

Blazor para desarrolladores de ASP.NET Web Forms

18/06/2025



Blazor for ASP.NET Web Forms Developers



Daniel Roth

Jeff Fritz

Taylor Southwick

Version 6.0

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

EDITION v8.0 : actualizado a .NET 8

Consulte el [registro de cambios](#) ↗ para las actualizaciones del libro y las contribuciones de la comunidad.

PUBLICADO POR

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

Una división de Microsoft Corporation

One Microsoft Way

Redmond, Washington 98052-6399

Copyright © 2023 de Microsoft Corporation

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

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

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

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

Mac y macOS son marcas comerciales de Apple Inc.

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

Autores:

Daniel Roth ↗, director de programas principal, Microsoft Corp.

Jeff Fritz ↗, gerente sénior del programa, Microsoft Corp.

Taylor Southwick ↗, ingeniero sénior de software, Microsoft Corp.

[Scott Addie](#) , desarrollador sénior de contenido, Microsoft Corp.

[Steve "@ardalis" Smith](#) , arquitecto de software y entrenador, NimblePros.com

Introducción

.NET ha apoyado durante mucho tiempo el desarrollo de aplicaciones web a través de ASP.NET, un conjunto completo de frameworks y herramientas para desarrollar cualquier tipo de aplicación web. ASP.NET tiene su propio linaje de marcos web y tecnologías, empezando por Active Server Pages (ASP). Marcos como ASP.NET Web Forms, ASP.NET MVC, ASP.NET Web Pages y, más recientemente, ASP.NET Core, proporcionan una manera productiva y eficaz de crear aplicaciones web *representadas por el servidor*, donde el contenido de la interfaz de usuario se genera dinámicamente en el servidor en respuesta a solicitudes HTTP. Cada marco de ASP.NET se ocupa de una filosofía diferente de creación de aplicaciones y audiencias. ASP.NET Web Forms se incluyeron con la versión original de .NET Framework y habilitaron el desarrollo web usando muchos de los patrones familiares para los desarrolladores de escritorio, como controles de interfaz de usuario reutilizables con manejo de eventos sencillo. Sin embargo, ninguna de las ofertas de ASP.NET proporciona una manera de ejecutar código que se ejecuta en el explorador del usuario. Para ello, es necesario escribir JavaScript y usar cualquiera de los muchos marcos y herramientas de JavaScript que han cambiado su popularidad a lo largo de los años: jQuery, Knockout, Angular, React, y demás.

[Blazor](#)  es un nuevo marco web que cambia lo que es posible al compilar aplicaciones web con .NET. Blazor es un marco de interfaz de usuario web del lado cliente basado en C# en lugar de JavaScript. Con Blazor puede escribir la lógica del lado cliente y los componentes de la interfaz de usuario en C#, compilarlos en ensamblados .NET normales y, a continuación, ejecutarlos directamente en el explorador mediante un nuevo estándar web abierto denominado WebAssembly. O bien, Blazor puede ejecutar los componentes de la interfaz de usuario de .NET en el servidor y controlar todas las interacciones de la interfaz de usuario de forma fluida a través de una conexión en tiempo real con el explorador. Al emparejar Blazor con la instancia de .NET que se ejecuta en el servidor, permite el desarrollo web de pila completa con .NET. Aunque Blazor comparte muchas commonalities con ASP.NET Web Forms, como tener un modelo de componente reutilizable y una manera sencilla de controlar eventos de usuario, también se basa en las bases de .NET para proporcionar una experiencia de desarrollo web moderna y de alto rendimiento.

En este libro se presenta a los desarrolladores de ASP.NET Web Forms a Blazor de una manera que sea familiar y conveniente. Presenta conceptos en paralelo con conceptos análogos Blazor en ASP.NET Web Forms, a la vez que explica nuevos conceptos que pueden ser menos conocidos. Trata una amplia gama de temas y preocupaciones, como la creación de componentes, el enrutamiento, el diseño, la configuración y la seguridad. Y aunque el

contenido de este libro es principalmente para habilitar el nuevo desarrollo, también trata directrices y estrategias para migrar formularios web existentes de ASP.NET a Blazor para cuando quiera modernizar una aplicación existente.

Quién debe usar el libro

Este libro es para ASP.NET desarrolladores de Web Forms que buscan una introducción a Blazor que se relaciona con sus conocimientos y aptitudes existentes. Este libro puede ayudar a empezar rápidamente con un nuevo proyecto basado en Blazor o ayudar a trazar un mapa de ruta para modernizar una aplicación existente de ASP.NET Web Forms.

Cómo usar el libro

La primera parte de este libro trata lo que Blazor es y lo compara con el desarrollo de aplicaciones web con ASP.NET Web Forms. A continuación, el libro trata una variedad de Blazor temas, capítulo por capítulo y relaciona cada Blazor concepto con el concepto correspondiente en ASP.NET Web Forms, o explica completamente cualquier concepto completamente nuevo. El libro también hace referencia periódicamente a una aplicación de ejemplo completa implementada en ASP.NET Web Forms y Blazor para mostrar Blazor características y proporcionar un caso práctico para migrar de ASP.NET Web Forms a Blazor. Puede encontrar ambas implementaciones de la aplicación de ejemplo (ASP.NET web Forms y Blazor versiones) en [GitHub](#).

Lo que este libro no cubre

Este libro es una introducción a Blazor, no una guía de migración completa. Aunque incluye instrucciones sobre cómo abordar la migración de un proyecto de ASP.NET Web Forms a Blazor, no intenta cubrir todos los matices y detalles. Para obtener instrucciones más generales sobre cómo migrar de ASP.NET a ASP.NET Core, consulte la [guía de migración](#) en la documentación de ASP.NET Core.

Recursos adicionales

Puede encontrar la página principal oficial Blazor y la documentación en <https://blazor.net>.

Enviar sus comentarios

Este libro y ejemplos relacionados evolucionan constantemente, por lo que se agradecen sus comentarios. Si tiene comentarios sobre cómo se puede mejorar este libro y está leyendo este

libro en <learn.microsoft.com>, use la sección **Comentarios** en la parte inferior de la página:

Feedback

Submit and view feedback for

[🔄 View all page feedback](#)

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

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

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

Próximo

Introducción a Blazor para desarrolladores de formularios web de ASP.NET

19/06/2025

💡 Sugerencia

Este contenido es un extracto del libro electrónico "Blazor for ASP.NET Web Forms Developers for Azure" (Blazor para desarrolladores de ASP.NET Web Forms), disponible en [Documentación de .NET](#) o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF



El marco de ASP.NET Web Forms ha sido un elemento básico del desarrollo web de .NET desde que .NET Framework se envió por primera vez en 2002. De nuevo, cuando la Web todavía estaba en su infancia, ASP.NET Web Forms hizo que la creación de aplicaciones web sea sencilla y productiva mediante la adopción de muchos de los patrones que se usaron para el desarrollo de escritorio. En ASP.NET Web Forms, las páginas web se pueden componer rápidamente a partir de controles de interfaz de usuario reutilizables. Las interacciones del usuario se controlan de forma natural como eventos. Hay un amplio ecosistema de controles de interfaz de usuario de Formularios Web Forms proporcionados por Microsoft y proveedores de control. Los controles facilitan los esfuerzos de conectarse a orígenes de datos y mostrar visualizaciones de datos enriquecidas. Para los que prefieren lo visual, el diseñador de Web Forms proporciona una sencilla interfaz de arrastrar y soltar para administrar controles.

A lo largo de los años, Microsoft ha introducido nuevos marcos web basados en ASP.NET para abordar las tendencias de desarrollo web. Algunos de estos marcos web incluyen ASP.NET MVC, ASP.NET Web Pages y, más recientemente, ASP.NET Core. Con cada nuevo framework, algunos han pronosticado la inminente desaparición de ASP.NET Web Forms y lo han criticado como un marco web anticuado y obsoleto. A pesar de estas predicciones, muchos desarrolladores web de .NET siguen buscando ASP.NET Web Forms una manera sencilla, estable y productiva de realizar su trabajo.

En el momento de escribirlo, casi medio millón de desarrolladores web usan ASP.NET formularios web cada mes. El marco de web Forms de ASP.NET es estable hasta el punto de que los documentos, ejemplos, libros y entradas de blog de hace una década siguen siendo útiles y relevantes. Para muchos desarrolladores web de .NET, "ASP.NET" sigue siendo sinónimo de "ASP.NET Web Forms" como era cuando .NET se concibió por primera vez. Los argumentos sobre las ventajas y desventajas de ASP.NET Web Forms en comparación con otros nuevos marcos web de .NET pueden continuar intensamente. ASP.NET Web Forms sigue siendo un marco popular para crear aplicaciones web.

Incluso así, las innovaciones en el desarrollo de software no se ralentizan. Todos los desarrolladores de software necesitan mantenerse al día de las nuevas tecnologías y tendencias. Cabe tener en cuenta dos tendencias en particular:

1. El cambio a código abierto y multiplataforma
2. El cambio de la lógica de la aplicación al cliente

Un .NET multiplataforma y de código abierto

Cuando .NET y ASP.NET Web Forms fueron lanzados por primera vez, el ecosistema de plataformas era muy diferente al de hoy en día. Los mercados de escritorio y servidor estaban dominados por Windows. Las plataformas alternativas como macOS y Linux todavía estaban luchando por ganar tracción. ASP.NET Web Forms se incluye con .NET Framework como componente exclusivo para Windows, lo que significa que las aplicaciones ASP.NET Web Forms solo se pueden ejecutar en servidores Windows. Muchos entornos modernos ahora usan diferentes tipos de plataformas para servidores y máquinas de desarrollo, de modo que la compatibilidad multiplataforma con muchos usuarios es un requisito absoluto.

La mayoría de los marcos web modernos ahora también son de código abierto, lo que tiene una serie de ventajas. Los usuarios no están limitados a un único propietario del proyecto para corregir errores y agregar características. Los proyectos de código abierto proporcionan una mayor transparencia en el progreso del desarrollo y los próximos cambios. Los proyectos de código abierto disfrutan de contribuciones de toda una comunidad y fomentan un ecosistema de código abierto de apoyo. A pesar de los riesgos de código abierto, muchos consumidores y colaboradores han encontrado mitigaciones adecuadas que les permiten disfrutar de las ventajas de un ecosistema de código abierto de una manera segura y razonable. Algunos ejemplos de estas mitigaciones son los contratos de licencia de colaborador, las licencias descriptivas, los exámenes de pedigrí y las bases de soporte.

La comunidad de .NET ha adoptado compatibilidad multiplataforma y código abierto. .NET Core es una implementación multiplataforma y de código abierto de .NET que se ejecuta en una gran cantidad de plataformas, como Windows, macOS y varias distribuciones de Linux. Mono es una versión de código abierto de .NET que se ejecuta en Android, iOS y una variedad

de otros formatos, incluidos relojes y televisores inteligentes. En 2020, Microsoft lanzó [.NET 5](#) que concilió .NET Core y Mono en "un único entorno de ejecución y marco de .NET que se puede usar en todas partes y que tiene comportamientos uniformes en tiempo de ejecución y experiencias de desarrollador".

¿Se beneficiará ASP.NET Web Forms del traslado a compatibilidad multiplataforma y de código abierto? La respuesta, desafortunadamente, es no, o, al menos, no en la misma medida que el resto de la plataforma. El equipo de .NET [dejó claro](#) que ASP.NET Web Forms no se migrará a .NET Core ni a .NET 8. ¿Por qué ocurre esto?

Hubo esfuerzos en los primeros días de .NET Core para migrar ASP.NET Web Forms. Se comprobó que el número de cambios importantes necesarios era demasiado drástico. También hay una admisión aquí que incluso para Microsoft, hay un límite para el número de marcos web que puede admitir simultáneamente. Quizás alguien de la comunidad asumirá la causa de crear una versión de ASP.NET Web Forms que sea multiplataforma y de código abierto. El [código fuente de ASP.NET Web Forms](#) se ha puesto a disposición públicamente en forma de referencia. Pero por el momento, parece que ASP.NET Web Forms seguirá siendo solo Windows y sin un modelo de contribución de código abierto. Si la compatibilidad multiplataforma o el código abierto son importantes para sus escenarios, deberá buscar algo nuevo.

¿Esto significa que ASP.NET formularios web está *inactivo* y ya no se debe usar? ¡Claro que no! Siempre que .NET Framework se incluya como parte de Windows, ASP.NET Web Forms será un marco compatible. Para muchos desarrolladores de Web Forms, la falta de compatibilidad multiplataforma y de código abierto no es un problema. Si no tiene ningún requisito para la compatibilidad multiplataforma, el código abierto o ninguna de las otras características nuevas de .NET Core o .NET 8, después seguir con ASP.NET Web Forms en Windows es correcto. ASP.NET Web Forms seguirá siendo una manera productiva de escribir aplicaciones web durante muchos años.

Pero hay otra tendencia que merece la pena tener en cuenta y eso es el cambio al cliente.

Desarrollo web del lado cliente

Todos los marcos web basados en .NET, incluidos ASP.NET Web Forms, han tenido históricamente una cosa en común: se *renderizan en el servidor*. En las aplicaciones web representadas por el servidor, el explorador realiza una solicitud al servidor, que ejecuta algún código (código .NET en ASP.NET aplicaciones) para generar una respuesta. Esa respuesta se devuelve al navegador para gestionarla. En este modelo, el explorador se usa como un motor de representación fino. El trabajo duro de generar la interfaz de usuario, ejecutar la lógica de negocios y administrar el estado se produce en el servidor.

Sin embargo, los exploradores se han convertido en plataformas versátiles. Implementan un número cada vez mayor de estándares web abiertos que conceden acceso a las funcionalidades de la máquina del usuario. ¿Por qué no aprovechar la potencia de proceso, el almacenamiento, la memoria y otros recursos del dispositivo cliente? Las interacciones de la interfaz de usuario en particular pueden beneficiarse de una sensación más enriquecida y más interactiva cuando se administran al menos parcialmente o completamente en el lado cliente. La lógica y los datos que deben manejarse en el servidor todavía pueden gestionarse del lado del servidor. Se pueden usar llamadas a API web o incluso protocolos en tiempo real, como WebSockets. Estas ventajas están disponibles para los desarrolladores web de forma gratuita si están dispuestos a escribir JavaScript. Los marcos de interfaz de usuario del lado cliente, como Angular, React y Vue, simplifican el desarrollo web del lado cliente y han crecido en popularidad. Los desarrolladores de ASP.NET Web Forms también pueden beneficiarse del uso del cliente e incluso contar con la compatibilidad de serie con marcos de JavaScript integrados como ASP.NET AJAX.

Pero el puente entre dos plataformas y ecosistemas diferentes (.NET y JavaScript) conlleva un costo. La experiencia es necesaria en dos mundos paralelos con diferentes lenguajes, marcos y herramientas. El código y la lógica no se pueden compartir fácilmente entre el cliente y el servidor, lo que da lugar a la sobrecarga de duplicación e ingeniería. También puede ser difícil mantenerse al día con el ecosistema de JavaScript, que tiene un historial de evolución a velocidad vertiginosa. Las preferencias de la herramienta de compilación y el marco de front-end cambian rápidamente. La industria ha observado la progresión de Grunt a Gulp a Webpack, etc. El mismo ritmo vertiginoso se ha producido en marcos de front-end como jQuery, Knockout, Angular, React y Vue. Sin embargo, dado el monopolio del explorador de JavaScript, había poca opción en el asunto. Es decir, hasta que la comunidad web se reunió y causó que un *milagro* sucediera.

WebAssembly satisface una necesidad

En 2015, los principales proveedores de exploradores se unieron a las fuerzas de un grupo de la comunidad W3C para crear un nuevo estándar web abierto denominado WebAssembly. WebAssembly es un código de bytes para la Web. Si puede compilar el código en WebAssembly, puede ejecutarse en cualquier explorador de cualquier plataforma a una velocidad nativa cercana. Los esfuerzos iniciales se centraron en C/C++. El resultado fue una demostración dramática de la ejecución de motores gráficos 3D nativos directamente en el explorador sin complementos. WebAssembly desde entonces ha sido estandarizado e implementado por todos los exploradores principales.

El trabajo para ejecutar .NET en WebAssembly se anunció a finales de 2017 y se publicó en 2020, incluida la compatibilidad de .NET 5 y versiones posteriores. La capacidad de ejecutar

código de .NET directamente en el explorador permite el desarrollo web de pila completa con .NET.

Blazor: desarrollo web completo con .NET

Por su cuenta, la capacidad de ejecutar código .NET en un explorador no proporciona una experiencia de un extremo a otro para crear aplicaciones web del lado cliente. Ahí es donde Blazor entra. Blazor es un marco de interfaz de usuario web del lado cliente basado en C# en lugar de JavaScript. Blazor puede ejecutarse directamente en el explorador a través de WebAssembly. No se requieren complementos de explorador. Como alternativa, las Blazor aplicaciones pueden ejecutar el lado servidor en .NET y controlar todas las interacciones del usuario a través de una conexión en tiempo real con el explorador.

Blazor tiene una gran compatibilidad con herramientas en Visual Studio y Visual Studio Code. El marco también incluye un modelo de componentes de interfaz de usuario completo y tiene instalaciones integradas para:

- Formularios y validación
- Inserción de dependencia
- Enrutamiento del lado cliente
- Diseños
- Depuración en el explorador
- Interoperabilidad de JavaScript

Blazor tiene mucho en común con ASP.NET Web Forms. Ambos marcos ofrecen modelos de programación de interfaz de usuario con estado basados en componentes, controlados por eventos. La principal diferencia arquitectónica es que ASP.NET Web Forms solo se ejecuta en el servidor. Blazor se puede ejecutar en el cliente en el explorador. Pero si viene de una experiencia en ASP.NET Web Forms, hay mucho en Blazor que le resultará familiar. Blazor es una solución natural para ASP.NET desarrolladores de Web Forms que buscan una manera de aprovechar el desarrollo del lado cliente y el futuro multiplataforma de código abierto de .NET.

Este libro proporciona una introducción a Blazor orientada específicamente a desarrolladores de ASP.NET Web Forms. Cada Blazor concepto se presenta en el contexto de características y prácticas análogas de ASP.NET Web Forms. Al final de este libro, comprenderá lo siguiente:

- Cómo crear Blazor aplicaciones.
- Cómo Blazor funciona.
- Cómo Blazor se relaciona con .NET.
- Estrategias razonables para migrar aplicaciones existentes de ASP.NET Web Forms a Blazor cuando corresponda.

Empieza con Blazor

Empezar con Blazor es fácil. Vaya a <https://blazor.net> y siga los vínculos para instalar el SDK de .NET y Blazor las plantillas de proyecto adecuadas. También encontrará instrucciones para configurar las Blazor herramientas en Visual Studio o Visual Studio Code.

Siguiente anterior

Comparación de arquitectura de ASP.NET Web Forms y Blazor

22/06/2025

💡 Sugerencia

Este contenido es un extracto del libro electrónico "Blazor for ASP.NET Web Forms Developers for Azure" (Blazor para desarrolladores de ASP.NET Web Forms), disponible en [Documentación de .NET](#) o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF



Aunque ASP.NET Web Forms y Blazor tienen muchos conceptos similares, existen diferencias en el funcionamiento. En este capítulo se examinan los trabajos internos y las arquitecturas de ASP.NET Web Forms y Blazor.

formularios web de ASP.NET

El marco de ASP.NET Web Forms se basa en una arquitectura centrada en páginas. Cada solicitud HTTP para una ubicación de la aplicación es una página independiente con la que ASP.NET responde. A medida que se solicitan páginas, el contenido del explorador se reemplaza por los resultados de la página solicitada.

Las páginas constan de los siguientes componentes:

- Marcado HTML
- Código de C# o Visual Basic
- Una clase de código subyacente que contiene funcionalidades de lógica y control de eventos
- Controles

Los controles son unidades reutilizables de la interfaz de usuario web que se pueden colocar e interactuar mediante programación con en una página. Las páginas se componen de archivos que terminan con `.aspx` que contienen marcado, controles y código. Las clases de código subyacente están en archivos con el mismo nombre base y una extensión `.aspx.cs` o `.aspx.vb`, según el lenguaje de programación usado. Interesantemente, el servidor web interpreta el contenido de los archivos `.aspx` y los compila cada vez que cambian. Esta recompilación se produce incluso si el servidor web ya se está ejecutando.

Los controles se pueden compilar con marcado y entregarse como controles de usuario. Un control de usuario se deriva de la `UserControl` clase y tiene una estructura similar a page. El marcado de los controles de usuario se almacena en un archivo `.ascx`. Una clase de código subyacente que acompaña reside en un archivo `.ascx.cs` o `.ascx.vb`. Los controles también se pueden construir completamente con código, heredando de la clase base `WebControl` o `CompositeControl`.

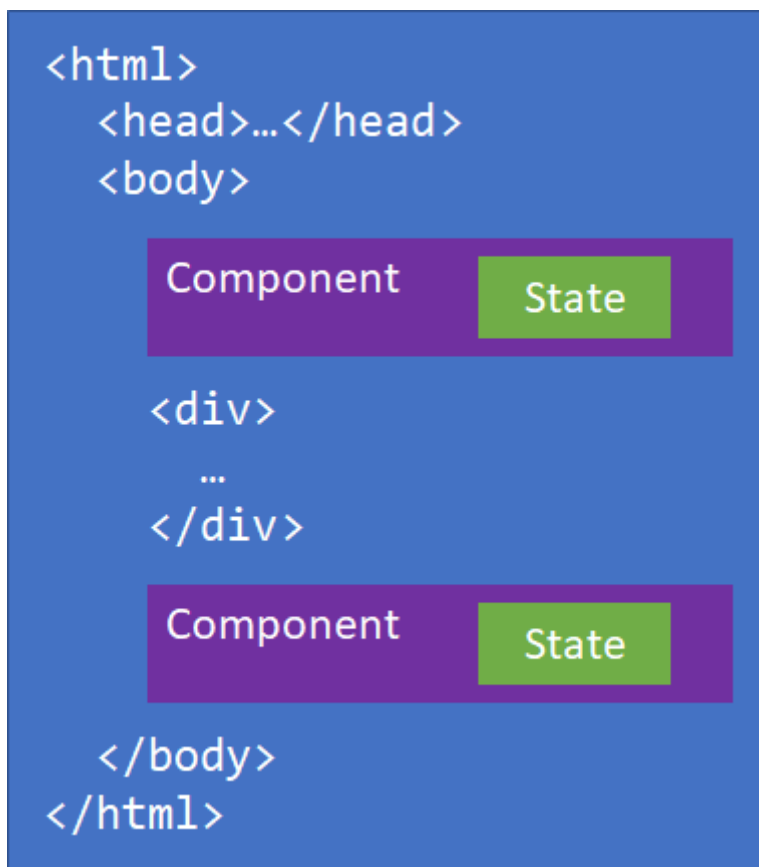
Las páginas también tienen un amplio ciclo de vida de eventos. Cada página genera eventos para los eventos de inicialización, carga, representación previa y descarga que se producen a medida que el tiempo de ejecución de ASP.NET ejecuta el código de la página para cada solicitud.

Normalmente, los controles de una página se reenvían a la misma página que ha presentado el control y llevan una carga de un campo de formulario oculto denominado `ViewState`. El `ViewState` campo contiene información sobre el estado de los controles en el momento en que se representaron y presentaron en la página, lo que permite que el entorno de ejecución de ASP.NET compare e identifique los cambios en el contenido enviado al servidor.

Blazor

Blazor es un marco de interfaz de usuario web del lado cliente similar a los marcos de front-end de JavaScript, como Angular o React. Blazor controla las interacciones del usuario y representa las actualizaciones de interfaz de usuario necesarias. Blazor *no* se basa en un modelo de solicitud-respuesta. Las interacciones del usuario se controlan como eventos que no están en el contexto de ninguna solicitud HTTP determinada.

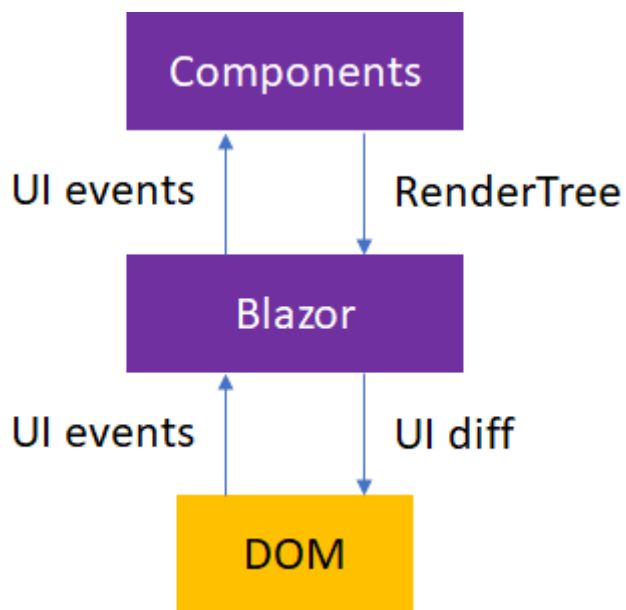
Blazor las aplicaciones constan de uno o varios componentes raíz que se representan en una página HTML.



Cómo el usuario especifica dónde se deben representar los componentes y cómo se conectan los componentes para las interacciones del usuario es específico del [modelo de hospedaje](#) .

Blazor [los componentes](#) son clases de .NET que representan una parte reutilizable de la interfaz de usuario. Cada componente mantiene su propio estado y especifica su propia lógica de representación, que puede incluir la representación de otros componentes. Los componentes especifican controladores de eventos para interacciones de usuario específicas para actualizar el estado del componente.

Después de que un componente controle un evento, Blazor representa el componente y realiza un seguimiento de lo que ha cambiado en la salida representada. Los componentes no se representan directamente en el Modelo de objetos de documento (DOM). En su lugar, se representan en una representación en memoria del DOM denominada `RenderTree` para que Blazor pueda seguir los cambios. Blazor compara la salida recién representada con la salida anterior para calcular una diferencia de interfaz de usuario que, a continuación, se aplica eficazmente al DOM.



