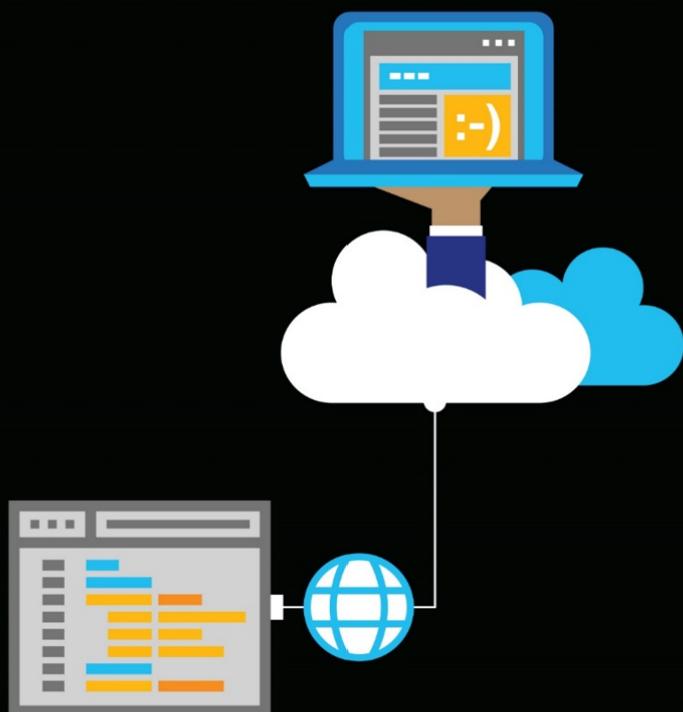


# Diseño de aplicaciones web modernas con ASP.NET Core y Azure

Artículo • 15/02/2023 • Tiempo de lectura: 4 minutos



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



Steve “ardalis” Smith

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

PUBLICADO POR

Equipos de producto de la División de desarrolladores de Microsoft, .NET y Visual Studio

División de Microsoft Corporation

One Microsoft Way

Redmond, Washington 98052-6399

Copyright © 2022 de Microsoft Corporation

Todos los derechos reservados. No se puede reproducir ni transmitir de ninguna forma ni por ningún medio ninguna parte del contenido de este libro sin la autorización por escrito del publicador.

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

Algunos ejemplos descritos aquí se proporcionan únicamente con fines ilustrativos y son ficticios. No debe deducirse ninguna asociación ni conexión reales.

Microsoft y las marcas comerciales indicadas en <https://www.microsoft.com> en la página web "Marcas comerciales" pertenecen al grupo de empresas de Microsoft.

Mac y macOS son marcas comerciales de Apple Inc.

El logotipo de la ballena de Docker es una marca registrada de Docker, Inc. Se usa con permiso.

El resto de marcas y logotipos pertenece a sus respectivos propietarios.

Autor:

Steve "ardalis" Smith: instructor y arquitecto de software de [Ardalis.com](http://Ardalis.com)

Editores:

Maira Wenzel

## Vínculos de acción

- Este libro electrónico también está disponible en formato PDF (solo versión en inglés) [Descargar ↗](#)
- Clonación o bifurcación de la aplicación de referencia [eShopOnWeb on GitHub ↗](#)

## Introducción

.NET 6 y ASP.NET Core ofrecen varias ventajas con respecto al desarrollo tradicional con .NET. Debe usar .NET 6 para las aplicaciones de servidor si algunos o todos los elementos siguientes son importantes para el éxito de su aplicación:

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

Las aplicaciones de .NET 4.x son compatibles con muchos de estos requisitos, pero ASP.NET Core y .NET 6 se han optimizado para ofrecer una compatibilidad mejorada para los escenarios anteriores.

Cada vez más organizaciones deciden hospedar sus aplicaciones web en la nube con servicios como Microsoft Azure. Considere la posibilidad de hospedar su aplicación en la nube si los siguientes elementos son importantes para la aplicación o la organización:

- Inversión reducida en costos de centros de datos (hardware, software, espacio, utilidades, administración de servidores, etc.).
- Precios flexibles (pago en función del uso, y no por la capacidad inactiva).
- Confiabilidad extrema.
- Movilidad mejorada de la aplicación; modificación sencilla de dónde y cómo implementar la aplicación.
- Capacidad flexible; escalado o reducción vertical en función de las necesidades reales.

La compilación de aplicaciones web con ASP.NET Core, hospedadas en Azure, ofrece muchas ventajas competitivas con respecto a las alternativas tradicionales. Se ha

optimizado ASP.NET Core para escenarios de hospedaje en la nube y prácticas de desarrollo de aplicaciones web modernas. En esta guía se ofrece información sobre cómo diseñar aplicaciones con ASP.NET Core para sacar el máximo provecho de estas funcionalidades.

## Versión

Esta guía se ha revisado para abarcar la versión .NET 6.0 junto con muchas actualizaciones adicionales relacionadas con la misma "oleada" de tecnologías (es decir, Azure y otras tecnologías de terceros) que coincidan en el tiempo con la versión de .NET 6.0. Este es el motivo por el que la versión del libro se ha actualizado también a la versión 6.0.

## Propósito

En esta guía se proporcionan instrucciones integrales sobre cómo compilar aplicaciones web *monolíticas* con ASP.NET Core y Azure. En este contexto, "monolíticas" hace referencia al hecho de que estas aplicaciones se implementan como una sola unidad, no como una colección de aplicaciones y servicios que interactúan. En algunos contextos, el término *monolítica* se puede usar de forma peyorativa, pero en la gran mayoría de las situaciones una sola aplicación es mucho más fácil de compilar, implementar y depurar que una aplicación compuesta de muchos servicios diferentes, sin tener que comprometer el cumplimiento de los requisitos empresariales.

Esta guía complementa los microservicios de .NET "[. Arquitectura para aplicaciones .NET en contenedor](#)" más centrada en Docker, los microservicios y la implementación de contenedores para hospedar aplicaciones empresariales.

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

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

## Destinatarios de esta guía

Los destinatarios de esta guía son principalmente desarrolladores, jefes de desarrollo y arquitectos interesados en crear aplicaciones web modernas con tecnologías y servicios

de Microsoft en la nube.

Otros destinatarios secundarios son los responsables de tomar decisiones técnicas que ya están familiarizados con ASP.NET o Azure y que buscan información sobre si tiene sentido actualizar a ASP.NET Core para los proyectos nuevos o existentes.

## Cómo leer esta guía

Esta guía se ha comprimido en un documento relativamente pequeño que se centra en la creación de aplicaciones web con modernas tecnologías de .NET y Azure. Por lo tanto, se puede leer completa para proporcionar una base de conocimiento sobre estas aplicaciones y sus consideraciones técnicas. La guía, junto con su aplicación de ejemplo, también puede servir como punto inicial o referencia. Use la aplicación de ejemplo asociada como una plantilla para las aplicaciones propias o para consultar cómo se pueden organizar los componentes de la aplicación. Consulte los principios y la cobertura de la arquitectura, las opciones tecnológicas y las consideraciones para la toma de decisiones de esta guía a la hora de sopesar estas opciones para su propia aplicación.

No dude en reenviar esta guía a su equipo para ayudarlo a garantizar una comprensión común de estas consideraciones y oportunidades. El hecho de que todos los usuarios trabajen a partir de un conjunto común de principios subyacentes y terminología permite garantizar una aplicación coherente de las prácticas y los patrones de diseño.

## Referencias

- Elección entre .NET 6 y .NET Framework para aplicaciones de servidor  
<https://docs.microsoft.com/dotnet/standard/choosing-core-framework-server>

Siguiente

# Características de las aplicaciones web modernas

Artículo • 28/11/2022 • Tiempo de lectura: 7 minutos

## 💡 Sugerencia

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

[Descargar PDF](#)



"... con el diseño adecuado, las características son más baratas. Este enfoque es complicado, pero sigue siendo útil".

- Dennis Ritchie

Las aplicaciones web modernas tienen expectativas más altas por parte del usuario y mayores demandas que nunca. Se espera que las aplicaciones web actuales estén disponibles 24 horas al día los siete días de la semana desde cualquier lugar del mundo y se puedan usar desde prácticamente cualquier dispositivo o tamaño de pantalla. Las aplicaciones web deben ser seguras, flexibles y escalables para satisfacer los picos de demanda. Cada vez más, los escenarios complejos deben controlarse mediante experiencias de usuario enriquecidas creadas en el cliente con JavaScript y comunicarse de forma eficaz a través de las API web.

ASP.NET Core se ha optimizado para aplicaciones web modernas y escenarios de hospedaje basados en la nube. Su diseño modular permite que las aplicaciones dependan solo de aquellas características que realmente usan, lo que mejora la seguridad y el rendimiento de la aplicación mientras se reducen los requisitos de recursos de hospedaje.

# Aplicación de referencia: eShopOnWeb

En esta guía se incluye una aplicación de referencia, *eShopOnWeb*, en la que se muestran algunos de los principios y las recomendaciones. La aplicación es una tienda en línea sencilla que admite búsquedas por medio de un catálogo de camisetas, tazas de café y otros productos de marketing. La aplicación de referencia es deliberadamente sencilla para que sea fácil de entender.

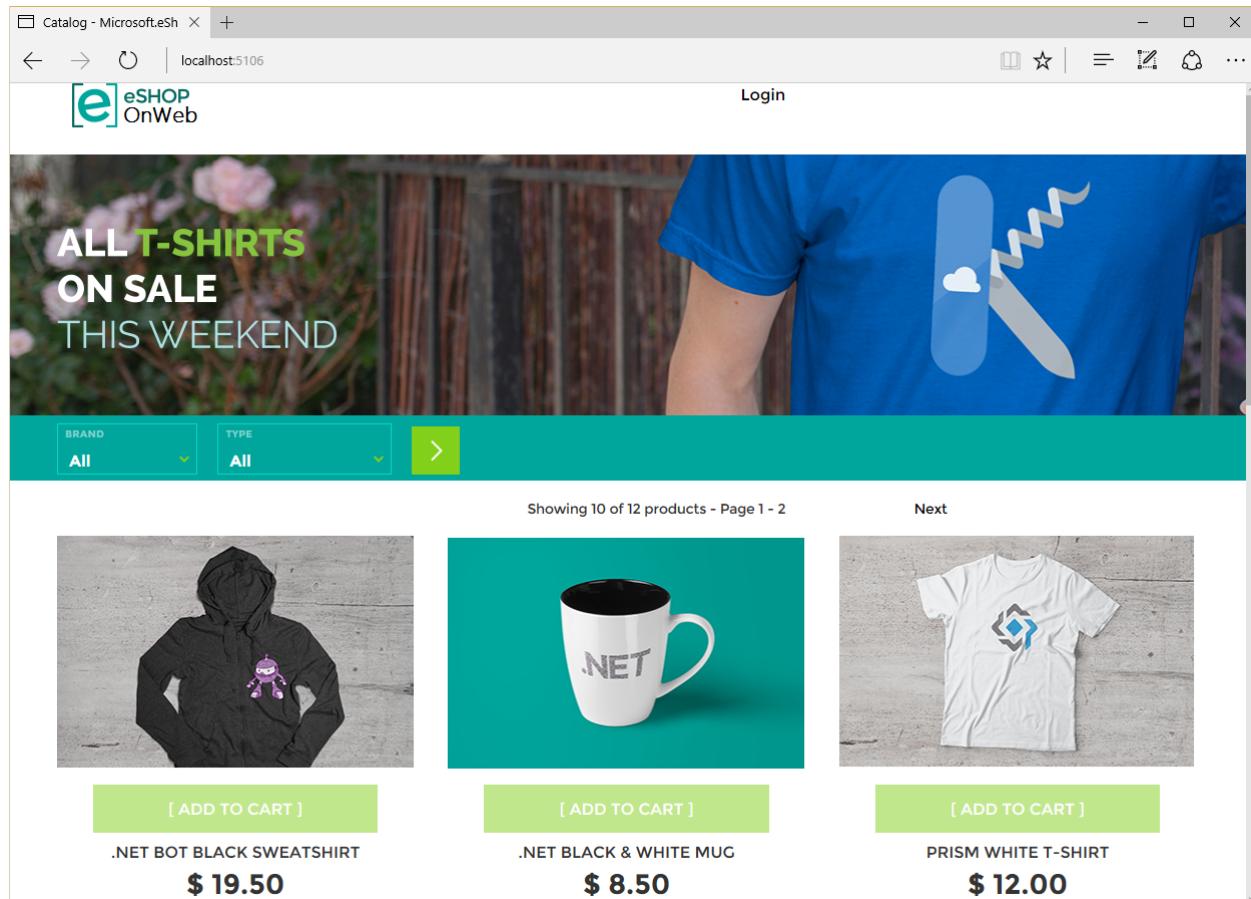


Figura 2-1. eShopOnWeb

## Aplicación de referencia

- eShopOnWeb  
[https://github.com/dotnet/eShopOnWeb ↗](https://github.com/dotnet/eShopOnWeb)

## Hospedada en la nube y escalable

ASP.NET Core se ha optimizado para la nube (pública, privada, cualquier nube) debido a su memoria baja y rendimiento alto. La superficie más pequeña de las aplicaciones ASP.NET Core significa que se puede hospedar un mayor número de ellas en el mismo hardware, y que se paga por menos recursos cuando se usan servicios de hospedaje en la nube de pago por uso. El rendimiento mayor significa que se puede servir a más

clientes desde una aplicación dado el mismo hardware, reduciendo así la necesidad de invertir en infraestructura y servidores de hospedaje.

## Multiplataforma

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

## Modular y de acoplamiento flexible

Los paquetes NuGet son objetos de primera clase en .NET Core y las aplicaciones ASP.NET Core se componen de muchas bibliotecas de NuGet. Esta granularidad de la funcionalidad ayuda a garantizar que las aplicaciones solo dependen e implementan la funcionalidad que realmente necesitan, lo que reduce su superficie y el área expuesta a vulnerabilidades de seguridad.

ASP.NET Core también es totalmente compatible con la [inserción de dependencias](#), tanto de forma interna como en la aplicación. Las interfaces pueden tener varias implementaciones que se pueden intercambiar según sea necesario. La inserción de dependencias permite acoplar aplicaciones de forma flexible a esas interfaces en lugar de a implementaciones específicas, lo que facilita la ampliación, el mantenimiento y las pruebas correspondientes.

## Pruebas sencillas con pruebas automatizadas

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

# Comportamientos tradicionales y de SPA admitidos

Las aplicaciones web tradicionales apenas contaban con comportamiento del lado cliente, y en su lugar se basaban en el servidor para todas las operaciones de navegación, consultas y actualizaciones que la aplicación tuviera que realizar. Cada operación nueva realizada por el usuario se convertiría en una nueva solicitud web, con el resultado de una recarga de página completa en el explorador del usuario final. Los marcos de controlador de vista de modelos (MVC) clásicos normalmente siguen este enfoque, en el que cada solicitud nueva se corresponde a otra acción de controlador, lo que a su vez podría funcionar con un modelo y devolver una vista. Es posible que algunas operaciones individuales en una página determinada se mejoraran con funcionalidad de AJAX (JavaScript asincrónico y XML), pero la arquitectura global de la aplicación usaba muchas vistas MVC distintas y extremos de URL. Además, ASP.NET Core MVC también admite Razor Pages, una forma más sencilla de organizar las páginas de tipo MVC.

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

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

## Implementación y desarrollo simples

Las aplicaciones de ASP.NET Core se pueden escribir mediante interfaces de línea de comandos y editores de texto simples, o bien con entornos de desarrollo completos como Visual Studio. Las aplicaciones monolíticas normalmente se implementan en un solo punto de conexión. Las implementaciones se pueden automatizar con facilidad para que tengan lugar como parte de una canalización de integración continua (CI) y entrega continua (CD). Además de las herramientas de CI/CD tradicionales, Microsoft Azure tiene compatibilidad integrada para repositorios de Git y puede

implementar automáticamente las actualizaciones que se realicen en una rama o etiqueta de Git especificada. Azure DevOps proporciona una canalización de compilación e implementación de CI/CD con todas las características; por su parte, Acciones de GitHub proporciona otra opción para los proyectos hospedados allí.

## ASP.NET tradicional y formularios Web Forms

Además de ASP.NET Core, ASP.NET 4.x tradicional sigue siendo una plataforma sólida y confiable para compilar aplicaciones web. ASP.NET es compatible con los modelos de desarrollo de MVC y API web, así como los formularios Web Forms, que resultan muy adecuados para el desarrollo de aplicaciones basadas en páginas y ofrecen un ecosistema enriquecido de componentes de terceros. Desde hace tiempo, Microsoft Azure ofrece una gran compatibilidad con las aplicaciones de ASP.NET 4.x y muchos desarrolladores están familiarizados con esta plataforma.

## Blazor

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

Blazor lado servidor se lanzó en 2019 con ASP.NET Core 3.0. Como su nombre implica, se ejecuta en el servidor, lo que hace que los cambios en el documento cliente se representen en el explorador a través de la red. Blazor lado servidor proporciona una experiencia de cliente enriquecida sin requerir JavaScript del lado cliente y sin la necesidad de cargas de páginas independientes para cada interacción de la página del cliente. El servidor solicita y procesa los cambios en la página cargada y, después, estos se devuelven al cliente mediante SignalR.

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

## Referencias: aplicaciones web modernas

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

[Anterior](#)

[Siguiente](#)

# Elección entre aplicaciones web tradicionales y aplicaciones de página única (SPA)

Artículo • 28/11/2022 • Tiempo de lectura: 8 minutos

## 💡 Sugerencia

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

[Descargar PDF](#)



"Ley de Atwood: cualquier aplicación que se pueda escribir en JavaScript, se acabará escribiendo en JavaScript".

- Jeff Atwood

En la actualidad hay dos enfoques generales para compilar aplicaciones web: las aplicaciones web tradicionales que ejecutan la mayor parte de la lógica de aplicación en el servidor, y las aplicaciones de página única (SPA) que ejecutan la mayor parte de la lógica de la interfaz de usuario en un explorador web y se comunican con el servidor web principalmente mediante API web. También es posible un enfoque híbrido; el más sencillo es hospedar una o más subaplicaciones enriquecidas de tipo SPA dentro de una aplicación web tradicional más grande.

Use las aplicaciones web tradicionales en los casos siguientes:

- Los requisitos del lado cliente de la aplicación son sencillos o incluso de solo lectura.

- La aplicación necesita funcionar en exploradores que no admiten JavaScript.

Use una SPA en los casos siguientes:

- La aplicación tenga que exponer una interfaz de usuario enriquecida con muchas características.
- Su equipo está familiarizado con el desarrollo en JavaScript, TypeScript o BlazorWebAssembly.
- La aplicación ya tiene que exponer una API para otros clientes (internos o públicos).

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

Las mejoras en la experiencia del usuario que posibilita el enfoque de SPA deben ponderarse con estas consideraciones.

## Blazor

ASP.NET Core incluye un modelo para crear interfaces de usuario detalladas, interactivas y que admiten composición, el cual se denomina Blazor. El lado servidor de Blazor permite a los desarrolladores crear una interfaz de usuario con C# y Razor en el servidor, así como la conexión interactiva de la interfaz de usuario al explorador en tiempo real mediante una conexión de SignalR persistente. BlazorWebAssembly presenta otra opción para las aplicaciones Blazor, que les permite ejecutarse en el explorador mediante WebAssembly. Dado que se trata de la ejecución real de código de .NET en WebAssembly, puede volver a usar el código y las bibliotecas de las partes del lado servidor de la aplicación.

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

Considere la posibilidad de compilar la aplicación web con Blazor en los casos siguientes:

- La aplicación tiene que exponer una interfaz de usuario Enriquecida.
- Su equipo está más familiarizado con el desarrollo de .NET que con el desarrollo de JavaScript o TypeScript.

Si tiene una aplicación de Web Forms que está pensando en migrar a .NET Core o la versión más reciente de .NET, puede que quiera consultar el libro electrónico gratuito [Blazor para desarrolladores de Web Forms](#) y ver si tiene sentido migrarla a Blazor.

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

## Casos en los que elegir aplicaciones web tradicionales

En la sección siguiente se explican de manera más detallada las razones indicadas anteriormente para elegir aplicaciones web tradicionales.

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

La mayoría de los usuarios usan muchas aplicaciones web principalmente en un modo de solo lectura. Las aplicaciones de solo lectura (o principalmente de solo lectura) tienden a ser mucho más sencillas que las que mantienen y manipulan una gran cantidad de elementos de estado. Por ejemplo, es posible que un motor de búsqueda conste de un único punto de entrada con un cuadro de texto y una segunda página para mostrar los resultados de la búsqueda. Los usuarios anónimos pueden realizar solicitudes con facilidad y la necesidad de lógica del lado cliente es escasa. Del mismo modo, una aplicación de acceso público de un blog o sistema de administración de contenido suele estar compuesta principalmente de contenido con poco comportamiento del lado cliente. Estas aplicaciones se compilan fácilmente como aplicaciones web tradicionales basadas en servidor que ejecutan la lógica en el servidor web y representan HTML que se mostrará en el explorador. El hecho de que cada página del sitio tenga su propia dirección URL que los motores de búsqueda pueden guardar como marcador e indexar (de forma predeterminada, sin tener que agregar esta función como una característica independiente de la aplicación) también es una ventaja clara en este tipo de escenarios.

### La aplicación necesita funcionar en exploradores que no admiten JavaScript

Las aplicaciones web que tienen que funcionar en exploradores con compatibilidad limitada con JavaScript o sin ella se deben escribir mediante flujos de trabajo de

aplicaciones web tradicionales (o al menos que se puedan revertir a este comportamiento). Las SPA requieren JavaScript del lado cliente para poder funcionar; si no está disponible, las SPA no son una buena elección.

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

Si el equipo no está familiarizado con JavaScript o TypeScript, pero sí con el desarrollo de aplicaciones web del lado servidor, probablemente podrán ofrecer una aplicación web tradicional más rápidamente que una SPA. A menos que aprender a programar SPA sea un objetivo o que se necesite la experiencia del usuario que ofrece una SPA, las aplicaciones web tradicionales son una opción más productiva para los equipos que ya están familiarizados con su creación.

## **Casos en los que elegir SPA**

En la sección siguiente se ofrece una explicación más detallada de cuándo se debe elegir un estilo de desarrollo de aplicaciones de página única (SPA) para las aplicaciones web.

### **La aplicación tiene que exponer una interfaz de usuario Enriquecida con muchas características**

Las SPA pueden admitir funciones enriquecidas del lado cliente que no requieran volver a cargar la página cuando los usuarios realicen acciones o naveguen entre las áreas de la aplicación. Las SPA se pueden cargar más rápidamente, recuperar datos en segundo plano y las acciones individuales de los usuarios tienen más capacidad de respuesta ya que las recargas de página completas son poco frecuentes. Las SPA pueden admitir actualizaciones incrementales y guardar formularios o documentos completados parcialmente sin que el usuario tenga que hacer clic en un botón para enviar un formulario. Las SPA pueden admitir comportamientos enriquecidos del lado cliente, como arrastrar y colocar, mucho más fácilmente que las aplicaciones tradicionales. Las SPA se pueden diseñar para ejecutarse en un modo sin conexión y realizar actualizaciones en un modelo del lado cliente que se sincronizan finalmente al servidor una vez que se restablece la conexión. Elija una aplicación de estilo SPA si los requisitos de la aplicación incluyen funcionalidad enriquecida que va más allá de la que ofrecen los formularios HTML habituales.

Con frecuencia, las SPA tienen que implementar características integradas en las aplicaciones web tradicionales, como mostrar una dirección URL significativa en la barra de direcciones que refleje la operación actual (y que permita a los usuarios guardarla como marcador o vínculo profundo para volver a ella). Las SPA también deben permitir

a los usuarios hacer clic en los botones Atrás y Adelante del explorador con resultados que no les sorprendan.

### **El equipo está familiarizado con el desarrollo de JavaScript o TypeScript**

Para escribir SPA es necesario estar familiarizado con JavaScript o TypeScript y técnicas de programación y bibliotecas del lado cliente. El equipo debería ser competente en la escritura de JavaScript moderno con un marco de SPA como Angular.

## **Referencias: marcos de SPA**

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

### **La aplicación ya tiene que exponer una API para otros clientes (internos o públicos)**

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

## **Casos en los que elegir Blazor**

En la sección siguiente ofrecemos una explicación más detallada sobre por qué elegir Blazor para su aplicación web.

### **La aplicación tiene que exponer una interfaz de usuario enriquecida**

Al igual que las SPA basadas en JavaScript, las aplicaciones de Blazor pueden admitir el comportamiento de cliente enriquecido sin recargas de página. Estas aplicaciones tienen más capacidad de respuesta a los usuarios y solo capturan los datos (o HTML) necesarios para responder a una interacción determinada del usuario. Diseñadas correctamente, las aplicaciones del lado servidor de Blazor se pueden configurar para que se ejecuten como aplicaciones del lado cliente de Blazor con cambios mínimos cuando se admite esta característica.

### **Su equipo está más familiarizado con el desarrollo de .NET que con el desarrollo de JavaScript o TypeScript**

Muchos desarrolladores son más productivos con .NET y Razor que con los lenguajes del lado cliente, como JavaScript o TypeScript. Dado que el lado servidor de la

aplicación ya se está desarrollando con .NET, el uso de Blazor garantiza que todos los desarrolladores de .NET del equipo puedan entender y crear potencialmente el comportamiento del front-end de la aplicación.

## Tabla de decisiones

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

Factor	Aplicación web tradicional	Aplicación de una sola página	Aplicación de Blazor
Familiaridad del equipo necesaria con JavaScript o TypeScript	Mínima	Obligatoria	Mínima
Compatibilidad con exploradores sin scripting	Compatible	No compatible	Compatible
Comportamiento mínimo del lado cliente de la aplicación	Adecuado	Excesivo	Viable
Requisitos de la interfaz de usuario enriquecidos y complejos	Limitados	Adecuados	Adecuados

[Anterior](#)

[Siguiente](#)

# Principios de la arquitectura

Artículo • 28/11/2022 • Tiempo de lectura: 10 minutos

## 💡 Sugerencia

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

[Descargar PDF](#)



"Si los constructores construyeran los edificios como los programadores escriben los programas, el primer pájaro carpintero que apareciera destruiría la civilización".

- *Gerald Weinberg*

Las soluciones de software se deben diseñar y crear con el mantenimiento en mente. Los principios que se describen en esta sección le ayudarán a tomar decisiones arquitectónicas que darán como resultado aplicaciones limpias y fácil de mantener. Por lo general, estos principios le ayudarán a compilar aplicaciones a partir de componentes discretos que no están estrechamente relacionados con otras partes de la aplicación, sino que se comunican a través de interfaces explícitas o sistemas de mensajería.

## Principios de diseño comunes

### Separación de intereses

Un principio fundamental durante el desarrollo es la **separación de intereses**. Este principio afirma que el software se debe separar en función de los tipos de trabajo que realiza. Por ejemplo, considere una aplicación que incluye lógica para identificar los

elementos de interés que se van a mostrar al usuario y que da formato a estos elementos de una manera determinada para hacerlos más evidentes. El comportamiento responsable de elegir a qué elementos dar formato se debe mantener separado del comportamiento responsable de dar formato a los elementos, puesto que estos son intereses independientes que solo se relacionan entre sí de manera casual.

Desde el punto de vista de la arquitectura, las aplicaciones se pueden crear de forma lógica para seguir este principio mediante la separación del comportamiento de negocios principal de la lógica de la interfaz de usuario y la infraestructura. Idealmente, la lógica y las reglas de negocios deben residir en un proyecto independiente, que no debería depender de otros proyectos de la aplicación. Esta separación ayuda a garantizar que el modelo de negocio sea fácil de probar y pueda evolucionar sin estar estrechamente relacionado con los detalles de implementación de bajo nivel (también ayuda si los problemas de la infraestructura dependen de abstracciones definidas en la capa de negocio). La separación de intereses es una consideración clave detrás del uso de capas en arquitecturas de aplicación.

## Encapsulación

Las diferentes partes de una aplicación deben usar la **encapsulación** para aislarse de otras partes de la aplicación. Las capas y los componentes de la aplicación deben poder ajustar su implementación interna sin interrumpir a sus colaboradores mientras no se infrinjan los externos. El uso correcto de la encapsulación contribuye a lograr el acoplamiento flexible y la modularidad en los diseños de aplicaciones, ya que los objetos y paquetes se pueden reemplazar con implementaciones alternativas, siempre y cuando se mantenga la misma interfaz.

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

El estado global mutable es antítetico a la encapsulación. No se puede confiar en que un valor obtenido del estado global mutable de una función tenga el mismo valor en otra función, o incluso más en la misma función. Comprender los problemas con el estado global mutable es uno de los motivos por los que los lenguajes de programación

como C# admiten distintas reglas de ámbito, que se usan en todas partes, desde instrucciones y métodos hasta clases. Merece la pena tener en cuenta que las arquitecturas controladas por datos que se basan en una base de datos central para la integración dentro y entre aplicaciones son, por sí mismas, las que deciden depender del estado global mutable representado por la base de datos. Una consideración clave en el diseño controlado por el dominio, y la arquitectura limpia es la forma de encapsular el acceso a los datos y de asegurarse de que el estado de la aplicación no sea válido mediante el acceso directo a su formato de persistencia.

