# Contents

# Blazor para desarrolladores de ASP.NET Web Forms

18/03/2020 • 9 minutes to read • Edit Online

**IMPORTANT**

**EDICIÓN EN VERSIÓN PRELIMINAR**

En este artículo se proporciona contenido anticipado de un libro que está actualmente en elaboración. Si tiene algún comentario, envíelo en https://aka.ms/ebookfeedback.

Microsoft

# Blazor for ASP.NET Web Forms Developers

Daniel Roth
Jeff Fritz
Taylor Southwick

Preview Edition

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

PUBLICADO POR

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

División de Microsoft Corporation

One Microsoft Way

Redmond, Washington 98052-6399

Autores:

> **Daniel Roth** , administrador de programas principal, Microsoft Corp.

> **Jeff Fritz** , administrador de programas sénior, Microsoft Corp.

> **Taylor Southwick** , ingeniero de software sénior, Microsoft Corp.

> **Scott Addie** , desarrollador de contenidos sénior, Microsoft Corp.

# Introducción

Hace mucho tiempo que .NET permite desarrollar aplicaciones web a través de ASP.NET, un conjunto de plataformas y herramientas muy completo para compilar cualquier tipo de aplicación web. ASP.NET tiene su propio linaje de tecnologías y plataformas web, empezando por las clásicas páginas Active Server (ASP). Las plataformas, como ASP.NET Web Forms, ASP.NET MVC, ASP.NET Web Pages y ahora ASP.NET Core, ofrecen una manera productiva y eficaz de compilar aplicaciones web *representadas por el servidor*, donde el contenido de la interfaz de usuario se genera dinámicamente en el servidor en respuesta a las solicitudes HTTP. Cada plataforma de ASP.NET sigue una filosofía de creación de aplicaciones y satisface una audiencia en particular. ASP.NET Web Forms se incluía con la versión original de .NET Framework y permitía el desarrollo web mediante muchos de los patrones conocidos por los desarrolladores de escritorio, como los controles de IU reutilizables con un control de eventos sencillo. Sin embargo, ninguna de las ofertas de ASP.NET ofrece una manera de ejecutar el código que se ejecutaba en el explorador del usuario. Para ello, es necesario escribir en JavaScript y usar cualquiera de las muchas plataformas y herramientas de JavaScript que han ido ganando y perdiendo popularidad con los años: jQuery, Knockout, Angular, React, etc.

Blazor es una nueva plataforma web que lo cambia todo al crear aplicaciones web con .NET. Blazor es una plataforma de interfaz de usuario web del lado cliente basada 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 ejecutarlos directamente en el explorador mediante un nuevo estándar web abierto denominado WebAssembly. O, si lo prefiere, 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 tiene mucho en común con ASP.NET Web Forms, como un modelo de componentes reutilizable y una forma sencilla de controlar los eventos de usuario, Blazor también se basa en los cimientos de .NET Core para ofrecer una experiencia de desarrollo web moderna y de alto rendimiento.

Este libro es una introducción de Blazor para los desarrolladores de ASP.NET Web Forms que resulta muy útil, a su vez, y familiar. Se presentan conceptos de Blazor comparándolos con conceptos análogos de ASP.NET Web Forms, a la vez que se explican los nuevos conceptos que pueden resultar menos conocidos. Abarca toda una variedad de temas y problemas, como la creación de componentes, el enrutamiento, el diseño, la configuración y la seguridad. Aunque el contenido de este libro abarca principalmente el desarrollo de nuevas aplicaciones, también incluye instrucciones y estrategias para migrar las instancias de ASP.NET Web Forms existentes a Blazor para cuando se quiera modernizar una aplicación existente.

## Destinatarios de este libro

Este libro está dirigido a desarrolladores de ASP.NET Web Forms que buscan una introducción a Blazor, relacionándolo con sus conocimientos y habilidades existentes. Con este libro será más fácil empezar a trabajar rápidamente en un nuevo proyecto de Blazor o crear un mapa de ruta para modernizar una aplicación ASP.NET Web Forms existente.

## Cómo usar este libro

En la primera parte de este libro se presenta Blazor y se ofrece una comparación con el desarrollo de aplicaciones web con ASP.NET Web Forms. En los capítulos posteriores se tratan distintos temas sobre Blazor y se relaciona cada concepto la plataforma con el concepto correspondiente de ASP.NET Web Forms, o bien se explica en detalle cualquier concepto que sea completamente nuevo. En el libro también se hace referencia con regularidad a una aplicación de ejemplo completa, implementada en ASP.NET Web Forms y en Blazor, para mostrar las características de Blazor y proporcionar un caso práctico para la migración de ASP.NET Web Forms a Blazor. Puede encontrar ambas implementaciones de la aplicación de ejemplo (ASP.NET Web Forms y Blazor) en GitHub.

## Aspectos no tratados en este libro

Este libro es una introducción a Blazor, no una guía completa sobre la migración. Aunque incluye información sobre cómo enfocar la migración de un proyecto de ASP.NET Web Forms a Blazor, no pretende cubrir todos sus matices y detalles. Si quiere consultar una guía general sobre cómo realizar una migración de ASP.NET a ASP.NET Core, lea la guía de migración en la documentación de ASP.NET Core.

**Recursos adicionales**

Encontrará la documentación y la página oficial de Blazor en https://blazor.net.

## Envíe sus comentarios

Nos gustaría recibir sus comentarios al respecto para contribuir al desarrollo constante del libro y sus ejemplos relacionados. Si tiene algún comentario sobre cómo mejorar este libro, escríbalo en la sección pertinente situada en la parte inferior de cualquier página creada en Problemas de GitHub.

SIGUIENTE

# Una introducción a ASP.NET para desarrolladores de formularios Web Forms

23/11/2019 • 19 minutes to read • Edit Online

**IMPORTANT**

EDICIÓN EN VERSIÓN PRELIMINAR

En este artículo se proporciona contenido anticipado de un libro que está actualmente en elaboración. Si tiene algún comentario, envíelo en https://aka.ms/ebookfeedback.

El marco de trabajo de formularios Web Forms de ASP.NET ha sido una grapa del desarrollo web de .NET, dado que el .NET Framework se envió por primera vez en 2002. De nuevo, cuando la web todavía se encontraba en su gran cantidad de aplicaciones Web, los formularios Web Forms ASP.NET crearon muchos de los patrones que se usaban para el desarrollo de escritorio. En los formularios Web Forms de ASP.NET, las páginas web se pueden crear rápidamente a partir de controles de interfaz de usuario reutilizables. Las interacciones del usuario se administran de forma natural como eventos. Existe un completo ecosistema de controles de interfaz de usuario de formularios Web Forms que proporcionan los proveedores de Microsoft y de control. Los controles facilitan los esfuerzos de conectarse a orígenes de datos y mostrar visualizaciones de datos enriquecidas. En el caso de la inclinación visual, el diseñador de formularios Web Forms proporciona una sencilla interfaz de arrastrar y colocar para administrar controles.

A lo largo de los años, Microsoft ha incorporado 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 ASP.NET Core más recientemente. Con cada nuevo marco de trabajo, algunos han predicho la disminución inminente de formularios Web Forms de ASP.NET y criticized como un marco web outmoded obsoleto. A pesar de estas predicciones, muchos desarrolladores web de .NET continúan ASP.NET Web Forms de forma sencilla, estable y productiva para realizar su trabajo.

En el momento de la escritura, casi la mitad de un millón de desarrolladores web usan formularios Web Forms de ASP.NET cada mes. El marco de trabajo de formularios Web Forms ASP.NET es estable hasta el punto en que los documentos, ejemplos, libros y entradas de blog desde hace una década siguen siendo útiles y relevantes. Para muchos desarrolladores web de .NET, "ASP.NET" sigue siendo sinónimo de "ASP.NET Web Forms", tal y como era cuando se diseñó .NET por primera vez. Los argumentos de las ventajas y desventajas de los formularios Web Forms de ASP.NET en comparación con los otros marcos Web de .NET nuevos pueden aparecer en Rage. Los formularios Web Forms de ASP.NET siguen siendo un marco popular para la creación de aplicaciones Web.

Aún así, las innovaciones en el desarrollo de software no se ralentizan. Todos los desarrolladores de software deben mantenerse al día de las nuevas tecnologías y tendencias. Merece la pena considerar dos tendencias en particular:

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

## .NET de código abierto y multiplataforma

Cuando se distribuyen los formularios Web de .NET y ASP.NET por primera vez, el ecosistema de la plataforma parecía mucho diferente de lo que tiene actualmente. Windows domina los mercados de escritorio y servidor. Las plataformas alternativas, como macOS y Linux, todavía lucharon a ganar. Los formularios Web Forms de ASP.NET se distribuyen con el .NET Framework como componente solo de Windows, lo que significa que las aplicaciones de

formularios Web Forms de ASP.NET solo se pueden ejecutar en máquinas con Windows Server. Muchos entornos modernos ahora usan diferentes tipos de plataformas para servidores y equipos de desarrollo, de modo que la compatibilidad entre plataformas para muchos usuarios es un requisito absoluto.

La mayoría de los marcos web modernos ahora también son de código abierto, que ofrece una serie de ventajas. Los usuarios no se mantienen en un único propietario del proyecto para corregir errores y agregar características. Los proyectos de código abierto proporcionan una mayor transparencia en el progreso de desarrollo y en los próximos cambios. Los proyectos de código abierto disfrutan de las contribuciones de una comunidad completa y fomentan un ecosistema de código abierto compatible. 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 forma 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 auxiliares.

La comunidad de .NET ha adoptado compatibilidad entre plataformas 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 diversas distribuciones de Linux. Xamarin proporciona mono, una versión de código abierto de .NET. Mono se ejecuta en Android, iOS y otros factores de forma, incluidos los relojes y los televisores inteligentes. Microsoft ha anunciado que .net 5 va a conciliar .net Core y mono en "un solo entorno de tiempo de ejecución de .net y un marco de trabajo que se pueden usar en todas partes y que tiene comportamientos de tiempo de ejecución uniformes y experiencias del desarrollador".

¿Se beneficiará de los formularios Web Forms de ASP.NET para la compatibilidad multiplataforma y de código abierto? La respuesta, desafortunadamente, es no o, al menos, no es la misma extensión que el resto de la plataforma. El equipo de .NET lo hizo poco claro que los formularios web forms de ASP.net no se trasladarán a .net Core o .net 5. ¿Por qué?

Se produjeron esfuerzos en los primeros días de .NET Core para ASP.NET formularios Web Forms. Se encontró que el número de cambios importantes necesarios era demasiado drástico. También hay una admisión aquí que, incluso para Microsoft, existe un límite en cuanto al número de Marcos web que puede admitir simultáneamente. Es posible que alguien de la Comunidad asuma la causa de la creación de una versión de código abierto y multiplataforma de formularios Web Forms de ASP.NET. El código fuente de los formularios Web Forms de ASP.net se ha puesto a disposición públicamente en forma de referencia. Pero, por el momento, parece que los formularios Web Forms de ASP.NET seguirán siendo Windows y sin un modelo de contribución de código abierto. Si la compatibilidad entre plataformas o el código abierto son importantes para los escenarios, deberá buscar algo nuevo.

¿Esto significa que los formularios Web Forms de ASP.NET están *inactivos* y ya no se deben usar? Por supuesto, no. Siempre que el .NET Framework se distribuya como parte de Windows, los formularios Web Forms de ASP.NET serán un marco compatible. Para muchos desarrolladores de formularios Web Forms, la falta de Compatibilidad multiplataforma y de código abierto no es un problema. Si no tiene un requisito para la compatibilidad multiplataforma, código abierto o cualquiera de las otras características nuevas de .NET Core o .NET 5, el paso de los formularios Web Forms de ASP.NET en Windows es correcto. Los formularios Web Forms de ASP.NET seguirán 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. Los marcos Web basados en .net, incluidos los formularios Web Forms de ASP.NET, han tenido históricamente algo en común: están *representados por el servidor*. En las aplicaciones web representadas por el servidor, el explorador realiza una solicitud al servidor, que ejecuta código (código .NET en aplicaciones ASP.NET) para generar una respuesta. Esa respuesta se devuelve al explorador para controlar. En este modelo, el explorador se usa como un motor de representación fino. El trabajo duro de producir la interfaz de usuario, la ejecución de la lógica de negocios y el estado de administración 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 capacidades del equipo del usuario. ¿Por qué no sacar provecho de 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 parcial o totalmente del lado cliente. La lógica y los datos que deben administrarse en el servidor todavía se pueden controlar en el lado servidor. Se pueden usar llamadas de 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 la popularidad. Los desarrolladores de formularios Web Forms de ASP.NET también pueden beneficiarse del uso del cliente de e incluso tener compatibilidad con marcos de JavaScript integrados como ASP.NET AJAX.