Los componentes también pueden indicar manualmente que deben representarse si su estado cambia fuera de un evento de interfaz de usuario normal. Blazor usa un elemento `SynchronizationContext` para aplicar un único subproceso lógico de ejecución. Los métodos de ciclo de vida de un componente y todas las devoluciones de llamada de eventos que Blazor genere se ejecutan en dicho elemento `SynchronizationContext`.

[Siguiendo anterior](#)

Blazor modelos de hospedaje de aplicaciones

19/06/2025

💡 Sugerencia

Este contenido es un extracto del libro electrónico "Blazor for ASP.NET Web Forms Developers for Azure" (Blazor para desarrolladores de ASP.NET Web Forms), disponible en [Documentación de .NET](#) o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF



Blazor Las aplicaciones se pueden hospedar de una de las maneras siguientes:

- En el lado cliente en el explorador en WebAssembly.
- En el lado servidor en una aplicación ASP.NET Core.

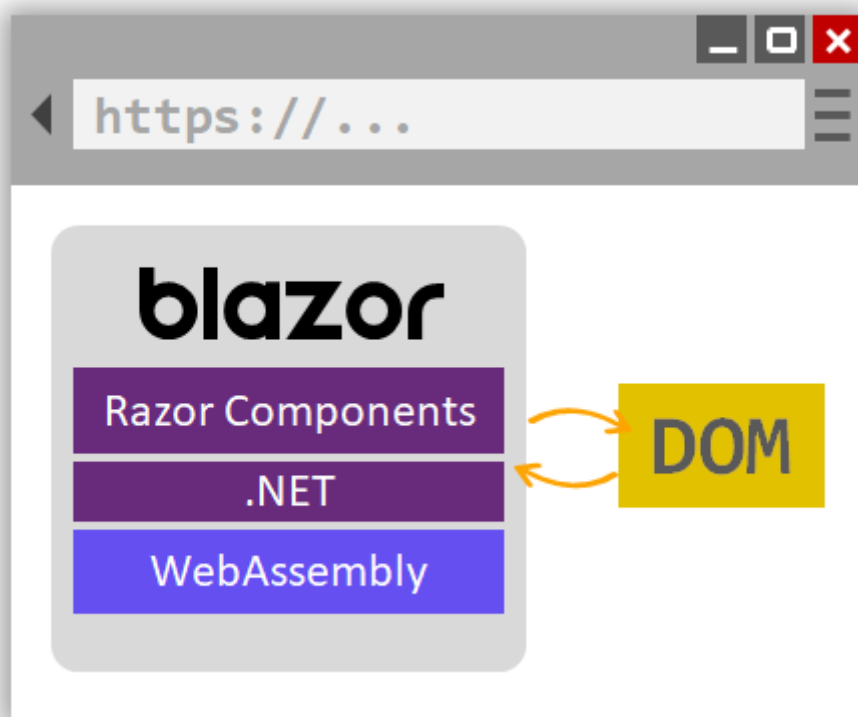
Blazor WebAssembly Aplicaciones

Blazor WebAssembly las aplicaciones se ejecutan directamente en el navegador en un entorno de ejecución basado en .NET. Blazor WebAssembly las aplicaciones funcionan de forma similar a los marcos de JavaScript front-end como Angular o React. Sin embargo, en lugar de escribir JavaScript, escriba C#. El entorno de ejecución de .NET se descarga con la aplicación junto con el ensamblado de la aplicación y las dependencias necesarias. No se requieren complementos o extensiones del explorador.

Los ensamblados descargados son ensamblados de .NET normales, como usaría en cualquier otra aplicación de .NET. Dado que el entorno de ejecución admite .NET Standard, puede usar las bibliotecas de .NET Standard existentes con la BlazorWebAssembly aplicación. Sin embargo, estos ensamblados se seguirán ejecutando en el espacio aislado de seguridad del explorador.

Algunas funcionalidades pueden producir [PlatformNotSupportedException](#), como intentar acceder al sistema de archivos o abrir conexiones de red arbitrarias.

Cuando se carga la aplicación, se inicia el entorno de ejecución de .NET y se apunta al ensamblado de la aplicación. La lógica de inicio de la aplicación se ejecuta y se representan los componentes raíz. Blazor calcula las actualizaciones de la interfaz de usuario en función de la salida representada de los componentes. A continuación, se aplican las actualizaciones del DOM.



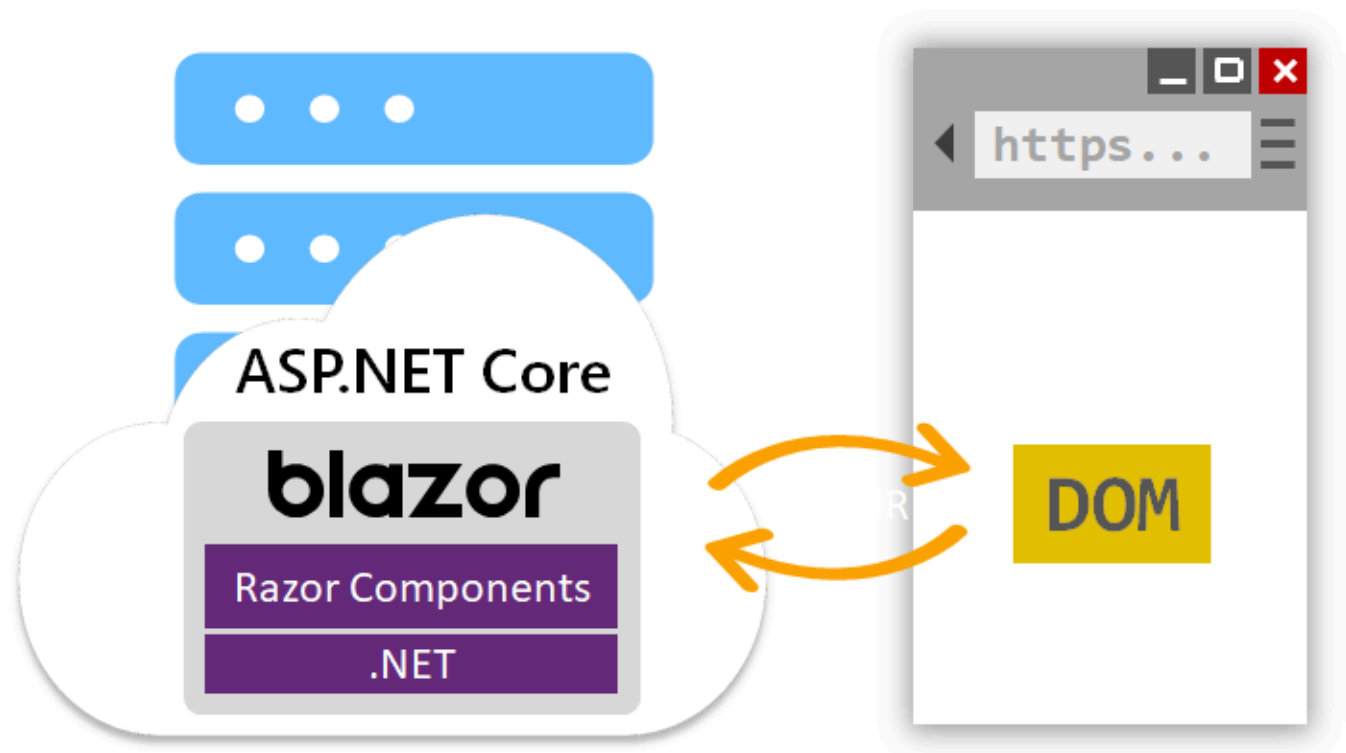
Blazor WebAssembly las aplicaciones se ejecutan exclusivamente en el lado cliente. Estas aplicaciones se pueden implementar en soluciones de hospedaje de sitios estáticos, como GitHub Pages o Hospedaje de sitios web estáticos de Azure. .NET no es necesario en el servidor. La vinculación profunda a partes de la aplicación normalmente requiere una solución de enrutamiento en el servidor. La solución de enrutamiento redirige las solicitudes a la raíz de la aplicación. Por ejemplo, este redireccionamiento se puede controlar mediante reglas de reescritura de direcciones URL en IIS.

Para obtener todos los beneficios de Blazor y del desarrollo web completo de .NET, hospede su aplicación BlazorWebAssembly con ASP.NET Core. Mediante el uso de .NET tanto en el cliente como en el servidor, puede compartir fácilmente código y compilar la aplicación mediante un conjunto coherente de lenguajes, marcos y herramientas. Blazor proporciona plantillas cómodas para configurar una solución que contiene una BlazorWebAssembly aplicación y un proyecto host de ASP.NET Core. Cuando se compila la solución, los archivos estáticos compilados de la aplicación Blazor se hospedan en la aplicación de ASP.NET Core con el enrutamiento de reserva ya configurado.

Blazor Aplicaciones de servidor

Recordará de la descripción de la arquitectura de [Blazor](#) que los componentes de Blazor representan su salida en una abstracción intermedia denominada `RenderTree`. A continuación, el Blazor marco compara lo que se ha representado con lo que se ha representado anteriormente. Las diferencias se aplican al DOM. Blazor Los componentes se desacoplan de cómo se aplica su salida representada. Por lo tanto, los propios componentes no tienen que ejecutarse en el mismo proceso que el proceso que actualiza la interfaz de usuario. De hecho, ni siquiera se tienen que ejecutar en la misma máquina.

En Blazor Aplicaciones de servidor, los componentes se ejecutan en el servidor en lugar del lado cliente en el explorador. Los eventos de interfaz de usuario que se producen en el explorador se envían al servidor a través de una conexión en tiempo real. Los eventos se envían a las instancias de componente correctas. Los componentes se representan y la diferencia de interfaz de usuario calculada se serializa y se envía al explorador donde se aplica al DOM.



El modelo de hospedaje del servidor puede resultarle familiar si ha utilizado ASP.NET AJAX y el control Blazor. El `UpdatePanel` control controla la aplicación de actualizaciones parciales de página en respuesta a los eventos desencadenadores en la página. Cuando se desencadena, solicita `UpdatePanel` una actualización parcial y, a continuación, la aplica sin necesidad de actualizar la página. El estado de la interfaz de usuario se administra mediante `ViewState`. Blazor Las aplicaciones de servidor son ligeramente diferentes en que la aplicación requiere una conexión activa con el cliente. Además, se mantiene todo el estado de la interfaz de

usuario en el servidor. Aparte de esas diferencias, los dos modelos son conceptualmente similares.

Cómo elegir el modelo de hospedaje adecuado Blazor

Como se describe en la documentación del modelo de hospedaje de [Blazor](#), los distintos modelos de hospedaje de Blazor tienen diferentes inconvenientes.

El BlazorWebAssembly modelo de hospedaje tiene las siguientes ventajas:

- No hay ninguna dependencia del lado servidor de .NET. La aplicación funciona completamente después de descargarse en el cliente.
- Los recursos y funcionalidades de cliente se aprovechan completamente.
- El trabajo se descarga del servidor al cliente.
- No es necesario que un servidor web ASP.NET Core hospede la aplicación. Los escenarios de implementación sin servidor son posibles (por ejemplo, servir la aplicación desde una red CDN).

Las desventajas del BlazorWebAssembly modelo de hospedaje son:

- Las funcionalidades del explorador restringen la aplicación.
- Se necesita hardware y software cliente compatible (por ejemplo, con WebAssembly).
- El tamaño de descarga es mayor y las aplicaciones tardan más tiempo en cargarse.
- El entorno de ejecución y las herramientas de .NET son menos maduros. Por ejemplo, existen limitaciones en la compatibilidad y la depuración de [.NET Standard](#).

Por el contrario, el Blazor modelo de hospedaje del servidor ofrece las siguientes ventajas:

- El tamaño de descarga es mucho menor que una aplicación del lado cliente y la aplicación se carga mucho más rápido.
- La aplicación aprovecha al máximo las funcionalidades del servidor, incluido el uso de cualquier API compatible con .NET.
- .NET en el servidor se usa para ejecutar la aplicación, por lo que las herramientas de .NET existentes, como la depuración, funcionan según lo previsto.
- Se admiten clientes ligeros. Por ejemplo, las aplicaciones del lado servidor funcionan con exploradores que no admiten WebAssembly y en dispositivos con restricciones de recursos.
- La base de código .NET/C# de la aplicación, incluido el código de componente de la aplicación, no se proporciona a los clientes.

Las desventajas del modelo de hospedaje del Blazor servidor son:

- Mayor latencia de la interfaz de usuario. Cada interacción del usuario implica un salto de red.
- No hay soporte técnico sin conexión. Si se produce un error en la conexión de cliente, la aplicación deja de funcionar.
- La escalabilidad es difícil para las aplicaciones con muchos usuarios. El servidor debe administrar varias conexiones de cliente y controlar el estado del cliente.
- Se requiere un servidor ASP.NET Core para atender la aplicación. Los escenarios de implementación sin servidor no son posibles. Por ejemplo, no se puede servir la aplicación desde una red CDN.

La lista anterior de desventajas puede ser intimidante, pero el modelo de hospedaje se puede cambiar más adelante. Independientemente del Blazor modelo de hospedaje seleccionado, el modelo de componentes es *el mismo*. En principio, se pueden usar los mismos componentes con cualquier modelo de hospedaje. El código de tu aplicación no cambia; sin embargo, es una buena práctica introducir abstracciones para que los componentes permanezcan independientes del modelo de hospedaje. Las abstracciones permiten a la aplicación adoptar más fácilmente un modelo de hospedaje diferente.

Implementación de la aplicación

Las aplicaciones de formularios Web Forms de ASP.NET normalmente se hospedan en IIS en un clúster o equipo Windows Server. Blazor las aplicaciones también pueden:

- Se hospeda en IIS, ya sea como archivos estáticos o como una aplicación de ASP.NET Core.
- Aproveche la flexibilidad de ASP.NET Core para hospedarse en varias plataformas e infraestructuras de servidor. Por ejemplo, puede hospedar una Blazor aplicación mediante [Nginx](#) o [Apache](#) en Linux. Para obtener más información sobre cómo publicar e implementar Blazor aplicaciones, consulte la Blazor documentación [de hospedaje e implementación](#).

En la sección siguiente, veremos cómo se configuran los proyectos para las aplicaciones de servidor Blazor, WebAssembly y Blazor.

Siguiente anterior

Estructura de proyecto de aplicaciones Blazor

Artículo • 08/06/2023

💡 Sugerencia

Este contenido es un extracto del libro electrónico "Blazor for ASP.NET Web Forms Developers for Azure" (Blazor para desarrolladores de ASP.NET Web Forms), disponible en [Documentación de .NET](#) o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF



A pesar de las diferencias importantes en las estructuras de proyecto, ASP.NET Web Forms y Blazor comparten muchos conceptos similares. Aquí veremos la estructura de un proyecto Blazor y lo comparamos con otro de ASP.NET Web Forms.

Para crear su primera aplicación de Blazor, siga las instrucciones que se indican en los [pasos de introducción de Blazor](#). Puede seguir las instrucciones para crear una aplicación Blazor Server o una BlazorWebAssembly hospedada en ASP.NET Core. A excepción de la lógica específica del modelo de hospedaje, la mayor parte del código en ambos proyectos es el mismo.

Archivo del proyecto

Las aplicaciones Blazor Server son proyectos de .NET. El archivo del proyecto para la aplicación Blazor Server es muy sencilla:

XML

```
<Project Sdk="Microsoft.NET.Sdk.Web">
```

```

    <PropertyGroup>
      <TargetFramework>net7.0</TargetFramework>
    </PropertyGroup>

  </Project>

```

El archivo del proyecto de una aplicación BlazorWebAssembly tiene un aspecto ligeramente más complicado (los números de versión exactos pueden variar):

XML

```

<Project Sdk="Microsoft.NET.Sdk.BlazorWebAssembly">

  <PropertyGroup>
    <TargetFramework>net7.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.Components.WebAssembly"
Version="7.0.0" />
    <PackageReference
Include="Microsoft.AspNetCore.Components.WebAssembly.DevServer"
Version="7.0.0" PrivateAssets="all" />
  </ItemGroup>

</Project>

```

BlazorWebAssemblyEl proyecto tiene como destino

Microsoft.NET.Sdk.BlazorWebAssembly, en lugar del SDK de Microsoft.NET.Sdk.Web, porque se ejecutan en el explorador, en un entorno de ejecución .NET basado en WebAssembly. No se puede instalar .NET en un explorador web, tal como se podría hacer en un servidor o una máquina del desarrollador. Por consiguiente, el proyecto hace referencia al marco Blazor mediante referencias de paquete individuales.

En cambio, un proyecto de ASP.NET Web Forms predeterminado incluye casi 300 líneas de XML en su archivo *.csproj*, la mayoría de las cuales están enumerando explícitamente los distintos archivos de código y contenido en el proyecto. Desde la publicación de .NET 5, las aplicaciones de Blazor Server y BlazorWebAssembly pueden compartir fácilmente un entorno de ejecución unificado.

Aunque están admitidas, las referencias de ensamblado individuales son menos comunes en los proyectos de .NET. La mayoría de las dependencias del proyecto se controlan como referencias de paquetes NuGet. Solo es necesario hacer referencia a las dependencias de paquetes de nivel superior en los proyectos de .NET. Las dependencias transitivas se incluyen automáticamente. En lugar de usar el archivo *packages.config*, que

se encuentra normalmente en los proyectos de ASP.NET Web Forms para hacer referencia a los paquetes, las referencias de paquete se agregan al archivo del proyecto mediante el elemento `<PackageReference>`.

XML

```
<ItemGroup>
  <PackageReference Include="Newtonsoft.Json" Version="13.0.2" />
</ItemGroup>
```

Punto de entrada

El punto de entrada de la aplicación Blazor Server se define en el archivo *Program.cs*, tal como se verá en una aplicación de la consola. Cuando se ejecuta la aplicación, crea y ejecuta una instancia de host web con los valores predeterminados específicos de las aplicaciones web. El host web administra el ciclo de vida de la aplicación Blazor Server y configura los servicios de nivel de host. Algunos ejemplos de estos servicios son la configuración, el registro, la inserción de dependencias y el servidor HTTP. Este código es principalmente reutilizable y a menudo se deja sin cambios.

C#

```
using BlazorApp3.Areas.Identity;
using BlazorApp3.Data;
using Microsoft.AspNetCore.Components.Authorization;
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
var connectionString =
builder.Configuration.GetConnectionString("DefaultConnection");
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(connectionString));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();
builder.Services.AddDefaultIdentity<IdentityUser>(options =>
options.SignIn.RequireConfirmedAccount = true)
    .AddEntityFrameworkStores<ApplicationDbContext>();
builder.Services.AddRazorPages();
builder.Services.AddServerSideBlazor();
builder.Services.AddScoped<AuthenticationStateProvider,
RevalidatingIdentityAuthenticationStateProvider<IdentityUser>>();
builder.Services.AddSingleton<WeatherForecastService>();

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.UseAuthorization();

app.MapControllers();
app.MapBlazorHub();
app.MapFallbackToPage("/_Host");

app.Run();

```

BlazorWebAssembly Las aplicaciones también definen un punto de entrada en *Program.cs*. El código tiene un aspecto ligeramente distinto. El código es similar en que está configurando el host de la aplicación para proporcionar los mismos servicios de nivel de host a la aplicación. Sin embargo, el host de la aplicación WebAssembly no configura un servidor HTTP porque se ejecuta directamente en el explorador.

Las aplicaciones Blazor no usan un archivo *Global.asax* para definir la lógica de inicio de la aplicación. En su lugar, esta lógica se incluye en *Program.cs* o en una clase `Startup` relacionada a la que se hace referencia desde *Program.cs*. En cualquier caso, este código se usa para configurar la aplicación y los servicios específicos de la aplicación.

En una aplicación Blazor Server, el archivo *Program.cs* mostrado se usa para configurar el punto de conexión para la conexión en tiempo real que utiliza Blazor entre los exploradores y el servidor del cliente.

En una aplicación BlazorWebAssembly, el archivo *Program.cs* define los componentes raíz de la aplicación y dónde se deben representar:

C#

```

using BlazorApp1;
using Microsoft.AspNetCore.Components.Web;
using Microsoft.AspNetCore.Components.WebAssembly.Hosting;

```

```
var builder = WebAssemblyHostBuilder.CreateDefault(args);
builder.RootComponents.Add<App>("#app");
builder.RootComponents.Add<HeadOutlet>("head::after");

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

await builder.Build().RunAsync();
```

Archivos estáticos

A diferencia de los proyectos de ASP.NET Web Forms, no todos los archivos de un proyecto de Blazor se pueden solicitar como archivos estáticos. Solo los archivos de la carpeta *wwwroot* son direccionables por web. Esta carpeta se conoce como "raíz web" de la aplicación. Todo lo demás que esté fuera de la raíz web de la aplicación *no es* direccionable por web. Esta configuración proporciona un nivel de seguridad adicional que evita la exposición accidental de archivos del proyecto a través de la web.

Configuración

Por lo general, la configuración de las aplicaciones de ASP.NET Web Forms se controla mediante uno o varios archivos *web.config*. Normalmente, las aplicaciones Blazor no tienen archivos *web.config*. Si lo hacen, el archivo solo se utiliza para configurar los valores específicos de IIS cuando se hospedan en este software. En su lugar, las aplicaciones Blazor Server usan las abstracciones de configuración de ASP.NET Core (actualmente, las aplicaciones BlazorWebAssembly no admiten las mismas abstracciones de configuración, pero es posible que dicha característica se agregue en el futuro). Por ejemplo, la aplicación Blazor Server predeterminada almacena algunos valores en *appsettings.json*.

JSON

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*"
}
```

En la sección [Configuración](#), encontrará más información sobre la configuración en proyectos de ASP.NET Core.

Componentes de Razor

La mayoría de los archivos de los proyectos de Blazor son archivos *.razor*. Razor es un lenguaje de plantillas basado en HTML y C# que se usa para generar dinámicamente la interfaz de usuario web. Los archivos *.razor* definen los componentes que componen la interfaz de usuario de la aplicación. En su mayor parte, los componentes son idénticos tanto para las aplicaciones Blazor Server como para las aplicaciones BlazorWebAssembly. Los componentes de Blazor son análogos a los controles de usuario de ASP.NET Web Forms.

Cada archivo de componente de Razor se compila en una clase .NET cuando se compila el proyecto. La clase generada captura el estado del componente, la lógica de representación, los métodos de ciclo de vida, los controladores de eventos y otra lógica. En la sección [Creación de componentes de interfaz de usuario reutilizables con Blazor](#), obtendrá más información sobre los componentes de creación.

Los archivos *_Imports.razor* no son archivos de componentes de Razor. En su lugar, definen un conjunto de directivas de Razor que se van a importar en otros archivos *.razor*, dentro de la misma carpeta y en sus subcarpetas. Por ejemplo, un archivo *_Imports.razor* es una manera convencional de agregar directivas `using` para los espacios de nombres usados habitualmente:

razor

```
@using System.Net.Http
@using System.Net.Http.Json
@using Microsoft.AspNetCore.Components.Forms
@using Microsoft.AspNetCore.Components.Routing
@using Microsoft.AspNetCore.Components.Web
@using Microsoft.AspNetCore.Components.Web.Virtualization
@using Microsoft.AspNetCore.Components.WebAssembly.Http
@using Microsoft.JSInterop
@using BlazorApp1
@using BlazorApp1.Shared
```

Páginas

¿Dónde están las páginas de las aplicaciones Blazor? Blazor no define una extensión de archivo independiente para las páginas direccionables, como los archivos *.aspx* de las aplicaciones de ASP.NET Web Forms. En su lugar, las páginas se definen mediante la

asignación de rutas a los componentes. Normalmente, una ruta se asigna mediante la directiva de Razor `@page`. Por ejemplo, el componente `Counter` creado en el archivo `Pages/Counter.razor` define la ruta siguiente:

```
razor
```

```
@page "/counter"
```

El enrutamiento en Blazor se controla en el lado cliente, no en el servidor. A medida que el usuario navega en el explorador, Blazor intercepta la navegación y, después, representa el componente con la ruta coincidente.

Las rutas del componente no las infiere actualmente la ubicación del archivo del componente, tal como sucede con las páginas `.aspx` o Razor Pages en ASP.NET Core. Es posible que en el futuro se agregue esta característica. Cada ruta debe especificarse explícitamente en el componente. El almacenamiento de componentes enrutables en una carpeta `Pages` no tiene ningún significado especial y no es más que una convención.

En la sección [Páginas, enrutamiento y diseños](#) encontrará más información sobre el enrutamiento en Blazor.

Layout

En las aplicaciones de ASP.NET Web Forms, un diseño de página común se controla mediante páginas maestras (*Site.Master*). En aplicaciones Blazor, el diseño de página se controla mediante componentes de diseño (*Shared/MainLayout.razor*). Los componentes de diseño se tratan con más detalle en la sección [Páginas, enrutamiento y diseños](#).

Arranque de Blazor

Para arrancar Blazor, la aplicación debe realizar lo siguiente:

- Especificar en qué lugar de la página se debe representar el componente raíz (*App.Razor*).
- Agregar el script de marco de Blazor correspondiente.

En la aplicación Blazor Server, la página host del componente raíz se define en el archivo `_Host.cshtml`. Este archivo define una página de Razor, no un componente. Razor Pages usa la sintaxis Razor para definir una página direccionable por el servidor, de forma muy parecida a una página `.aspx`.

razor

```
@page "/"
@namespace BlazorApp3.Pages
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@{
    Layout = "_Layout";
}

<component type="typeof(App)" render-mode="ServerPrerendered" />
```

El atributo `render-mode` se utiliza para definir dónde se debe representar un componente de nivel de raíz. La opción `RenderMode` indica la manera en que se debe representar el componente. En la tabla siguiente se describen las opciones de `RenderMode` admitidas.

Opción	Descripción
<code>RenderMode.Server</code>	Representado de forma interactiva una vez que se establece una conexión con el explorador.
<code>RenderMode.ServerPrerendered</code>	Representado previamente en primer lugar y, después, representado de forma interactiva.
<code>RenderMode.Static</code>	Representado como contenido estático.

El archivo `_Layout.cshtml` incluye el código HTML predeterminado para la aplicación y sus componentes.

razor

```
@using Microsoft.AspNetCore.Components.Web
@namespace BlazorApp3.Pages
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <base href="~/>
    <link rel="stylesheet" href="css/bootstrap/bootstrap.min.css" />
    <link href="css/site.css" rel="stylesheet" />
    <link href="BlazorApp3.styles.css" rel="stylesheet" />
    <component type="typeof(HeadOutlet)" render-mode="ServerPrerendered" />
</head>
<body>
    @RenderBody()

    <div id="blazor-error-ui">
```

```

        <environment include="Staging,Production">
            An error has occurred. This application may no longer respond
            until reloaded.
        </environment>
        <environment include="Development">
            An unhandled exception has occurred. See browser dev tools for
            details.
        </environment>
        <a href="" class="reload">Reload</a>
        <a class="dismiss">✕</a>
    </div>

    <script src="_framework/blazor.server.js"></script>
</body>
</html>

```

La referencia de script a `_framework/blazor.server.js` establece la conexión en tiempo real con el servidor y, después, trata todas las interacciones del usuario y las actualizaciones de la interfaz de usuario.

En la aplicación BlazorWebAssembly, la página host es un archivo HTML estático sencillo que se encuentra en `wwwroot/index.html`. El elemento `<div>` con el identificador denominado `app` se utiliza para indicar dónde se debe representar el componente raíz.

HTML

```

<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0,
    maximum-scale=1.0, user-scalable=no" />
    <title>BlazorApp1</title>
    <base href="/" />
    <link href="css/bootstrap/bootstrap.min.css" rel="stylesheet" />
    <link href="css/app.css" rel="stylesheet" />
    <link href="BlazorApp1.styles.css" rel="stylesheet" />
</head>

<body>
    <div id="app">Loading...</div>

    <div id="blazor-error-ui">
        An unhandled error has occurred.
        <a href="" class="reload">Reload</a>
        <a class="dismiss">✕</a>
    </div>
    <script src="_framework/blazor.webassembly.js"></script>
</body>

```

```
</html>
```

El componente raíz que se va a representar se especifica en el archivo de la aplicación *Program.cs* con la flexibilidad de registrar los servicios mediante la inserción de dependencias. Para obtener más información, vea [Inserción de dependencias de Blazor de ASP.NET Core](#).