## Inversión de dependencias

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

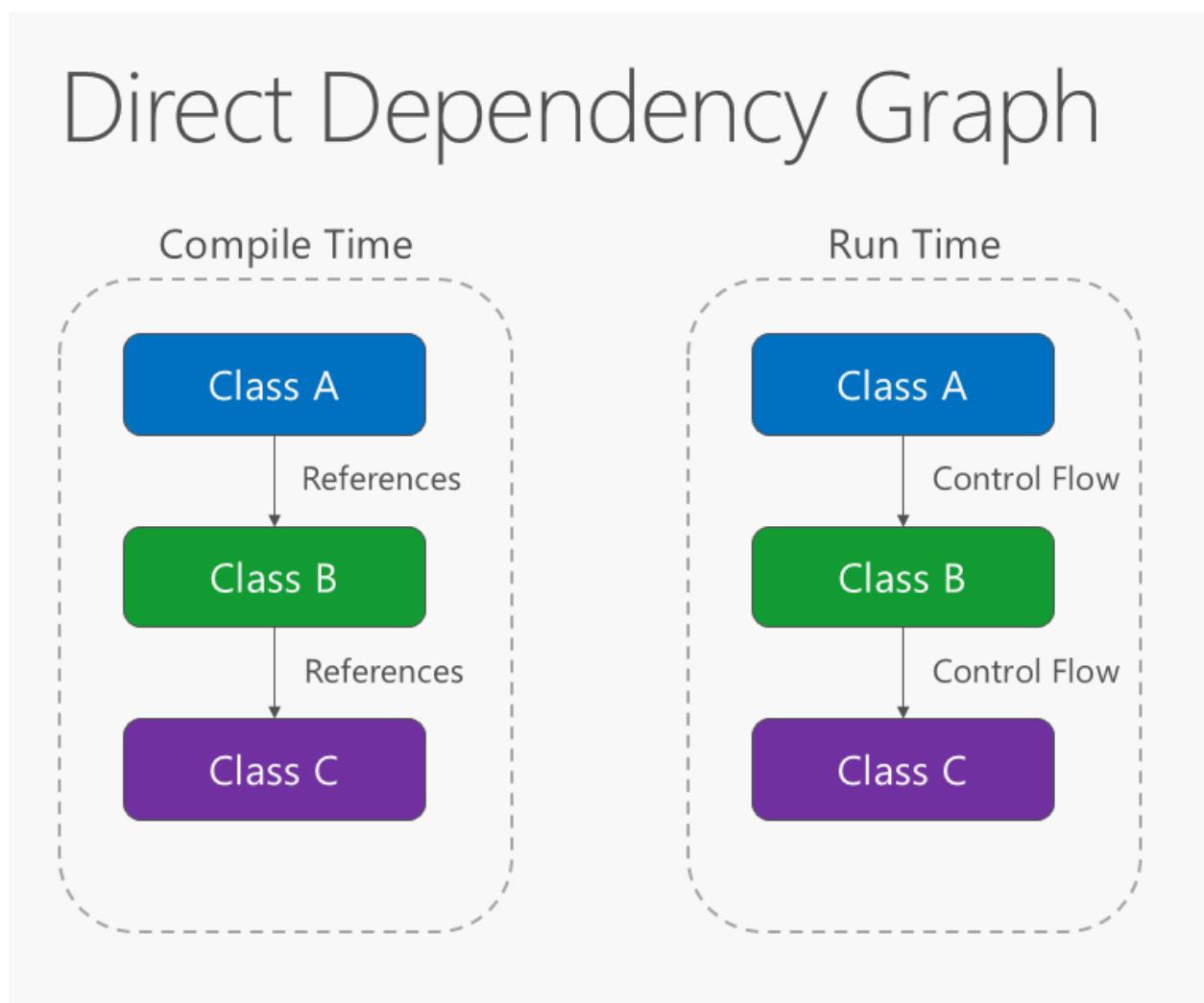


Figura 4-1. Gráfico de dependencias directas.

Aplicar el principio de inversión de dependencias permite que A llame a métodos en una abstracción que implementa B, lo que hace posible que A llame a B en tiempo de ejecución, pero que B dependa de una interfaz que controla A en tiempo de compilación (por tanto, se *invierte* la dependencia de tiempo de compilación normal). En tiempo de ejecución, el flujo de ejecución del programa no cambia, pero la introducción de interfaces significa que se pueden conectar fácilmente otras implementaciones de estas interfaces.

## Inverted Dependency Graph

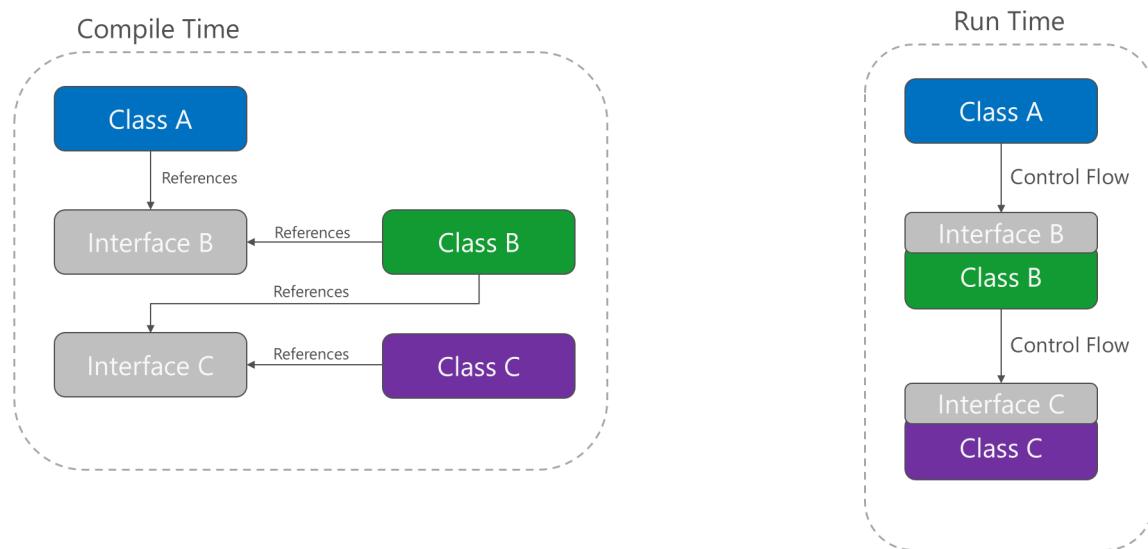


Figura 4-2. Gráfico de dependencias invertidas.

La **Inversión de dependencias** es una parte fundamental de la creación de aplicaciones de acoplamiento flexible, ya que se pueden escribir detalles de implementación de los que depender e implementar abstracciones de nivel superior, en lugar de hacerlo al revés. Como resultado, las aplicaciones son modulares y más fáciles de probar y mantener. La práctica de la *inversión de dependencias* es posible si se sigue el principio de inversión de dependencias.

## Dependencias explícitas

Los métodos y las clases deben requerir explícitamente todos los objetos de colaboración que necesiten para funcionar correctamente. Esto se conoce como el [Principio de dependencias explícitas](#). Los constructores de clases proporcionan una oportunidad para que las clases identifiquen lo que necesitan para poder tener un estado válido y funcionar correctamente. Si se definen clases que se pueden construir y a las que se puede llamar, pero que solo funcionarán correctamente si existen determinados componentes globales o de infraestructura, estas clases *no estarán siendo honestas* con sus clientes. El contrato de constructor indica al cliente que solo necesita los elementos especificados (posiblemente ninguno si la clase solo usa un constructor

sin parámetros), pero después, en tiempo de ejecución, en realidad el objeto necesita algo más.

Si siguen el principio de dependencias explícitas, las clases y métodos estarán siendo sinceros con sus clientes con respecto a lo que necesitan para poder funcionar. Seguir el principio hace que el código sea más autoexplicativo y los contratos de codificación más fáciles de usar, puesto que los usuarios confiarán en que siempre que proporcionen lo que se necesita en forma de parámetros de método o constructor, los objetos con los trabajan se comportarán correctamente en tiempo de ejecución.

## Responsabilidad única

El principio de responsabilidad única se aplica al diseño orientado a objetos, pero también se puede considerar como un principio de arquitectura similar a la separación de intereses. Indica que los objetos solo deben tener una responsabilidad y solo una razón para cambiar. En concreto, la única situación en la que el objeto debe cambiar es si hay que actualizar la manera en la que lleva a cabo su única responsabilidad. El hecho de seguir este principio ayuda a generar sistemas más modulares y de acoplamiento flexible, dado que muchos tipos de comportamientos nuevos se pueden implementar como clases nuevas, en lugar de mediante la incorporación de responsabilidad adicional a las clases existentes. Agregar clases nuevas siempre es más seguro que cambiar las existentes, puesto que todavía no hay código que dependa de las clases nuevas.

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

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

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

## Una vez y solo una (DRY)

La aplicación debe evitar especificar el comportamiento relacionado con un determinado concepto en varios lugares, ya que esta práctica es una fuente de errores

frecuente. En algún momento, un cambio en los requisitos requerirá cambiar este comportamiento. Es probable que al menos una instancia del comportamiento no se pueda actualizar, y que el sistema se comporte de manera incoherente.

En lugar de duplicar la lógica, se puede encapsular en una construcción de programación. Convierta esta construcción en la única autoridad sobre este comportamiento y haga que cualquier otro elemento de la aplicación que requiera este comportamiento use la nueva construcción.

### ⓘ Nota

Evite el enlace conjunto del comportamiento que solo se repita de forma causal. Por ejemplo, solo porque dos constantes diferentes tengan el mismo valor no significa que debería tener solo una constante, si conceptualmente se refieren a cosas diferentes. Siempre es preferible la duplicación en lugar del acoplamiento a una abstracción incorrecta.

## Omisión de persistencia

La **Omisión de persistencia** (PI) hace referencia a los tipos que se deben conservar, pero cuyo código no se ve afectado por la elección de la tecnología de persistencia. En .NET, estos tipos a veces se denominan objeto CRL estándar (POCO), ya que no necesitan heredar de una clase base concreta o implementar una interfaz determinada. La omisión de persistencia es útil porque permite conservar el mismo modelo de negocio de varias formas, lo que ofrece flexibilidad adicional a la aplicación. Es posible que las opciones de persistencia cambien con el tiempo, de una tecnología de base de datos a otra, o bien que se necesiten otras formas de persistencia además de las iniciales de la aplicación (por ejemplo, el uso de una caché en Redis o Azure Cosmos DB además de un base de datos relacional).

Algunos ejemplos de las infracciones de este principio son estos:

- Una clase base requerida.
- Una implementación de interfaz requerida.
- Clases responsables de guardarse a sí mismas (por ejemplo, el patrón de registro activo).
- Constructor sin parámetros requerido.
- Propiedades que requieren la palabra clave virtual.

- Atributos requeridos específicos de la persistencia.

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

## Contextos delimitados

Los **contextos delimitados** son un patrón esencial en el diseño controlado por dominios. Proporcionan una manera de abordar la complejidad en organizaciones o aplicaciones de gran tamaño dividiéndola en módulos conceptuales independientes. Después, cada módulo conceptual representa un contexto que está separado de otros contextos (por tanto, delimitado) y que puede evolucionar independientemente. Idealmente, cada contexto delimitado debería poder elegir sus propios nombres para los conceptos que contiene y tener acceso exclusivo a su propio almacén de persistencia.

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

## Recursos adicionales

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

[Anterior](#)

[Siguiente](#)

# Arquitecturas de aplicaciones web comunes

Artículo • 11/03/2023 • Tiempo de lectura: 25 minutos

## 💡 Sugerencia

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

[Descargar PDF](#)



"Si cree que una buena arquitectura es cara, pruebe una mala arquitectura" (*Brian Foote y Joseph Yoder*).

La mayoría de aplicaciones .NET tradicionales se implementan como unidades únicas que corresponden a un archivo ejecutable o una sola aplicación web que se ejecuta dentro de un único dominio de aplicación de IIS. Este enfoque es el modelo de implementación más sencillo, y sirve muy bien a muchas aplicaciones internas y públicas más pequeñas. Pero incluso con esta única unidad de implementación, la mayoría de las aplicaciones de negocio importantes se aprovechan de cierta separación lógica en varias capas.

## ¿Qué es una aplicación monolítica?

Una aplicación monolítica es aquella completamente independiente, en términos de su comportamiento. Puede interactuar con otros servicios o almacenes de datos en el transcurso de sus operaciones, pero el núcleo de su comportamiento se ejecuta dentro

de su propio proceso y toda la aplicación normalmente se implementa como una única unidad. Si es necesario escalar horizontalmente este tipo de aplicación, normalmente la aplicación completa se duplica en varios servidores o máquinas virtuales.

## Aplicaciones todo en uno

El menor número posible de proyectos para una arquitectura de aplicación es uno. En esta arquitectura, toda la lógica de la aplicación está contenida en un solo proyecto, se compila en un único ensamblado y se implementa como una sola unidad.

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

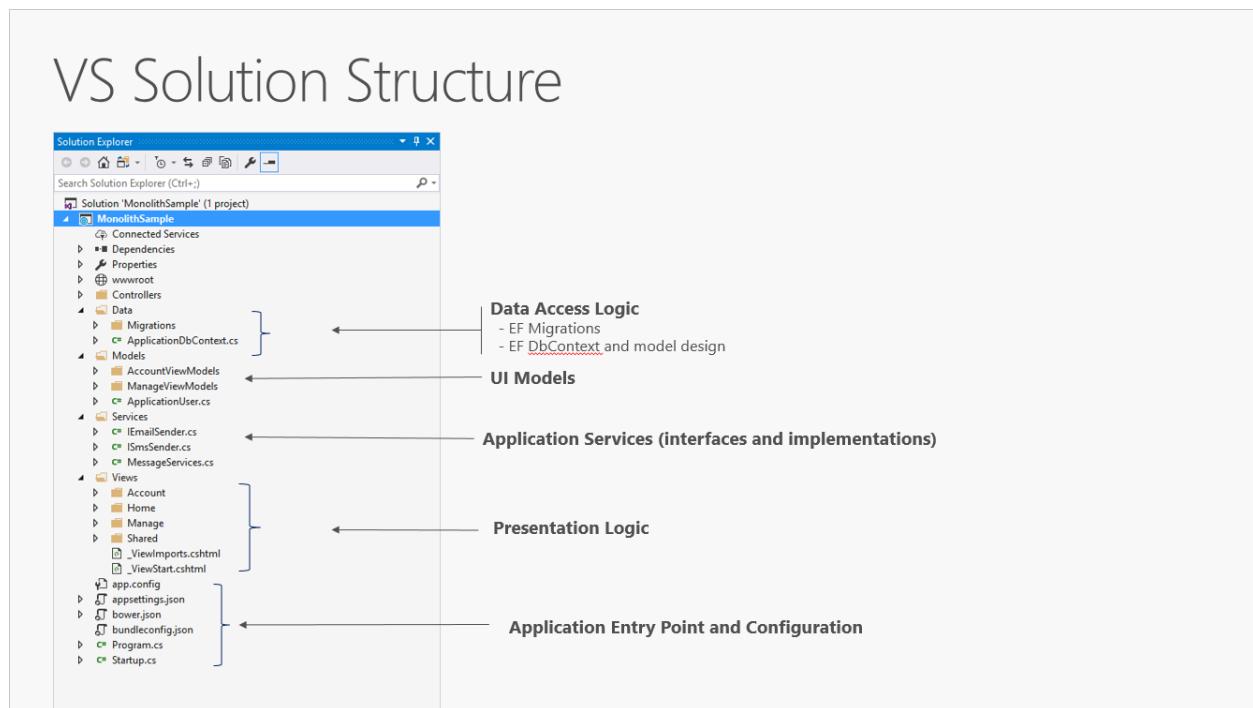


Figura 5-1. Una aplicación ASP.NET Core de un solo proyecto.

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

Aunque es simple, la solución monolítica de un solo proyecto tiene algunas desventajas. A medida que aumenta el tamaño y la complejidad del proyecto, el número de archivos y carpetas también seguirá creciendo. Los intereses de la interfaz de usuario (IU) (modelos, vistas, controladores) residen en varias carpetas, que no están agrupadas alfabéticamente. Este problema empeora cuando se agregan otras construcciones de nivel de la interfaz de usuario, como filtros o ModelBinders, en sus propias carpetas. La lógica de negocios se distribuye entre las carpetas Modelos y Servicios, y no hay ninguna indicación clara de qué clases de qué carpetas deben depender de otras. Esta falta de organización en el nivel del proyecto suele dar lugar a [código espagueti](#).

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

## ¿Qué son las capas?

Cuando aumenta la complejidad de las aplicaciones, una manera de administrarla consiste en dividir la aplicación según sus responsabilidades o intereses. Este enfoque sigue el principio de separación de intereses y puede ayudar a mantener organizado un código base que crece para que los desarrolladores puedan encontrar fácilmente dónde se implementa una función determinada. Pero la arquitectura en capas ofrece una serie de ventajas que van más allá de la simple organización del código.

Al organizar el código en capas, la funcionalidad común de bajo nivel se puede reutilizar en toda la aplicación. Esta reutilización es beneficiosa ya que significa escribir menos código y puede permitir que la aplicación se estandarice en una sola implementación, siguiendo el principio [Una vez y solo una \(DRY\)](#).

Con una arquitectura en capas, las aplicaciones pueden aplicar restricciones sobre qué capas se pueden comunicar con otras capas. Esta arquitectura permite lograr la encapsulación. Cuando se cambia o reemplaza una capa, solo deberían verse afectadas aquellas capas que funcionan con ella. Mediante la limitación de qué capas dependen de otras, se puede mitigar el impacto de los cambios para que un único cambio no afecte a toda la aplicación.

Las capas (y la encapsulación) facilitan considerablemente el reemplazo de funcionalidad dentro de la aplicación. Por ejemplo, es posible que una aplicación use inicialmente su propia base de datos de SQL Server para la persistencia, pero más adelante podría optar por usar una estrategia de persistencia basada en la nube, o situada detrás de una API web. Si la aplicación ha encapsulado correctamente su implementación de persistencia dentro de una capa lógica, esa capa específica de

SQL Server se podría reemplazar por una nueva que implementara la misma interfaz pública.

Además de la posibilidad de intercambiar las implementaciones en respuesta a cambios futuros en los requisitos, las capas de aplicación también facilitan el intercambio de implementaciones con fines de prueba. En lugar de tener que escribir pruebas que funcionan en la capa de datos reales o de la interfaz de usuario de la aplicación, estas capas se pueden reemplazar en tiempo de prueba con implementaciones falsas que proporcionen respuestas conocidas a las solicitudes. Este enfoque normalmente hace que las pruebas sean mucho más fáciles de escribir y mucho más rápidas de ejecutar en comparación con la ejecución de pruebas sobre la infraestructura real de la aplicación.

Las capas lógicas son una técnica común para mejorar la organización del código en las aplicaciones de software empresarial, y hay varias formas de organizar el código en capas.

#### Nota

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

## Aplicaciones tradicionales de arquitectura de "N capas"

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

# Application Layers

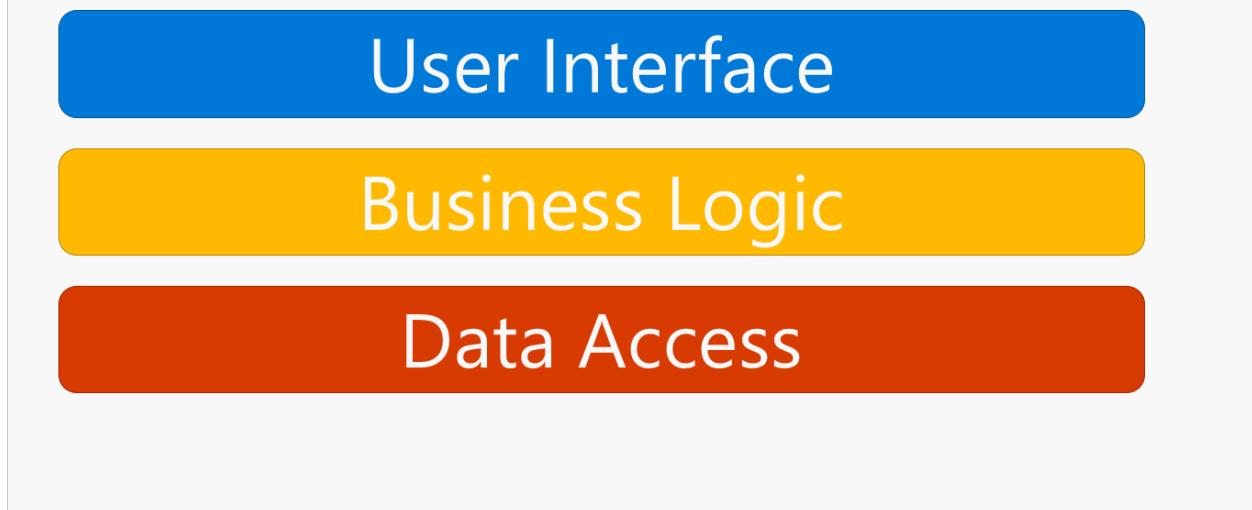


Figura 5-2. Capas de aplicación típicas.

Estas capas se suelen abreviar como UI (interfaz de usuario), BLL (capa de lógica de negocios) y DAL (capa de acceso a datos). Con esta arquitectura, los usuarios realizan solicitudes a través de la capa de interfaz de usuario, que interactúa con la capa BLL. BLL, a su vez, puede llamar a DAL para las solicitudes de acceso de datos. La capa de interfaz de usuario no debe realizar solicitudes directamente a DAL, ni debe interactuar con la persistencia de forma directa a través de otros medios. Del mismo modo, BLL solo debe interactuar con la persistencia a través de DAL. De este modo, cada capa tiene su propia responsabilidad conocida.

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

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

# VS Solution Structure

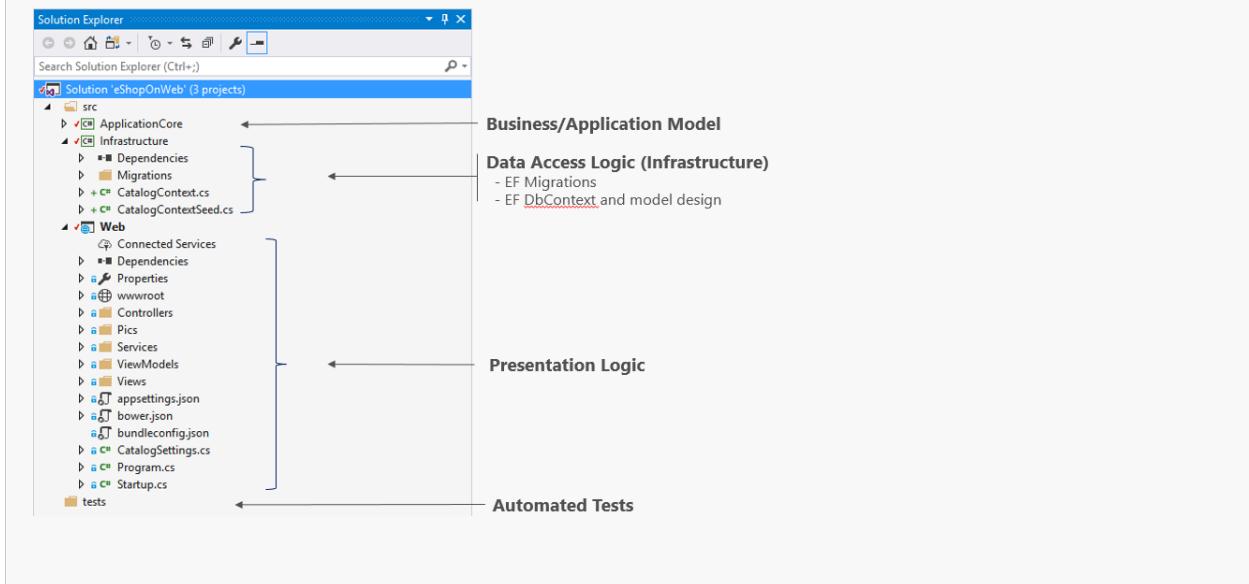


Figura 5-3. Una aplicación monolítica sencilla con tres proyectos.

Aunque en esta aplicación se usan varios proyectos por motivos organizativos, se sigue implementando como una sola unidad y sus clientes interactúan con ella como una sola aplicación web. Esto permite que el proceso de implementación sea muy simple. En la figura 5-4 se muestra cómo se podría hospedar este tipo de aplicación con Azure.

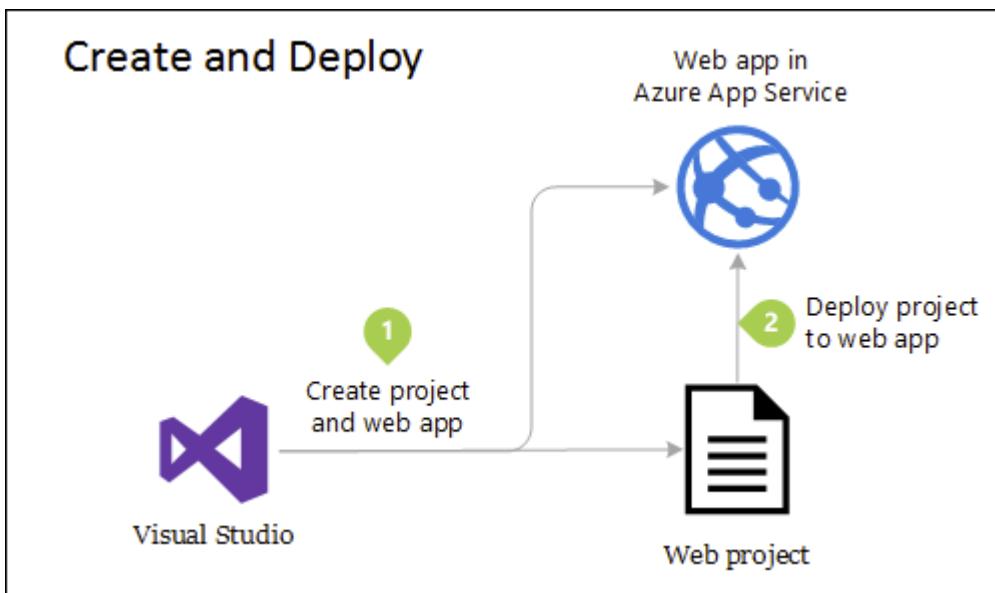


Figura 5-4. Implementación simple de una aplicación web de Azure

Cuando aumenten las necesidades de la aplicación, se pueden necesitar soluciones de implementación más sólidas y complejas. En la figura 5-5 se muestra un ejemplo de un plan de implementación más complejo que admite funciones adicionales.

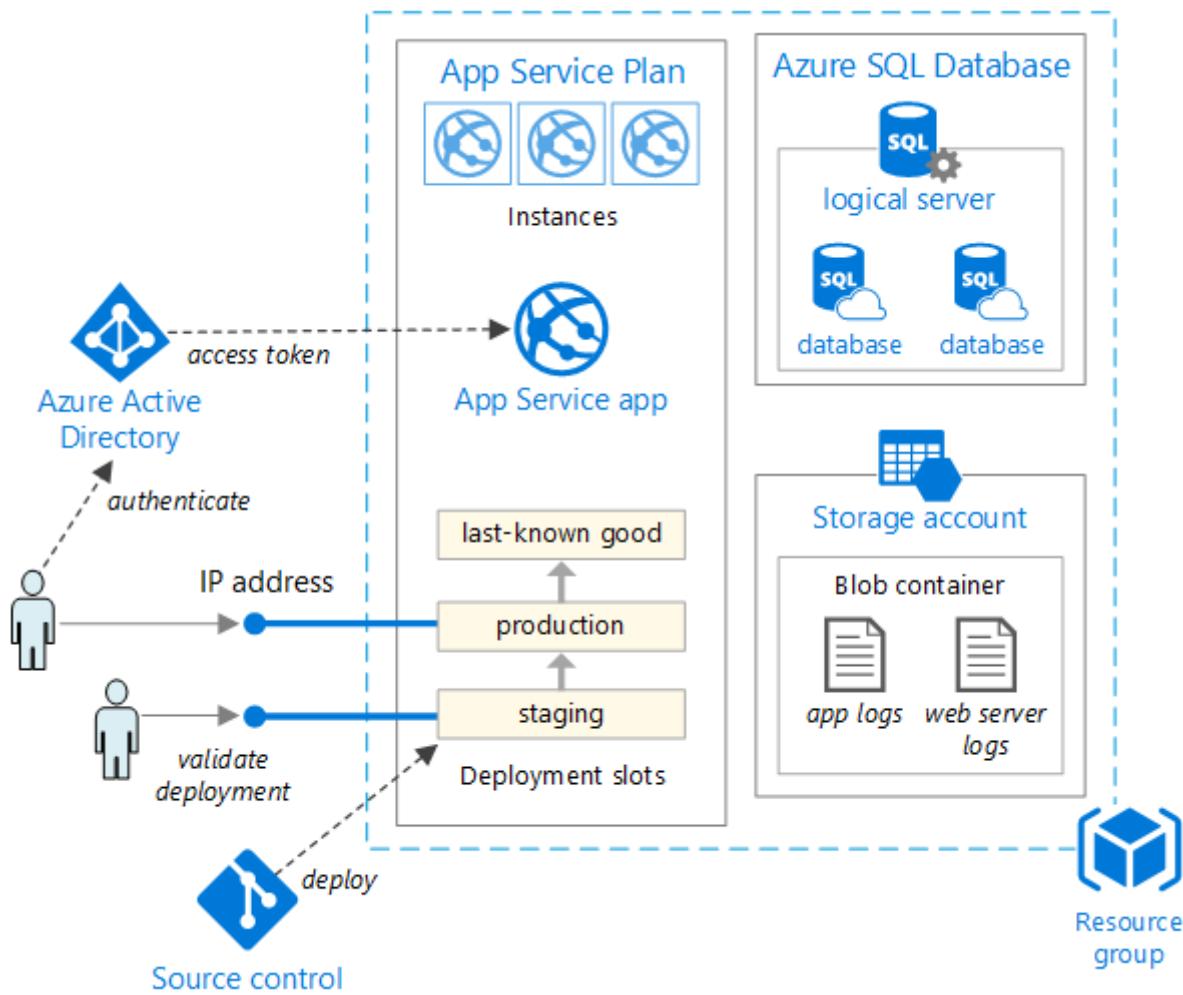


Figura 5-5. Implementación de una aplicación web en Azure App Service

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

Esta unidad se puede escalar vertical u horizontalmente para aprovechar la escalabilidad a petición basada en la nube. El escalado vertical significa agregar más CPU, memoria, espacio en disco u otros recursos al servidor en el que se hospeda la aplicación. El escalado horizontal significa agregar instancias adicionales de estos servidores, con independencia de que sean servidores físicos, máquinas virtuales o contenedores. Cuando la aplicación se hospeda en varias instancias, se usa un equilibrador de carga para asignar solicitudes a instancias individuales de la aplicación.

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

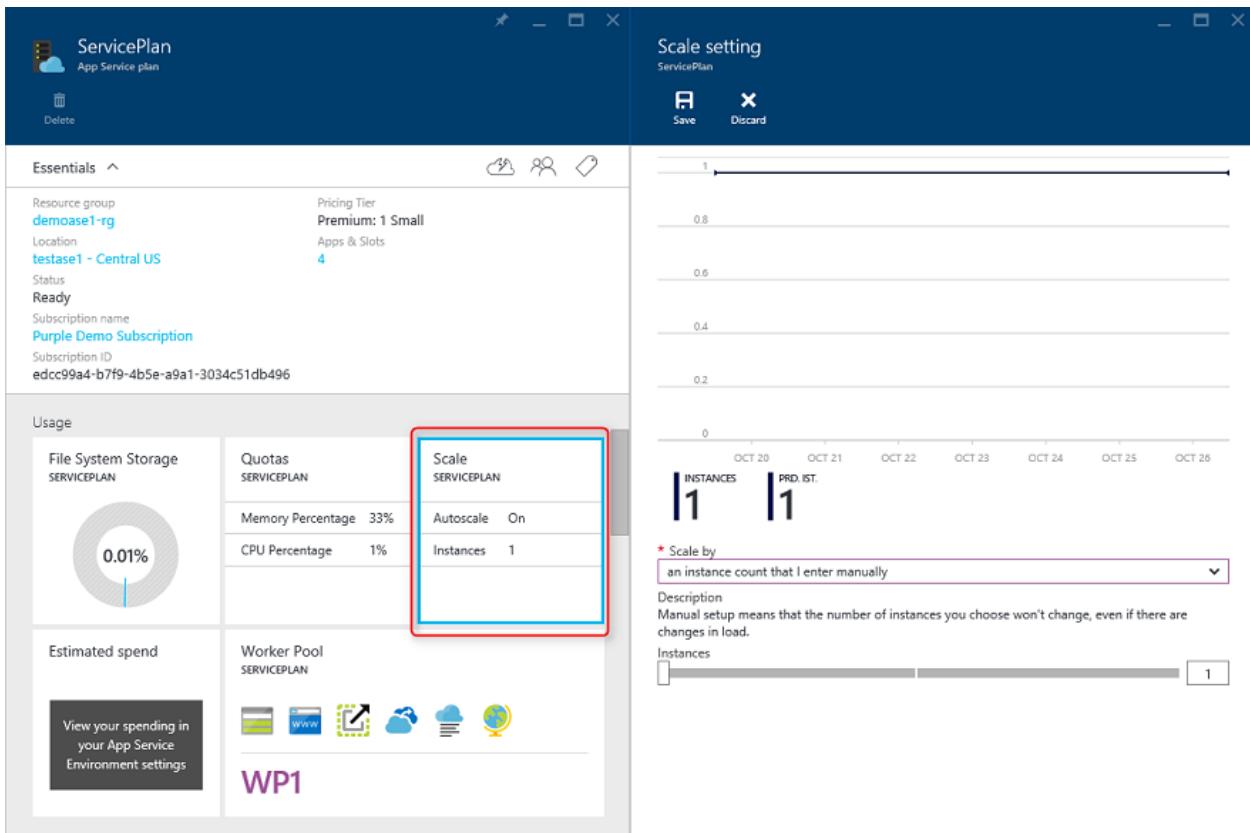


Figura 5-6. Escalado de plan de App Service en Azure.