Pero el puente de dos plataformas y ecosistemas diferentes (.NET y JavaScript) se incluye con un costo. La experiencia es necesaria en dos mundos en paralelo con diferentes lenguajes, marcos y herramientas. El código y la lógica no se pueden compartir fácilmente entre el cliente y el servidor, lo que produce una sobrecarga de ingeniería y duplicación. También puede ser difícil mantenerse al día con el ecosistema de JavaScript, que tiene un historial de evolución a velocidad vértigo. Las preferencias del marco front-end y de la herramienta de compilación cambian rápidamente. El sector ha observado la progresión desde la imparte a Gulp a WebPack, etc. Se ha producido la misma renovación de Restless con Marcos front-end como jQuery, Knockout, angular, reAct y Vue. Pero dado el monopolio del explorador de JavaScript, había pocas opciones en la materia. Es decir, hasta que la comunidad web se reunió y causó que se produjera un *Miracle*.

## Webassembly satisface una necesidad

En 2015, los principales proveedores del explorador se han unido a las fuerzas de un grupo de la comunidad de W3C para crear un nuevo estándar abierto Web 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 la velocidad nativa. Esfuerzos iniciales centrados en CC++/. El resultado era una demostración dramática de la ejecución directa de motores de gráficos 3D nativos en el explorador sin complementos. Webassembly se ha estandarizado e implementado por todos los exploradores principales.

El trabajo en ejecución de .NET en webassembly se anunció a finales de 2017 y se espera que se distribuya en 2020, incluido el soporte técnico de .NET 5. La capacidad de ejecutar código .NET directamente en el explorador habilita el desarrollo web de pila completa con .NET.

## Increíble: desarrollo web de pila completa 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. Aquí es donde entra más increíble. Blazor es una plataforma de interfaz de usuario web del lado cliente basada en C#, en lugar de JavaScript. El increíble puede ejecutarse directamente en el explorador a través de webassembly. No se requiere ningún complemento de explorador. Como alternativa, las aplicaciones increíbles pueden ejecutar el lado servidor en .NET Core y administrar todas las interacciones del usuario a través de una conexión en tiempo real con el explorador.

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

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

Increíble es mucho en común con los formularios Web Forms de ASP.NET. 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. El cliente puede ejecutarse en el explorador. Sin embargo, si procede de un fondo de formularios Web Forms ASP.NET, hay mucho más increíble que le resultará familiar. Extraordinariamente es una solución natural para los desarrolladores de formularios Web Forms de ASP.NET que buscan una forma de aprovechar el desarrollo del lado cliente y el futuro de .NET multiplataforma de código abierto.

En este libro se proporciona una introducción a la Extraordinariaidad que se ofrece específicamente a los desarrolladores de formularios Web Forms de ASP.NET. Cada concepto extraordinaria se presenta en el contexto de las características y prácticas de formularios Web Forms análogos de ASP.NET. Al final de este libro, tendrá conocimientos sobre:

- Cómo crear aplicaciones increíbles.
- Cómo funciona más extraordinariamente.
- Cómo se relaciona con .NET Core.
- Estrategias razonables para migrar aplicaciones de formularios Web Forms de ASP.NET existentes a extraordinarias, cuando sea necesario.

## Introducción a más increíble

Es fácil empezar a trabajar con extraordinarias. Vaya a https://blazor.net y siga los vínculos para instalar las plantillas de proyecto de SDK de .NET Core y extraordinarias adecuadas. También encontrará instrucciones para configurar las herramientas de extraordinarias en Visual Studio o Visual Studio Code.

# Comparación de la arquitectura de los formularios Web Forms de ASP.NET y el increíble

23/11/2019 • 7 minutes to read • Edit Online

**IMPORTANT**

**EDICIÓN EN VERSIÓN PRELIMINAR**

En este artículo se proporciona contenido anticipado de un libro que está actualmente en elaboración. Si tiene algún comentario, envíelo en https://aka.ms/ebookfeedback.

Aunque los formularios Web Forms y el increíble ASP.NET tienen muchos conceptos similares, existen diferencias en el modo en que funcionan. En este capítulo se examinan los trabajos internos y las arquitecturas de los formularios Web Forms de ASP.NET y el increíbles.

## Formularios Web Forms de ASP.NET

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

Las páginas se componen de los siguientes componentes:

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

Los controles son unidades reutilizables de la interfaz de usuario Web que se pueden colocar e interactuar mediante programación en una página. Las páginas se componen de archivos que terminan en *.aspx* y contienen marcado, controles y código. Las clases de código subyacente se encuentran en archivos con el mismo nombre base y una extensión. *aspx.CS* o *.aspx.VB*, según el lenguaje de programación utilizado. Curiosamente, 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 se proporcionan como controles de usuario. Un control de usuario se deriva de la clase `UserControl` y tiene una estructura similar a la página. El marcado de los controles de usuario se almacena en un archivo *.ascx*. Una clase de código subyacente adjunta se encuentra en un archivo *.ascx.CS* o *.ascx.VB*. Los controles también se pueden compilar completamente con código, heredando de la clase base `WebControl` o `CompositeControl`.

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

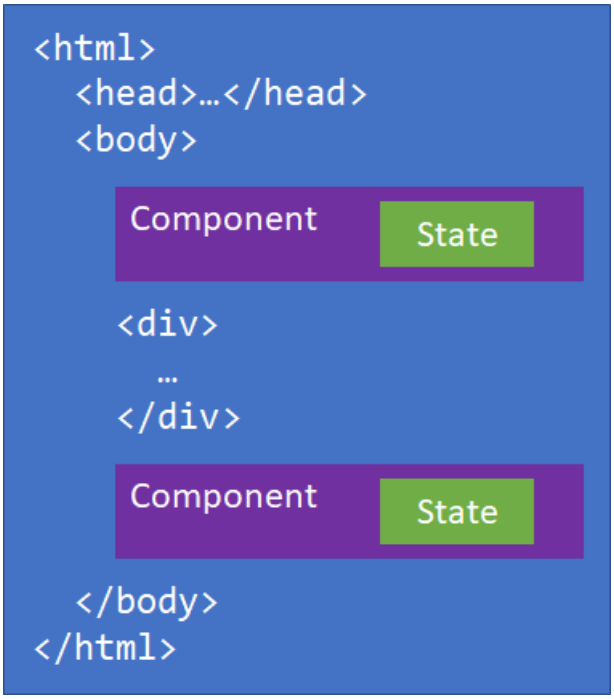
Normalmente, los controles de una página se reenvían a la misma página que presentó el control y llevan una carga de un campo de formulario oculto denominado `ViewState`. El campo `ViewState` contiene información sobre el estado de los controles en el momento en que se representaron y se presentaron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

servidor.

## Blazor

Increíble es un marco de interfaz de usuario Web del lado cliente similar en naturaleza a las plataformas de front-end de JavaScript como angular o reAct. Increíblemente controla las interacciones del usuario y representa las actualizaciones necesarias de la interfaz de usuario. El increíble *no se* basa en un modelo de solicitud-respuesta. Las interacciones del usuario se controlan como eventos que no están en el contexto de una solicitud HTTP determinada.

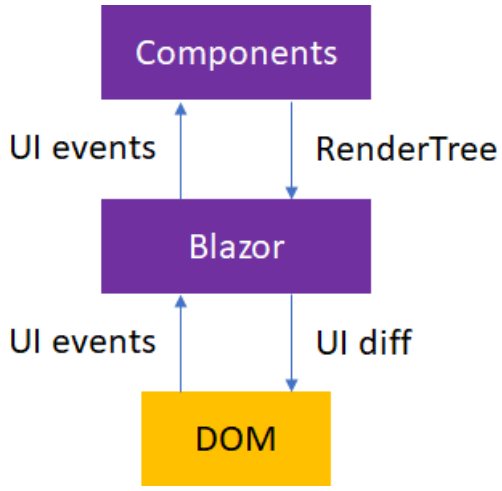
Las aplicaciones increíbles se componen de uno o más componentes raíz que se representan en una página HTML.




La forma en que el usuario especifica dónde se deben representar los componentes y cómo se conectan los componentes para las interacciones de usuario es específico del modelo de hospedaje .

Los componentes increíbles son clases .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 de especifican los controladores de eventos para interacciones de usuario específicas para actualizar el estado del componente.

Después de que un componente controle un evento, el increíble procesador representa el componente y realiza un seguimiento de lo que ha cambiado en la salida representada. Los componentes no se representan directamente en el Document Object Model (DOM). En su lugar, se representan en una representación en memoria del DOM denominada un `RenderTree` de modo que el increíblemente puede realizar el seguimiento de los cambios. Increíbles compara la salida recién representada con la salida anterior para calcular una diferencia de interfaz de usuario que, a continuación, se aplica de forma eficaz al DOM.




Los componentes también pueden indicar manualmente que se deben representar si su estado cambia fuera de un evento normal de la interfaz de usuario. Increíble utiliza un `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 evento que se producen con el método increíblemente se ejecutan en este `SynchronizationContext`.

ANTERIOR SIGUIENTE

# Blazor app hosting modelos

17/04/2020 • 12 minutes to read • Edit Online

**IMPORTANT**

**EDICIÓN EN VERSIÓN PRELIMINAR**

En este artículo se proporciona contenido anticipado de un libro que está actualmente en elaboración. Si tiene algún comentario, envíelo en https://aka.ms/ebookfeedback.

Las aplicaciones Blazor se pueden hospedar en IIS al igual que ASP.NET aplicaciones de formularios Web Forms. Las aplicaciones de Blazor también se pueden hospedar de una de las siguientes maneras:

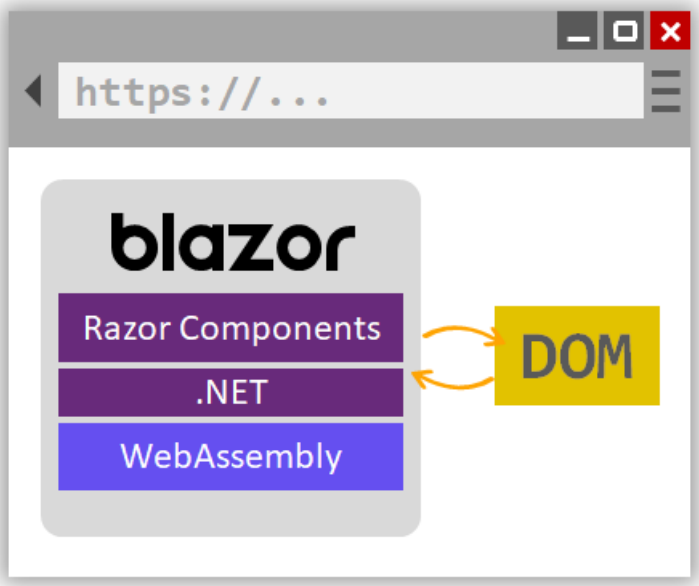
- El lado del cliente en el explorador en WebAssembly.
- Servidor en una aplicación ASP.NET Core.

## Aplicaciones Blazor WebAssembly

Las aplicaciones Blazor WebAssembly se ejecutan directamente en el explorador en un tiempo de ejecución de .NET basado en WebAssembly. Las aplicaciones Blazor WebAssembly 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 tiempo de ejecución de .NET se descarga con la aplicación junto con el ensamblado de la aplicación y las dependencias necesarias. No se requieren plugins o extensiones del navegador.