```
C#
```

```
var builder = WebAssemblyHostBuilder.CreateDefault(args);  
builder.RootComponents.Add<App>("#app");  
builder.RootComponents.Add<HeadOutlet>("head::after");
```

Resultado de la compilación

Cuando se compila un proyecto Blazor, todos los componentes de Razor y los archivos de código se compilan en un único ensamblado. A diferencia de los proyectos de ASP.NET Web Forms, Blazor no admite la compilación en tiempo de ejecución de la lógica de la interfaz de usuario.

Ejecución de la aplicación con Recarga activa

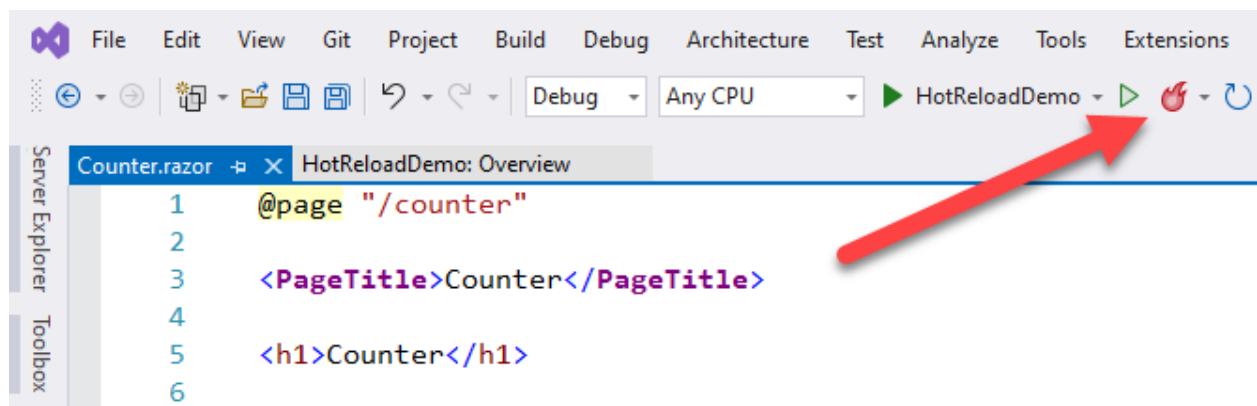
Para ejecutar la aplicación Blazor Server, presione **F5** en Visual Studio para ejecutarla con el depurador asociado, o **Ctrl** + **F5** para ejecutarla sin el depurador asociado.

Para ejecutar la aplicación BlazorWebAssembly, elija uno de los siguientes enfoques:

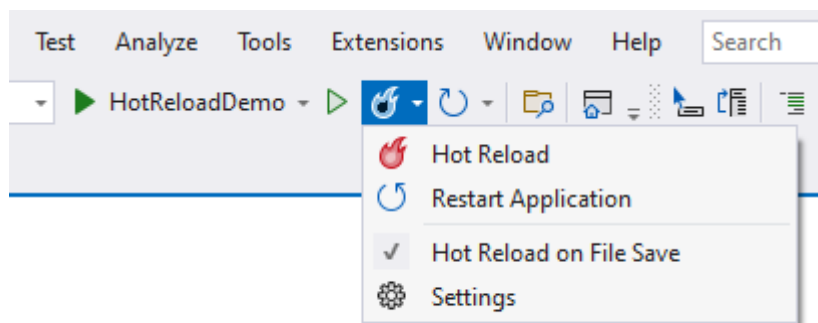
- Ejecutar el proyecto de cliente directamente con el servidor de desarrollo.
- Ejecutar el proyecto de servidor al hospedar la aplicación con ASP.NET Core.

Blazor Las aplicaciones de WebAssembly se pueden depurar tanto en el explorador como en Visual Studio. Para más información, consulte [Depuración de ASP.NET Core BlazorWebAssembly](#).

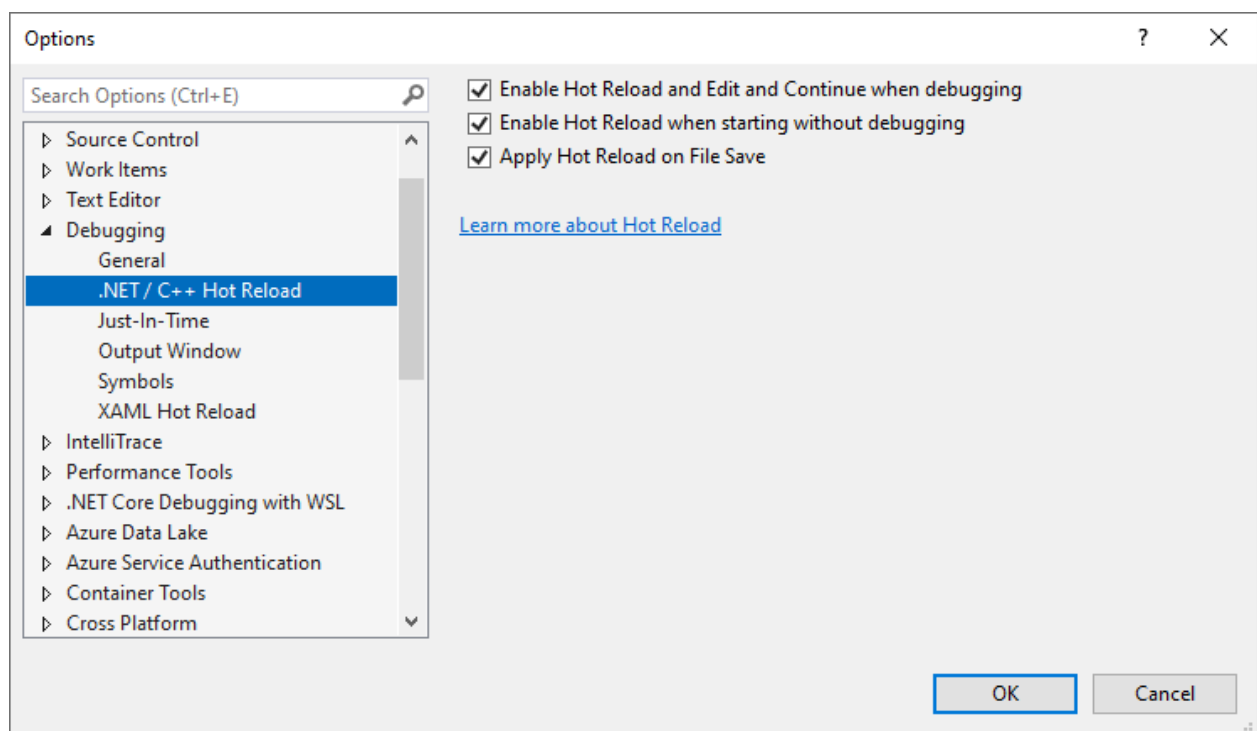
Tanto las aplicaciones Blazor Server como BlazorWebAssembly admiten Recarga activa en Visual Studio. Recarga activa es una característica que actualiza automáticamente los cambios realizados en una aplicación Blazor activa, en el explorador. Puede decidir si Recarga activa se habilita desde su icono en la barra de herramientas:



Al seleccionar el símbolo de intercalación situado junto al icono, se revelan opciones adicionales. Puede activar o desactivar Recarga activa, reiniciar la aplicación y decidir si Recarga activa debe producirse cada vez que se guarde un archivo.



También puede acceder a opciones de configuración adicionales. El cuadro de diálogo de configuración permite especificar si Recarga activa debe habilitarse al depurar (junto con Editar y continuar), al iniciar sin depurar o al guardarse un archivo.



El "bucle interno del desarrollador" se ha simplificado de manera considerable con Recarga activa. Sin Recarga activa, lo habitual sería que los desarrolladores de Blazor tuvieran que reiniciar y volver a ejecutar la aplicación después de cada cambio, y desplazarse a la parte adecuada de la aplicación según fuera necesario. Con Recarga activa, se pueden realizar cambios en la aplicación en ejecución sin necesidad de reiniciar en la mayoría de los casos. Recarga activa incluso conserva el estado de las páginas, por lo que no es necesario tener que volver a escribir los valores del formulario o, de otro modo, volver a obtener la aplicación donde la necesite.

[Anterior](#)[Siguiete](#)

Inicio de la aplicación

18/06/2025

💡 Sugerencia

Este contenido es un extracto del libro electrónico Blazor para desarrolladores de ASP.NET Web Forms para Azure, disponible en [documentos de .NET](#) o como un PDF descargable gratuito que se puede leer sin conexión.

Descargar PDF



Las aplicaciones escritas para ASP.NET normalmente tienen un `global.asax.cs` archivo que define el `Application_Start` evento que controla qué servicios están configurados y están disponibles para la representación HTML y el procesamiento de .NET. En este capítulo se examina cómo las cosas son ligeramente diferentes con ASP.NET Core y Blazor Server.

Application_Start y Web Forms

El método de formularios `Application_Start` web predeterminados ha crecido a lo largo de años para controlar muchas tareas de configuración. Un nuevo proyecto de formularios web forms con la plantilla predeterminada en Visual Studio 2022 ahora contiene la siguiente lógica de configuración:

- `RouteConfig` - Enrutamiento de direcciones URL de la aplicación
- `BundleConfig` - Agrupación y minificación de CSS y JavaScript

Cada uno de estos archivos individuales reside en la `App_Start` carpeta y se ejecuta solo una vez al principio de la aplicación. `RouteConfig` en la plantilla predeterminada del proyecto agrega el `FriendlyUrlSettings` en los formularios web, permitiendo que las direcciones URL de la aplicación omitan la extensión de archivo `.aspx`. La plantilla predeterminada también contiene una directiva que proporciona códigos de estado de redirección HTTP permanentes

(HTTP 301) para las `.ASPX` páginas a la URL amigable con el nombre de archivo que omite la extensión.

Con ASP.NET Core y Blazor, estos métodos se simplifican y consolidan en la `Startup` clase o se eliminan en favor de las tecnologías web comunes.

Estructura de inicio del servidor Blazor

Las aplicaciones blazor Server residen sobre una versión de ASP.NET Core 3.0 o posterior. aplicaciones web de ASP.NET Core se configuran en *Program.cs*, o a través de un par de métodos en la clase `Startup.cs`. A continuación se muestra un archivo *Program.cs* de ejemplo:

C#

```
using BlazorApp1.Data;
using Microsoft.AspNetCore.Components;
using Microsoft.AspNetCore.Components.Web;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddRazorPages();
builder.Services.AddServerSideBlazor();
builder.Services.AddSingleton<WeatherForecastService>();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
    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.MapBlazorHub();
app.MapFallbackToPage("/_Host");

app.Run();
```

Los servicios necesarios de la aplicación se agregan a la colección `WebApplicationBuilder` de la instancia `Services`. Así es como se configuran los distintos servicios de ASP.NET Core Framework con el contenedor integrado de inserción de dependencias del marco. Los distintos `builder.Services.Add*` métodos agregan servicios que habilitan características como la

autenticación, las páginas de razor, el enrutamiento del controlador MVC, SignalR y las interacciones del servidor Blazor entre muchas otras. Este método no era necesario en formularios web, ya que el análisis y control de los archivos ASPX, ASCX, ASHX y ASMX se definieron haciendo referencia a ASP.NET en el archivo de configuración de web.config. Puede encontrar más información sobre la inserción de dependencias en ASP.NET Core en la [documentación en línea](#).

Después de que el `app` haya sido construido por el `builder`, el resto de las llamadas en `app` configuran su canalización HTTP. Con estas llamadas, desde la parte superior a la inferior se declara el [middleware](#) que controlará todas las solicitudes enviadas a la aplicación. La mayoría de estas características de la configuración predeterminada se dispersaron entre los archivos de configuración de formularios web y ahora están en un solo lugar para facilitar la referencia.

Ya no se ubica la configuración de la página de error personalizada en un archivo `web.config`, sino que ahora se configura para mostrarse siempre si el entorno de la aplicación no está etiquetado `Development`. Además, las aplicaciones de ASP.NET Core ahora están configuradas para servir páginas seguras con TLS de forma predeterminada con la `UseHttpsRedirection` llamada al método .

A continuación, se realiza una llamada de método de configuración inesperada a `UseStaticFiles`. En ASP.NET Core, la compatibilidad con solicitudes de archivos estáticos (como JavaScript, CSS y archivos de imagen) debe habilitarse explícitamente y solo los archivos de la carpeta `wwwroot` de la aplicación se pueden direccionables públicamente de forma predeterminada.

La línea siguiente es la primera que replica una de las opciones de configuración de formularios web: `UseRouting`. Este método agrega el enrutador de ASP.NET Core a la canalización y se puede configurar aquí o en los archivos individuales que puede tener en cuenta para el enrutamiento. Puede encontrar más información sobre la configuración de enrutamiento en la [sección Enrutamiento](#).

Las llamadas finales `app.Map*` de esta sección definen los puntos de conexión en los que ASP.NET Core está monitorizando. Estas rutas son las ubicaciones accesibles desde la web a las que puede acceder en el servidor web y reciben contenido controlado por .NET y que se le devuelve. La primera entrada de `MapBlazorHub` configura un centro de SignalR para proporcionar la conexión en tiempo real y persistente al servidor, donde se controla el estado y la representación de los componentes de Blazor. La `MapFallbackToPage` llamada al método indica la ubicación accesible desde la web de la página que inicia la aplicación Blazor y también configura la aplicación para controlar las solicitudes de vinculación profunda desde el lado cliente. Verá esta característica en funcionamiento si abre un explorador y navega directamente a la ruta controlada de Blazor en la aplicación, como `/counter` en la plantilla de proyecto

predeterminada. La solicitud se gestiona mediante la página de respaldo `_Host.cshtml`, que a continuación ejecuta el enrutador Blazor y representa la página del contador.

La última línea inicia la aplicación, algo que no era necesario en los formularios web (ya que se basaba en IIS para ejecutarse).

Actualización del proceso BundleConfig

Las tecnologías para agrupar recursos como hojas de estilos CSS y archivos JavaScript han cambiado significativamente, con otras tecnologías que proporcionan herramientas y técnicas en constante evolución para administrar estos recursos. Para ello, se recomienda usar una herramienta de línea de comandos node como Grunt / Gulp / WebPack para empaquetar los recursos estáticos.

Las herramientas de línea de comandos Grunt, Gulp y WebPack y sus configuraciones asociadas se pueden agregar a la aplicación y ASP.NET Core omitirá silenciosamente esos archivos durante el proceso de compilación de la aplicación. Puede agregar una llamada para ejecutar sus tareas; para ello, incluya `Target` dentro del archivo del proyecto con una sintaxis similar a la siguiente que desencadenará un script de gulp y el destino `min` dentro de ese script.

XML

```
<Target Name="MyPreCompileTarget" BeforeTargets="Build">
  <Exec Command="gulp min" />
</Target>
```

Encontrará más detalles sobre ambas estrategias para administrar los archivos CSS y JavaScript en la documentación de [Agrupación y minimización de recursos estáticos en ASP.NET Core](#).

Siguiente anterior

Creación de componentes de interfaz de usuario reutilizables con Blazor

💡 Sugerencia

Este contenido es un extracto del libro electrónico "Blazor para desarrolladores de ASP.NET Web Forms para Azure", disponible en [Documentación de .NET](#), o como un PDF gratuito descargable que puede leerse sin conexión.

Descargar PDF



Uno de los atractivos de ASP.NET Web Forms es cómo permite la encapsulación de fragmentos reutilizables del código de la interfaz de usuario (IU) en controles de interfaz de usuario reutilizables. Los controles de usuario personalizados pueden definirse en el marcado utilizando archivos `.ascx`. También puede crear controles de servidor elaborados directamente en código, y disfrutar de total compatibilidad con el diseñador.

Blazor también admite la encapsulación de la interfaz de usuario por medio de *componentes*. Un componente:

- Es un fragmento independiente de la interfaz de usuario.
- Mantiene su propio estado y lógica de representación.
- Puede definir controladores de eventos de la interfaz de usuario, enlazarse a datos de entrada y administrar su propio ciclo de vida.
- Normalmente se define en un archivo `.razor` mediante sintaxis de Razor.

Introducción a Razor

Razor es un lenguaje de plantillas de marcado ligero basado en HTML y C#. Con Razor, puede realizar la transición sin problemas entre el marcado y el código de C# para definir la lógica de representación de los componentes. Cuando se compila el archivo `.razor`, la lógica de representación se captura de forma estructurada en una clase de .NET. El nombre de la clase

compilada se toma del nombre del archivo `.razor`. El espacio de nombres se deriva del espacio de nombres predeterminado del proyecto y de la ruta de la carpeta, o se puede especificar explícitamente mediante la directiva `@namespace` (a continuación, se proporcionará más información sobre las directivas de Razor).

La lógica de representación de un componente se crea mediante marcado HTML normal con lógica dinámica agregada por medio de C#. El carácter `@` se usa para realizar la transición a C#. Normalmente, Razor es inteligente para detectar cuándo se ha vuelto a HTML. Por ejemplo, el componente siguiente representa una etiqueta `<p>` con la hora actual:

```
razor
<p>@DateTime.Now</p>
```

Para especificar de forma explícita el principio y el final de una expresión de C#, use paréntesis:

```
razor
<p>@(DateTime.Now)</p>
```

Razor también facilita el uso del flujo de control de C# en la lógica de representación. Por ejemplo, puede representar condicionalmente un código HTML similar al siguiente:

```
razor
@if (value % 2 == 0)
{
    <p>The value was even.</p>
}
```

O bien, puede generar una lista de elementos mediante un bucle `foreach` normal de C# como el siguiente:

```
razor
<ul>
@foreach (var item in items)
{
    <li>@item.Text</li>
}
</ul>
```

Las directivas de Razor, como las de ASP.NET Web Forms, controlan muchos aspectos de la compilación de un componente de Razor. Algunos ejemplos son estas partes del componente:

- Espacio de nombres
- Clase base
- Interfaces implementadas
- Parámetros genéricos
- Espacios de nombres importados
- Rutas

Las directivas de Razor comienzan con el carácter `@` y se suelen usar al principio de una nueva línea al principio del archivo. Por ejemplo, la directiva `@namespace` define el espacio de nombres del componente:

```
razor

@namespace MyComponentNamespace
```

En la tabla siguiente se resumen las distintas directivas de Razor que se usan en Blazor y sus equivalentes en ASP.NET Web Forms, si existen.

[Expandir tabla](#)

Directiva	Descripción	Ejemplo	Equivalente en Web Forms
<code>@attribute</code>	Agrega un atributo de nivel de clase al componente.	<code>@attribute [Authorize]</code>	Ninguno
<code>@code</code>	Agrega miembros de clase al componente.	<code>@code { ... }</code>	<code><script runat="server">...</script></code>
<code>@implements</code>	Implementa la interfaz especificada.	<code>@implements IDisposable</code>	Uso de código subyacente
<code>@inherits</code>	Hereda de la clase base especificada.	<code>@inherits MyComponentBase</code>	<code><%@ Control Inherits="MyUserControlBase" %></code>
<code>@inject</code>	Inserta un servicio en el componente.	<code>@inject IJSRuntime JS</code>	Ninguno
<code>@layout</code>	Especifica un componente de diseño para el componente.	<code>@layout MainLayout</code>	<code><%@ Page MasterPageFile="~/Site.Master" %></code>
<code>@namespace</code>	Establece el espacio de nombres para el componente.	<code>@namespace MyNamespace</code>	Ninguno

Directiva	Descripción	Ejemplo	Equivalente en Web Forms
@page	Especifica la ruta del componente.	@page "/product/{id}"	<%@ Page %>
@typeparam	Especifica un parámetro de tipo genérico para el componente.	@typeparam TItem	Uso de código subyacente
@using	Especifica un espacio de nombres que se va a incluir en el ámbito.	@using MyComponentNamespace	Adición de espacio de nombres en <i>web.config</i>

Los componentes de Razor también hacen un uso intensivo de *atributos de directiva* en los elementos para controlar diversos aspectos de cómo se compilan los componentes (control de eventos, enlace de datos, referencias de elementos y componentes, etc.). Todos los atributos de directiva siguen una sintaxis genérica común en la que los valores entre paréntesis son opcionales:

razor
@directive(-suffix(:name))("value")

En la tabla siguiente se resumen los distintos atributos para las directivas de Razor que se usan en Blazor.

[Expandir tabla](#)

Atributo	Descripción	Ejemplo
@attributes	Representa un diccionario de atributos.	<input @attributes="ExtraAttributes" />
@bind	Crea un enlace de datos bidireccional.	<input @bind="username" @bind:event="oninput" />
@on{event}	Agrega un controlador de eventos para el evento especificado.	<button @onclick="IncrementCount">Click me!</button>
@key	Especifica una clave que va a usar el algoritmo de comparación para conservar los elementos de una colección.	<DetailsEditor @key="person" Details="person.Details" />
@ref	Captura una referencia al componente o elemento HTML.	<MyDialog @ref="myDialog" />

Los distintos atributos de directiva utilizados por Blazor (@onclick, @bind, @ref, etc.) se describen en las secciones siguientes y en capítulos posteriores.

Muchas de las sintaxis que se usan en los archivos `.aspx` y `.ascx` tienen sintaxis paralelas en Razor. A continuación se muestra una comparación sencilla de las sintaxis para ASP.NET Web Forms y Razor.

[Expandir tabla](#)

Característica	Formularios Web	Sintaxis	Afeitadora	Sintaxis
Directivas	<code><%@ [directive] %></code>	<code><%@ Page %></code>	<code>@[directive]</code>	<code>@page</code>
Bloques de código	<code><% %></code>	<code><% int x = 123; %></code>	<code>@{ }</code>	<code>@{ int x = 123; }</code>
Expresiones (Con codificación HTML)	<code><%= %></code>	<code><%=DateTime.Now %></code>	Implícita: <code>@</code> Explícito: <code>@()</code>	<code>@DateTime.Now</code> <code>@(DateTime.Now)</code>
Comentarios	<code><!-- --%></code>	<code><!-- Commented -- %></code>	<code>@* *@</code>	<code>@* Commented *@</code>
Enlace de datos	<code><## %></code>	<code><## Bind("Name") %></code>	<code>@bind</code>	<code><input @bind="username" /></code>

Para agregar miembros a la clase de componente de Razor, use la directiva `@code`. Esta técnica es similar a usar un bloque `<script runat="server">...</script>` en una página o control de usuario de ASP.NET Web Forms.