## Arquitectura limpia

Las aplicaciones que siguen el principio de inversión de dependencias, así como los principios de diseño controlado por dominios (DDD), tienden a llegar a una arquitectura similar. Esta arquitectura ha pasado por muchos nombres con los años. Uno de los primeros nombres fue Arquitectura hexagonal, seguido por Puertos y adaptadores. Más recientemente, se ha citado como [arquitectura cebolla](#) o [arquitectura limpia](#). Este último nombre, Arquitectura limpia, es el que se usa para esta arquitectura en este libro electrónico.

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

La arquitectura limpia coloca el modelo de lógica de negocios y aplicación en el centro de la aplicación. En lugar de tener lógica de negocios que depende del acceso a datos o de otros aspectos de infraestructura, esta dependencia se invierte: los detalles de la infraestructura y la implementación dependen del núcleo de la aplicación. Esta función se logra mediante la definición de abstracciones o interfaces, en el núcleo de la aplicación, que después se implementan mediante tipos definidos en el nivel de

infraestructura. Una forma habitual de visualizar esta arquitectura es usar una serie de círculos concéntricos, similares a una cebolla. En la figura 5-7 se muestra un ejemplo de este estilo de representación de la arquitectura.

## Clean Architecture Layers (Onion view)

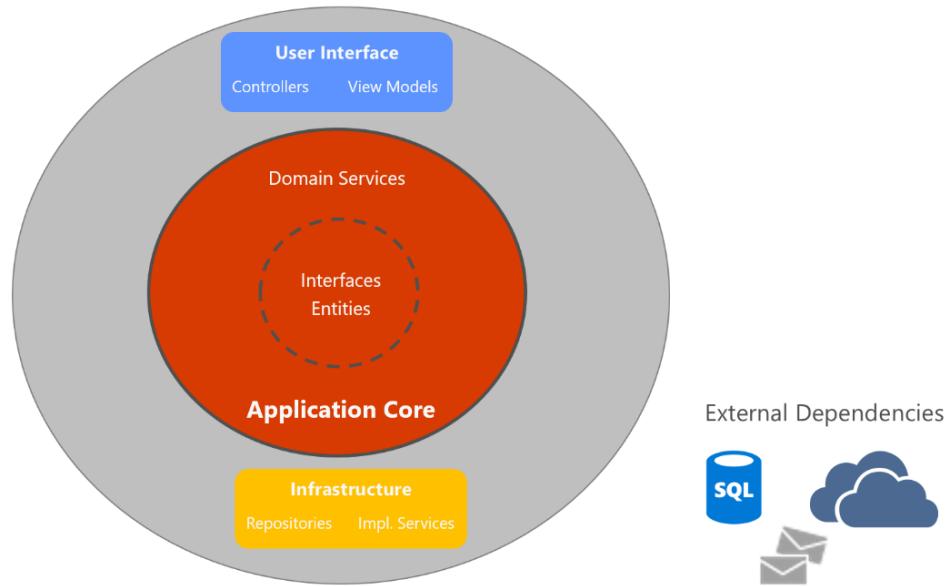


Figura 5-7. Arquitectura limpia; vista de cebolla

En este diagrama, las dependencias fluyen hacia el círculo más interno. El núcleo de la aplicación toma su nombre de su posición en el núcleo de este diagrama. Y en el diagrama puede ver que el núcleo de la aplicación no tiene dependencias de otros niveles de la aplicación. Las entidades e interfaces de la aplicación se encuentran justo en el centro. En el exterior, pero todavía en el núcleo de la aplicación, están los servicios de dominio, que normalmente implementan interfaces definidas en el círculo interior. Fuera del núcleo de la aplicación, las capas de la interfaz de usuario y la infraestructura dependen del núcleo de la aplicación, pero no una de la otra (necesariamente).

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

# Clean Architecture Layers

Optional Compile-Time Dependency  
Compile-Time Dependency

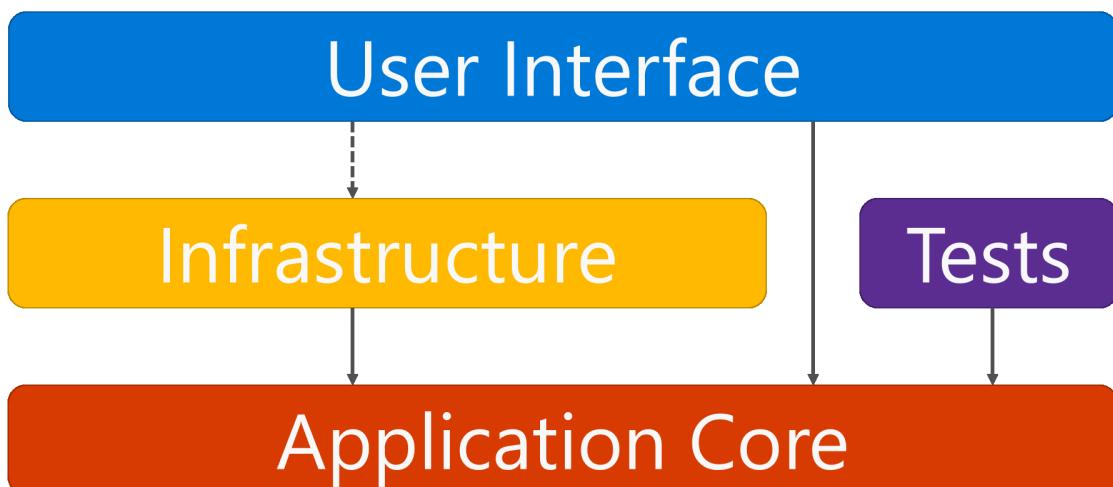


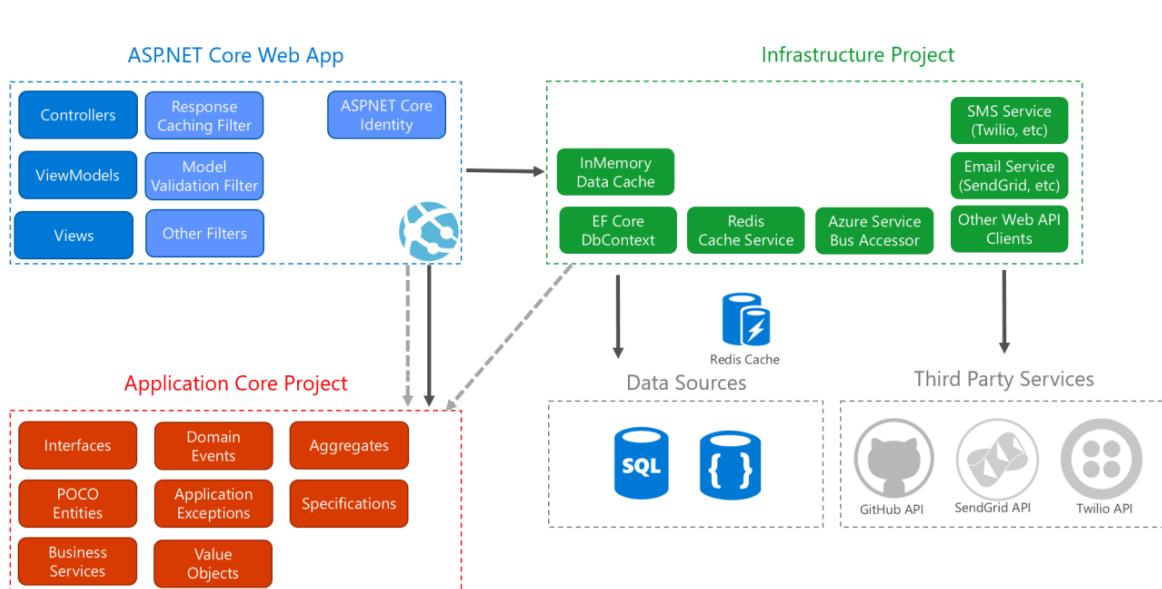
Figura 5-8. Arquitectura limpia; vista de capas horizontales

Tenga en cuenta que las flechas sólidas representan las dependencias de tiempo de compilación, mientras que la flecha discontinua representa una dependencia solo de tiempo de ejecución. Con la arquitectura limpia, la capa de interfaz de usuario funciona con las interfaces definidas en el núcleo de la aplicación en tiempo de compilación y lo ideal es que no conozca los tipos de implementación definidos en el nivel de infraestructura. Pero en tiempo de ejecución, estos tipos de implementación son necesarios para ejecutar la aplicación, por lo que deben estar presentes y conectados a las interfaces del núcleo de la aplicación a través de la inserción de dependencias.

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

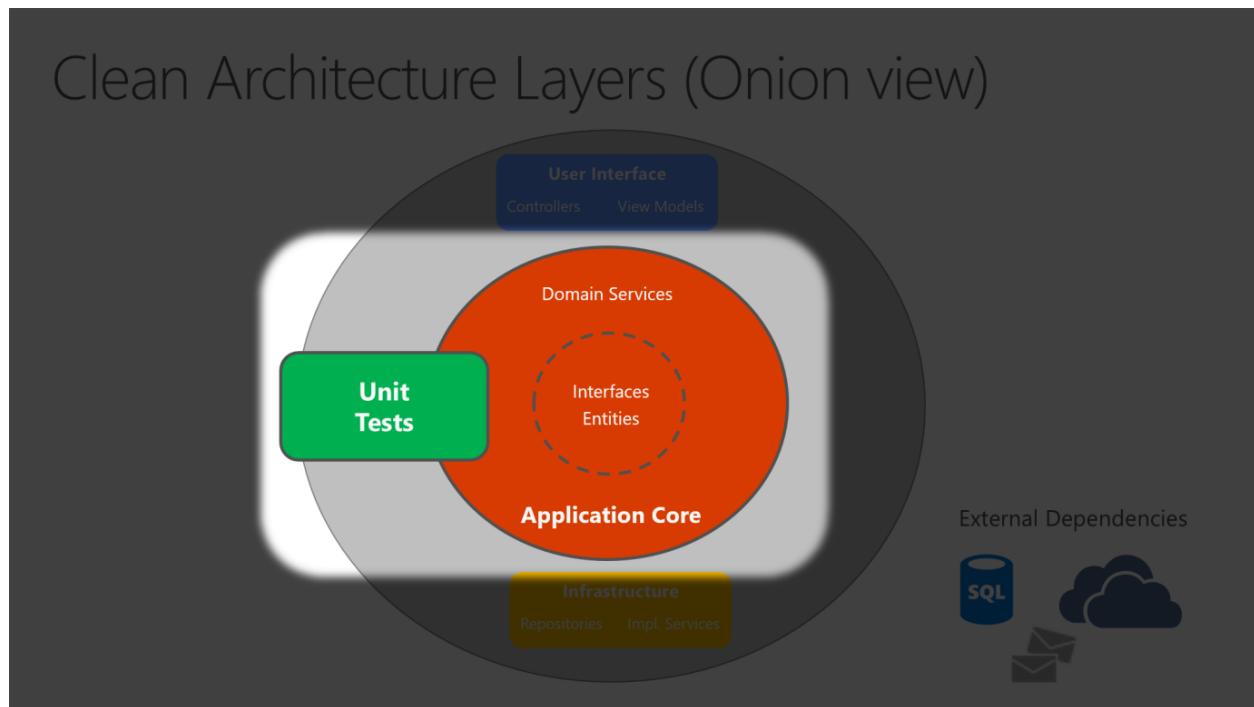
## ASP.NET Core Architecture

Compile Time Dependency  
Run Time Dependency

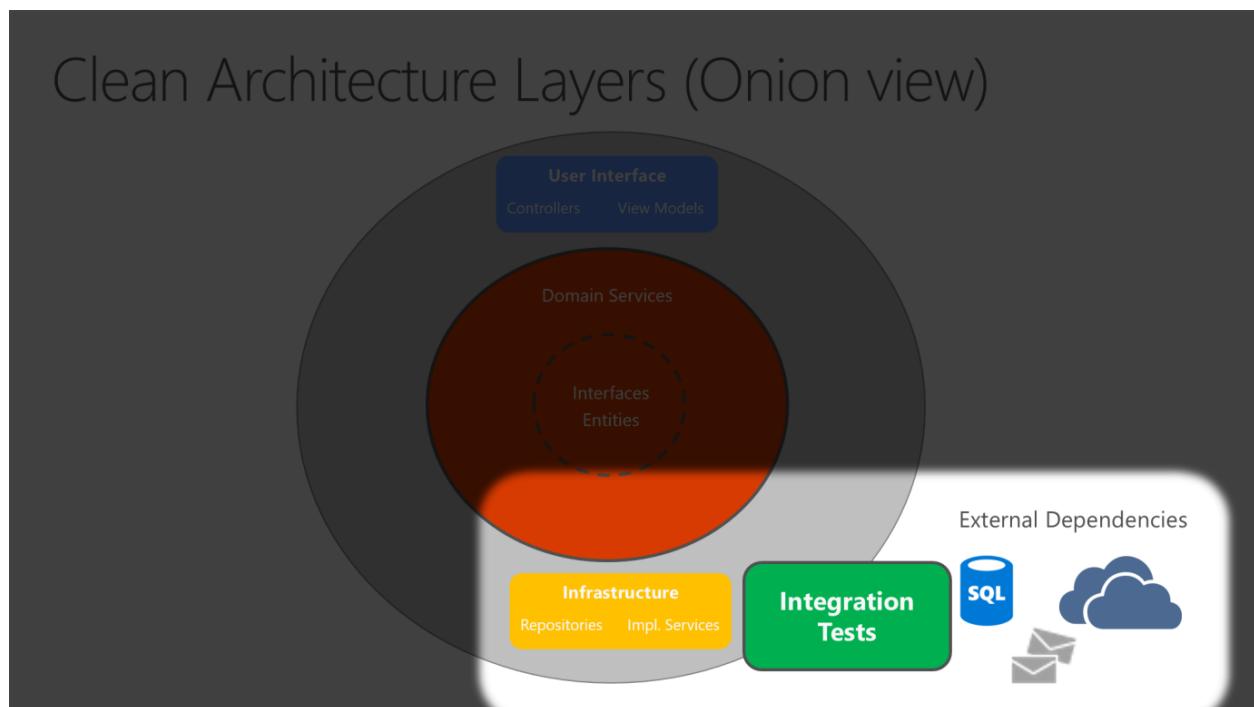


**Figura 5-9.** Diagrama de arquitectura de ASP.NET Core en el que se sigue la arquitectura limpia.

Como el núcleo de la aplicación no depende de la infraestructura, es muy fácil escribir pruebas unitarias automatizadas para esta capa. En las figuras 5-10 y 5-11 se muestra cómo encajan las pruebas en esta arquitectura.



**Figura 5-10.** Pruebas unitarias del núcleo de la aplicación de forma aislada.



**Figura 5-11.** Pruebas de integración de las implementaciones de infraestructura con dependencias externas.

Como la capa de interfaz de usuario no tiene ninguna dependencia directa de los tipos definidos en el proyecto de infraestructura, también es muy fácil intercambiar las implementaciones, ya sea para facilitar las pruebas o en respuesta a los requisitos cambiantes de la aplicación. El uso y la compatibilidad integrados de ASP.NET Core con la inserción de dependencias hace que esta arquitectura sea la forma más adecuada de estructurar aplicaciones monolíticas no triviales.

Para las aplicaciones monolíticas, los proyectos de núcleo de la aplicación, infraestructura e interfaz de usuario se ejecutan como una sola aplicación. La arquitectura de la aplicación en tiempo de ejecución podría ser similar a la de la figura 5-12.

## ASP.NET Core Architecture

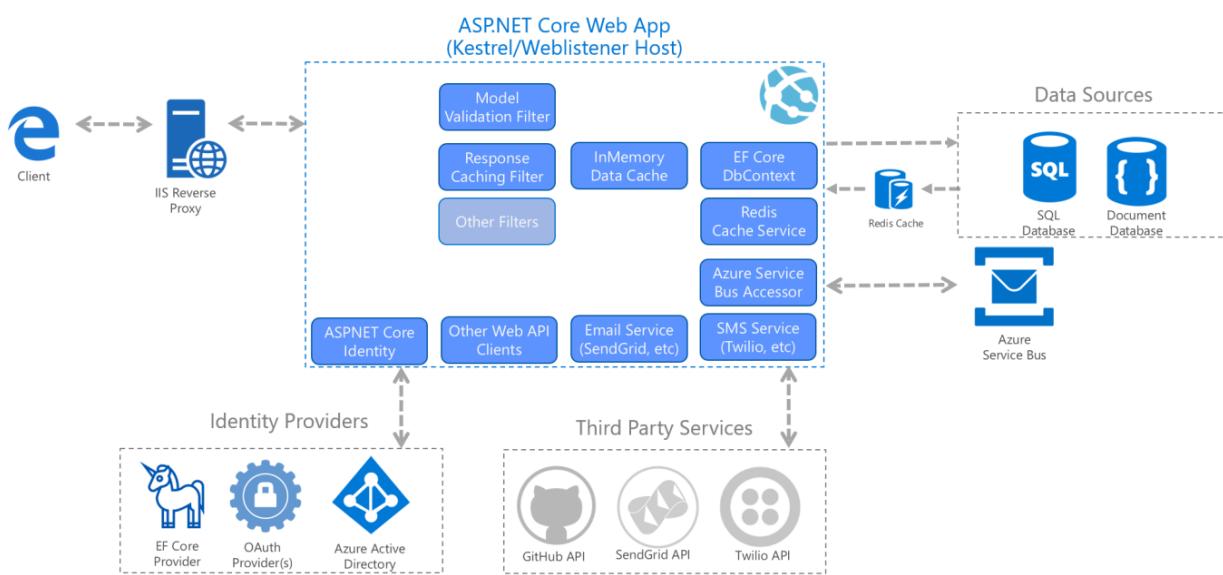


Figura 5-12. Ejemplo de arquitectura en tiempo de ejecución de la aplicación ASP.NET Core.

## Organización del código en la arquitectura limpia

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

### Núcleo de aplicación

El núcleo de la aplicación contiene el modelo de negocio, que incluye entidades, servicios e interfaces. Estas interfaces incluyen abstracciones para las operaciones que se llevarán a cabo mediante la infraestructura, como el acceso a datos, el acceso al sistema de archivos, las llamadas de red, etc. En ocasiones los servicios o interfaces definidos en

este nivel tendrán que trabajar con tipos sin entidad que no tienen dependencias en la interfaz de usuario o la infraestructura. Estos se pueden definir como Objetos de transferencia de datos (DTO) simples.

## Tipos de núcleo de la aplicación

- Entidades (las clases de modelo de negocio que se conservan)
- Agregados (grupos de entidades)
- Interfaces
- Servicios de dominio
- Especificaciones
- Excepciones personalizadas y cláusulas de restricción
- Controladores y eventos de dominio

## Infraestructura

El proyecto de infraestructura incluye normalmente las implementaciones de acceso a datos. En una aplicación web ASP.NET Core típica, estas implementaciones incluyen DbContext de Entity Framework (EF), todos los objetos `Migration` de EF Core que se hayan definido y las clases de implementación de acceso a datos. La manera más común de abstraer el código de implementación de acceso a datos consiste en usar el [modelo de diseño de repositorio ↴](#).

Además de las implementaciones de acceso a datos, el proyecto de infraestructura debe contener las implementaciones de los servicios que tienen que interactuar con los intereses de infraestructura. Estos servicios deben implementar interfaces definidas en el núcleo de la aplicación, por lo que la infraestructura deberá tener una referencia al proyecto del núcleo de la aplicación.

## Tipos de infraestructura

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

## Capa de interfaz de usuario

La capa de interfaz de usuario en una aplicación ASP.NET Core MVC es el punto de entrada para la aplicación. Este proyecto debe hacer referencia al proyecto Application

Core y sus tipos deben interactuar con la infraestructura estrictamente a través de las interfaces definidas en Application Core. En la capa de interfaz de usuario no se debe permitir la creación de instancias directas o llamadas estáticas a los tipos de la capa de infraestructura.

## Tipos de capa de interfaz de usuario

- Controladores
- Filtros personalizados
- Middleware personalizado
- Vistas
- ViewModels
- Inicio

La clase `Startup` o el archivo `Program.cs` es responsable de configurar la aplicación y de conectar los tipos de implementación a las interfaces. El lugar donde se ejecuta esta lógica se conoce como la *raíz de composición* de la aplicación y es lo que permite que la inserción de dependencias funcione correctamente en tiempo de ejecución.

### ⓘ Nota

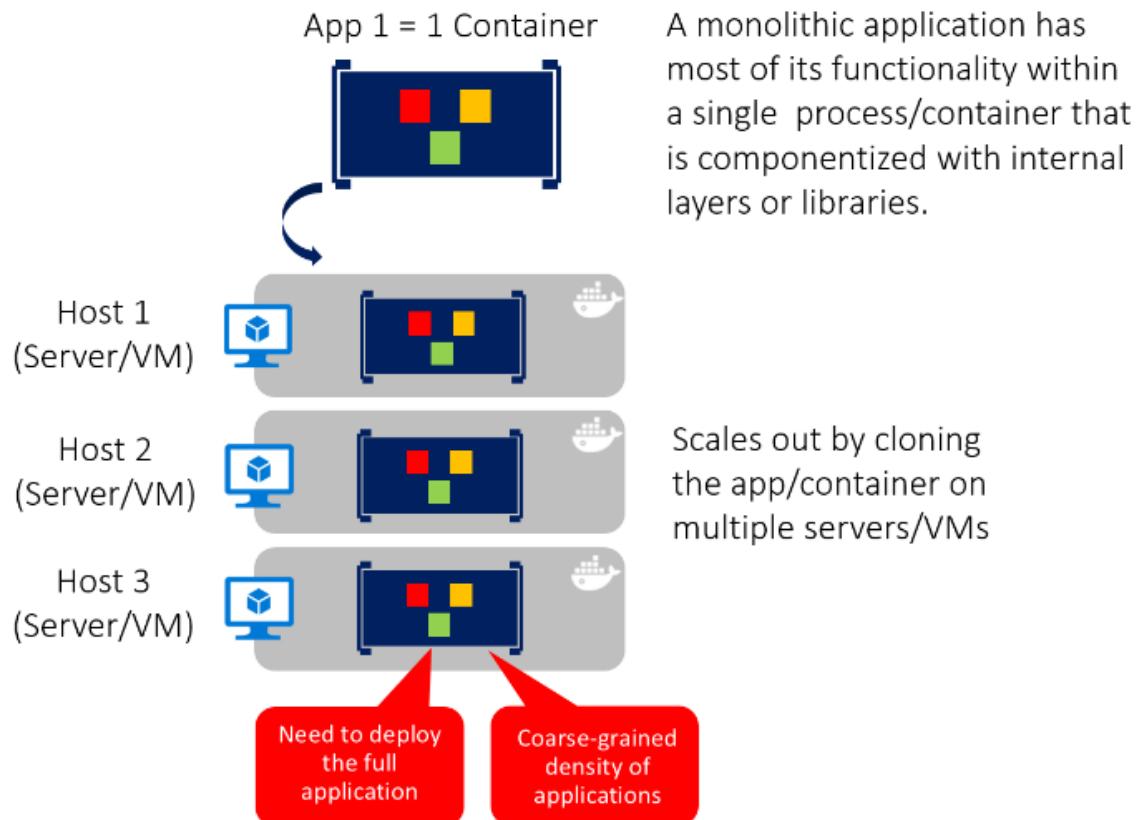
Para conectar la inserción de dependencias durante el inicio de la aplicación, es posible que el proyecto de capa de interfaz de usuario tenga que hacer referencia al proyecto de infraestructura. Esta dependencia se puede eliminar más fácilmente mediante un contenedor de inserción de dependencias personalizado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

## Aplicaciones monolíticas y contenedores

Puede compilar una aplicación o un servicio web único basado en una implementación monolítica e implementarlos como un contenedor. Dentro de la aplicación, es posible que no sean de tipo monolítico, sino que se organicen en varias bibliotecas, componentes o capas. Externamente es un contenedor único con un proceso único, una aplicación web única o un servicio único.

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

## Monolithic Containerized application



Puede incluir varios componentes, bibliotecas o capas internas en cada contenedor, como se muestra en la figura 5-13. Pero, al seguir el principio de contenedor "*un contenedor realiza una acción y lo hace en un proceso*", es posible que el patrón monolítico entre en conflicto.

El inconveniente de este enfoque aparece si o cuando la aplicación aumenta y debe escalarse. Si se escala toda la aplicación, realmente no es un problema. Pero en la mayoría de los casos, solo algunos elementos de la aplicación son los puntos de obstrucción que deben escalarse, mientras que otros componentes se usan menos.

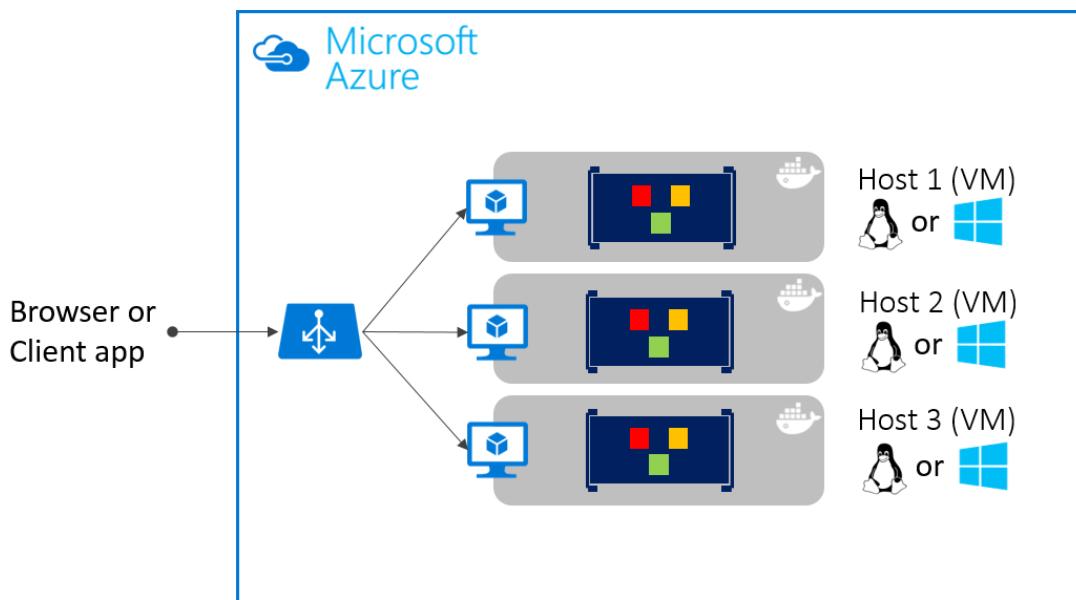
En el ejemplo típico de comercio electrónico, lo que probablemente sea necesario escalar es el componente de información del producto. Hay muchos más clientes que buscan productos de los que los compran. Más clientes usan la cesta en lugar de usar la canalización de pago. Menos clientes publican comentarios o consultan su historial de compras. Y es probable que solo tenga un grupo reducido de empleados, en una única

región, que tenga que administrar las campañas de contenido y marketing. Al escalar el diseño monolítico, todo el código se implementa varias veces.

Además del problema de "escalarlo todo", los cambios en un único componente requieren volver a probar por completo toda la aplicación e implementar por completo todas las instancias.

El enfoque monolítico es habitual y muchas organizaciones realizan el desarrollo con este enfoque de diseño. Muchas obtienen resultados bastante positivos, mientras que otras alcanzan los límites. Muchas diseñaron sus aplicaciones con este modelo, ya que crear arquitecturas orientadas a servicios (SOA) con infraestructura y herramientas resultaba demasiado difícil, y no vieron la necesidad hasta que la aplicación creció. Si comprueba que está alcanzando los límites del enfoque monolítico, dividir la aplicación para que pueda aprovechar mejor los contenedores y microservicios puede ser el siguiente paso lógico.

## Architecture in Docker infrastructure for monolithic applications



La implementación de aplicaciones monolíticas en Microsoft Azure se puede conseguir con máquinas virtuales dedicadas para cada instancia. Con [Azure Virtual Machine Scale Sets](#), las máquinas virtuales se pueden escalar fácilmente. [Azure App Services](#) puede ejecutar aplicaciones monolíticas y escalar fácilmente instancias sin necesidad de administrar las máquinas virtuales. Azure App Services también puede ejecutar instancias únicas de contenedores de Docker, lo que simplifica la implementación. Con Docker, se puede implementar una única máquina virtual como un host de Docker y

ejecutar varias instancias. Con el equilibrador de Azure, como se muestra en la figura 5-14, se puede administrar el escalado.

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

## Una aplicación monolítica implementada como un contenedor

El uso de contenedores para administrar las implementaciones de aplicaciones monolíticas tiene una serie de ventajas. Escalar las instancias de los contenedores es mucho más rápido y fácil que implementar máquinas virtuales adicionales. Incluso cuando se usan conjuntos de escalado de máquinas virtuales para escalar las máquinas virtuales, la creación tarda un tiempo. Cuando se implementa como instancias de aplicación, la configuración de la aplicación se administra como parte de la máquina virtual.

Implementar las actualizaciones como imágenes de Docker es mucho más rápido y eficaz en la red. Normalmente, las imágenes de Docker se inician en segundos, lo que acelera las implementaciones. Anular una instancia de Docker es tan fácil como ejecutar un comando `docker stop`, que normalmente se completa en menos de un segundo.

Dado que, por diseño, los contenedores son intrínsecamente inmutables, no tendrá que preocuparse de que las máquinas virtuales resulten dañadas, mientras que es posible que los scripts de actualización olviden tener en cuenta alguna configuración concreta o archivo que se conserve en el disco.