Los ensamblados descargados son ensamblados normales de .NET, como usaría en cualquier otra aplicación .NET. Dado que el tiempo de ejecución admite .NET Standard, puede usar bibliotecas de .NET Standard existentes con la aplicación Blazor WebAssembly. Sin embargo, estos ensamblados se seguirán ejecutando en el entorno limitado de seguridad del explorador. Algunas funciones PlatformNotSupportedExceptionpueden producir un , como intentar acceder al sistema de archivos o abrir conexiones de red arbitrarias.

Cuando se carga la aplicación, el tiempo de ejecución de .NET se inicia y apunta al ensamblado de la aplicación. Se ejecuta la lógica de inicio de la aplicación 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 DE DOM.



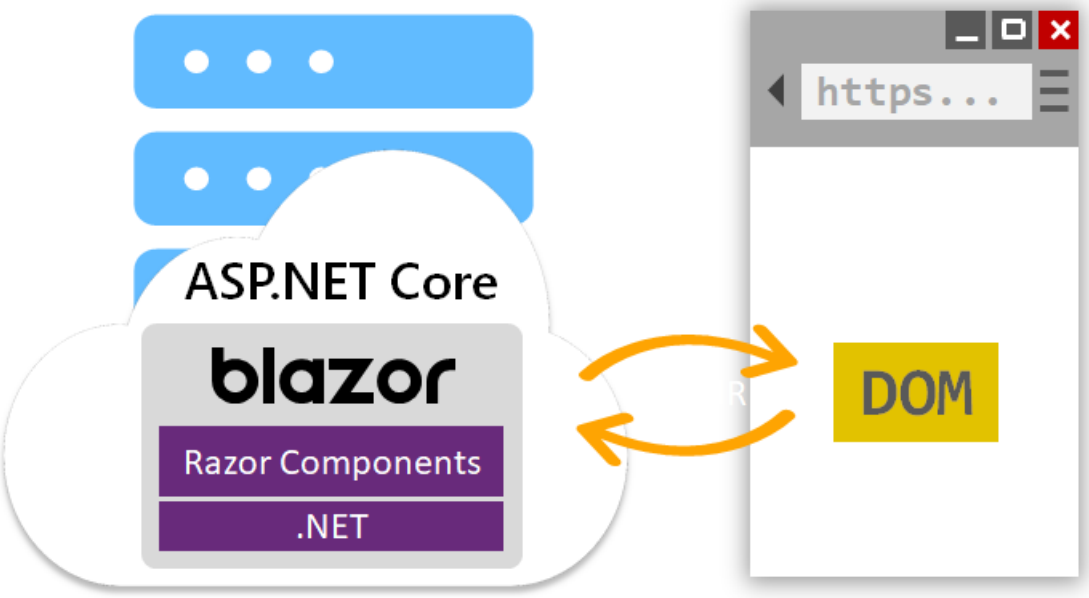

Las aplicaciones Blazor WebAssembly se ejecutan puramente en el lado del cliente. Estas aplicaciones se pueden implementar en soluciones de hospedaje de sitios estáticos como Páginas de GitHub o Hosting de sitios web estáticos de Azure. .NET no es necesario en el servidor en absoluto. La vinculación 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, esta redirección se puede controlar mediante reglas de reescritura de direcciones URL en IIS.

Para obtener todas las ventajas de Blazor y el desarrollo web de .NET de pila completa, hospede la aplicación Blazor WebAssembly 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 convenientes para configurar una solución que contiene una aplicación Blazor WebAssembly y un proyecto host ASP.NET Core. Cuando se compila la solución, los archivos estáticos compilados de la aplicación Blazor se hospedan en la aplicación ASP.NET Core con enrutamiento de reserva ya configurado.

## Aplicaciones de Blazor Server

Recuerde de la discusión de la arquitectura Blazor que los componentes de Blazor representan su salida a una abstracción intermedia llamada `RenderTree` un archivo . A continuación, el marco blazor compara lo que se representó con lo que se representó anteriormente. Las diferencias se aplican al DOM. Los componentes blazor se desacoplan de cómo se aplica su salida renderizada. Por lo tanto, los propios componentes no tienen que ejecutarse en el mismo proceso que el proceso de actualización de la interfaz de usuario. De hecho, ni siquiera tienen que funcionar en la misma máquina.

En las aplicaciones de Blazor Server, 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 distribuyen 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 de Blazor Server puede sonar UpdatePanel familiar si ha utilizado ASP.NET AJAX y el control. El `UpdatePanel` control controla la aplicación de actualizaciones parciales de página en respuesta a eventos de desencadenador en la página. Cuando se `UpdatePanel` desencadena, las solicitudes de una actualización parcial y, a continuación, la aplica sin necesidad de actualizar la página. El estado de la `ViewState` interfaz de usuario se administra mediante . Las aplicaciones de Blazor Server son ligeramente diferentes en que la aplicación requiere una conexión activa con el cliente. Además, todo el estado de la interfaz de usuario se mantiene en el servidor. Aparte de esas diferencias, los dos modelos son conceptualmente similares.

## Cómo elegir el modelo de alojamiento Blazor adecuado

Como se describe en los documentos del modelo de alojamiento blazor, los diferentes modelos de alojamiento Blazor tienen diferentes compensaciones.

El modelo de alojamiento Blazor WebAssembly tiene las siguientes ventajas:

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

Las desventajas del modelo de alojamiento Blazor WebAssembly son:

- Las capacidades del explorador restringen la aplicación.
- Se requiere hardware y software de cliente capaz (por ejemplo, compatibilidad con WebAssembly).
- El tamaño de descarga es mayor y las aplicaciones tardan más en cargarse.
- La compatibilidad con el tiempo de ejecución y las herramientas de .NET es menos madura. Por ejemplo, hay limitaciones en la compatibilidad y depuración de .NET Standard.

Por el contrario, el modelo de alojamiento de Blazor Server ofrece las siguientes ventajas:

- El tamaño de descarga es mucho más pequeño que una aplicación del lado cliente, y la aplicación se carga mucho más rápido.
- La aplicación aprovecha al máximo las capacidades del servidor, incluido el uso de cualquier API compatible con .NET Core.

- .NET Core en el servidor se usa para ejecutar la aplicación, por lo que las herramientas de .NET existentes, como la depuración, funcionan según lo esperado.
- Se admiten clientes ligeros. Por ejemplo, las aplicaciones del lado servidor funcionan con exploradores que no admiten WebAssembly y en dispositivos con recursos limitados.
- La base de código de la aplicación en .NET/C, incluido el código de componente de la aplicación, no se sirve a los clientes.

Las desventajas del modelo de alojamiento de Blazor Server son:

- Mayor latencia de la interfaz de usuario. Cada interacción del usuario implica un salto de red.
- No hay soporte fuera de línea. Si se produce un error en la conexión de cliente, la aplicación deja de funcionar.
- La escalabilidad es un reto 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 servir la aplicación. Los escenarios de implementación sin servidor no son posibles. Por ejemplo, no puede servir la aplicación desde una red CDN.

La lista anterior de compensaciones puede ser intimidante, pero el modelo de hospedaje se puede cambiar más adelante. Independientemente del modelo de alojamiento de Blazor seleccionado, el modelo de componente es *el mismo*. En principio, los mismos componentes se pueden utilizar con cualquiera de los modelos de hospedaje. El código de la aplicación no cambia; sin embargo, es una buena práctica introducir abstracciones para que sus componentes permanezcan hospedando independientemente del modelo. Las abstracciones permiten que la aplicación adopte más fácilmente un modelo de hospedaje diferente.

## Implementación de la aplicación

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

- Hospedarse en IIS, ya sea como archivos estáticos o como una aplicación ASP.NET Core.
- Aproveche ASP.NET flexibilidad de Core para hospedarse en varias plataformas e infraestructuras de servidores. Por ejemplo, puede alojar una aplicación Blazor utilizando Nginx o Apache en Linux. Para obtener más información acerca de cómo publicar e implementar aplicaciones blazor, consulte la documentación de blazor Hosting e implementación.

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

# Estructura del proyecto para aplicaciones increíbles

15/05/2020 • 18 minutes to read • Edit Online

**IMPORTANT**

**EDICIÓN EN VERSIÓN PRELIMINAR**

En este artículo se proporciona contenido anticipado de un libro que está actualmente en elaboración. Si tiene algún comentario, envíelo en https://aka.ms/ebookfeedback.

A pesar de las diferencias importantes de la estructura del proyecto, los formularios Web Forms de ASP.NET y el increíbles comparten muchos conceptos similares. Aquí veremos la estructura de un proyecto increíble y lo comparamos con un proyecto de formularios Web Forms ASP.NET.

Para crear su primera aplicación increíble, siga las instrucciones de los pasos de introducción más rápido. Puede seguir las instrucciones para crear una aplicación de servidor más brillante o una aplicación webassembly increíblemente 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 de servidor increíbles son proyectos de .NET Core. El archivo de proyecto de la aplicación de servidor de extraordinariamente es tan sencillo como puede obtener:

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

  <PropertyGroup>
    <TargetFramework>netcoreapp3.0</TargetFramework>
  </PropertyGroup>

</Project>
```

El archivo de proyecto de una aplicación de webassembly increíblemente es algo más complicado (los números de versión exactos pueden variar):

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

  <PropertyGroup>
    <TargetFramework>netstandard2.0</TargetFramework>
    <RazorLangVersion>3.0</RazorLangVersion>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.Blazor" Version="3.1.0" />
    <PackageReference Include="Microsoft.AspNetCore.Blazor.Build" Version="3.1.0" PrivateAssets="all" />
    <PackageReference Include="Microsoft.AspNetCore.Blazor.HttpClient" Version="3.1.0" />
    <PackageReference Include="Microsoft.AspNetCore.Blazor.DevServer" Version="3.1.0" PrivateAssets="all" />
  </ItemGroup>

  <ItemGroup>
    <ProjectReference Include="..\Shared\BlazorWebAssemblyApp1.Shared.csproj" />
  </ItemGroup>

</Project>
```

Los proyectos de webassembly increíbles tienen como destino .NET Standard en lugar de .NET Core porque se ejecutan en el explorador en un entorno de tiempo de ejecución .NET basado en webassembly. No se puede instalar .NET en un explorador Web como puede hacerlo en un servidor o un equipo del desarrollador. Por consiguiente, el proyecto hace referencia al marco de trabajo de extraordinarias mediante referencias de paquetes individuales.

Por comparación, un proyecto de formularios Web Forms de ASP.NET predeterminado incluye casi 300 líneas de XML en su archivo *. csproj* , la mayoría de las cuales está enumerando explícitamente los distintos archivos de código y contenido en el proyecto. Muchas de las simplificaciones en los proyectos basados en .NET Core y .NET Standard proceden de los destinos y propiedades predeterminados importados mediante referencia al `Microsoft.NET.Sdk.Web` SDK, que a menudo se conoce como el SDK Web. El SDK de web incluye caracteres comodín y otras ventajas que simplifican la inclusión de archivos de código y contenido en el proyecto. No es necesario enumerar los archivos explícitamente. Cuando el destino es .NET Core, el SDK de web también agrega referencias de marco de trabajo a .NET Core y ASP.NET Core Marcos compartidos. Los marcos de trabajo son visibles desde el nodo marcos de **dependencias** > **Frameworks** de la ventana de **Explorador de soluciones** . Los marcos compartidos son colecciones de ensamblados que se instalaron en el equipo al instalar .NET Core.

Aunque se admiten, las referencias de ensamblado individuales son menos comunes en los proyectos de .NET Core. La mayoría de las dependencias del proyecto se administran como referencias de paquetes NuGet. Solo necesita hacer referencia a las dependencias de paquete de nivel superior en los proyectos de .NET Core. Las dependencias transitivas se incluyen automáticamente. En lugar de usar el archivo *packages. config* que se encuentra normalmente en los proyectos de formularios web forms de ASP.net para hacer referencia a los paquetes, las referencias de paquete se agregan al archivo de proyecto mediante el `<PackageReference>` elemento.

```
<ItemGroup>
  <PackageReference Include="Newtonsoft.Json" Version="12.0.2" />
</ItemGroup>
```

## Punto de entrada

El punto de entrada de la aplicación de servidor de increíbles se define en el archivo *Program.CS* , como se verá en una aplicación de consola. Cuando se ejecuta la aplicación, crea y ejecuta una instancia de host web con los valores predeterminados específicos de Web Apps. El host Web administra el ciclo de vida de la aplicación de servidor increíble 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.

