `INavigationService` interfaz, que se muestra en el ejemplo de código siguiente:

```
public interface INavigationService
{
    ViewModelBase PreviousPageViewModel { get; }
    Task InitializeAsync();
    Task NavigateToAsync<TViewModel>() where TViewModel : ViewModelBase;
    Task NavigateToAsync<TViewModel>(object parameter) where TViewModel : ViewModelBase;
    Task RemoveLastFromBackStackAsync();
    Task RemoveBackStackAsync();
}
```

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

MÉTODO	PROPÓSITO
`InitializeAsync`	Realiza la navegación a uno de dos páginas cuando se inicia la aplicación.
`NavigateToAsync`	Realiza la navegación jerárquica en una página especificada.
`NavigateToAsync(parameter)`	Realiza la navegación jerárquica en una página especificada, pasando un parámetro.
`RemoveLastFromBackStackAsync`	Quita la página anterior de la pila de navegación.
`RemoveBackStackAsync`	Quita todas las páginas anteriores de la pila de navegación.

Además, el `INavigationService` interfaz especifica que una clase de implementación debe proporcionar un `PreviousPageViewModel` propiedad. Esta propiedad devuelve el tipo de modelo de vista asociado a la página anterior en la pila de navegación.

> **NOTE**
> Un `INavigationService` interfaz sería normalmente también especificar una `GoBackAsync` método, que se usa para devolver mediante programación a la página anterior en la pila de navegación. Sin embargo, este método es que faltan desde la aplicación móvil de eShopOnContainers porque no es necesario.

Creación de la instancia NavigationService

El `NavigationService` clase que implementa el `INavigationService` de la interfaz, se registra como un singleton con el contenedor de inserción de dependencia de Autofac, como se muestra en el ejemplo de código siguiente:

```
builder.RegisterType<NavigationService>().As<INavigationService>().SingleInstance();
```

El `INavigationService` interfaz se resuelve en el `ViewModelBase` constructor de clase, como se muestra en el ejemplo de código siguiente:

```
NavigationService = ViewModelLocator.Resolve<INavigationService>();
```

Esto devuelve una referencia a la `NavigationService` objeto que se almacena en el contenedor de inserción de dependencia de Autofac, que es creado por el `InitNavigation` método en el `App` clase. Para obtener más información, consulte navegar cuando la aplicación se inicia.

El `ViewModelBase` clase almacena el `NavigationService` de instancia en un `NavigationService` propiedad de tipo `INavigationService`. Por lo tanto, todos ver las clases de modelo, que se derivan de la `ViewModelBase` de clases, puede usar el `NavigationService` propiedad para tener acceso a los métodos especificados por el `INavigationService` interfaz. Esto evita la sobrecarga de insertar el `NavigationService` objeto desde el contenedor de inserción de dependencias de Autofac en cada clase de modelo de vista.

Controlar las solicitudes de navegación

Xamarin.Forms proporciona el `NavigationPage` (clase), que implementa una experiencia de navegación jerárquica en el que el usuario es capaz de navegar a través de páginas hacia delante y hacia atrás, según sea necesario. Para obtener más información sobre la navegación jerárquica, consulte Hierarchical Navigation (Navegación jerárquica).

En lugar de utilizar el `NavigationPage` directamente, la clase el ajusta de la aplicación eShopOnContainers el `NavigationPage` clase en el `CustomNavigationView` clase, como se muestra en el ejemplo de código siguiente:

```
public partial class CustomNavigationView : NavigationPage
{
    public CustomNavigationView() : base()
    {
        InitializeComponent();
    }

    public CustomNavigationView(Page root) : base(root)
    {
        InitializeComponent();
    }
}
```

El propósito de este ajuste es para facilitar la aplicación de estilos del `NavigationPage` instancia dentro del archivo XAML para la clase.

Navegación se realiza dentro de las clases de modelo de vista mediante la invocación de uno de los `NavigateToAsync` métodos, que especifica el tipo de modelo de vista de la página que se navega a, como se muestra en el ejemplo de código siguiente:

```
await NavigationService.NavigateToAsync<MainViewModel>();
```

El siguiente ejemplo de código muestra la `NavigateToAsync` métodos proporcionados por el `NavigationService` clase:

```
public Task NavigateToAsync<TViewModel>() where TViewModel : ViewModelBase
{
    return InternalNavigateToAsync(typeof(TViewModel), null);
}

public Task NavigateToAsync<TViewModel>(object parameter) where TViewModel : ViewModelBase
{
    return InternalNavigateToAsync(typeof(TViewModel), parameter);
}
```

Cada método permite que cualquier clase de modelo de vista que se deriva el `ViewModelBase` clase para realizar la navegación jerárquica invocando el `InternalNavigateToAsync` método. Además, el segundo `NavigateToAsync` método permite que los datos de navegación que se especifican como un argumento que se pasa al modelo de vista que se navega, donde normalmente se usa para realizar la inicialización. Para obtener más información, consulte pasar parámetros durante la navegación.

El `InternalNavigateToAsync` método se ejecuta la solicitud de navegación y se muestra en el ejemplo de código siguiente:

```
private async Task InternalNavigateToAsync(Type viewModelType, object parameter)
{
    Page page = CreatePage(viewModelType, parameter);

    if (page is LoginView)
    {
        Application.Current.MainPage = new CustomNavigationView(page);
    }
    else
    {
        var navigationPage = Application.Current.MainPage as CustomNavigationView;
        if (navigationPage != null)
        {
            await navigationPage.PushAsync(page);
        }
        else
        {
            Application.Current.MainPage = new CustomNavigationView(page);
        }
    }

    await (page.BindingContext as ViewModelBase).InitializeAsync(parameter);
}

private Type GetPageTypeForViewModel(Type viewModelType)
{
    var viewName = viewModelType.FullName.Replace("Model", string.Empty);
    var viewModelAssemblyName = viewModelType.GetTypeInfo().Assembly.FullName;
    var viewAssemblyName = string.Format(
                CultureInfo.InvariantCulture, "{0}, {1}", viewName, viewModelAssemblyName);
    var viewType = Type.GetType(viewAssemblyName);
    return viewType;
}

private Page CreatePage(Type viewModelType, object parameter)
{
    Type pageType = GetPageTypeForViewModel(viewModelType);
    if (pageType == null)
    {
        throw new Exception($"Cannot locate page type for {viewModelType}");
    }

    Page page = Activator.CreateInstance(pageType) as Page;
    return page;
}
```

El `InternalNavigateToAsync` método realiza la navegación a un modelo de vista por llamar primero a la `CreatePage` método. Este método localiza la vista que corresponde al tipo de modelo de vista especificada y crea y devuelve una instancia de este tipo de vista. Buscar la vista que se corresponde con el tipo de modelo de vista usa un enfoque basado en convenciones, que se da por supuesto que:

- Las vistas son del mismo ensamblado como tipos de modelo de vista.
- Las vistas están en una. Espacio de nombres de las vistas secundarias.
- Los modelos de vista están en una. Espacio de nombres secundario ViewModels.
- Se corresponden los nombres de vista para ver los nombres de modelo, con "Modelo" quitado.

Cuando se crea una instancia de una vista, esta se asocia con su modelo de vista correspondiente. Para obtener más información acerca de cómo ocurre esto, consulte creación automática de un modelo de vista con un localizador de modelo de vista.

Si la vista que se va a crear es un `LoginView`, se ajusta dentro de una nueva instancia de la `CustomNavigationView` clase y se asigna a la `Application.Current.MainPage` propiedad. En caso contrario, el `CustomNavigationView`

instancia es recuperar así que no es null, el `PushAsync` se invoca el método para insertar la vista se crea en la pila de navegación. Sin embargo, si el objeto recuperado `CustomNavigationView` instancia es `null`, la vista se crea se ajusta dentro de una nueva instancia de la `CustomNavigationView` clase y se asigna a la `Application.Current.MainPage` propiedad. Este mecanismo garantiza que durante la navegación, las páginas se agregan correctamente a la pila de navegación cuando está vacía y cuando no contiene datos.

> **TIP**
> Considere la posibilidad de almacenar en caché páginas. Página de almacenamiento en caché de resultados en el consumo de memoria para las vistas que no se muestran actualmente. Sin embargo, sin almacenamiento en caché de página significa que el análisis de XAML y la construcción de la página y su modelo de vista se producirán cada vez que se navega una página nueva, que puede afectar al rendimiento de una página compleja. Para una página bien diseñada que no use un número excesivo de controles, el rendimiento debería ser suficiente. Sin embargo, el almacenamiento en caché de página podría resultar útil si se producen tiempos de carga de páginas lentas.

Después de la vista se crea y se navega a, el `InitializeAsync` se ejecuta el método asociada de la vista modelo de vista. Para obtener más información, consulte pasar parámetros durante la navegación.

Navegar por la aplicación cuando se inicia

Cuando se inicia la aplicación, el `InitNavigation` método en el `App` se invoca la clase. El ejemplo de código siguiente muestra este método:

```
private Task InitNavigation()
{
    var navigationService = ViewModelLocator.Resolve<INavigationService>();
    return navigationService.InitializeAsync();
}
```

El método crea un nuevo `NavigationService` objeto en el contenedor de inserción de dependencia de Autofac y devuelve una referencia a él, antes de invocar su `InitializeAsync` método.

> **NOTE**
> Cuando el `INavigationService` interfaz se resuelve mediante la `ViewModelBase` (clase), el contenedor devuelve una referencia a la `NavigationService` objeto creado cuando se invoca el método InitNavigation.

El siguiente ejemplo de código muestra la `NavigationService` `InitializeAsync` método:

```
public Task InitializeAsync()
{
    if (string.IsNullOrEmpty(Settings.AuthAccessToken))
        return NavigateToAsync<LoginViewModel>();
    else
        return NavigateToAsync<MainViewModel>();
}
```

El `MainView` se navega a si la aplicación tiene un token de acceso almacenada en caché que se usa para la autenticación. En caso contrario, el `LoginView` se navega.

Para obtener más información sobre el contenedor de inserción de dependencia de Autofac, consulte Introducción a la inserción de dependencias.

Pasar parámetros durante la navegación

Uno de los `NavigateToAsync` métodos, especificados por el `INavigationService` interfaz, los datos de exploración

permite que se especifican como un argumento que se pasa al modelo de vista que se navega, donde normalmente se usa para realizar la inicialización.

Por ejemplo, el `ProfileViewModel` clase contiene un `OrderDetailCommand` que se ejecuta cuando el usuario selecciona un pedido en la `ProfileView` página. A su vez, esto se ejecuta el `OrderDetailAsync` método, que se muestra en el ejemplo de código siguiente:

```
private async Task OrderDetailAsync(Order order)
{
    await NavigationService.NavigateToAsync<OrderDetailViewModel>(order);
}
```

Este método invoca la navegación a la `OrderDetailViewModel`, pasando un `Order` instancia que representa el orden en que el usuario ha seleccionado en el `ProfileView` página. Cuando el `NavigationService` clase crea el `OrderDetailView`, `OrderDetailViewModel` crea y asigna a la vista de la clase `BindingContext`. Después de navegar a la `OrderDetailView`, `InternalNavigateToAsync` método se ejecuta el `InitializeAsync` método de la vista asociada al modelo de vista.

El `InitializeAsync` método se define en el `ViewModelBase` clase como un método que se puede invalidar. Este método especifica un `object` argumento que representa los datos que se pasará a un modelo de vista durante una operación de navegación. Por lo tanto, las clases de modelo de vista que desea recibir datos de una operación de navegación que proporcionen su propia implementación de la `InitializeAsync` método para realizar la inicialización necesaria. El siguiente ejemplo de código muestra la `InitializeAsync` método desde el `OrderDetailViewModel` clase:

```
public override async Task InitializeAsync(object navigationData)
{
    if (navigationData is Order)
    {
        ...
        Order = await _ordersService.GetOrderAsync(
                        Convert.ToInt32(order.OrderNumber), authToken);
        ...
    }
}
```

Este método recupera el `Order` detalles de instancia que se pasó en el modelo de vista durante la operación de navegación y lo usa para recuperar el orden completo desde el `OrderService` instancia.

Uso de comportamientos de navegación invocación

Navegación desde una vista se desencadena normalmente por una interacción del usuario. Por ejemplo, el `LoginView` realiza la navegación sigue una autenticación correcta. El ejemplo de código siguiente muestra cómo se invoca la navegación por un comportamiento:

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

En tiempo de ejecución, el `EventToCommandBehavior` responderá a la interacción con el `WebView`. Cuando el `WebView` navega a una página web, el `Navigating` se desencadenará el evento, que se ejecutará el `NavigateCommand` en el `LoginViewModel`. De forma predeterminada, los argumentos de evento para el evento se

pasan al comando. Estos datos se convierten como se pasa entre el origen y destino por el convertidor especificado en el `EventArgsConverter` propiedad, que devuelve el `Url` desde el `WebNavigatingEventArgs` . Por lo tanto, cuando el `NavigationCommand` es ejecuta, se pasa la dirección Url de la página web como un parámetro al registrado `Action`.

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

```
private async Task NavigateAsync(string url)
{
    ...
    await NavigationService.NavigateToAsync<MainViewModel>();
    await NavigationService.RemoveLastFromBackStackAsync();
    ...
}
```

Este método invoca la navegación a la `MainViewModel`, y después en panel de navegación, quita el `LoginView` página de la pila de navegación.

Confirmar o cancelar la navegación

Una aplicación podría necesitar interactuar con el usuario durante una operación de navegación, por lo que el usuario puede confirmar o cancelar la navegación. Esto podría ser necesario, por ejemplo, cuando el usuario intenta navegar antes por completo una vez completada una página de entrada de datos. En esta situación, una aplicación debe proporcionar una notificación que permite al usuario a salir de la página, o para cancelar la operación de navegación antes de que ocurra. Esto puede lograrse en una clase de modelo de vista mediante el uso de la respuesta de una notificación para controlar si se invoca la navegación.

Resumen

Xamarin.Forms incluye compatibilidad con la navegación de página, que normalmente da como resultado de la interacción del usuario con la interfaz de usuario o de la aplicación, como resultado de los cambios de estado controlado por la lógica interna. Sin embargo, navegación puede ser difícil de implementar en las aplicaciones que usan el patrón MVVM.

Este capítulo presentada un `NavigationService` (clase), que se usa para realizar la navegación de model first de vista de modelos de vista. Colocar en la vista lógica de navegación clases de modelo significa que la lógica puede realizarse a través de pruebas automatizadas. Además, el modelo de vista, a continuación, puede implementar la lógica para controlar el desplazamiento para garantizar que se apliquen determinadas reglas empresariales.

Vínculos relacionados

- Descargar libro electrónico (PDF de 2Mb)
- eShopOnContainers (GitHub) (ejemplo)

Validación en aplicaciones empresariales

13/11/2018 • 20 minutes to read • Edit Online

Cualquier aplicación que acepta entradas de los usuarios debe asegurarse de que la entrada es válida. Por ejemplo, podría comprobar una aplicación para la entrada que contiene solo caracteres en un intervalo determinado, es de una determinada longitud o coincide con un formato determinado. Sin validación, el usuario puede proporcionar datos a los que hará que la aplicación producirá un error. La validación aplica las reglas de negocios y evita que un atacante inserta datos malintencionados.

En el contexto de Model-View-ViewModel (MVVM) de patrón, un modelo de vista o modelo a menudo se requerirá para realizar la validación de datos y señalar los errores de validación a la vista para que el usuario puede corregirlos. La aplicación móvil de eShopOnContainers realiza la validación de cliente sincrónico 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 muestre los mensajes de error que informan al usuario de por qué los datos no están válidos. Figura 6-1 muestra las clases implicadas en realizar la validación en la aplicación móvil de eShopOnContainers.

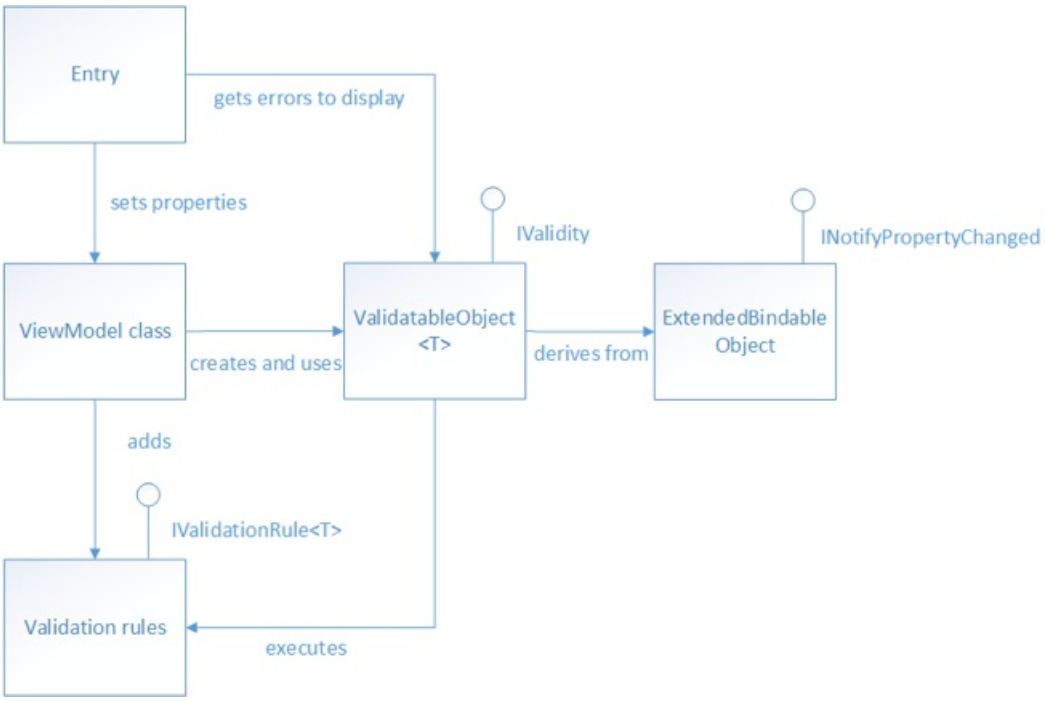

Figura 6-1: clases de validación en la aplicación móvil de eShopOnContainers

Las propiedades del modelo de vista que requieren validación son del tipo `ValidatableObject<T>` y cada `ValidatableObject<T>` instancia tiene reglas de validación que se agrega a su `Validations` propiedad. Se invoca la validación del modelo de vista mediante una llamada a la `Validate` método de la `ValidatableObject<T>` instancia, que recupera la validación de reglas y los ejecuta contra la `ValidatableObject<T>` `Value` propiedad. Errores de validación se colocan en el `Errors` propiedad de la `ValidatableObject<T>` instancia y el `IsValid` propiedad de la `ValidatableObject<T>` instancia se actualiza para indicar si la validación se realizó correctamente o no.

Notificación de cambio de propiedad proporcionada por el `ExtendedBindableObject` (clase), por lo que un `Entry` puede enlazar el control a la `IsValid` propiedad de `ValidatableObject<T>` instancia en la clase view model para recibir una notificación o no los datos introducidos son válidos.

Especificar reglas de validación

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

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

Esta interfaz especifica que una clase de regla de validación se debe proporcionar un `boolean` `Check` método que se usa para realizar la validación necesaria, y un `ValidationMessage` propiedad cuyo valor es el mensaje de error de validación que se mostrará si se produce un error de validación.