Puede usar contenedores de Docker para una implementación monolítica de aplicaciones web más sencillas. Este enfoque mejora las canalizaciones de integración continua e implementación continua, y permite llevar a cabo correctamente el proceso desde la implementación hasta la producción. Ya no tendrá que pensar en por qué no funciona en producción, aunque sí que funcione en su equipo.

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

Es posible que una aplicación monolítica no se pueda descomponer fácilmente en microservicios bien separados. Los microservicios deberían funcionar de manera

independiente para proporcionar una aplicación más resistente. Si no puede proporcionar sectores de características independientes de la aplicación, el hecho de separarla solo conlleva más complejidad.

Una aplicación podría no necesitar inicialmente el escalado de las características por separado. Cuando es necesario escalar una aplicación más allá de una única instancia, muchas aplicaciones pueden hacerlo a través del proceso relativamente sencillo de clonación de esa instancia completa. El trabajo adicional de separar la aplicación en servicios discretos apenas proporciona ventajas cuando el escalado de instancias completas de la aplicación es simple y rentable.

En las fases tempranas del desarrollo de una aplicación, es posible que no se tenga una idea clara de dónde están los límites funcionales naturales. Mientras desarrolla un producto mínimamente viable, puede que todavía no sea evidente la separación natural. Algunas de estas condiciones pueden ser temporales. Podría empezar creando una aplicación monolítica y, más adelante, separar algunas características para desarrollarlas e implementarlas como microservicios. Otras condiciones podrían ser básicas para el espacio de problemas de la aplicación y, en consecuencia, tal vez no se pueda dividir nunca en varios microservicios.

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

La [aplicación de referencia eShopOnWeb](#), mucho más sencilla, admite el uso de un único contenedor monolítico. La aplicación incluye una aplicación web con vistas MVC tradicionales, API web y Razor Pages. Opcionalmente, puede ejecutar el componente de administración basado en Blazor de la aplicación, que también requiere que se ejecute un proyecto de API independiente.

La aplicación se puede iniciar desde la raíz de la solución mediante los comandos `docker-compose build` y `docker-compose up`. Este comando configura un contenedor para la instancia web mediante el elemento `Dockerfile` que se encuentra en la raíz de cada proyecto web y ejecuta el contenedor en un puerto específico. Puede descargar de GitHub el código fuente para esta aplicación y ejecutarlo de forma local. Incluso esta aplicación monolítica se beneficia de la implementación en un entorno de contenedor.

Ante todo, la implementación en contenedor implica que cada instancia de la aplicación se ejecuta en el mismo entorno. Este enfoque incluye el entorno de desarrollo donde tienen lugar las pruebas y el desarrollo iniciales. El equipo de desarrollo puede ejecutar la aplicación en un entorno en contenedor que coincida con el entorno de producción.

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

Por último, el hecho de incluir la aplicación en un contenedor fuerza la separación entre la lógica de negocios y el servidor de almacenamiento. Cuando la aplicación se escala horizontalmente, los diversos contenedores se basan en un único medio de almacenamiento físico. Normalmente este medio de almacenamiento es un servidor de alta disponibilidad que ejecuta una base de datos de SQL Server.

## Compatibilidad con Docker

El proyecto `eShopOnWeb` se ejecuta en .NET. Por lo tanto, se puede ejecutar en contenedores basados en Linux o en Windows. Tenga en cuenta que, para la implementación de Docker, le interesa usar el mismo tipo de host para SQL Server. Los contenedores basados en Linux permiten una superficie menor y son preferibles.

Puede usar Visual Studio 2017 o versiones posteriores para agregar compatibilidad con Docker a una aplicación existente haciendo clic con el botón derecho en **Explorador de soluciones** y seleccionando **Agregar>Compatibilidad con Docker**. Este paso agrega los archivos necesarios y modifica el proyecto para poder usarlos. En el ejemplo `eShopOnWeb` actual ya se incluyen estos archivos.

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

```
yml

version: '3'

services:
  eshopwebmvc:
    image: eshopwebmvc
    build:
      context: .
    dockerfile: src/Web/Dockerfile
```

```
environment:  
  - ASPNETCORE_ENVIRONMENT=Development  
ports:  
  - "5106:5106"  
  
networks:  
  default:  
    external:  
      name: nat
```

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

#### Dockerfile

```
FROM mcr.microsoft.com/dotnet/sdk:6.0 AS build  
WORKDIR /app  
  
COPY *.sln .  
COPY ..  
WORKDIR /app/src/Web  
RUN dotnet restore  
  
RUN dotnet publish -c Release -o out  
  
FROM mcr.microsoft.com/dotnet/aspnet:6.0 AS runtime  
WORKDIR /app  
COPY --from=build /app/src/Web/out ./  
  
ENTRYPOINT ["dotnet", "Web.dll"]
```

## Solución de problemas de Docker

Una vez que se ejecute la aplicación en contenedores, se sigue ejecutando hasta que se detenga. Con el comando `docker ps` puede ver qué contenedores se están ejecutando. Puede detener un contenedor en ejecución si usa el comando `docker stop` y especifica el identificador del contenedor.

Tenga en cuenta que los contenedores de Docker en ejecución pueden estar enlazados a puertos que, en otros casos, es posible que intentara usar en el entorno de desarrollo. Si intenta ejecutar o depurar una aplicación con el mismo puerto que un contenedor de Docker en ejecución, obtendrá un error que indica que el servidor no se puede enlazar a ese puerto. Una vez más, detener el contenedor debería resolver el problema.

Si quiere agregar compatibilidad con Docker a la aplicación mediante Visual Studio, asegúrese de que Docker Desktop se esté ejecutando. Si Docker Desktop no está funcionando cuando se inicia el asistente, el asistente no se ejecutará correctamente. Además, el asistente examinará el contenedor que ha elegido actualmente para agregar la compatibilidad correcta con Docker. Si quiere agregar compatibilidad con contenedores de Windows, debe ejecutar el asistente mientras Docker Desktop se ejecuta con contenedores de Windows configurados. Si quiere agregar compatibilidad con contenedores de Linux, ejecute el asistente mientras Docker se ejecuta con contenedores de Linux configurados.

## Otros estilos de arquitectura de aplicación web

- **Web-Cola-Trabajo:** los componentes principales de esta arquitectura son un front-end web que atiende solicitudes de cliente y un trabajo que realiza tareas que consumen muchos recursos, flujos de trabajo de ejecución prolongada o trabajos por lotes. El front-end web se comunica con el trabajo a través de una cola de mensajes.
- **Niveles:** una arquitectura de niveles divide una aplicación en capas lógicas y niveles físicos.
- **Microservicio:** una arquitectura de microservicios consta de una colección de servicios autónomos y pequeños. Cada uno de servicio es independiente y debe implementar una funcionalidad de negocio individual dentro de un contexto delimitado.

## Referencias: arquitecturas web comunes

- **The Clean Architecture** (La arquitectura limpia)  
[https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html ↗](https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html)
- **The Onion Architecture** (La arquitectura cebolla)  
[https://jeffreypalermo.com/blog/the-onion-architecture-part-1/ ↗](https://jeffreypalermo.com/blog/the-onion-architecture-part-1/)
- **The Repository Pattern** (El modelo de repositorio)  
[https://deviq.com/repository-pattern/ ↗](https://deviq.com/repository-pattern/)
- **Plantilla de solución de arquitectura limpia**  
[https://github.com/ardalis/cleanarchitecture ↗](https://github.com/ardalis/cleanarchitecture)
- **Architecting Microservices e-book** (Libro electrónico de arquitectura de microservicios)  
[https://aka.ms/MicroservicesEbook ↗](https://aka.ms/MicroservicesEbook)
- **DDD (diseño guiado por el dominio)**  
<https://learn.microsoft.com/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/>

[Anterior](#)

[Siguiente](#)

# Tecnologías web comunes del lado cliente

Artículo • 28/11/2022 • Tiempo de lectura: 14 minutos

## 💡 Sugerencia

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

[Descargar PDF](#)



"Los sitios web deben tener una apariencia correcta, tanto interna como externa." -  
*Paul Cookson*

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

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

### Nota

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

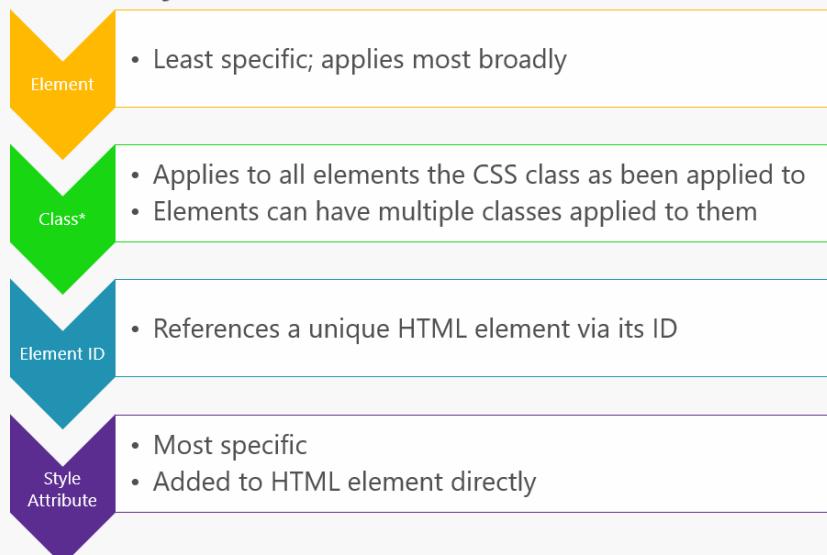
## HTML

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

## CSS

CSS (Hoja de estilos en cascada) se usa para controlar la apariencia y el diseño de los elementos HTML. Los estilos CSS se pueden aplicar directamente a un elemento HTML, o bien definirse por separado en la misma página o en un archivo independiente al que la página haga referencia. Los estilos se aplican en cascada en función de cómo se usan para seleccionar un elemento HTML determinado. Por ejemplo, es posible que un estilo se aplique a todo el documento, pero que se reemplace por un estilo que se aplica a un elemento determinado. Del mismo modo, un estilo específico del elemento se reemplazaría por un estilo que se aplica a una clase CSS aplicada al elemento, que a su vez se reemplazaría por un estilo destinado a una instancia específica de ese elemento (a través de su identificador). Figura 6-1

# CSS Specificity



\* Attribute and pseudo-class selectors also apply at this level

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

Se recomienda mantener los estilos en sus propios archivos de hoja de estilos independientes y aplicarlos en cascada en función de la selección para implementar estilos coherentes y reutilizables dentro de la aplicación. Se debe evitar colocar las reglas de estilo en el código HTML y aplicar estilos a elementos individuales específicos (en lugar de clases completas de elementos, o bien elementos a los que se ha aplicado una clase CSS determinada) debería ser la excepción, no la regla.

## Preprocesadores CSS

Las hojas de estilo CSS carecen de compatibilidad con la lógica condicional, las variables y otras características de los lenguajes de programación. Por tanto, las hojas de estilo grandes suelen incluir bastantes repeticiones, ya que el mismo color, fuente u otra configuración se aplica a distintas variaciones de elementos HTML y clases CSS. Los preprocesadores CSS pueden ayudar a que las hojas de estilo sigan el [Principio DRY](#) agregando compatibilidad para las variables y la lógica.

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

# JavaScript

JavaScript es un lenguaje de programación interpretado y dinámico que se ha estandarizado en la especificación del lenguaje ECMAScript. Es el lenguaje de programación de la web. Como CSS, JavaScript se puede definir como atributos dentro de los elementos HTML, como bloques de script dentro de una página o en archivos independientes. Al igual que CSS, se recomienda organizar JavaScript en archivos independientes y, en la medida de lo posible, mantenerlos separados del código HTML que se encuentra en las páginas web individuales o vistas de la aplicación.

Cuando se trabaja con JavaScript en la aplicación web, hay algunas tareas que normalmente es necesario realizar:

- Seleccionar un elemento HTML y recuperar o actualizar su valor.
- Consultar datos en una API web.
- Enviar un comando a una API web (y responder a una devolución de llamada con su resultado).
- Realizar la validación.

Todas estas tareas se pueden realizar con JavaScript por sí solo, pero existen muchas bibliotecas para facilitarlas. Una de las primeras de estas bibliotecas y de mayor éxito es jQuery, que sigue siendo una opción popular para simplificar estas tareas en las páginas web. Para aplicaciones de página única (SPA), jQuery no proporciona muchas de las características deseadas que ofrecen Angular y React.

## Aplicaciones web heredadas con jQuery

Aunque de acuerdo con los estándares de marco de trabajo de JavaScript se considere antigua, jQuery sigue siendo una biblioteca de uso común para trabajar con HTML y CSS, y crear aplicaciones que realizan llamadas AJAX a las API web. Pero jQuery funciona en el nivel del modelo de objetos de documento (DOM) del explorador y, de forma predeterminada, ofrece un modelo imperativo, en lugar de declarativo.

Por ejemplo, imagine que, si el valor de un cuadro de texto es superior a 10, se deba mostrar un elemento en la página. En jQuery, esta función normalmente se implementaría mediante la escritura de un controlador de eventos con código que inspeccionaría el valor del cuadro de texto y establecería la visibilidad del elemento de destino en función de ese valor. Este proceso es un enfoque imperativo basado en código. Es posible que, en su lugar, otro marco de trabajo usara el enlace de datos para enlazar mediante declaración la visibilidad del elemento al valor del cuadro de texto. Este enfoque no requeriría escribir código, solo habría que decorar los elementos implicados con atributos de enlace de datos. Cuando aumenta la complejidad de los

comportamientos del lado cliente, los enfoques de enlace de datos suelen dar como resultado soluciones más simples con menos código y complejidad condicional.

## Diferencias entre jQuery y un marco de trabajo de SPA

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

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

El enlace de datos es un buen ejemplo de esta función. En jQuery, normalmente basta con una línea de código para obtener el valor de un elemento DOM, o bien para establecer el valor de un elemento. Pero tendrá que escribir este código siempre que necesite cambiar el valor del elemento y, a veces, esto ocurrirá en varias funciones de una página. Otro ejemplo común es la visibilidad de los elementos. En jQuery, es posible que haya muchos lugares diferentes en los que tendría que escribir código para controlar si determinados elementos son visibles. En cada uno de estos casos, cuando se usa el enlace de datos, no sería necesario escribir código. Simplemente se podría enlazar el valor o la visibilidad de los elementos en cuestión a un *modelo de vista* en la página y los cambios en ese modelo de vista se reflejarán automáticamente en los elementos enlazados.

## SPA de Angular

Angular sigue siendo uno de los marcos de JavaScript más populares del mundo. Desde Angular 2, el equipo recompiló el marco desde cero (con [TypeScript](#)) y cambió el

nombre de AngularJS original a Angular. El Angular rediseñado, que ya tiene varios años, sigue siendo un marco robusto para la creación de aplicaciones de página única.

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

```
JavaScript

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

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

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

Los componentes se definen mediante la función decorador `@Component`, que acepta metadatos sobre el componente. La propiedad `selector` identifica el identificador del elemento en la página donde se va a mostrar este componente. La propiedad `template` es una plantilla HTML sencilla que incluye un marcador de posición que corresponde a la propiedad `name` del componente, definida en la última línea.

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

Puede desarrollar aplicaciones de Angular mediante una CLI. La introducción al desarrollo local de Angular (suponiendo que ya se haya instalado npm y git) consiste en clonar simplemente un repositorio de GitHub y ejecutar `npm install` y `npm start`.

Aparte de esto, Angular suministra su propia CLI que puede crear proyectos, agregar archivos y ayudar con las tareas de pruebas, agrupación e implementación. Esta facilidad de uso de la CLI hace que Angular sea especialmente compatible con ASP.NET Core, que también incluye una excelente compatibilidad con la CLI.

Microsoft ha desarrollado una aplicación de referencia, eShopOnContainers, que incluye una implementación de SPA de Angular. Esta aplicación incluye módulos de Angular

para administrar la cesta de la compra de la tienda en línea, cargar y presentar los artículos del catálogo y controlar la creación de pedidos. Puede ver y descargar la aplicación de ejemplo en [GitHub](#).

## React

A diferencia de Angular, que ofrece una implementación completa del modelo Model-View-Controller, React solo se ocupa de las vistas. No es un marco de trabajo, solo es una biblioteca, de modo que para compilar una SPA se debe recurrir a bibliotecas adicionales. Hay una serie de bibliotecas que están diseñadas para usarse con React con el fin de generar aplicaciones de una sola página enriquecidas.

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

El funcionamiento de React con HTML también es muy particular. En lugar de tener una separación estricta entre el código y el marcado (posiblemente con referencias a JavaScript en los atributos HTML), React agrega HTML directamente en su código de JavaScript como JSX. JSX es la sintaxis de estilo HTML que se puede compilar hasta JavaScript puro. Por ejemplo:

JavaScript

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

Si ya conoce JavaScript, el aprendizaje de React debería ser sencillo. No hay tanta curva de aprendizaje o una sintaxis especial como sucede con Angular u otras bibliotecas populares.

Como React no es un marco de trabajo completo, normalmente le interesarán otras bibliotecas para controlar aspectos como el enrutamiento, las llamadas a API web y la administración de dependencias. Lo mejor es que se puede elegir la mejor biblioteca para cada uno de estos aspectos, pero el inconveniente es que será necesario tomar todas estas decisiones y comprobar que todas las bibliotecas elegidas funcionan bien de forma conjunta cuando se haya terminado. Si quiere un buen punto de partida, puede

usar un starter kit como React Slingshot, en el que se preempaquesta un conjunto de bibliotecas compatibles con React.

## Vue

En su guía de introducción se dice que Vue es un marco progresivo para la creación de interfaces de usuario. A diferencia de otros marcos monolíticos, Vue se ha diseñado desde el principio para que se pueda adoptar de forma incremental. La biblioteca principal solo se centra en el nivel de vista y es fácil de adoptar e integrar con otras bibliotecas o proyectos existentes. Por otro lado, Vue es perfectamente capaz de potenciar aplicaciones de una sola página sofisticadas al usarlo en combinación con herramientas modernas y bibliotecas auxiliares.

Empezar a usar Vue solo requiere incluir su script en un archivo HTML:

HTML

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

Con el marco agregado, podrá representar los datos de forma declarativa en el DOM mediante la sintaxis de plantillas sencillas de Vue:

HTML

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

Después, agregue el siguiente script:

JavaScript

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

Esto es suficiente para representarse "Hello Vue!" en la página. Sin embargo, tenga en cuenta que Vue no representa simplemente el mensaje incluido en el elemento "div" una vez. Admite el enlace de datos y las actualizaciones dinámicas, de modo que, si el

valor de `message` cambia, el valor de `<div>` se actualizará inmediatamente para reflejar dicho cambio.

Por supuesto, esto es solo una muy pequeña parte de lo que es capaz de hacer Vue. Se ha vuelto muy popular en los últimos años y cuenta con el respaldo de una comunidad bastante grande. También hay una lista [enorme y creciente de bibliotecas y componentes de apoyo](#) que funcionan con Vue y permiten ampliarlo. Si quiere agregar un comportamiento de lado cliente a la aplicación web o está pensando en crear una SPA completa, merece la pena valorar Vue.

## Blazor WebAssembly

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

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

El código de WebAssembly puede acceder a toda la funcionalidad del explorador mediante JavaScript, denominada interoperabilidad de JavaScript. El código de .NET que se ejecuta a través de WebAssembly en el explorador se ejecuta a su vez en el espacio aislado de JavaScript del explorador con las protecciones que proporciona dicho espacio aislado contra acciones malintencionadas en la máquina cliente.

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

## Elección de un marco de trabajo de SPA

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

- ¿El equipo está familiarizado con el marco de trabajo y sus dependencias (incluido TypeScript en algunos casos)?
- ¿Qué opiniones suscita el marco de trabajo; está de acuerdo con su modo predeterminado de hacer las cosas?

- ¿Incluye (o bien una biblioteca complementaria) todas las características que requiere la aplicación?
- ¿Está bien documentado?
- ¿Qué nivel de actividad tiene su comunidad? ¿Se compilan proyectos nuevos con él?
- ¿Qué nivel de actividad tiene su equipo principal? ¿Se resuelven los problemas y se publican periódicamente versiones nuevas?

Los marcos siguen evolucionando a una velocidad de vértigo. Use las consideraciones enumeradas anteriormente para ayudar a mitigar el riesgo de elegir un marco del que más adelante se arrepienta de depender. Si es especialmente reacio a los riesgos, considere la posibilidad de un marco de trabajo que ofrezca soporte técnico comercial o esté desarrollado por una gran empresa.

## Referencias: tecnologías web cliente

- **HTML y CSS**  
<https://www.w3.org/standards/webdesign/htmlcss> ↗
- **Sass vs. LESS** (Diferencias entre Sass y LESS)  
<https://www.keycdn.com/blog/sass-vs-less/> ↗
- **Aplicación de estilos a aplicaciones ASP.NET Core con LESS, Sass y Font Awesome**  
<https://learn.microsoft.com/aspnet/core/client-side/less-sass-fontawesome>
- **Desarrollo del lado cliente en ASP.NET Core**  
<https://learn.microsoft.com/aspnet/core/client-side/>
- **jQuery**  
<https://jquery.com/> ↗
- **jQuery vs AngularJS** (Diferencias entre jQuery y AngularJS)  
<https://www.airpair.com/angularjs/posts/jquery-angularjs-comparison-migration-walkthrough> ↗
- **Angular**  
<https://angular.io/> ↗
- **React**  
<https://reactjs.org/> ↗
- **Vue**  
<https://vuejs.org/> ↗
- **Comparación entre Angular, React y Vue: qué marco elegir en 2020**  
<https://www.codeinwp.com/blog/angular-vs-vue-vs-react/> ↗

- Principales marcos de JavaScript para el desarrollo de front-end en 2020

[https://www.freecodecamp.org/news/complete-guide-for-front-end-developers-javascript-frameworks-2019/ ↗](https://www.freecodecamp.org/news/complete-guide-for-front-end-developers-javascript-frameworks-2019/)

Anterior

Siguiente

# Desarrollo de aplicaciones ASP.NET Core MVC

Artículo • 10/03/2023 • Tiempo de lectura: 47 minutos

## 💡 Sugerencia

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

[Descargar PDF](#)



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

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

## MVC y Razor Pages

ASP.NET Core MVC ofrece muchas características que son útiles para crear API y aplicaciones basadas en web. El término MVC significa "Modelo-Vista-Controlador", un patrón de interfaz de usuario que divide las responsabilidades de responder a las

solicitudes de los usuarios en varias partes. Además de seguir este patrón, también puede implementar características en sus aplicaciones ASP.NET Core, como Razor Pages.

Razor Pages está integrado en ASP.NET Core MVC y usa las mismas características de enrutamiento, enlace de modelos, filtros, autorización, etcétera. Pero, en lugar de tener archivos y carpetas independientes para los controladores, los modelos, las vistas y otros elementos, y usar el enrutamiento basado en atributos, Razor Pages se coloca en una sola carpeta ("~/Páginas"). La ruta se basa en su ubicación relativa en la carpeta, y las solicitudes se controlan mediante controladores en lugar de acciones de controlador. Como resultado, al trabajar con Razor Pages, todos los archivos y clases que necesita se colocan normalmente, no se distribuyen por todo el proyecto web.

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

Al crear una aplicación ASP.NET Core, debe tener un plan en cuenta para el tipo de aplicación que quiera crear. Al crear un proyecto, en el IDE o mediante el comando `dotnet new` de la CLI, elegirá entre varias plantillas. Las plantillas de proyecto más comunes son Vacía, API web, Aplicación web y Aplicación web (Modelo-Vista-Controlador). Aunque solo puede tomar esta decisión al crear un proyecto, no es irrevocable. El proyecto API web usa controladores de Modelo-Vista-Controlador estándar; simplemente carece de las vistas de forma predeterminada. Del mismo modo, la plantilla predeterminada Aplicación web usa Razor Pages, por lo que tampoco incluye la carpeta de vistas. Puede agregar la carpeta de vistas para estos proyectos más adelante para admitir el comportamiento basado en vistas. Los proyectos de tipo API web y Modelo-Vista-Controlador no incluyen la carpeta de páginas de forma predeterminada, pero puede agregar una más tarde para admitir el comportamiento basado en Razor Pages. Estos tres tipos de plantilla están diseñados para tres tipos de interacción predeterminada del usuario distintos: datos (API web), basada en páginas y basada en vistas. Si quiere, puede combinar cualquiera de estas plantillas, o todas ellas, en un mismo proyecto.

## ¿Por qué Razor Pages?

Razor Pages es el método predeterminado para nuevas aplicaciones web en Visual Studio. Razor Pages ofrece una manera más fácil de crear características de aplicaciones basadas en páginas, como los formularios que no son de aplicaciones de página única. Al usar controladores y vistas, era habitual que las aplicaciones tuvieran controladores muy grandes que funcionaban con varias dependencias y varios modelos de vista distintos, así como que devolvieran muchas vistas diferentes. Esto generaba una mayor complejidad y, a menudo, daba lugar a controladores que no seguían el principio de

responsabilidad única o los principios de apertura y cierre de forma eficaz. Razor Pages soluciona este problema, ya que encapsula la lógica del lado servidor para una determinada "página" lógica en una aplicación web con el marcado de Razor. Una página de Razor Pages que no tenga ninguna lógica del lado servidor solo puede constar de un archivo de Razor (por ejemplo, "Index.cshtml"). Sin embargo, la mayoría de Razor Pages no triviales tendrá una clase de modelo de página asociado, que, por convención, tendrá el mismo nombre que el archivo Razor con una extensión ".cs" (por ejemplo, "Index.cshtml.cs").

Un modelo de página de Razor Pages combina las responsabilidades de un controlador de MVC y un modelo de vista. En lugar de controlar las solicitudes con los métodos de acción de controlador, se ejecutan los controladores de modelo de página, como "OnGet()", con lo que se representa la página asociada de forma predeterminada. Razor Pages simplifica el proceso de compilar páginas individuales en la aplicación ASP.NET Core sin dejar de proporcionar todas las características arquitectónicas de ASP.NET Core MVC. Es una buena opción predeterminada para la nueva funcionalidad basada en páginas.

## Cuándo usar MVC

Si va a crear interfaces API web, el patrón MVC tiene más sentido que Razor Pages. Si el proyecto va a exponer solo los puntos de conexión de la API Web, lo ideal es empezar a partir de la plantilla de proyecto de API web. De lo contrario, es fácil agregar controladores y puntos de conexión de API asociados a cualquier aplicación ASP.NET Core. Use el método MVC basado en vista si quiere realizar la migración de una aplicación ASP.NET MVC 5 existente o versiones anteriores a ASP.NET Core MVC y quiere hacerlo de la forma más fácil posible. Una vez que haya realizado la migración inicial, podrá evaluar si tiene sentido adoptar Razor Pages para las nuevas características o incluso como migración general. Para obtener más información sobre cómo migrar las aplicaciones de .NET 4.x a .NET 7, vea [Migración de aplicaciones existentes de ASP.NET a .NET Core](#).

Si opta por crear su aplicación web mediante Razor Pages o las vistas de MVC, la aplicación tendrá un rendimiento similar e incluirá compatibilidad con la inserción de dependencias, los filtros, el enlace de modelos, la validación y otras características.

## Asignación de solicitudes a respuestas

En su núcleo, las aplicaciones ASP.NET Core asignan las solicitudes entrantes a las respuestas salientes. En un nivel bajo, esta asignación se realiza mediante middleware, y las aplicaciones y microservicios sencillos de ASP.NET Core pueden constar únicamente

de middleware personalizado. Cuando se usa ASP.NET Core MVC, se puede trabajar en un cierto nivel superior y pensar en términos de *rutas*, *controladores* y *acciones*. Cada solicitud entrante se compara con la tabla de enrutamiento de la aplicación y, si se encuentra una ruta coincidente, se llama al método de acción asociado (perteneciente a un controlador) para controlar la solicitud. Si no se encuentra ninguna ruta que coincida, se llama a un controlador de errores (en este caso, se devuelve un resultado de `NotFound`).

Las aplicaciones ASP.NET Core MVC pueden usar rutas convencionales, rutas de atributo o las dos. Las rutas convencionales se definen en el código, especificando *convenciones* de enrutamiento con una sintaxis similar a la del ejemplo siguiente:

C#

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

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

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

C#

```
[Route("Home")]
public class HomeController : Controller
{
    [Route("")]] // Combines to define the route template "Home"
```

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

Las rutas se pueden especificar en atributos [HttpGet] y similares, evitando la necesidad de agregar atributos [Route] independientes. Las rutas de atributo también pueden usar tokens para reducir la necesidad de repetir los nombres de acciones o controladores, como se muestra a continuación:

C#

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

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

C#

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

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

HTTP

```
/Products/123
```

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

consulta. Suponiendo que se encuentre un valor `id`, se convertirá en un entero antes de pasarlo al método de acción.

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

Si se usa la validación del modelo, siempre se debe comprobar que el modelo es válido antes de ejecutar cualquier comando de modificación del estado, para asegurarse de que la aplicación no resulta dañada por datos no válidos. Se puede usar un [filtro](#) para evitar la necesidad de agregar código para esta validación en todas las acciones. Los filtros de ASP.NET Core MVC ofrecen una manera de interceptar grupos de solicitudes, para poder aplicar directivas comunes e intereses transversales por cada destino. Los filtros se pueden aplicar a acciones individuales, controladores completos o de forma global para una aplicación.

Para las API web, ASP.NET Core MVC admite la [negociación de contenido](#), lo que permite a las solicitudes especificar cómo se debe aplicar formato a las respuestas. Según los encabezados proporcionados en la solicitud, las acciones que devuelven datos darán formato a la respuesta en XML, JSON o en otro formato compatible. Esta característica permite usar la misma API en varios clientes con requisitos de formato de datos diferentes.

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

## Mantenimiento de los controladores bajo control

En el caso de las aplicaciones basadas en páginas, Razor Pages hace un gran trabajo evitando que los controladores se vuelvan demasiado grandes. Cada página tiene sus propios archivos y clases dedicados únicamente para sus controladores. Antes de la

introducción de Razor Pages, muchas aplicaciones centradas en las vistas tenían grandes clases de controladores responsables de muchas acciones y vistas diferentes. Estas clases crecían de forma natural hasta tener muchas responsabilidades y dependencias, lo que dificultaba el mantenimiento. Si sus controladores basados en vistas están creciendo demasiado, considere la posibilidad de refactorizarlos para usar Razor Pages o de incluir un patrón como mediador.

El patrón de diseño de mediador se usa para reducir el acoplamiento entre las clases y permitir la comunicación entre ellas. En las aplicaciones de ASP.NET Core MVC, este patrón suele emplearse para dividir los controladores en partes más pequeñas mediante el uso de *rutinas de controladores* para realizar el trabajo de los métodos de acción. El popular paquete NuGet [MediatR](#) se usa a menudo para lograr esto. Normalmente, los controladores incluyen muchos métodos de acción diferentes, y cada uno puede requerir ciertas dependencias. El conjunto de todas las dependencias requeridas por cualquier acción debe transferirse al constructor del controlador. Cuando se usa MediatR, la única dependencia que un controlador tendrá normalmente es una instancia del mediador. Cada acción usa la instancia del mediador para enviar un mensaje, el cual es procesado por un controlador. El controlador es específico para una única acción y, por tanto, solo necesita las dependencias que requiere dicha acción. A continuación se muestra un ejemplo de un controlador que usa MediatR:

```
C#  
  
public class OrderController : Controller  
{  
    private readonly IMediator _mediator;  
  
    public OrderController(IMediator mediator)  
    {  
        _mediator = mediator;  
    }  
  
    [HttpGet]  
    public async Task<IActionResult> MyOrders()  
    {  
        var viewModel = await _mediator.Send(new  
GetMyOrders(User.Identity.Name));  
        return View(viewModel);  
    }  
    // other actions implemented similarly  
}
```

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

```
C#
```

```

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

    public async Task<IEnumerable<OrderViewModel>> Handle(GetMyOrders request,
CancellationToken cancellationToken)
    {
        var specification = new
CustomerOrdersWithItemsSpecification(request.UserName);
        var orders = await _orderRepository.ListAsync(specification);
        return orders.Select(o => new OrderViewModel
        {
            OrderDate = o.OrderDate,
            OrderItems = o.OrderItems?.Select(oi => new
OrderItemViewModel()
            {
                PictureUrl = oi.ItemOrdered.PictureUri,
                ProductId = oi.ItemOrdered.CatalogItemId,
                ProductName = oi.ItemOrdered.ProductName,
                UnitPrice = oi.UnitPrice,
                Units = oi.Units
            }).ToList(),
            OrderNumber = o.Id,
            ShippingAddress = o.ShipToAddress,
            Total = o.Total()
        });
    }
}

```

El resultado final de este enfoque es que los controladores deben ser mucho más pequeños y estar centrados principalmente en el enrutamiento y el enlace de modelos; mientras que las rutinas de controladores individuales son responsables de las tareas específicas que necesita un punto de conexión determinado. Este enfoque también se puede lograr sin MediatR mediante el uso del [paquete NuGet ApiEndpoints](#), que intenta proporcionar a los controladores de API las mismas ventajas que proporciona Razor Pages a los controladores basados en vistas.

## Referencias: asignación de solicitudes a respuestas

- **Routing to Controller Actions (Enrutamiento a acciones del controlador)**  
<https://learn.microsoft.com/aspnet/core/mvc/controllers/routing>
- **Enlace de modelos**  
<https://learn.microsoft.com/aspnet/core/mvc/models/model-binding>

- **Introduction to model validation in ASP.NET Core MVC** (Introducción a la validación de modelos en ASP.NET Core MVC)  
<https://learn.microsoft.com/aspnet/core/mvc/models/validation>
- **Filtros**  
<https://learn.microsoft.com/aspnet/core/mvc/controllers/filters>
- **Atributo ApiController**  
<https://learn.microsoft.com/aspnet/core/web-api/>

## Trabajar con dependencias

ASP.NET Core tiene compatibilidad integrada con una técnica conocida como [inserción de dependencias](#) y la usa de manera interna. La inserción de dependencias es una técnica que permite el acoplamiento flexible entre los distintos elementos de una aplicación. El acoplamiento más flexible es deseable porque facilita aislar los elementos de la aplicación, lo que permite realizar pruebas o reemplazos. También hace que sea menos probable que un cambio en un elemento de la aplicación tenga un impacto inesperado en otro lugar de la aplicación. La inserción de dependencias se basa en el principio de inversión de dependencias y suele ser es clave para lograr el principio abierto o cerrado. Al evaluar el funcionamiento de la aplicación con sus dependencias, tenga cuidado del problema de la [estática](#) en el código y recuerde el aforismo "[lo nuevo se pega](#)".

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

Muchos desarrolladores comprenden los riesgos de la estática y el estado global, pero siguen acoplando estrechamente el código a implementaciones específicas a través de la creación directa de instancias. "Lo nuevo se pega" está pensado para ser un aviso de este acoplamiento y no un rechazo general del uso de la palabra clave `new`. Como sucede con las llamadas de métodos estáticos, las instancias nuevas de tipos que no tienen dependencias externas normalmente no acoplan estrechamente código a los

detalles de implementación ni dificultan las pruebas. Pero cada vez que se crea una instancia de una clase, dedique un instante a considerar si tiene sentido integrar como parte del código esa instancia específica en esa ubicación concreta, o si un mejor diseño sería solicitar esa instancia como una dependencia.

## Declarar las dependencias

ASP.NET Core se basa en hacer que los métodos y las clases declaren sus dependencias y las soliciten como argumentos. Las aplicaciones de ASP.NET normalmente se configuran en *Program.cs* o en una clase `Startup`.

### ⓘ Nota

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

## Configuración de servicios en *Program.cs*

Para aplicaciones muy sencillas, puede conectar dependencias directamente en el archivo *Program.cs* mediante `WebApplicationBuilder`. Una vez que se han agregado todos los servicios necesarios, el generador se usa para crear la aplicación.

C#

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

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

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

## Configuración de servicios en *Startup.cs*

*Startup.cs* está configurado para admitir la inserción de dependencias en varios puntos. Si usa una clase `Startup`, puede asignarle un constructor y es posible solicitar dependencias a través de él, de la siguiente manera:

C#

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

La clase `Startup` es interesante en el sentido de que no hay requisitos de tipos explícitos para ella. No hereda de una clase base `Startup` especial, ni implementa ninguna interfaz determinada. Se le puede asignar un constructor, o no, y se pueden especificar tantos parámetros en el constructor como se quiera. Cuando se inicia el host web configurado para la aplicación, llamará a la clase `Startup` (si le ha indicado que use una) y usará la inserción de dependencias para llenar todas las dependencias que la clase `Startup` requiera. Por supuesto, si se solicitan parámetros que no están configurados en el contenedor de servicios que usa ASP.NET Core, se obtendrá una excepción, pero siempre que se respeten las dependencias que el contenedor conoce, se puede solicitar lo que se quiera.

La inserción de dependencias se integra en las aplicaciones ASP.NET Core desde el principio, cuando se crea la instancia de la clase de inicio. No se detiene ahí para la clase de inicio. Las dependencias también se pueden solicitar en el método `Configure`:

C#

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

}
```

El método `ConfigureServices` es la excepción a este comportamiento; solo debe tomar un parámetro de tipo `IServiceCollection`. Realmente no necesita admitir la inserción de dependencias, ya que por un lado es responsable de agregar objetos al contenedor de servicios y, por otro, tiene acceso a todos los servicios configurados actualmente a través del parámetro `IServiceCollection`. Por tanto, se puede trabajar con las dependencias definidas en la colección de servicios de ASP.NET Core en todos los

elementos de la clase `Startup`, ya sea solicitando el servicio necesario como un parámetro o trabajando con `IServiceCollection` en `ConfigureServices`.

### ⓘ Nota

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

La clase de inicio es un modelo de cómo estructurar otros elementos de la aplicación ASP.NET Core, desde controladores a software intermedio o filtros para sus propios servicios. En cada caso, se debe seguir el [principio de dependencias explícitas ↗](#), y solicitar las dependencias en lugar de crearlas directamente, y aprovechar la inserción de dependencias en toda la aplicación. Tenga cuidado de dónde y cómo se crean instancias directas de las implementaciones, especialmente de servicios y objetos que funcionan con la infraestructura o tienen efectos secundarios. Es preferible trabajar con abstracciones definidas en el núcleo de la aplicación y pasadas como argumentos que integrar las referencias como parte del código en tipos de implementación específicos.

## Estructuración de la aplicación

Las aplicaciones monolíticas suelen tener un único punto de entrada. En el caso de una aplicación web ASP.NET Core, el punto de entrada será el proyecto web ASP.NET Core. Pero eso no significa que la solución deba constar simplemente de un solo proyecto. Resulta útil dividir la aplicación en diferentes capas para seguir la separación de intereses. Una vez dividida en capas, resulta útil ir más allá de las carpetas para separar los proyectos, lo que puede ayudar a conseguir una mejor encapsulación. El mejor método para lograr estos objetivos con una aplicación ASP.NET Core consiste en una variación de la arquitectura limpia que se describe en el capítulo 5. A raíz de este enfoque, la solución de la aplicación estará compuesta por bibliotecas independientes para la interfaz de usuario, la infraestructura y `ApplicationCore`.

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

El modelo de objetos y las interfaces de la aplicación se deben colocar en el proyecto `ApplicationCore`. Este proyecto tendrá el menor número posible de dependencias, y ninguna en cuestiones específicas de la infraestructura, y los otros proyectos de la solución le harán referencia. Las entidades de negocio que deban conservarse se

definen en el proyecto `ApplicationCore`, al igual que los servicios que no dependen directamente de la infraestructura.

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

El propio proyecto de interfaz de usuario de ASP.NET Core es responsable de cualquier interés del nivel de la interfaz de usuario, pero no debe incluir lógica empresarial o detalles de infraestructura. De hecho, idealmente ni siquiera debería tener una dependencia en el proyecto de infraestructura, lo que ayudará a garantizar que no se introduce por accidente ninguna dependencia entre los dos proyectos. Esto se puede lograr mediante un contenedor de DI de terceros, como Autofac, que permite definir reglas de DI en clases del módulo en cada proyecto.

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

## Organización de las características

De forma predeterminada, las aplicaciones ASP.NET Core organizan su estructura de carpetas para que incluir controladores y vistas, y con frecuencia `ViewModels`. El código del lado cliente para admitir estas estructuras del lado servidor normalmente se almacena por separado en la carpeta `wwwroot`. Pero es posible que esta organización genere problemas en aplicaciones grandes, puesto que trabajar con cualquier característica determinada a menudo requiere saltar entre estas carpetas. Esto se complica cada vez más a medida que aumenta el número de archivos y subcarpetas en cada carpeta, lo que da lugar a una gran cantidad de desplazamiento por el Explorador de soluciones. Una solución a este problema consiste en organizar el código de aplicación por *característica* en lugar de por tipo de archivo. Este estilo profesional se conoce normalmente como carpetas de características o [sectores de características](#) (vea también: [Vertical Slice ↗](#) (Sectores verticales)).

ASP.NET Core MVC admite las áreas para este propósito. Con las áreas, se pueden crear conjuntos independientes de carpetas de controladores y vistas (así como los modelos asociados) en cada carpeta de área. En la figura 7-1 se muestra una estructura de carpetas de ejemplo, en la que se usan áreas.

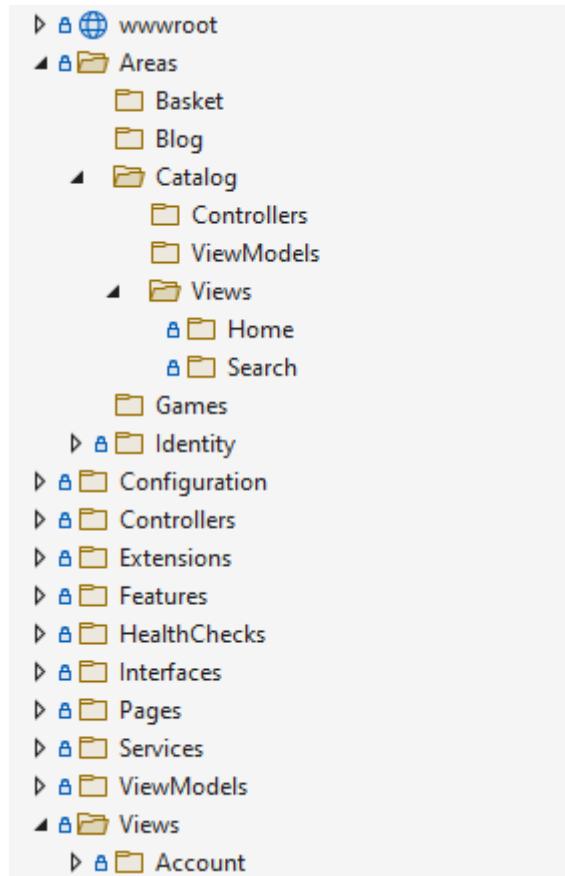


Figura 7-1. Organización de áreas de ejemplo

Cuando se usan las áreas, se deben usar atributos para decorar los controladores con el nombre del área a la que pertenecen:

```
C#  
  
[Area("Catalog")]
public class HomeController
{}
```

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

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

```
{controller=Home}/{action=Index}/{id?}");  
});
```

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

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

C#

```
public class FeatureConvention : IControllerModelConvention  
{  
    public void Apply(ControllerModel controller)  
    {  
        controller.Properties.Add("feature",  
            GetFeatureName(controller.ControllerType));  
    }  
  
    private string GetFeatureName(TypeInfo controllerType)  
    {  
        string[] tokens = controllerType.FullName.Split('.');  
        if (!tokens.Any(t => t == "Features")) return "";  
        string featureName = tokens  
            .SkipWhile(t => !t.Equals("features",  
StringComparison.CurrentCultureIgnoreCase))  
            .Skip(1)  
            .Take(1)  
            .FirstOrDefault();  
        return featureName;  
    }  
}
```

Después, esta convención se especifica como una opción al agregar compatibilidad para MVC a la aplicación en `ConfigureServices`, o bien en `Program.cs`:

C#

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

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

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

## API y aplicaciones Blazor

Si la aplicación incluye un conjunto de API web que se deba proteger, estas API idealmente se deben configurar como un proyecto independiente de la aplicación de vista o de Razor Pages. La separación de las API, sobre todo las API públicas, de la aplicación web del lado servidor tiene una serie de ventajas. Estas aplicaciones a menudo tendrán características de implementación y carga únicas. También es muy probable que adopten diferentes mecanismos de seguridad; las aplicaciones estándar basadas en formularios aprovechan la autenticación basada en cookies y las API probablemente usan la autenticación basada en tokens.

Además, las aplicaciones Blazor, tanto si usan Blazor Server como BlazorWebAssembly, deben compilarse como proyectos independientes. Estas aplicaciones tienen características de entorno de ejecución y modelos de seguridad diferentes. Es probable que comparten tipos comunes con la aplicación web del lado servidor (o el proyecto de API), y estos tipos se deben definir en un proyecto compartido común.

La adición de una interfaz de administrador de BlazorWebAssembly a eShopOnWeb hizo necesaria la adición de varios proyectos nuevos: El propio proyecto de BlazorWebAssembly, `BlazorAdmin`; el proyecto `PublicApi`, donde se define un nuevo conjunto de puntos de conexión de API pública usados por `BlazorAdmin` y configurados para usar la autenticación basada en tokens; y un nuevo proyecto `BlazorShared` que almacena ciertos tipos compartidos que se usan en los dos proyectos anteriores.

Podría preguntarse por qué es necesario agregar un proyecto `BlazorShared` independiente cuando ya existe un proyecto `ApplicationCore` común que podría usarse para compartir los tipos que requieren tanto `PublicApi` como `BlazorAdmin`. La respuesta es que este proyecto incluye toda la lógica empresarial de la aplicación y, por tanto, es

mucho mayor de lo necesario y es mucho más probable que deba mantenerse protegido en el servidor. Recuerde que cualquier biblioteca a la que haga referencia `BlazorAdmin` se descargará en los exploradores de los usuarios cuando carguen la aplicación Blazor.

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

## Intereses transversales

A medida que las aplicaciones aumentan de tamaño, resulta cada vez más importante separar los intereses transversales para eliminar la duplicación y mantener la coherencia. Algunos ejemplos de intereses transversales en aplicaciones ASP.NET Core son la autenticación, las reglas de validación del modelo, el almacenamiento en caché de salida y el control de errores, aunque hay muchos otros. Los [filtros](#) de ASP.NET Core MVC permiten ejecutar código antes o después de ciertas fases de la canalización de procesamiento de la solicitud. Por ejemplo, se puede ejecutar un filtro antes y después del enlace del modelo, antes y después de una acción, o antes y después del resultado de una acción. También se puede usar un filtro de autorización para controlar el acceso al resto de la canalización. En la figura 7-2 se muestra cómo solicitar flujos de ejecución a través de filtros, si se han configurado.

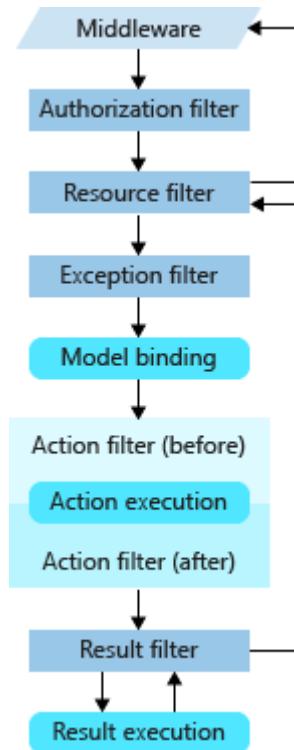


Figura 7-2. Ejecución de la solicitud a través de los filtros y la canalización de solicitud.

Normalmente, los filtros se implementan como atributos, para que se puedan aplicar a controladores o acciones, o incluso de forma global. Cuando se agregan de esta forma, los filtros especificados en el nivel de acción invalidan o se basan en los filtros especificados en el nivel de controlador, que a su vez invalidan los filtros globales. Por ejemplo, el atributo `[Route]` se puede usar para crear rutas entre controladores y acciones. Del mismo modo, la autorización se puede configurar en el nivel de controlador y, después, reemplazarse por acciones individuales, como se muestra en el ejemplo siguiente:

C#

```

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

```

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

Los filtros se pueden usar para eliminar la duplicación en forma de directivas de control de errores comunes para las API. Por ejemplo, una directiva de API típica es devolver una respuesta `NotFound` a las solicitudes que hacen referencia a claves que no existen y una respuesta `BadRequest` si se produce un error en la validación del modelo. En el ejemplo siguiente se muestran estas dos directivas en funcionamiento:

C#

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

No permita que los métodos de acción se llenen con código condicional similar a este. En su lugar, extraiga las directivas en filtros que se puedan aplicar según sea necesario. En este ejemplo, la comprobación de validación del modelo, que debería producirse siempre que se envíe un comando a la API, se puede reemplazar por el atributo siguiente:

C#

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

Puede agregar el elemento `ValidateModelAttribute` al proyecto como una dependencia de NuGet ; para ello, debe incluir el paquete [Ardalis.ValidateModel](#). Para las API, puede usar el atributo `ApiController` para aplicar este comportamiento sin tener que usar un filtro `ValidateModel` independiente.

Del mismo modo, se puede usar un filtro para comprobar si existe un registro y devolver un error 404 antes de ejecutar la acción, lo que elimina la necesidad de realizar estas comprobaciones en la acción. Una vez que se han extraído las convenciones comunes y se ha organizado la solución para separar el código de infraestructura y la lógica de negocio de la interfaz de usuario, los métodos de acción de MVC deben ser sumamente ligeros:

C#

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

Puede obtener más detalles sobre la implementación de filtros y descargar un ejemplo funcional en el artículo de MSDN Magazine [ASP.NET Core: filtros reales de ASP.NET Core MVC](#).

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

## Referencias: estructuración de aplicaciones

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

# Seguridad

La protección de las aplicaciones web es un tema amplio, con muchas consideraciones. En su nivel más básico, la seguridad implica asegurarse de saber de quién procede una solicitud determinada y, después, asegurarse de que la solicitud solo tiene acceso a los recursos que debe. La autenticación es el proceso de comparación de las credenciales proporcionadas con una solicitud con las de un almacén de datos de confianza, para ver si la solicitud se debe tratar como procedente de una entidad conocida. La autorización es el proceso de restringir el acceso a determinados recursos en función de la identidad del usuario. Un tercer interés de seguridad es proteger las solicitudes contra el espionaje por parte de terceros, para lo que al menos se debe [asegurar de que la aplicación usa SSL](#).

## Identidad

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

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

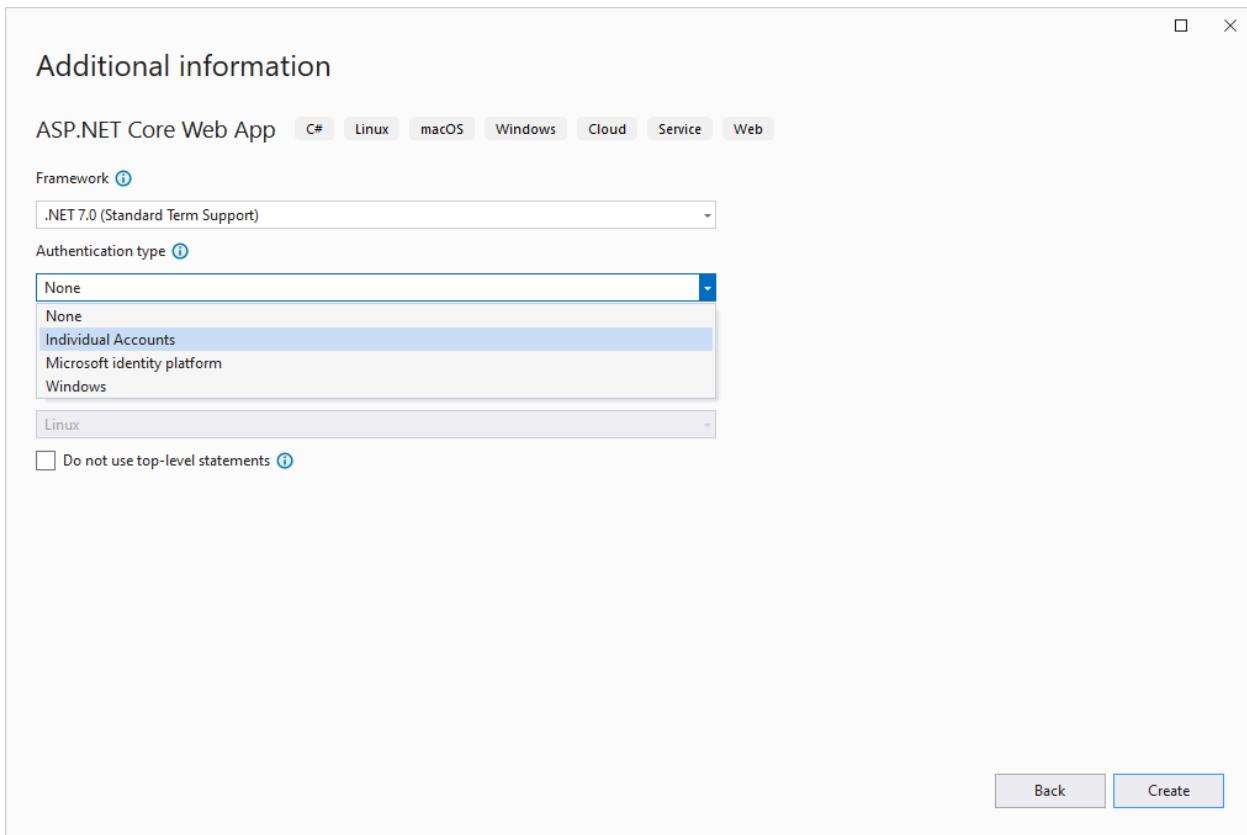


Figura 7-3. Figura 7-3 Selección de Cuentas de usuario individuales para preconfigurar Identity.

La compatibilidad con identidades se configura en *Program.cs* o *Startup*, e incluye la configuración de servicios, así como middleware.

## Configuración de la identidad en *Program.cs*

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

```
C#  
  
var builder = WebApplication.CreateBuilder(args);  
  
// Add services to the container.  
var connectionString =  
builder.Configuration.GetConnectionString("DefaultConnection");  
builder.Services.AddDbContext<ApplicationContext>(options =>  
    options.UseSqlServer(connectionString));  
builder.Services.AddDatabaseDeveloperPageExceptionFilter();  
  
builder.Services.AddDefaultIdentity<IdentityUser>(options =>  
    options.SignIn.RequireConfirmedAccount = true)  
    .AddEntityFrameworkStores<ApplicationContext>();
```

```
builder.Services.AddRazorPages();

var app = builder.Build();

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

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

app.UseRouting();

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

app.MapRazorPages();

app.Run();
```

## Configuración de la identidad al iniciar la aplicación

C#

```
// Add framework services.
builder.Services.AddDbContext<ApplicationContext>(options =>

options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection"))
);
builder.Services.AddIdentity<ApplicationUser, IdentityRole>()
    .AddEntityFrameworkStores<ApplicationContext>()
    .AddDefaultTokenProviders();
builder.Services.AddMvc();

var app = builder.Build();

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

```
    app.UseHsts();  
}  
  
app.UseHttpsRedirection();  
app.UseStaticFiles();  
  
app.UseRouting();  
  
app.UseAuthentication();  
app.UseAuthorization();  
  