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

Las aplicaciones de webassembly increíbles también definen un punto de entrada en *Program.CS*. El código tiene un aspecto ligeramente diferente. 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 aplicación de webassembly no configura un servidor HTTP porque se ejecuta directamente en el explorador.

Las aplicaciones increíbles tienen una `Startup` clase en lugar de un archivo *global. asax* para definir la lógica de inicio de la aplicación. La `Startup` clase se usa para configurar la aplicación y los servicios específicos de la aplicación. En la aplicación de servidor de extraordinarias, la `Startup` clase se usa para configurar el punto de conexión para la conexión en tiempo real utilizada por el increíbles entre los exploradores cliente y el servidor. En la aplicación webassembly de extraordinarias, la `Startup` clase define los componentes raíz de la aplicación y dónde se deben representar. Veremos más en profundidad la `Startup` clase en la sección de inicio de la aplicación .

## Archivos estáticos

A diferencia de los proyectos de formularios Web Forms de ASP.NET, no todos los archivos de un proyecto más increíblemente 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. Cualquier cosa fuera de la raíz Web de la aplicación *no es* direccionable por Web. Este programa de instalación proporciona un nivel de seguridad adicional que evita la exposición accidental de archivos de proyecto a través de la Web.

## Configuración

La configuración de las aplicaciones de formularios Web Forms de ASP.NET se controla normalmente mediante uno o más archivos *Web. config* . Las aplicaciones increíbles no suelen tener archivos *Web. config* . Si lo hacen, el archivo solo se utiliza para configurar los valores específicos de IIS cuando se hospedan en IIS. En su lugar, las aplicaciones de servidor increíbles usan las abstracciones de configuración de ASP.NET Core (las aplicaciones webassembly increíblemente no admiten actualmente las mismas abstracciones de configuración, pero pueden ser una característica agregada en el futuro). Por ejemplo, la aplicación de servidor increíblemente predeterminada almacena algunos valores en *appSettings. JSON*.

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

Más información sobre la configuración en proyectos de ASP.NET Core en la sección de configuración .

## Componentes de Razor

La mayoría de los archivos de los proyectos más increíbles 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 la mayoría de los casos, los componentes son idénticos para las aplicaciones de servidor y de webassembler increíblemente. Los componentes de extraordinariamente son análogos a los controles de usuario de formularios Web Forms de ASP.NET.

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. Veremos la creación de componentes en la sección creación de componentes de interfaz de usuario reutilizables con más increíble.

Los archivos *_Imports. Razor* no son archivos de componentes de Razor. En su lugar, definen un conjunto de directivas de Razor para 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 `using` directivas para los espacios de nombres usados comúnmente:

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

## Páginas

¿Dónde están las páginas de las aplicaciones increíbles? El increíble no define una extensión de archivo independiente para las páginas direccionables, como los archivos *. aspx* de las aplicaciones de formularios Web Forms de ASP.net. En su lugar, las páginas se definen asignando rutas a los componentes. Normalmente, una ruta se asigna mediante la `@page` Directiva Razor. Por ejemplo, el `Counter` componente creado en el archivo *pages/Counter. Razor* define la ruta siguiente:

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

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

Las rutas del componente no se deducen actualmente por la ubicación del archivo del componente, como sucede con las páginas *. aspx* . Esta característica se puede Agregar en el futuro. Cada ruta debe especificarse explícitamente en el componente. El almacenamiento de componentes enrutables en una carpeta *pages* no tiene ningún significado especial y es meramente una Convención.

En la sección páginas, enrutamiento y diseños veremos más detalladamente el enrutamiento en el nivel de detalle.

## Diseño

En las aplicaciones de formularios Web Forms de ASP.NET, el diseño de página común se controla mediante

páginas maestras (*site. Master*). En las aplicaciones increíbles, el diseño de página se controla mediante componentes de diseño (*Shared/MainLayout. Razor*). Los componentes de diseño se tratarán con más detalle en la sección página, enrutamiento y diseños .

## Arranque rápido

Para arrancar increíble, la aplicación debe:

- Especifique en qué lugar de la página se debe representar el componente raíz (*app. Razor*).
- Agregue el correspondiente script del marco de trabajo.

En la aplicación de servidor de extraordinarias, 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 usar sintaxis Razor para definir una página direccionable por el servidor, de forma muy parecida a una página *. aspx* . El `Html.RenderComponentAsync<TComponent>(RenderMode)` método se utiliza para definir dónde se debe representar un componente de nivel de raíz. La `RenderMode` opción indica la manera en que se debe representar el componente. En la tabla siguiente se describen las opciones admitidas `RenderMode` .

| OPCIÓN | DESCRIPCIÓN |
|---|---|
| `RenderMode.Server` | Se representa de forma interactiva una vez que se establece una conexión con el explorador |
| `RenderMode.ServerPrerendered` | Primer prerepresentado y después representado de forma interactiva |
| `RenderMode.Static` | Se representa como contenido estático |

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

```
@page "/"
@namespace BlazorApp1.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" />
    <title>BlazorApp1</title>
    <base href="~/" />
    <link rel="stylesheet" href="css/bootstrap/bootstrap.min.css" />
    <link href="css/site.css" rel="stylesheet" />
</head>
<body>
    <app>
        @(await Html.RenderComponentAsync<App>(RenderMode.ServerPrerendered))
    </app>

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

En la aplicación increíblemente webassembly, la página host es un simple archivo HTML estático en *wwwroot/index.html*. El `<app>` elemento se utiliza para indicar dónde se debe representar el componente raíz.

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width" />
    <title>BlazorApp2</title>
    <base href="/" />
    <link href="css/bootstrap/bootstrap.min.css" rel="stylesheet" />
    <link href="css/site.css" rel="stylesheet" />
</head>
<body>
    <app>Loading...</app>

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

El componente específico que se va a representar se configura en el método de la aplicación `Startup.Configure` con un selector de CSS correspondiente que indica dónde se debe representar el componente.

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
    }

    public void Configure(IComponentsApplicationBuilder app)
    {
        app.AddComponent<App>("app");
    }
}
```

## Resultado de la compilación

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

## Ejecución la aplicación

Para ejecutar la aplicación de servidor de extraordinarias, presione `F5` en Visual Studio. Las aplicaciones increíbles no admiten la compilación en tiempo de ejecución. Para ver los resultados de los cambios de código y marcado de componentes, vuelva a compilar y reiniciar la aplicación con el depurador asociado. Si se ejecuta sin el depurador adjunto ( `Ctrl+F5` ), Visual Studio inspecciona los cambios de archivo y reinicia la aplicación a medida que se realizan cambios. Actualice manualmente el explorador a medida que se realicen cambios.

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

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

Las aplicaciones de webassembly increíbles no admiten la depuración con Visual Studio. Para ejecutar la aplicación, use `Ctrl+F5` en lugar de `F5` . En su lugar, puede depurar aplicaciones webassembly increíblemente directamente en el explorador. Consulte depuración de ASP.net Core extraordinarias para obtener más información.

ANTERIOR SIGUIENTE

# Inicio de la aplicación

23/11/2019 • 2 minutes to read • Edit Online

**IMPORTANT**

**EDICIÓN EN VERSIÓN PRELIMINAR**

En este artículo se proporciona contenido anticipado de un libro que está actualmente en elaboración. Si tiene algún comentario, envíelo en https://aka.ms/ebookfeedback.

*Este contenido estará disponible próximamente.*

# Cree componentes de interfaz de usuario reutilizables con un increíble

13/05/2020 • 32 minutes to read • Edit Online

**IMPORTANT**

**EDICIÓN EN VERSIÓN PRELIMINAR**

En este artículo se proporciona contenido anticipado de un libro que está actualmente en elaboración. Si tiene algún comentario, envíelo en https://aka.ms/ebookfeedback.

Una de las cosas atractivas sobre los formularios Web Forms de ASP.NET es cómo permite la encapsulación de partes reutilizables del código de la interfaz de usuario (IU) en controles de interfaz de usuario reutilizables. Los controles de usuario personalizados se pueden definir en el marcado mediante archivos *.ascx*. También puede crear controles de servidor elaborados en el código con compatibilidad completa con el diseñador.

Increíbles también admite la encapsulación de la interfaz de usuario a través de *componentes*. Un componente:

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

## Introducción a Razor

Razor es un lenguaje de plantillas de marcado ligero basado en HTML y C#. Con Razor, puede realizar una transición sin problemas entre código de marcado y C# para definir la lógica de representación de componentes. Cuando se compila el archivo *.Razor*, la lógica de representación se captura de forma estructurada en una clase .net. El nombre de la clase compilada se toma del nombre de archivo *.Razor*. El espacio de nombres se toma del espacio de nombres predeterminado para el proyecto y la ruta de acceso de la carpeta, o bien se puede especificar explícitamente el espacio de nombres mediante la `@namespace` Directiva (más información sobre las directivas de Razor a continuación).

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

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

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

```
<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:

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

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

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

Las directivas de Razor, como las directivas de formularios Web Forms de ASP.NET, controlan muchos aspectos de cómo se compila un componente de Razor. Algunos ejemplos son:

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

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

```
@namespace MyComponentNamespace
```

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

| DIRECTIVA | DESCRIPCIÓN | EJEMPLO | EQUIVALENTES DE FORMULARIOS WEB FORMS |
|---|---|---|---|
| `@attribute` | Agrega un atributo de nivel de clase al componente. | `@attribute [Authorize]` | None |
| `@code` | Agrega miembros de clase al componente. | `@code { ... }` | `<script runat="server">... </script>` |
| `@implements` | Implementa la interfaz especificada. | `@implements IDisposable` | Uso de código subyacente |
| `@inherits` | Hereda de la clase base especificada | `@inherits MyComponentBase` | `<%@ Control Inherits="MyUserControlBase" %>` |
| `@inject` | Inserta un servicio en el componente. | `@inject IJSRuntime JS` | None |
| `@layout` | Especifica un componente de diseño para el componente | `@layout MainLayout` | `<%@ Page MasterPageFile="~/Site.Master" %>` |

| DIRECTIVA | DESCRIPCIÓN | EJEMPLO | EQUIVALENTES DE FORMULARIOS WEB FORMS |
|---|---|---|---|
| `@namespace` | Establece el espacio de nombres del componente. | `@namespace MyNamespace` | None |
| `@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` | Agregar espacio de nombres en *Web. config* |

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

```
@directive(-suffix(:name))(="value")
```

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

| 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 el increíbles ( `@onclick` , `@bind` „ etc `@ref` .) se describen en las secciones siguientes y en los 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 de los formularios Web Forms de ASP.NET y Razor.

| CARACTERÍSTICA | FORMULARIOS WEB FORMS | SINTAXIS | RAZOR | SINTAXIS |
|---|---|---|---|---|

| CARACTERÍSTICA | FORMULARIOS WEB FORMS | SINTAXIS | RAZOR | SINTAXIS |
|---|---|---|---|---|
| Directivas | `<%@ [directive] %>` | `<%@ Page %>` | `@[directive]` | `@page` |
| Bloques de código | `<% %>` | `<% int x = 123; %>` | `@{ }` | `@{ int x = 123; }` |
| Expresiones (Codificado en HTML) | `<%: %>` | `<%:DateTime.Now %>` | Explícito `@` Explícita `@()` | `@DateTime.Now` `@(DateTime.Now)` |
| Comentarios | `<%-- --%>` | `<%-- Commented -- %>` | `@* *@` | `@* Commented *@` |
| Enlace de datos | `<%# %>` | `<%# Bind("Name") %>` | `@bind` | `<input @bind="username" />` |

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

```
@code {
    int count = 0;

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

Dado que Razor se basa en C#, se debe compilar desde un proyecto de C# (. *csproj*). No se pueden compilar archivos . *Razor* desde un proyecto de Visual Basic (. *vbproj*). Todavía puede hacer referencia a proyectos de Visual Basic desde el proyecto más brillante. Lo contrario también es true.