```
razor

@code {
    int count = 0;

    void IncrementCount()
    {
        count++;
    }
}
```

Como Razor se basa en C#, se debe compilar desde un proyecto de C# (`.csproj`). Los archivos `.razor` no se pueden compilar desde un proyecto de Visual Basic (`.vbproj`). Todavía puede hacer referencia a proyectos de Visual Basic desde el proyecto Blazor. Lo contrario también es cierto.

Para obtener una referencia completa de la sintaxis de Razor, vea [Referencia de sintaxis de Razor para ASP.NET Core](#).

Uso de componentes

Además de código HTML normal, los componentes también pueden usar otros componentes como parte de su lógica de representación. La sintaxis para utilizar un componente en Razor es similar al uso de un control de usuario en una aplicación de ASP.NET Web Forms. Los componentes se especifican mediante una etiqueta de elemento que coincide con el nombre de tipo del componente. Por ejemplo, puede agregar un componente `Counter` de esta forma:

razor

```
<Counter />
```

A diferencia de ASP.NET Web Forms, los componentes de Blazor:

- No usan un prefijo de elemento (por ejemplo, `asp:`).
- No es necesario registrarlos en la página ni en *web.config*.

Piense en los componentes de Razor como si fueran tipos de .NET, ya que son exactamente eso. Si se hace referencia al ensamblado que contiene el componente, el componente está disponible para su uso. Para poner el espacio de nombres del componente en el ámbito, aplique la directiva `@using`:

razor

```
@using MyComponentLib
```

```
<Counter />
```

Como se ve en los proyectos de Blazor predeterminados, es habitual colocar las directivas `@using` en un archivo *_Imports.razor* para que se importen en todos los archivos *.razor* en el mismo directorio y en los directorios secundarios.

Si el espacio de nombres de un componente no está en el ámbito de aplicación, puede especificar un componente con su nombre de tipo completo, como se puede en C#.

razor

```
<MyComponentLib.Counter />
```

Modificación del título de páginas a partir de componentes

Al compilar aplicaciones de estilo SPA, es habitual que las partes de una página se vuelvan a cargar sin volver a cargar toda la página. Aun así, puede ser útil tener el título de la página cambiado en función del componente que se carga actualmente. Para ello, incluya la etiqueta `<PageTitle>` en la página de Razor del componente:

razor

```
@page "/"
<PageTitle>Home</PageTitle>
```

El contenido de este elemento puede ser dinámico, por ejemplo, mostrando el recuento actual de mensajes:

razor

```
<PageTitle>@MessageCount messages</PageTitle>
```

Tenga en cuenta que si varios componentes de una página determinada incluyen etiquetas `<PageTitle>`, solo se mostrará el último (ya que cada uno sobrescribirá el anterior).

Parámetros del componente

En ASP.NET Web Forms, los parámetros y los datos se pueden enviar a los controles mediante propiedades públicas. Estas propiedades se pueden establecer en el marcado mediante atributos, o bien establecerse directamente en el código. Los componentes de Razor funcionan de manera similar, aunque las propiedades del componente también se deben marcar con el atributo `[Parameter]` para que se consideren parámetros de componente.

El componente `Counter` siguiente define un parámetro de componente denominado `IncrementAmount` que se puede usar para especificar la cantidad que se debe incrementar `Counter` cada vez que se haga clic en el botón.

razor

```
<h1>Counter</h1>

<p>Current count: @currentCount</p>

<button class="btn btn-primary" @onclick="IncrementCount">Click me</button>

@code {
    int currentCount = 0;

    [Parameter]
```

```
public int IncrementAmount { get; set; } = 1;

void IncrementCount()
{
    currentCount+=IncrementAmount;
}
}
```

Para especificar un parámetro de componente en Blazor, use un atributo como haría en ASP.NET Web Forms:

razor

```
<Counter IncrementAmount="10" />
```

Parámetros de cadena de consulta

Los componentes de Razor también pueden utilizar los valores de la cadena de consulta de la página donde se representan como fuente de parámetros. Para habilitar esto, agregue el atributo `[SupplyParameterFromQuery]` al parámetro. Por ejemplo, la siguiente definición de parámetro obtendría su valor de la solicitud con el formato `?IncBy=2`:

C#

```
[Parameter]
[SupplyParameterFromQuery(Name = "IncBy")]
public int IncrementAmount { get; set; } = 1;
```

Si no proporciona un elemento `Name` personalizado en el atributo `[SupplyParameterFromQuery]`, de forma predeterminada coincidirá con el nombre de la propiedad (`IncrementAmount` en este caso).

Componentes y límites de errores

De forma predeterminada, las aplicaciones de Blazor detectarán excepciones no controladas y mostrarán un mensaje de error en la parte inferior de la página sin detalles adicionales. Para restringir las partes de la aplicación que se ven afectadas por un error no controlado, por ejemplo, para limitar el impacto a un único componente, la etiqueta `<ErrorBoundary>` se puede encapsular alrededor de las declaraciones del componente.

Por ejemplo, para protegerse frente a posibles excepciones producidas por el componente `Counter`, declárelo dentro de `<ErrorBoundary>` y, opcionalmente, especifique un mensaje para mostrar si hay una excepción:

razor

```
<ErrorBoundary>
  <ChildContent>
    <Counter />
  </ChildContent>
  <ErrorContent>
    Oops! The counter isn't working right now; please try again later.
  </ErrorContent>
</ErrorBoundary>
```

Si no necesita especificar contenido de error personalizado, solo tiene que encapsular el componente directamente:

razor

```
<ErrorBoundary>
  <Counter />
</ErrorBoundary>
```

Se mostrará un mensaje predeterminado con el texto "Se ha producido un error." si se produce una excepción no controlada en el componente ajustado.

Controladores de eventos

Tanto en ASP.NET Web Forms como en Blazor se proporciona un modelo de programación basado en eventos para controlar los eventos de la interfaz de usuario. Entre los ejemplos de estos eventos se incluyen los clics de botón y la entrada de texto. En ASP.NET Web Forms, se usan controles de servidor HTML para controlar los eventos de la interfaz de usuario expuestos por el DOM, o bien se pueden controlar los expuestos por controles de servidor web. Los eventos se muestran en el servidor mediante solicitudes de reenvío de formulario. Considere el ejemplo siguiente de clic de botón de Web Forms:

Counter.ascx

ASP.NET (C#)

```
<asp:Button ID="ClickMeButton" runat="server" Text="Click me!"
OnClick="ClickMeButton_Click" />
```

Counter.ascx.cs

C#

```
public partial class Counter : System.Web.UI.UserControl
{
    protected void ClickMeButton_Click(object sender, EventArgs e)
    {
        Console.WriteLine("The button was clicked!");
    }
}
```

En Blazor, puede registrar controladores para eventos de interfaz de usuario del DOM directamente mediante atributos de directiva con el formato `@on{event}`. El marcador de posición `{event}` representa el nombre del evento. Por ejemplo, puede realizar escuchas de clics de botón de esta manera:

razor

```
<button @onclick="OnClick">Click me!</button>

@code {
    void OnClick()
    {
        Console.WriteLine("The button was clicked!");
    }
}
```

Los controladores de eventos pueden aceptar un argumento opcional específico del evento para proporcionar más información sobre el evento. Por ejemplo, los eventos del mouse pueden tomar un argumento `MouseEventArgs`, pero no es necesario.

razor

```
<button @onclick="OnClick">Click me!</button>

@code {
    void OnClick(MouseEventArgs e)
    {
        Console.WriteLine($"Mouse clicked at {e.ScreenX}, {e.ScreenY}.");
    }
}
```

En lugar de hacer referencia a un grupo de métodos para un controlador de eventos, puede usar una expresión lambda. Una expresión lambda permite cerrar otros valores en el ámbito.

razor

```
@foreach (var buttonLabel in buttonLabels)
{
    <button @onclick="() => Console.WriteLine($"The {buttonLabel} button was
```

```
clicked!")">@buttonLabel</button>
}
```

Los controladores de eventos se pueden ejecutar de forma sincrónica o asincrónica. Por ejemplo, el controlador de eventos `OnClick` siguiente se ejecuta de forma asincrónica:

razor

```
<button @onclick="OnClick">Click me!</button>

@code {
    async Task OnClick()
    {
        var result = await Http.GetAsync("api/values");
    }
}
```

Una vez que se ha controlado un evento, el componente se representará para tener en cuenta los cambios de estado del componente. Con los controladores de eventos asincrónicos, el componente se representa inmediatamente después de que se complete la ejecución del controlador. El componente se representará de *nuevo* después de que se complete el objeto `Task` asincrónico. Este modo de ejecución asincrónica proporciona una oportunidad para representar una interfaz de usuario adecuada mientras el objeto `Task` asincrónico todavía está en curso.

razor

```
<button @onclick="ShowMessage">Get message</button>

@if (showMessage)
{
    @if (message == null)
    {
        <p><em>Loading...</em></p>
    }
    else
    {
        <p>The message is: @message</p>
    }
}

@code
{
    bool showMessage = false;
    string message;

    public async Task ShowMessage()
    {
        showMessage = true;
    }
}
```

```
        message = await MessageService.GetMessageAsync();
    }
}
```

Los componentes también pueden definir eventos propios si definen un parámetro de componente de tipo `EventCallback<TValue>`. Los callbacks de evento admiten todas las variaciones de los controladores de eventos de UI del DOM: argumentos opcionales, sincrónicos o asincrónicos, grupos de métodos o expresiones lambda.

razor

```
<button class="btn btn-primary" @onclick="OnClick">Click me!</button>

@code {
    [Parameter]
    public EventCallback<MouseEventArgs> OnClick { get; set; }
}
```

Enlace de datos

Blazor proporciona un mecanismo sencillo para enlazar datos de un componente de interfaz de usuario al estado del componente. Este enfoque difiere de las características de ASP.NET Web Forms para enlazar datos desde orígenes de datos a controles de interfaz de usuario. En la sección [Tratamiento de los datos](#) se describirá el control de datos de orígenes de datos diferentes.

Para crear un enlace de datos bidireccional desde un componente de interfaz de usuario al estado del componente, use el atributo de directiva `@bind`. En el ejemplo siguiente, el valor de la casilla está enlazado al campo `isChecked`.

razor

```
<input type="checkbox" @bind="isChecked" />

@code {
    bool isChecked;
}
```

Cuando se representa el componente, el valor de la casilla se establece en el del campo `isChecked`. Cuando el usuario activa la casilla, se desencadena el evento `onchange` y el campo `isChecked` se establece en el nuevo valor. En este caso, la sintaxis `@bind` es equivalente al marcado siguiente:

razor

```
<input value="@isChecked" @onchange="(UIChangeEventArgs e) => isChecked = e.Value"
/>
```

Para cambiar el evento utilizado para el enlace, use el atributo `@bind:event`.

razor

```
<input @bind="text" @bind:event="oninput" />
<p>@text</p>

@code {
    string text;
}
```

Los componentes también pueden admitir el enlace de datos a sus parámetros. Para vincular datos, defina un parámetro de retorno de llamada de evento con el mismo nombre que el parámetro enlazable. El sufijo "Changed" se agrega al nombre.

PasswordBox.razor

razor

```
Password: <input
    value="@Password"
    @oninput="OnPasswordChanged"
    type="@ (showPassword ? "text" : "password")" />

<label><input type="checkbox" @bind="showPassword" />Show password</label>

@code {
    private bool showPassword;

    [Parameter]
    public string Password { get; set; }

    [Parameter]
    public EventCallback<string> PasswordChanged { get; set; }

    private Task OnPasswordChanged(ChangeEventArgs e)
    {
        Password = e.Value.ToString();
        return PasswordChanged.InvokeAsync(Password);
    }
}
```

Para encadenar un enlace de datos a un elemento de la interfaz de usuario subyacente, establezca el valor y controle el evento directamente en el elemento de la interfaz de usuario en lugar de usar el atributo `@bind`.

Para enlazar a un parámetro de componente, use un atributo `@bind-{Parameter}` para especificar el parámetro al que quiera enlazar.

razor

```
<PasswordBox @bind-Password="password" />

@code {
    string password;
}
```

Cambios de estado

Si el estado del componente ha cambiado fuera de un evento de interfaz de usuario normal o una devolución de llamada de evento, el componente debe indicar manualmente que se debe volver a representar. Para indicar que el estado de un componente ha cambiado, llame al método `StateHasChanged` en el componente.

En el ejemplo siguiente, un componente muestra un mensaje de un servicio `AppState` que otras partes de la aplicación pueden actualizar. El componente registra su método `StateHasChanged` con el evento `AppState.OnChange` para que se represente el componente cada vez que se actualice el mensaje.

C#

```
public class AppState
{
    public string Message { get; }

    // Lets components receive change notifications
    public event Action OnChange;

    public void UpdateMessage(string message)
    {
        Message = message;
        NotifyStateChanged();
    }

    private void NotifyStateChanged() => OnChange?.Invoke();
}
```

razor

```
@inject AppState AppState

<p>App message: @AppState.Message</p>
```

```
@code {
    protected override void OnInitialized()
    {
        AppState.OnChange += StateHasChanged
    }
}
```

Ciclo de vida del componente

El marco ASP.NET Web Forms tiene métodos de ciclo de vida bien definidos para módulos, páginas y controles. Por ejemplo, el control siguiente implementa controladores de eventos para los eventos de ciclo de vida `Init`, `Load` y `Unload`:

Counter.ascx.cs

C#

```
public partial class Counter : System.Web.UI.UserControl
{
    protected void Page_Init(object sender, EventArgs e) { ... }
    protected void Page_Load(object sender, EventArgs e) { ... }
    protected void Page_Unload(object sender, EventArgs e) { ... }
}
```

Los componentes de Razor también tienen un ciclo de vida bien definido. El ciclo de vida de un componente se puede usar para inicializar el estado del componente e implementar comportamientos de componentes avanzados.

Todos los métodos de ciclo de vida del componente Blazor tienen versiones sincrónicas y asincrónicas. La representación de componentes es sincrónica. No se puede ejecutar lógica asincrónica como parte de la representación de componentes. Toda la lógica asincrónica se debe ejecutar como parte de un método de ciclo de vida `async`.

OnInitialized

Los métodos `OnInitialized` y `OnInitializedAsync` se usan para inicializar el componente. Normalmente, un componente se inicializa después de que se represente por primera vez. Una vez que se ha inicializado un componente, se puede representar varias veces antes de eliminarlo. El método `OnInitialized` es similar al evento `Page_Load` en las páginas y controles de ASP.NET Web Forms.

C#

```
protected override void OnInitialized() { ... }
protected override async Task OnInitializedAsync() { await ... }
```

Al establecer parámetros

Los métodos `OnParametersSet` y `OnParametersSetAsync` se llaman cuando un componente ha recibido parámetros de su elemento primario y el valor se asigna a propiedades. Estos métodos se ejecutan después de la inicialización del componente y *cada vez que se representa*.

C#

```
protected override void OnParametersSet() { ... }
protected override async Task OnParametersSetAsync() { await ... }
```

OnAfterRender

Los métodos `OnAfterRender` y `OnAfterRenderAsync` se llaman después de que un componente haya terminado de representarse. En este punto se rellenan las referencias de elementos y componentes (a continuación obtendrá más información sobre estos conceptos). En este momento se habilita la interactividad con el explorador. Las interacciones con el DOM y la ejecución de JavaScript pueden tener lugar de forma segura.

C#

```
protected override void OnAfterRender(bool firstRender)
{
    if (firstRender)
    {
        ...
    }
}
protected override async Task OnAfterRenderAsync(bool firstRender)
{
    if (firstRender)
    {
        await ...
    }
}
```

No se llaman `OnAfterRender` y `OnAfterRenderAsync` durante la pre-renderización en el servidor.

El parámetro `firstRender` es `true` la primera vez que se representa el componente; de lo contrario, su valor es `false`.

IDisposable

Los componentes de Razor pueden implementar `IDisposable` para desechar recursos cuando el componente se elimina de la UI. Un componente de Razor puede implementar `IDisposable` mediante la directiva `@implements`:

razor

```
@using System
@implements IDisposable

...

@code {
    public void Dispose()
    {
        ...
    }
}
```

Captura de referencias a componentes

En ASP.NET Web Forms, es habitual manipular una instancia de control directamente en el código haciendo referencia a su identificador. En Blazor, también es posible capturar y manipular una referencia a un componente, aunque es mucho menos frecuente.

Para capturar una referencia de componente en Blazor, use el atributo de directiva `@ref`. El valor del atributo debe coincidir con el nombre de un campo configurable con el mismo tipo que el componente al que se hace referencia.

razor

```
<MyLoginDialog @ref="loginDialog" ... />

@code {
    MyLoginDialog loginDialog = default!;

    void OnSomething()
    {
        loginDialog.Show();
    }
}
```

Cuando se representa el componente primario, el campo se rellena con la instancia del componente secundario. Después, puede llamar a métodos en la instancia del componente o manipularla de otro modo.

No se recomienda manipular el estado del componente directamente mediante referencias de componentes. Al hacerlo, se impide que el componente se represente de forma automática en los momentos correctos.

Captura referencias a elementos

Los componentes de Razor pueden capturar referencias a un elemento. A diferencia de los controles de servidor HTML en ASP.NET Web Forms, no se puede manipular el DOM directamente mediante una referencia de elemento en Blazor. Blazor controla de forma automática la mayoría de las interacciones del DOM con su algoritmo de diferenciación del DOM. Las referencias de elemento capturadas en Blazor son opacas. Pero se usan para pasar una referencia de elemento específica en una llamada de interoperabilidad de JavaScript. Para obtener más información sobre la interoperabilidad de JavaScript, vea [Interoperabilidad de JavaScript para Blazor en ASP.NET Core](#).

Componentes con plantilla

En ASP.NET Web Forms, puede crear *controles basados en modelo*. Los controles basados en modelo permiten al desarrollador especificar una parte del código HTML que se usa para representar un control de contenedor. La mecánica de la creación de controles basados en modelo de servidor es compleja, pero permiten escenarios eficaces para representar los datos de forma personalizable. Entre los ejemplos de controles basados en modelo se incluyen `Repeater` y `DataList`.

Los componentes de Razor también se pueden templat al definir parámetros de componente de tipo `RenderFragment` o `RenderFragment<T>`. `RenderFragment` representa un fragmento de código Razor que será renderizado por el componente. `RenderFragment<T>` es un fragmento de marcado de Razor que acepta un parámetro, el cual puede especificarse cuando se renderiza el fragmento de renderizado.

Contenido hijo

Los componentes de Razor pueden capturar su contenido secundario como `RenderFragment` y representarlo como parte de la representación de componentes. Para capturar contenido secundario, defina un parámetro de componente de tipo `RenderFragment` y asígnele el nombre `ChildContent`.

ChildContentComponent.razor

```

<h1>Component with child content</h1>

<div>@ChildContent</div>

@code {
    [Parameter]
    public RenderFragment ChildContent { get; set; }
}

```

Un componente padre puede entonces proporcionar contenido hijo utilizando la sintaxis normal de Razor.

razor

```

<ChildContentComponent>
    <ChildContent>
        <p>The time is @DateTime.Now</p>
    </ChildContent>
</ChildContentComponent>

```

Parámetros de plantilla

Un componente de Razor basado en modelo también puede definir varios parámetros de componente de tipo `RenderFragment` o `RenderFragment<T>`. El parámetro de `RenderFragment<T>` se puede especificar durante la invocación. Para especificar un parámetro de tipo genérico para un componente, use la directiva `@typeparam` de Razor.

SimpleListView.razor

razor

```

@typeparam TItem

@Heading

<ul>
    @foreach (var item in Items)
    {
        <li>@ItemTemplate(item)</li>
    }
</ul>

@code {
    [Parameter]
    public RenderFragment Heading { get; set; }

    [Parameter]
    public RenderFragment<TItem> ItemTemplate { get; set; }
}

```

```
[Parameter]
public IEnumerable<TItem> Items { get; set; }
}
```

Al usar un componente con plantilla, los parámetros de plantilla se pueden especificar utilizando los elementos secundarios que coinciden con los nombres de los parámetros. Los argumentos de componente de tipo `RenderFragment<T>` pasados como elementos tienen un parámetro implícito denominado `context`. Puede cambiar el nombre de este parámetro de implementación mediante el atributo `Context` en el elemento secundario. Se puede especificar cualquier parámetro de tipo genérico mediante un atributo que coincida con el nombre del parámetro de tipo. Si es posible, el parámetro de tipo se infiere:

razor

```
<SimpleListView Items="messages" TItem="string">
  <Heading>
    <h1>My list</h1>
  </Heading>
  <ItemTemplate Context="message">
    <p>The message is: @message</p>
  </ItemTemplate>
</SimpleListView>
```

La salida de este componente tiene el siguiente aspecto:

HTML

```
<h1>My list</h1>
<ul>
  <li><p>The message is: message1</p></li>
  <li><p>The message is: message2</p></li>
</ul>
```

Código subyacente

Normalmente, un componente de Razor se crea en un solo archivo *.razor*. Pero también es posible separar el código y el marcado mediante un archivo de código subyacente. Para usar un archivo de código subyacente, agregue un archivo de C# que coincida con el nombre de archivo del archivo de componente, pero con una extensión *de .cs* agregada (*Counter.razor.cs*).

El enfoque preferido es declarar la clase de código subyacente como una `partial` clase con el mismo nombre que el componente. Dado que el archivo de componente de Razor ya genera

una clase parcial con el mismo nombre, las dos partes se combinan en tiempo de compilación, y no necesita la directiva `@inherits`.

Counter.razor

razor

```
<h1>Counter</h1>

<p>Current count: @currentCount</p>

<button @onclick="IncrementCount">Click me</button>
```

Counter.razor.cs

C#

```
public partial class Counter
{
    private int currentCount = 0;

    private void IncrementCount()
    {
        currentCount++;
    }
}
```

Con el enfoque de clase parcial, los miembros no necesitan ser `protected`. `private` Los miembros son accesibles porque ambas partes de la clase parcial se compilan en la misma clase.

Como alternativa, puede usar una clase base. Defina una clase base que derive de `ComponentBase` y agregue la `@inherits` directiva en el archivo de componente de Razor para especificar la clase base. Es habitual asignar un nombre a la clase base después del componente con un `Base` sufijo (`CounterBase`).

Counter.razor

razor

```
@inherits CounterBase

<h1>Counter</h1>

<p>Current count: @currentCount</p>

<button @onclick="IncrementCount">Click me</button>
```

C#

```
public class CounterBase : ComponentBase
{
    protected int currentCount = 0;

    protected void IncrementCount()
    {
        currentCount++;
    }
}
```

La visibilidad de los miembros de la clase base debe ser `protected` o `public` para que sean visibles para la clase de componente. Además, con el enfoque de clase base, el IDE puede mostrar tanto el componente como su clase base en las listas de IntelliSense. Use el enfoque de clase parcial para evitar esa duplicación.

Recursos adicionales

Las secciones anteriores no son un tratamiento exhaustivo de todos los aspectos de los componentes de Razor. Para obtener más información sobre cómo [Crear y usar componentes de Razor en ASP.NET Core](#), vea la documentación de Blazor.

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

📌 **Nota:** El autor ha creado este artículo con ayuda de inteligencia artificial. [Más información](#)

Last updated on 19/03/2026

Páginas, enrutamiento y diseños

18/06/2025

💡 Sugerencia

Este contenido es un extracto del libro electrónico "Blazor for ASP.NET Web Forms Developers for Azure" (Blazor para desarrolladores de ASP.NET Web Forms), disponible en [Documentación de .NET](#) o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF



ASP.NET aplicaciones de Web Forms se componen de páginas definidas en *archivos .aspx*. La dirección de cada página se basa en su ruta de acceso de archivo física en el proyecto. Cuando un explorador realiza una solicitud a la página, el contenido de la página se representa dinámicamente en el servidor. La representación considera tanto el marcado HTML de la página como los controles del servidor.

En Blazor, cada página de la aplicación es un componente, normalmente definido en un archivo *.razor*, con una o varias rutas especificadas. El enrutamiento se produce principalmente en el lado cliente sin implicar una solicitud de servidor específica. El explorador realiza primero una solicitud a la dirección raíz de la aplicación. A continuación, un componente raíz **Router** de la Blazor aplicación controla la interceptación de solicitudes de navegación y los reenvía al componente correcto.

Blazor también admite *enlaces profundos*. La vinculación profunda se produce cuando el explorador realiza una solicitud a una ruta específica distinta de la raíz de la aplicación. Las solicitudes de enlaces profundos enviadas al servidor se enrutan a la aplicación Blazor, que luego enruta la solicitud del lado del cliente al componente correcto.

Una página sencilla de ASP.NET Web Forms puede contener el marcado siguiente:

Name.aspx

ASP.NET (C#)

```
<%@ Page Title="Name" Language="C#" MasterPageFile="~/Site.Master"
AutoEventWireup="true" CodeBehind="Name.aspx.cs" Inherits="WebApplication1.Name"
%>

<asp:Content ID="BodyContent" ContentPlaceHolderID="MainContent" runat="server">
    <div>
        What is your name?<br />
        <asp:TextBox ID="TextBox1" runat="server"></asp:TextBox>
        <asp:Button ID="Button1" runat="server" Text="Submit"
OnClick="Button1_Click" />
    </div>
    <div>
        <asp:Literal ID="Literal1" runat="server" />
    </div>
</asp:Content>
```

Name.aspx.cs

C#

```
public partial class Name : System.Web.UI.Page
{
    protected void Button1_Click1(object sender, EventArgs e)
    {
        Literal1.Text = "Hello " + TextBox1.Text;
    }
}
```

La página equivalente de una Blazor aplicación tendría este aspecto:

Name.razor

razor

```
@page "/Name"
@layout MainLayout

<div>
    What is your name?<br />
    <input @bind="text" />
    <button @onclick="OnClick">Submit</button>
</div>
<div>
    @if (name != null)
    {
        @:Hello @name
    }
</div>
```

```
@code {
    string text;
    string name;

    void OnClick() {
        name = text;
    }
}
```

Crear páginas

Para crear una página en Blazor, cree un componente y agregue la `@page` directiva Razor para especificar la ruta del componente. La `@page` directiva toma un único parámetro, que es la plantilla de ruta que se va a agregar a ese componente.

razor

```
@page "/counter"
```

El parámetro de la plantilla de ruta es obligatorio. A diferencia de ASP.NET Web Forms, la ruta a un Blazor componente *no se deduce* de su ubicación en el archivo (aunque esa funcionalidad podría añadirse en el futuro).

La sintaxis de la plantilla de ruta es la misma sintaxis básica que se usa para el enrutamiento en ASP.NET Web Forms. Los parámetros de ruta se especifican en la plantilla mediante corchetes. Blazor enlazará los valores de ruta a los parámetros de componente con el mismo nombre (sin distinción de mayúsculas y minúsculas).

razor

```
@page "/product/{id}"

<h1>Product @Id</h1>

@code {
    [Parameter]
    public string Id { get; set; }
}
```

También puede especificar restricciones en el valor del parámetro route. Por ejemplo, para restringir el identificador de producto para que sea un `int`:

razor

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

<h1>Product @Id</h1>

@code {
    [Parameter]
    public int Id { get; set; }
}
```

Para obtener una lista completa de las restricciones de ruta admitidas por Blazor, consulte [Restricciones de ruta](#).