app.MapRazorPages();
```

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

Puede obtener más información sobre la [configuración de la autenticación en dos fases](#) y la [habilitación de proveedores de inicio de sesión externos](#) en la documentación oficial de ASP.NET Core.

## Authentication

La autenticación es el proceso de determinar quién tiene acceso al sistema. Si usa ASP.NET Core Identity y los métodos de configuración que hemos visto en la sección anterior, se configurarán automáticamente algunos valores predeterminados de autenticación en la aplicación. Pero también puede configurar estos valores predeterminados de forma manual o reemplazar los valores definidos por `AddIdentity`. Si usa Identity, este sistema configura la autenticación basada en cookies como el *esquema* predeterminado.

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

- Autenticar: use la información que proporciona el cliente a fin de crearle una identidad para que la use dentro de la aplicación.
- Desafiar: esta acción se usa para exigir que el cliente se identifique.
- Prohibir: informe al cliente de que tiene prohibido realizar una acción.
- Iniciar sesión: conserve el cliente existente de alguna manera.
- Cerrar sesión: quite el cliente de la persistencia.

Hay una serie de técnicas comunes para llevar a cabo la autenticación en aplicaciones web. Se conocen como "esquemas". Un esquema determinado definirá las acciones de algunas o de todas las opciones anteriores. Algunos esquemas solo admiten un subconjunto de acciones y pueden requerir un esquema independiente para realizar aquellas que no admiten. Por ejemplo, el esquema OpenId Connect (OIDC) no admite el inicio o el cierre de sesión, sino que se configura normalmente para usar la autenticación de cookies para esta persistencia.

En la aplicación de ASP.NET Core, puede configurar un esquema `DefaultAuthenticateScheme`, así como esquemas específicos opcionales para cada una de las acciones descritas anteriormente. Por ejemplo, `DefaultChallengeScheme`, `DefaultForbidScheme`, etc. Al llamar a `AddIdentity<TUser,TRole>`, se configuran varios aspectos de la aplicación y se agregan muchos servicios necesarios. También se incluye esta llamada para configurar el esquema de autenticación:

C#

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

De forma predeterminada, estos esquemas usan cookies para la persistencia y el redireccionamiento a páginas de inicio de sesión para la autenticación. Estos esquemas son adecuados para las aplicaciones web que interactúan con los usuarios a través de exploradores web, pero no se recomiendan para las API. En su lugar, las API suelen usar otra forma de autenticación, como los tokens de portador JWT.

Las API web se consumen mediante código, como `HttpClient` en aplicaciones .NET y tipos equivalentes en otros marcos de trabajo. Estos clientes esperan una respuesta que puedan usar de una llamada API, o bien un código de estado que indique qué problema se ha producido, si hubiera uno. Estos clientes no interactúan a través de un explorador y tampoco interactúan con ningún código HTML que pueda devolver una API, ni lo representan. Por lo tanto, no es adecuado que los puntos de conexión de API redirijan a sus clientes a páginas de inicio de sesión si estos no se han autenticado. Hay otro esquema más adecuado.

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

C#

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

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

## Autenticación en aplicaciones Blazor

Las aplicaciones Blazor Server pueden aprovechar las mismas características de autenticación que cualquier otra aplicación ASP.NET Core. Sin embargo, las aplicaciones BlazorWebAssembly no pueden usar los proveedores de identidad de autenticación integrados, ya que se ejecutan en el explorador. Las aplicaciones BlazorWebAssembly pueden almacenar el estado de autenticación de los usuarios localmente y acceder a las notificaciones para determinar qué acciones deberían poder llevar a cabo los usuarios. Todas las comprobaciones de autenticación y autorización deben realizarse en el servidor, independientemente de la lógica implementada dentro de la aplicación BlazorWebAssembly, ya que los usuarios pueden omitir fácilmente la aplicación e interactuar directamente con las API.

## Referencias: autenticación

- Acciones de autenticación y valores predeterminados  
[https://stackoverflow.com/a/52493428 ↗](https://stackoverflow.com/a/52493428)
- Autenticación y autorización para aplicaciones de página única  
[https://learn.microsoft.com/aspnet/core/security/authentication/identity-api-...](https://learn.microsoft.com/aspnet/core/security/authentication/identity-api-)

## authorization

- Autenticación y autorización de Blazor de ASP.NET Core  
<https://learn.microsoft.com/aspnet/core/blazor/security/>
- Seguridad: Autenticación y autorización en ASP.NET Web Forms y Blazor  
<https://learn.microsoft.com/dotnet/architecture/blazor-for-web-forms-developers/security-authentication-authorization>

## Autorización

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

C#

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

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

La especificación de determinados conjuntos de roles como cadenas en muchos controladores y acciones diferentes puede dar lugar a repeticiones no deseadas. Como mínimo, defina las constantes para estos literales de cadena y use las constantes donde necesite especificar la cadena. También se pueden configurar directivas de autorización, que encapsulan las reglas de autorización y, después, especificar la directiva en lugar de roles individuales al aplicar el atributo `[Authorize]`:

C#

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

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

## Notificaciones

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

C#

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

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

## Protección de las API web

La mayoría de las API web deben implementar un sistema de autenticación basado en tokens. La autenticación por tokens carece de estado y está diseñada para que sea escalable. En un sistema de autenticación basado en tokens, el cliente debe autenticarse primero con el proveedor de autenticación. Si se realiza correctamente, se emite un token para el cliente, que simplemente es una cadena de caracteres criptográficamente significativa. El formato más común para los tokens es JSON Web Token o JWT (que a menudo se pronuncia "jot"). Después, cuando el cliente tiene que emitir una solicitud a una API, agrega este token como un encabezado a la solicitud. Luego, el servidor valida el token que se encuentra en el encabezado de solicitud antes de completar la solicitud. En la figura 7-4 se muestra este proceso.

# Token-Based Authentication

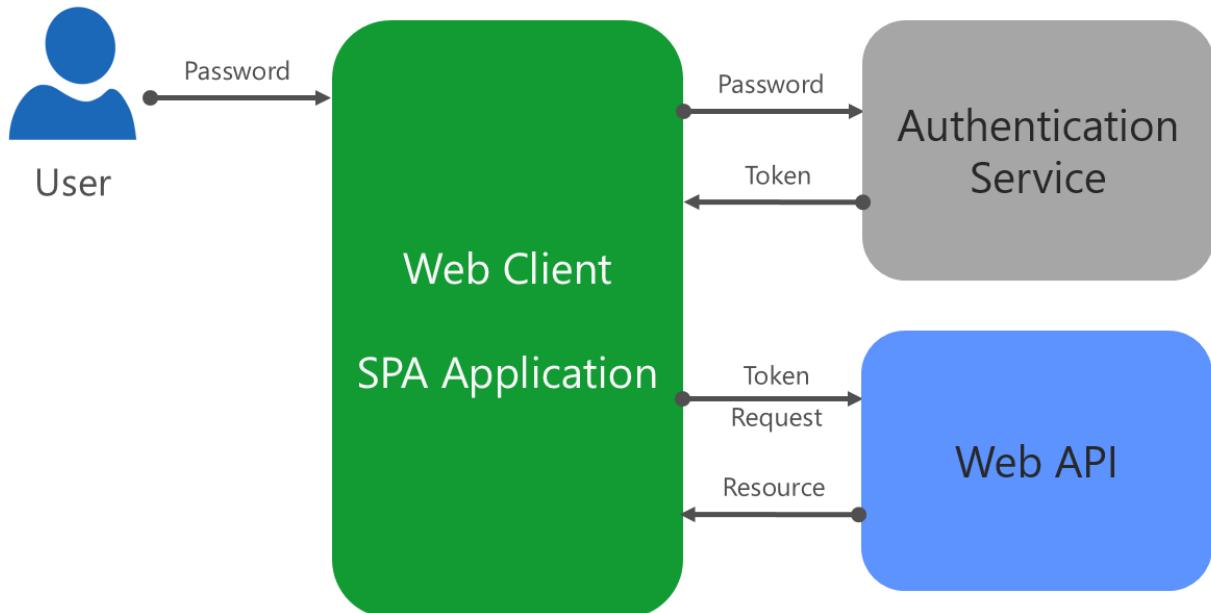


Figura 7-4. Autenticación basada en tokens para las API web.

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

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

Al usar tokens JWT con aplicaciones de página única o BlazorWebAssembly, debe almacenar el token en algún lugar del cliente y, después, agregarlo a cada llamada API. Esta actividad se suele hacer como encabezado, tal y como se muestra en el código siguiente:

C#

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

Después de llamar al método anterior, las solicitudes realizadas con `_httpClient` tendrán el token insertado en los encabezados de solicitud, lo que permite que la API

del lado servidor autentique y autorice la solicitud.

## Seguridad personalizada

### ⊗ Precaución

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

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

## Referencias: seguridad

- **Introducción a la seguridad de ASP.NET Core**  
<https://learn.microsoft.com/aspnet/core/security/>
- **Exigir SSL en una aplicación ASP.NET básica**  
<https://learn.microsoft.com/aspnet/core/security/enforcing-ssl>
- **Introducción a Identity**  
<https://learn.microsoft.com/aspnet/core/security/authentication/identity>
- **Introducción a la autorización**  
<https://learn.microsoft.com/aspnet/core/security/authorization/introduction>
- **Autenticación y autorización para aplicaciones de API en Azure App Service**  
<https://learn.microsoft.com/azure/app-service-api/app-service-api-authentication>
- **Servidor de identidades**  
[https://github.com/IdentityServer ↗](https://github.com/IdentityServer)

## Comunicación de cliente

Además de servir páginas y responder a las solicitudes de datos a través de las API web, las aplicaciones ASP.NET Core se pueden comunicar directamente con los clientes conectados. En esta comunicación de salida se puede usar una amplia variedad de tecnologías de transporte, siendo WebSockets la más común. ASP.NET Core SignalR es una biblioteca que simplifica la funcionalidad de agregar comunicación de servidor a cliente en tiempo real en las aplicaciones. SignalR admite diversas tecnologías de

transporte, incluyendo WebSockets, y abstrae muchos de los detalles de implementación del desarrollador.

La comunicación de cliente en tiempo real, con independencia de que se use WebSockets directamente u otras técnicas, es útil en una variedad de escenarios de aplicación. Estos son algunos ejemplos:

- Aplicaciones de salón de chat en vivo
- Aplicaciones de supervisión
- Actualizaciones de progreso de trabajo
- Notificaciones
- Aplicaciones de formularios interactivos

Al compilar la comunicación de cliente en las aplicaciones, normalmente hay dos componentes:

- El administrador de conexiones del lado servidor (SignalR Hub, WebSocketManager WebSocketHandler)
- La biblioteca del lado cliente

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

C#

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

    public static async Task StartConnectionAsync()
    {
```

```
        _connection = new Connection();
        await _connection.StartConnectionAsync("ws://localhost:65110/chat");
    }

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

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

## Referencias: comunicación de cliente

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

## ¿Se debe aplicar el diseño controlado por dominios?

El Diseño controlado por dominios (DDD) es un enfoque ágil para la creación de software que resalta centrarse en el *dominio de negocio*. Coloca un gran énfasis en la comunicación e interacción con los expertos de dominio de negocio, que pueden identificarse con los desarrolladores sobre cómo funciona el sistema del mundo real. Por ejemplo, si está creando un sistema que controla cotizaciones de bolsa, es posible que el experto de dominio sea un broker de bolsa con experiencia. DDD está diseñado para resolver problemas empresariales grandes y complejos, y no suele ser adecuado para aplicaciones más pequeñas y sencillas, dado que la inversión necesaria para comprender y modelar el dominio no es rentable.

Al compilar software con un enfoque de DDD, el equipo (incluidas las partes interesadas sin experiencia técnica y los colaboradores) deberían desarrollar un *lenguaje ubicuo* para el espacio del problema. Es decir, se debe usar la misma terminología para el concepto del mundo real que se está modelando, el equivalente de software y las estructuras que podrían existir para conservar el concepto (por ejemplo, las tablas de base de datos). Por tanto, los conceptos descritos en el lenguaje ubicuo deberían ser la base del *modelo de dominio*.

El modelo de dominio se compone de objetos que interactúan entre sí para representar el comportamiento del sistema. Estos objetos se dividen en las categorías siguientes:

- [Entidades](#), que representan objetos con un subproceso de identidad. Normalmente las entidades se almacenan en la persistencia con una clave por la que después se pueden recuperar.
- [Agregados](#), que representan grupos de objetos que se deben conservar como una unidad.
- [Objetos de valor](#), que representan conceptos que se pueden comparar en función de la suma de sus valores de propiedad. Por ejemplo, DateRange consta de una fecha de inicio y una fecha de finalización.
- [Eventos de dominio](#), que representan lo que ocurre en el sistema que resulta de interés para otros elementos del sistema.

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

Además de estos tipos de modelo, DDD normalmente emplea diversos modelos:

- [Repositorio](#), para abstraer los detalles de persistencia.
- [Generador](#), para encapsular la creación de objetos complejos.
- [Servicios](#), para encapsular un comportamiento complejo o los detalles de implementación de la infraestructura.
- [Comando](#), para desacoplar la emisión de comandos y la ejecución del propio comando.
- [Especificación](#), para encapsular los detalles de la consulta.

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

## Casos en los que se debe aplicar DDD

DDD es ideal para aplicaciones de gran tamaño con complejidad de negocio significativa (no solo técnica). La aplicación debe requerir el conocimiento de expertos

de dominio. Debe haber un comportamiento importante en el propio modelo de dominio, que represente las reglas de negocio e interacciones más allá de simplemente almacenar y recuperar el estado actual de diferentes registros desde almacenes de datos.

## Casos en los que no se debe aplicar DDD

DDD implica invertir en modelado, arquitectura y comunicación, lo que puede que no esté garantizado para aplicaciones más pequeñas o aplicaciones que esencialmente son de tipo CRUD (crear/leer/actualizar/eliminar). Si se elige el enfoque de DDD para la aplicación, pero se descubre que el dominio tiene un modelo anémico sin comportamiento, es posible que haya que reconsiderar el enfoque. O la aplicación no necesita DDD o es posible que necesite asistencia para refactorizarla con el fin de encapsular la lógica de negocios en el modelo de dominio, en lugar de la interfaz de usuario o la base de datos.

Un enfoque híbrido consistiría en usar DDD solo para las áreas más complejas o transaccionales de la aplicación, pero no para partes CRUD más sencillas o de solo lectura. Por ejemplo, no se necesitan las restricciones de un agregado si se están consultando datos para mostrar un informe o visualizar datos para un panel. Es absolutamente aceptable tener un modelo de lectura más sencillo e independiente para estos requisitos.

## Referencias: diseño controlado por dominios

- **DDD in Plain English (StackOverflow Answer)** (DDD en términos sencillos [respuesta en StackOverflow])  
<https://stackoverflow.com/questions/1222392/can-someone-explain-domain-driven-design-ddd-in-plain-english-please/1222488#1222488> ↗

## Implementación

En el proceso de implementación de la aplicación ASP.NET Core hay algunos pasos implicados, independientemente de dónde se vaya a hospedar. El primer paso consiste en publicar la aplicación, lo que se puede hacer con el comando de CLI `dotnet publish`. Este paso compilará la aplicación y colocará todos los archivos necesarios para ejecutarla en una carpeta designada. Cuando se implementa desde Visual Studio, este paso se realiza de forma automática. La carpeta `publish` contiene archivos .exe y .dll de la aplicación y sus dependencias. Una aplicación independiente también incluirá una

versión del runtime de .NET. Las aplicaciones ASP.NET Core también incluyen archivos de configuración, activos de cliente estáticos y vistas MVC.

Las aplicaciones ASP.NET Core son aplicaciones de consola que se deben iniciar cuando se inicia el servidor y reiniciarse si la aplicación (o el servidor) se bloquea. Para automatizar este proceso se puede usar un administrador de procesos. Los administradores de procesos más comunes para ASP.NET Core son Nginx y Apache en Linux, e IIS y Servicio de Windows en Windows.

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

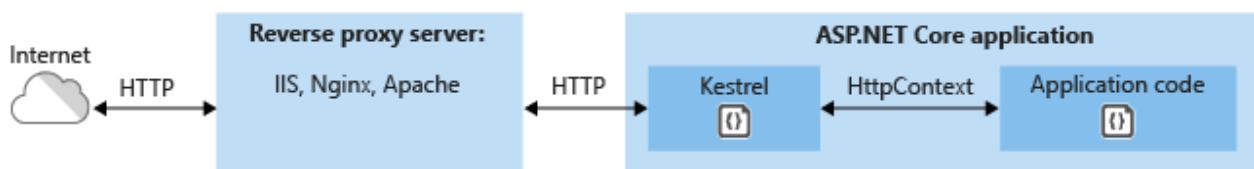


Figura 7-5. ASP.NET hospedado en Kestrel detrás de un servidor proxy inverso

Otro escenario en el que un proxy inverso puede ser útil es para proteger varias aplicaciones mediante SSL/HTTPS. En este caso, solo sería necesario configurar SSL en el proxy inverso. La comunicación entre el servidor proxy inverso y Kestrel se podría llevar a cabo a través de HTTPS, como se muestra en la figura 7-6.

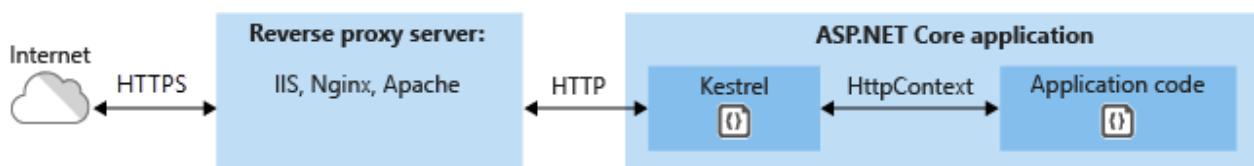


Figura 7-6. ASP.NET hospedado detrás de un servidor proxy inverso protegido mediante HTTPS

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

Si la aplicación se va a hospedar en Azure, se puede usar Microsoft Azure Application Gateway como un dispositivo virtual dedicado para proporcionar varios servicios.

Además de actuar como un proxy inverso para aplicaciones individuales, Application Gateway también puede ofrecer las características siguientes:

- Equilibrio de carga HTTP
- Descarga SSL (SSL solo para Internet)
- SSL de extremo a extremo
- Enrutamiento de varios sitios (hasta 20 sitios consolidados en una sola instancia de Application Gateway)
- Firewall de aplicación web
- Compatibilidad de WebSocket
- Diagnósticos avanzados

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

## Referencias: implementación

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

[Anterior](#)

[Siguiente](#)

# Trabajar con datos en aplicaciones ASP.NET Core

Artículo • 27/02/2023 • Tiempo de lectura: 31 minutos

## 💡 Sugerencia

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

[Descargar PDF](#)



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

Tim Berners-Lee

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

## Entity Framework Core (para bases de datos relacionales)

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

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

CLI de .NET

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

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

CLI de .NET

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

## DbContext

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

C#

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

    }

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

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

## Configuración de EF Core

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

C#

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

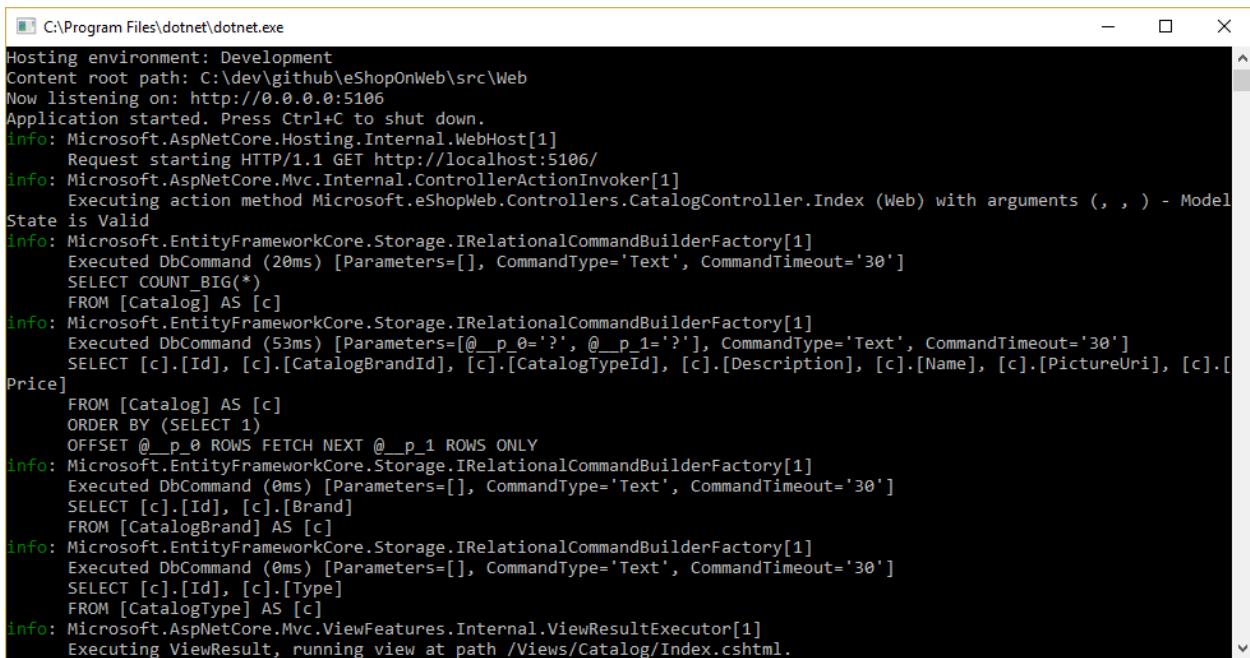
Para usar la base de datos en memoria:

C#

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

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

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



C:\Program Files\dotnet\dotnet.exe

Hosting environment: Development

Content root path: C:\dev\github\eShopOnWeb\src\Web

Now listening on: http://0.0.0.0:5106

Application started. Press Ctrl+C to shut down.

info: Microsoft.AspNetCore.Hosting.Internal.WebHost[1]

Request starting HTTP/1.1 GET http://localhost:5106/

info: Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker[1]

Executing action method Microsoft.eShopWeb.Controllers.CatalogController.Index (Web) with arguments (, , ) - ModelState is Valid

info: Microsoft.EntityFrameworkCore.Storage.IRelationalCommandBuilderFactory[1]

Executed DbCommand (20ms) [Parameters=[], CommandType='Text', CommandTimeout='30']

SELECT COUNT\_BIG(\*)

FROM [Catalog] AS [c]

info: Microsoft.EntityFrameworkCore.Storage.IRelationalCommandBuilderFactory[1]

Executed DbCommand (53ms) [Parameters=[@\_\_p\_0='?', @\_\_p\_1='?'], CommandType='Text', CommandTimeout='30']

SELECT [c].[Id], [c].[CatalogBrandId], [c].[CatalogTypeId], [c].[Description], [c].[Name], [c].[PictureUri], [c].[Price]

FROM [Catalog] AS [c]

ORDER BY (SELECT 1)

OFFSET @\_p\_0 ROWS FETCH NEXT @\_p\_1 ROWS ONLY

info: Microsoft.EntityFrameworkCore.Storage.IRelationalCommandBuilderFactory[1]

Executed DbCommand (0ms) [Parameters=[], CommandType='Text', CommandTimeout='30']

SELECT [c].[Id], [c].[Brand]

FROM [CatalogBrand] AS [c]

info: Microsoft.EntityFrameworkCore.Storage.IRelationalCommandBuilderFactory[1]

Executed DbCommand (0ms) [Parameters=[], CommandType='Text', CommandTimeout='30']

SELECT [c].[Id], [c].[Type]

FROM [CatalogType] AS [c]

info: Microsoft.AspNetCore.Mvc.ViewFeatures.Internal.ViewResultExecutor[1]

Executing ViewResult, running view at path /Views/Catalog/Index.cshtml.

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

## Recuperación y almacenamiento de los datos

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

```
C#
```

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

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

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

C#

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

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

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

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

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

## Recuperación de datos relacionados

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

C#

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

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

C#

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

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

C#

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

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

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

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

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

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

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

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

## Encapsulación de datos

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

C#

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

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

```

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

```

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

C#

```

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

    navigation.SetPropertyAccessMode(PropertyAccessMode.Field);
}

```

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

C#

```

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

```

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

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

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

## Conexiones resistentes

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

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

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

C#

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

## Estrategias de ejecución y transacciones explícitas mediante `BeginTransaction` y varios `DbContexts`

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

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

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

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

C#

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

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

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

## Referencias: Entity Framework Core

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

## ¿EF Core o micro-ORM?

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

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

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

```
C#  
  
// EF Core  
private readonly CatalogContext _context;  
public async Task<IEnumerable<CatalogType>> GetCatalogTypes()  
{  
    return await _context.CatalogTypes.ToListAsync();  
}  
  
// Dapper  
private readonly SqlConnection _conn;  
public async Task<IEnumerable<CatalogType>> GetCatalogTypesWithDapper()  
{  
    return await _conn.QueryAsync<CatalogType>("SELECT * FROM CatalogType");  
}
```

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

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

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

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

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

```
C#
```

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

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

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

## SQL o NoSQL

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

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

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

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

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

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

## Azure Cosmos DB

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

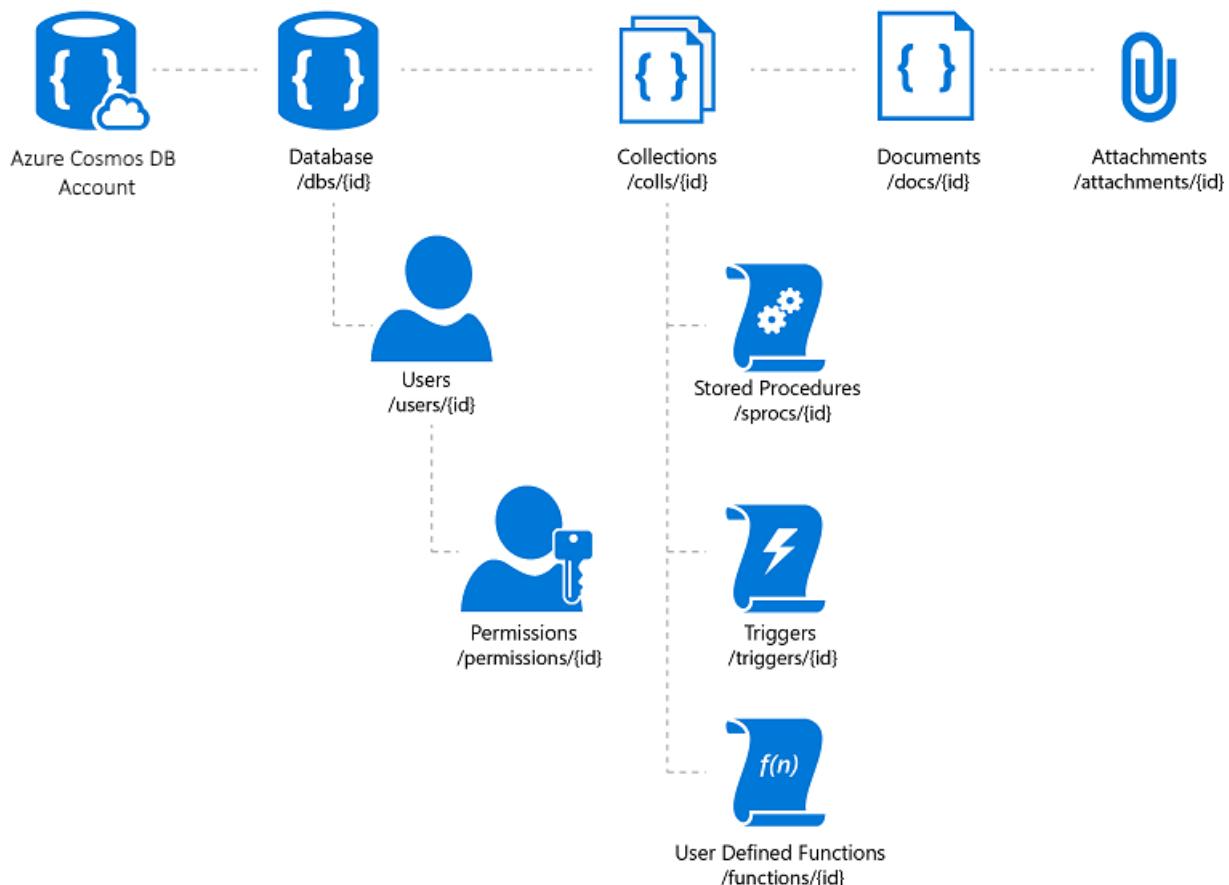


Figura 8-2. Organización de recursos de Azure Cosmos DB.

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

#### Referencias: Azure Cosmos DB

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

## Otras opciones de persistencia

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

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

#### Referencias: Azure Storage

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

## Almacenamiento en memoria caché

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

ASP.NET Core admite el almacenamiento de respuestas en caché, para almacenar en caché páginas completas y el almacenamiento de datos en caché, que admite un comportamiento de almacenamiento en caché más granular.

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

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

## Almacenamiento en caché de respuestas de ASP.NET Core

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

C#

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

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

HTML

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

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

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

C#

```
builder.Services.AddResponseCaching();

// other code omitted, including building the app

app.UseResponseCaching();
```

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

## Almacenamiento de datos en caché

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

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

```
C#
```

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

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

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

```
C#
```

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

```

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

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

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

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

```

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

C#

```

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

```

Después de agregar este código, las llamadas de base de datos para recuperar los cambios de los datos del catálogo solo se realizarán una vez por minuto, en lugar de en cada solicitud. Según el tráfico al sitio, esto puede tener un impacto significativo en el número de consultas realizadas a la base de datos y el tiempo medio de carga de la página principal que depende actualmente de las tres consultas expuestas por este servicio.

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

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

```
C#
```

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

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

```
C#
```

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

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

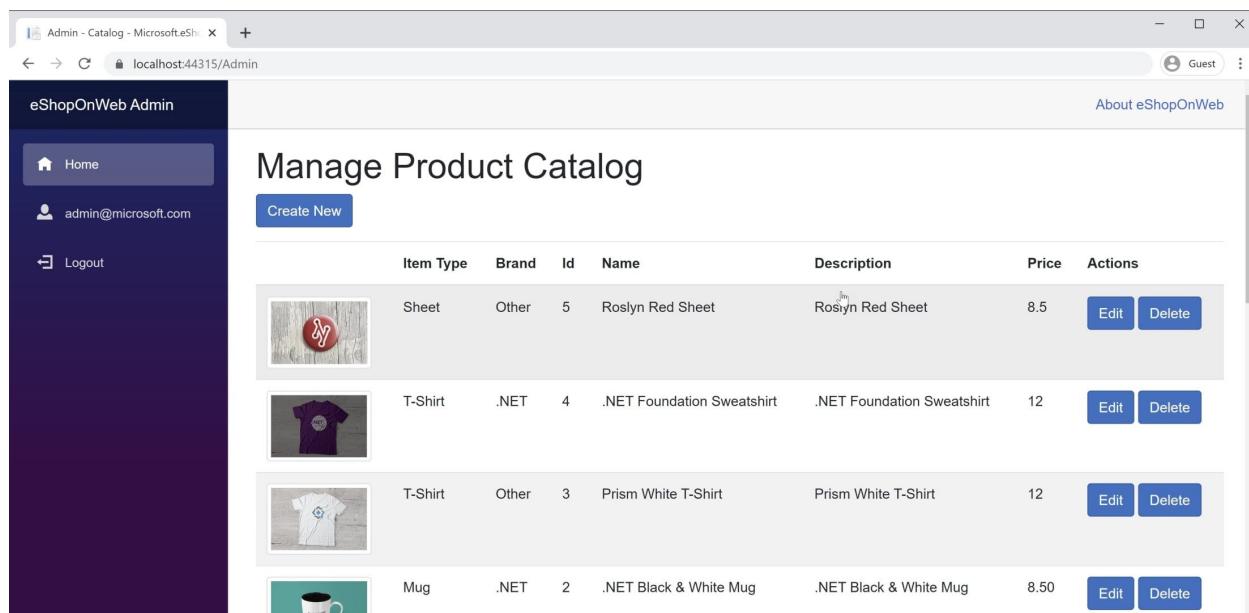
El almacenamiento en caché puede mejorar considerablemente el rendimiento de las páginas web que solicitan repetidamente los mismos valores de la base de datos. Asegúrese de medir el acceso de datos y el rendimiento de la página antes de aplicar el almacenamiento en caché y aplíquelo solo donde vea una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

## Obtención de datos para aplicaciones de BlazorWebAssembly

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

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

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



The screenshot shows a Microsoft Edge browser window titled "Admin - Catalog - Microsoft.eShopOnWeb". The URL is "localhost:44315/Admin". The page is titled "Manage Product Catalog" and features a sidebar with "Home", "admin@microsoft.com", and "Logout" buttons. The main content area displays a table of products:

	Item Type	Brand	Id	Name	Description	Price	Actions
	Sheet	Other	5	Roslyn Red Sheet	Roslyn Red Sheet	8.5	<button>Edit</button> <button>Delete</button>
	T-Shirt	.NET	4	.NET Foundation Sweatshirt	.NET Foundation Sweatshirt	12	<button>Edit</button> <button>Delete</button>
	T-Shirt	Other	3	Prism White T-Shirt	Prism White T-Shirt	12	<button>Edit</button> <button>Delete</button>
	Mug	.NET	2	.NET Black & White Mug	.NET Black & White Mug	8.50	<button>Edit</button> <button>Delete</button>

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

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

```
C#
```

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

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

```
C#
```

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

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

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

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

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