Para obtener una referencia de sintaxis Razor completa, consulte Sintaxis Razor Reference for ASP.net Core.

## Uso de componentes

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

```
<Counter />
```

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

- No use un prefijo de elemento (por ejemplo, `asp:` ).
- No es necesario registrarse en la página o en el *archivo Web. config*.

Piense en los componentes de Razor como en los tipos de .NET, ya que eso es exactamente lo que son. 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 `@using` Directiva:

```
@using MyComponentLib

<Counter />
```

Como se ve en los proyectos increíblemente predeterminados, es habitual colocar `@using` directivas 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, puede especificar un componente con su nombre de tipo completo, como puede hacerlo en C#:

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

## Parámetros del componente

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

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

```
<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 increíble, use un atributo tal como lo haría en los formularios Web Forms de ASP.NET:

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

## Controladores de eventos

Tanto los formularios Web Forms de ASP.NET como el increíbles proporcionan 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 los formularios Web Forms de ASP.NET, se usan controles de servidor HTML para controlar los eventos de interfaz de usuario expuestos por el DOM, o se pueden controlar los

eventos expuestos por los controles de servidor Web. Los eventos se muestran en el servidor a través de solicitudes de reenvío de formulario. Considere el siguiente ejemplo de clic de botón de formularios Web Forms:

*Counter. ascx*

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

*Counter.ascx.cs*

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

En increíble, puede registrar Controladores para eventos de la interfaz de usuario DOM directamente mediante atributos de directiva del formulario `@on{event}` . El `{event}` marcador de posición representa el nombre del evento. Por ejemplo, puede realizar escuchas de clics de botón de la siguiente manera:

```
<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 `MouseEventArgs` argumento, pero no es necesario.

```
<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.

```
@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 siguiente `OnClick` controlador de eventos se ejecuta de forma asincrónica:

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

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

Una vez que se controla un evento, el componente se representa para tener en cuenta los cambios de estado de 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 la asincrónica `Task` . Este modo de ejecución asincrónica proporciona una oportunidad para representar una interfaz de usuario adecuada mientras la asincrónica `Task` todavía está en curso.

```
<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 sus propios eventos definiendo un parámetro de componente de tipo `EventCallback<TValue>` . Las devoluciones de llamada de evento admiten todas las variaciones de los controladores de eventos de la interfaz de usuario DOM: argumentos opcionales, sincrónicos o asincrónicos, grupos de métodos o expresiones lambda.

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

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

## Enlace de datos

Increíbles 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 los formularios Web Forms de ASP.NET para enlazar datos de orígenes de datos a controles de interfaz de usuario. Trataremos el control de los datos de

diferentes orígenes de datos en la sección tratamiento de los datos .

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

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

@code {
    bool isChecked;
}
```

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

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

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

```
<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 enlazar datos, defina un parámetro de devolución de llamada de evento con el mismo nombre que el parámetro enlazable. El sufijo "Changed" se agrega al nombre.

*PasswordBox. 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 `@bind` atributo.

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

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

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

```
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();
}
```

```
@inject AppState AppState

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

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

## Ciclo de vida de los componentes

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

*Counter.ascx.cs*

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

**Inicializado**

Los `OnInitialized` `OnInitializedAsync` métodos y se utilizan para inicializar el componente. Normalmente, un componente se inicializa una vez que se representa por primera vez. Una vez inicializado un componente, se puede representar varias veces antes de que se elimine. El `OnInitialized` método es similar al `Page_Load` evento en las páginas y controles de formularios Web Forms de ASP.net.

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

**OnParametersSet**

`OnParametersSet` `OnParametersSetAsync` Se llama a los métodos y 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 el componente*.

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

**OnAfterRender**

`OnAfterRender` `OnAfterRenderAsync` Se llama a los métodos y una vez que un componente ha terminado de representar. Las referencias de elementos y componentes se rellenan en este punto (más información sobre estos conceptos a continuación). 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.

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

`OnAfterRender` no se llama a y `OnAfterRenderAsync` *al representarse en el servidor*.

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

**IDisposable**

Los componentes increíbles pueden implementar `IDisposable` para desechar los recursos cuando el componente se quita de la interfaz de usuario. Un componente de Razor puede implementar mediante `IDispose` la `@implements` Directiva:

```
@using System
@implements IDisposable

...

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

## Referencias de componentes de captura

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

Para capturar una referencia de componente en extraordinaria, use el `@ref` atributo de directiva. 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.

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

@code {
    MyLoginDialog loginDialog;

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

Cuando se representa el componente primario, el campo se rellena con la instancia del componente secundario. Después, puede llamar a los métodos en la instancia del componente o manipular 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 automáticamente en los momentos correctos.

## Referencias del elemento Capture

Los componentes increíbles pueden capturar referencias a un elemento. A diferencia de los controles de servidor HTML en formularios Web Forms de ASP.NET, no se puede manipular el DOM directamente mediante una referencia de elemento en increíble. La mayoría de las interacciones de DOM se manejan con el algoritmo de diferenciación DOM. Las referencias de elemento capturadas en Increíblementeers son opacas. Sin embargo, se utilizan para pasar una referencia de elemento específica en una llamada de interoperabilidad de JavaScript. Para obtener más información sobre la interoperabilidad de JavaScript, vea ASP.net Core la interoperabilidad de

JavaScriptincreíble.

# Componentes con plantilla

En los formularios Web Forms de ASP.NET, puede crear *controles con plantilla*. Los controles con plantilla 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 de servidor con plantilla es compleja, pero habilita escenarios eficaces para representar los datos de forma personalizable. Entre los ejemplos de controles con plantilla se incluyen `Repeater` y `DataList` .

Los componentes increíbles también se pueden incluir en la plantilla definiendo los parámetros de componente de tipo `RenderFragment` o `RenderFragment<T>` . `RenderFragment` Representa un fragmento de marcado de Razor que se puede representar mediante el componente. Un `RenderFragment<T>` es un fragmento de marcado de Razor que toma un parámetro que se puede especificar cuando se representa el fragmento de representación.

### Contenido secundario

Los componentes increíbles pueden capturar su contenido secundario como `RenderFragment` y representar ese contenido 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; }
}
```

Después, un componente primario puede proporcionar contenido secundario mediante sintaxis Razor normales.

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

### Parámetros de plantilla

Un componente increíblemente con plantilla también puede definir varios parámetros de componente de tipo `RenderFragment` o `RenderFragment<T>` . El parámetro de un `RenderFragment<T>` puede especificarse cuando se invoca. Para especificar un parámetro de tipo genérico para un componente, use la `@typeparam` Directiva Razor.

*SimpleListView. 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; }
}
```

Cuando se usa 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 `Context` atributo en el elemento secundario. Los parámetros de tipo genérico se pueden especificar mediante un atributo que coincida con el nombre del parámetro de tipo. Si es posible, se infiere el parámetro de tipo:

```
<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:

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

## Código subyacente

Normalmente, un componente más rápido se crea en un solo archivo *. Razor* . Sin embargo, también es posible separar el código y el marcado mediante un archivo de código subyacente. Para usar un archivo de componente, agregue un archivo de C# que coincida con el nombre de archivo del archivo de componente, pero con una extensión *. CS* agregada (*Counter.Razor.CS*). Use el archivo de C# para definir una clase base para el componente. Puede asignar a la clase base cualquier cosa que desee, pero es habitual asignar un nombre a la clase igual a la clase de componente, pero con una `Base` extensión agregada ( `CounterBase` ). La clase basada en componente también debe derivarse de `ComponentBase` . Después, en el archivo de componente de Razor, agregue la `@inherits` Directiva para especificar la clase base del componente ( `@inherits CounterBase` ).

*Counter. Razor*

```
@inherits CounterBase

<h1>Counter</h1>

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

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

*Counter.razor.cs*

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

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

La visibilidad de los miembros del componente en la clase base debe estar `protected` o `public` ser visible para la clase de componente.

## Recursos adicionales

Lo anterior no es un tratamiento exhaustivo de todos los aspectos de los componentes increíbles. Para obtener más información sobre cómo crear y usar los componentes de Razor ASP.net Core, consulte la documentación de extraordinarias.

# Páginas, enrutamiento y diseños

23/11/2019 • 12 minutes to read • Edit Online

**IMPORTANT**

**EDICIÓN EN VERSIÓN PRELIMINAR**

En este artículo se proporciona contenido anticipado de un libro que está actualmente en elaboración. Si tiene algún comentario, envíelo en https://aka.ms/ebookfeedback.

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

En increíble, cada página de la aplicación es un componente, que normalmente se define 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. En primer lugar, el explorador realiza una solicitud a la dirección raíz de la aplicación. Un componente de `Router` raíz de la aplicación extraordinariamente controla las solicitudes de navegación de interceptación y el componente correcto.

El increíble también admite la *vinculación profunda*. 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 vínculos profundos que se envían al servidor se enrutan a la aplicación extraordinaria, que luego enruta el lado cliente de la solicitud al componente correcto.

Una página simple de formularios Web Forms de ASP.NET podría contener el marcado siguiente:

*Nombre. aspx*

```
<%@ 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*

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

La página equivalente en una aplicación extraordinaria sería similar a la siguiente:

*Nombre. 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 increíble, cree un componente y agregue la Directiva de Razor `@page` para especificar la ruta del componente. La Directiva `@page` toma un único parámetro, que es la plantilla de ruta que se va a agregar a ese componente.

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

El parámetro de la plantilla de ruta es obligatorio. A diferencia de los formularios Web Forms de ASP.NET, la ruta a un componente increíblemente *no se* deduce de su ubicación de archivo (aunque puede ser una característica agregada 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 llaves. Más increíble enlazará los valores de ruta a los parámetros de componente con el mismo nombre (sin distinción de mayúsculas y minúsculas).

```
@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 a un `int`:

```
@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 el increíbles, consulte restricciones de ruta.

## Componente de enrutador

El componente `Router` controla el enrutamiento en increíble. El componente de `Router` se usa normalmente en el componente raíz de la aplicación (*app. 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 componente `Router` detecta los componentes enrutables en el `AppAssembly` especificado y en la `AdditionalAssemblies` especificada opcionalmente. Cuando el explorador navega, el `Router` intercepta la navegación y representa el contenido de su parámetro `Found` con el `RouteData` extraído si una ruta coincide con la dirección; de lo contrario, el `Router` representa su parámetro `NotFound`.

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

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

## Navegación

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

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

Normalmente, no es posible devolver una respuesta de redirección en increíble. El increíbles no usa un modelo de solicitud-respuesta. Sin embargo, puede desencadenar directamente las navegaciones del explorador, como se puede hacer con JavaScript.

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

- Obtener la dirección del explorador actual

- Obtención de la dirección base
- Navegación de desencadenador
- Recibir una notificación cuando cambie la dirección

Para navegar a una dirección diferente, utilice el método `NavigateTo`:

```
@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 de estado de navegación.

## Direcciones URL base

Si la aplicación increíblemente se implementa en una ruta de acceso base, debe especificar la dirección URL base en los metadatos de la página con la etiqueta `<base>` para la propiedad enrutamiento a trabajo. Si la página host de la aplicación se representa en el servidor mediante Razor, puede usar la sintaxis de `~/` para especificar la dirección base de la aplicación. Si la página host es HTML estático, debe especificar explícitamente la dirección URL base.

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

## Diseño de página

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

*Site. Master*

```
<%@ 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 increíble, se controla el diseño de página mediante componentes de diseño. Los componentes de diseño heredan de `LayoutComponentBase`, que define una única propiedad `Body` de tipo `RenderFragment`, que se puede utilizar para representar el contenido de la página.

*MainLayout. Razor*

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

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

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

```
@layout MainLayout
```

Puede especificar el diseño de todos los componentes de una carpeta y subcarpetas mediante un archivo *_Imports. Razor* . También puede especificar un diseño predeterminado para todas las páginas mediante el componente del enrutador.