Componente de enrutador

El enrutamiento en Blazor está gestionado por el componente `Router`. El `Router` componente se usa normalmente en el componente raíz de la aplicación (*App.razor*).

razor

```
<Router AppAssembly="@typeof(Program).Assembly">
    <Found Context="routeData">
        <RouteView RouteData="@routeData" DefaultLayout="@typeof(MainLayout)" />
    </Found>
    <NotFound>
        <LayoutView Layout="@typeof(MainLayout)">
            <p>Sorry, there's nothing at this address.</p>
        </LayoutView>
    </NotFound>
</Router>
```

El `Router` componente descubre los componentes enrutables en el `AppAssembly` especificado y, opcionalmente, en el `AdditionalAssemblies` especificado. Cuando el explorador navega, `Router` intercepta la navegación y representa el contenido de su `Found` parámetro con el extraído `RouteData` si una ruta coincide con la dirección; de lo contrario `Router`, representa su `NotFound` parámetro.

El `RouteView` componente controla la representación del componente coincidente especificado por `RouteData` con su diseño si tiene uno. Si el componente coincidente no tiene un diseño, entonces se utiliza el `DefaultLayout` especificado opcionalmente.

El componente `LayoutView` representa su contenido secundario dentro del diseño especificado. Veremos los diseños con más detalle más adelante en este capítulo.

Navegación

En los formularios de ASP.NET Web Forms, se desencadena la navegación a otra página devolviendo una respuesta de redirección al explorador. Por ejemplo:

C#

```
protected void NavigateButton_Click(object sender, EventArgs e)
{
    Response.Redirect("Counter");
}
```

La devolución de una respuesta de redirección no suele ser posible en Blazor. Blazor no usa un modelo de solicitud-respuesta. Sin embargo, puede iniciar navegaciones en el navegador web directamente, como lo hace con JavaScript.

Blazor proporciona un `NavigationManager` servicio que se puede usar para:

- Obtención de la dirección actual del explorador
- Obtención de la dirección base
- Desencadenar navegaciones
- Recibir una notificación cuando cambie la dirección

Para navegar a otra dirección, use el `NavigateTo` método :

razor

```
@page "/"
@inject NavigationManager NavigationManager

<button @onclick="Navigate">Navigate</button>

@code {
    void Navigate() {
        NavigationManager.NavigateTo("counter");
    }
}
```

Para obtener una descripción de todos los miembros de `NavigationManager`, vea [Aplicaciones auxiliares de URI y estado de navegación](#).

Direcciones URL base

Si su aplicación Blazor se implementa en una ruta base, debe especificar la URL base en los metadatos de la página mediante la etiqueta `<base>` para que el enrutamiento funcione

correctamente. Si la página anfitriona de la aplicación se renderiza en el servidor usando Razor, entonces puedes usar la sintaxis `~/` para especificar la dirección base de la aplicación. Si la página host es HTML estática, debe especificar explícitamente la dirección URL base.

HTML

```
<base href="~/ " />
```

Diseño de página

El diseño de página en ASP.NET Web Forms se controla mediante páginas maestras. Las páginas maestras definen una plantilla con uno o varios marcadores de posición de contenido que luego pueden ser completados por páginas individuales. Las páginas maestras se definen en archivos `.master` y comienzan con la `<%@ Master %>` directiva. El contenido de los archivos `.master` se codifica como una página `.aspx`, pero con la adición de `<asp:ContentPlaceHolder>` controles para marcar donde las páginas pueden proporcionar contenido.

Site.master

ASP.NET (C#)

```
<%@ Master Language="C#" AutoEventWireup="true" CodeBehind="Site.master.cs"
Inherits="WebApplication1.SiteMaster" %>

<!DOCTYPE html>
<html lang="en">
<head runat="server">
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title><%= Page.Title %> - My ASP.NET Application</title>
    <link href="~/favicon.ico" rel="shortcut icon" type="image/x-icon" />
</head>
<body>
    <form runat="server">
        <div class="container body-content">
            <asp:ContentPlaceHolder ID="MainContent" runat="server">
            </asp:ContentPlaceHolder>
            <hr />
            <footer>
                <p>&copy; <%= DateTime.Now.Year %> - My ASP.NET Application</p>
            </footer>
        </div>
    </form>
</body>
</html>
```

En Blazor, se controla el diseño de página mediante componentes de diseño. Los componentes de diseño heredan de `LayoutComponentBase`, que define una sola `Body` propiedad de tipo `RenderFragment`, que se puede usar para representar el contenido de la página.

MainLayout.razor

```
razor

@inherits LayoutComponentBase
<h1>Main layout</h1>
<div>
    @Body
</div>
```

Cuando se representa la página con un diseño, la página se representa dentro del contenido del diseño especificado en la ubicación donde el diseño representa su `Body` propiedad.

Para aplicar un diseño a una página, use la `@layout` directiva :

```
razor

@layout MainLayout
```

Puede especificar el diseño de todos los componentes de una carpeta y subcarpeta mediante un archivo `_Imports.razor`. También puede especificar un diseño predeterminado para todas las páginas mediante el [componente Enrutador](#).

Las páginas maestras pueden definir varios marcadores de posición de contenido, pero los diseños de Blazor solo tienen una propiedad `Body`. Esperamos que esta limitación de componentes de diseño de Blazor se aborde en una versión futura.

Las páginas maestras en ASP.NET Web Forms se pueden anidar. Es decir, una página maestra también puede usar una página maestra. Los componentes de diseño de Blazor también se pueden anidar. Puede aplicar un componente de diseño a un componente de diseño. El contenido del diseño interno se representará dentro del diseño exterior.

ChildLayout.razor

```
razor

@layout MainLayout
<h2>Child layout</h2>
<div>
    @Body
</div>
```

razor

```
@page "/"
@layout ChildLayout
<p>I'm in a nested layout!</p>
```

La salida representada para la página sería:

HTML

```
<h1>Main layout</h1>
<div>
  <h2>Child layout</h2>
  <div>
    <p>I'm in a nested layout!</p>
  </div>
</div>
```

Los diseños de Blazor no suelen definir los elementos HTML raíz de una página (`<html>`, `<body>`, `<head>`, etc.). Los elementos HTML raíz se definen en su lugar en la página host de una Blazor aplicación, que se usa para representar el contenido HTML inicial de la aplicación (consulte [Bootstrap Blazor](#)). La página host puede representar varios componentes raíz para la aplicación con marcado circundante.

Los componentes de Blazor, incluidas las páginas, no pueden representar `<script>` etiquetas. Esta restricción de representación existe porque `<script>` las etiquetas se cargan una vez y, a continuación, no se pueden cambiar. Puede producirse un comportamiento inesperado si intenta representar las etiquetas dinámicamente mediante la sintaxis de Razor. En su lugar, se deben agregar todas las `<script>` etiquetas a la página host de la aplicación.

Siguiente anterior

Administración de estados

18/06/2025

💡 Sugerencia

Este contenido es un extracto del libro electrónico Blazor para desarrolladores de ASP.NET Web Forms para Azure, disponible en [documentos de .NET](#) o como un PDF descargable gratuito que se puede leer sin conexión.

Descargar PDF



La administración de estados es un concepto clave de las aplicaciones de Formularios Web Forms, que se facilita a través de las características ViewState, Session State, Application State y Postback. Estas características con estado del marco ayudaron a ocultar la administración de estado necesaria para una aplicación y permiten a los desarrolladores de aplicaciones centrarse en ofrecer su funcionalidad. Con ASP.NET Core y Blazor, algunas de estas características se han reubicado y algunas se han quitado por completo. En este capítulo se revisa cómo mantener el estado y ofrecer la misma funcionalidad con las nuevas características de Blazor.

Solicitud de administración de estado con ViewState

Al analizar la administración de estado en la aplicación de Formularios Web Forms, muchos desarrolladores pensarán inmediatamente en ViewState. En Web Forms, ViewState gestiona el estado del contenido entre las solicitudes HTTP enviando un bloque de texto codificado de gran tamaño de ida y vuelta al navegador. El campo ViewState podría sobrecargarse con el contenido de una página que contiene muchos elementos, lo que podría expandirse a varios megabytes de tamaño.

Con Blazor Server, la aplicación mantiene una conexión continua con el servidor. El estado de la aplicación, denominado *circuito*, se mantiene en la memoria del servidor mientras la conexión se considera activa. El estado solo se eliminará cuando el usuario salga de la

aplicación o de una página determinada de la aplicación. Todos los miembros de los componentes activos están disponibles entre interacciones con el servidor.

Hay varias ventajas de esta característica:

- El estado del componente está disponible fácilmente y no se vuelve a generar entre las interacciones.
- El estado no se transmite al explorador.

Sin embargo, hay algunas desventajas en la persistencia del estado del componente en memoria para tener en cuenta:

- Si el servidor se reinicia entre la solicitud, se pierde el estado.
- La solución de equilibrio de carga del servidor web de la aplicación debe incluir sesiones permanentes para asegurarse de que todas las solicitudes del mismo explorador vuelvan al mismo servidor. Si una solicitud va a otro servidor, se perderá el estado.
- La persistencia del estado de componente en el servidor puede provocar presión de memoria en el servidor web.

Por las razones anteriores, no dependa solo de que el estado del componente permanezca en la memoria del servidor. La aplicación también debe incluir algún almacén de datos de respaldo para los datos entre solicitudes. Algunos ejemplos sencillos de esta estrategia:

- En una aplicación de carro de la compra, conserve el contenido de los nuevos elementos agregados al carro en un registro de base de datos. Si se pierde el estado en el servidor, puede reconstituirlo de los registros de la base de datos.
- En un formulario web de varios elementos, los usuarios esperarán que la aplicación recuerde valores entre cada solicitud. Escriba los datos entre cada una de las publicaciones del usuario en un almacén de datos para que se puedan capturar y ensamblar en la estructura de respuesta del formulario final cuando se complete el formulario de varias partes.

Para obtener más información sobre cómo administrar el estado en las aplicaciones Blazor, consulte [ASP.NET Administración de estado de Blazor core](#).

Mantenimiento del estado con Session

Los desarrolladores de Web Forms pueden mantener información sobre el usuario activo actual con el objeto de diccionario [Microsoft.AspNetCore.Http.ISession](#). Es lo suficientemente fácil agregar un objeto con una clave de cadena a `Session` y ese objeto estaría disponible más adelante durante las interacciones del usuario con la aplicación. En un intento de eliminar la administración de la interacción con HTTP, el `Session` objeto facilita el mantenimiento del estado.

La firma del objeto de .NET Framework `Session` no es la misma que el objeto ASP.NET Core `Session`. Tenga [en cuenta la documentación de la nueva sesión principal de ASP.NET](#) antes de decidir migrar y usar la nueva característica de estado de sesión.

La sesión está disponible en ASP.NET Core y Blazor Server, pero se desaconseja su uso en favor del almacenamiento de datos en un repositorio de manera adecuada. El estado de sesión tampoco es funcional si los visitantes rechazan el uso de cookies HTTP en la aplicación debido a problemas de privacidad.

La configuración para ASP.NET Core y estado de sesión está disponible en el artículo [Administración de estado y sesión en ASP.NET Core](#).

Estado de la aplicación

El `Application` objeto del marco de Web Forms proporciona un repositorio masivo de solicitudes cruzadas para interactuar con la configuración y el estado del ámbito de la aplicación. El estado de la aplicación era un lugar ideal para almacenar varias propiedades de configuración de aplicaciones a las que se haría referencia en todas las solicitudes, independientemente del usuario que realice la solicitud. El problema con el `Application` objeto era que los datos no persistieron en varios servidores. El estado del objeto de aplicación se perdió entre reinicios.

Al igual que con `Session`, se recomienda que los datos se muevan a un almacén de respaldo persistente al que varias instancias de servidor puedan tener acceso. Si hay datos volátiles a los que le gustaría poder acceder a través de solicitudes y usuarios, podría almacenarlo fácilmente en un servicio singleton que se pueda insertar en componentes que requieran esta información o interacción.

La construcción de un objeto para mantener el estado de la aplicación y su consumo podría parecerse a la siguiente implementación:

C#

```
public class MyApplicationState
{
    public int VisitorCounter { get; private set; } = 0;

    public void IncrementCounter() => VisitorCounter += 1;
}
```

C#

```
app.AddSingleton<MyApplicationState>();
```

razor

```
@inject MyApplicationState AppState  
  
<label>Total Visitors: @AppState.VisitorCounter</label>
```

El `MyApplicationState` objeto se crea una sola vez en el servidor y el valor `VisitorCounter` se captura y genera en la etiqueta del componente. El `VisitorCounter` valor debe conservarse y recuperarse de un almacén de datos de respaldo para la durabilidad y la escalabilidad.

En el explorador

Los datos de la aplicación también se pueden almacenar en el lado cliente en el dispositivo del usuario para que esté disponible más adelante. Hay dos características del explorador que permiten la persistencia de datos en distintos ámbitos del explorador del usuario:

- `localStorage` : tiene como ámbito todo el explorador del usuario. Si se vuelve a cargar la página, el explorador se cierra y vuelve a abrir, u otra pestaña se abre con la misma dirección URL, el explorador proporciona lo mismo `localStorage` .
- `sessionStorage` : tiene como ámbito la pestaña actual del explorador del usuario. Si se vuelve a cargar la pestaña, el estado persiste. Sin embargo, si el usuario abre otra pestaña en la aplicación o cierra y vuelve a abrir el explorador, se pierde el estado.

Puede escribir código JavaScript personalizado para interactuar con estas características o hay una serie de paquetes NuGet que puede usar para proporcionar esta funcionalidad. Uno de estos paquetes es [Microsoft.AspNetCore.ProtectedBrowserStorage](#) [↗](#).

Para obtener instrucciones sobre cómo usar este paquete para interactuar con `localStorage` y `sessionStorage`, consulte el artículo [Administración de estado de Blazor](#) .

Siguiente anterior

Formularios y validación

18/06/2025

💡 Sugerencia

Este contenido es un extracto del libro electrónico "Blazor for ASP.NET Web Forms Developers for Azure" (Blazor para desarrolladores de ASP.NET Web Forms), disponible en [Documentación de .NET](#) o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF



El marco de ASP.NET Web Forms incluye un conjunto de controles de servidor de validación que controlan la validación de la entrada del usuario escrita en un formulario (`RequiredFieldValidator`, `CompareValidator`, `RangeValidator`, etc.). El marco de ASP.NET Web Forms también admite el enlace de modelos y la validación del modelo en función de anotaciones de datos (`[Required]`, `[StringLength]`, `[Range]` etc.). La lógica de validación se puede aplicar tanto en el servidor como en el cliente mediante una validación no discreta basada en JavaScript. El `ValidationSummary` control de servidor se usa para mostrar un resumen de los errores de validación al usuario.

Blazor admite el uso compartido de la lógica de validación entre el cliente y el servidor. ASP.NET proporciona implementaciones de JavaScript pregeneradas de muchas validaciones de servidor comunes. En muchos casos, el desarrollador todavía tiene que escribir JavaScript para implementar completamente su lógica de validación específica de la aplicación. Los mismos tipos de modelo, anotaciones de datos y lógica de validación se pueden usar tanto en el servidor como en el cliente.

Blazor proporciona un conjunto de componentes de entrada. Los componentes de entrada manejan la vinculación de los datos de campo a un modelo y validan la entrada del usuario al enviar el formulario.

Componente de entrada	Elemento HTML representado
<code>InputCheckbox</code>	<code><input type="checkbox"></code>
<code>InputDate</code>	<code><input type="date"></code>
<code>InputNumber</code>	<code><input type="number"></code>
<code>InputSelect</code>	<code><select></code>
<code>InputText</code>	<code><input></code>
<code>InputTextArea</code>	<code><textarea></code>

El `EditForm` componente encapsula estos componentes de entrada y organiza el proceso de validación a través de `.EditContext`. Al crear un `EditForm`, se especifica la instancia de modelo a la que se va a enlazar mediante el `Model` parámetro. Normalmente, la validación se realiza mediante anotaciones de datos y es extensible. Para habilitar la validación basada en anotaciones de datos, agregue el `DataAnnotationsValidator` componente como elemento secundario de `.EditForm`. El componente `EditForm` proporciona un evento práctico para controlar los envíos válidos (`OnValidSubmit`) y no válidos (`OnInvalidSubmit`). También hay un evento más genérico `OnSubmit` que le permite desencadenar y controlar la validación usted mismo.

Para mostrar un resumen de errores de validación, use el `ValidationSummary` componente. Para mostrar mensajes de validación para un campo de entrada específico, use el `ValidationMessage` componente y especifique una expresión lambda para el `For` parámetro que apunta al miembro del modelo adecuado.

El siguiente tipo de modelo define varias reglas de validación mediante anotaciones de datos:

C#

```
using System;
using System.ComponentModel.DataAnnotations;

public class Starship
{
    [Required]
    [StringLength(16,
        ErrorMessage = "Identifier too long (16 character limit).")]
    public string Identifier { get; set; }

    public string Description { get; set; }

    [Required]
```

```

    public string Classification { get; set; }

    [Range(1, 100000,
        ErrorMessage = "Accommodation invalid (1-100000).")]
    public int MaximumAccommodation { get; set; }

    [Required]
    [Range(typeof(bool), "true", "true",
        ErrorMessage = "This form disallows unapproved ships.")]
    public bool IsValidatedDesign { get; set; }

    [Required]
    public DateTime ProductionDate { get; set; }
}

```

En el componente siguiente se demuestra cómo construir un formulario en Blazor basado en el tipo de modelo `Starship`.

razor

```

<h1>New Ship Entry Form</h1>

<EditForm Model="@starship" OnValidSubmit="@HandleValidSubmit">
    <DataAnnotationsValidator />
    <ValidationSummary />

    <p>
        <label for="identifier">Identifier: </label>
        <InputText id="identifier" @bind-Value="starship.Identifier" />
        <ValidationMessage For="() => starship.Identifier" />
    </p>
    <p>
        <label for="description">Description (optional): </label>
        <InputTextArea id="description" @bind-Value="starship.Description" />
    </p>
    <p>
        <label for="classification">Primary Classification: </label>
        <InputSelect id="classification" @bind-Value="starship.Classification">
            <option value="">Select classification ...</option>
            <option value="Exploration">Exploration</option>
            <option value="Diplomacy">Diplomacy</option>
            <option value="Defense">Defense</option>
        </InputSelect>
        <ValidationMessage For="() => starship.Classification" />
    </p>
    <p>
        <label for="accommodation">Maximum Accommodation: </label>
        <InputNumber id="accommodation" @bind-
Value="starship.MaximumAccommodation" />
        <ValidationMessage For="() => starship.MaximumAccommodation" />
    </p>
    <p>
        <label for="valid">Engineering Approval: </label>

```

```

        <InputCheckbox id="valid" @bind-Value="starship.IsValidatedDesign" />
        <ValidationMessage For="() => starship.IsValidatedDesign" />
    </p>
    <p>
        <label for="productionDate">Production Date: </label>
        <InputDate id="productionDate" @bind-Value="starship.ProductionDate" />
        <ValidationMessage For="() => starship.ProductionDate" />
    </p>

    <button type="submit">Submit</button>
</EditForm>

@code {
    private Starship starship = new Starship();

    private void HandleValidSubmit()
    {
        // Save the data
    }
}

```

Después del envío del formulario, los datos enlazados al modelo no se han guardado en ningún almacén de datos, como una base de datos. En una Blazor WebAssembly aplicación, los datos deben enviarse al servidor. Por ejemplo, mediante una solicitud HTTP POST. En una Blazor aplicación Server, los datos ya están en el servidor, pero deben conservarse. Controlar el acceso a datos en Blazor aplicaciones es el asunto de la sección [Tratar con datos](#) .

Recursos adicionales

Para obtener más información sobre [los formularios y la validación](#) en Blazor aplicaciones, consulte la Blazor documentación.

Siguiente anterior

Trabajar con datos

19/06/2025

💡 Sugerencia

Este contenido es un extracto del libro electrónico "Blazor for ASP.NET Web Forms Developers for Azure" (Blazor para desarrolladores de ASP.NET Web Forms), disponible en [Documentación de .NET](#) o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF



El acceso a datos es la red troncal de una aplicación de ASP.NET Web Forms. Si va a crear formularios para la web, ¿qué ocurre con esos datos? Con Web Forms, había varias técnicas de acceso a datos que se podían usar para interactuar con una base de datos:

- Orígenes de datos
- ADO.NET
- Entity Framework (un marco de trabajo para gestionar bases de datos relacionales)

Los orígenes de datos eran controles que podía colocar en una página de formularios Web Forms y configurar como otros controles. Visual Studio proporcionó un conjunto amigable de diálogos para configurar y enlazar los controles a las páginas de Web Forms. Los desarrolladores que disfrutaban de un enfoque de "código bajo" o "sin código" prefieren esta técnica cuando se publicó por primera vez Web Forms.

SqlDataSource - CustomersData				
CustomerID	CompanyName	FirstName	LastName	
Databound	Databound	Databound	Databound	Click Me! Databound
Databound	Databound	Databound	Databound	Click Me! Databound
Databound	Databound	Databound	Databound	Click Me! Databound
Databound	Databound	Databound	Databound	Click Me! Databound
Databound	Databound	Databound	Databound	Click Me! Databound

ADO.NET es el enfoque de bajo nivel para interactuar con una base de datos. Las aplicaciones podrían crear una conexión a la base de datos con comandos, tablas de datos y conjuntos de datos para interactuar. Los resultados se podrían enlazar a campos en pantalla sin mucho código. El inconveniente de este enfoque era que cada conjunto de objetos de ADO.NET (`Connection`, `Command` y `DataTable`) estaba enlazado a bibliotecas proporcionadas por un proveedor de base de datos. El uso de estos componentes hizo que el código sea rígido y difícil de migrar a una base de datos diferente.

Entity Framework (un marco de trabajo para gestionar bases de datos relacionales)

Entity Framework (EF) es el marco de asignación relacional de objetos de código abierto mantenido por .NET Foundation. Inicialmente publicado con .NET Framework, EF permite generar código para las conexiones de base de datos, los esquemas de almacenamiento y las interacciones. Con esta abstracción, puede centrarse en las reglas de negocios de la aplicación y permitir que un administrador de bases de datos de confianza administre la base de datos. En .NET, puede usar una versión actualizada de EF denominada EF Core. EF Core ayuda a generar y mantener las interacciones entre el código y la base de datos con una serie de comandos disponibles para usted mediante la `dotnet ef` herramienta de línea de comandos. Echemos un vistazo a algunos ejemplos para ayudarle a trabajar con una base de datos.

Code First de EF

Una manera rápida de empezar a crear las interacciones de la base de datos es empezar con los objetos de clase con los que desea trabajar. EF proporciona una herramienta para ayudar a generar el código de base de datos adecuado para las clases. Este enfoque se denomina desarrollo de "Code First". Considere la siguiente `Product` clase para una aplicación de escaparate de ejemplo que queremos almacenar en una base de datos relacional como Microsoft SQL Server.

C#

```
public class Product
{
    public int Id { get; set; }

    [Required]
    public string Name { get; set; }

    [MaxLength(4000)]
    public string Description { get; set; }

    [Range(0, 99999,99)]
    [DataType(DataType.Currency)]
    public decimal Price { get; set; }
}
```

El producto tiene una clave principal y tres campos adicionales que se crearían en nuestra base de datos:

- EF identificará la `Id` propiedad como clave principal por convención.
- `Name` se almacenará en una columna configurada para el almacenamiento de texto. El `[Required]` atributo que decora esta propiedad agregará una `not null` restricción para ayudar a aplicar este comportamiento declarado de la propiedad.
- `Description` se almacenará en una columna configurada para el almacenamiento de texto y tendrá una longitud máxima configurada de 4000 caracteres según lo dictado por el `[MaxLength]` atributo . El esquema de la base de datos se configurará con una columna denominada `MaxLength` usando el tipo `varchar(4000)` de datos.
- La propiedad `Price` se almacenará como moneda. El `[Range]` atributo generará restricciones adecuadas para evitar el almacenamiento de datos fuera de los valores mínimo y máximo declarados.

Es necesario agregar esta `Product` clase a una clase de contexto de base de datos que defina las operaciones de conexión y traducción con nuestra base de datos.

C#

```
public class MyDbContext : DbContext
{
    public DbSet<Product> Products { get; set; }
}
```

La `MyDbContext` clase proporciona la propiedad que define el acceso y la traducción de la `Product` clase . La aplicación configura esta clase para la interacción con la base de datos mediante las siguientes entradas en el método `Startup` de la clase `ConfigureServices` (o en la

ubicación adecuada en *Program.cs* utilizando la propiedad `builder.Services` en lugar de `services`):

C#

```
services.AddDbContext<MyDbContext>(options =>
    options.UseSqlServer("MY DATABASE CONNECTION STRING"));
```

El código anterior se conectará a una base de datos de SQL Server con la cadena de conexión especificada. Puede colocar la cadena de conexión en el archivo *appsettings.json*, las variables de entorno o otras ubicaciones de almacenamiento de configuración y reemplazar esta cadena insertada adecuadamente.

A continuación, puede generar la tabla de base de datos adecuada para esta clase mediante los siguientes comandos:

CLI de .NET

```
dotnet ef migrations add 'Create Product table'
dotnet ef database update
```

El primer comando define los cambios que va a realizar en el esquema de la base de datos como una nueva migración de EF denominada `Create Product table`. Una migración define cómo aplicar y quitar los nuevos cambios en la base de datos.