```
C#
```

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

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

## Referencias: datos de Blazor

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

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

# Prueba de aplicaciones ASP.NET Core MVC

Artículo • 28/11/2022 • Tiempo de lectura: 18 minutos

## 💡 Sugerencia

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

[Descargar PDF](#)



*"Si no le gusta realizar pruebas unitarias de su producto, lo más probable es que a los clientes tampoco les guste probarlo."\_- Anónimo-*

En el software de cualquier complejidad se pueden producir errores inesperados en respuesta a los cambios. Por tanto, es necesario realizar pruebas después de realizar cambios en todas las aplicaciones menos en las más triviales (o las menos críticas). Las pruebas manuales son la forma más lenta, menos confiable y más costosa de probar software. Desafortunadamente, si las aplicaciones no están diseñadas para que se puedan probar, puede ser el único medio de prueba disponible. Las aplicaciones escritas para seguir los principios de arquitecturas que se indican en el [capítulo 4](#) deben poder someterse en gran medida a pruebas unitarias. Las aplicaciones ASP.NET Core deben admitir pruebas de integración y funcionales automatizadas.

## Tipos de pruebas automatizadas

Hay muchos tipos de pruebas automatizadas para las aplicaciones de software. La prueba más sencilla y de nivel más bajo es la prueba unitaria. En un nivel ligeramente superior se encuentran las pruebas de integración y las pruebas funcionales. Otros tipos de pruebas, como las de interfaz de usuario, de carga, de esfuerzo y de humo, quedan fuera del ámbito de este documento.

## Pruebas unitarias

Una prueba unitaria prueba un único elemento de la lógica de la aplicación. Se puede describir aún más enumerando algunas de las cosas que no hace. Una prueba unitaria no prueba el funcionamiento del código con dependencias o infraestructura; eso lo comprueban las pruebas de integración. Una prueba unitaria no prueba el marco para el que se escribe el código; se debe asumir que funciona o, si se detecta que no lo hace, registrar un error y codificar una solución alternativa. Una prueba unitaria se ejecuta completamente en memoria y en proceso. No se comunica con el sistema de archivos, la red o una base de datos. Las pruebas unitarias solo deben probar el código.

Las pruebas unitarias, puesto que solo prueban una unidad del código, sin dependencias externas, se deben ejecutar muy rápidamente. Por tanto, debe ser capaz de ejecutar conjuntos de cientos de pruebas unitarias en unos segundos. Ejecute las pruebas con frecuencia, idealmente antes de cada inserción en un repositorio de control de código fuente compartido y, por supuesto, con cada compilación automatizada en el servidor de compilación.

## Pruebas de integración

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

Las pruebas de integración a menudo tendrán procedimientos más complejos de instalación y desmontaje que las pruebas unitarias. Por ejemplo, una prueba de integración dirigida a una base de datos real necesitará un modo de devolver la base de

datos a un estado conocido antes de cada serie de pruebas. Cuando se agregan nuevas pruebas y el esquema de base de datos de producción evoluciona, estos scripts de prueba tienden a aumentar de tamaño y complejidad. En muchos sistemas de gran tamaño, no resulta práctico ejecutar conjuntos completos de pruebas de integración en las estaciones de trabajo de desarrollador antes de insertar en el repositorio los cambios en el control de código fuente compartido. En estos casos, las pruebas de integración se pueden ejecutar en un servidor de compilación.

## Pruebas funcionales

Las pruebas de integración se escriben desde la perspectiva del desarrollador, para comprobar que algunos componentes del sistema funcionen correctamente entre sí. Las pruebas funcionales se escriben desde la perspectiva del usuario y comprueban la exactitud del sistema en función de sus requisitos. En el fragmento siguiente se ofrece una analogía útil para saber cómo pensar en las pruebas funcionales, en comparación con las pruebas unitarias:

"En muchas ocasiones, el desarrollo de un sistema se asemeja a la construcción de una casa. Aunque esta analogía no es del todo correcta, podemos ampliarla para explicar la diferencia entre las pruebas unitarias y las funcionales. Las pruebas unitarias son similares a un inspector municipal que visita la obra de construcción de una casa. Se centra en los distintos sistemas internos de la casa, los cimientos, los muros, la instalación eléctrica, la fontanería y así sucesivamente. Se asegura (prueba) de que las partes de la casa funcionarán correctamente y de manera segura, es decir, cumplen el código de construcción. En este escenario, las pruebas funcionales son análogas al propietario que visita esta misma obra. Da por supuesto que los sistemas internos se comportarán de forma adecuada, que el inspector municipal está realizando su trabajo. El propietario se centra en cómo será vivir en esta casa. Le preocupa el aspecto de la casa, si las distintas habitaciones tienen un tamaño cómodo, si la casa se ajusta a las necesidades de la familia, si las ventanas están en una zona adecuada para captar el sol por la mañana. El propietario está realizando pruebas funcionales de la casa. Tiene la perspectiva del usuario. El inspector municipal está realizando pruebas unitarias en la casa. Tiene la perspectiva del constructor".

Fuente: [Unit Testing versus Functional Tests ↗](#) (Diferencias entre pruebas unitarias y pruebas funcionales)

Me gusta decir "Como desarrolladores, tenemos dos formas de fracasar: compilando como no debemos o compilando lo que no debemos". Las pruebas unitarias garantizan

que está compilando como debe; las pruebas funcionales garantizan que está compilando lo que debe.

Como las pruebas funcionales operan en el nivel de sistema, pueden requerir cierto grado de automatización de la interfaz de usuario. Al igual que las pruebas de integración, también suelen funcionar con algún tipo de infraestructura de pruebas. Esta actividad hace que sean más lentas y frágiles que las pruebas unitarias y de integración. Solo se deben tener las pruebas funcionales necesarias para estar seguro de que el sistema se comporta tal y como esperan los usuarios.

## Pirámide de pruebas

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

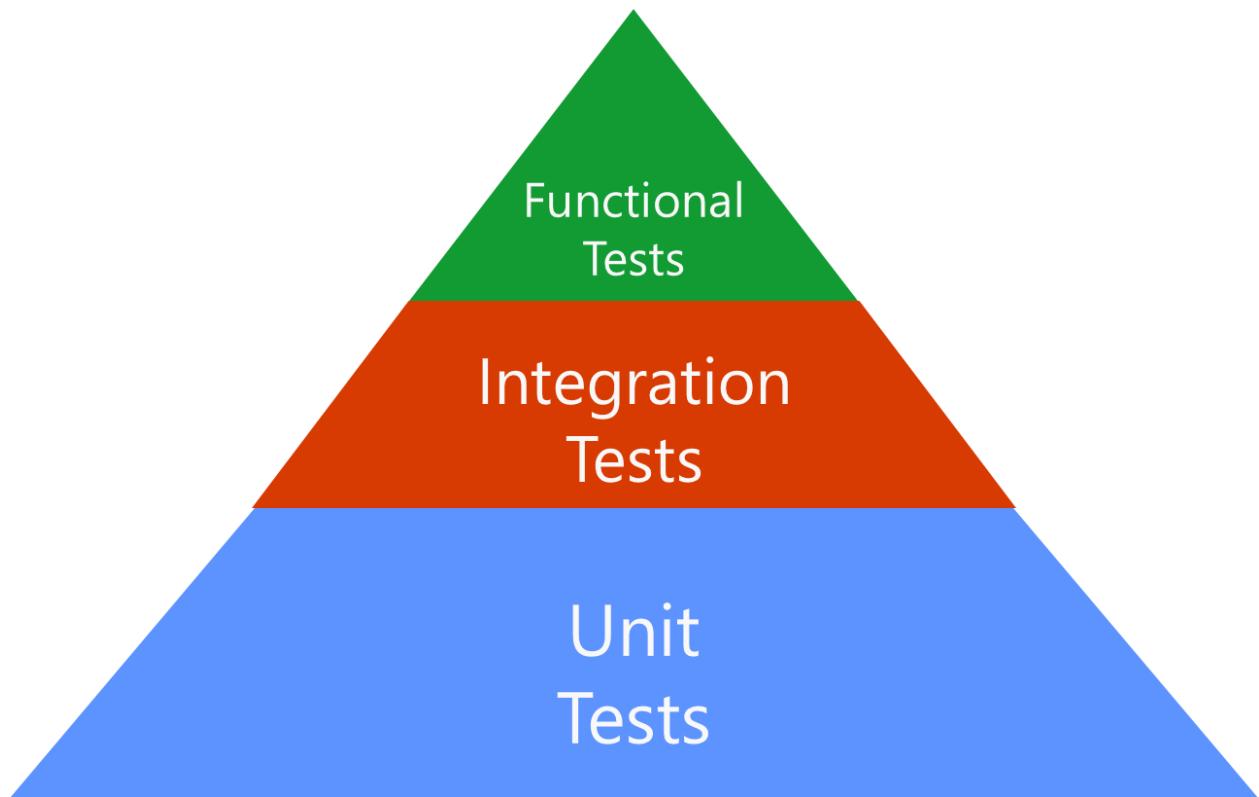


Figura 9-1. Pirámide de pruebas

Los diferentes niveles de la pirámide y sus tamaños relativos representan distintos tipos de pruebas y cuántas se deben escribir para la aplicación. Como se puede ver, la recomendación es tener una base grande de pruebas unitarias, respaldada por un nivel más pequeño de pruebas de integración, con un nivel incluso más pequeño de pruebas funcionales. Idealmente, cada nivel solo debería incluir las pruebas que no se puedan realizar de forma adecuada en un nivel inferior. Tenga presente la pirámide de pruebas al tratar de decidir qué tipo de prueba necesita para un escenario determinado.

## Qué se va a probar

Un problema común para los desarrolladores sin experiencia con la escritura de pruebas automatizadas es determinar lo que se debe probar. Un buen punto de partida es probar la lógica condicional. Siempre que haya un método con un comportamiento que cambia en función de una instrucción condicional (if-else, switch, etc.), debería poder dar al menos un par de pruebas que confirmen el comportamiento correcto para determinadas condiciones. Si el código tiene condiciones de error, es conveniente escribir al menos una prueba para la "ruta feliz" a través del código (sin errores) y al menos una prueba para la "ruta triste" (con errores o resultados pocos frecuentes) para confirmar que la aplicación se comporta según lo previsto ante los errores. Por último, intente centrarse en probar cosas en las que se pueda producir un error, en lugar de centrarse en métricas como la cobertura de código. Por lo general, es mejor tener más cobertura de código que menos. Pero escribir algunas pruebas más de un método complejo y esencial para la empresa suele ser un mejor uso del tiempo que escribir pruebas para propiedades automáticas solo para mejorar la métrica de cobertura del código de prueba.

## Organización de los proyectos de prueba

Los proyectos de prueba se pueden organizar de la manera que mejor funcione. Es una buena idea separar las pruebas por tipo (prueba unitaria, prueba de integración) y por lo que van a probar (por proyecto, por espacio de nombres). Que esta separación conste de carpetas dentro de un único proyecto de prueba o de varios es una decisión de diseño. Lo más sencillo es un proyecto, pero para proyectos grandes con muchas pruebas, o bien para ejecutar más fácilmente otros conjuntos de pruebas, es posible que le interese tener varios proyectos de prueba distintos. Muchos equipos organizan los proyectos de prueba en función del proyecto que se está probando, que para las aplicaciones con más de unos pocos proyectos puede provocar un gran número de proyectos de prueba, en especial si se siguen dividiendo según el tipo de pruebas de cada proyecto. Un enfoque de compromiso consiste en disponer de un proyecto por tipo de prueba, por aplicación, con carpetas dentro de los proyectos de prueba para indicar el proyecto (y la clase) que se van a probar.

Un enfoque común consiste en organizar los proyectos de la aplicación en una carpeta "src" y los proyectos de prueba en una carpeta "tests" paralela. Si esta organización le parece útil, puede crear carpetas de solución coincidentes en Visual Studio.

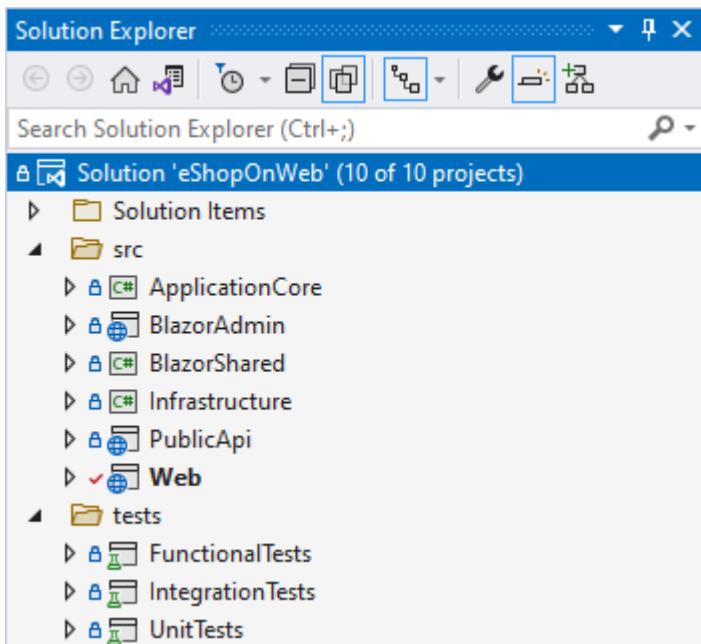


Figura 9-2. Organización de las pruebas en la solución

Puede usar el marco de pruebas que prefiera. El marco de trabajo xUnit funciona bien y es en el que se escriben todas las pruebas de ASP.NET Core y EF Core. Puede agregar un proyecto de prueba de xUnit en Visual Studio con la plantilla que se muestra en la figura 9-3 o desde la CLI con `dotnet new xunit`.

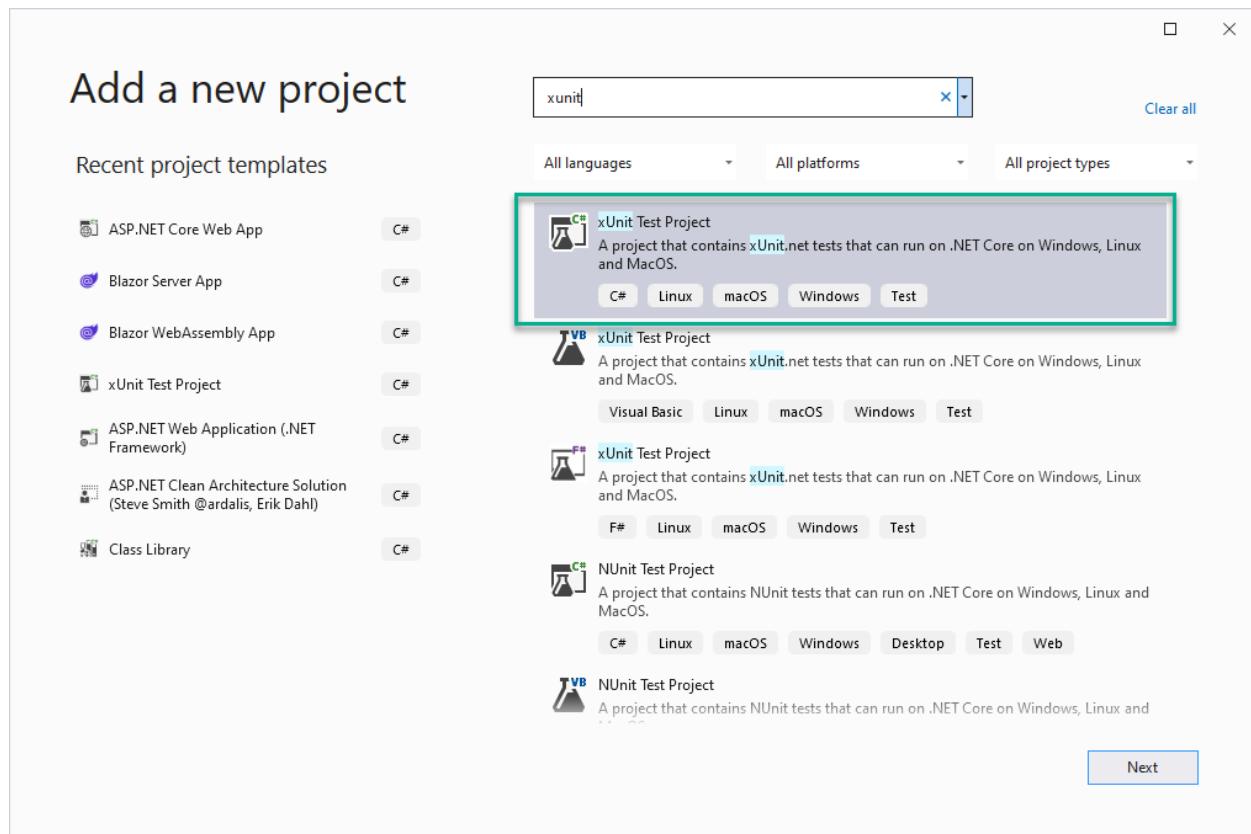


Figura 9-3. Agregar un proyecto de prueba de xUnit en Visual Studio

## Nombres de pruebas

Asigne nombres a las pruebas de forma coherente, con un nombre que indique lo que hace cada prueba. Un enfoque con el que he tenido buenos resultados consiste en asignar nombres a las clases de prueba en función de la clase y el método que estén probando. Este enfoque da como resultado muchas clases de prueba pequeñas, pero deja muy claro la responsabilidad de cada una. Con el nombre de la clase de prueba configurado para identificar la clase y el método que se van a probar, se puede usar el nombre del método de prueba para especificar el comportamiento que se está probando. Este nombre debe incluir el comportamiento esperado, y las entradas o suposiciones que deban producir este comportamiento. Algunos nombres de prueba de ejemplo:

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

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

- `CatalogControllerGetImage Should . Call ImageServiceWithId`
- `CatalogControllerGetImage Should . Log WarningGivenImageMissingException`

Para algunos equipos el segundo método de nomenclatura es más claro, aunque un poco más detallado. En cualquier caso, intente usar una convención de nomenclatura que proporcione información del comportamiento de la prueba, para que cuando se produzca un error en una o varias pruebas, sea obvio a partir de los nombres en qué casos se ha producido el error. Evite asignar nombres vagos a las pruebas, como `ControllerTests.Test1`, dado que estos nombres no ofrecen ningún valor cuando se ven en los resultados de las pruebas.

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

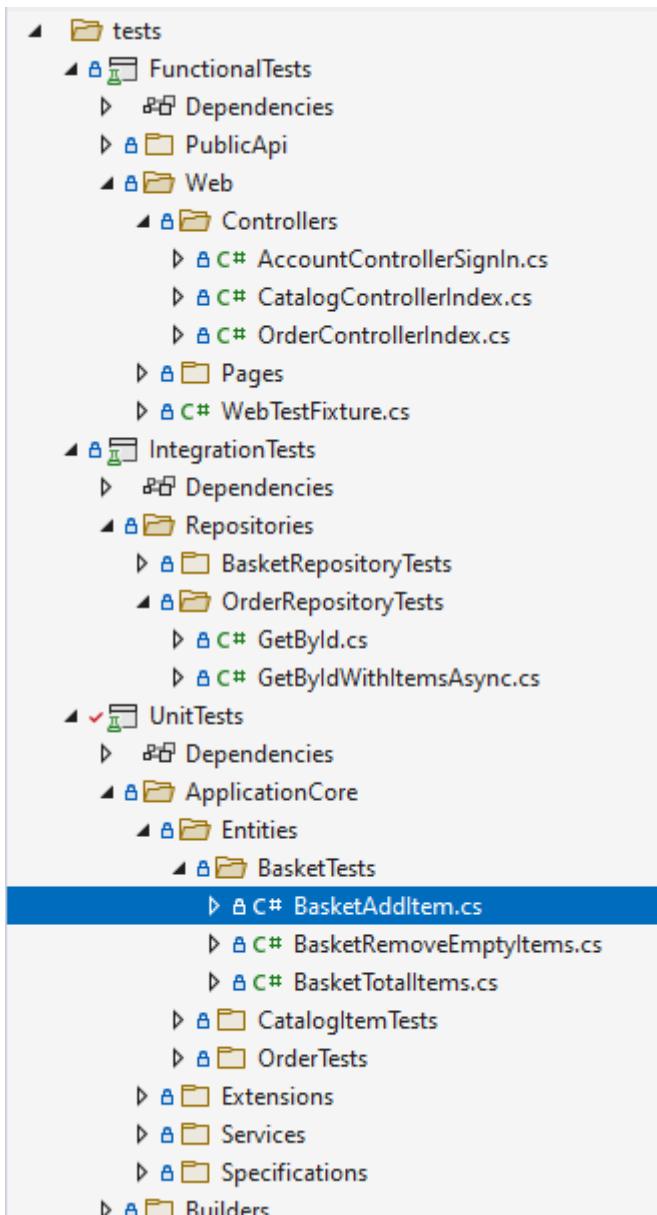


Figura 9-4. Organización de las clases de prueba por carpeta en función de la clase que se está probando.

Si una clase de aplicación determinada tiene muchos métodos para probar (y, por tanto, muchas clases de prueba), tiene sentido colocar estas clases en una carpeta correspondiente a la clase de aplicación. Esta organización es similar a cómo se podrían organizar los archivos en carpetas en otra parte. Si tiene más de tres o cuatro archivos relacionados en una carpeta que contiene otros muchos archivos, suele resultar útil moverlos a su propia subcarpeta.

## Pruebas unitarias de aplicaciones ASP.NET Core

En una aplicación ASP.NET Core bien diseñada, la mayor parte de la complejidad y la lógica de negocios se encapsulará en entidades de negocio y una variedad de servicios. La propia aplicación ASP.NET Core MVC, con sus controladores, filtros, modelos de vista y vistas, no debería requerir muchas pruebas unitarias. Gran parte de la funcionalidad de

una acción determinada se encuentra fuera del propio método de acción. Con una prueba unitaria no se puede comprobar de forma eficaz si el enrutamiento o el control de errores globales funcionan correctamente. Del mismo modo, los filtros, incluidos los de autenticación y validación del modelo y los de autorización, no se pueden someter a pruebas unitarias con una prueba que tiene como objetivo el método de acción de un controlador. Sin estas fuentes de comportamiento, la mayoría de los métodos de acción deberían ser pequeños, y delegar la mayor parte de su trabajo a servicios que se puedan probar de forma independiente al controlador que los usa.

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

C#

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

La realización de pruebas unitarias en este método se complica debido a su dependencia directa de `System.IO.File`, que usa para leer el sistema de archivos. Puede probar este comportamiento para asegurarse de que funciona de la forma esperada, pero si lo hace con archivos reales será una prueba de integración. Cabe destacar que no se puede realizar la prueba unitaria de la ruta de este método: verá cómo hacerla, con una prueba funcional, en breve.

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

C#

```
[HttpGet("[controller]/pic/{id}")]
public IActionResult GetImage(int id)
{
```

```
byte[] imageBytes;
try
{
    imageBytes = _imageService.GetImageBytesById(id);
}
catch (CatalogImageMissingException ex)
{
    _logger.LogWarning($"No image found for id: {id}");
    return NotFound();
}
return File(imageBytes, "image/png");
}
```

`_logger` y `_imageService` se insertan como dependencias. Ahora se puede probar que el mismo identificador que se pasa al método de acción se pasa a `_imageService` y que los bytes resultantes se devuelven como parte de `FileResult`. También se puede probar que el registro de errores se esté realizando de la forma esperada y que se devuelva un resultado de `NotFound` si la imagen no se encuentra, suponiendo que se trate de un comportamiento importante de la aplicación (es decir, no solo código temporal que el desarrollador ha agregado para diagnosticar una incidencia). La lógica real del archivo se ha trasladado a un servicio de implementación independiente y se ha ampliado para devolver una excepción específica de la aplicación en el caso de un archivo que falta. Puede probar esta implementación de forma independiente, con una prueba de integración.

En la mayoría de los casos, es recomendable que use controladores de excepciones globales en los controladores, por lo que la cantidad de lógica que contengan debería ser mínima y probablemente no valga la pena realizar pruebas unitarias. Haga la mayoría de las pruebas de acciones de controlador con pruebas funcionales y la clase `TestServer`, que se describe a continuación.

## Pruebas de integración en aplicaciones ASP.NET Core

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

# Pruebas funcionales en aplicaciones ASP.NET Core

Para las aplicaciones ASP.NET Core, la clase `TestServer` facilita considerablemente la escritura de pruebas funcionales. Para configurar un elemento `TestServer`, use directamente `WebHostBuilder` (o `HostBuilder`), como lo haría para su aplicación, o bien el tipo `WebApplicationFactory` (disponible a partir de la versión 2.1). Intente que el host de prueba coincida lo máximo posible con el host de producción para que las pruebas ejecuten un comportamiento similar al que tendrá la aplicación en la fase de producción. La clase `WebApplicationFactory` es útil para configurar `ContentRoot` de `TestServer`, que ASP.NET Core usa para localizar recursos estáticos como las vistas.

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

Después de incorporar esta interfaz, el accesorio de prueba puede crear un cliente con el método `CreateClient` de la fábrica:

```
C#  
  
public class BasicWebTests : IClassFixture<WebApplicationFactory<Program>>  
{  
    protected readonly HttpClient _client;  
  
    public BasicWebTests(WebApplicationFactory<Program> factory)  
    {  
        _client = factory.CreateClient();  
    }  
  
    // write tests that use _client  
}
```

## 💡 Sugerencia

Si usa una configuración de API mínima en el archivo `Program.cs`, de forma predeterminada la clase se declarará como interna y no se podrá acceder a ella desde el proyecto de prueba. En su lugar, puede elegir cualquier otra clase de instancia del proyecto web o agregarla al archivo `Program.cs`:

```
C#
```

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

Con frecuencia, querrá configurar opciones adicionales del sitio antes de ejecutar cada prueba, como configurar la aplicación para que use un almacén de datos en memoria y, después, propagar datos de prueba en la aplicación. Para lograr esta función, cree su propia subclase de `WebApplicationFactory<TEntryPoint>` e invalide su método `ConfigureWebHost`. El ejemplo siguiente es del proyecto eShopOnWeb FunctionalTests y se usa como parte de las pruebas en la aplicación web principal.

C#

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

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

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

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

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

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

```

        options.UseInternalServiceProvider(provider);
    });

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

    // Create a scope to obtain a reference to the database
    // context (ApplicationDbContext).
    using (var scope = sp.CreateScope())
    {
        var scopedServices = scope.ServiceProvider;
        var db = scopedServices.GetRequiredService<CatalogContext>();
        var loggerFactory =
            scopedServices.GetRequiredService<ILoggerFactory>();