El siguiente ejemplo de código muestra la `IsNotNullOrEmptyRule<T>` regla de validación, que se usa para realizar la validación del nombre de usuario y la contraseña escrita por el usuario el `LoginView` al usar servicios ficticios en la aplicación móvil de eShopOnContainers:

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

    public bool Check(T value)
    {
        if (value == null)
        {
            return false;
        }

        var str = value as string;
        return !string.IsNullOrWhiteSpace(str);
    }
}
```

El `Check` método devuelve un `boolean` que indica si el argumento de valor es `null`, vacía o consta únicamente de caracteres de espacio en blanco.

Aunque no se usa por la aplicación móvil de eShopOnContainers, en el ejemplo de código siguiente se muestra una regla de validación para validar direcciones de correo electrónico:

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

    public bool Check(T value)
    {
        if (value == null)
        {
            return false;
        }

        var str = value as string;
        Regex regex = new Regex(@"^([\w\.\-]+)@([\w\-]+)((\.(\w){2,3})+)$");
        Match match = regex.Match(str);

        return match.Success;
    }
}
```

El `Check` método devuelve un `boolean` que indica si el argumento de valor es una dirección de correo electrónico válida. Esto se logra mediante la búsqueda en el argumento de valor para la primera aparición del patrón de

expresión regular especificada en el `Regex` constructor. Si se ha encontrado el patrón de expresión regular en la cadena de entrada se puede determinar examinando el valor de la `Match` del objeto `Success` propiedad.

> **NOTE**
> Validación de propiedades a veces puede implicar a las propiedades dependientes. Un ejemplo de propiedades dependientes es cuando el conjunto de valores válidos para una propiedad depende del valor concreto que se ha establecido en la propiedad B. Para comprobar que el valor de propiedad A es uno de los valores permitidos implicaría recuperar el valor de propiedad B. Además, cuando cambia el valor de propiedad B, necesitaría una propiedad volver a validar.

Agregar reglas de validación a una propiedad

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

```
public ValidatableObject<string> UserName
{
    get
    {
        return _userName;
    }
    set
    {
        _userName = value;
        RaisePropertyChanged(() => UserName);
    }
}

public ValidatableObject<string> Password
{
    get
    {
        return _password;
    }
    set
    {
        _password = value;
        RaisePropertyChanged(() => Password);
    }
}
```

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

```
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 el `IsNotNullOrEmptyRule<T>` regla de validación del `Validations` colección de cada `ValidatableObject<T>` instancia, especifique valores para la regla de validación `ValidationMessage` propiedad, que

especifica el mensaje de error de validación que se mostrará si se produce un error de validación.

Desencadenar la validación

El enfoque de validación que se usa en la aplicación móvil de eShopOnContainers puede desencadenar manualmente la validación de una propiedad y automáticamente la validación del desencadenador cuando cambia una propiedad.

Desencadenar manualmente la validación

Validación se puede desencadenar manualmente para una propiedad de modelo de vista. Por ejemplo, esto ocurre en la aplicación móvil de eShopOnContainers cuando el usuario pulsa el **inicio de sesión** situado en el `LoginView`, cuando se usa servicios ficticios. El comando delegado llama el `MockSignInAsync` método en el `LoginViewModel`, que invoca la validación mediante la ejecución de la `Validate` método, que se muestra en el ejemplo de código siguiente:

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

```
public bool Validate()
{
    Errors.Clear();

    IEnumerable<string> errors = _validations
        .Where(v => !v.Check(Value))
        .Select(v => v.ValidationMessage);

    Errors = errors.ToList();
    IsValid = !Errors.Any();

    return this.IsValid;
}
```

Este método borra el `Errors` colección y, a continuación, recupera cualquier validación de reglas que se han agregado para el objeto `Validations` colección. El `Check` se ejecuta el método para cada regla de validación recuperada y el `ValidationMessage` valor de propiedad para cualquier regla de validación que se produce un error al validar los datos se agrega a la `Errors` colección de la `ValidatableObject<T>` instancia. Por último, el `IsValid` se establece la propiedad y su valor se devuelve al método de llamada, que indica si la validación se realizó correctamente o no.

Desencadenar la validación cuando cambian las propiedades

También se puede desencadenar la validación cuando cambia una propiedad enlazada. Por ejemplo, cuando un enlace bidireccional en el `LoginView` establece la `UserName` o `Password` se desencadena la propiedad de validación. El siguiente ejemplo de código muestra cómo se produce esto:

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

El `Entry` control se enlaza a la `UserName.Value` propiedad de la `ValidatableObject<T>` instancia y el control `Behaviors` colección tiene un `EventToCommandBehavior` instancia se agregan a él. Este comportamiento se ejecuta el `ValidateUserNameCommand` en respuesta a las [`TextChanged`] evento y activar el `Entry`, que se produce cuando el texto en el `Entry` cambios. A su vez, el `ValidateUserNameCommand` delegado se ejecuta el `ValidateUserName` método, que se ejecuta el `Validate` método en el `ValidatableObject<T>` instancia. Por lo tanto, cada vez el usuario escribe un carácter en el `Entry` se realiza el control para el nombre de usuario, la validación de los datos especificados.

Para obtener más información acerca de los comportamientos, consulte implementar comportamientos.

Mostrar errores de validación

La aplicación móvil de eShopOnContainers notifica al usuario los errores de validación resaltando el control que contiene los datos no válidos con una línea roja y mostrando un mensaje de error que informa al usuario por qué no están válidos por debajo del control que contiene los datos de la datos no válidos. Cuando los datos no válidos se ha corregido, la línea cambia a negro y se quita el mensaje de error. Figura 6-2 se muestra la LoginView en la aplicación móvil de eShopOnContainers si hay errores de validación.

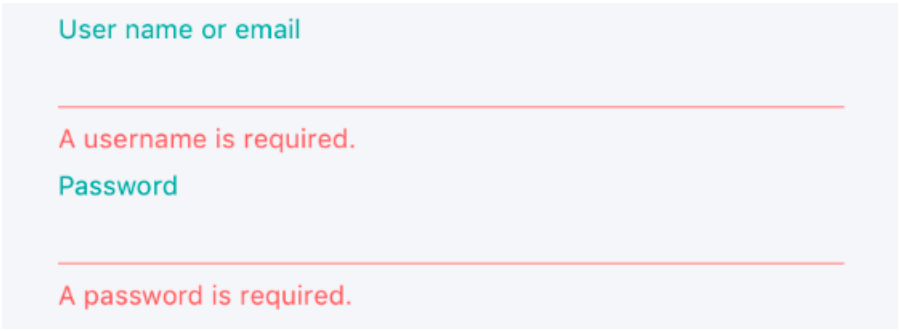

Figura 6-2: mostrar errores de validación durante el inicio de sesión

Al resaltar un Control que contiene datos no válidos

El `LineColorBehavior` comportamiento asociado se usa para resaltar `Entry` controles donde se han producido errores de validación. El siguiente ejemplo de código muestra cómo el `LineColorBehavior` comportamiento asociado se adjunta a un `Entry` control:

```
<Entry Text="{Binding UserName.Value, Mode=TwoWay}">
    <Entry.Style>
        <OnPlatform x:TypeArguments="Style">
            <On Platform="iOS, Android" Value="{StaticResource EntryStyle}" />
            <On Platform="UWP" Value="{StaticResource UwpEntryStyle}" />
        </OnPlatform>
    </Entry.Style>
    ...
</Entry>
```

El `Entry` control consume un estilo explícito, que se muestra en el ejemplo de código siguiente:

```
<Style x:Key="EntryStyle"
       TargetType="{x:Type Entry}">
    ...
    <Setter Property="behaviors:LineColorBehavior.ApplyLineColor"
            Value="True" />
    <Setter Property="behaviors:LineColorBehavior.LineColor"
            Value="{StaticResource BlackColor}" />
    ...
</Style>
```

Este estilo establece la `ApplyLineColor` y `LineColor` adjunta propiedades de la `LineColorBehavior` adjunta el comportamiento en el `Entry` control. Para obtener más información sobre los estilos, consulte estilos.

Cuando el valor de la `ApplyLineColor` propiedad adjunta es conjunto o los cambios, el `LineColorBehavior` comportamiento asociado se ejecuta el `OnApplyLineColorChanged` método, que se muestra en el ejemplo de código siguiente:

```
public static class LineColorBehavior
{
    ...
    private static void OnApplyLineColorChanged(
                BindableObject bindable, object oldValue, object newValue)
    {
        var view = bindable as View;
        if (view == null)
        {
            return;
        }

        bool hasLine = (bool)newValue;
        if (hasLine)
        {
            view.Effects.Add(new EntryLineColorEffect());
        }
        else
        {
            var entryLineColorEffectToRemove =
                    view.Effects.FirstOrDefault(e => e is EntryLineColorEffect);
            if (entryLineColorEffectToRemove != null)
            {
                view.Effects.Remove(entryLineColorEffectToRemove);
            }
        }
    }
}
```

Los parámetros de este método proporcionan la instancia del control que está asociado el comportamiento a y los valores antiguos y nuevos de la `ApplyLineColor` propiedad adjunta. El `EntryLineColorEffect` se agrega al control de la clase `Effects` colección si el `ApplyLineColor` propiedad adjunta es `true`, en caso contrario, se quita el control `Effects` colección. Para obtener más información acerca de los comportamientos, consulte implementar comportamientos.

El `EntryLineColorEffect` subclases el `RoutingEffect` clase y se muestra en el ejemplo de código siguiente:

```
public class EntryLineColorEffect : RoutingEffect
{
    public EntryLineColorEffect() : base("eShopOnContainers.EntryLineColorEffect")
    {
    }
}
```

El `RoutingEffect` clase representa un efecto de independiente de la plataforma que ajusta un efecto interno que es específico de la plataforma. Esto simplifica el proceso de eliminación del efecto, ya que no hay ningún acceso de tiempo de compilación a la información de tipo para un efecto específico de la plataforma. El `EntryLineColorEffect` llama al constructor de clase base, pasando un parámetro que consta de una concatenación del nombre del grupo de resolución y el identificador único que se especifica en cada clase efecto específico de la plataforma.

El siguiente ejemplo de código muestra la `eShopOnContainers.EntryLineColorEffect` implementación para iOS:

```
[assembly: ResolutionGroupName("eShopOnContainers")]
[assembly: ExportEffect(typeof(EntryLineColorEffect), "EntryLineColorEffect")]
namespace eShopOnContainers.iOS.Effects
{
    public class EntryLineColorEffect : PlatformEffect
    {
        UITextField control;

        protected override void OnAttached()
        {
            try
            {
                control = Control as UITextField;
                UpdateLineColor();
            }
            catch (Exception ex)
            {
                Console.WriteLine("Can't set property on attached control. Error: ", ex.Message);
            }
        }

        protected override void OnDetached()
        {
            control = null;
        }

        protected override void OnElementPropertyChanged(PropertyChangedEventArgs args)
        {
            base.OnElementPropertyChanged(args);

            if (args.PropertyName == LineColorBehavior.LineColorProperty.PropertyName ||
                args.PropertyName == "Height")
            {
                Initialize();
                UpdateLineColor();
            }
        }

        private void Initialize()
        {
            var entry = Element as Entry;
            if (entry != null)
            {
                Control.Bounds = new CGRect(0, 0, entry.Width, entry.Height);
            }
        }

        private void UpdateLineColor()
```

```
        {
            BorderLineLayer lineLayer = control.Layer.Sublayers.OfType<BorderLineLayer>()
                                                        .FirstOrDefault();

            if (lineLayer == null)
            {
                lineLayer = new BorderLineLayer();
                lineLayer.MasksToBounds = true;
                lineLayer.BorderWidth = 1.0f;
                control.Layer.AddSublayer(lineLayer);
                control.BorderStyle = UITextBorderStyle.None;
            }

            lineLayer.Frame = new CGRect(0f, Control.Frame.Height-1f, Control.Bounds.Width, 1f);
            lineLayer.BorderColor = LineColorBehavior.GetLineColor(Element).ToCGColor();
            control.TintColor = control.TextColor;
        }

        private class BorderLineLayer : CALayer
        {
        }
    }
}
```

El `OnAttached` método recupera el control nativo para Xamarin.Forms `Entry` controlar y actualiza el color de línea mediante una llamada a la `UpdateLineColor` método. El `OnElementPropertyChanged` invalidación responde a los cambios de propiedad enlazable en la `Entry` control actualizando el color de línea si el archivo adjunto `LineColor` los cambios de propiedad, o la `Height` propiedad de la `Entry` cambios. Para obtener más información acerca de los efectos, vea efectos.

Cuando se especificaron datos válidos en el `Entry` control, se aplicará una línea negra en la parte inferior del control, para indicar que no hay ningún error de validación. Figura 6-3 se muestra un ejemplo de esto.

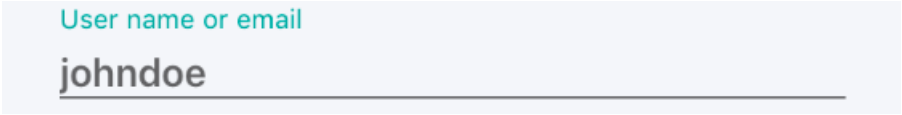

Figura 6-3: línea negra que se indica ningún error de validación

El `Entry` control también tiene un `DataTrigger` agregado a su `Triggers` colección. El siguiente ejemplo de código muestra la `DataTrigger`:

```
<Entry Text="{Binding UserName.Value, Mode=TwoWay}">
    ...
    <Entry.Triggers>
        <DataTrigger
            TargetType="Entry"
            Binding="{Binding UserName.IsValid}"
            Value="False">
            <Setter Property="behaviors:LineColorBehavior.LineColor"
                    Value="{StaticResource ErrorColor}" />
        </DataTrigger>
    </Entry.Triggers>
</Entry>
```

Esto `DataTrigger` monitores el `UserName.IsValid` propiedad y si es el valor se convierte en `false`, se ejecuta el `Setter`, los cambios que el `LineColor` adjunta propiedad de la `LineColorBehavior` adjunta el comportamiento a rojo. Figura 6-4 se muestra un ejemplo de esto.

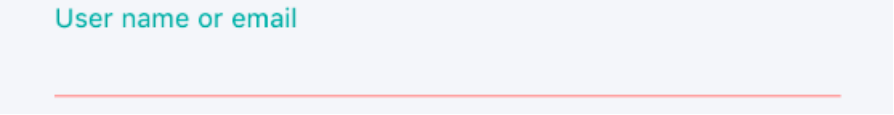

Figura 6-4: línea roja que indica el error de validación

La línea en el `Entry` control permanecerá rojo, mientras que los datos especificados no están válidos, en caso contrario, cambiará a negro para indicar que los datos introducidos son válidos.

Para obtener más información acerca de los desencadenadores, consulte desencadenadores.

Mostrar mensajes de Error

La interfaz de usuario muestra mensajes de error de validación en los controles de etiqueta debajo de cada control cuyos datos no pudo validar. El siguiente ejemplo de código muestra la `Label` que muestra un mensaje de error de validación si el usuario no ha especificado un nombre de usuario válido:

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

Cada `Label` enlaza a la `Errors` propiedad del objeto de modelo de vista que se va a validar. El `Errors` propiedad proporcionado por el `ValidatableObject<T>` clase y es de tipo `List<string>`. Dado que el `Errors` propiedad puede contener varios errores de validación, el `FirstValidationErrorConverter` instancia se usa para recuperar el primer error de la colección para su presentación.

Resumen

La aplicación móvil de eShopOnContainers realiza la validación de cliente sincrónico 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 muestre los mensajes de error que informan al usuario ¿Por qué los datos no están válidos.

Las propiedades del modelo de vista que requieren validación son del tipo `ValidatableObject<T>` y cada `ValidatableObject<T>` instancia tiene reglas de validación que se agrega a su `Validations` propiedad. Se invoca la validación del modelo de vista mediante una llamada a la `Validate` método de la `ValidatableObject<T>` instancia, que recupera la validación de reglas y los ejecuta contra la `ValidatableObject<T>` `Value` propiedad. Errores de validación se colocan en el `Errors` propiedad de la `ValidatableObject<T>` instancia y el `IsValid` propiedad de la `ValidatableObject<T>` instancia se actualiza para indicar si la validación se realizó correctamente o no.

Vínculos relacionados

- Descargar libro electrónico (PDF de 2Mb)
- eShopOnContainers (GitHub) (ejemplo)

Administración de configuración

13/07/2018 • 13 minutes to read • Edit Online

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

Configuración de la aplicación es datos que una aplicación crea y administra. Puede incluir datos como extremos de servicio web fijos, claves de API y estado en tiempo de ejecución. Configuración de la aplicación está asociada a la existencia de la aplicación y solo es significativa para esa aplicación.

Configuración de usuario es la configuración de una aplicación personalizable que afectan al comportamiento de la aplicación y no requiere el ajuste de volver a frecuentes. Por ejemplo, una aplicación podría permiten al usuario especificar dónde debe recuperar los datos y cómo se mostrará en la pantalla.

Xamarin.Forms incluye un diccionario persistente que se puede usar para almacenar los datos de configuración. Este diccionario se puede acceder mediante el `Application.Current.Properties` propiedad y los datos que se colocan en él se guarda cuando la aplicación entra en un estado de suspensión y se restaura cuando la aplicación se reanuda o se inicie de nuevo. Además, el `Application` clase también tiene un `SavePropertiesAsync` método que permite que una aplicación tenga su configuración guardada cuando sea necesario. Para obtener más información acerca de este diccionario, vea diccionario de propiedades.

Un inconveniente de almacenar datos mediante el diccionario persistente Xamarin.Forms es que no es fácilmente datos enlazados a. Por lo tanto, la aplicación móvil de eShopOnContainers utiliza la biblioteca Xam.Plugins.Settings, disponible en NuGet. Esta biblioteca proporciona un enfoque coherente, segura y multiplataforma para persistir y recuperar la configuración de aplicación y el usuario, al usar la administración de configuración nativa proporcionada por cada plataforma. Además, es fácil de usar el enlace de datos para tener acceso a datos de configuración expuestos por la biblioteca.

> **NOTE**
> Mientras que la biblioteca Xam.Plugin.Settings puede almacenar la configuración de aplicación y el usuario, hace ninguna distinción entre los dos.

Creación de una clase de configuración

Cuando se usa la biblioteca Xam.Plugins.Settings, debe crearse una única clase estática que contiene la configuración de aplicación y usuario requerida por la aplicación. El ejemplo de código siguiente muestra la clase de configuración en la aplicación móvil de eShopOnContainers:

```
public static class Settings
{
    private static ISettings AppSettings
    {
        get
        {
            return CrossSettings.Current;
        }
    }
    ...
}
```

Se puede leer y escribir a través de configuración de la `ISettings` API, que proporciona la biblioteca Xam.Plugins.Settings. Esta biblioteca proporciona un singleton que se puede usar para acceder a la API, `CrossSettings.Current`, y la clase de configuración de una aplicación debe exponer este singleton a través de un `ISettings` propiedad.

> **NOTE**
> Directivas using para los espacios de nombres Plugin.Settings y Plugin.Settings.Abstractions deben agregarse a una clase que requiere acceso a los tipos de biblioteca Xam.Plugins.Settings.

Adición de un valor

Cada opción de configuración consta de una clave, valor predeterminado y una propiedad. El siguiente ejemplo de código muestra los tres elementos en una configuración de usuario que representa la dirección URL base para los servicios en línea que conecta la aplicación móvil de eShopOnContainers con:

```
public static class Settings
{
    ...
    private const string IdUrlBase = "url_base";
    private static readonly string UrlBaseDefault = GlobalSetting.Instance.BaseEndpoint;
    ...

    public static string UrlBase
    {
        get
        {
            return AppSettings.GetValueOrDefault<string>(IdUrlBase, UrlBaseDefault);
        }
        set
        {
            AppSettings.AddOrUpdateValue<string>(IdUrlBase, value);
        }
    }
}
```

La clave siempre es una cadena constante que define el nombre de clave, con el valor predeterminado para la configuración que se va un valor de solo lectura estáticos del tipo requerido. Proporcionar un valor predeterminado, se garantiza que está disponible si se recupera un valor no establecido un valor válido.

El `UrlBase` propiedad estática usa dos métodos desde el `ISettings` API para leer o escribir el valor de configuración. El `ISettings.GetValueOrDefault` método se utiliza para recuperar un valor de configuración de almacenamiento específico de la plataforma. Si no se define ningún valor para la configuración, su valor predeterminado se recupera en su lugar. De forma similar, la `ISettings.AddOrUpdateValue` método se usa para conservar un valor de configuración para almacenamiento específico de la plataforma.

En su lugar que defina un valor predeterminado dentro de la `Settings` (clase), el `UrlBaseDefault` cadena obtiene su valor de la `GlobalSetting` clase. El siguiente ejemplo de código muestra la `BaseEndpoint` propiedad y `UpdateEndpoint` método de esta clase:

```
public class GlobalSetting
{
    ...
    public string BaseEndpoint
    {
        get { return _baseEndpoint; }
        set
        {
            _baseEndpoint = value;
            UpdateEndpoint(_baseEndpoint);
        }
    }
    ...

    private void UpdateEndpoint(string baseEndpoint)
    {
        RegisterWebsite = string.Format("{0}:5105/Account/Register", baseEndpoint);
        CatalogEndpoint = string.Format("{0}:5101", baseEndpoint);
        OrdersEndpoint = string.Format("{0}:5102", baseEndpoint);
        BasketEndpoint = string.Format("{0}:5103", baseEndpoint);
        IdentityEndpoint = string.Format("{0}:5105/connect/authorize", baseEndpoint);
        UserInfoEndpoint = string.Format("{0}:5105/connect/userinfo", baseEndpoint);
        TokenEndpoint = string.Format("{0}:5105/connect/token", baseEndpoint);
        LogoutEndpoint = string.Format("{0}:5105/connect/endsession", baseEndpoint);
        IdentityCallback = string.Format("{0}:5105/xamarincallback", baseEndpoint);
        LogoutCallback = string.Format("{0}:5105/Account/Redirecting", baseEndpoint);
    }
}
```

Cada vez que el `BaseEndpoint` propiedad está establecida, el `UpdateEndpoint` se llama al método. Este método actualiza una serie de propiedades, que se basan en el `UrlBase` configuración de usuario proporcionada por el `Settings` clases que representan los distintos puntos de conexión que se conecta la aplicación móvil de eShopOnContainers.

Enlace de datos a la configuración de usuario

En la aplicación móvil de eShopOnContainers, el `SettingsView` expone dos configuraciones de usuario. Esta configuración permite que la configuración de si la aplicación debe recuperar los datos de microservicios que se implementan como contenedores de Docker, o si la aplicación debe recuperar los datos de servicios ficticios que no requieren una conexión a internet. Al elegir esta opción recuperar datos de microservicios en contenedores, se debe especificar una dirección URL de punto de conexión base para los microservicios. Figura 7-1 se muestra el `SettingsView` cuando el usuario ha elegido recuperar datos de microservicios en contenedores.

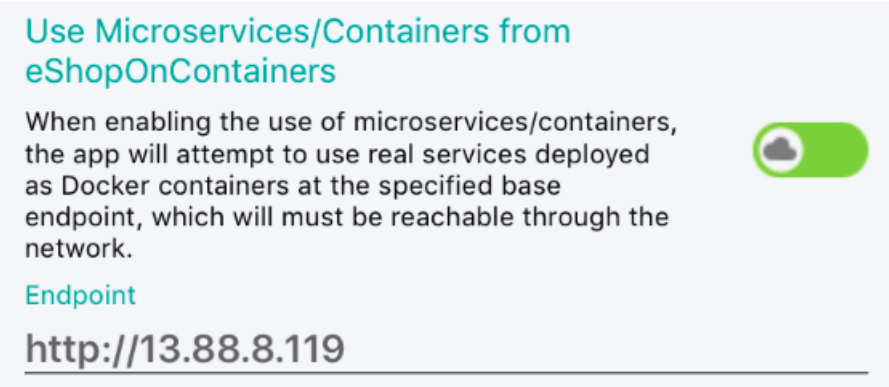

Figura 7-1: configuración de usuario expuestos por la aplicación móvil de eShopOnContainers

Enlace de datos puede usarse para recuperar y establecer la configuración expuesta por el `Settings` clase. Esto se logra mediante los controles en el enlace de vista para ver las propiedades de modelo que a su vez tienen acceso a las propiedades en el `Settings` clase y generar una propiedad notificar los cambios si ha cambiado el valor de

configuración. Para obtener información acerca de cómo la aplicación móvil de eShopOnContainers crea la vista de modelos y los asocia a vistas, consulte creación automática de un modelo de vista con un localizador de modelo de vista.

El siguiente ejemplo de código muestra la `Entry` controlar desde la `SettingsView` que permite al usuario que escriba una dirección URL de punto de conexión de base de los microservicios en contenedores:

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

Esto `Entry` control se enlaza a la `Endpoint` propiedad de la `SettingsViewModel` clase, utilizando un enlace bidireccional. El ejemplo de código siguiente muestra la propiedad de punto de conexión:

```
public string Endpoint
{
    get { return _endpoint; }
    set
    {
        _endpoint = value;

        if(!string.IsNullOrEmpty(_endpoint))
        {
            UpdateEndpoint(_endpoint);
        }

        RaisePropertyChanged(() => Endpoint);
    }
}
```

Cuando el `Endpoint` propiedad está establecida la `UpdateEndpoint` se llama al método, siempre que el valor proporcionado es válido, y cambia la propiedad notificación se genera. El siguiente ejemplo de código muestra la `UpdateEndpoint` método:

```
private void UpdateEndpoint(string endpoint)
{
    Settings.UrlBase = endpoint;
}
```

Este método actualiza el `UrlBase` propiedad en el `Settings` clase con el valor de dirección URL de punto de conexión base especificado por el usuario, lo que hace que se conservan en almacenamiento específico de la plataforma.

Cuando el `SettingsView` se navega a, el `InitializeAsync` método en el `SettingsViewModel` se ejecuta la clase. El ejemplo de código siguiente muestra este método:

```
public override Task InitializeAsync(object navigationData)
{
    ...
    Endpoint = Settings.UrlBase;
    ...
}
```

El método establece el `Endpoint` en el valor de la `UrlBase` propiedad en el `Settings` clase. Obtener acceso a la `UrlBase` propiedad hace que la biblioteca Xam.Plugins.Settings recuperar el valor de configuración de almacenamiento específico de la plataforma. Para obtener información acerca de cómo los `InitializeAsync` se invoca el método, consulte pasar parámetros durante la navegación.

Este mecanismo garantiza que cada vez que un usuario navega a la SettingsView, configuración de usuario se

recuperan de almacenamiento específico de la plataforma y presentan mediante enlace de datos. A continuación, si el usuario cambia los valores de configuración, el enlace de datos garantiza que se conservan inmediatamente al almacenamiento específico de la plataforma.

Resumen

La configuración permite la separación de datos que se configura el comportamiento de una aplicación desde el código, lo que permite el comportamiento que se puede cambiar sin volver a generar la aplicación. Configuración de la aplicación es datos que una aplicación crea y administra y configuración de usuario es la configuración de una aplicación personalizable que afectan al comportamiento de la aplicación y no requiere el ajuste de volver a frecuentes.

La biblioteca Xam.Plugins.Settings proporciona una coherente, con seguridad de tipos, enfoque multiplataforma para conservar y recuperar la aplicación y la configuración de usuario y el enlace de datos puede utilizarse para tener acceso a la configuración creada con la biblioteca.

Vínculos relacionados

- Descargar libro electrónico (PDF de 2Mb)
- eShopOnContainers (GitHub) (ejemplo)

Microservicios en contenedores

09/06/2018 • 23 minutes to read • Edit Online

Desarrollar aplicaciones cliente / servidor ha resultado en un enfoque en la creación de aplicaciones en capas que usan tecnologías específicas en cada nivel. Estas aplicaciones se conocen a menudo como *monolítico* aplicaciones y se empaquetan en hardware previamente escalada para cargas máximas. El principal inconveniente de este enfoque de desarrollo es el acoplamiento apretado entre los componentes dentro de cada nivel, que los componentes individuales no se puede escalar fácilmente y el costo de las pruebas. Una actualización simple puede tener efectos imprevistos en el resto de la capa, y por lo que un cambio en un componente de aplicación requiere su nivel todo a probar y volver a implementar.

Relativo a especialmente en el tiempo de la nube, que los componentes individuales no pueden ser fácilmente se escala. Una aplicación monolítica contiene funcionalidad específica del dominio y normalmente se divide por capas funcionales como front-end, la lógica de negocios y el almacenamiento de datos. Una aplicación monolítica se escala mediante la clonación de toda la aplicación en varios equipos, como se muestra en la figura 8-1.

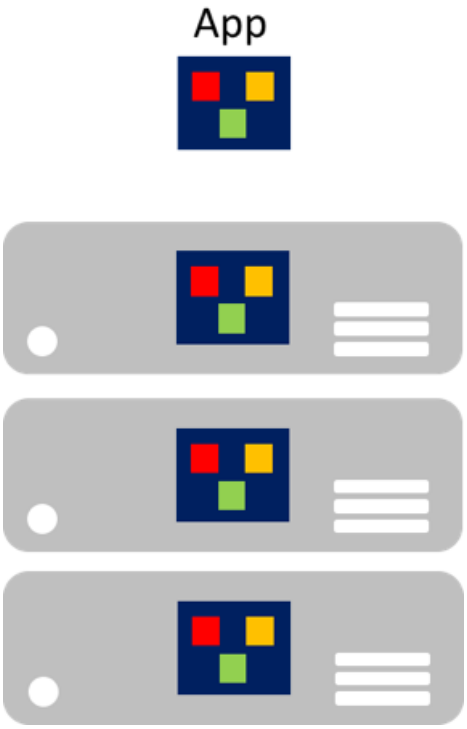

Figura 8-1: aplicación monolítico enfoque de ajuste de escala

Microservicios

Microservicios ofrecen un enfoque diferente para la implementación y desarrollo de aplicaciones, un enfoque es adecuado para la agilidad, escala y requisitos de confiabilidad de las aplicaciones modernas en la nube. Una aplicación de microservicios se descompone en componentes independientes que funcionan conjuntamente para proporcionar funcionalidad general de la aplicación. El término microservicio hace hincapié en que las aplicaciones deben estar formadas servicios lo suficientemente pequeños como para reflejar los riesgos independientes, por lo que cada microservicio implementa una única función. Además, cada microservicio tiene contratos bien definidos para que otros microservicios puedan comunicarse y compartir datos con él. Algunos ejemplos típicos de microservicios incluyen la cesta de Amazon, procesamiento, compra subsistemas y procesamiento de pago.

Microservicios pueden escalado horizontal de forma independiente, en comparación con gigantes aplicaciones

monolíticas que escalan juntos. Esto significa que se puede escalar un área funcional específica, que requiere más procesamiento alimentación eléctrica o red de ancho de banda para admitir la demanda, en lugar de innecesariamente escala horizontal de las otras áreas de la aplicación. Figura 8-2 muestra este enfoque, donde microservicios se implementan y escalar de forma independiente, la creación de instancias de servicios en los equipos.

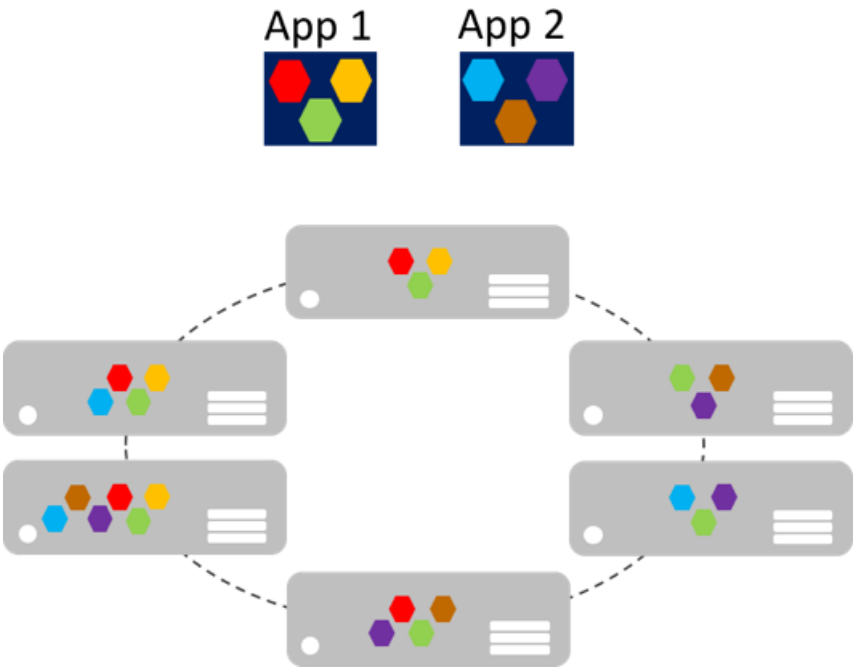

Figura 8-2: enfoque de escala de las aplicaciones Microservicios

Escalabilidad de Microservicio puede ser casi instantánea, lo que permite una aplicación para adaptarse a las cambiantes cargas. Por ejemplo, un microservicio único en la funcionalidad de web a través de una aplicación podría ser el microservicio solo en la aplicación que necesita para escalar horizontalmente para controlar el tráfico entrante adicional.

El modelo clásico de escalabilidad de la aplicación es para un nivel de equilibrio de carga, sin estado con un almacén de datos externo compartido para almacenar datos permanentes. Con estado microservicios administran sus propios datos persistentes, normalmente almacenarla de forma local en los servidores en el que se colocan, para evitar la sobrecarga de red de acceso y la complejidad de servicios cruzados operaciones. Esto permite un procesamiento más rápido posible de datos y puede eliminar la necesidad de almacenamiento en caché de sistemas. Además, escalables microservicios con estado suelen dividir los datos entre sus instancias, administrar el rendimiento de tamaño y la transferencia de datos más allá del cual puede admitir un único servidor.

Microservicios también admiten actualizaciones independientes. Este acoplamiento flexible entre microservicios proporciona una evolución aplicación rápida y confiable. Su naturaleza distribuida, independiente admite actualizaciones sucesivas, donde solo un subconjunto de instancias de una sola microservicio se actualizará en un momento dado. Por lo tanto, si se detecta un problema, una actualización con errores puede puede revertirse, antes de actualizan todas las instancias con el código defectuoso o la configuración. De forma similar, microservicios normalmente usan versiones de esquema, por lo que los clientes verán una versión coherente cuando se están aplicando las actualizaciones, independientemente de qué microservicio instancia se está comunicando con.

Por lo tanto, las aplicaciones de microservicio tienen muchas ventajas respecto a las aplicaciones monolíticas:

- Cada microservicio es relativamente pequeños, fáciles de administrar y desarrollar.
- Cada microservicio se pueden desarrollar e implementar independientemente de otros servicios.
- Cada microservicio puede ser escalado horizontal por separado. Por ejemplo, un servicio de catálogo o el servicio de cesta de compra posible que deba ser escalado horizontal más de un servicio de ordenación. Por lo

tanto, la infraestructura resultante más eficazmente consumirá recursos mientras se escala horizontalmente.

- Cada microservicio aísla los problemas. Por ejemplo, si hay un problema en un servicio solo afecta a ese servicio. Los demás servicios pueden continuar controlar las solicitudes.
- Cada microservicio puede utilizar las tecnologías más recientes. Dado que microservicios son autónomo y ejecución simultánea, las últimas tecnologías y marcos de trabajo pueden utilizarse, en lugar de verse obligado a utilizar un marco anterior que puede usarse en una aplicación monolítica.

Sin embargo, una solución de microservicio según también tiene desventajas potenciales:

- Selección de particiones de una aplicación en microservicios puede ser un reto, ya que cada microservicio debe ser completamente autónomo, to-end, incluidas la responsabilidad de sus orígenes de datos.
- Los programadores deben implementar la comunicación entre servicio, que agrega complejidad y la latencia a la aplicación.
- Las transacciones atómicas entre varios microservicios normalmente no son posibles. Por lo tanto, deben adoptar la coherencia definitiva entre microservicios requisitos empresariales.
- En producción, hay una complejidad operativa en la implementación y administración de un sistema en peligro de muchos servicios independientes.
- Comunicación de cliente a microservicio directa puede dificultar la refactorizar los contratos de microservicios. Por ejemplo, con el tiempo cómo el sistema se crean particiones en servicios deba cambiar. Un único servicio puede dividir en dos o más servicios, y pueden combinar los dos servicios. Cuando los clientes se comunican directamente con microservicios, este trabajo refactorización puede interrumpir la compatibilidad con aplicaciones de cliente.

Inclusión en contenedores

Inclusión en contenedores es un enfoque de desarrollo de software en el que una aplicación y su conjunto de dependencias, además de su configuración de entorno que se extraen como archivos de manifiesto de implementación, con control de versiones se empaquetan como una imagen de contenedor, probada como una unidad, y implementar en un sistema operativo host.

Un contenedor es un recurso controlado y portátil entorno operativo aislado, donde una aplicación puede ejecutarse sin tocar los recursos de otros contenedores, o el host. Por lo tanto, un contenedor busca y actúa como un equipo físico recién instalados o una máquina virtual.

Hay muchas similitudes entre los contenedores y las máquinas virtuales, como se muestra en la figura 8-3.

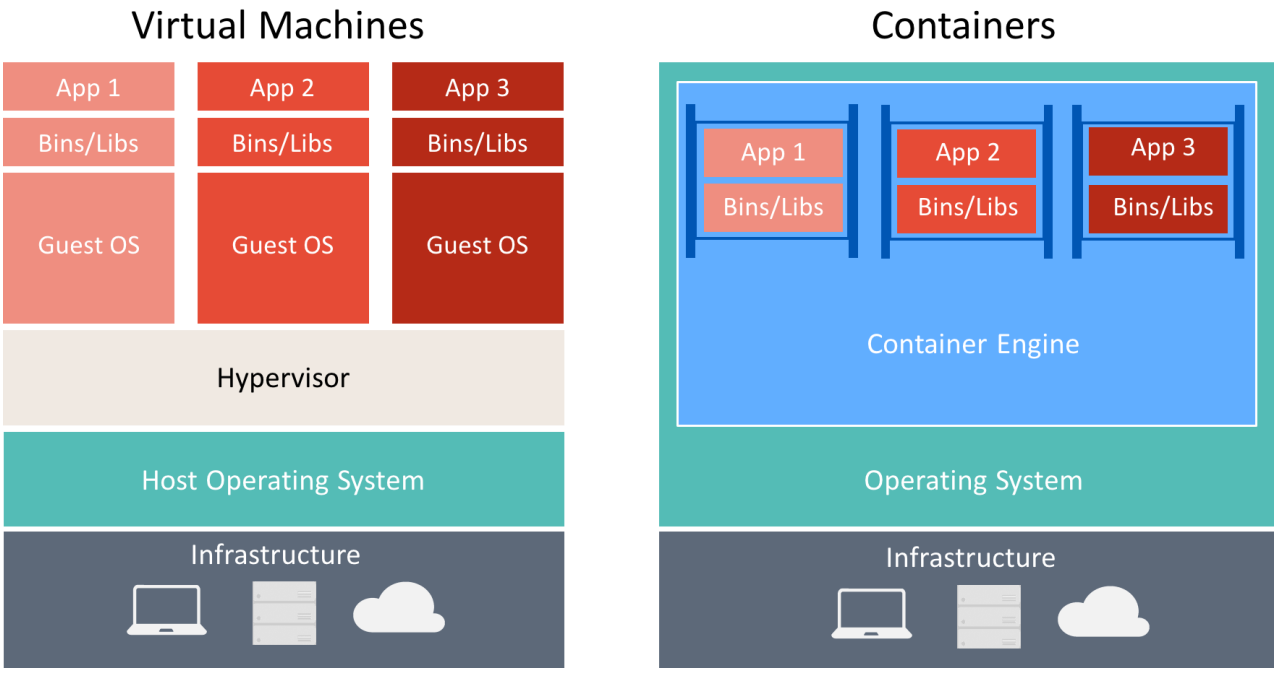

Figura 8-3: comparación de máquinas virtuales y contenedores

Un contenedor ejecuta un sistema operativo, tiene un sistema de archivos y puede tener acceso a través de una red como si fuera una máquina física o virtual. Sin embargo, la tecnología y los conceptos que se usan los contenedores son muy diferentes de las máquinas virtuales. Máquinas virtuales se incluyen las aplicaciones, las dependencias necesarias y un sistema operativo invitado completa. Contenedores incluyen la aplicación y sus dependencias, pero compartan el sistema operativo con otros contenedores, que se ejecutan como procesos aislados en el sistema operativo del host (aparte de los contenedores de Hyper-V que se ejecutan dentro de una máquina virtual especial por contenedor). Por lo tanto, contenedores de compartan recursos y suelen requieran menos recursos de las máquinas virtuales.

La ventaja de un enfoque de desarrollo e implementación orientada a servicios de contenedor es que elimina la mayoría de los problemas que surgen de configuraciones de entorno incoherentes y los problemas que se incluyen con ellas. Además, contenedores permiten la funcionalidad de ampliación rápida de aplicaciones mediante creación de instancias de nuevos contenedores según sea necesario.

Los conceptos clave al crear y trabajar con contenedores son:

- Host de contenedor: La máquina física o virtual configurado para contenedores del host. El host de contenedor ejecutará uno o varios contenedores.
- Imagen de contenedor: Una imagen consta de una unión de filesystems superpuesto apilados unos encima de otros y es la base de un contenedor. Una imagen no tiene estado y nunca cambia tal y como se implementa en diferentes entornos.
- Contenedor: Un contenedor es una instancia en tiempo de ejecución de una imagen.
- Imagen del sistema operativo de contenedor: Contenedores se implementan a partir de imágenes. La imagen de sistema operativo del contenedor es la primera capa de potencialmente muchas capas de imagen que componen un contenedor. Un sistema operativo de contenedor es inmutable y no se puede modificar.
- Repositorio de contenedor: Cada vez que se crea una imagen de contenedor, la imagen 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 modo que pueden usarse en hosts de contenedor diferentes.

Las empresas están adoptando cada vez más contenedores cuando aplicaciones basadas en la implementación de microservicio y Docker se ha convertido en la implementación de contenedor estándar que se ha optado por la mayoría de las plataformas de software y los proveedores de nube.

La aplicación de referencia eShopOnContainers usa Docker para hospedar cuatro microservicios en contenedores de back-end, como se muestra en la figura 8-4.

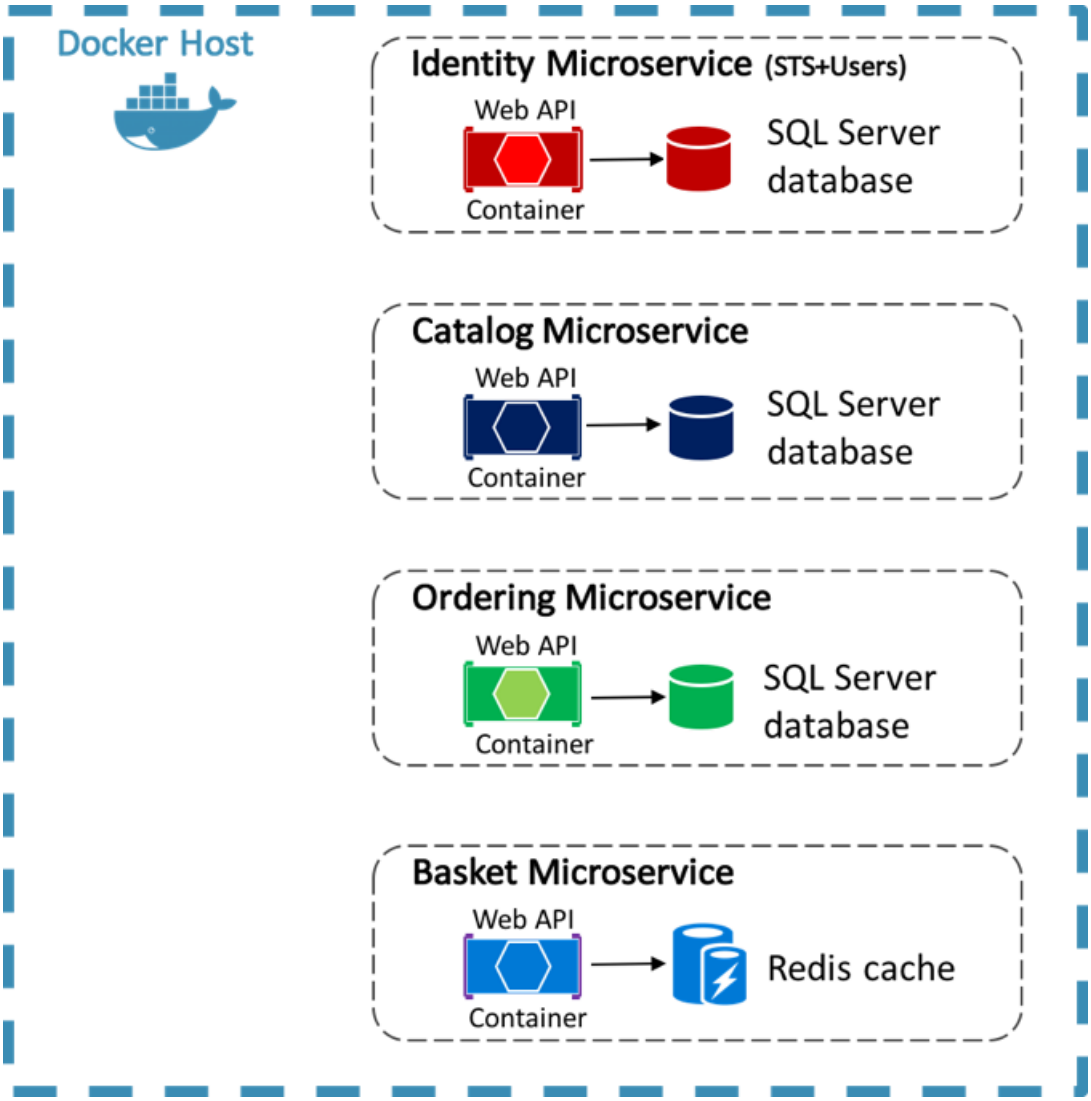

Figura 8-4: eShopOnContainers hacen referencia a microservicios de back-end de aplicación

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

Cada microservicio tiene su propia base de datos, lo que le permite totalmente se separa de los otros microservicios. En caso necesario, la coherencia entre 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 acerca de la aplicación de referencia, vea Microservicios. NET: arquitectura de aplicaciones de .NET en contenedores.

Comunicación entre cliente y Microservicios

La aplicación móvil eShopOnContainers se comunica con el uso de microservicios en contenedores de back-end *dirigir el cliente a microservicio* comunicación, que se muestra en la figura 8-5.

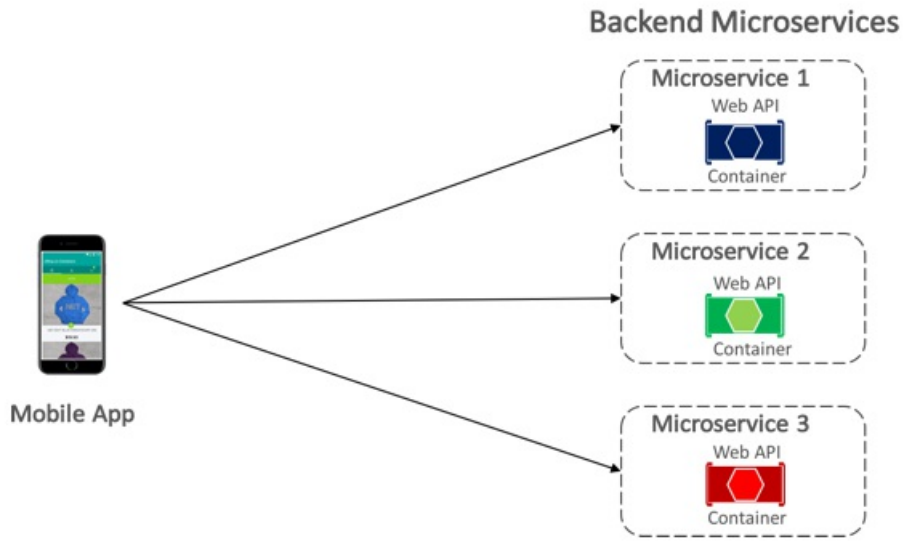

Figura 8-5: cliente a microservicio comunicación directa

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

> **TIP**
> Considere la posibilidad de utilizar la comunicación de puerta de enlace de API. Comunicación de cliente a microservicio directa puede tener desventajas al crear un microservicio grande y complejo en función de aplicación, pero resulta más adecuado para una pequeña aplicación. Al diseñar un microservicio grande según la aplicación con decenas de microservicios, considere la posibilidad de utilizar la comunicación de puerta de enlace de API. Para obtener más información, consulte Microservicios. NET: arquitectura de aplicaciones de .NET en contenedores.

Comunicación entre Microservicios

Una aplicación en función de microservicios es un sistema distribuido, ejecuta potencialmente en varias máquinas. Lo habitual es que cada instancia de servicio sea un proceso. Por lo tanto, los servicios deben interactúan mediante un protocolo de comunicación entre procesos, como HTTP, TCP, Advanced Message Queuing Protocol (AMQP) o protocolos binarios, según la naturaleza de cada servicio.

Los dos enfoques comunes para la comunicación de microservicio a microservicio son basado en HTTP comunicación REST al consultar los datos y la mensajería asincrónica ligera al comunicarse las actualizaciones a través de varios microservicios.

Mensajería basada en orientada a eventos comunicación asincrónica es fundamental para propagar cambios a través de varios microservicios. Con este enfoque, un microservicio publica un evento cuando se produce algún importantes sucede, por ejemplo, cuando actualiza una entidad de negocio. Otros microservicios suscriben a estos eventos. A continuación, cuando un microservicio recibe un evento, actualiza sus propia entidades empresariales, que a su vez podrían dar lugar a más eventos que se está publicados. Esta publicación y suscripción funcionalidad normalmente se logra con un bus de eventos.

Un bus de eventos permite la comunicación entre microservicios, sin necesidad de los componentes para tenga explícitamente entre sí, tal como se muestra en la figura 8-6 de publicación y suscripción.

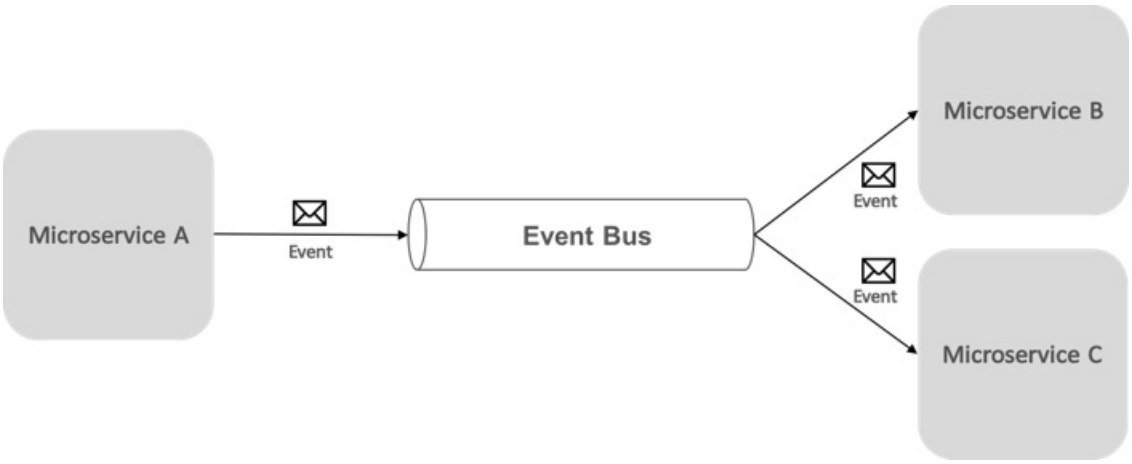

Figura 8-6: suscribirse a la publicación con un bus de eventos

Desde una perspectiva de la aplicación, el bus de eventos es simplemente una publicación-suscribirse expuesta a través de una interfaz de canal. Sin embargo, puede variar la manera en que se implementa el bus de eventos. Por ejemplo, podría utilizar una implementación de bus de eventos RabbitMQ, Service Bus de Azure u otros bus de servicio como NServiceBus y MassTransit. Figura 8-7 muestra cómo se utiliza un bus de eventos en la aplicación de referencia de eShopOnContainers.

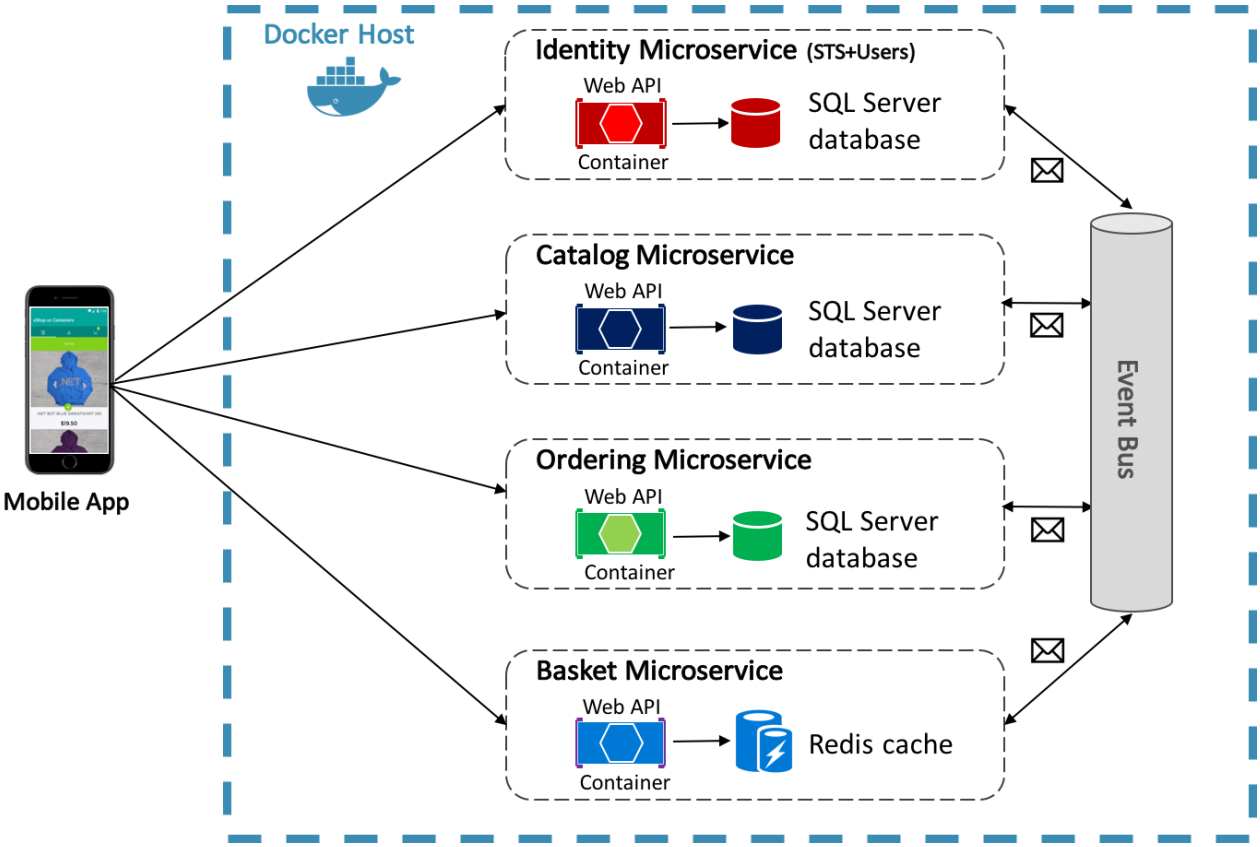

Figura 8-7: comunicación asincrónica orientada a eventos en la aplicación de referencia

El bus de eventos eShopOnContainers, implementado mediante RabbitMQ, se proporciona uno a varios asincrónica funcionalidad de publicación y suscripción. Esto significa que después de publicar un evento, puede haber varios suscriptores escuchando el mismo evento. Figura 8-9 ilustra esta relación.

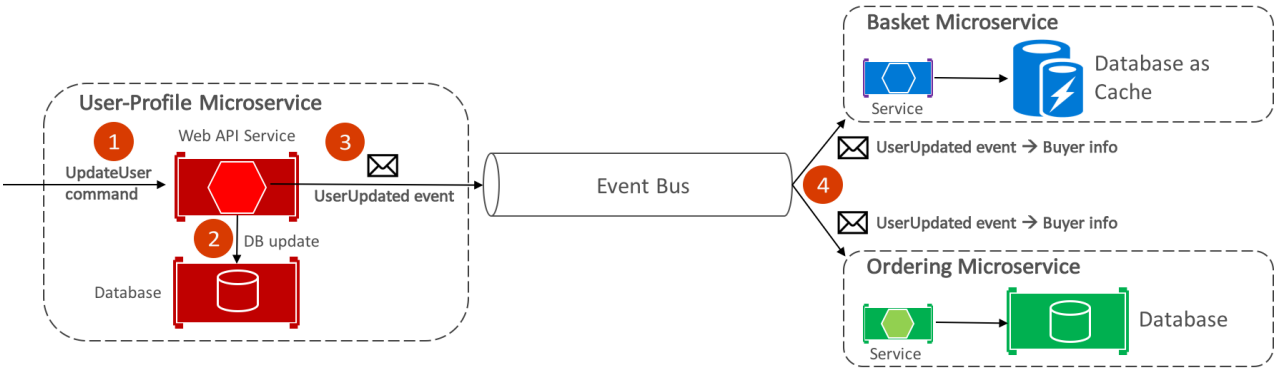

Figura 8-9: uno a varios comunicación

Este método de comunicación de uno a varios utiliza eventos para implementar las transacciones de negocio que abarcan varios servicios, garantizar la coherencia final entre los servicios. Una transacción eventual coherente consta de una serie de pasos distribuidas. Por lo tanto, cuando el perfil de usuario microservicio recibe el comando UpdateUser, actualiza los detalles del usuario en su base de datos y publica el evento UserUpdated en el bus de eventos. El microservicio de la cesta de compra y la ordenación microservicio se ha suscrito para recibir este evento y en respuesta actualizar su información de comprador en sus respectivas bases de datos.

> **NOTE**
> El bus de eventos eShopOnContainers, implementado mediante RabbitMQ, está pensado para usarse solo como una prueba de concepto. Para los sistemas de producción, se deben considerar las implementaciones del bus de eventos alternativo.

Para obtener información acerca de la implementación del bus de eventos, vea Microservicios. NET: arquitectura de aplicaciones de .NET en contenedores.

Resumen

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

Un contenedor es un recurso controlado y portátil entorno operativo aislado, donde una aplicación puede ejecutarse sin tocar los recursos de otros contenedores, o el host. Las empresas están adoptando cada vez más contenedores cuando aplicaciones basadas en la implementación de microservicio y Docker se ha convertido en la implementación de contenedor estándar que se ha optado por la mayoría de las plataformas de software y los proveedores de nube.

Vínculos relacionados

- Descargar libros electrónicos (PDF de 2Mb)
- eShopOnContainers (GitHub) (ejemplo)

Autenticación y autorización

30/01/2019 • 40 minutes to read • Edit Online

La autenticación es el proceso de obtención de credenciales de identificación como el nombre y la contraseña de un usuario y validar dichas credenciales con una entidad. Si las credenciales son válidas, la entidad que envió las credenciales se considera una identidad autenticada. Una vez que se ha autenticado una identidad, un proceso de autorización determina si esa identidad tiene acceso a un recurso determinado.

Existen varios enfoques para integrar la autenticación y autorización en una aplicación de Xamarin.Forms que se comunica con una aplicación web de ASP.NET MVC, incluido el uso de ASP.NET Core Identity, proveedores de autenticación externos como Microsoft, Google, Middleware de Twitter o Facebook y la autenticación. La aplicación móvil de eShopOnContainers realiza la autenticación y autorización con un microservicio en contenedor de identidad que usa 4 IdentityServer. La aplicación móvil solicita tokens de seguridad de IdentityServer, para autenticar un usuario o para tener acceso a un recurso. Para IdentityServer para emitir tokens en nombre de un usuario, el usuario debe iniciar sesión para IdentityServer. Sin embargo, IdentityServer no proporciona una interfaz de usuario o la base de datos para la autenticación. Por lo tanto, en la aplicación de referencia eShopOnContainers, ASP.NET Core Identity se utiliza para este propósito.

Autenticación

Se requiere autenticación, cuando una aplicación necesita conocer la identidad del usuario actual. Mecanismo principal de ASP.NET Core para identificar a los usuarios es el sistema de pertenencia de ASP.NET Core Identity, que almacena información de usuario en un almacén de datos configurado por el programador. Normalmente, este almacén de datos será un almacén de Entity Framework, aunque almacenes personalizados o paquetes de terceros pueden usarse para almacenar información de identidad en Azure storage, Azure Cosmos DB u otras ubicaciones.

Para escenarios de autenticación que usan un almacén de datos de usuario local y que conservan la información de identidad entre las solicitudes a través de las cookies (como es habitual en aplicaciones web ASP.NET MVC), ASP.NET Core Identity es una solución adecuada. Sin embargo, las cookies no son siempre una manera natural de conservar y transmitir datos. Por ejemplo, una aplicación web de ASP.NET Core que expone extremos REST que se accede desde una aplicación móvil normalmente deberá usar la autenticación de token de portador, puesto que las cookies no se puede usar en este escenario. Sin embargo, los tokens de portador pueden fácilmente recuperar e incluir en el encabezado de autorización de solicitudes web realizadas desde la aplicación móvil.

Emisión de Tokens de portador con IdentityServer 4

IdentityServer 4 es un marco de código abierto OAuth 2.0 y OpenID Connect para ASP.NET Core, que puede usarse para muchos escenarios de autenticación y autorización incluyendo la emisión de tokens de seguridad para los usuarios locales de ASP.NET Core Identity.

> **NOTE**
> OpenID Connect y OAuth 2.0 son muy similares, al tiempo que tiene responsabilidades diferentes.

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

La combinación de OpenID Connect y OAuth 2.0 se combinan los dos problemas de seguridad fundamentales de

autenticación y acceso de API y IdentityServer 4 es una implementación de estos protocolos.

En las aplicaciones que utilizan la comunicación directa de cliente a microservicio, por ejemplo, la aplicación de referencia eShopOnContainers, un microservicio de autenticación dedicado que actúe como un servicio de Token de seguridad (STS) puede usarse para autenticar a los usuarios, como se muestra en figura 9-1. Para obtener más información acerca de la comunicación directa de cliente a microservicio, consulte comunicación entre cliente y Microservicios.

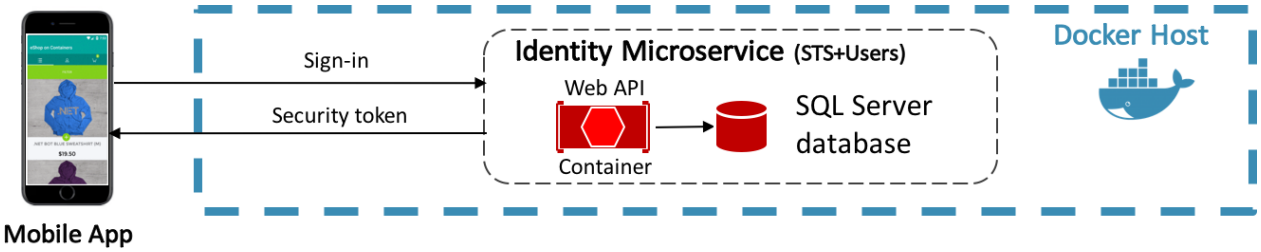

Figura 9-1: Autenticación de un microservicio de autenticación dedicado

La aplicación móvil de eShopOnContainers se comunica con el microservicio de identidad, que IdentityServer 4 se utiliza para realizar la autenticación y control de acceso para las API. Por lo tanto, la aplicación móvil solicita tokens desde IdentityServer, para autenticar un usuario o para tener acceso a un recurso:

- Autenticar a los usuarios con IdentityServer se logra mediante la aplicación móvil que solicita un *identidad* token, que representa el resultado de un proceso de autenticación. Como mínimo, contiene un identificador para el usuario e información sobre cómo y cuándo el usuario autenticado. También puede contener datos de identidad adicionales.
- Acceso a un recurso con IdentityServer se logra mediante la aplicación móvil que solicita un *acceso* token, que permite el acceso a un recurso de API. Los clientes solicitar tokens de acceso y las reenvían a la API. Los tokens de acceso contienen información sobre el cliente y el usuario (si existe). Las API, a continuación, usan esa información para autorizar el acceso a sus datos.

> **NOTE**
> Un cliente debe registrarse con IdentityServer antes de que puede solicitar tokens.

Agregar IdentityServer a una aplicación Web

En el orden de una aplicación web ASP.NET Core usan IdentityServer 4, debe agregarse a la solución de Visual Studio de la aplicación web. Para obtener más información, consulte Introducción en la documentación de IdentityServer.

Una vez que IdentityServer se incluye en la solución de Visual Studio de la aplicación web, debe agregarse a la canalización de procesamiento de solicitudes HTTP de la aplicación web para que puede atender solicitudes a puntos de conexión de OpenID Connect y OAuth 2.0. Esto se logra en el `Configure` método en la aplicación web `Startup` clase, como se muestra en el ejemplo de código siguiente:

```
public void Configure(
    IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
{
    ...
    app.UseIdentity();
    ...
}
```

Es importante el orden de procesamiento de canalización de solicitudes HTTP de la aplicación web. Por lo tanto, IdentityServer debe agregarse a la canalización antes el marco de interfaz de usuario que implementa la pantalla

de inicio de sesión.

Configurar IdentityServer

IdentityServer debe estar configurado en el `ConfigureServices` método en la aplicación web `Startup` clase mediante una llamada a la `services.AddIdentityServer` método, como se muestra en el siguiente ejemplo de código de la aplicación de referencia eShopOnContainers:

```
public void ConfigureServices(IServiceCollection services)
{
    ...
    services.AddIdentityServer(x => x.IssuerUri = "null")
        .AddSigningCredential(Certificate.Get())
        .AddAspNetIdentity<ApplicationUser>()
        .AddConfigurationStore(builder =>
            builder.UseSqlServer(connectionString, options =>
                options.MigrationsAssembly(migrationsAssembly)))
        .AddOperationalStore(builder =>
            builder.UseSqlServer(connectionString, options =>
                options.MigrationsAssembly(migrationsAssembly)))
        .Services.AddTransient<IProfileService, ProfileService>();
}
```

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

- Credenciales usadas para la firma.
- Recursos de API y la identidad que los usuarios podrían solicitar acceso.
- Clientes que se va a conectar para solicitar tokens.
- Identidad de ASP.NET Core.

> **Sugerencia**: Cargar dinámicamente la configuración de IdentityServer 4. Las API del IdentityServer 4 permiten configurar IdentityServer desde una lista en memoria de objetos de configuración. En la aplicación de referencia eShopOnContainers, estas colecciones en memoria están codificados 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 acerca de cómo configurar IdentityServer para usar ASP.NET Core Identity, vea usando ASP.NET Core Identity en la documentación de IdentityServer.

Configuración de recursos de API

Al configurar los recursos de la API, el `AddInMemoryApiResources` método espera un `IEnumerable<ApiResource>` colección. El siguiente ejemplo de código muestra la `GetApis` aplicación de referencia de método que proporciona esta recopilación en eShopOnContainers:

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

Este método especifica que debe proteger IdentityServer los pedidos y las API de la cesta de compras. Por lo tanto, IdentityServer administra el acceso a los tokens se solicitará al realizar llamadas a estas API. Para obtener más información sobre la `ApiResource` , vea recurso de la API en la documentación IdentityServer 4.

Configuración de recursos de identidad

Al configurar los recursos de identidad, el `AddInMemoryIdentityResources` método espera un `IEnumerable<IdentityResource>` colección. Recursos de identidad son datos como Id. de usuario, nombre o dirección de correo electrónico. Cada recurso de identidad tiene un nombre único y tipos de notificaciones arbitrarias se pueden asignar a él, que, a continuación, se incluirá en el token de identidad para el usuario. El siguiente ejemplo de código muestra la `GetResources` aplicación de referencia de método que proporciona esta recopilación en eShopOnContainers:

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

La especificación de OpenID Connect especifica algunos recursos estándar de identidad. El requisito mínimo es que se proporciona soporte técnico para emitir un identificador único para los usuarios. Esto se consigue al exponer el `IdentityResources.OpenId` recursos de identidad.

> **NOTE**
> La `IdentityResources` clase es compatible con todos los ámbitos definidos en la especificación de OpenID Connect (openid, correo electrónico, perfiles, teléfono y dirección).

IdentityServer también admite la definición de recursos de identidad personalizada. Para obtener más información, consulte definir recursos de identidad personalizada en la documentación de IdentityServer. Para obtener más información sobre la `IdentityResource` , vea recursos de identidad en la documentación IdentityServer 4.

Configuración de clientes

Los clientes son aplicaciones que pueden solicitar tokens de IdentityServer. Normalmente, las siguientes opciones deben definirse para cada cliente como mínimo:

- Un identificador único cliente.
- Las interacciones permitidas con el servicio de token (conocido como el tipo de concesión).
- La ubicación donde se envían los tokens de identidad y acceso (conocido como un URI de redirección).
- Una lista de recursos que el cliente puede tener acceso a (conocido como ámbitos).

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

```
public static IEnumerable<Client> GetClients(Dictionary<string,string> clientsUrl)
{
    return new List<Client>
    {
        ...
        new Client
        {
            ClientId = "xamarin",
            ClientName = "eShop Xamarin OpenId Client",
            AllowedGrantTypes = GrantTypes.Hybrid,
            ClientSecrets =
            {
                new Secret("secret".Sha256())
            },
            RedirectUris = { clientsUrl["Xamarin"] },
            RequireConsent = false,
            RequirePkce = true,
            PostLogoutRedirectUris = { $"{clientsUrl["Xamarin"]}/Account/Redirecting" },
            AllowedCorsOrigins = { "http://eshopxamarin" },
            AllowedScopes = new List<string>
            {
                IdentityServerConstants.StandardScopes.OpenId,
                IdentityServerConstants.StandardScopes.Profile,
                IdentityServerConstants.StandardScopes.OfflineAccess,
                "orders",
                "basket"
            },
            AllowOfflineAccess = true,
            AllowAccessTokensViaBrowser = true
        },
        ...
    };
}
```

Esta configuración especifica los datos de las siguientes propiedades:

- `ClientId`: Un identificador único para el cliente.
- `ClientName`: El nombre para mostrar cliente, que se usa para el registro y la pantalla de consentimiento.
- `AllowedGrantTypes`: Especifica cómo un cliente desea interactuar con IdentityServer. Para obtener más información, consulte configurar el flujo de autenticación.
- `ClientSecrets`: Especifica las credenciales del secreto de cliente que se usan al solicitar tokens desde el punto de conexión de token.
- `RedirectUris`: Especifica al URI permitido que se va a devolver códigos de autorización o tokens.
- `RequireConsent`: Especifica si se requiere una pantalla de consentimiento.
- `RequirePkce`: Especifica si los clientes que usan un código de autorización deben enviar una clave de prueba.
- `PostLogoutRedirectUris`: Especifica los URI para redirigir a después de cierre de sesión permitidos.
- `AllowedCorsOrigins`: Especifica el origen del cliente para que IdentityServer puede permitir llamadas entre orígenes desde el origen.
- `AllowedScopes`: Especifica los recursos que el cliente tiene acceso a. De forma predeterminada, un cliente no tiene acceso a todos los recursos.
- `AllowOfflineAccess`: Especifica si el cliente puede solicitar tokens de actualización.

Configurar el flujo de autenticación

El flujo de autenticación entre un cliente y IdentityServer puede configurarse mediante la especificación de los tipos de concesión en el `Client.AllowedGrantTypes` propiedad. Las especificaciones de OAuth 2.0 y OpenID Connect definen un número de flujos de autenticación, incluidos:

- Implícita. Este flujo está optimizado para aplicaciones basadas en explorador y se debe usar sólo la

autenticación de usuario o solicitudes de token de acceso y autenticación. Todos los tokens se transmiten a través del explorador y, por tanto, características avanzados, como los tokens de actualización no se permiten.

- Código de autorización. Este flujo proporciona la capacidad de recuperar los tokens en un canal, en lugar del canal frontal de explorador, al tiempo que también admite la autenticación de cliente.
- Híbrido. Este flujo es una combinación de implícito y tipos de concesión de código de autorización. El token de identidad se transmite a través del canal de explorador y contiene la respuesta de protocolo con signo, junto con otros artefactos, como el código de autorización. Tras la validación de la respuesta, el canal debe usarse para recuperar el acceso y el token de actualización.

> **TIP**
> Use el flujo de autenticación híbrido. El flujo de autenticación híbrida mitiga un número de ataques que se aplican al canal de explorador y es el flujo recomendado para aplicaciones nativas que se desea recuperar los tokens de acceso (y posiblemente los tokens de actualización).

Para obtener más información sobre los flujos de autenticación, consulte tipos de concesión en la documentación IdentityServer 4.

Realizar la autenticación

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

La aplicación móvil de eShopOnContainers se autentica con IdentityServer con el flujo de autenticación híbrido, que se muestra en la figura 9-2.

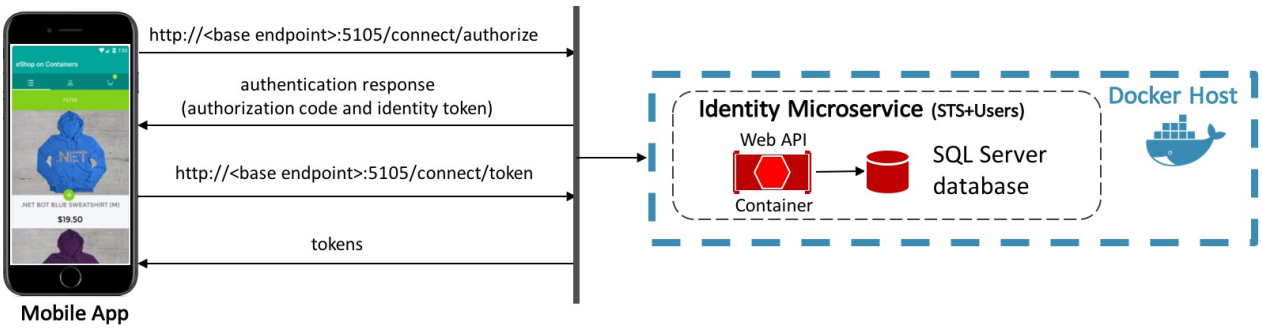

Figura 9-2: Descripción general del proceso de inicio de sesión

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

El eShopOnContainers aplicación móvil signos horizontal de IdentityServer enviando una solicitud a `<base endpoint>:5105/connect/endsession`, con parámetros adicionales. Después de cierre de sesión, IdentityServer responde enviando un URI de redireccionamiento de cierre de sesión de post a la aplicación móvil. Figura 9-3 se ilustra este proceso.

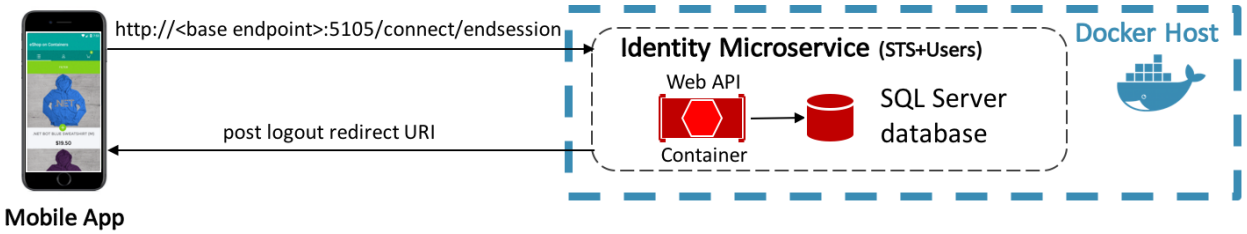

Figura 9-3: Descripción general del proceso de cierre de sesión

En la aplicación móvil de eShopOnContainers, la comunicación con IdentityServer se realiza mediante el `IdentityService` clase que implementa el `IIdentityService` interfaz. Esta interfaz especifica que debe proporcionar la clase de implementación `CreateAuthorizationRequest`, `CreateLogoutRequest`, y `GetTokenAsync` métodos.

Inicio de sesión

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

```
private async Task SignInAsync()
{
    ...
    LoginUrl = _identityService.CreateAuthorizationRequest();
    IsLogin = true;
    ...
}
```

Este método invoca el `CreateAuthorizationRequest` método en el `IdentityService` (clase), que se muestra en el ejemplo de código siguiente:

```
public string CreateAuthorizationRequest()
{
    // Create URI to authorization endpoint
    var authorizeRequest = new AuthorizeRequest(GlobalSetting.Instance.IdentityEndpoint);

    // Dictionary with values for the authorize request
    var dic = new Dictionary<string, string>();
    dic.Add("client_id", GlobalSetting.Instance.ClientId);
    dic.Add("client_secret", GlobalSetting.Instance.ClientSecret);
    dic.Add("response_type", "code id_token");
    dic.Add("scope", "openid profile basket orders locations marketing offline_access");
    dic.Add("redirect_uri", GlobalSetting.Instance.IdentityCallback);
    dic.Add("nonce", Guid.NewGuid().ToString("N"));
    dic.Add("code_challenge", CreateCodeChallenge());
    dic.Add("code_challenge_method", "S256");

    // Add CSRF token to protect against cross-site request forgery attacks.
    var currentCSRFToken = Guid.NewGuid().ToString("N");
    dic.Add("state", currentCSRFToken);

    var authorizeUri = authorizeRequest.Create(dic);
    return authorizeUri;
}
```

Este método crea el URI del IdentityServer extremo de autorización, con los parámetros necesarios. Es el punto de conexión de autorización en `/connect/authorize` 5105 del punto de conexión base que se expone como una configuración de usuario en el puerto. Para obtener más información acerca de la configuración de usuario, consulte Configuration Management.

NOTE

Se reduce la superficie de ataque de la aplicación móvil de eShopOnContainers mediante la implementación de la clave de prueba de extensión de código Exchange (PKCE) para OAuth. PKCE evita que el código de autorización que se va a utilizar si se interceptan. Esto se logra mediante el cliente de generación de un comprobador de secreto, un hash de los cuales se pasa la solicitud de autorización, y que se presenta este tipo al canjear el código de autorización. Para obtener más información sobre PKCE, consulte clave de prueba para el intercambio de código por los clientes públicos de OAuth en el sitio web de Internet Engineering Task Force.

El identificador URI devuelto se almacena en el `LoginUrl` propiedad de la `LoginViewModel` clase. Cuando el `IsLogin` propiedad pasa a ser `true`, `WebView` en el `LoginView` se vuelve visible. El `WebView` enlaza su `Source` propiedad a la `LoginUrl` propiedad de la `LoginViewModel` clase y, por lo que realiza una solicitud de inicio de sesión a IdentityServer cuando el `LoginUrl` propiedad está establecida en Punto de conexión de autorización del IdentityServer. Cuando IdentityServer recibe esta solicitud y el usuario no está autenticado, el `WebView` le redirigirá a la página de inicio de sesión configurada, que se muestra en la figura 9-4.

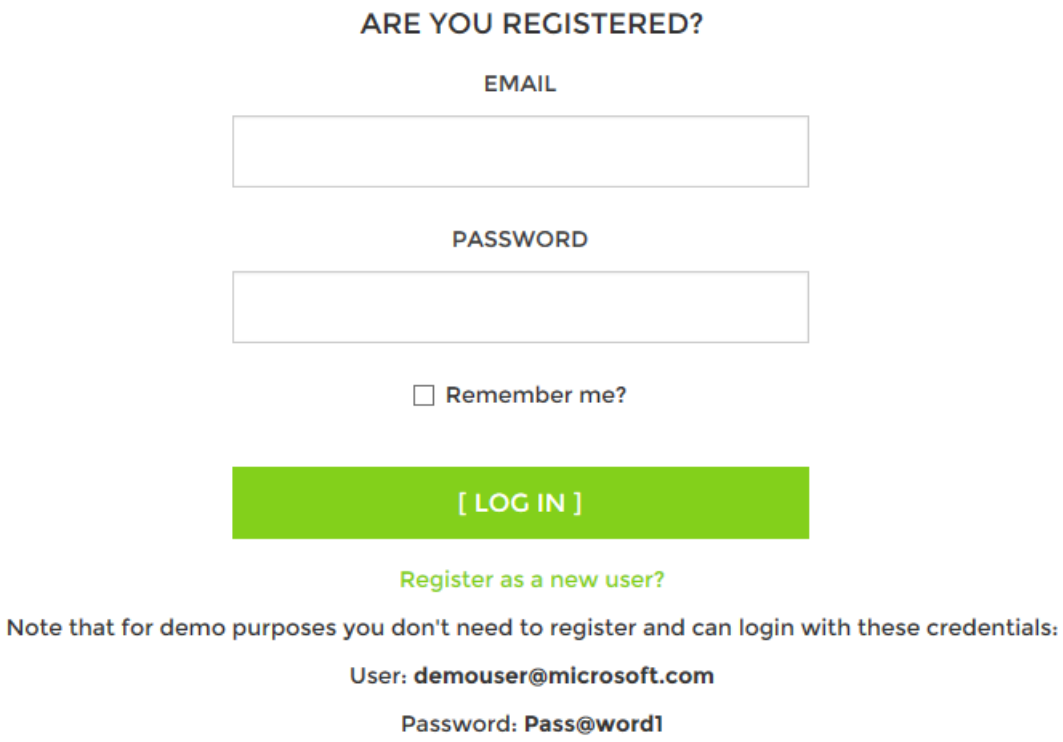

Figura 9-4: Página de inicio de sesión que se muestra la vista Web

Una vez completado el inicio de sesión, el `WebView` será redirigido a un URI devuelto. Esto `WebView` navegación provocará la `NavigateAsync` método en el `LoginViewModel` clase se ejecutarán, que se muestra en el ejemplo de código siguiente:

```
private async Task NavigateAsync(string url)
{
    ...
    var authResponse = new AuthorizeResponse(url);
    if (!string.IsNullOrWhiteSpace(authResponse.Code))
    {
        var userToken = await _identityService.GetTokenAsync(authResponse.Code);
        string accessToken = userToken.AccessToken;

        if (!string.IsNullOrWhiteSpace(accessToken))
        {
            Settings.AuthAccessToken = accessToken;
            Settings.AuthIdToken = authResponse.IdentityToken;

            await NavigationService.NavigateToAsync<MainViewModel>();
            await NavigationService.RemoveLastFromBackStackAsync();
        }
    }
    ...
}
```

Este método analiza la respuesta de autenticación que se encuentra en el URI devuelto, y siempre que un código de autorización válido está presente, realiza una solicitud del IdentityServer extremo de token, pasando el código de autorización, el Comprobador de secreto PKCE y otros parámetros necesitan. El extremo de token es en `/connect/token` 5105 del punto de conexión base que se expone como una configuración de usuario en el puerto. Para obtener más información acerca de la configuración de usuario, consulte Configuration Management.

> **Sugerencia**: Validar los URI devuelto. Aunque la aplicación móvil de eShopOnContainers no valida el URI devuelto, el procedimiento recomendado es validar que el URI devuelto hace referencia a una ubicación conocida, para evitar ataques de redireccionamiento de abierto.

Si el extremo de token recibe un código de autorización válido y el Comprobador de secreto PKCE, responde con un token de acceso, el token de identidad y el token de actualización. El token de acceso (que permite el acceso a recursos de la API) y el token de identidad, a continuación, se almacenan como configuración de la aplicación, y se realiza la navegación de páginas. Por lo tanto, el efecto general en la aplicación móvil de eShopOnContainers es esto: siempre que los usuarios pueden autenticarse correctamente con IdentityServer, se les dirige a la `MainView` página, que es un `TabbedPage` que muestra el `CatalogView` como su pestaña seleccionada.

Para obtener información sobre la navegación de página, vea navegación. Para obtener información acerca de cómo `WebView` navegación hace que un método de modelo de vista se ejecuta, consulte utilizando los comportamientos de navegación invocar. Para obtener información acerca de la configuración de la aplicación, consulte Configuration Management.

> **NOTE**
> EShopOnContainers también permite un inicio de sesión de simulacro cuando la aplicación está configurada para usar servicios ficticios en el `SettingsView`. En este modo, la aplicación no se comunica con IdentityServer, en su lugar, lo que permite al usuario que inicie sesión con las credenciales.

Signing-out

Cuando el usuario pulsa el **cerrar sesión** situado en la `ProfileView`, el `LogoutCommand` en el `ProfileViewModel` clase se ejecuta, que a su vez se ejecuta el `LogoutAsync` método. Este método realiza la navegación de página a la `LoginView` página, pasando un `LogoutParameter` instancia establecida en `true` como un parámetro. Para obtener más información sobre cómo pasar parámetros durante la navegación de página, vea pasar parámetros durante la navegación.

Cuando se crea una vista y se navega a, el `InitializeAsync` se ejecuta el método asociada de la vista modelo de vista, que, a continuación, se ejecuta el `Logout` método de la `LoginViewModel` (clase), que se muestra en el ejemplo de código siguiente:

```
private void Logout()
{
    var authIdToken = Settings.AuthIdToken;
    var logoutRequest = _identityService.CreateLogoutRequest(authIdToken);

    if (!string.IsNullOrEmpty(logoutRequest))
    {
        // Logout
        LoginUrl = logoutRequest;
    }
    ...
}
```

Este método invoca el `CreateLogoutRequest` método en el `IdentityService` clase, pasa el token de identidad se recupera de la configuración de la aplicación como un parámetro. Para obtener más información acerca de la configuración de la aplicación, consulte Configuration Management. El siguiente ejemplo de código muestra el método `CreateLogoutRequest`:

```
public string CreateLogoutRequest(string token)
{
    ...
    return string.Format("{0}?id_token_hint={1}&post_logout_redirect_uri={2}",
        GlobalSetting.Instance.LogoutEndpoint,
        token,
        GlobalSetting.Instance.LogoutCallback);
}
```

Este método crea el URI del IdentityServer terminar en punto de conexión de sesión, con los parámetros necesarios. Es el punto de conexión de sesión final en `/connect/endsession` 5105 del punto de conexión base que se expone como una configuración de usuario en el puerto. Para obtener más información acerca de la configuración de usuario, consulte Configuration Management.

El identificador URI devuelto se almacena en el `LoginUrl` propiedad de la `LoginViewModel` clase. Mientras el `IsLogin` propiedad es `true`, `WebView` en el `LoginView` está visible. El `WebView` enlaza su `Source` propiedad a la `LoginUrl` propiedad de la `LoginViewModel` clase y, por lo que realiza una solicitud de cierre de sesión a IdentityServer cuando el `LoginUrl` propiedad está establecida en Punto de conexión de sesión del IdentityServer end. Cuando IdentityServer recibe esta solicitud, siempre que sea el usuario ha iniciado sesión, se produce cierre de sesión. La autenticación se realiza un seguimiento con una cookie administrada por el middleware de autenticación de cookies de ASP.NET Core. Por lo tanto, cerrando la sesión IdentityServer quita la cookie de autenticación y envía un redireccionamiento posterior al cierre de sesión URI se devuelva al cliente.

En la aplicación móvil, el `WebView` le redirigirá a la URI de redireccionamiento de cierre de sesión de post. Esto `WebView` navegación provocará la `NavigateAsync` método en el `LoginViewModel` clase se ejecutarán, que se muestra en el ejemplo de código siguiente:

```
private async Task NavigateAsync(string url)
{
    ...
    Settings.AuthAccessToken = string.Empty;
    Settings.AuthIdToken = string.Empty;
    IsLogin = false;
    LoginUrl = _identityService.CreateAuthorizationRequest();
    ...
}
```

Este método borra el token de identidad y el token de acceso de configuración de la aplicación y establece el `IsLogin` propiedad `false`, lo que hace que el `WebView` en el `LoginView` página sea invisible . Por último, el `LoginUrl` propiedad está establecida en el URI de IdentityServer extremo de autorización, con los parámetros necesarios, como preparación para la próxima vez que el usuario inicia una sesión.

Para obtener información sobre la navegación de página, vea navegación. Para obtener información acerca de cómo `WebView` navegación hace que un método de modelo de vista se ejecuta, consulte utilizando los comportamientos de navegación invocar. Para obtener información acerca de la configuración de la aplicación, consulte Configuration Management.

> **NOTE**
> EShopOnContainers también permite un simulacro cierre de sesión cuando la aplicación está configurada para usar servicios ficticios en el SettingsView. En este modo, la aplicación no se comunica con IdentityServer y borra los tokens almacenados de configuración de la aplicación.

Autorización

Tras la autenticación, la web de ASP.NET Core API a menudo se necesitan para autorizar el acceso, lo que permite un servicio para que API disponible para algunos usuarios autenticados, pero no a todos.

Restringir el acceso a una ruta de ASP.NET Core MVC puede lograrse mediante la aplicación de un atributo Authorize a un controlador o acción, lo que limita el acceso al controlador o acción que los usuarios autenticados, como se muestra en el ejemplo de código siguiente:

```
[Authorize]
public class BasketController : Controller
{
    ...
}
```

Si un usuario no autorizado intenta obtener acceso a un controlador o acción que está marcado con el `Authorize` atributo, el marco de MVC devuelve un código de estado HTTP 401 (no autorizado).

> **NOTE**
> Se pueden especificar parámetros en el `Authorize` atributo para restringir una API a usuarios específicos. Para obtener más información, consulte autorización.

IdentityServer se puede integrar en el flujo de trabajo de autorización para que los tokens de acceso proporciona autorización de control. Este método se muestra en la figura 9-5.

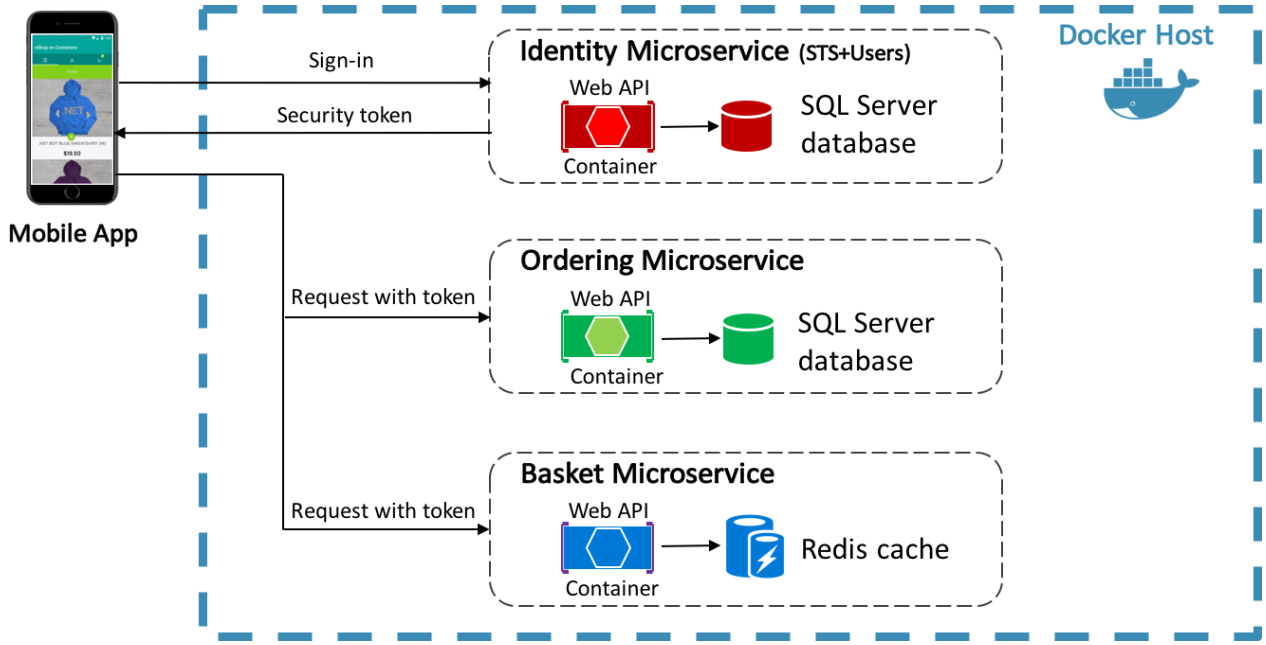

Figura 9-5: Autorización de token de acceso

La aplicación móvil de eShopOnContainers se comunica con el microservicio de identidad y solicita un token de acceso como parte del proceso de autenticación. El token de acceso, a continuación, se reenvía a las API expuestas por los microservicios ordering y cesta como parte de las solicitudes de acceso. Los tokens de acceso contienen información sobre el cliente y el usuario. Las API, a continuación, usan esa información para autorizar el acceso a sus datos. Para obtener información sobre cómo configurar IdentityServer para proteger las API, consulte configurar recursos de la API.

Configurar IdentityServer para realizar la autorización

Para realizar la autorización con IdentityServer, su middleware de autorización debe agregarse a la canalización de solicitudes HTTP de la aplicación web. El software intermedio se agrega en el `ConfigureAuth` método en la aplicación web `Startup` (clase), que se invoca desde el `Configure` método y se muestra en el siguiente ejemplo de código de la aplicación de referencia eShopOnContainers:

```
protected virtual void ConfigureAuth(IApplicationBuilder app)
{
    var identityUrl = Configuration.GetValue<string>("IdentityUrl");
    app.UseIdentityServerAuthentication(new IdentityServerAuthenticationOptions
    {
        Authority = identityUrl.ToString(),
        ScopeName = "basket",
        RequireHttpsMetadata = false
    });
}
```

Este método garantiza que la API solo se puede acceder con un token de acceso válido. El software intermedio valida el token entrante para asegurarse de 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, devolverá un 401 (no autorizado) código de estado HTTP, que indica que se requiere un token de acceso al controlador de ordenación o de la cesta de compras de exploración.

> **NOTE**
> Middleware de autorización del IdentityServer debe agregarse a la canalización de solicitudes HTTP de la aplicación web antes de agregar MVC con `app.UseMvc()` o `app.UseMvcWithDefaultRoute()`.

Hacer que las solicitudes de acceso a las API

Al realizar peticiones a los microservicios de la cesta de compras y ordenación, el acceso de token, obtenida IdentityServer durante el proceso de autenticación, debe incluirse en la solicitud, tal como se muestra en el ejemplo de código siguiente:

```
var authToken = Settings.AuthAccessToken;
Order = await _ordersService.GetOrderAsync(Convert.ToInt32(order.OrderNumber), authToken);
```

El token de acceso se almacena como una configuración de aplicación y es recuperar desde el almacenamiento específico de la plataforma e incluido en la llamada a la `GetOrderAsync` método en el `OrderService` clase.

De forma similar, el token de acceso debe incluirse al enviar datos a un IdentityServer protegidos API, tal como se muestra en el ejemplo de código siguiente:

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

El token de acceso es recuperar desde el almacenamiento específico de la plataforma y se incluye en la llamada a la `UpdateBasketAsync` método en el `BasketService` clase.

El `RequestProvider` (clase), en la aplicación móvil de eShopOnContainers, usa el `HttpClient` clase para realizar solicitudes a las API de REST expuestas por la aplicación de referencia eShopOnContainers. Al realizar solicitudes a los pedidos y la cesta de la API, que requieren autorización, debe incluirse un token de acceso válido con la solicitud. Esto se logra agregando el token de acceso a los encabezados de la `HttpClient` de instancia, tal como se muestra en el ejemplo de código siguiente:

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

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

Para obtener más información acerca de cómo la aplicación móvil de eShopOnContainers realiza solicitudes web, consulte acceso a datos remotos.

Resumen

Existen muchos enfoques para integrar la autenticación y autorización en una aplicación de Xamarin.Forms que se comunica con una aplicación web ASP.NET MVC. La aplicación móvil de eShopOnContainers realiza la autenticación y autorización con un microservicio en contenedor de identidad que usa 4 IdentityServer. IdentityServer es un marco de código abierto de OAuth 2.0 y OpenID Connect para ASP.NET Core que se integra con ASP.NET Core Identity para realizar la autenticación de token de portador.

La aplicación móvil solicita tokens de seguridad de IdentityServer, para autenticar un usuario o para tener acceso a un recurso. Al acceder a un recurso, un token de acceso debe incluirse en la solicitud a las API que requieren autorización. Middleware de IdentityServer valida los tokens de acceso entrante para asegurarse de que se envían desde un emisor de confianza y que son válidos para su uso con la API que recibe.

Vínculos relacionados

- Descargar libro electrónico (PDF de 2Mb)
- eShopOnContainers (GitHub) (ejemplo)

Acceso a datos remotos

26/10/2018 • 42 minutes to read • Edit Online

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

Las aplicaciones cliente deben ser capaces de usar la API web sin necesidad de saber cómo se implementan los datos o las operaciones que expone la API. Esto requiere que la API se rija por las normas comunes que permiten a un servicio web y aplicación de cliente acuerden qué formatos de datos que se usarán y la estructura de los datos que se intercambian entre las aplicaciones cliente y el servicio web.

Introducción a la transferencia de estado representacional

Representational State Transfer (REST) es un estilo de arquitectura para la creación de sistemas distribuidos basados en hipermedia. Una ventaja importante del modelo REST es que se basa en estándares abiertos y no vincula la implementación del modelo o las aplicaciones cliente que acceden a ellos a cualquier implementación específica. Por lo tanto, un servicio web REST podría implementarse mediante Microsoft ASP.NET Core MVC y se podrían desarrollar aplicaciones cliente mediante cualquier lenguaje y conjunto de herramientas que puede generar solicitudes HTTP y analizar las respuestas HTTP.

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

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

Solicitud de la respuesta de una 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 (correcto), mientras que una solicitud que no se puede encontrar o eliminar un recurso especificado debe devolver una respuesta que incluye el código de estado HTTP 404 (no encontrado).

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 expone y usar las funciones proporcionadas por HTTP para operar con estos datos.

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

adecuada.

Para obtener más información acerca de REST, consulte diseño de API y implementación de la API.

Usar API de RESTful

La aplicación móvil de eShopOnContainers usa el patrón Model-View-ViewModel (MVVM) y los elementos del modelo de la representen patrón las entidades de dominio que se usan en la aplicación. Las clases de controlador y el repositorio en la aplicación de referencia eShopOnContainers 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 móvil y los microservicios en contenedores. La principal ventaja de usar dto para pasar los datos y recibir datos de un servicio web es mediante la transmisión de más datos en una única llamada remota, la aplicación puede reducir el número de llamadas remotas que deben realizarse.

Realización de las solicitudes web

La aplicación móvil de eShopOnContainers utiliza la `HttpClient` clase para realizar solicitudes a través de HTTP con JSON que se va a usar como el tipo de medio. Esta clase proporciona la funcionalidad para enviar de forma asincrónica las solicitudes HTTP y recibir respuestas HTTP de un URI identifica el recurso. La `HttpResponseMessage` clase representa un mensaje de respuesta HTTP recibido desde una API de REST después de realizar una solicitud HTTP. Contiene información acerca de la respuesta, incluido el código de estado, los encabezados y cualquier cuerpo. El `HttpContent` clase representa el cuerpo HTTP y encabezados de contenido, como `Content-Type` y `Content-Encoding`. El contenido se puede leer utilizando cualquiera de los `ReadAs` métodos, como `ReadAsStringAsync` y `ReadAsByteArrayAsync`, según el formato de los datos.

Realizar una solicitud GET

La `CatalogService` clase se utiliza para administrar el proceso de recuperación de datos desde el microservicio de catálogo. En el `RegisterDependencies` método en el `ViewModelLocator` (clase), el `CatalogService` clase se registra como una asignación de tipo con el `ICatalogService` tipo con el contenedor de inserción de dependencia de Autofac. A continuación, cuando una instancia de la `CatalogViewModel` se crea una clase, su constructor acepta un `ICatalogService` escribe, que se resuelve como Autofac, devolver una instancia de la `CatalogService` clase. Para obtener más información acerca de la inserción de dependencias, consulte Introducción a la inserción de dependencias.

Figura 10-1 se muestra la interacción de las clases que leen datos del catálogo desde el microservicio de catálogo para su visualización en el `CatalogView`.

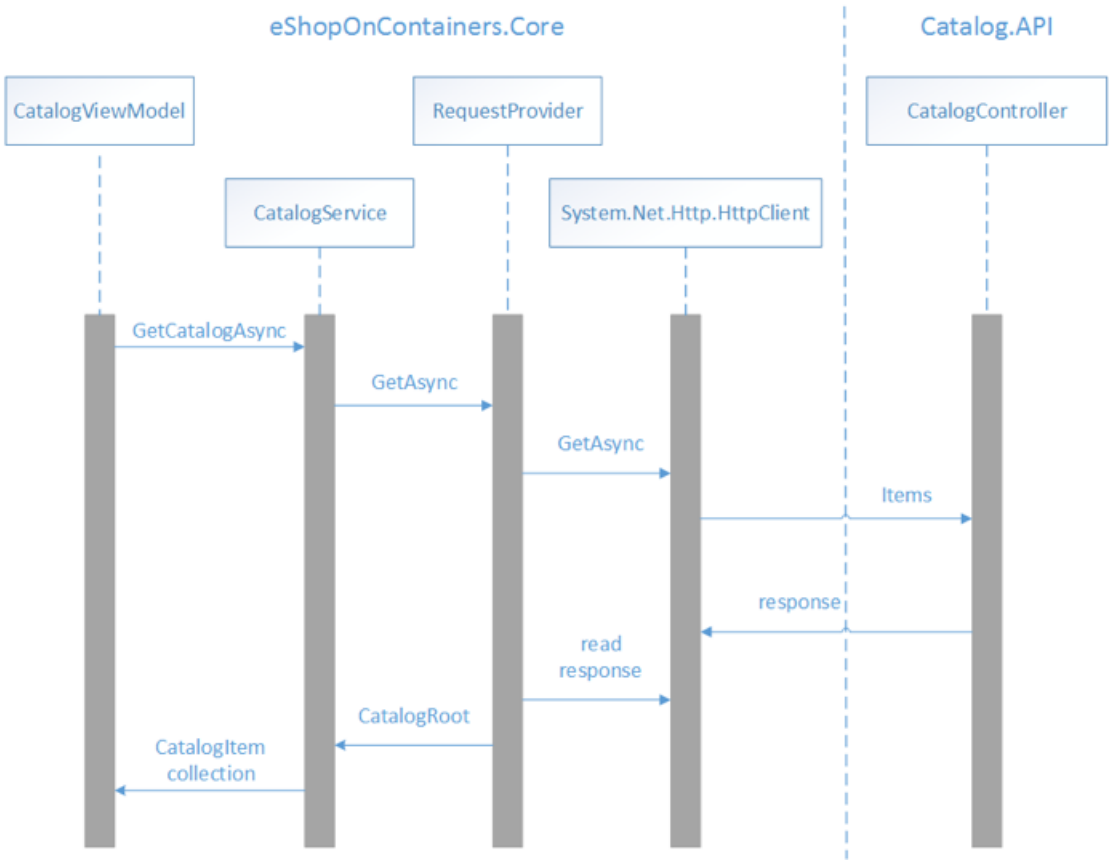

Figura 10-1: recuperación de datos desde el microservicio de catálogo

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

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

Este método llama a la `GetCatalogAsync` método de la `CatalogService` instancia insertado en el `CatalogViewModel` por Autofac. El siguiente ejemplo de código muestra la `GetCatalogAsync` método:

```
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.ToObservableCollection();
}
```

Este método genera el identificador URI que identifica el recurso de la solicitud se enviará a y utiliza el `RequestProvider` clase para invocar el método HTTP GET en el recurso, antes de devolver los resultados a la `CatalogViewModel`. La `RequestProvider` clase contiene la funcionalidad que envía una solicitud en forma de un URI que identifica un recurso, un método HTTP que indica la operación que se realizará en ese recurso, y un cuerpo que contiene todos los datos necesarios para realizar la operación. Para obtener información acerca de cómo los

`RequestProvider` clase se inserta en la `CatalogService class`, consulte Introducción a la inserción de dependencias.

El siguiente ejemplo de código muestra la `GetAsync` método en el `RequestProvider` clase:

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

    await HandleResponse(response);
    string serialized = await response.Content.ReadAsStringAsync();

    TResult result = await Task.Run(() =>
        JsonConvert.DeserializeObject<TResult>(serialized, _serializerSettings));

    return result;
}
```

Este método llama a la `CreateHttpClient` método, que devuelve una instancia de la `HttpClient` clase con el conjunto de encabezados adecuados. A continuación, envía una solicitud GET asincrónica en el recurso identificado por el identificador URI, con la respuesta que se almacenan en el `HttpResponseMessage` instancia. El `HandleResponse`, a continuación, se invoca el método, que produce una excepción si la respuesta no incluye un código de estado HTTP correcto. A continuación, se lee la respuesta como una cadena, convertida a partir de JSON a un `CatalogRoot` de objetos y devuelve al `CatalogService`.

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

```
private HttpClient CreateHttpClient(string token = "")
{
    var httpClient = new HttpClient();
    httpClient.DefaultRequestHeaders.Accept.Add(
        new MediaTypeWithQualityHeaderValue("application/json"));

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

Este método crea una nueva instancia de la `HttpClient` clase y se establece la `Accept` encabezado de las solicitudes realizadas por el `HttpClient` instancia a `application/json`, lo que indica que espera que el contenido de cualquier respuesta tenga el formato mediante JSON. A continuación, si un token de acceso que se pasó como argumento a la `CreateHttpClient` método, se agrega a la `Authorization` encabezado de las solicitudes realizadas por el `HttpClient` instancia, el prefijo con la cadena `Bearer`. Para obtener más información acerca de la autorización, consulte autorización.

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

```
[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 SQL database mediante Entity Framework y lo devuelve como un mensaje de respuesta que incluye un código de estado HTTP correcto y una colección de JSON con formato `CatalogItem` instancias.

Realizar una solicitud POST

La `BasketService` clase se utiliza para administrar la recuperación de datos y el proceso de actualización con el microservicio de cesta. En el `RegisterDependencies` método en el `ViewModelLocator` (clase), el `BasketService` clase se registra como una asignación de tipo con el `IBasketService` tipo con el contenedor de inserción de dependencia de Autofac. A continuación, cuando una instancia de la `BasketViewModel` se crea una clase, su constructor acepta un `IBasketService` escribe, que se resuelve como Autofac, devolver una instancia de la `BasketService` clase. Para obtener más información acerca de la inserción de dependencias, consulte Introducción a la inserción de dependencias.

Figura 10-2 se muestra la interacción de las clases que envían los datos de la cesta de compras mostrados por la `BasketView`, el microservicio de cesta de la compra.

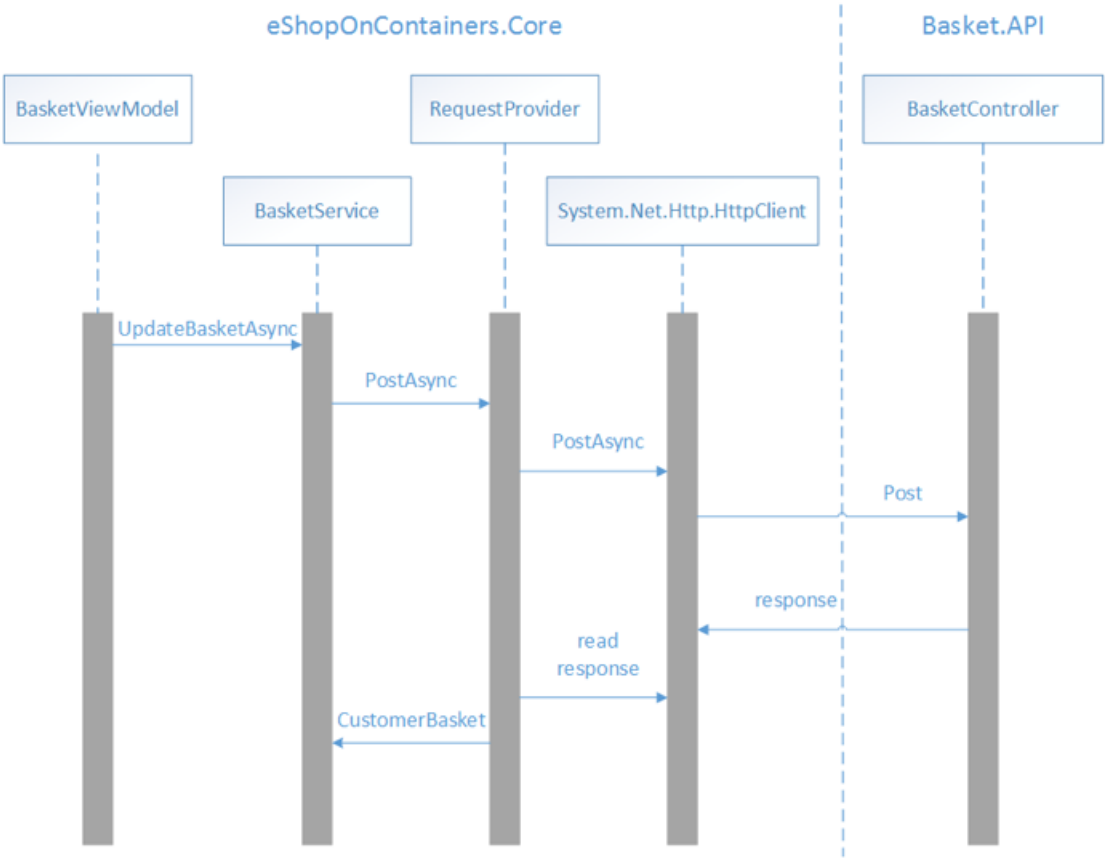

Figura 10-2: envío de datos para el microservicio de cesta

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

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

Este método llama a la `UpdateBasketAsync` método de la `BasketService` instancia insertado en el `BasketViewModel` por Autofac. El método siguiente se muestra el `UpdateBasketAsync` método:

```
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 genera el identificador URI que identifica el recurso de la solicitud se enviará a y utiliza el `RequestProvider` clase para invocar el método HTTP POST en el recurso, antes de devolver los resultados a la `BasketViewModel`. Tenga en cuenta que un token de acceso obtenida IdentityServer durante el proceso de autenticación, es necesario para autorizar las solicitudes para el microservicio de cesta. Para obtener más información acerca de la autorización, consulte autorización.

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

```
public async Task<TResult> PostAsync<TResult>(
    string uri, TResult data, string token = "", string header = "")
{
    HttpClient httpClient = CreateHttpClient(token);
    ...
    var content = new StringContent(JsonConvert.SerializeObject(data));
    content.Headers.ContentType = new MediaTypeHeaderValue("application/json");
    HttpResponseMessage response = await httpClient.PostAsync(uri, content);

    await HandleResponse(response);
    string serialized = await response.Content.ReadAsStringAsync();

    TResult result = await Task.Run(() =>
        JsonConvert.DeserializeObject<TResult>(serialized, _serializerSettings));

    return result;
}
```

Este método llama a la `CreateHttpClient` método, que devuelve una instancia de la `HttpClient` clase con el conjunto de encabezados adecuados. A continuación, envía una solicitud POST asincrónica en el recurso identificado por el identificador URI, con los datos serializados de la cesta de compras que se envían en formato JSON y la respuesta que se almacenan en el `HttpResponseMessage` instancia. El `HandleResponse` , a continuación, se invoca el método, que produce una excepción si la respuesta no incluye un código de estado HTTP correcto. A continuación, se lee la respuesta como una cadena, convertida a partir de JSON a un `CustomerBasket` de objetos y devuelve a la `BasketService`. Para obtener más información sobre la `CreateHttpClient` método, consulte realizar una solicitud GET de.

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

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

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

Realizar una solicitud DELETE

Figura 10-3 se muestran las interacciones de clases que eliminarán datos de la cesta de compras desde el microservicio de cesta de la `CheckoutView`.

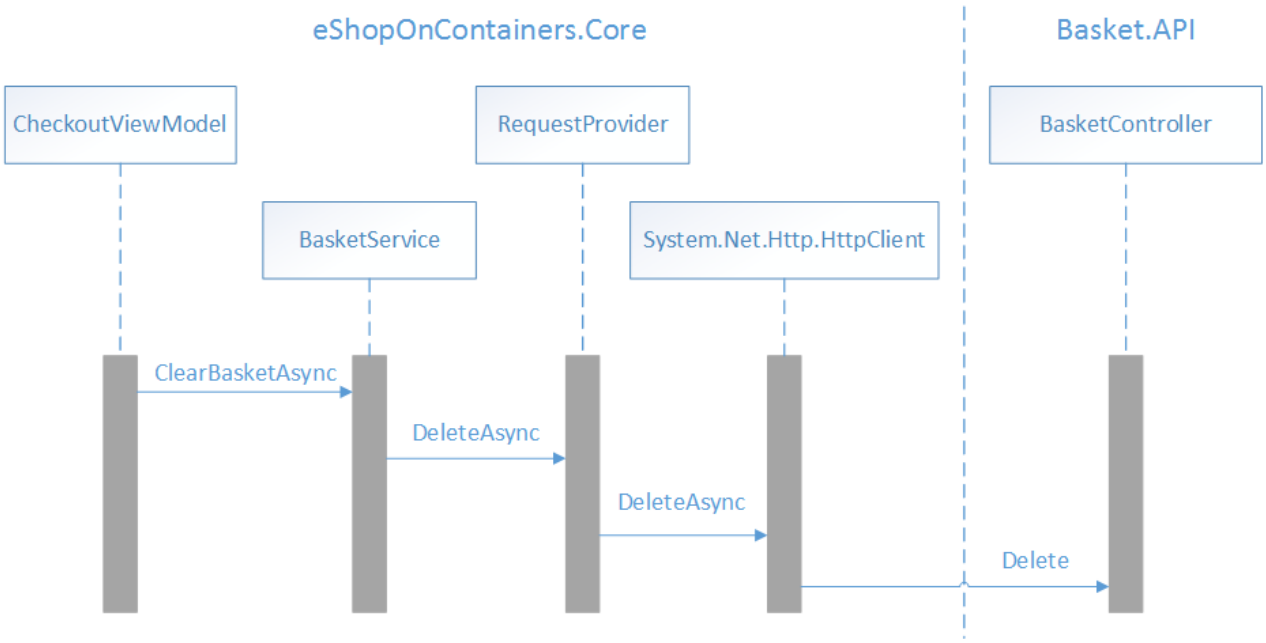

Figura 10-3: eliminar datos en el microservicio de cesta

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

```
private async Task CheckoutAsync()
{
    ...
    await _basketService.ClearBasketAsync(_shippingAddress.Id.ToString(), authToken);
    ...
}
```

Este método llama a la `ClearBasketAsync` método de la `BasketService` instancia insertado en el `CheckoutViewModel` por Autofac. El método siguiente se muestra el `ClearBasketAsync` método:

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

Este método genera el identificador URI que identifica el recurso que se enviará la solicitud a y utiliza el `RequestProvider` clase para invocar el método HTTP DELETE en el recurso. Tenga en cuenta que un token de acceso obtenida IdentityServer durante el proceso de autenticación, es necesario para autorizar las solicitudes para el microservicio de cesta. Para obtener más información acerca de la autorización, consulte autorización.

El siguiente ejemplo de código muestra la `DeleteAsync` método en el `RequestProvider` clase:

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

Este método llama a la `CreateHttpClient` método, que devuelve una instancia de la `HttpClient` clase con el

conjunto de encabezados adecuados. A continuación, envía una solicitud de eliminación asincrónica en el recurso identificado por el URI. Para obtener más información sobre la `CreateHttpClient` método, consulte realizar una solicitud GET de.

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

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

Este método utiliza una instancia de la `RedisBasketRepository` clase para eliminar los datos de la cesta de compras desde la caché en Redis.

Almacenar datos en caché

Se puede mejorar el rendimiento de una aplicación almacenando en caché datos de acceso con frecuencia en almacenamiento rápido ubicado cerca la aplicación. Si el almacenamiento rápido se encuentra más cerca de la aplicación que el origen original, el almacenamiento en caché puede mejorar significativamente la respuesta si se agota el tiempo de cuando se recuperan datos.

La forma más común de almacenamiento en caché es el almacenamiento en caché plano, donde una aplicación recupera los datos haciendo referencia a la memoria caché. Si los datos no están en la memoria caché, ha recuperado desde el almacén de datos y agrega a la caché. Pueden implementar aplicaciones de almacenamiento en caché de lectura simultánea con el patrón cache-aside. Este patrón se determina si el elemento está actualmente en la memoria caché. Si el elemento no está en la memoria caché, ha leído desde el almacén de datos y agrega a la caché. Para obtener más información, consulte el Cache-Aside patrón.

> **TIP**
> Almacenar en caché los datos que se leen con frecuencia y que cambian con poca frecuencia. Estos datos se pueden agregar a la memoria caché a petición la primera vez que se recupera mediante una aplicación. Esto significa que la aplicación debe capturar los datos de una sola vez desde el almacén de datos, y que el posterior acceso se puede satisfacer mediante el uso de la memoria caché.

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

- Una memoria caché compartida, que se puede acceder mediante varios procesos o equipos.
- Una caché privada, donde los datos se guardan localmente en el dispositivo que ejecuta la aplicación.

La aplicación móvil de eShopOnContainers utiliza una caché privada, donde los datos se guardan localmente en el dispositivo que se está ejecutando una instancia de la aplicación. Para obtener información acerca de la memoria caché utilizada por la aplicación de referencia eShopOnContainers, consulte Microservicios de. NET: arquitectura para aplicaciones .NET en contenedor.

> **TIP**
> Piense en la memoria caché como un almacén de datos transitorios que podría desaparecer en cualquier momento. Asegúrese de que los datos se mantienen en el almacén de datos original, así como la memoria caché. A continuación, se minimiza el riesgo de perder datos si la caché deja de estar disponible.

Administrar la expiración de los datos

No resulta práctico que puede esperar que los datos almacenados en caché siempre será coherentes con los datos originales. Datos de almacén de datos original pueden cambiar después de se haya almacenado en caché, haciendo que los datos almacenados en caché se vuelvan obsoletos. Por lo tanto, las aplicaciones deben implementar una estrategia que ayuda a garantizar que los datos en la memoria caché son tan actualizados como sea posible, pero también puede detectar y tratar situaciones que surgen cuando los datos en la memoria caché ha quedado obsoletos. Mecanismos de almacenamiento en caché más habilitar la memoria caché deberá estar configurado para caducar los datos y, por lo tanto, reducen el período para el que los datos podrían ser actualizados.

> **TIP**
> Establecer una expiración predeterminada al configurar una memoria caché de tiempo. Muchas de las memorias caché implementan expiración, que invalida los datos y lo quita de la memoria caché si no se accede durante un período especificado. Sin embargo, debe tener cuidado al elegir el período de expiración. Si se realiza es demasiado corta, datos expirará demasiado rápido y se reducirá las ventajas del almacenamiento en caché. Si se realiza demasiado largo, los riesgos de datos que se vuelvan obsoletos. Por lo tanto, la hora de expiración debe coincidir con el patrón de acceso para las aplicaciones que usan los datos.

Cuando los datos almacenados en caché expiran, se debe quitar de la memoria caché y la aplicación debe recuperar los datos de los datos originales almacenarán y lo colocan en la caché.

También es posible que una memoria caché se rellene si se permiten datos va a permanecer durante un período demasiado largo. Por lo tanto, las solicitudes para agregar nuevos elementos a la memoria caché podrían ser necesario quitar algunos elementos en un proceso conocido como *expulsión*. Servicios de almacenamiento en caché normalmente expulsan los datos de forma usados menos recientemente. Sin embargo, hay otras directivas de expulsión, incluidos los usados más recientemente y primero en el primero en salir. Para obtener más información, consulte almacenamiento en caché de instrucciones.

Almacenamiento en caché de imágenes

La aplicación móvil de eShopOnContainers consume las imágenes de producto remoto que se benefician del almacenamiento en caché. Estas imágenes se muestran mediante el `Image` control y el `CachedImage` control proporcionado por el FFImageLoading biblioteca.

Xamarin.Forms `Image` control admite el almacenamiento en caché de imágenes descargadas. Almacenamiento en caché está habilitado de forma predeterminada y almacenará la imagen localmente durante 24 horas. Además, la hora de expiración se puede configurar con el `CacheValidity` propiedad. Para obtener más información, consulte descargar imagen de almacenamiento en caché.

Del FFImageLoading `CachedImage` control es un sustituto de Xamarin.Forms `Image` control, que proporciona propiedades adicionales que habilitan la funcionalidad adicional. Entre esta funcionalidad, el control proporciona configurable de almacenamiento en caché, mientras que admiten el error y cargar los marcadores de posición de imagen. El ejemplo de código siguiente muestra cómo se usa la aplicación móvil de eShopOnContainers el `CachedImage` controlar en el `ProductTemplate`, que es la plantilla de datos utilizada por el `ListView` controlar en el `CatalogView`:

```
<ffimageloading:CachedImage
    Grid.Row="0"
    Source="{Binding PictureUri}"
    Aspect="AspectFill">
    <ffimageloading:CachedImage.LoadingPlaceholder>
        <OnPlatform x:TypeArguments="ImageSource">
            <On Platform="iOS, Android" Value="default_campaign" />
            <On Platform="UWP" Value="Assets/default_campaign.png" />
        </OnPlatform>
    </ffimageloading:CachedImage.LoadingPlaceholder>
    <ffimageloading:CachedImage.ErrorPlaceholder>
        <OnPlatform x:TypeArguments="ImageSource">
            <On Platform="iOS, Android" Value="noimage" />
            <On Platform="UWP" Value="Assets/noimage.png" />
        </OnPlatform>
    </ffimageloading:CachedImage.ErrorPlaceholder>
</ffimageloading:CachedImage>
```

El `CachedImage` conjuntos de controles el `LoadingPlaceholder` y `ErrorPlaceholder` propiedades a las imágenes específicas de la plataforma. El `LoadingPlaceholder` propiedad especifica la imagen que se mostrará mientras la imagen especificada por el `Source` se recupera la propiedad y el `ErrorPlaceholder` propiedad especifica la imagen que se mostrará si se produce un error al intentar recuperar la imagen especificado por el `Source` propiedad.

Como el nombre implica, el `CachedImage` control almacena en caché remotas imágenes en el dispositivo durante el tiempo especificado por el valor de la `CacheDuration` propiedad. Cuando este valor de propiedad no se establece explícitamente, se aplica el valor predeterminado de 30 días.

Aumentar 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 corrección automática, y si se repite la acción tras un retraso adecuado es probable que tenga éxito.

Los errores transitorios pueden tener un impacto enorme 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 los servicios remoto funcione de manera confiable, debe ser capaz de hacerlo siguiente:

- Detectar errores cuando se producen y determinar si los errores suelen ser transitorio.
- Si determina que el error es probable que sea transitorio y realizar un seguimiento de la cantidad de veces que se vuelve a intentar la operación, vuelva a intentar la operación.
- Usar una estrategia adecuada de reintentos, que especifica el número de reintentos, el retraso entre cada intento y las acciones a realizar después de un intento fallido.

Este control de errores transitorios puede lograrse ajustando todos los intentos de acceso 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 en cualquiera de las maneras siguientes:

- Volver a intentar la operación. La aplicación se vuelva a intentar la solicitud con error inmediatamente.
- Reintentando la operación después de un retraso. La aplicación debe esperar una cantidad adecuada de tiempo antes de reintentar la solicitud.
- Cancelando la operación. La aplicación debe cancelar la operación y notificar una excepción.

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

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

Si una solicitud sigue sin funcionar después de un número de reintentos, es mejor para la aplicación para evitar que las solicitudes posteriores en el mismo recurso y para notificar un error. A continuación, tras un período establecido, la aplicación puede realizar una o varias solicitudes al recurso para ver si son correctas. Para obtener más información, consulte patrón Circuit Breaker.

> **TIP**
> Nunca implemente un mecanismo de reintento infinito. Use un número finito de reintentos o implemente el disyuntor patrón para permitir que un servicio recuperar.

La aplicación móvil de eShopOnContainers implementa actualmente el patrón de reintento al realizar solicitudes web RESTful. Sin embargo, el `CachedImage` control, que proporciona el FFImageLoading biblioteca admite el control de errores transitorios Reintentando la carga de la imagen. Si se produce un error en la carga de imágenes, se realizarán más intentos. El número de intentos especificado por el `RetryCount` propiedad y reintentos, se producirán después de un retraso especificado por el `RetryDelay` propiedad. Si estos valores de propiedad no se establecen explícitamente, su valor predeterminado es se aplican los valores: 3 para el `RetryCount` propiedad y 250 ms para el `RetryDelay` propiedad. Para obtener más información sobre la `CachedImage` control, vea almacenamiento en caché de imágenes.

La aplicación de referencia eShopOnContainers implementa el patrón de reintento. Para obtener más información, incluida una discusión acerca de cómo combinar el patrón de reintento con el `HttpClient` de clases, vea Microservicios de. NET: arquitectura para aplicaciones .NET en contenedor.

Para obtener más información sobre el patrón de reintento, consulte el vuelva a intentar patrón.

Patrón de disyuntor

En algunas situaciones, pueden producirse errores debidos a eventos anticipados que tardan más tiempo para corregir. Estos errores pueden oscilar entre una pérdida parcial de conectividad y el error completo de un servicio. En estas situaciones, es inútil para una aplicación volver a intentar una operación que no es probable que se realiza correctamente y en su lugar, debe aceptar que la operación ha fallado y controlar este error en consecuencia.

El patrón circuit breaker puede impedir que una aplicación intente repetidamente ejecutar una operación que es propenso a errores, al mismo tiempo que la aplicación detectar si se ha resuelto el error.

> **NOTE**
> El propósito del patrón de disyuntor es diferente del patrón de reintento. El patrón de reintento permite que una aplicación reintentar la operación con la expectativa de que se podrá ejecutar correctamente. El patrón de interruptor impide que una aplicación realizando una operación que es propenso a errores.

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

La aplicación móvil de eShopOnContainers no implementa el patrón circuit breaker actualmente. Sin embargo, no el de eShopOnContainers. Para obtener más información, consulte Microservicios de. NET: arquitectura para aplicaciones .NET en contenedor.

> **TIP**
> Combinar los patrones de interruptor y reinténtelo. Una aplicación puede combinar los patrones de reintento y el interruptor de circuito mediante el patrón de reintento para invocar una operación a través de un disyuntor. Sin embargo, la lógica de reintento debe ser sensible a las excepciones devueltas por el disyuntor y dejar de reintentar si el interruptor indica que un error no es transitorio.

Para obtener más información sobre el patrón de disyuntor, consulte el disyuntor patrón.

Resumen

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

Se puede mejorar el rendimiento de una aplicación almacenando en caché datos de acceso con frecuencia en almacenamiento rápido ubicado cerca la aplicación. Pueden implementar aplicaciones de almacenamiento en caché de lectura simultánea con el patrón cache-aside. Este patrón se determina si el elemento está actualmente en la memoria caché. Si el elemento no está en la memoria caché, ha leído desde el almacén de datos y agrega a la caché.

Cuando se comunica 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 corrección automática, y si se repite la acción tras un retraso adecuado, a continuación, es probable que se realice correctamente. Por lo tanto, las aplicaciones deben ajustar todos los intentos de acceso a una API web en el código que implementa un mecanismo de control de errores transitorios.

Vínculos relacionados

- Descargar libro electrónico (PDF de 2Mb)
- eShopOnContainers (GitHub) (ejemplo)

Pruebas unitarias de aplicaciones de Enterprise

13/07/2018 • 17 minutes to read • Edit Online

Aplicaciones móviles tienen problemas únicos que escritorio y aplicaciones basadas en web no tienen que preocuparse. Los usuarios móviles variarán los dispositivos que está usando, conectividad de red, la disponibilidad de servicios y un intervalo de otros factores. Por lo tanto, se deben probar aplicaciones móviles que se usará en el mundo real para mejorar su calidad, confiabilidad y rendimiento. Hay muchos tipos de pruebas que deben realizarse en una aplicación, incluidas las pruebas unitarias, pruebas de integración y pruebas, con las pruebas que se va a la forma más común de las pruebas unitarias de interfaz de usuario.

Una prueba unitaria toma una unidad pequeña de la aplicación, normalmente un método, aísla del resto del código y comprueba si se comporta según lo previsto. Su objetivo es comprobar que cada unidad de funcionalidad se ejecuta según lo previsto, así que los errores no propagan a través de la aplicación. Detectar un error en el que ocurre es más eficaz que observar el efecto de un error indirectamente en un punto de error secundario.

Las pruebas unitarias tienen el mayor efecto en la calidad del código cuando es una parte integral del flujo de trabajo de desarrollo de software. Tan pronto como se ha escrito un método, se deben escribir pruebas unitarias que comprueben el comportamiento del método en respuesta a estándar, límite e incorrectos de los casos de datos de entrada y dicha comprobación cualquier suposición explícita o implícita creada por el código. Como alternativa, con el desarrollo controlado por pruebas, pruebas unitarias se escriben antes del código. En este escenario, las pruebas unitarias actúan como documentación de diseño y las especificaciones funcionales.

> **NOTE**
> Las pruebas unitarias son muy eficaces contra la regresión: es decir, funcionalidad que funcionan pero ha sido ve perturbada por una actualización fallida.

Pruebas unitarias suelen usan el patrón assert para organizar act:

- El *organizar* sección del método de prueba unitaria inicializa objetos y establece el valor de los datos que se pasan al método sometido a prueba.
- El *actuar* sección invoca el método sometido a prueba con los argumentos necesarios.
- El *assert* sección comprueba si la acción del método sometido a prueba se comporta según lo previsto.

Si sigue este patrón garantiza que las pruebas unitarias son legibles y coherente.

Inserción de dependencias y las 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 Autofac es la `OrderService` clase. En el ejemplo de código siguiente se muestra un esquema de esta clase:

```
public class OrderDetailViewModel : ViewModelBase
{
    private IOrderService _ordersService;

    public OrderDetailViewModel(IOrderService ordersService)
    {
        _ordersService = ordersService;
    }
    ...
}
```

El `OrderDetailViewModel` clase tiene una dependencia en el `IOrderService` escriba que el contenedor resuelve cuando crea una instancia de un `OrderDetailViewModel` objeto. Sin embargo, en lugar de crear un `OrderService` objeto pruebas unitarias para el `OrderDetailViewModel` (clase), en su lugar, reemplace el `OrderService` objeto con una simulación con el fin de las pruebas. Figura 10-1 se ilustra esta relación.

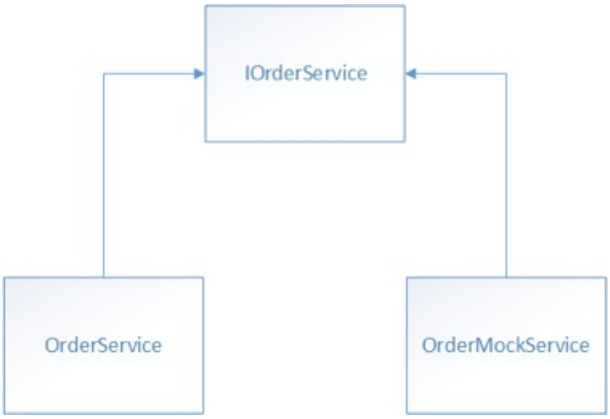

Figura 10-1: las clases que implementan la interfaz IOrderService

Este enfoque permite la `OrderService` objeto que se pasan los `OrderDetailViewModel` de clase en tiempo de ejecución y en aras de la capacidad de prueba, permite el `OrderMockService` clase que se pasan la `OrderDetailViewModel` clase en tiempo de prueba. La principal ventaja de este enfoque es que permite que las pruebas unitarias que se ejecutará sin necesidad de difícil de manejar recursos como servicios web o bases de datos.

Probar las aplicaciones MVVM

Probar los modelos y los modelos de vista de las aplicaciones MVVM es idéntica a las pruebas de otras clases y las mismas herramientas y técnicas, como la prueba unitaria y simulación, se pueden usar. Sin embargo, hay algunos patrones típicos de modelo y las clases de modelo de vista, que pueden beneficiarse de las técnicas de pruebas de unidad específica.

> **TIP**
> Probar una cosa con cada prueba unitaria. No tener la tentación de hacer que una unidad de prueba más de uno de los aspectos del comportamiento de la unidad de ejercicio. Esto puede dar lugar a las pruebas que son difíciles de leer y actualizar. También puede provocar confusión al interpretar un error.

Los usos de la aplicación móvil de eShopOnContainers xUnit para realizar las pruebas unitarias, que admite dos tipos diferentes de las pruebas unitarias:

- Los hechos son las pruebas que están siempre es true, que prueba las condiciones invariantes.
- Teorías son pruebas que sólo son true para un determinado conjunto de datos.

Las pruebas unitarias incluidas con la aplicación móvil de eShopOnContainers son pruebas de hechos y, por lo

que cada método de prueba unitaria está decorada con el `[Fact]` atributo.

> **NOTE**
> un ejecutor de pruebas ejecuta las pruebas de xUnit. Para ejecutar el ejecutor de pruebas, ejecute el proyecto eShopOnContainers.TestRunner para la plataforma requerida.

Probar la funcionalidad asincrónica

Al implementar el patrón MVVM, los modelos de vista normalmente invocan operaciones en servicios, a menudo forma asincrónica. Pruebas para el código que invoca normalmente estas operaciones utilizan objetos ficticios como reemplazos para los servicios reales. El siguiente ejemplo de código muestra cómo probar la funcionalidad asincrónica al pasar de un servicio ficticio a un modelo de vista:

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

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

    Assert.NotNull(orderViewModel.Order);
}
```

Esta prueba unitaria que comprueba la `Order` propiedad de la `OrderDetailViewModel` instancia tendrá un valor después de la `InitializeAsync` ha invocado al método. El `InitializeAsync` método se invoca cuando se navega vista correspondiente del modelo de vista. Para obtener más información sobre la navegación, consulte navegación.

Cuando el `OrderDetailViewModel` se crea una instancia, espera un `OrderService` instancia que se especifique como argumento. Sin embargo, el `OrderService` recupera datos de un servicio web. Por lo tanto, un `OrderMockService` instancia, que es una versión ficticia de la `OrderService` de clase, se especifica como argumento para el `OrderDetailViewModel` constructor. Después, cuando el modelo de vista `InitializeAsync` se invoca el método, que invoca `IOrderService` operaciones, datos simulados están recuperado en lugar de establecer la comunicación con un servicio web.

Probar las implementaciones de INotifyPropertyChanged

Implementar el `INotifyPropertyChanged` interfaz permite que las vistas para reaccionar ante los cambios que se originan en la vista de modelos 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 de modelo de vista que hacen que se puede iniciar las animaciones o controles va a deshabilitar.

Se pueden probar las propiedades que se pueden actualizar directamente por la prueba unitaria adjuntando un controlador de eventos para el `PropertyChanged` eventos y comprobar si el evento se genera después de establecer un nuevo valor para la propiedad. El ejemplo de código siguiente muestra una prueba de este tipo:

```
[Fact]
public async Task SettingOrderPropertyShouldRaisePropertyChanged()
{
    bool 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.True(invoked);
}
```

Esta prueba unitaria invoca el `InitializeAsync` método de la `OrderViewModel` clase, lo que hace que su `Order` propiedad que se va a actualizarse. Pasará la prueba unitaria, siempre que el `PropertyChanged` evento se desencadena para el `Order` propiedad.

Probar la comunicación basada en mensajes

Vista de modelos que usan el `MessagingCenter` clase para la comunicación entre el acoplamiento de clases puede unidad probarse suscribiéndose a los mensajes enviados por el código sometido a prueba, como se muestra en el ejemplo de código siguiente:

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

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

    Assert.True(messageReceived);
}
```

Esta prueba unitaria que comprueba la `CatalogViewModel` publica el `AddProduct` mensaje de respuesta a su `AddCatalogItemCommand` que se está ejecutando. Dado que el `MessagingCenter` clase es compatible con suscripciones de mensajes de multidifusión, la prueba unitaria puede suscribirse a la `AddProduct` del mensaje y ejecutar un delegado de devolución de llamada en respuesta a recibirlo. Este delegado de devolución de llamada, especificado como una expresión lambda, Establece una `boolean` campo utilizado por el `Assert` instrucción para comprobar el comportamiento de la prueba.

Las pruebas de control de excepciones

Pruebas unitarias también se pueden escribir esa comprobación que se producen excepciones específicas de las acciones no válidas o entradas, como se muestra en el ejemplo de código siguiente:

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

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

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

> **Sugerencia**: evitar escribir pruebas unitarias que examine las cadenas de mensaje de excepción. Las cadenas de mensaje de excepción pueden cambiar con el tiempo y, por lo que se consideran las pruebas unitarias que se basan en su presencia frágiles.

Las pruebas de validación

Hay dos aspectos a la implementación de la validación de pruebas: pruebas que se haya implementado correctamente las reglas de validación y las pruebas que el `ValidatableObject<T>` clase se realiza según lo previsto.

Lógica de validación es normalmente más simple probar, porque normalmente es un proceso independiente, donde el resultado depende de la entrada. Debe haber pruebas en los resultados de invocar el `Validate` método en cada propiedad que tiene al menos una regla de validación asociado, como se muestra en el ejemplo de código siguiente:

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

    bool isValid = mockViewModel.Validate();

    Assert.True(isValid);
}
```

Esta prueba unitaria comprueba que la validación es correcta cuando los dos `ValidatableObject<T>` propiedades en el `MockViewModel` instancia ambos tienen datos.

Así como la comprobación de que la validación se realiza correctamente, las pruebas unitarias de validación también deberían comprobar los valores de la `Value`, `IsValid`, y `Errors` propiedad de cada uno `ValidatableObject<T>` instancia, para comprobar que la clase se realiza según lo previsto. El ejemplo de código siguiente muestra una prueba unitaria que se hace esto:

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

    bool isValid = mockViewModel.Validate();

    Assert.False(isValid);
    Assert.NotNull(mockViewModel.Forename.Value);
    Assert.Null(mockViewModel.Surname.Value);
    Assert.True(mockViewModel.Forename.IsValid);
    Assert.False(mockViewModel.Surname.IsValid);
    Assert.Empty(mockViewModel.Forename.Errors);
    Assert.NotEmpty(mockViewModel.Surname.Errors);
}
```

Esta prueba unitaria comprueba que falla la validación cuando el `Surname` propiedad de la `MockViewModel` no tiene ningún dato y el `Value`, `IsValid`, y `Errors` propiedad de cada uno `ValidatableObject<T>` instancia se han definido correctamente.

Resumen

Una prueba unitaria toma una unidad pequeña de la aplicación, normalmente un método, aísla del resto del código y comprueba si se comporta según lo previsto. Su objetivo es comprobar que cada unidad de funcionalidad se ejecuta según lo previsto, así que los errores no propagan a través de la aplicación.

El comportamiento de un objeto sometido a prueba se puede aislar mediante la sustitución de los objetos dependientes con objetos ficticios que simulan el comportamiento de los objetos dependientes. Esto permite que las pruebas unitarias que se ejecutará sin necesidad de difícil de manejar recursos como servicios web o bases de datos.

Probar los modelos y los modelos de vista de las aplicaciones MVVM es idéntica a las pruebas de otras clases, y se pueden usar las mismas herramientas y técnicas.

Vínculos relacionados

- Descargar libro electrónico (PDF de 2Mb)
- eShopOnContainers (GitHub) (ejemplo)