Una vez aplicada, tiene una tabla sencilla `Product` en la base de datos y algunas clases nuevas agregadas al proyecto que ayudan a administrar el esquema de la base de datos. Puede encontrar estas clases generadas, de forma predeterminada, en una nueva carpeta denominada *Migraciones*. Al realizar cambios en la `Product` clase o agregar más clases relacionadas que le gustaría interactuar con la base de datos, debe volver a ejecutar los comandos de la línea de comandos con un nuevo nombre de la migración. Este comando generará otro conjunto de clases de migración para actualizar el esquema de la base de datos.

EF Database First

Para las bases de datos existentes, puede generar las clases para EF Core mediante las herramientas de línea de comandos de .NET. Para aplicar scaffolding a las clases, use una variación del comando siguiente:

CLI de .NET

```
dotnet ef dbcontext scaffold "CONNECTION STRING"
```

```
Microsoft.EntityFrameworkCore.SqlServer -c MyDbContext -t Product -t Customer
```

El comando anterior se conecta a la base de datos mediante la cadena de conexión y el `Microsoft.EntityFrameworkCore.SqlServer` proveedor especificados. Una vez conectado, se crea una clase de contexto de base de datos denominada `MyDbContext`. Además, se crean clases auxiliares para las `Product` tablas y `Customer` que se especificaron con las `-t` opciones. Hay muchas opciones de configuración para este comando para generar la jerarquía de clases adecuada para la base de datos. Para obtener una referencia completa, consulte [la documentación del comando](#).

Puede encontrar más información sobre [EF Core](#) en el sitio de Microsoft Docs.

Interactuar con servicios web

Cuando se lanzó ASP.NET por primera vez, los servicios SOAP eran la manera preferida de intercambiar datos de servidores web y clientes. Ha cambiado mucho desde ese momento y las interacciones preferidas con los servicios han cambiado a interacciones directas del cliente HTTP. Con ASP.NET Core y Blazor, puede registrar la configuración de su `HttpClient` en `Program.cs` o en el método `Startup` de la clase `ConfigureServices`. Use esa configuración cuando necesite interactuar con el punto de conexión HTTP. Tenga en cuenta el código de configuración siguiente:

C#

```
// in Program.cs
builder.Services.AddHttpClient("github", client =>
{
    client.BaseAddress = new Uri("http://api.github.com/");
    // GitHub API versioning
    client.DefaultRequestHeaders.Add("Accept", "application/vnd.github.v3+json");
    // GitHub requires a user-agent
    client.DefaultRequestHeaders.Add("User-Agent", "BlazorWebForms-Sample");
});
```

Siempre que necesite acceder a los datos desde GitHub, cree un cliente con un nombre de `github`. El cliente se configura con la dirección base y los encabezados de solicitud se establecen correctamente. Inserte `IHttpClientFactory` en los componentes de Blazor con la directiva `@inject` o un atributo `[Inject]` en una propiedad. Cree el cliente con nombre e interactúe con los servicios mediante la sintaxis siguiente:

razor

```

@Inject IHttpClientFactory factory

...

@code {
    protected override async Task OnInitializedAsync()
    {
        var client = factory.CreateClient("github");
        var response = await client.GetAsync("repos/dotnet/docs/issues");
        response.EnsureStatusCode();
        var content = await response.Content.ReadAsStringAsync();
    }
}

```

Este método devuelve la cadena que describe la colección de problemas en el repositorio de GitHub *dotnet/docs* . Devuelve contenido en formato JSON y se deserializa en los objetos de incidencias de GitHub adecuados. Hay muchas maneras de configurar el `HttpClientFactory` para entregar objetos preconfigurados `HttpClient` . Pruebe a configurar varias `HttpClient` instancias con nombres y puntos de conexión diferentes para los distintos servicios web con los que trabaja. Este enfoque hará que sea más fácil trabajar con las interacciones con esos servicios en cada página. Para obtener más información, consulte [Realización de solicitudes HTTP con IHttpClientFactory](#).

Siguiente anterior

Módulos, controladores y middleware

18/06/2025

💡 Sugerencia

Este contenido es un extracto del libro electrónico "Blazor for ASP.NET Web Forms Developers for Azure" (Blazor para desarrolladores de ASP.NET Web Forms), disponible en [Documentación de .NET](#) o como un PDF descargable y gratuito que se puede leer sin conexión.

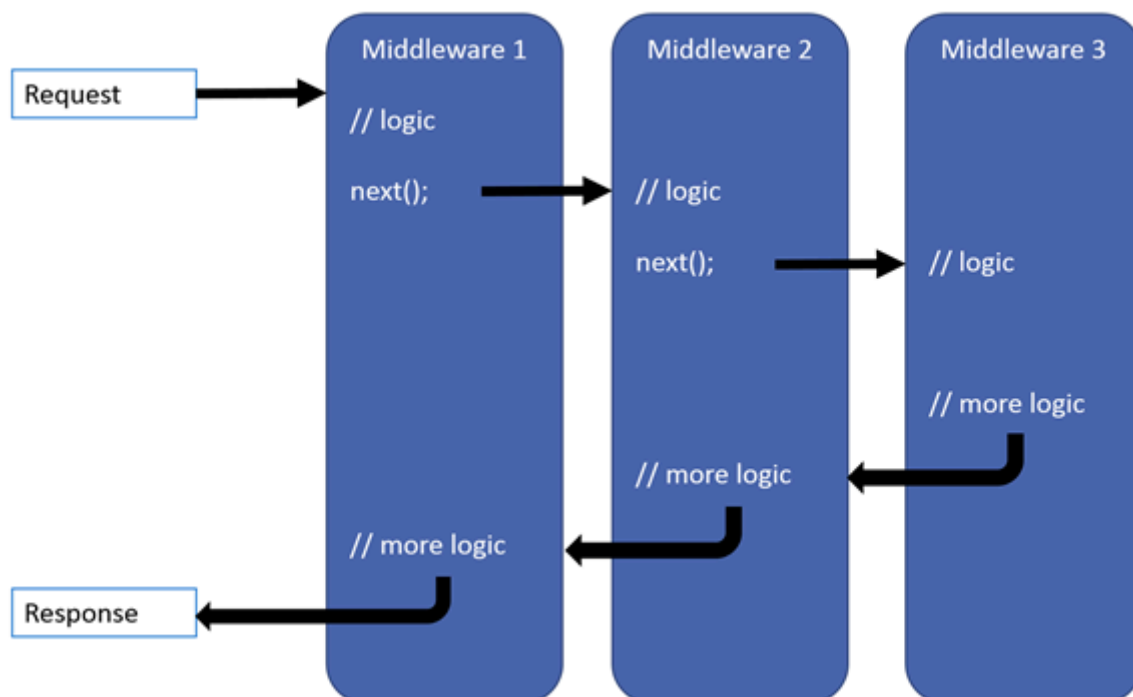
Descargar PDF



Una aplicación ASP.NET Core se basa en una serie de *middleware*. El middleware es un controlador que se organiza en un pipeline para gestionar solicitudes y respuestas. En una aplicación de Web Forms, los controladores HTTP y los módulos resuelven problemas similares. En ASP.NET Core, los módulos, controladores, *Global.asax.cs* y el ciclo de vida de la aplicación se reemplazan por middleware. En este capítulo, obtendrá información sobre el middleware en el contexto de una Blazor aplicación.

Información general

La canalización de solicitudes de ASP.NET Core consiste en una secuencia de delegados de solicitud a los que se llama de uno en uno. En el siguiente diagrama se muestra este concepto. El subproceso de ejecución sigue las flechas negras.



El diagrama anterior carece de un concepto de eventos de ciclo de vida. Este concepto es fundamental para la forma en que se controlan las solicitudes de formularios Web Forms ASP.NET. Este sistema facilita el razonamiento sobre el proceso que está ocurriendo y permite insertar el middleware en cualquier punto. El middleware se ejecuta en el orden en que se agrega a la canalización de solicitudes. También se agregan en código en lugar de archivos de configuración, normalmente en *Startup.cs*.

Katana

Los lectores familiarizados con Katana se sentirán cómodos en ASP.NET Core. De hecho, Katana es un marco del que se deriva ASP.NET Core. Introdujo patrones de middleware y canalización similares para ASP.NET 4.x. El middleware diseñado para Katana se puede adaptar para trabajar con la canalización de ASP.NET Core.

Middleware común

ASP.NET 4.x incluye muchos módulos. De forma similar, ASP.NET Core también tiene muchos componentes de middleware disponibles. Los módulos de IIS se pueden usar en algunos casos con ASP.NET Core. En otros casos, el middleware nativo ASP.NET Core puede estar disponible.

En la tabla siguiente se enumeran los componentes y middleware de reemplazo en ASP.NET Core.

Módulo	módulo ASP.NET 4.x	opción ASP.NET Core
Errores HTTP	<code>CustomErrorModule</code>	Middleware de páginas de códigos de estado
Documento predeterminado	<code>DefaultDocumentModule</code>	Middleware de archivos predeterminados
Exploración de directorios	<code>DirectoryListingModule</code>	Middleware de exploración de directorios
Compresión dinámica	<code>DynamicCompressionModule</code>	Middleware de compresión de respuesta
Seguimiento de solicitudes fallidas	<code>FailedRequestsTracingModule</code>	Gestión de registros de ASP.NET Core
Almacenamiento en caché de archivos	<code>FileCacheModule</code>	Middleware de Cacheo de Respuestas
Almacenamiento en caché HTTP	<code>HttpCacheModule</code>	Middleware de caché de respuestas
Registro HTTP	<code>HttpLoggingModule</code>	Registro de ASP.NET Core
Redireccionamiento HTTP	<code>HttpRedirectionModule</code>	Middleware de reescritura de URL
Filtros ISAPI	<code>IsapiFilterModule</code>	Middleware
ISAPI	<code>IsapiModule</code>	Middleware
Filtrado de solicitudes	<code>RequestFilteringModule</code>	IRule de middleware de reescritura de URL
Reescritura de direcciones URL†	<code>RewriteModule</code>	Middleware de reescritura de dirección URL
Compresión estática	<code>StaticCompressionModule</code>	Middleware de compresión de respuesta
Contenido estático	<code>StaticFileModule</code>	Middleware de archivos estáticos
Autorización de direcciones URL	<code>UrlAuthorizationModule</code>	ASP.NET Core Identity

Esta lista no es exhaustiva, pero debe dar una idea de qué mapeo existe entre los dos marcos. Para obtener una lista más detallada, consulte [Módulos de IIS con ASP.NET Core](#).

Middleware personalizado

Es posible que el middleware integrado no controle todos los escenarios necesarios para una aplicación. En tales casos, tiene sentido crear su propio middleware. Hay varias maneras de

definir middleware, siendo lo más sencillo un delegado simple. Considere el middleware siguiente, que acepta una solicitud de referencia cultural de una cadena de consulta:

C#

```
public class Startup
{
    public void Configure(IApplicationBuilder app)
    {
        app.Use(async (context, next) =>
        {
            var cultureQuery = context.Request.Query["culture"];

            if (!string.IsNullOrEmpty(cultureQuery))
            {
                var culture = new CultureInfo(cultureQuery);

                CultureInfo.CurrentCulture = culture;
                CultureInfo.CurrentUICulture = culture;
            }

            // Call the next delegate/middleware in the pipeline
            await next();
        });

        app.Run(async (context) =>
            await context.Response.WriteAsync(
                $"Hello {CultureInfo.CurrentCulture.DisplayName}"));
    }
}
```

El middleware también se puede definir como clase, ya sea mediante la implementación de la interfaz `IMiddleware` o siguiendo una convención de middleware. Para obtener más información, vea [Escribir middleware personalizado de ASP.NET Core](#).

Siguiente anterior

Configuración de la aplicación

18/06/2025

💡 Sugerencia

Este contenido es un extracto del libro electrónico "Blazor for ASP.NET Web Forms Developers for Azure" (Blazor para desarrolladores de ASP.NET Web Forms), disponible en [Documentación de .NET](#) o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF



La forma principal de cargar la configuración de la aplicación en Formularios Web Forms es con entradas en el archivo *web.config*, ya sea en el servidor o en un archivo de configuración relacionado al que *hace referenciaweb.config*. Puede usar el objeto estático

`ConfigurationManager` para interactuar con la configuración de la aplicación, las cadenas de conexión del repositorio de datos y otros proveedores de configuración extendidos que se agregan a la aplicación. Es habitual ver las interacciones con la configuración de la aplicación, como se muestra en el código siguiente:

C#

```
var configurationValue =  
ConfigurationManager.AppSettings["ConfigurationSettingName"];  
var connectionString =  
ConfigurationManager.ConnectionStrings["MyDatabaseConnectionName"].ConnectionString;
```

Con ASP.NET Core y servidor Blazor, el archivo *web.config* puede estar presente si la aplicación está hospedada en un servidor de Windows IIS. Pero, no hay ninguna interacción de `ConfigurationManager` con esta configuración y puede recibir una configuración de aplicación más estructurada de otros orígenes. Echemos un vistazo a cómo se recopila la configuración y cómo usted puede seguir accediendo a la configuración desde un archivo *web.config*.

Orígenes de configuración

ASP.NET Core reconoce que hay muchos orígenes de configuración que puede usar para la aplicación. El marco intenta ofrecer lo mejor de estas características de forma predeterminada. La configuración es leída y agregada por ASP.NET Core desde estos diversos orígenes. Los valores cargados posteriormente para la misma clave de configuración tienen prioridad sobre los valores anteriores.

ASP.NET Core se diseñó para ser consciente de la nube y facilitar la configuración de aplicaciones para los operadores y desarrolladores. ASP.NET Core es consciente del entorno y sabe si se está ejecutando en el entorno `Production` o `Development`. El indicador de entorno se establece en la variable de entorno del `ASPNETCORE_ENVIRONMENT` sistema. Si no hay ningún valor configurado, la aplicación se ejecuta de forma predeterminada en el `Production` entorno.

Tu aplicación puede activar y agregar configuración desde varios orígenes dependiendo del nombre del entorno. De forma predeterminada, la configuración se carga desde los siguientes recursos en el orden indicado:

1. *appsettings.json* archivo, si está presente
2. El archivo *appsettings.{NOMBRE_DEL_ENTORNO}.json*, si existe
3. Archivo de secretos de usuario en el disco, si está presente
4. Variables de entorno
5. Argumentos de la línea de comandos

Formato del archivo appsettings.json y cómo acceder a él

El archivo *appsettings.json* puede ser jerárquico con valores estructurados como el siguiente JSON:

JSON

```
{
  "section0": {
    "key0": "value",
    "key1": "value"
  },
  "section1": {
    "key0": "value",
    "key1": "value"
  }
}
```

Cuando se le presenta el código JSON anterior, el sistema de configuración acopla los valores secundarios y hace referencia a sus rutas de acceso jerárquicas completas. Las propiedades de la jerarquía se separan mediante un carácter de dos puntos (:). Por ejemplo, la clave `section1:key0` de configuración accede al valor `key0` del objeto literal `section1`.

Secretos de usuario

Los secretos de usuario son:

- Valores de configuración que se almacenan en un archivo JSON en la estación de trabajo del desarrollador, fuera de la carpeta de desarrollo de aplicaciones.
- Solo se carga cuando se ejecuta en el `Development` entorno.
- Asociado a una aplicación específica.
- Administrado con el comando `user-secrets` de la CLI de .NET.

Configure la aplicación para el almacenamiento de secretos ejecutando el `user-secrets` comando :

CLI de .NET

```
dotnet user-secrets init
```

El comando anterior agrega un `UserSecretsId` elemento al archivo del proyecto. El elemento contiene un GUID, que se usa para asociar secretos a la aplicación. A continuación, puede definir un secreto con el `set` comando . Por ejemplo:

CLI de .NET

```
dotnet user-secrets set "Parent:ApiKey" "12345"
```

El comando anterior hace que la `Parent:ApiKey` clave de configuración esté disponible en la estación de trabajo de un desarrollador con el valor `12345`.

Para obtener más información sobre cómo crear, almacenar y administrar secretos de usuario, consulte el [documento Almacenamiento seguro de secretos de aplicaciones en desarrollo en ASP.NET Core](#) .

Variables de entorno

El siguiente conjunto de valores cargados en la configuración de la aplicación es las variables de entorno del sistema. Ahora se puede acceder a todos los valores de la variable de entorno

del sistema a través de la API de configuración. Los valores jerárquicos se acoplan y separan mediante caracteres de dos puntos cuando se leen dentro de la aplicación. Pero algunos sistemas operativos no admiten los nombres de variables de entorno con caracteres de dos puntos. ASP.NET Core soluciona esta limitación mediante la conversión de valores que tienen caracteres de subrayado doble (__) en un signo de dos puntos cuando se accede a ellos. El valor de `Parent:ApiKey` de la sección de secretos de usuario anterior se puede sobrescribir con la variable de entorno `Parent__ApiKey`.

Argumentos de la línea de comandos

La configuración también se puede proporcionar como argumentos de línea de comandos cuando se inicia la aplicación. Use la notación de doble guión (--) o barra diagonal (/) para indicar el nombre del valor de configuración que se va a establecer y el valor que se va a configurar. La sintaxis es similar a los siguientes comandos:

CLI de .NET

```
dotnet run CommandLineKey1=value1 --CommandLineKey2=value2 /CommandLineKey3=value3
dotnet run --CommandLineKey1 value1 /CommandLineKey2 value2
dotnet run Parent:ApiKey=67890
```

El regreso de web.config

Si ha implementado la aplicación en Windows en IIS, el archivo *web.config* sigue configurando IIS para administrar la aplicación. De forma predeterminada, IIS agrega una referencia al módulo principal de ASP.NET (ANCM). ANCM es un módulo IIS nativo que hospeda la aplicación en lugar del servidor web Kestrel. Esta sección *web.config* es similar al siguiente marcado XML:

XML

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <location path="." inheritInChildApplications="false">
    <system.webServer>
      <handlers>
        <add name="aspNetCore" path="*" verb="*" modules="AspNetCoreModuleV2"
resourceType="Unspecified" />
      </handlers>
      <aspNetCore processPath=".\\MyApp.exe"
        stdoutLogEnabled="false"
        stdoutLogFile=".\logs\\stdout"
        hostingModel="inprocess" />
    </system.webServer>
  </location>
</configuration>
```

```
</location>
</configuration>
```

La configuración específica de la aplicación se puede definir anidando un elemento `environmentVariables` dentro del elemento `aspNetCore`. Los valores definidos en esta sección se presentan a la aplicación ASP.NET Core como variables de entorno. Las variables de entorno se cargan correctamente durante ese segmento de inicio de la aplicación.

XML

```
<aspNetCore processPath="dotnet"
  arguments=".\\MyApp.dll"
  stdoutLogEnabled="false"
  stdoutLogFile=".\logs\\stdout"
  hostingModel="inprocess">
  <environmentVariables>
    <environmentVariable name="ASPNETCORE_ENVIRONMENT" value="Development" />
    <environmentVariable name="Parent:ApiKey" value="67890" />
  </environmentVariables>
</aspNetCore>
```

Leer la configuración en la aplicación

ASP.NET Core proporciona la configuración de la aplicación a través de la `IConfiguration` interfaz. Los Blazor componentes, las Blazor páginas y cualquier otra clase administrada por ASP.NET Core que necesite acceso a la configuración deben solicitar esta interfaz de configuración. El marco ASP.NET Core rellenará automáticamente esta interfaz con la configuración resuelta configurada anteriormente. En una página Blazor o en el marcado Razor de un componente, puede insertar el objeto `IConfiguration` utilizando una directiva `@inject` en la parte superior del archivo `.razor` de la siguiente manera:

razor

```
@inject IConfiguration Configuration
```

Esta instrucción anterior hace que el `IConfiguration` objeto esté disponible como la `Configuration` variable en todo el resto de la plantilla de Razor.

Las opciones de configuración individuales se pueden leer especificando la jerarquía de configuración buscada como parámetro de indexador:

C#

```
var mySetting = Configuration["section1:key0"];
```

Puede capturar secciones de configuración completas mediante el `GetSection` método para recuperar una colección de claves en una ubicación específica con una sintaxis similar a `GetSection("section1")` para recuperar la configuración de `section1` del ejemplo anterior.

Configuración fuertemente tipada

Con Web Forms, era posible crear un tipo de configuración fuertemente tipado que heredaba del tipo `ConfigurationSection` y de los tipos asociados. Un `ConfigurationSection` le permite configurar algunas reglas de negocio y el procesamiento para esos valores de configuración.

En ASP.NET Core, puede especificar una jerarquía de clases que recibirá los valores de configuración. Estas clases:

- No necesitan heredarse de una clase primaria.
- Debe incluir las propiedades `public` que coincidan con las propiedades y las referencias de tipo de la configuración que desea capturar.

En el *ejemplo anterior de appsettings.json*, podría definir las siguientes clases para capturar los valores:

C#

```
public class MyConfig
{
    public MyConfigSection section0 { get; set; }

    public MyConfigSection section1 { get; set; }
}

public class MyConfigSection
{
    public string key0 { get; set; }

    public string key1 { get; set; }
}
```

Esta jerarquía de clases se puede rellenar agregando la siguiente línea al `Startup.ConfigureServices` método (o la ubicación adecuada en *Program.cs* mediante la `builder.Services` propiedad en lugar de `services`):

C#

```
services.Configure<MyConfig>(Configuration);
```

En el resto de la aplicación, puede agregar un parámetro de entrada a las clases o una directiva `@inject` en las plantillas de Razor de tipo `IOptions<MyConfig>` para recibir los valores de configuración fuertemente tipados. La `IOptions<MyConfig>.Value` propiedad producirá el valor `MyConfig` obtenido a partir de los valores de configuración.

razor

```
@inject IOptions<MyConfig> options
@code {
    var MyConfiguration = options.Value;
    var theSetting = MyConfiguration.section1.key0;
}
```

Puede encontrar más información sobre la característica de opciones en el documento [Patrón de opciones de ASP.NET Core](#).

Siguiente anterior

Seguridad: Autenticación y autorización en ASP.NET Web Forms y Blazor

Artículo • 08/06/2023

💡 Sugerencia

Este contenido es un extracto del libro electrónico "Blazor for ASP.NET Web Forms Developers for Azure" (Blazor para desarrolladores de ASP.NET Web Forms), disponible en [Documentación de .NET](#) o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

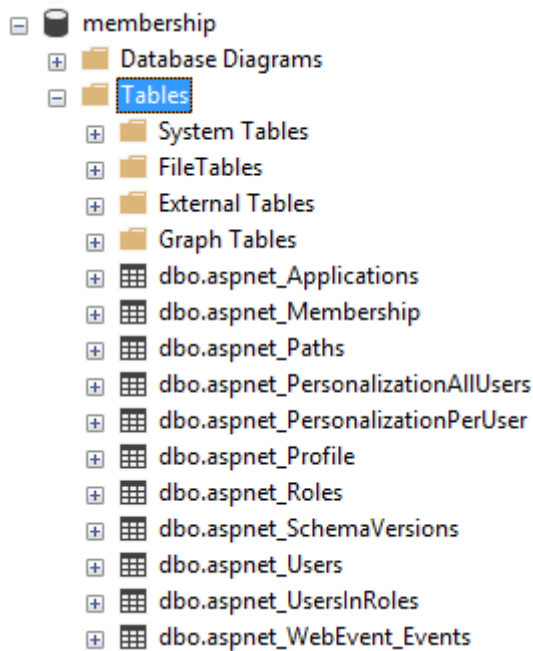


Para la migración de una aplicación de ASP.NET Web Forms a Blazor seguramente sea necesario actualizar cómo se realizan la autenticación y la autorización, siempre que en la aplicación se haya configurado la autenticación. En este capítulo se describe cómo realizar la migración desde el modelo de proveedor universal de ASP.NET Web Forms (para la pertenencia, los roles y los perfiles de usuario) y cómo trabajar con ASP.NET Core Identity desde aplicaciones Blazor. Aunque en este capítulo se tratarán los pasos y las consideraciones generales, en la documentación a la que se hace referencia se pueden encontrar pasos y scripts detallados.

Proveedores universales de ASP.NET

Desde ASP.NET 2.0, la plataforma ASP.NET Web Forms admite un modelo de proveedor para distintas características, incluida la pertenencia. El proveedor de pertenencia universal, junto con el proveedor de roles opcional, se implementa normalmente con aplicaciones de ASP.NET Web Forms. Ofrece una forma sólida y segura de administrar la autenticación y la autorización que todavía funciona correctamente en la actualidad. La oferta más reciente de estos proveedores universales está disponible como un paquete NuGet, [Microsoft.AspNet.Providers](#) [↗](#).

Proveedores universales funcionan con un esquema de base de datos SQL que incluye tablas como `aspnet_Applications`, `aspnet_Membership`, `aspnet_Roles` y `aspnet_Users`. Cuando se configuran mediante la ejecución del comando `aspnet_regsql.exe`, los proveedores instalan tablas y procedimientos almacenados que proporcionan todas las consultas y los comandos necesarios para trabajar con los datos subyacentes. El esquema de la base de datos y estos procedimientos almacenados no son compatibles con los sistemas ASP.NET Identity y ASP.NET Core Identity más recientes, por lo que los datos existentes se deben migrar al nuevo sistema. En la figura 1 se muestra un esquema de tabla de ejemplo configurado para proveedores universales.



El proveedor universal controla los usuarios, la pertenencia, los roles y los perfiles. A los usuarios se les asignan identificadores únicos globales y la información básica como `userId`, `userName`, etc., se almacenan en la tabla `aspnet_Users`. La información de autenticación, como la contraseña, el formato y la sal de contraseña, los contadores de bloqueo y los detalles, etc., se almacenan en la tabla `aspnet_Membership`. Los roles constan simplemente de nombres e identificadores únicos, que se asignan a los usuarios mediante la tabla de asociación `aspnet_UsersInRoles`, lo que proporciona una relación de varios a varios.

Si el sistema existente usa roles además de la pertenencia, tendrá que migrar las cuentas de usuario, las contraseñas asociadas, los roles y la pertenencia a roles a ASP.NET Core Identity. Lo más probable es que tenga que actualizar el código en el que actualmente realiza las comprobaciones de rol mediante instrucciones `if` para aprovechar en su lugar filtros declarativos, atributos o asistentes de etiquetas. Al final de este capítulo se revisarán las consideraciones de migración con más detalle.

Configuración de autorización en Web Forms

Para configurar el acceso autorizado a determinadas páginas en una aplicación de ASP.NET Web Forms, normalmente se especifica que los usuarios anónimos no pueden acceder determinadas páginas o carpetas anónimos. Esta configuración se realiza en el archivo web.config:

XML

```
<?xml version="1.0"?>
<configuration>
  <system.web>
    <authentication mode="Forms">
      <forms defaultUrl="~/home.aspx" loginUrl="~/login.aspx"
        slidingExpiration="true" timeout="2880"></forms>
    </authentication>

    <authorization>
      <deny users="?" />
    </authorization>
  </system.web>
</configuration>
```

En la sección de configuración `authentication` se configura la autenticación de formularios para la aplicación. La sección `authorization` se usa para impedir usuarios anónimos en toda la aplicación. Pero puede proporcionar reglas de autorización más pormenorizadas por cada ubicación, además de aplicar comprobaciones de autorización basadas en roles.

XML

```
<location path="login.aspx">
  <system.web>
    <authorization>
      <allow users="*" />
    </authorization>
  </system.web>
</location>
```

La configuración anterior, cuando se combina con la primera, permitiría a los usuarios anónimos acceder a la página de inicio de sesión, lo que invalidaría la restricción en todo el sitio para los usuarios no autenticados.

XML

```
<location path="/admin">
  <system.web>
    <authorization>
      <allow roles="Administrators" />
      <deny users="*" />
    </authorization>
  </system.web>
</location>
```

```
</authorization>
</system.web>
</location>
```

La configuración anterior, cuando se combina con las demás, restringe el acceso a la carpeta `/admin` y a todos los recursos que contiene a los miembros del rol "Administradores". Esta restricción también se puede aplicar si se coloca un archivo `web.config` independiente dentro de la raíz de la carpeta `/admin`.

Código de autorización en Web Forms

Además de configurar el acceso con `web.config`, puede configurar mediante programación el acceso y el comportamiento en la aplicación de Web Forms. Por ejemplo, puede restringir la capacidad de realizar determinadas operaciones o ver determinados datos según el rol del usuario.

Este código se puede usar en la lógica de código subyacente, así como en la propia página:

HTML

```
<% if (HttpContext.Current.User.IsInRole("Administrators")) { %>
  <a href="/admin">Go To Admin</a>
<% } %>
```

Además de comprobar la pertenencia al rol de usuario, también puede determinar si está autenticado (aunque esto normalmente se realiza mejor mediante la configuración basada en la ubicación que se ha explicado anteriormente). A continuación se muestra un ejemplo de este enfoque.

C#

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!User.Identity.IsAuthenticated)
    {
        FormsAuthentication.RedirectToLoginPage();
    }
    if (!Roles.IsUserInRole(User.Identity.Name, "Administrators"))
    {
        MessageLabel.Text = "Only administrators can view this.";
        SecretPanel.Visible = false;
    }
}
```

En el código anterior, se usa el control de acceso basado en rol (RBAC) para determinar si elementos concretos de la página, como `SecretPanel1`, son visibles según el rol del usuario actual.