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

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

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

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

```

Las pruebas pueden hacer uso de esta clase `WebApplicationFactory` personalizada para crear a un cliente y, después, realizar solicitudes a la aplicación mediante esta instancia de cliente. La aplicación tendrá datos propagados que se pueden usar como parte de las aserciones de la prueba. La prueba que encontrará a continuación comprueba que la página principal de la aplicación eShopOnWeb se carga correctamente y que incluye una lista de productos que se ha agregado a la aplicación como parte de los datos de inicialización.

C#

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

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

    public HttpClient Client { get; }

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

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

Esta prueba funcional ejecuta la pila de la aplicación ASP.NET Core MVC o Razor Pages completa, incluido todo el middleware, los filtros y los enlazadores, que puedan estar definidos. Comprueba que una ruta determinada ("/") devuelve el código de estado correcto esperado y la salida HTML. Lo hace sin configurar un servidor web real y evita gran parte de la fragilidad que supone el uso de un servidor web real para realizar pruebas (por ejemplo, problemas con la configuración de firewall). Las pruebas funcionales que se ejecutan en TestServer normalmente son más lentas que las pruebas unitarias y de integración, pero son mucho más rápidas que las que se ejecutarían a través de la red a un servidor web de prueba. Use pruebas funcionales para garantizar que la pila de front-end de la aplicación funciona según lo previsto. Estas pruebas son especialmente útiles al buscar duplicados en controladores o páginas, y solucionar la duplicación mediante la adición de filtros. Idealmente, esta refactorización no cambiará el comportamiento de la aplicación, y un conjunto de pruebas funcionales comprobará que sea así.

## Referencias: prueba de aplicaciones ASP.NET Core MVC

- Pruebas y depuración en ASP.NET Core  
<https://learn.microsoft.com/aspnet/core/testing/>
- Convención de nomenclatura de prueba unitaria  
<https://ardalis.com/unit-test-naming-convention> ↗
- Pruebas de EF Core  
<https://learn.microsoft.com/ef/core/miscellaneous/testing/>
- Pruebas de integración en ASP.NET Core  
<https://learn.microsoft.com/aspnet/core/test/integration-tests>

Anterior

Siguiente

# Proceso de desarrollo para Azure

Artículo • 28/11/2022 • Tiempo de lectura: 6 minutos

## 💡 Sugerencia

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

[Descargar PDF](#)



*"Con la nube, los particulares y las pequeñas empresas pueden configurar servicios de clase empresarial en un abrir y cerrar de ojos".*

- Roy Stephan

## Visión

*Desarrolle aplicaciones ASP.NET Core bien diseñadas como quiera, con Visual Studio o la CLI de DotNet y código de Visual Studio, o bien con el editor que prefiera.*

## Entorno de desarrollo para aplicaciones ASP.NET Core

### Opciones de herramientas de desarrollo: IDE o editor

Con independencia de que prefiera un IDE eficaz y completo, o un editor ligero y ágil, Microsoft le puede ayudar en el desarrollo de aplicaciones ASP.NET Core.

**Visual Studio 2022.** Visual Studio 2022 es el mejor IDE para desarrollar aplicaciones para ASP.NET Core. Ofrece una gran cantidad de características que aumentan la productividad de los desarrolladores. Puede usarlo para desarrollar una aplicación y, a continuación, analizar su rendimiento y otras características. El depurador integrado le permite pausar la ejecución de código, y avanzar y retroceder por este sobre la marcha mientras se ejecuta. Su compatibilidad con las recargas activas le permite seguir trabajando con la aplicación donde lo dejó, incluso después de realizar cambios en el código, sin tener que reiniciar la aplicación. El ejecutor de pruebas integrado le permite organizar las pruebas y sus resultados, e incluso realizar pruebas unitarias en vivo mientras usted programa. Con Live Share, puede colaborar en tiempo real con otros desarrolladores y compartir la sesión de código sin problemas a través de la red. Y cuando esté a punto, Visual Studio incluye todo lo que necesita para publicar la aplicación en Azure o donde pueda hospedarla.

[Descarga de Visual Studio 2022 ↗](#)

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

[Descarga del SDK de .NET ↗](#)

[Descargar Visual Studio Code ↗](#)

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

El ciclo de vida de desarrollo de una aplicación se inicia en el equipo de cada desarrollador, donde se codifica la aplicación con el lenguaje preferido y se prueba de forma local. Los desarrolladores pueden elegir su sistema de control de código fuente preferido y configurar la integración continua (CI) o entrega e implementación continua (CD) con un servidor de compilación o en función de las características integradas de Azure.

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

## Instalación inicial

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

- [Comparta el código con Git y Visual Studio](#), o bien
- [Comparta el código con TFVC y Visual Studio](#).

Cree un Azure App Service donde se va a implementar la aplicación. Vaya a la hoja App Services de Azure Portal y cree una aplicación web. Haga clic en +Aregar, seleccione la plantilla Aplicación web, haga clic en Crear y proporcione un nombre y otros detalles. La aplicación web será accesible desde {nombre}.azurewebsites.net.

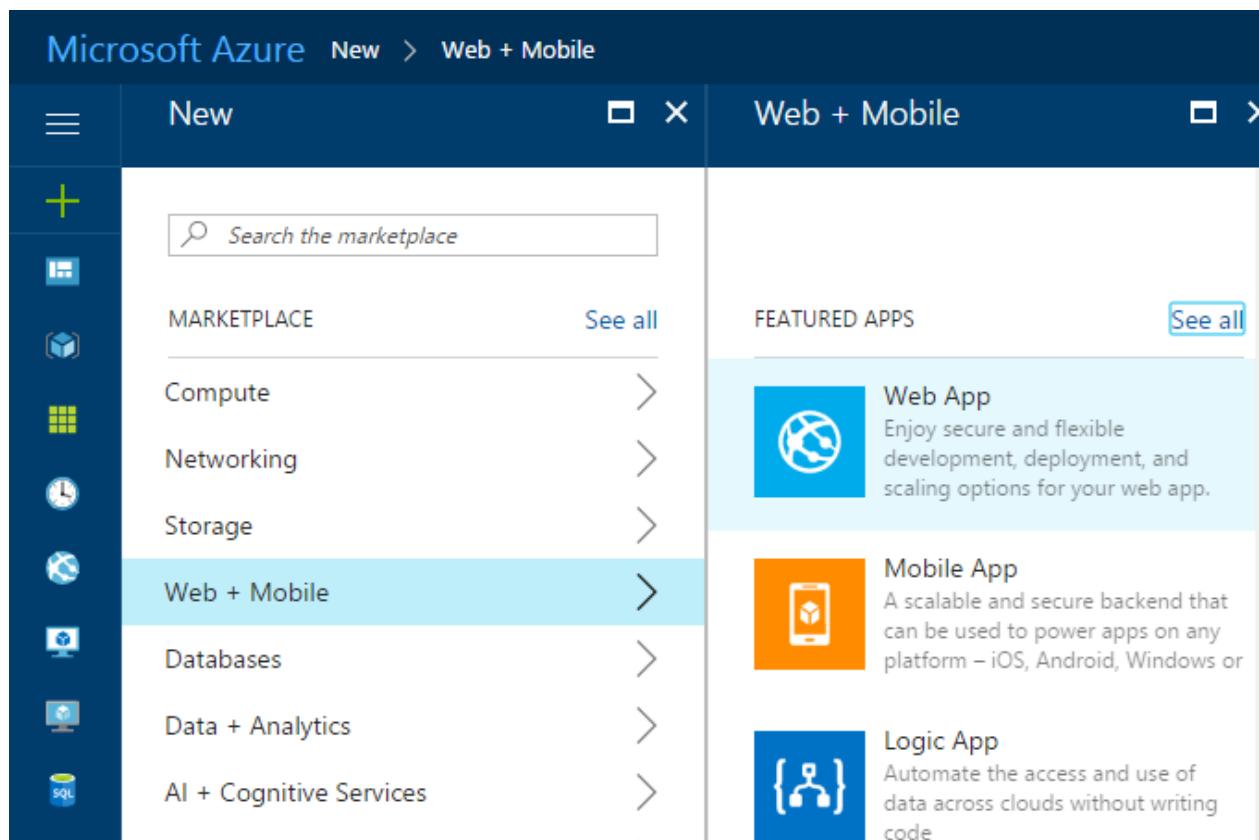


Figura 10-1. Creación de una aplicación web de Azure App Service en Azure Portal.

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

[Definir el proceso de compilación de CI](#)

Asegúrese de habilitar la integración continua para que el sistema ponga en cola una compilación cada vez que alguien del equipo confirme código nuevo. Pruebe la compilación y compruebe que está generando un paquete de implementación web como uno de sus artefactos.

Cuando la compilación se realice correctamente, el proceso de CD implementará los resultados de la compilación de CI en la aplicación web de Azure. Para configurar este paso, se crea y configura una *Versión*, que se implementará en Azure App Service.

### [Implementación de una aplicación web de Azure](#)

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

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

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

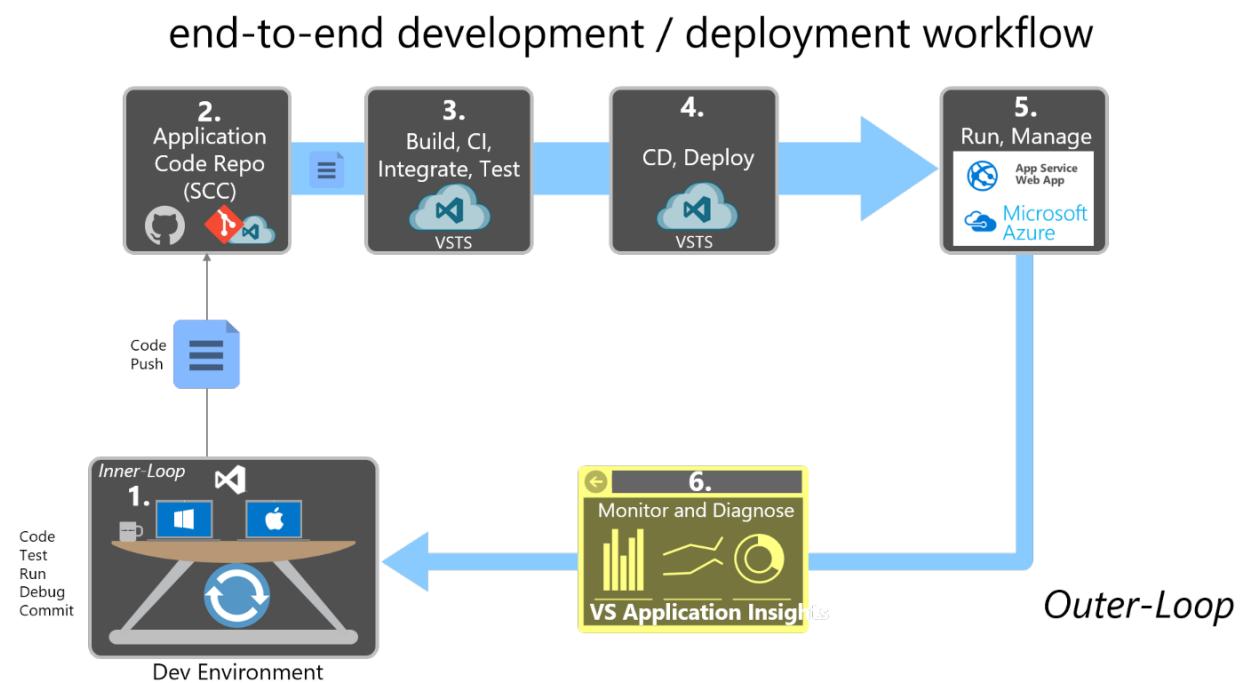


Figura 10-2. Flujo de trabajo paso a paso para compilar aplicaciones ASP.NET Core y hospedarlas en Azure

## Paso 1. Bucle interno del entorno de desarrollo local

El desarrollo de la aplicación ASP.NET Core para su implementación en Azure no es distinto al desarrollo de la aplicación en otros casos. Use el entorno de desarrollo local que prefiera, ya sea Visual Studio 2019 o la CLI de dotnet y Visual Studio Code, o bien el editor de su elección. Puede escribir código, ejecutar y depurar los cambios, ejecutar pruebas automatizadas y realizar confirmaciones locales en el control de código fuente hasta que esté listo para insertar los cambios en el repositorio de control de código fuente compartido.

## Paso 2. Repositorio de código de la aplicación

Cuando esté listo para compartir el código con el equipo, debe insertar los cambios desde el repositorio de origen local al repositorio de código fuente compartido del equipo. Si ha estado trabajando en una rama personalizada, este paso normalmente implica combinar el código en una rama compartida (posiblemente por medio de una [solicitud de incorporación de cambios](#)).

## Paso 3. Servidor de compilación: Integración continua. Compilar, probar, empaquetar

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

## Paso 4. Servidor de compilación: Entrega continua.

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

## Paso 5. Aplicación web de Azure App Service

Una vez implementada, la aplicación ASP.NET Core se ejecuta en el contexto de una aplicación de web de Azure App Service. Esta aplicación web se puede supervisar y

configurar más mediante Azure Portal.

## Paso 6. Supervisión de producción y diagnóstico

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

## Referencias

**Build and Deploy Your ASP.NET Core App to Azure** (Compilación e implementación de la aplicación ASP.NET Core en Azure)

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

[Anterior](#)

[Siguiente](#)

# Recomendaciones de hospedaje en Azure de aplicaciones web ASP.NET Core

Artículo • 28/11/2022 • Tiempo de lectura: 11 minutos

## 💡 Sugerencia

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

[Descargar PDF](#)



"Los líderes de línea de negocio en todas partes evitan que los departamentos de TI tengan que obtener aplicaciones de la nube (también conocidas como SaaS) y pagar por ellas como harían con la suscripción a una revista. Y cuando el servicio ya no es necesario, pueden cancelar la suscripción sin dejar ningún equipo sin usar en un rincón".

*: Daryl Plummer, analista de Gartner*

Sean cuales sean las necesidades y la arquitectura de la aplicación, Microsoft Azure puede darle servicio. Las necesidades de hospedaje pueden ser tan simples como un sitio web estático o una aplicación sofisticada formada por decenas de servicios. Para las aplicaciones web monolíticas de ASP.NET Core y los servicios complementarios, se recomiendan varias configuraciones conocidas. Las recomendaciones de este artículo se agrupan según el tipo de recurso que se va a hospedar, ya sean aplicaciones completas, procesos individuales o datos.

## Aplicaciones web

Las aplicaciones web se pueden hospedar con:

- App Service Web Apps
- Contenedores (varias opciones)
- Máquinas virtuales (VM)

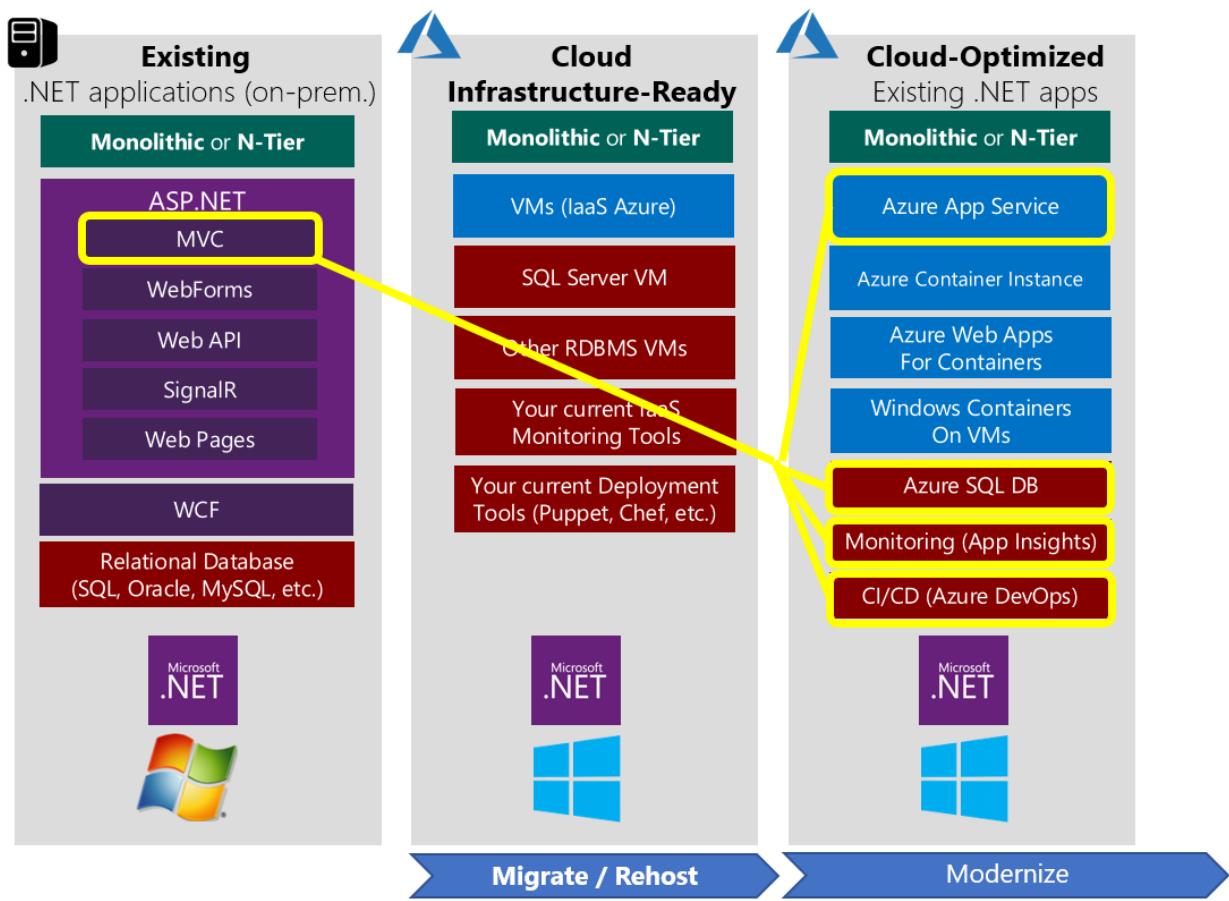
De estas, App Service Web Apps es el enfoque recomendado para la mayoría de los escenarios, incluidas las aplicaciones sencillas basadas en contenedores. Para las arquitecturas de microservicios, considere la posibilidad de un enfoque basado en contenedores. Si necesita más control sobre los equipos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

## App Service Web Apps

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

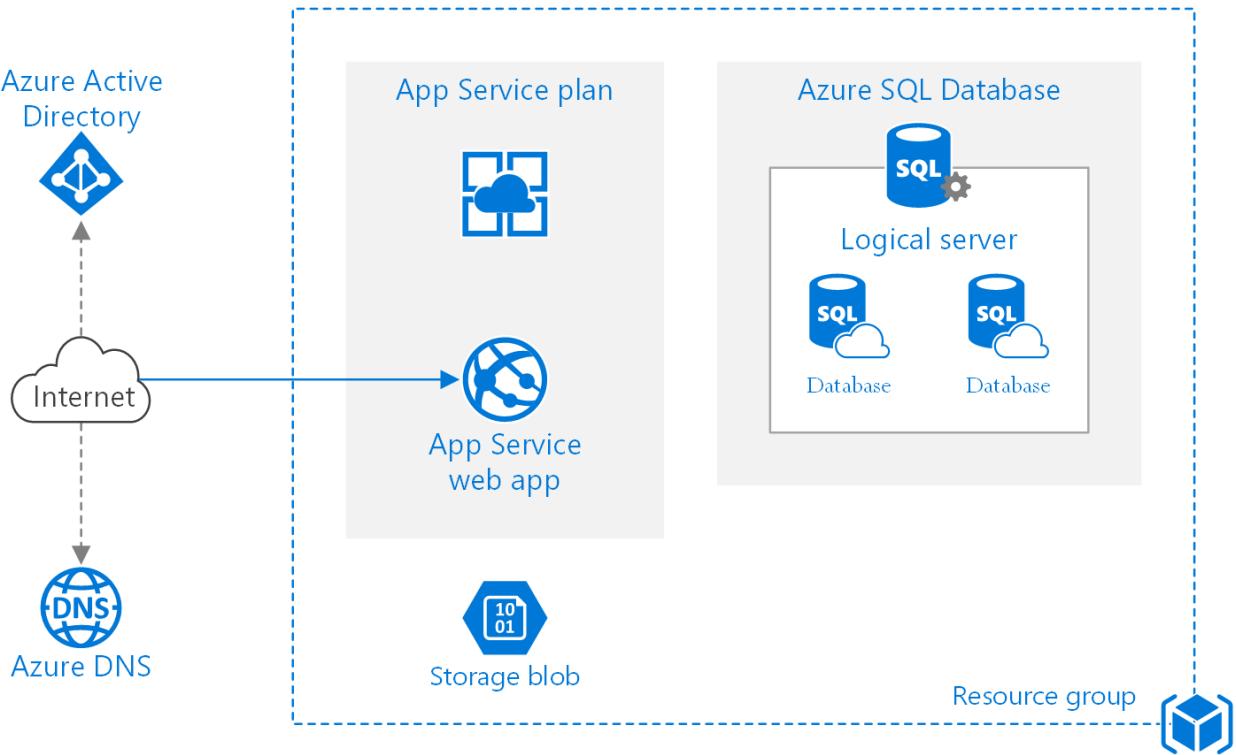
- Optimización de DevOps (integración y entrega continuas, varios entornos, pruebas A/B, compatibilidad con scripting).
- Escala global y alta disponibilidad.
- Conexiones a plataformas SaaS y datos locales.
- Seguridad y cumplimiento.
- Integración de Visual Studio.

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



En la mayoría de los casos, el traslado de una aplicación ASP.NET hospedada localmente a una aplicación web de App Service es un proceso sencillo. Se requiere poca o ninguna modificación de la propia aplicación, y puede comenzar rápidamente a aprovechar las muchas características que ofrece Azure App Service Web Apps.

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

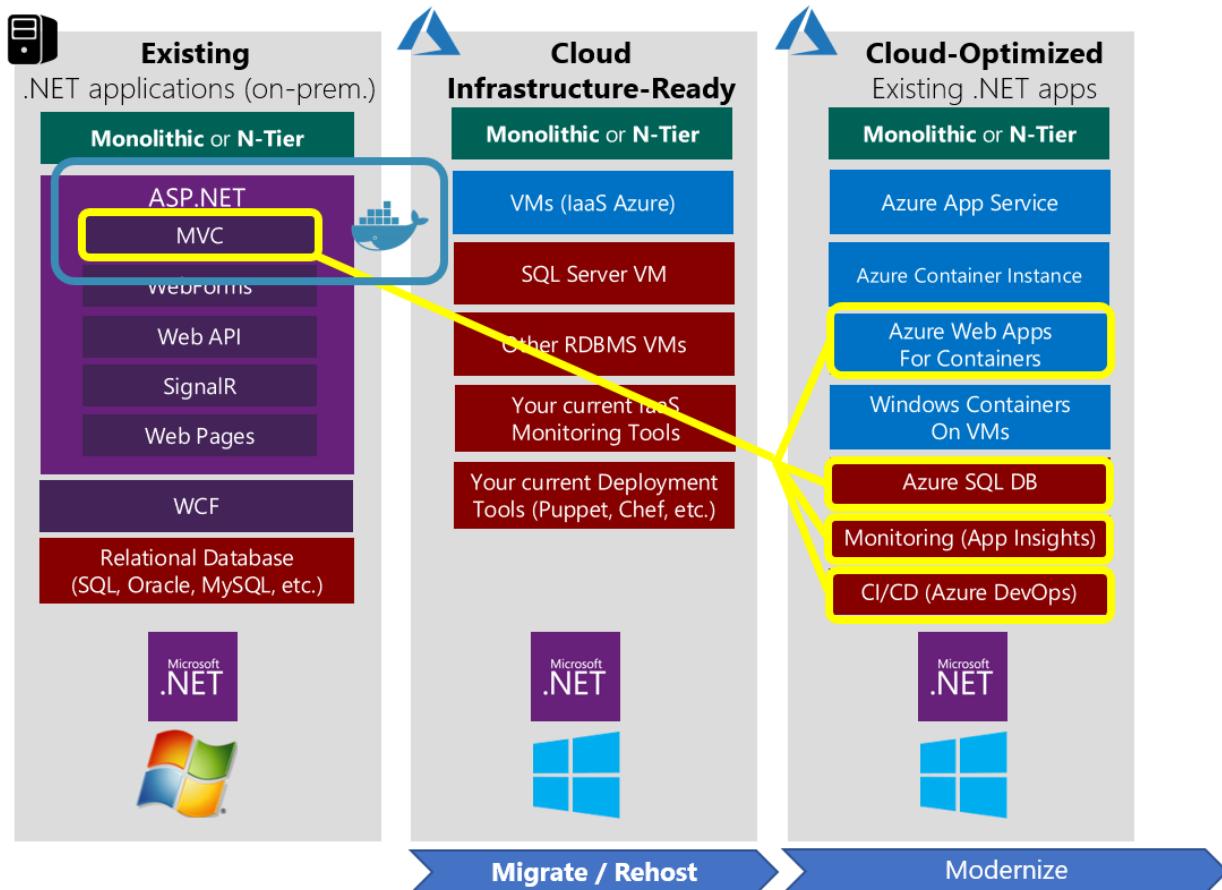


Un número pequeño de recursos en un único grupo de recursos es normalmente suficiente para administrar este tipo de aplicación. Las aplicaciones que se implementan normalmente como una sola unidad, en lugar de las aplicaciones que se componen de muchos procesos independientes, son buenas candidatas para este [enfoque arquitectónico básico](#). A pesar de su arquitectura sencilla, este enfoque permite a la aplicación hospedada escalarse de manera vertical (más recursos por nodo) y horizontal (más nodos hospedados) para satisfacer cualquier aumento en la demanda. Con el escalado automático, la aplicación puede configurarse para ajustar automáticamente el número de nodos que hospedan la aplicación según la demanda y la carga media entre los nodos.

## App Service Web Apps for Containers

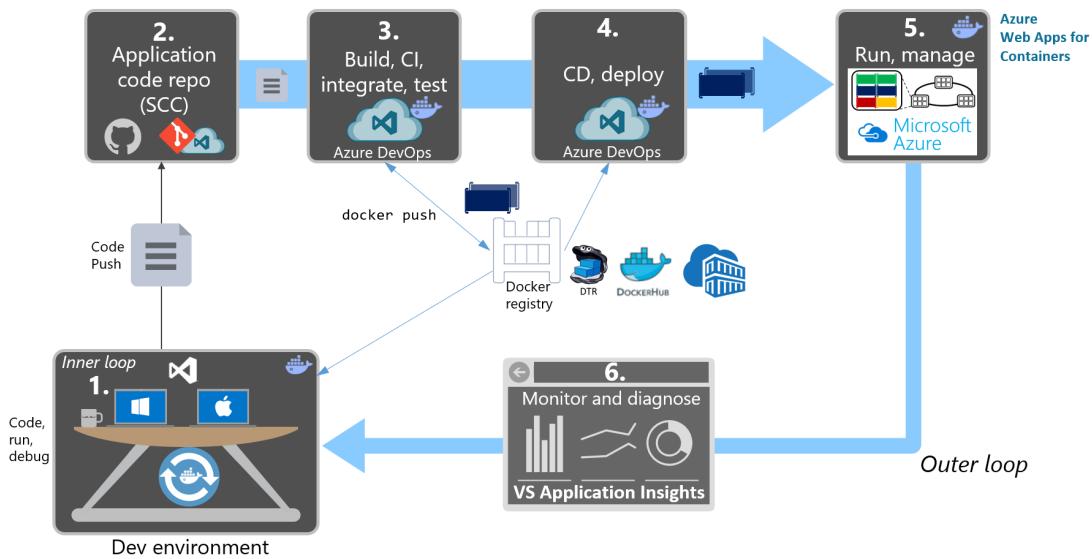
Además de para brindar compatibilidad para hospedar aplicaciones web directamente, [App Service Web Apps for Containers](#) puede usarse para ejecutar aplicaciones en contenedores en Windows y Linux. Con este servicio, puede implementar y ejecutar fácilmente aplicaciones en contenedores que se pueden escalar con su negocio. Las aplicaciones tienen todas las características de App Service Web Apps mencionadas anteriormente. Además, Web Apps for Containers admite CI/CD simplificadas con Docker Hub, Azure Container Registry y GitHub. Puede usar Azure DevOps para definir las canalizaciones de compilación e implementación que publican los cambios en un registro. Estos cambios pueden probarse entonces en un entorno de ensayo e implementarse automáticamente en producción mediante ranuras de implementación, permitiendo así actualizaciones sin tiempo de inactividad. La reversión a versiones anteriores puede hacerse fácilmente.

Hay algunos escenarios donde Web Apps for Containers es la solución más conveniente. Si tiene aplicaciones que puede incluir en contenedores, ya sean de Windows o Linux, puede hospedarlas fácilmente con este conjunto de herramientas. Simplemente publique su contenedor y, después, configure Web Apps for Containers para extraer la versión más reciente de esa imagen del registro de su elección. Este es un enfoque de "elevación y desplazamiento" para migrar desde los modelos de hospedaje de aplicaciones clásicos a un modelo optimizado para la nube.



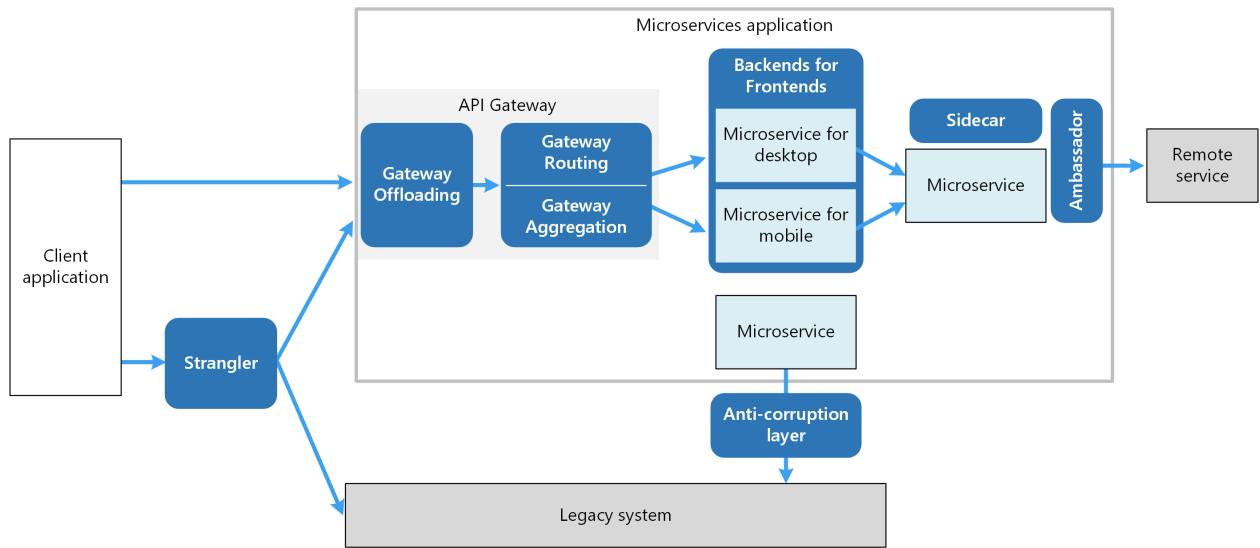
Este enfoque también funciona bien si el equipo de desarrollo puede pasarse a un proceso de desarrollo basado en contenedores. El "bucle interno" del desarrollo de aplicaciones con contenedores incluye la creación de la aplicación con contenedores. Los cambios realizados en el código, así como en la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

# End to End Docker DevOps Lifecycle Workflow



El desarrollo con contenedores ofrece muchas ventajas, especialmente cuando se usan contenedores en producción. Se usa la misma configuración de los contenedores para hospedar la aplicación en cada entorno en el que se ejecuta, desde la máquina de desarrollo local hasta la compilación y prueba de sistemas en producción. Este enfoque reduce en gran medida la probabilidad de que surjan defectos resultantes de las diferencias en la configuración de la máquina o las versiones de software. Los desarrolladores también pueden usar las herramientas con las que sean más productivos, incluido el sistema operativo, ya que los contenedores pueden ejecutarse en cualquier sistema operativo. En algunos casos, las aplicaciones distribuidas que implican muchos contenedores pueden requerir una gran cantidad de recursos para ejecutarse en una sola máquina de desarrollo. En este escenario, puede ser conveniente actualizar para utilizar Kubernetes y Azure Dev Spaces, que se describen en la siguiente sección.

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



Más información sobre los patrones de diseño que se deben considerar al crear sistemas basados en microservicios.

## Azure Kubernetes Service

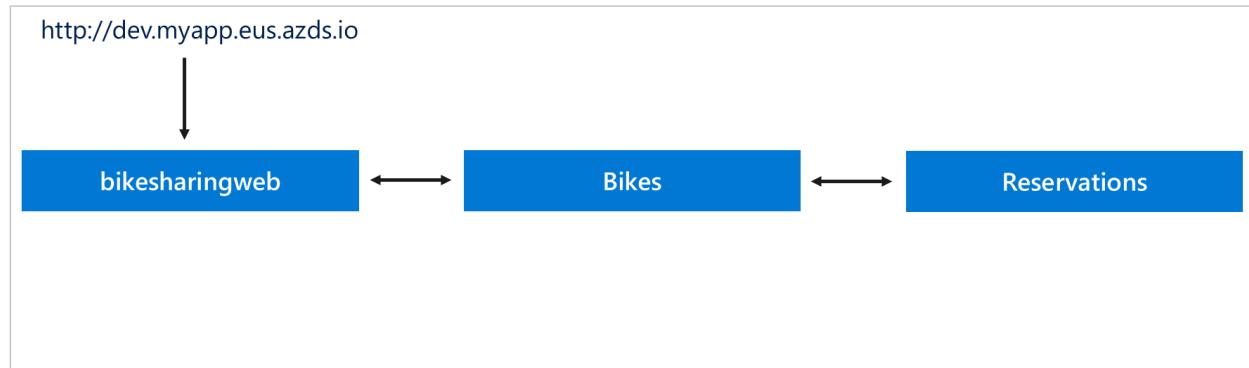
Azure Kubernetes Service (AKS) administra el entorno hospedado de Kubernetes, lo que acelera y facilita la implementación y administración de aplicaciones en contenedores sin necesidad de tener conocimientos de orquestación de contenedores. También se elimina la carga de las operaciones en curso y el mantenimiento mediante el aprovisionamiento, la actualización y el escalado de los recursos a petición, sin tener que desconectar las aplicaciones.

AKS reduce la complejidad y la sobrecarga operativa de la administración de un clúster de Kubernetes mediante la descarga de gran parte de esa responsabilidad en Azure. Como un servicio de Kubernetes hospedado, Azure controla de forma automática tareas críticas como el seguimiento de estado y el mantenimiento. Además, solo se paga por los nodos de agente en los clústeres, no por los nodos principales. Como servicio de Kubernetes administrado, AKS proporciona lo siguiente:

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

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

Los equipos que están aprovechando AKS también pueden sacar partido de Azure Dev Spaces. Azure Dev Spaces ayuda a los equipos a centrarse en el desarrollo y la iteración rápida de su aplicación de microservicio, ya que permite a los equipos trabajar directamente con su aplicación o arquitectura de microservicios completa que se ejecuta en AKS. Azure Dev Spaces también proporciona una manera de actualizar independientemente partes de la arquitectura de microservicios de forma aislada sin afectar al resto del clúster de AKS o de otros desarrolladores.



Azure Dev Spaces:

- Minimizar los requisitos de recursos y el tiempo de configuración de la máquina local
- Permitir que los equipos iteren de manera más rápida
- Reducir el número de entornos de integración que requiere un equipo
- Eliminar la necesidad de simular determinados servicios en un sistema distribuido para desarrollo y pruebas

[Más información sobre Azure Dev Spaces.](#)

## Azure Virtual Machines

Si tiene una aplicación existente que requeriría modificaciones importantes para ejecutarse en App Service, podría elegir Virtual Machines con el fin de simplificar la migración a la nube. Aunque la configuración, la protección y el mantenimiento correctos de las máquinas virtuales requiere mucho más tiempo y experiencia en TI en comparación con Azure App Service. Si está pensando en Azure Virtual Machines, asegúrese de tener en cuenta el esfuerzo de mantenimiento continuado necesario para aplicar revisiones, actualizar y administrar el entorno de máquinas virtuales.

Azure Virtual Machines es una infraestructura como servicio (IaaS), mientras que App Service es PaaS. También debe considerar si la implementación de la aplicación como un contenedor de Windows en Web App for Containers podría ser una opción viable para su escenario.

# Procesos lógicos

Los procesos lógicos individuales que se pueden desacoplar del resto de la aplicación se pueden implementar por separado en Azure Functions de forma "sin servidor". Azure Functions permite escribir simplemente el código que se necesita para un problema determinado, sin preocuparse por la aplicación o infraestructura para ejecutarlo. Se puede elegir entre una variedad de lenguajes de programación, incluyendo C#, F#, Node.js, Python y PHP, lo que permite elegir el lenguaje más productivo para la tarea en cuestión. Al igual que la mayoría de las soluciones basadas en la nube, solo se paga por la cantidad de tiempo que se usa y se puede confiar en Azure Functions para escalar verticalmente según sea necesario.

## data

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

Para los datos transaccionales y relacionales, Azure SQL Database es la mejor opción. Para los datos de alto rendimiento y principalmente de solo lectura, una caché en Redis respaldada por Azure SQL Database es una buena solución.

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

Los datos transitorios basados en eventos o comandos que se usan para coordinar el comportamiento de la aplicación pueden usar Azure Service Bus o Azure Storage Queue. Azure Storage Bus ofrece más flexibilidad y es el servicio recomendado para mensajería no trivial dentro y entre las aplicaciones.

## Recomendaciones de arquitectura

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

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

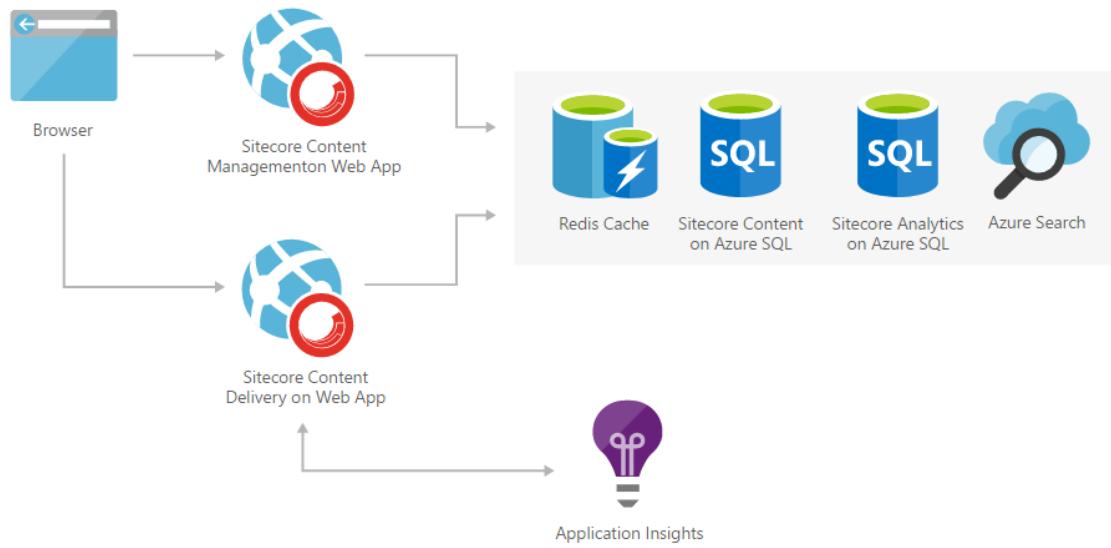


Figura 11-1. Arquitectura de referencia del sitio web de marketing Sitecore.

#### Referencias: recomendaciones de hospedaje de Azure

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