Las páginas maestras pueden definir varios marcadores de posición de contenido, pero los diseños de increíble solo tienen una única propiedad de `Body`. Esta limitación de los componentes de diseño de Extraordinariaidad se solucionará en una futura versión.

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

*ChildLayout. Razor*

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

*Index. Razor*

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

La salida representada de la página sería:

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

Normalmente, los diseños de más increíblemente no definen los elementos HTML raíz de una página ( `<html>`, `<body>`, `<head>`, etc.). En su lugar, los elementos HTML raíz se definen en la página del host de una aplicación, que se usa para representar el contenido HTML inicial de la aplicación (consulte bootstrap). La página host puede representar varios componentes raíz para la aplicación con el marcado circundante.

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

# Administración de estado

23/11/2019 • 2 minutes to read • Edit Online

**IMPORTANT**

**EDICIÓN EN VERSIÓN PRELIMINAR**

En este artículo se proporciona contenido anticipado de un libro que está actualmente en elaboración. Si tiene algún comentario, envíelo en https://aka.ms/ebookfeedback.

*Este contenido estará disponible próximamente.*

ANTERIOR SIGUIENTE

# Formularios y validación

23/11/2019 • 5 minutes to read • Edit Online

**IMPORTANT**

**EDICIÓN EN VERSIÓN PRELIMINAR**

En este artículo se proporciona contenido anticipado de un libro que está actualmente en elaboración. Si tiene algún comentario, envíelo en https://aka.ms/ebookfeedback.

El marco de trabajo de formularios Web Forms de ASP.NET incluye un conjunto de controles de servidor de validación que controlan la validación de los datos especificados por el usuario en un formulario ( `RequiredFieldValidator` , `CompareValidator` , `RangeValidator` , etc.). El marco de trabajo de formularios Web Forms de ASP.NET también admite el enlace de modelos y la validación del modelo en función de las anotaciones de datos ( `[Required]` , `[StringLength]` , `[Range]` , etc.). La lógica de validación se puede aplicar tanto en el servidor como en el cliente mediante la validación basada en JavaScript discreta. El control de servidor `ValidationSummary` se usa para mostrar un resumen de los errores de validación al usuario.

Increíblemente es compatible con 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 la lógica de validación específica de la aplicación. Se pueden usar los mismos tipos de modelo, anotaciones de datos y lógica de validación tanto en el servidor como en el cliente.

Increíbles proporciona un conjunto de componentes de entrada. Los componentes de entrada controlan los datos de campo de enlace a un modelo y validan la entrada del usuario cuando se envía el formulario.

| COMPONENTE DE ENTRADA | ELEMENTO HTML REPRESENTADO |
|---|---|
| `InputCheckbox` | `<input type="checkbox">` |
| `InputDate` | `<input type="date">` |
| `InputNumber` | `<input type="number">` |
| `InputSelect` | `<select>` |
| `InputText` | `<input>` |
| `InputTextArea` | `<textarea>` |

El componente `EditForm` ajusta estos componentes de entrada y organiza el proceso de validación a través de un `EditContext` . Al crear una `EditForm` , se especifica la instancia de modelo a la que se va a enlazar mediante el parámetro `Model` . La validación se realiza normalmente con anotaciones de datos y es extensible. Para habilitar la validación basada en anotaciones de datos, agregue el componente `DataAnnotationsValidator` como un elemento secundario del `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.

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

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

```
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 muestra cómo crear un formulario en extraordinariamente basado en el tipo de modelo de `Starship`:

```
<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 aplicación increíblemente webassembly, los datos se deben enviar al servidor. Por ejemplo, mediante una solicitud HTTP POST. En una aplicación de servidor más brillante, los datos ya están en el servidor, pero debe conservarse. Controlar el acceso a los datos en aplicaciones increíbles es el asunto de la sección tratamiento de los datos .

## Recursos adicionales

Para obtener más información sobre los formularios y la validación en aplicaciones increíbles, consulte la documentación de extraordinarias.

ANTERIOR SIGUIENTE

# Trabajar con datos

11/06/2020 • 12 minutes to read • Edit Online

**IMPORTANT**

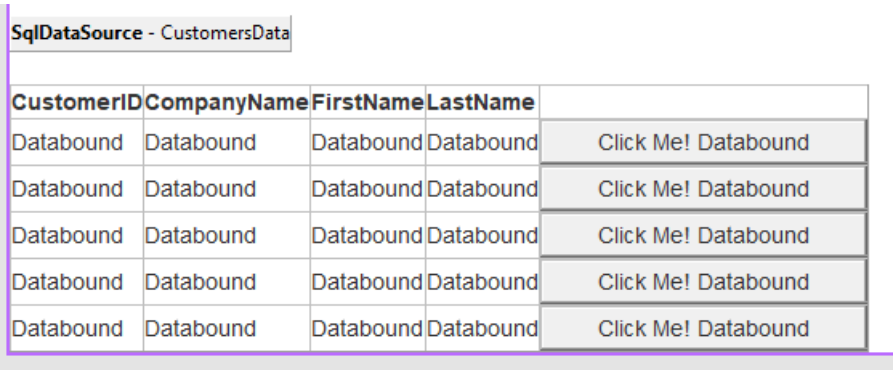
**EDICIÓN EN VERSIÓN PRELIMINAR**

En este artículo se proporciona contenido anticipado de un libro que está actualmente en elaboración. Si tiene algún comentario, envíelo en https://aka.ms/ebookfeedback.

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

- Orígenes de datos
- ADO.NET
- Entity Framework

Los orígenes de datos eran controles que podía colocar en una página de formularios Web Forms y configurarlos como otros controles. Visual Studio proporciona un conjunto descriptivo de cuadros de diálogo para configurar y enlazar los controles a las páginas de formularios Web Forms. Los desarrolladores que disfrutan de un enfoque de "código bajo" o "sin código" prefieren esta técnica cuando se lanzaron por primera vez los formularios Web Forms.




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

## Entity Framework

Entity Framework (EF) es el marco de trabajo de asignación relacional de objetos de código abierto mantenido por .NET Foundation. En la primera versión de .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 Core, 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 que están disponibles para usted mediante la `dotnet ef` herramienta de línea de comandos. Echemos un vistazo

a algunos ejemplos para trabajar con una base de datos.

**Code First EF**

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

```
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 exigir 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 como indica el `[MaxLength]` atributo. El esquema de la base de datos se configurará con una columna denominada `MaxLength` con el tipo de datos `varchar(4000)` .
- La `Price` propiedad se almacenará como moneda. El `[Range]` atributo generará las restricciones adecuadas para evitar el almacenamiento de datos fuera de los valores mínimo y máximo declarados.

Necesitamos 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.

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

La `MyDbContext` clase proporciona una 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 `Startup` método de la clase `ConfigureServices` :

```
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 u otras ubicaciones de almacenamiento de configuración y reemplazar esta cadena incrustada de forma adecuada.

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

comandos:

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

El primer comando define los cambios que se realizan 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 de base de datos.

Una vez aplicado, tiene una `Product` tabla simple 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*. Cuando realice cambios en la `Product` clase o agregue más clases relacionadas que le gustaría interactuar con la base de datos, deberá 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.

**Database First EF**

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

```
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 utilizando la cadena de conexión especificada y el `Microsoft.EntityFrameworkCore.SqlServer` proveedor. Una vez conectado, se crea una clase de contexto de base de datos denominada `MyDbContext` . Además, las clases auxiliares se crean para `Product` las `Customer` tablas y que se especificaron con las `-t` Opciones. Hay muchas opciones de configuración para este comando con el fin de 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.

## Interacción con servicios Web

Cuando ASP.NET se lanzó por primera vez, los servicios SOAP eran el mejor método para que los clientes y servidores web intercambien datos. Ha cambiado mucho desde ese momento y las interacciones preferidas con los servicios se han desplazado a interacciones directas del cliente HTTP. Con ASP.NET Core y extraordinaria, puede registrar la configuración de su `HttpClient` en el `Startup` método de la clase `ConfigureServices` . Use esa configuración cuando necesite interactuar con el punto de conexión HTTP. Considere el siguiente código de configuración:

```
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 tener acceso a los datos de GitHub, cree un cliente con el nombre `github` . El cliente está configurado con la dirección base y los encabezados de solicitud se establecen adecuadamente. Inserte el `IHttpClientFactory` en sus componentes más increíbles con la `@inject` Directiva o un `[Inject]` atributo en una