Normalmente, las aplicaciones de ASP.NET Web Forms configuran la seguridad en el archivo `web.config` y, después, agregan comprobaciones adicionales cuando sea necesario en las páginas `.aspx` y sus archivos de código subyacente `.aspx.cs` relacionados. La mayoría de las aplicaciones aprovechan el proveedor de pertenencia universal, habitualmente junto al proveedor de roles adicional.

Identidad de ASP.NET Core

Aunque se siga encargando de la autenticación y la autorización, en ASP.NET Core Identity se usa otro conjunto de abstracciones y suposiciones en comparación con los proveedores universales. Por ejemplo, el nuevo modelo de Identity admite la autenticación de terceros, lo que permite a los usuarios autenticarse mediante una cuenta de medios sociales u otro proveedor de autenticación de confianza.

ASP.NET Core Identity admite la interfaz de usuario para las páginas que normalmente son necesarias, como las de inicio de sesión, cierre de sesión y registro. Aprovecha EF Core para su acceso a datos y usa migraciones de EF Core para generar el esquema necesario para admitir su modelo de datos. En esta [Introducción a Identity en ASP.NET Core](#) se proporciona información general de lo que se incluye con ASP.NET Core Identity y cómo empezar a trabajar con ella. Si todavía no ha configurado ASP.NET Core Identity en la aplicación y su base de datos, le ayudará a empezar.

Roles, notificaciones y directivas

Tanto los proveedores universales como ASP.NET Core Identity admiten el concepto de roles. Puede crear roles para usuarios y asignar usuarios a roles. Los usuarios pueden pertenecer a cualquier número de roles y se puede comprobar la pertenencia a roles como parte de la implementación de autorización.

Además de los roles, ASP.NET Core Identity admite los conceptos de notificaciones y directivas. Mientras que un rol se debe corresponder específicamente a un conjunto de recursos a los que un usuario de ese rol debe poder acceder, una notificación es simplemente parte de la identidad de un usuario. Una notificación es un par de valor y nombre que representa lo que es el sujeto, no lo que puede hacer.

Es posible inspeccionar directamente las notificaciones de un usuario y determinar, según estos valores, si se le debe conceder acceso a un recurso. Pero estas

comprobaciones suelen ser repetitivas y dispersas por todo el sistema. Un enfoque mejor consiste en definir una *directiva*.

Una directiva de autorización consta de uno o varios requisitos. Las directivas se registran como parte de la configuración del servicio de autorización en el método `ConfigureServices` de `Startup.cs`. Por ejemplo, en el fragmento de código siguiente se configura una directiva denominada "CanadiansOnly", que incluye el requisito de que el usuario tenga la notificación Country con el valor "Canada".

C#

```
services.AddAuthorization(options =>
{
    options.AddPolicy("CanadiansOnly", policy =>
policy.RequireClaim(ClaimTypes.Country, "Canada"));
});
```

Puede [obtener más información sobre cómo crear directivas personalizadas en la documentación](#).

Si usa directivas o roles, puede especificar que en una página concreta de la aplicación Blazor se necesite ese rol o esa directiva con el atributo `[Authorize]`, aplicado con la directiva `@attribute`.

Si se necesita un rol:

C#

```
@attribute [Authorize(Roles ="administrators")]
```

Si se tiene que satisfacer una directiva:

C#

```
@attribute [Authorize(Policy ="CanadiansOnly")]
```

Si necesita acceso al estado de autenticación de un usuario, a los roles o a las notificaciones en el código, hay dos formas principales de lograr esta funcionalidad. La primera consiste en recibir el estado de autenticación como un parámetro en cascada. La segunda consiste en acceder al estado mediante un elemento `AuthenticationStateProvider` insertado. Los detalles de cada uno de estos enfoques se describen en la [documentación de seguridad de Blazor](#).

En el código siguiente se muestra la forma de recibir `AuthenticationState` como un parámetro en cascada:

C#

```
[CascadingParameter]
private Task<AuthenticationState> authenticationStateTask { get; set; }
```

Con este parámetro implementado, puede obtener el usuario mediante este código:

C#

```
var authState = await authenticationStateTask;
var user = authState.User;
```

En el código siguiente se muestra cómo insertar `AuthenticationStateProvider`:

C#

```
@using Microsoft.AspNetCore.Components.Authorization
@Inject AuthenticationStateProvider AuthenticationStateProvider
```

Con el proveedor implementado, puede obtener acceso al usuario con el código siguiente:

C#

```
AuthenticationState authState = await
AuthenticationStateProvider.GetAuthenticationStateAsync();
ClaimsPrincipal user = authState.User;

if (user.Identity.IsAuthenticated)
{
    // work with user.Claims and/or user.Roles
}
```

Nota: El componente `AuthorizeView`, que se describe más adelante en este capítulo, proporciona una manera declarativa de controlar lo que un usuario ve en una página o un componente.

Para trabajar con usuarios y notificaciones (en aplicaciones Blazor Server), es posible que también tenga que insertar un objeto `userManager<T>` (use `IdentityUser` de forma predeterminada), que puede utilizar para enumerar y modificar las notificaciones de un usuario. En primer lugar, inserte el tipo y asígnelo a una propiedad:

C#

```
@inject UserManager<IdentityUser> MyUserManager
```

Después, úselo para trabajar con las notificaciones del usuario. En el ejemplo siguiente se muestra cómo agregar y conservar una notificación para un usuario:

C#

```
private async Task AddCountryClaim()
{
    var authState = await
AuthenticationStateProvider.GetAuthenticationStateAsync();
    var user = authState.User;
    var identityUser = await
MyUserManager.FindByNameAsync(user.Identity.Name);

    if (!user.HasClaim(c => c.Type == ClaimTypes.Country))
    {
        // stores the claim in the cookie
        ClaimsIdentity id = new ClaimsIdentity();
        id.AddClaim(new Claim(ClaimTypes.Country, "Canada"));
        user.AddIdentity(id);

        // save the claim in the database
        await MyUserManager.AddClaimAsync(identityUser, new
Claim(ClaimTypes.Country, "Canada"));
    }
}
```

Si tiene que trabajar con roles, siga el mismo enfoque. Es posible que tenga que insertar un objeto `RoleManager<T>` (use `IdentityRole` para el tipo predeterminado) para enumerar y administrar los roles.

Nota: En los proyectos de Blazor WebAssembly, tendrá que proporcionar API de servidor para realizar estas operaciones (en lugar de usar `UserManager<T>` o `RoleManager<T>` directamente). Una aplicación cliente de Blazor WebAssembly administraría las notificaciones o los roles mediante la llamada segura de puntos de conexión de API expuestos para este fin.

Guía de migración

Para la migración desde ASP.NET Web Forms y proveedores universales a ASP.NET Core Identity se necesitan varios pasos:

1. Crear el esquema de base de datos de ASP.NET Core Identity en la base de datos de destino
2. Realizar la migración de los datos desde el esquema del proveedor universal al de ASP.NET Core Identity
3. Realizar la migración de la configuración desde `web.config` a middleware y servicios, normalmente en `Program.cs` (o una clase `Startup`)
4. Actualizar páginas individuales mediante controles y condicionales para usar aplicaciones auxiliares de etiquetas y nuevas API de identidad.

En las siguientes secciones se describen estos pasos.

Creación del esquema ASP.NET Core Identity

Hay varias maneras de crear la estructura de tablas necesaria que se usa para ASP.NET Core Identity. La más sencilla consiste en crear una aplicación web de ASP.NET Core. Elija Aplicación web y, después, cambie el tipo de autenticación para usar cuentas individuales.

The screenshot shows a window titled "Additional information" for creating a new ASP.NET Core Web App. At the top, there are tabs for different environments: C#, Linux, macOS, Windows, Cloud, Service, and Web. The "Web" tab is selected. Below the tabs, there are three sections: "Framework" with a dropdown menu set to ".NET 6.0 (Long-term support)", "Authentication type" with a dropdown menu set to "Individual Accounts", and "Docker OS" with a dropdown menu set to "Linux". There are also two checkboxes: "Configure for HTTPS" (checked) and "Enable Docker" (unchecked). At the bottom right, there are two buttons: "Back" and "Create".

Desde la línea de comandos, puede hacer lo mismo si ejecuta `dotnet new webapp -au Individual1`. Una vez que se ha creado la aplicación, ejecútela y regístrela en el sitio. Debería desencadenar una página como la que se muestra a continuación:


```
dotnet ef database update
```

Si prefiere ejecutar un script para aplicar el nuevo esquema a una base de datos existente, puede crear scripts de estas migraciones desde la línea de comandos. Ejecute este comando para generar el script:

PowerShell

```
dotnet ef migrations script -o auth.sql
```

El comando anterior generará un script SQL en el archivo de salida `auth.sql`, que se puede ejecutar en cualquier base de datos que prefiera. Si tiene algún problema al ejecutar los comandos `dotnet ef`, [asegúrese de que tiene las herramientas de EF Core instaladas en el sistema](#).

En el caso de que tenga columnas adicionales en las tablas de origen, tendrá que identificar la mejor ubicación para estas columnas en el nuevo esquema. Por lo general, las columnas que se encuentran en la tabla `aspnet_Membership` se deben asignar a la tabla `AspNetUsers`. Las columnas de `aspnet_Roles` se deben asignar a `AspNetRoles`. Cualquier columna adicional de la tabla `aspnet_UsersInRoles` se agregará a la tabla `AspNetUserRoles`.

También merece la pena considerar la posibilidad de colocar las columnas adicionales en tablas independientes. De este modo, las migraciones futuras no deberán tener en cuenta estas personalizaciones del esquema de identidad predeterminado.

Migración de datos desde proveedores universales a ASP.NET Core Identity

Una vez que tenga el esquema de tablas de destino, el siguiente paso consiste en realizar la migración de los registros de usuario y de rol al nuevo esquema. [Aquí](#) puede encontrar una lista completa de las diferencias de esquema, incluidas qué columnas se asignan a las columnas nuevas.

Para realizar la migración de los usuarios de la pertenencia a las nuevas tablas de identidad, debe [seguir los pasos descritos en la documentación](#). Después de seguir estos pasos y el script proporcionado, los usuarios tendrán que cambiar su contraseña la próxima vez que inicien sesión.

Se puede realizar la migración de las contraseñas de usuario, pero el proceso es mucho más complicado. Exigir a los usuarios que actualicen sus contraseñas como parte del

proceso de migración y animarles a usar contraseñas nuevas y únicas, es probable que mejore la seguridad global de la aplicación.

Migración de la configuración de seguridad de web.config a inicio de la aplicación

Como se ha indicado antes, los proveedores de roles y la pertenencia a ASP.NET se configuran en el archivo `web.config` de la aplicación. Como las aplicaciones de ASP.NET Core no están asociadas a IIS y usan un sistema independiente para la configuración, esta configuración se debe establecer en otra parte. En su mayor parte, ASP.NET Core Identity se configura en el archivo `Program.cs`. Abra el proyecto web que se ha creado antes (para generar el esquema de la tabla de identidad) y revise el archivo `Program.cs` (o `Startup.cs`).

Este código agrega compatibilidad con EF Core e Identity:

C#

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

El método de extensión `AddDefaultIdentity` se usa para configurar Identity con el fin de utilizar el objeto `ApplicationDbContext` predeterminado y el tipo `IdentityUser` del marco. Si usa un objeto `IdentityUser` personalizado, asegúrese de especificar aquí su tipo. Si estos métodos de extensión no funcionan en la aplicación, compruebe que tiene las instrucciones using adecuadas y las referencias de paquetes NuGet necesarias. Por ejemplo, en el proyecto se debe hacer referencia a alguna versión de los paquetes `Microsoft.AspNetCore.Identity.EntityFrameworkCore` y `Microsoft.AspNetCore.Identity.UI`.

También en `Program.cs`, debería ver el middleware necesario configurado para el sitio. En concreto, se deben configurar `UseAuthentication` y `UseAuthorization`, y en la ubicación adecuada.

C#

```
// 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.MapControllers();
app.MapBlazorHub();
app.MapFallbackToPage("/_Host");
```

ASP.NET Identity no configura el acceso anónimo o basado en roles a ubicaciones desde *Program.cs*. Tendrá que realizar la migración de los datos de configuración de autorización específicos de la ubicación a filtros de ASP.NET Core. Anote las carpetas y páginas en las que se necesitarán estas actualizaciones. Realizará estos cambios en la sección siguiente.

Actualización de páginas individuales para usar abstracciones de ASP.NET Core Identity

En la aplicación de ASP.NET Web Forms, si tiene valores `web.config` para denegar a los usuarios anónimos el acceso a determinadas páginas o carpetas, debe realizar la migración de estos cambios mediante la adición del atributo `[Authorize]` a esas páginas:

```
razor
```

```
@attribute [Authorize]
```

Si ha denegado el acceso todavía más excepto para los usuarios que pertenecen a un rol concreto, tendría que migrar igualmente este comportamiento mediante la adición

de un atributo que especifique un rol:

razor

```
@attribute [Authorize(Roles ="administrators")]
```

El atributo `[Authorize]` solo funciona en los componentes `@page` a los que se accede por medio del enrutador de Blazor. El atributo no funciona con componentes secundarios que, en su lugar, deben usar `AuthorizeView`.

Si tiene lógica en el marcado de página para determinar si se va a mostrar código a un usuario concreto, puede reemplazarla por el componente `AuthorizeView`. El componente `AuthorizeView` muestra selectivamente la interfaz de usuario en función de si el usuario está autorizado para verla. También expone una variable `context` que se puede usar para acceder a información del usuario.

razor

```
<AuthorizeView>
  <Authorized>
    <h1>Hello, @context.User.Identity.Name!</h1>
    <p>You can only see this content if you are authenticated.</p>
  </Authorized>
  <NotAuthorized>
    <h1>Authentication Failure!</h1>
    <p>You are not signed in.</p>
  </NotAuthorized>
</AuthorizeView>
```

Puede acceder al estado de autenticación dentro de la lógica de procedimientos si accede al usuario desde un elemento `Task<AuthenticationState>` configurado con el atributo `[CascadingParameter]`. Esta configuración le permitirá acceder al usuario, lo que le permite determinar si está autenticado y si pertenece a un rol concreto. Si tiene evaluar una directiva mediante procedimientos, puede insertar una instancia de `IAuthorizationService` y, dentro, llamadas al método `AuthorizeAsync`. En el código de ejemplo siguiente se muestra cómo obtener información del usuario y permitir que un usuario autorizado realice una tarea restringida por la directiva `content-editor`.

razor

```
@using Microsoft.AspNetCore.Authorization
@inject IAuthorizationService AuthorizationService

<button @onclick="@DoSomething">Do something important</button>
```

```

@code {
    [CascadingParameter]
    private Task<AuthenticationState> authenticationStateTask { get; set; }

    private async Task DoSomething()
    {
        var user = (await authenticationStateTask).User;

        if (user.Identity.IsAuthenticated)
        {
            // Perform an action only available to authenticated (signed-in)
users.
        }

        if (user.IsInRole("admin"))
        {
            // Perform an action only available to users in the 'admin'
role.
        }

        if ((await AuthorizationService.AuthorizeAsync(user, "content-
editor"))
            .Succeeded)
        {
            // Perform an action only available to users satisfying the
            // 'content-editor' policy.
        }
    }
}

```

El primer elemento `AuthenticationState` se debe configurar como un valor en cascada antes de poder enlazarlo a un parámetro en cascada como este. Esto se suele hacer mediante el componente `CascadingAuthenticationState`. Normalmente esta configuración se realiza en `App.razor`:

razor

```

<CascadingAuthenticationState>
    <Router AppAssembly="@typeof(Program).Assembly">
        <Found Context="routeData">
            <AuthorizeRouteView RouteData="@routeData"
                DefaultLayout="@typeof(MainLayout)" />
        </Found>
        <NotFound>
            <LayoutView Layout="@typeof(MainLayout)">
                <p>Sorry, there's nothing at this address.</p>
            </LayoutView>
        </NotFound>
    </Router>
</CascadingAuthenticationState>

```

Resumen

Blazor usa el mismo modelo de seguridad que ASP.NET Core, que es ASP.NET Core Identity. La migración desde proveedores universales a ASP.NET Core Identity es relativamente sencilla, siempre que no se haya aplicado demasiada personalización al esquema de datos original. Una vez que se han migrado los datos, el trabajo con la autenticación y la autorización en las aplicaciones Blazor está bien documentado, con compatibilidad configurable y mediante programación con la mayoría de los requisitos de seguridad.

Referencias

- [Introducción a Identity en ASP.NET Core](#)
- [Migración de la autenticación de pertenencia de ASP.NET a ASP.NET Core 2.0 Identity](#)
- [Migración de la autenticación e Identity a ASP.NET Core](#)
- [Autenticación y autorización de ASP.NET Core Blazor](#)

[Anterior](#)[Siguiete](#)

Migración de ASP.NET Web Forms a Blazor

Artículo • 08/06/2023

💡 Sugerencia

Este contenido es un extracto del libro electrónico "Blazor for ASP.NET Web Forms Developers for Azure" (Blazor para desarrolladores de ASP.NET Web Forms), disponible en [Documentación de .NET](#) o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF



La migración de una base de código desde ASP.NET Web Forms a Blazor es una tarea para la que se necesita tiempo y planificación. En este capítulo se describe el proceso. Algo que puede facilitar la transición es asegurarse de que la aplicación se adhiera a una arquitectura de *n niveles*, donde el modelo de la aplicación (en este caso, Web Forms) es independiente de la lógica de negocios. Esta separación lógica de los niveles permite identificar con claridad lo que se debe trasladar a .NET Core y Blazor.

En este ejemplo, se usa la aplicación eShop disponible en [GitHub](#). eShop es un servicio de catálogo que proporciona funciones CRUD mediante la entrada y validación de formularios.

¿Por qué se debería migrar una aplicación funcional a Blazor? En muchos casos no es necesario. ASP.NET Web Forms seguirán contando con soporte técnico durante muchos años. Pero muchas de las características que proporciona Blazor solo se admiten en una aplicación migrada. Entre esas características se incluyen las siguientes:

- Mejoras de rendimiento en el marco como `Span<T>`
- Capacidad de ejecutarse como WebAssembly
- Compatibilidad multiplataforma para Linux y macOS

- Implementación local de la aplicación o implementación de un marco compartido sin afectar a otras aplicaciones

Si estas u otras características nuevas son lo suficientemente atractivas, es posible que la migración de la aplicación tenga su utilidad. La migración puede adoptar distintas formas; puede ser toda la aplicación o solo determinados puntos de conexión en los que se necesiten los cambios. En última instancia, la decisión de realizar la migración se basa en los problemas empresariales que el desarrollador tenga que resolver.

Diferencias entre el hospedaje del lado servidor y del lado cliente

Como se ha descrito en el capítulo sobre [modelos de hospedaje](#), una aplicación Blazor se puede hospedar de dos maneras diferentes: en el lado servidor y en el lado cliente. En el modelo del lado servidor se usan conexiones SignalR de ASP.NET Core para administrar las actualizaciones del DOM mientras se ejecuta código real en el servidor. El modelo del lado cliente se ejecuta como WebAssembly dentro de un explorador y no necesita conexión a ningún servidor. Hay varias diferencias que pueden afectar a la mejor opción para una aplicación específica:

- En la actualidad, la ejecución como WebAssembly no admite todas las características (como el subprocesamiento).
- La comunicación activa entre el cliente y el servidor puede producir problemas de latencia en el modo del lado servidor
- Para el acceso a bases de datos y a servicios internos o protegidos se necesita un servicio independiente con hospedaje en el lado cliente.

En el momento de escribir este documento, el modelo del lado servidor es el que más se parece al de Web Forms. La mayor parte de este capítulo se centra en el modelo de hospedaje del lado servidor, ya que está listo para la producción.

Creación de un nuevo proyecto

Este paso de migración inicial consiste en crear un proyecto. Este tipo de proyecto se basa en los proyectos de estilo SDK de .NET y simplifica gran parte del texto reutilizable que se ha usado en los formatos de proyecto anteriores. Para obtener más información, vea el capítulo sobre la [estructura del proyecto](#).

Una vez que se ha creado el proyecto, instale las bibliotecas que se han usado en el proyecto anterior. En proyectos de Web Forms anteriores, es posible que haya usado el archivo *packages.config* para enumerar los paquetes NuGet necesarios. En el nuevo

proyecto de estilo SDK, *packages.config* se ha reemplazado por elementos `<PackageReference>` en el archivo del proyecto. Una ventaja de este enfoque es que todas las dependencias se instalan de forma transitiva. Solo se muestran las dependencias de nivel superior que le interesan.

Muchas de las dependencias que se usan están disponibles para .NET, como Entity Framework 6 y log4net. Si no hay ninguna versión de .NET o .NET Standard disponible, a menudo se puede usar la de .NET Framework. Puede que tu experiencia sea diferente. Cualquier API que se use y no esté disponible en .NET provocará un error en tiempo de ejecución. Visual Studio le informa de estos paquetes. Aparece un icono amarillo en el nodo **Referencias** del proyecto en el **Explorador de soluciones**.

En el proyecto eShop basado en Blazor, puede ver los paquetes que están instalados. Anteriormente, en el archivo *packages.config* se enumeraban todos los paquetes usados en el proyecto, lo que genera un archivo de casi 50 líneas de longitud. Un fragmento de *packages.config* es este:

XML

```
<?xml version="1.0" encoding="utf-8"?>
<packages>
  ...
  <package id="Microsoft.ApplicationInsights.Agent.Intercept"
version="2.4.0" targetFramework="net472" />
  <package id="Microsoft.ApplicationInsights.DependencyCollector"
version="2.9.1" targetFramework="net472" />
  <package id="Microsoft.ApplicationInsights.PerfCounterCollector"
version="2.9.1" targetFramework="net472" />
  <package id="Microsoft.ApplicationInsights.Web" version="2.9.1"
targetFramework="net472" />
  <package id="Microsoft.ApplicationInsights.WindowsServer" version="2.9.1"
targetFramework="net472" />
  <package id="Microsoft.ApplicationInsights.WindowsServer.TelemetryChannel"
version="2.9.1" targetFramework="net472" />
  <package id="Microsoft.AspNet.FriendlyUrls" version="1.0.2"
targetFramework="net472" />
  <package id="Microsoft.AspNet.FriendlyUrls.Core" version="1.0.2"
targetFramework="net472" />
  <package id="Microsoft.AspNet.ScriptManager.MSAjax" version="5.0.0"
targetFramework="net472" />
  <package id="Microsoft.AspNet.ScriptManager.WebForms" version="5.0.0"
targetFramework="net472" />
  ...
  <package id="System.Memory" version="4.5.1" targetFramework="net472" />
  <package id="System.Numerics.Vectors" version="4.4.0"
targetFramework="net472" />
  <package id="System.Runtime.CompilerServices.Unsafe" version="4.5.0"
targetFramework="net472" />
  <package id="System.Threading.Channels" version="4.5.0"
targetFramework="net472" />
```

```
<package id="System.Threading.Tasks.Extensions" version="4.5.1"
targetFramework="net472" />
<package id="WebGrease" version="1.6.0" targetFramework="net472" />
</packages>
```

El elemento `<packages>` incluye todas las dependencias necesarias. Es difícil identificar cuál de estos paquetes se incluyen porque son necesarios. Algunos elementos `<package>` solo se muestran para satisfacer las necesidades de las dependencias que necesita.