propiedad. Cree el cliente con nombre e interactúe con los servicios mediante la siguiente sintaxis:

```
@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 = async 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 problema de GitHub correspondientes. Hay muchas maneras de configurar `HttpClientFactory` para proporcionar objetos preconfigurados `HttpClient` . Intente configurar varias `HttpClient` instancias con distintos nombres y puntos de conexión para los diversos servicios web con los que trabaja. Este enfoque hará que sus interacciones con esos servicios sean más fáciles de trabajar en cada página. Para obtener más información, lea la documentación de IHttpClientFactory.

# Módulos, controladores y middleware

06/01/2020 • 5 minutes to read • Edit Online

**IMPORTANT**

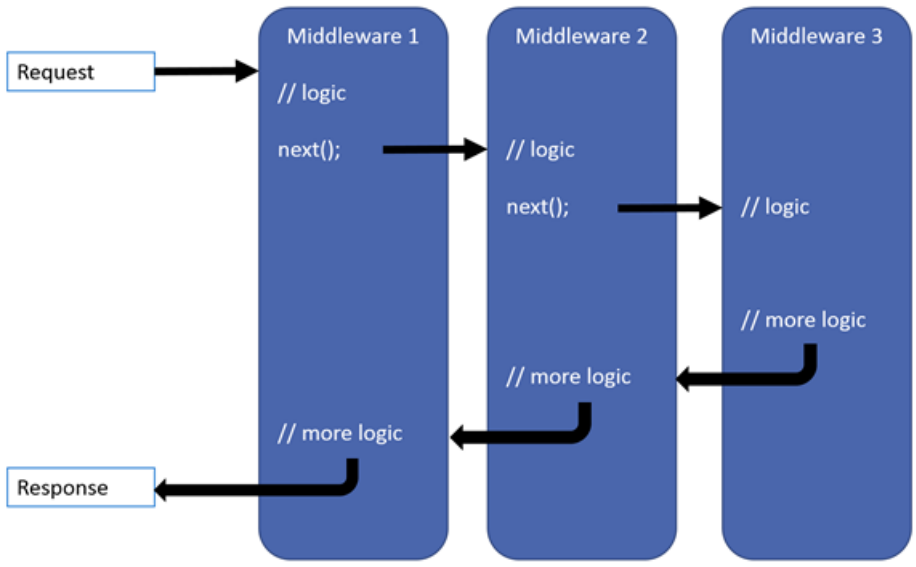
**EDICIÓN EN VERSIÓN PRELIMINAR**

En este artículo se proporciona contenido anticipado de un libro que está actualmente en elaboración. Si tiene algún comentario, envíelo en https://aka.ms/ebookfeedback.

Una aplicación ASP.NET Core se basa en una serie de *middleware*. El middleware es controladores que se organizan en una canalización para controlar solicitudes y respuestas. En una aplicación de formularios Web Forms, los controladores y módulos HTTP solucionan problemas similares. En ASP.NET Core, los módulos, los controladores, *global.asax.CS*y el ciclo de vida de la aplicación se reemplazan por middleware. En este capítulo, aprenderá qué middleware en el contexto de una aplicación increíblemente.

## Información general del

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 el modo en que se administran las solicitudes de formularios Web Forms de ASP.NET. Este sistema facilita la tarea de pensar en qué proceso está ocurriendo y permite insertar middleware en cualquier momento. El middleware se ejecuta en el orden en que se agrega a la canalización de solicitudes. También se agregan en el código en lugar de en los archivos de configuración, normalmente en *Startup.CS*.

## Katana

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

# Middleware común

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

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

| MODULE | MÓDULO ASP.NET 4. X | ASP.NET CORE, OPCIÓN |
|---|---|---|
| Errores HTTP | `CustomErrorModule` | Middleware de páginas de códigos de estado |
| Documento predeterminado | `DefaultDocumentModule` | Middleware de archivos predeterminados |
| Examen de directorios | `DirectoryListingModule` | Middleware de exploración de directorios |
| Compresión dinámica | `DynamicCompressionModule` | Middleware de compresión de respuestas |
| Seguimiento de solicitudes con error | `FailedRequestsTracingModule` | Registro de ASP.NET Core |
| Almacenamiento en caché de archivos | `FileCacheModule` | Middleware de almacenamiento en caché de respuestas |
| Almacenamiento en caché de HTTP | `HttpCacheModule` | Middleware de almacenamiento en caché de respuestas |
| Registro HTTP | `HttpLoggingModule` | Registro de ASP.NET Core |
| Redirección HTTP | `HttpRedirectionModule` | Middleware de reescritura de dirección URL |
| filtros ISAPI | `IsapiFilterModule` | Middleware |
| ISAPI | `IsapiModule` | Middleware |
| Solicitud de filtrado | `RequestFilteringModule` | Middleware de reescritura de URL IRule |
| Reescritura de URL† | `RewriteModule` | Middleware de reescritura de dirección URL |
| Compresión estática | `StaticCompressionModule` | Middleware de compresión de respuestas |
| Contenido estático | `StaticFileModule` | Middleware de archivos estáticos |
| Autorización URL | `UrlAuthorizationModule` | Identidad de ASP.NET Core |

Esta lista no es exhaustiva, pero debe dar una idea de la asignación que existe entre los dos marcos. Para obtener una lista más detallada, vea 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 el middleware, siendo la más simple un delegado simple. Considere el siguiente middleware, que acepta una solicitud de referencia cultural de una cadena de consulta:

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

            if (!string.IsNullOrWhiteSpace(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 implementando la interfaz de `IMiddleware` o mediante la siguiente Convención de middleware. Para obtener más información, consulte escritura de middleware de ASP.net Core personalizado.

ANTERIOR SIGUIENTE

# Configuración de la aplicación

02/04/2020 • 12 minutes to read • Edit Online

**IMPORTANT**

**EDICIÓN EN VERSIÓN PRELIMINAR**

En este artículo se proporciona contenido anticipado de un libro que está actualmente en elaboración. Si tiene algún comentario, envíelo en https://aka.ms/ebookfeedback.

La forma principal de cargar la configuración de la aplicación—en formularios Web Forms es con entradas en el archivo *web.config* en el servidor o un archivo de configuración relacionado al que hace referencia *web.config*. Puede usar el `ConfigurationManager` objeto estático para interactuar con la configuración de la aplicación, las cadenas de conexión del repositorio de datos y otros proveedores de configuración extendida que se agregan a la aplicación. Es típico ver las interacciones con la configuración de la aplicación como se ve en el código siguiente:

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

Con ASP.NET Core y Blazor del lado servidor, el archivo *web.config* PUEDE estar presente si la aplicación está hospedada en un servidor IIS de Windows. Sin embargo, `ConfigurationManager` no hay interacción 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 puede seguir accediendo a la información de configuración desde un archivo *web.config*.

## Orígenes de configuración

ASP.NET Core reconoce que hay muchos orígenes de configuración que puede que desee usar para la aplicación. El marco de trabajo intenta ofrecerle la mejor de estas características de forma predeterminada. La configuración se lee y se agrega de estos diversos orígenes por ASP.NET Core. Los valores cargados posteriormente para la misma clave de configuración tienen prioridad sobre los valores anteriores.

ASP.NET Core se diseñó para ser conscientes de la nube y para facilitar la configuración de aplicaciones tanto para los operadores como para los desarrolladores. ASP.NET Core es compatible con el entorno y `Production` sabe `Development` si se está ejecutando en su entorno o en su entorno. El indicador de entorno `ASPNETCORE_ENVIRONMENT` se establece en la variable de entorno del sistema. Si no se configura ningún valor, el `Production` valor predeterminado de la aplicación se ejecuta en el entorno.

La aplicación puede desencadenar y agregar la configuración de varios orígenes en función del nombre del entorno. De forma predeterminada, la configuración se carga desde los siguientes recursos en el orden indicado:

1. *appsettings.json,* si está presente
2. *appsettings. ENVIRONMENT_NAME Si* está presente, si está presente.
3. Archivo de secretos de usuario en el disco, si está presente
4. Variables de entorno
5. Argumentos de la línea de comandos

## appsettings.json formato y acceso

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

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

Cuando se presenta con el JSON anterior, el sistema de configuración aplana los valores secundarios y hace referencia a sus rutas jerárquicas completas. Un carácter de dos puntos ( `:` ) separa cada propiedad de la jerarquía. Por ejemplo, la `section1:key0` clave `section1` de configuración `key0` tiene acceso al valor del literal de objeto.

## 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 `Development` cuando se ejecuta en el entorno.
- Asociado a una aplicación específica.
- Administrado con el comando de `user-secrets` la CLI de .NET Core.

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

```
dotnet user-secrets init
```

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

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

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

Para obtener más información acerca de cómo crear, almacenar y administrar secretos de usuario, vea el almacenamiento seguro de secretos de aplicación en desarrollo en ASP.NET documento principal.

## Variables de entorno

El siguiente conjunto de valores cargados en la configuración de la aplicación son las variables de entorno del sistema. Todas las opciones de entorno del sistema ahora son accesibles para usted a través de la API de configuración. Los valores jerárquicos se acoplan y se separan por caracteres de dos puntos cuando se leen dentro de la aplicación. Sin embargo, algunos sistemas operativos no permiten los nombres de variables de entorno de caracteres de dos puntos. ASP.NET Core aborda esta limitación convirtiendo valores `__` que tienen guiones dobles ( ) en dos puntos cuando se accede a ellos. El `Parent:ApiKey` valor de la sección de secretos de `Parent__ApiKey` usuario anterior se puede invalidar con la variable de entorno .

## Argumentos de la línea de comandos

La configuración también se puede proporcionar como argumentos de línea de comandos cuando se inicia la aplicación. Utilice la notación `--` de doble guión ( ) o de 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:

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

## El retorno 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 de núcleo de ASP.NET (ANCM). ANCM es un módulo IIS nativo que hospeda la aplicación en lugar del servidor web de Kestrel. Esta sección *web.config* es similar al siguiente marcado 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>
```

La configuración específica de la aplicación `environmentVariables` se `aspNetCore` puede definir anidando un elemento en el elemento. 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.

```
<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 IConfiguration configuración de la aplicación a través de la interfaz. Esta interfaz de configuración debe ser solicitada por los componentes de Blazor, las páginas de Blazor y cualquier otra clase administrada por Core ASP.NET que necesite acceso a la configuración. El marco de trabajo de ASP.NET Core rellenará automáticamente esta interfaz con la configuración resuelta configurada anteriormente. En una página Blazor o en el marcado `IConfiguration` Razor de `@inject` un componente, puede insertar el objeto con una

directiva en la parte superior del archivo *.razor* como esta:

```
@inject IConfiguration Configuration
```

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

Los valores de configuración individuales se pueden leer especificando la jerarquía de valores de configuración buscada como parámetro de indexador:

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

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

## Configuración fuertemente tipada

Con formularios Web Forms, era posible crear un tipo ConfigurationSection de configuración fuertemente tipado que se heredó del tipo y los tipos asociados. Un `ConfigurationSection` permite configurar algunas reglas de negocio y 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 es necesario heredar de una clase primaria.
- Debe `public` incluir propiedades que coincidan con las propiedades y referencias de tipo para la estructura de configuración que desea capturar.

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

```
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 `Startup.ConfigureServices` agregando la siguiente línea al método:

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

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

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

Puede encontrar más información sobre la función Opciones en el patrón Opciones del documento ASP.NET Core.

# Seguridad: autenticación y autorización en formularios Web Forms ASP.NET y extraordinarias

23/11/2019 • 2 minutes to read • Edit Online

**IMPORTANT**

EDICIÓN EN VERSIÓN PRELIMINAR

En este artículo se proporciona contenido anticipado de un libro que está actualmente en elaboración. Si tiene algún comentario, envíelo en https://aka.ms/ebookfeedback.

*Este contenido estará disponible próximamente.*

ANTERIOR SIGUIENTE

# Migración de formularios Web Forms de ASP.NET a increíbles

11/06/2020 • 31 minutes to read • Edit Online

**IMPORTANT**

EDICIÓN EN VERSIÓN PRELIMINAR

En este artículo se proporciona contenido anticipado de un libro que está actualmente en elaboración. Si tiene algún comentario, envíelo en https://aka.ms/ebookfeedback.

La migración de una base de código de formularios Web Forms ASP.NET a un increíble proceso es una tarea que requiere un tiempo de planeamiento. 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, los formularios Web Forms) es independiente de la lógica de negocios. Esta separación lógica de las capas permite borrar lo que necesita para moverse a .NET Core y a increíble.

En este ejemplo, se usa la aplicación de eShop disponible en GitHub . eShop es un servicio de catálogo que proporciona capacidades CRUD mediante la entrada de formulario y la validación.

¿Por qué se debe migrar una aplicación de trabajo a un increíble? Muchas veces, no es necesario. Los formularios Web Forms de ASP.NET seguirán siendo compatibles durante muchos años. Sin embargo, muchas de las características que ofrece más increíble solo se admiten en una aplicación migrada. Estas características incluyen:

- Mejoras de rendimiento en el marco de trabajo como `Span<T>`
- Capacidad de ejecutar como webassembly
- Compatibilidad entre plataformas 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, puede haber un valor en la migración de la aplicación. La migración puede tomar distintas formas; puede ser toda la aplicación o solo determinados puntos de conexión que requieran los cambios. La decisión de migrar se basa en última instancia en los problemas empresariales que el desarrollador debe resolver.

## En el lado servidor frente al hospedaje del lado cliente

Tal y como se describe en el capítulo modelos de hospedaje , una aplicación increíblemente se puede hospedar de dos maneras diferentes: lado servidor y cliente. El modelo de servidor usa ASP.NET Core las conexiones de Signalr para administrar las actualizaciones del DOM mientras se ejecuta cualquier código real en el servidor. El modelo de cliente se ejecuta como webassembly dentro de un explorador y no requiere conexiones de servidor. Hay una serie de diferencias que pueden afectar a lo mejor para una aplicación específica:

- Ejecutar como webassembly está aún en desarrollo y es posible que no admita todas las características (como subprocesamiento) en el momento actual.
- La comunicación entre el cliente y el servidor puede producir problemas de latencia en el modo de servidor
- El acceso a las bases de datos y a los servicios internos o protegidos requiere un servicio independiente con hospedaje en el lado cliente.

En el momento de escribir este documento, el modelo del lado servidor se parece más al de los formularios 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.

## Crear un proyecto nuevo

Este paso de migración inicial consiste en crear un nuevo proyecto. Este tipo de proyecto se basa en los proyectos de estilo SDK de .NET Core y simplifica gran parte del texto reutilizable que se usó en los formatos de proyecto anteriores. Para obtener más información, consulte el capítulo sobre la estructura del proyecto.

Una vez creado el proyecto, instale las bibliotecas que se usaron en el proyecto anterior. En los proyectos de formularios Web Forms anteriores, es posible que haya usado el archivo *packages. config* para enumerar los paquetes de NuGet necesarios. En el nuevo proyecto de estilo SDK, *packages. config* se ha reemplazado por `<PackageReference>` elementos en el archivo de 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 está usando están disponibles para .NET Core, como Entity Framework 6 y log4net. Si no hay ninguna versión de .NET Core o .NET Standard disponible, a menudo se puede usar la versión de .NET Framework. Puede que tu experiencia sea diferente. Cualquier API usada que no esté disponible en .NET Core producirá un error en tiempo de ejecución. Visual Studio le informa de estos paquetes. Aparece un icono amarillo en el nodo **referencias** del proyecto en **Explorador de soluciones**.

En el proyecto de eShop basado en extraordinarias, puede ver los paquetes que están instalados. Anteriormente, el archivo *packages. config* enumeraba todos los paquetes usados en el proyecto, lo que da lugar a un archivo de casi 50 líneas de longitud. Un fragmento de código *packages. config* es:

```
<?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 `<packages>` elemento incluye todas las dependencias necesarias. Es difícil identificar cuál de estos paquetes se incluyen porque los necesita. Algunos `<package>` elementos se enumeran simplemente para satisfacer las necesidades de las dependencias que necesita.

El proyecto increíblemente bajo muestra las dependencias que necesita dentro de un `<ItemGroup>` elemento en el archivo de proyecto:

```
<ItemGroup>
    <PackageReference Include="Autofac" Version="4.9.3" />
    <PackageReference Include="EntityFramework" Version="6.3.0-preview9-19423-04" />
    <PackageReference Include="log4net" Version="2.0.8" />
</ItemGroup>
```

Un paquete NuGet que simplifica la vida de los desarrolladores de formularios Web Forms es el paquete de compatibilidad de Windows. Aunque .NET Core 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 servicios de directorio. El paquete agrega alrededor de 20.000 API y activa muchos servicios con los que es posible que ya esté familiarizado. El proyecto de eShop no requiere el paquete de compatibilidad; pero si los proyectos usan características específicas de Windows, el paquete facilita el trabajo de migración.

## Habilitar proceso de inicio

El proceso de inicio de increíbles ha cambiado de los formularios Web Forms y sigue una configuración similar para otros servicios de ASP.NET Core. Cuando se ejecutan los componentes más increíbles del lado servidor hospedado como parte de una aplicación de ASP.NET Core normal. Cuando se hospeda en el explorador con webassembly, los componentes increíbles usan 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 formularios Web Forms. En el proyecto de 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 requieren la invalidación de servicios específicos a través de la inserción de dependencias (DI).

Por ejemplo, el archivo *global.asax.CS* de eShop contiene el código siguiente:

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

    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://autofaccn.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 `Startup` en la clase del servidor:

```
public class Startup
{
    public Startup(IConfiguration configuration, IWebHostEnvironment env)
    {
        Configuration = configuration;
        Env = env;
    }

    public IConfiguration Configuration { get; }

    public IWebHostEnvironment Env { get; }

    // This method gets called by the runtime. Use this method to add services to the container.
    // For more information on how to configure your application, visit https://go.microsoft.com/fwlink/?
LinkID=398940
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddRazorPages();
        services.AddServerSideBlazor();

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

    // This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
    public void Configure(IApplicationBuilder app, ILoggerFactory loggerFactory)
    {
        loggerFactory.AddLog4Net("log4Net.xml");

        if (Env.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);
    }
```

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

            if (initializer != null)
            {
                Database.SetInitializer(initializer);
            }
        }
    }
}
```

Un cambio importante que puede observar de los formularios Web Forms es el importancia de DI. DI ha sido un principio de GUID en el diseño de ASP.NET Core. Admite la personalización de casi todos los aspectos del marco de ASP.NET Core. Hay incluso un proveedor de servicios integrado que se puede usar para muchos escenarios. Si se requiere más personalización, puede ser compatible con los numerosos proyectos de la comunidad. Por ejemplo, puede llevar a cabo la inversión de la biblioteca de DI de terceros.

En la aplicación de eShop original, hay alguna configuración para la administración de sesiones. Dado que el uso del servidor de ASP.NET Core es más increíble 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 requiere que se rediseñe antes de ejecutarse como una aplicación increíblemente larga.

Para obtener más información sobre el inicio de la aplicación, consulte inicio de la aplicación.

## Migración de módulos y controladores HTTP a middleware

Los módulos y controladores HTTP son patrones comunes en formularios Web Forms para controlar la canalización de solicitudes HTTP. Clases que implementan `IHttpModule` o que `IHttpHandler` se pueden registrar y procesar solicitudes entrantes. Formularios Web Forms configura módulos y controladores en el archivo *Web.config* . Los formularios Web Forms también se basan principalmente 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 `Configure` método de la `Startup` clase. El orden de ejecución del middleware viene determinado por el orden de registro.

En la sección Habilitar proceso de inicio , Web Forms generó un evento de ciclo de vida como el `Application_BeginRequest` método. Este evento no está disponible en ASP.NET Core. Una manera de lograr este comportamiento es implementar el middleware tal como se aprecia en el ejemplo de archivo *Startup.CS* . Este middleware realiza la misma lógica y, a continuación, transfiere el control al siguiente controlador en la canalización de middleware.

Para obtener más información sobre cómo migrar módulos y controladores, consulte migración de controladores y módulos http a middleware de ASP.net Core.

## Migrar archivos estáticos

Para servir archivos estáticos (por ejemplo, HTML, CSS, imágenes y JavaScript), los archivos deben estar expuestos por middleware. La llamada al `UseStaticFiles` método habilita el servicio de archivos estáticos de la ruta de acceso raíz Web. El directorio raíz Web predeterminado es *wwwroot*, pero se puede personalizar. Tal y como se incluye en el `Configure` método de la clase de eShop `Startup` :

```
public void Configure(IApplicationBuilder app)
{
    ...

    app.UseStaticFiles();

    ...
}
```

El proyecto de eShop permite el acceso a archivos estáticos básicos. Hay muchas personalizaciones disponibles para el acceso a archivos estáticos. Para obtener información sobre cómo habilitar archivos predeterminados o un explorador de archivos, consulte archivos estáticos en ASP.net Core.

## Migrar la configuración de agrupación y minificación de tiempo de ejecución

La agrupación y 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. JavaScript y CSS suelen someterse a una forma de agrupación o minificación antes de enviarse al cliente. En los formularios Web Forms de ASP.NET, 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. Un archivo enumera los archivos que se van a reducida, junto con la configuración específica de minificación.

Para obtener más información sobre la agrupación y la minificación, consulte agrupación y minimizar de recursos estáticos en ASP.net Core.

## Migrar páginas ASPX

Una página de una aplicación de formularios Web Forms es un archivo con la extensión *. aspx* . Una página de formularios Web Forms a menudo se puede asignar a un componente en increíble. Un componente increíblemente se crea en un archivo con la extensión *. Razor* . En el caso del proyecto de eShop, se convierten cinco páginas en una página de Razor.

Por ejemplo, la vista de detalles consta de tres archivos en el proyecto de formularios Web Forms: *details. aspx*, *details.aspx.CS*y *details.aspx.Designer.CS*. Al convertir a increíbles, el código subyacente y el marcado se combinan en *details. Razor*. La compilación de Razor (equivalente a lo que se encuentra en los archivos *. Designer.CS* ) se almacena en el directorio *obj* y no es visible de forma predeterminada en **Explorador de soluciones**. La página de formularios Web Forms consta del siguiente marcado:

```
<%@ 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:

```
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 increíble, la página de formularios Web Forms se convierte en el código siguiente:

```
@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">
        <img class="col-md-6 esh-picture" src="@($"/Pics/{_item.PictureFileName}")">

        <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 `@inject` atributo. Según la `@page` Directiva, se puede tener acceso a esta página en la `Catalog/Details/{id}` ruta. El valor del marcador de posición de la ruta se ha `{id}` restringido a un entero. Como se describe en la sección enrutamiento , a diferencia de los formularios Web Forms, un componente Razor indica explícitamente su ruta y los parámetros que se incluyen. Es posible que muchos controles de formularios Web Forms no tengan homólogos exactos en increíble. A menudo hay un fragmento de código HTML equivalente que tendrá el mismo propósito. Por ejemplo, el `<asp:Label />` control se puede reemplazar por un `<label>` elemento HTML.

**Validación de modelos en extraordinarias**

Si el código de formularios Web Forms incluye validación, puede transferir gran parte de lo que tiene con cambios poco a no. Una ventaja de ejecutarse en extraordinariamente 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 código de ejemplo tendría el siguiente aspecto:

```
<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 increíble, el marcado equivalente se proporciona en un archivo *Create. 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 `EditForm` contexto incluye compatibilidad con la validación y se puede ajustar en torno a la entrada. Las anotaciones de datos son una manera común de agregar validación. Esta compatibilidad de validación se puede Agregar a través del `DataAnnotationsValidator` componente. Para obtener más información sobre este mecanismo, vea ASP.net Core las formas y la validaciónde las increíbles.

# Migrar controles de formularios Web Forms integrados

*Este contenido estará disponible próximamente.*

# Migración de la configuración

En un proyecto de formularios Web Forms, los datos de configuración se almacenan normalmente en el archivo *Web. config* . Se tiene acceso a los datos de configuración con `ConfigurationManager` . A menudo, los servicios debían analizar objetos. Con .NET Framework 4.7.2, composición se agregó a la configuración a través de `ConfigurationBuilders` . Estos generadores permitían a los desarrolladores agregar varios orígenes de configuración que luego se componían en tiempo de ejecución para recuperar los valores necesarios.

ASP.NET Core presentó un sistema de configuración flexible que le permite definir el origen de configuración o los orígenes usados por la aplicación y la implementación. La `ConfigurationBuilder` infraestructura que puede usar en la aplicación de formularios Web Forms se modelaba después de los conceptos usados en el sistema de configuración de ASP.net Core.

El siguiente fragmento de código muestra cómo el proyecto de eShop de formularios Web Forms usa *Web. config* para almacenar los valores de configuración:

```
<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 el *archivo Web. config*. Los secretos se conservan inevitablemente en ubicaciones no seguras, como el control de código fuente. Con un increíble ASP.NET Core, la configuración basada en XML anterior se reemplaza por el 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; sin embargo, ASP.NET Core admite muchos otros formatos, incluido XML. También hay varios formatos admitidos por la comunidad.

El constructor de la clase del proyecto extraordinariamente `Startup` acepta una `IConfiguration` instancia a través de una técnica de di conocida como inserción del constructor:

```
public class Startup
{
    public Startup(IConfiguration configuration, IWebHostEnvironment env)
    {
        Configuration = configuration;
        Env = env;
    }

    ...
}
```

De forma predeterminada, las variables de entorno, los archivos JSON (*appSettings. JSON* y *appSettings. { Environment}. JSON*) y las opciones de línea de comandos se registran como orígenes de configuración válidos en el objeto de configuración. Se puede tener acceso a los orígenes de configuración a través 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 de eShop almacena información de catálogo en una base de datos y recupera los datos con Entity Framework (EF) 6. Dado que EF 6 es compatible con .NET Core 3,0, el proyecto puede seguir utilizándolo.

Los siguientes cambios relacionados con EF eran necesarios para eShop:

- En .NET Framework, el `DbContext` objeto acepta una cadena con el formato *Name = ConnectionString* y utiliza la cadena de conexión de `ConfigurationManager.AppSettings[ConnectionString]` para conectarse. No se admite en .NET Core. Se debe proporcionar la cadena de conexión.
- Se ha tenido acceso a la base de datos de forma sincrónica. Aunque esto funciona, la escalabilidad puede verse afectada. Esta lógica debe moverse a un patrón asincrónico.

Aunque no hay la misma compatibilidad nativa para el enlace de conjunto de los conjuntos de los mismos, increíble 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 extraordinarias, consulte el capítulo de acceso a datos .

## Cambios de arquitectura

Por último, hay algunas diferencias arquitectónicas importantes que se deben tener en cuenta al migrar a extraordinarias. Muchos de estos cambios se aplican a cualquier elemento basado en .NET Core o en ASP.NET Core.

Dado que el increíble se basa en .NET Core, existen consideraciones para garantizar la compatibilidad con .NET Core. Algunos de los cambios principales incluyen la eliminación de las siguientes características:

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

Para obtener más información sobre las técnicas para identificar los cambios necesarios para admitir la ejecución en .NET Core, vea portar el 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 principales cambios son:

- Sin contexto de sincronización, lo que significa que no hay ningún `HttpContext.Current` , `Thread.CurrentPrincipal` u otros descriptores de acceso estáticos
- Sin copias sombra
- Sin cola de solicitudes

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

## Conclusión de la migración

En este punto, ha visto muchos ejemplos de lo que se necesita para trasladar un proyecto de formularios Web Forms a un increíble. Para obtener un ejemplo completo, vea el proyecto eShopOnBlazor .

ANTERIOR