En el proyecto Blazor se enumeran las dependencias que necesita dentro de un elemento `<ItemGroup>` en el archivo del proyecto:

XML

```
<ItemGroup>
  <PackageReference Include="Autofac" Version="4.9.3" />
  <PackageReference Include="EntityFramework" Version="6.4.4" />
  <PackageReference Include="log4net" Version="2.0.12" />
  <PackageReference
Include="Microsoft.Extensions.Logging.Log4Net.AspNetCore" Version="2.2.12"
/>
</ItemGroup>
```

Un paquete NuGet que simplifica el trabajo de los desarrolladores de Web Forms es el [paquete de compatibilidad de Windows](#). Aunque .NET es multiplataforma, algunas características solo están disponibles en Windows. Las características específicas de Windows se ponen a disposición mediante la instalación del paquete de compatibilidad. Entre los ejemplos de estas características se incluyen el Registro, WMI y los servicios de directorio. El paquete agrega aproximadamente 20 000 API y activa muchos servicios con los que es posible que ya esté familiarizado. En el proyecto eShop no se necesita el paquete de compatibilidad; pero si en los proyectos se usan características específicas de Windows, el paquete facilita el trabajo de migración.

Habilitación del proceso de inicio

El proceso de inicio de Blazor ha cambiado con respecto a Web Forms y sigue una configuración similar a la de otros servicios de ASP.NET Core. Cuando se hospedan en el lado servidor, los componentes de Razor se ejecutan como parte de una aplicación normal de ASP.NET Core. Cuando se hospedan en el explorador con WebAssembly, los componentes de Razor utilizan un modelo de hospedaje similar. La diferencia es que los componentes se ejecutan como un servicio independiente de cualquiera de los procesos de back-end. En cualquier caso, el inicio es similar.

El archivo *Global.asax.cs* es la página de inicio predeterminada para los proyectos de Web Forms. En el proyecto eShop, este archivo configura el contenedor de inversión de control (IoC) y controla los distintos eventos de ciclo de vida de la aplicación o solicitud. Algunos de estos eventos se controlan con middleware (como `Application_BeginRequest`). Otros eventos necesitan la invalidación de servicios concretos por medio de la inserción de dependencias (DI).

Por ejemplo, el archivo *Global.asax.cs* para eShop contiene el código siguiente:

C#

```
public class Global : HttpApplication, IContainerProviderAccessor
{
    private static readonly ILog _log =
LogManager.GetLogger(System.Reflection.MethodBase.GetCurrentMethod().Declari
ngType);

    static IContainerProvider _containerProvider;
    IContainer container;

    public IContainerProvider ContainerProvider
    {
        get { return _containerProvider; }
    }

    protected void Application_Start(object sender, EventArgs e)
    {
        // Code that runs on app startup
        RouteConfig.RegisterRoutes(RouteTable.Routes);
        BundleConfig.RegisterBundles(BundleTable.Bundles);
        ConfigureContainer();
        ConfigDataBase();
    }

    /// <summary>
    /// Track the machine name and the start time for the session inside the
current session
    /// </summary>
    protected void Session_Start(Object sender, EventArgs e)
    {
        HttpContext.Current.Session["MachineName"] =
Environment.MachineName;
        HttpContext.Current.Session["SessionStartTime"] = DateTime.Now;
    }

    /// <summary>
    /// https://autofacn.readthedocs.io/en/latest/integration/webforms.html
    /// </summary>
    private void ConfigureContainer()
    {
        var builder = new ContainerBuilder();
        var mockData =
```

```

bool.Parse(ConfigurationManager.AppSettings["UseMockData"]);
    builder.RegisterModule(new ApplicationModule(mockData));
    container = builder.Build();
    _containerProvider = new ContainerProvider(container);
}

private void ConfigDataBase()
{
    var mockData =
bool.Parse(ConfigurationManager.AppSettings["UseMockData"]);

    if (!mockData)
    {
        Database.SetInitializer<CatalogDBContext>
(container.Resolve<CatalogDBInitializer>());
    }
}

protected void Application_BeginRequest(object sender, EventArgs e)
{
    //set the property to our new object
    LogicalThreadContext.Properties["activityid"] = new
ActivityIdHelper();

    LogicalThreadContext.Properties["requestinfo"] = new
WebRequestInfo();

    _log.Debug("Application_BeginRequest");
}
}

```

El archivo anterior se convierte en el archivo *Program.cs* en Blazor del lado del servidor:

C#

```

using eShopOnBlazor.Models;
using eShopOnBlazor.Models.Infrastructure;
using eShopOnBlazor.Services;
using log4net;
using System.Data.Entity;
using eShopOnBlazor;

var builder = WebApplication.CreateBuilder(args);

// add services

builder.Services.AddRazorPages();
builder.Services.AddServerSideBlazor();

if (builder.Configuration.GetValue<bool>("UseMockData"))
{
    builder.Services.AddSingleton<ICatalogService, CatalogServiceMock>();
}

```

```

else
{
    builder.Services.AddScoped<ICatalogService, CatalogService>();
    builder.Services.AddScoped<IDatabaseInitializer<CatalogDBContext>,
CatalogDBInitializer>();
    builder.Services.AddSingleton<CatalogItemHiLoGenerator>();
    builder.Services.AddScoped(_ => new
CatalogDBContext(builder.Configuration.GetConnectionString("CatalogDBContext
"))));
}

var app = builder.Build();

new LoggerFactory().AddLog4Net("log4Net.xml");

if (app.Environment.IsDevelopment())
{
    app.UseDeveloperExceptionPage();
}
else
{
    app.UseExceptionHandler("/Home/Error");
}

// Middleware for Application_BeginRequest
app.Use((ctx, next) =>
{
    LogicalThreadContext.Properties["activityid"] = new
ActivityIdHelper(ctx);
    LogicalThreadContext.Properties["requestinfo"] = new
WebRequestInfo(ctx);
    return next();
}));

app.UseStaticFiles();

app.UseRouting();

app.UseEndpoints(endpoints =>
{
    endpoints.MapBlazorHub();
    endpoints.MapFallbackToPage("/_Host");
});

ConfigDataBase(app);

void ConfigDataBase(IApplicationBuilder app)
{
    using (var scope = app.ApplicationServices.CreateScope())
    {
        var initializer =
scope.ServiceProvider.GetService<IDatabaseInitializer<CatalogDBContext>>();

        if (initializer != null)
        {

```

```
        Database.SetInitializer(initializer);  
    }  
}  
}  
  
app.Run();
```

Un cambio importante que se aprecia con respecto a Web Forms es la importancia de la inserción de dependencias (ID). La inserción de dependencias ha sido un principio básico del diseño de ASP.NET Core. Admite la personalización de casi todos los aspectos del marco ASP.NET Core. Incluso hay un proveedor de servicios integrado que se puede usar para muchos escenarios. Si se necesita más personalización, se puede admitir con muchos proyectos de la comunidad. Por ejemplo, puede sacar adelante su inversión en bibliotecas de inserción de dependencias de terceros.

En la aplicación eShop original, hay una configuración para la administración de sesiones. Como en Blazor del lado servidor se usa SignalR de ASP.NET Core para la comunicación, no se admite el estado de sesión, ya que las conexiones pueden producirse independientemente de un contexto HTTP. Una aplicación que usa el estado de sesión se tiene que rediseñar antes de poder ejecutarse como una aplicación Blazor.

Para obtener más información sobre el inicio de la aplicación, vea [Inicio de la aplicación](#).

Migración de módulos y controladores HTTP a middleware

Los módulos y controladores HTTP son patrones comunes en Web Forms para controlar la canalización de solicitudes HTTP. Las clases que implementan `IHttpModule` o `IHttpHandler` se pueden registrar y procesar solicitudes entrantes. En Web Forms, los módulos y controladores se configuran en el archivo *web.config*. En gran medida, Web Forms también se basa en el control de eventos del ciclo de vida de la aplicación. En su lugar, ASP.NET Core usa middleware. El middleware se registra en el método `Configure` de la clase `Startup`. El orden de ejecución de middleware se establece según el orden de registro.

En la sección [Habilitación del proceso de inicio](#), Web Forms ha generado un evento de ciclo de vida como el método `Application_BeginRequest`. Este evento no está disponible en ASP.NET Core. Una manera de lograr este comportamiento consiste en implementar el middleware como se ha visto en el ejemplo del archivo *Startup.cs*. Este middleware realiza la misma lógica y, después, transfiere el control al siguiente controlador en la canalización de middleware.

Para obtener más información sobre la migración de módulos y controladores, vea [Migración de controladores y módulos HTTP a middleware de ASP.NET Core](#).

Migración de archivos estáticos

Para ofrecer archivos estáticos (por ejemplo, HTML, CSS, imágenes y JavaScript), el middleware debe exponer los archivos. La llamada al método `UseStaticFiles` permite proporcionar archivos estáticos desde la ruta de acceso raíz web. El directorio raíz web predeterminado es `wwwroot`, pero se puede personalizar. Como se incluye en el archivo `Program.cs`:

```
C#  
  
...  
  
app.UseStaticFiles();  
  
...
```

El proyecto eShop permite el acceso a archivos estáticos básicos. Hay muchas personalizaciones disponibles para el acceso a los archivos estáticos. Para obtener información sobre cómo habilitar archivos predeterminados o un explorador de archivos, vea [Archivos estáticos en ASP.NET Core](#).

Migración de la configuración de agrupación y minificación del entorno de ejecución

La agrupación y la minificación son técnicas de optimización del rendimiento para reducir el número y el tamaño de las solicitudes de servidor para recuperar determinados tipos de archivo. Normalmente, JavaScript y CSS se someten a algún tipo de agrupación o minificación antes de enviarse al cliente. En ASP.NET Web Forms, estas optimizaciones se controlan en tiempo de ejecución. Las convenciones de optimización se definen como un archivo `App_Start/BundleConfig.cs`. En ASP.NET Core se adopta un enfoque más declarativo. Los archivos que se van a minificar se enumeran en un archivo, junto con la configuración de minificación específica.

Para obtener más información sobre la agrupación y la minificación, vea [Agrupación y minificación de recursos estáticos en ASP.NET Core](#).

Migración de páginas ASPX

Una página de una aplicación de Web Forms es un archivo con la extensión *.aspx*. Una página de Web Forms a menudo se puede asignar a un componente en Blazor. Un componente de Razor se crea en un archivo con la extensión *.razor*. Para el proyecto eShop, se convierten cinco páginas en una página de Razor.

Por ejemplo, la vista de detalles consta de tres archivos en el proyecto de Web Forms: *Details.aspx*, *Details.aspx.cs* y *Details.aspx.designer.cs*. Al realizar la conversión a Blazor, el código subyacente y el marcado se combinan en *Details.razor*. La compilación de Razor (equivalente al contenido de los archivos *.designer.cs*) se almacena en el directorio *obj* y, de forma predeterminada, no es visible en el **Explorador de soluciones**. La página de Web Forms consta del marcado siguiente:

ASP.NET (C#)

```
<%@ Page Title="Details" Language="C#" MasterPageFile="~/Site.Master"
AutoEventWireup="true" CodeBehind="Details.aspx.cs"
Inherits="eShopLegacyWebForms.Catalog.Details" %>

<asp:Content ID="Details" ContentPlaceHolderID="MainContent" runat="server">
    <h2 class="esh-body-title">Details</h2>

    <div class="container">
        <div class="row">
            <asp:Image runat="server" CssClass="col-md-6 esh-picture"
ImageUrl='<%=~/Pics/" + product.PictureFileName%' />
            <dl class="col-md-6 dl-horizontal">
                <dt>Name
                </dt>

                <dd>
                    <asp:Label runat="server" Text='<%=product.Name%' />
                </dd>

                <dt>Description
                </dt>

                <dd>
                    <asp:Label runat="server"
Text='<%=product.Description%' />
                </dd>

                <dt>Brand
                </dt>

                <dd>
                    <asp:Label runat="server"
Text='<%=product.CatalogBrand.Brand%' />
                </dd>

                <dt>Type
                </dt>
```

```

        <dd>
            <asp:Label runat="server"
Text='<%#product.CatalogType.Type%>' />
        </dd>
        <dt>Price
        </dt>

        <dd>
            <asp:Label CssClass="esh-price" runat="server"
Text='<%#product.Price%>' />
        </dd>

        <dt>Picture name
        </dt>

        <dd>
            <asp:Label runat="server"
Text='<%#product.PictureFileName%>' />
        </dd>

        <dt>Stock
        </dt>

        <dd>
            <asp:Label runat="server"
Text='<%#product.AvailableStock%>' />
        </dd>

        <dt>Restock
        </dt>

        <dd>
            <asp:Label runat="server"
Text='<%#product.RestockThreshold%>' />
        </dd>

        <dt>Max stock
        </dt>

        <dd>
            <asp:Label runat="server"
Text='<%#product.MaxStockThreshold%>' />
        </dd>

    </dl>
</div>

<div class="form-actions no-color esh-link-list">
    <a runat="server" href='<%# GetRouteUrl("EditProductRoute", new
{id =product.Id}) %>' class="esh-link-item">Edit
    </a>
    |
    <a runat="server" href="~" class="esh-link-item">Back to list
    </a>

```

```
</div>

</div>
</asp:Content>
```

El código subyacente del marcado anterior incluye el código siguiente:

```
C#

using eShopLegacyWebForms.Models;
using eShopLegacyWebForms.Services;
using log4net;
using System;
using System.Web.UI;

namespace eShopLegacyWebForms.Catalog
{
    public partial class Details : System.Web.UI.Page
    {
        private static readonly ILog _log =
LogManager.GetLogger(System.Reflection.MethodBase.GetCurrentMethod().DeclaringType);

        protected CatalogItem product;

        public ICatalogService CatalogService { get; set; }

        protected void Page_Load(object sender, EventArgs e)
        {
            var productId = Convert.ToInt32(Page.RouteData.Values["id"]);
            _log.Info($"Now loading... /Catalog/Details.aspx?id={productId}");
            product = CatalogService.FindCatalogItem(productId);

            this.DataBind();
        }
    }
}
```

Cuando se convierte a Blazor, la página de Web Forms se convierte al código siguiente:

```
razor

@page "/Catalog/Details/{id:int}"
@inject ICatalogService CatalogService
@inject ILogger<Details> Logger

<h2 class="esh-body-title">Details</h2>

<div class="container">
    <div class="row">
        
```

```
<dl class="col-md-6 dl-horizontal">
```

```
<dt>
```

```
    Name
```

```
</dt>
```

```
<dd>
```

```
    @_item.Name
```

```
</dd>
```

```
<dt>
```

```
    Description
```

```
</dt>
```

```
<dd>
```

```
    @_item.Description
```

```
</dd>
```

```
<dt>
```

```
    Brand
```

```
</dt>
```

```
<dd>
```

```
    @_item.CatalogBrand.Brand
```

```
</dd>
```

```
<dt>
```

```
    Type
```

```
</dt>
```

```
<dd>
```

```
    @_item.CatalogType.Type
```

```
</dd>
```

```
<dt>
```

```
    Price
```

```
</dt>
```

```
<dd>
```

```
    @_item.Price
```

```
</dd>
```

```
<dt>
```

```
    Picture name
```

```
</dt>
```

```
<dd>
```

```
    @_item.PictureFileName
```

```
</dd>
```

```
<dt>
```

```
    Stock
```

```
</dt>
```

```
<dd>
```

```

        @_item.AvailableStock
    </dd>

    <dt>
        Restock
    </dt>

    <dd>
        @_item.RestockThreshold
    </dd>

    <dt>
        Max stock
    </dt>

    <dd>
        @_item.MaxStockThreshold
    </dd>

</dl>
</div>

<div class="form-actions no-color esh-link-list">
    <a href="@($)/Catalog/Edit/{_item.Id}" class="esh-link-item">
        Edit
    </a>
    |
    <a href="/" class="esh-link-item">
        Back to list
    </a>
</div>

</div>

@code {
    private CatalogItem _item;

    [Parameter]
    public int Id { get; set; }

    protected override void OnInitialized()
    {
        Logger.LogInformation("Now loading... /Catalog/Details/{Id}", Id);

        _item = CatalogService.FindCatalogItem(Id);
    }
}

```

Observe que el código y el marcado están en el mismo archivo. Los servicios necesarios se hacen accesibles con el atributo `@inject`. Según la directiva `@page`, se puede acceder a esta página desde la ruta `Catalog/Details/{id}`. El valor del marcador de posición `{id}` de la ruta se ha restringido a un entero. Como se describe en la sección

[Enrutamiento](#), a diferencia de Web Forms, un componente de Razor indica explícitamente su ruta y los parámetros que se incluyen. Es posible que muchos controles de Web Forms no tengan homólogos exactos en Blazor. A menudo hay un fragmento de código HTML equivalente que tendrá el mismo propósito. Por ejemplo, el control `<asp:Label />` se puede reemplazar por un elemento `<label>` HTML.

Validación de modelos en Blazor

Si el código de Web Forms incluye validación, puede transferir gran parte de lo que ya tiene sin apenas cambios. Una ventaja de realizar la ejecución en Blazor es que se puede ejecutar la misma lógica de validación sin necesidad de JavaScript personalizado. Las anotaciones de datos permiten la validación sencilla del modelo.

Por ejemplo, la página *Create.aspx* tiene un formulario de entrada de datos con validación. Un fragmento de ejemplo podría tener este aspecto:

ASP.NET

```
<div class="form-group">
  <label class="control-label col-md-2">Name</label>
  <div class="col-md-3">
    <asp:TextBox ID="Name" runat="server" CssClass="form-control">
  </asp:TextBox>
    <asp:RequiredFieldValidator runat="server" ControlToValidate="Name"
    Display="Dynamic"
    CssClass="field-validation-valid text-danger" ErrorMessage="The
    Name field is required." />
  </div>
</div>
```

En Blazor, el marcado equivalente se proporciona en un archivo *Create.razor*:

razor

```
<EditForm Model="_item" OnValidSubmit="@...">
  <DataAnnotationsValidator />

  <div class="form-group">
    <label class="control-label col-md-2">Name</label>
    <div class="col-md-3">
      <InputText class="form-control" @bind-Value="_item.Name" />
      <ValidationMessage For="(() => _item.Name)" />
    </div>
  </div>

  ...
</EditForm>
```

El contexto `EditForm` incluye compatibilidad con la validación y se puede encapsular en torno a una entrada. Las anotaciones de datos son una manera común de agregar validación. Esta compatibilidad de validación se puede agregar por medio del componente `DataAnnotationsValidator`. Para obtener más información sobre este mecanismo, vea [Formularios y validación de Blazor de ASP.NET Core](#).

Migración de la configuración

En un proyecto de Web Forms, los datos de configuración se almacenan normalmente en el archivo `web.config`. A los datos de configuración se accede con `ConfigurationManager`. Los servicios normalmente tenían que analizar objetos. Con .NET Framework 4.7.2, se agregó capacidad de composición a la configuración por medio de `ConfigurationBuilders`. Estos generadores permitían a los desarrolladores agregar varios orígenes para la configuración que, después, se componía en tiempo de ejecución para recuperar los valores necesarios.

En ASP.NET Core se presentó un sistema de configuración flexible que le permite definir el origen (u orígenes) de configuración que usa la aplicación y la implementación. La infraestructura de `ConfigurationBuilder` que puede usar en la aplicación de Web Forms se modelaba en función de los conceptos usados en el sistema de configuración de ASP.NET Core.

En el fragmento de código siguiente se muestra cómo el proyecto eShop de Web Forms usa `web.config` para almacenar los valores de configuración:

XML

```
<configuration>
  <configSections>
    <section name="entityFramework"
type="System.Data.Entity.Internal.ConfigFile.EntityFrameworkSection,
EntityFramework, Version=6.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089" requirePermission="false" />
  </configSections>
  <connectionStrings>
    <add name="CatalogDBContext" connectionString="Data Source=
(localdb)\MSSQLLocalDB; Initial
Catalog=Microsoft.eShopOnContainers.Services.CatalogDb; Integrated
Security=True; MultipleActiveResultSets=True;"
providerName="System.Data.SqlClient" />
  </connectionStrings>
  <appSettings>
    <add key="UseMockData" value="true" />
    <add key="UseCustomizationData" value="false" />
  </appSettings>
</configuration>
```

Es habitual que los secretos, como las cadenas de conexión de base de datos, se almacenen en *web.config*. Los secretos se conservan inevitablemente en ubicaciones no seguras, como el control de código fuente. Con Blazor en ASP.NET Core, la configuración basada en XML anterior se reemplaza por el código JSON siguiente:

JSON

```
{
  "ConnectionStrings": {
    "CatalogDBContext": "Data Source=(localdb)\\MSSQLLocalDB; Initial
Catalog=Microsoft.eShopOnContainers.Services.CatalogDb; Integrated
Security=True; MultipleActiveResultSets=True;"
  },
  "UseMockData": true,
  "UseCustomizationData": false
}
```

JSON es el formato de configuración predeterminado; pero, ASP.NET Core admite muchos otros formatos, incluido XML. También hay varios formatos admitidos por la comunidad.

Puede acceder a valores de configuración desde el generador en *Program.cs*:

C#

```
if (builder.Configuration.GetValue<bool>("UseMockData"))
{
    builder.Services.AddSingleton<ICatalogService, CatalogServiceMock>();
}
else
{
    builder.Services.AddScoped<ICatalogService, CatalogService>();
    builder.Services.AddScoped<IDatabaseInitializer<CatalogDBContext>,
CatalogDBInitializer>();
    builder.Services.AddSingleton<CatalogItemHiLoGenerator>();
    builder.Services.AddScoped(_ => new
CatalogDBContext(builder.Configuration.GetConnectionString("CatalogDBContext
"))));
}
```

De forma predeterminada, las variables de entorno, los archivos JSON (*appsettings.json* y *yappsettings.{Entorno}.json*) y las opciones de línea de comandos se registran como orígenes de configuración válidos en el objeto de configuración. Se puede acceder a los orígenes de configuración por medio de `Configuration[key]`. Una técnica más avanzada consiste en enlazar los datos de configuración a los objetos mediante el patrón de opciones. Para obtener más información sobre la configuración y el patrón de opciones,

vea [Configuración en ASP.NET Core](#) y [Patrón de opciones en ASP.NET Core](#), respectivamente.

Migración del acceso a datos

El acceso a datos es un aspecto importante de cualquier aplicación. El proyecto eShop almacena información de catálogo en una base de datos y recupera los datos con Entity Framework (EF) 6. Como EF 6 es compatible con .NET 5, se puede seguir usando en el proyecto.

Los siguientes cambios relacionados con EF han sido necesarios para eShop:

- En .NET Framework, el objeto `DbContext` acepta una cadena con el formato *name=ConnectionString* y usa la cadena de conexión de `ConfigurationManager.AppSettings[ConnectionString]` para conectarse. En .NET Core esto no se admite. Se debe proporcionar la cadena de conexión.
- Se ha accedido a la base de datos de forma sincrónica. Aunque esto funcione, la escalabilidad puede verse afectada. Esta lógica se debe trasladar a un patrón asíncrono.

Aunque no hay la misma compatibilidad nativa para el enlace de conjuntos de datos, Blazor proporciona flexibilidad y capacidad con su compatibilidad con C# en una página de Razor. Por ejemplo, puede realizar cálculos y mostrar el resultado. Para obtener más información sobre los patrones de datos en Blazor, vea el capítulo [Acceso a datos](#).

Cambios de arquitectura

Por último, hay algunas diferencias arquitectónicas importantes que tener en cuenta a la hora de realizar la migración a Blazor. Muchos de estos cambios se aplican a cualquier elemento basado en .NET Core o en ASP.NET Core.

Como Blazor se basa en .NET Core, existen consideraciones a la hora de garantizar la compatibilidad en .NET Core. Algunos de los cambios principales incluyen la eliminación de las características siguientes:

- Varios dominios de aplicación
- Comunicación remota
- Seguridad de acceso del código (CAS)
- Transparencia de seguridad

Para obtener más información sobre las técnicas de identificación de los cambios necesarios para admitir la ejecución en .NET Core, vea [Portabilidad del código de](#)

[.NET Framework](#) a [.NET Core](#).

ASP.NET Core es una versión "reimaginada" de ASP.NET y tiene algunos cambios que es posible que no parezcan inicialmente obvios. Los cambios principales son los siguientes:

- No existe un contexto de sincronización, lo que significa que no hay `HttpContext.Current`, `Thread.CurrentPrincipal`, ni otros descriptores de acceso estáticos
- No se crean instantáneas
- No hay una cola de solicitudes

Muchas operaciones de ASP.NET Core son asincrónicas, lo que permite una descarga más sencilla de las tareas enlazadas a E/S. Es importante no realizar bloqueos mediante `Task.Wait()` o `Task.GetResult()`, lo que puede agotar rápidamente los recursos del grupo de subprocesos.

Conclusión de la migración

Llegados a este punto, ha visto muchos ejemplos de lo que se necesita para trasladar un proyecto de Web Forms a Blazor. Para obtener un ejemplo completo, vea el proyecto [eShopOnBlazor](#).

Anterior