

v2.0 EDICIÓN

Descarga disponible en: <https://aka.ms/webappebook>

PUBLICADO POR

DevDiv, .NET y equipos de producto de Visual Studio

Una división de Microsoft Corporation

Una manera Microsoft

Redmond, Washington 98052-6399

Copyright © 2017 por Microsoft Corporation

Todos los derechos reservados. Ninguna parte del contenido de este libro puede ser reproducida o transmitida en cualquier forma o por cualquier medio sin el permiso por escrito del editor.

Este libro se proporciona “tal cual” y expresa puntos de vista y las opiniones del autor. Los puntos de vista, opiniones e informaciones expresadas en este libro, incluidas las direcciones URL y otras referencias a sitios web de Internet, pueden cambiar sin previo aviso.

Algunos ejemplos descritos en el presente documento se proporcionan solamente con ilustración y son ficticios. No se encontró asociación o conexión real se pretende ni se debe inferir.

Microsoft y las marcas que figuran en http://www.microsoft.com en la página web “marcas” son marcas comerciales del grupo de compañías de Microsoft. Todas las demás marcas son propiedad de sus respectivos dueños.

Autor:

Steve Smith [(http://ardalis.com)](http://ardalis.com/)

Los participantes y los colaboradores:

Scott Hanselman, equipo de producto .NET, Microsoft

César de la Torre, equipo de producto .NET, Microsoft



Contenido

**Introducción** **1**

Versión 1

Propósito 2

Quién debería usar esta guía 2

¿Cómo se puede utilizar esta guía 2

**Características de las aplicaciones web modernas** **3**

Resumen 3

Aplicación Referencia: eShopOnWeb 3

Nube-Alojado y escalable 4

multiplataforma 4

Modular y de estructura flexible 5

Probado con facilidad las pruebas automatizadas 5

Los comportamientos tradicionales y compatibles SPA 5

Desarrollo e implementación sencilla 6

ASP.NET Web y las formas tradicionales 6

**La elección entre las aplicaciones Web tradicionales y Aplicaciones Una sola página (ZEPA)** **7**

Resumen 7

Plantillas de proyecto 8

Cuando de elegir las aplicaciones web tradicionales 10

Cuando elegir ZEPA 11

Tabla de decisiones - Web tradicional o SPA 13

**Principios de arquitectura** **14**

Resumen 14

principios de diseño comunes 14

Separación de intereses 14

La encapsulación 15

dependencia de Inversion 15

dependencias explícitas 17

yo

Responsabilidad individual 17

No te repitas (DRY) 18

La ignorancia de persistencia 19

Contextos acotadas 20

**Arquitecturas de aplicaciones Web comunes** **21**

Resumen 21

¿Qué es una aplicación monolítica? 21

Todo-en-Uno aplicaciones 21

¿Cuáles son las capas? 22

Las aplicaciones tradicionales “Arquitectura N-Layer” 24

arquitectura limpia 28

Las aplicaciones monolíticas y Contenedores 34

aplicación monolítica desplegado como un contenedor 36

**Tecnologías Web del lado del cliente común** **38**

Resumen 38

HTML 38

CSS 39

JavaScript 40

Aplicaciones Web legado con jQuery 40

jQuery vs un Marco SPA 40

SPAs angulares 41

Reaccionar 42

La elección de un Marco SPA 42

**El desarrollo de ASP.NET Core Aplicaciones** **44**

Resumen 44

MVC y Páginas de la maquinilla de afeitar 44

¿Por Razor páginas? 45

Cuándo utilizar MVC 46

Las solicitudes de mapeo a las respuestas 46

Trabajar con dependencias 48

Declarar las dependencias 50

La estructuración de la Aplicación 51

Organización función 52

Problemas transversales 54

Seguridad 57

Autenticación 57

Autorización 59

Comunicación cliente 63

ii

Dominio-Driven Design - En caso de aplicarlo? 64

¿Cuándo se debe aplicar DDD sesenta y cinco

Cuando no se debería solicitar DDD sesenta y cinco

Despliegue 67

**Trabajo con datos en ASP.NET Aplicaciones Core** **69**

Resumen 69

Entidad Marco de base (para bases de datos relacionales) 69

el DbContext 70

Configuración de EF Core 70

Obtención y almacenamiento de datos 72

Ir a buscar datos relacionados 73

datos de encapsulación 74

Las conexiones flexibles 75

EF Core o micro-ORM? 77

SQL o NoSQL 79

Azure Cosmos DB 80

Otras Opciones de persistencia 81

El almacenamiento en caché 81

ASP.NET Core Almacenamiento en caché de respuesta 82

El almacenamiento en caché de datos 82

**Pruebas de ASP.NET MVC Core Aplicaciones** **87**

Resumen 87

Tipos de pruebas automatizadas 87

Las pruebas unitarias 87

Las pruebas de integración 88

Las pruebas funcionales 89

Pirámide de pruebas 90

Lo que prueba 91

La organización de Proyectos de Prueba 91

Naming Test 93

Unidad de prueba ASP.NET Aplicaciones Core 94

Pruebas de integración ASP.NET Aplicaciones Core 96

Pruebas funcionales ASP.NET Aplicaciones Core 96

**proceso de desarrollo de aplicaciones ASP.NET Core alojados en Azure** **99**

Visión 99

Entorno de desarrollo para aplicaciones ASP.NET Core 99

Herramientas de desarrollo de opciones: IDE o editor 99

flujo de trabajo de desarrollo para aplicaciones ASP.NET Core alojados en Azure 100

iii

Configuración inicial 100

Flujo de trabajo para el desarrollo de aplicaciones ASP.NET Core alojados en Azure 102

referencias 103

**Azure de alojamiento Recomendaciones para ASP.NET Core Web Apps** **104**

Resumen 104

Aplicaciones web 104

Aplicaciones Web de servicios de aplicaciones 105

Envases y Azure Container Service 105

Azure Servicio Tela 106

Máquinas Virtuales Azure 106

Los procesos lógicos 107

Datos 107

Recomendaciones Arquitectura 108

iv

SECCIÓN 1

Introducción

.NET y ASP.NET Core Core ofrecen varias ventajas sobre el desarrollo tradicional .NET. Debe utilizar .NET Core para sus aplicaciones de servidor si alguno de los siguientes son importantes para el éxito de su aplicación:

* Compatibilidad entre plataformas
* El uso de microservicios
* El uso de contenedores acoplables
* requisitos de alto rendimiento y escalabilidad
* De lado a lado de versiones de versiones de .NET mediante la aplicación en el mismo servidor

aplicaciones .NET tradicionales pueden - y no - apoyar estos requisitos, pero ASP.NET y .NET Core Core se han optimizado para ofrecer un soporte mejorado para los escenarios anteriores.

Cada vez más organizaciones están eligiendo para alojar sus aplicaciones web en la nube usando servicios como Microsoft Azure. Usted debe considerar el alojamiento de su aplicación en la nube si los siguientes son importantes para su aplicación u organización:

* Reducción de la inversión en los costos del centro de datos (hardware, software, espacio, servicios públicos, etc.)
* Precios flexibles (pago en función del uso, no por capacidad ociosa)
* fiabilidad extrema
* la movilidad aplicación mejorada; cambiar fácilmente dónde y cómo se implementa la aplicación
* capacidad flexible; escalar hacia arriba o hacia abajo en base a las necesidades reales

La creación de aplicaciones web con ASP.NET Core, alojada en Microsoft Azure, ofrece numerosas ventajas competitivas frente a las alternativas tradicionales. ASP.NET núcleo está optimizado para las prácticas de desarrollo de aplicaciones web modernas y escenarios de alojamiento en la nube. En esta guía, usted aprenderá cómo diseñar sus aplicaciones ASP.NET Core a sacar el mejor provecho de estas capacidades.

**Versión**

Este libro ha sido revisado para cubrir la versión 2.0 de ASP.NET y .NET Core Core.

1 Capítulo 1 Introducción

Propósito

Esta guía proporciona una guía de extremo a extremo en la construcción de aplicaciones web utilizando ASP.NET monolíticos Core y Azure.

Esta guía es complementaria a la “Architecting y desarrollo en contenedores y MICROSERVICE - Aplicaciones basadas con .NET”, que se centra más en estibador, microservicios, y el despliegue de contenedores para albergar las aplicaciones empresariales.



**El desarrollo de la arquitectura de contenedores y Microservice aplicaciones basadas en .NET de libros electrónicos**

<http://aka.ms/MicroservicesEbook>

**.NET microservicios aplicación de ejemplo**

<http://aka.ms/microservicesarchitecture>

**Quién debería usar esta guía**

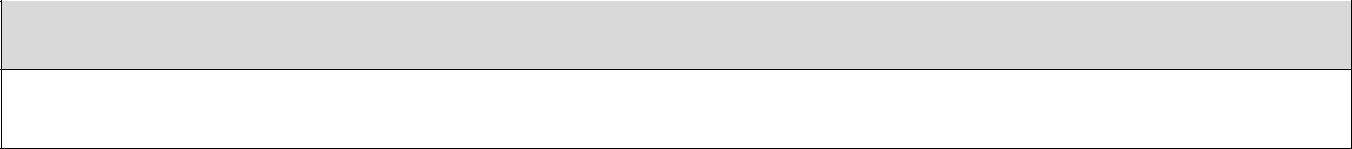
La audiencia de esta guía se debe principalmente a los desarrolladores, clientes potenciales de desarrollo, y los arquitectos que están interesados ​​en la construcción de modernas aplicaciones web utilizando tecnologías y servicios de Microsoft en la nube.

Una audiencia secundaria es que toman las decisiones técnicas que ya están familiarizados ASP.NET y / o Azure y están en busca de información sobre si tiene sentido para actualizar a ASP.NET Core para proyectos nuevos o existentes.

**¿Cómo se puede utilizar esta guía**

Esta guía ha sido condensada en un relativamente pequeño documento que se centra en la creación de aplicaciones web con tecnologías modernas y .NET Windows Azure. Como tal, se puede leer en su totalidad para proporcionar una base para entender este tipo de aplicaciones y sus consideraciones técnicas. La guía, junto con su aplicación de ejemplo, también puede servir como un punto de partida o de referencia. Utilizar la aplicación de ejemplo asociada como plantilla para sus propias aplicaciones, o para ver cómo se puede organizar partes componentes de la aplicación. Se refieren a los principios y la cobertura de Arquitectura y Tecnología de opciones y consideraciones de decisión cuando un peso de estas opciones para su propia aplicación de la guía.

Siéntase libre de enviar esta guía a su equipo para ayudar a asegurar una comprensión común de estas consideraciones y oportunidades. Tener todos los que trabajan a partir de un conjunto común de terminología y los principios subyacentes ayudará a asegurar la aplicación coherente de patrones y prácticas arquitectónicas.



**referencias**

**La elección entre .NET Core y .NET Framework para aplicaciones de servidor**

<https://docs.microsoft.com/en-us/dotnet/articles/standard/choosing-core-framework-server>

2 Capítulo 1 Introducción

Características de las aplicaciones web modernas

*“... Con un diseño adecuado, las características son baratos. Este enfoque es ardua, pero*

*continúa teniendo éxito “.*

Dennis Ritchie

Resumen

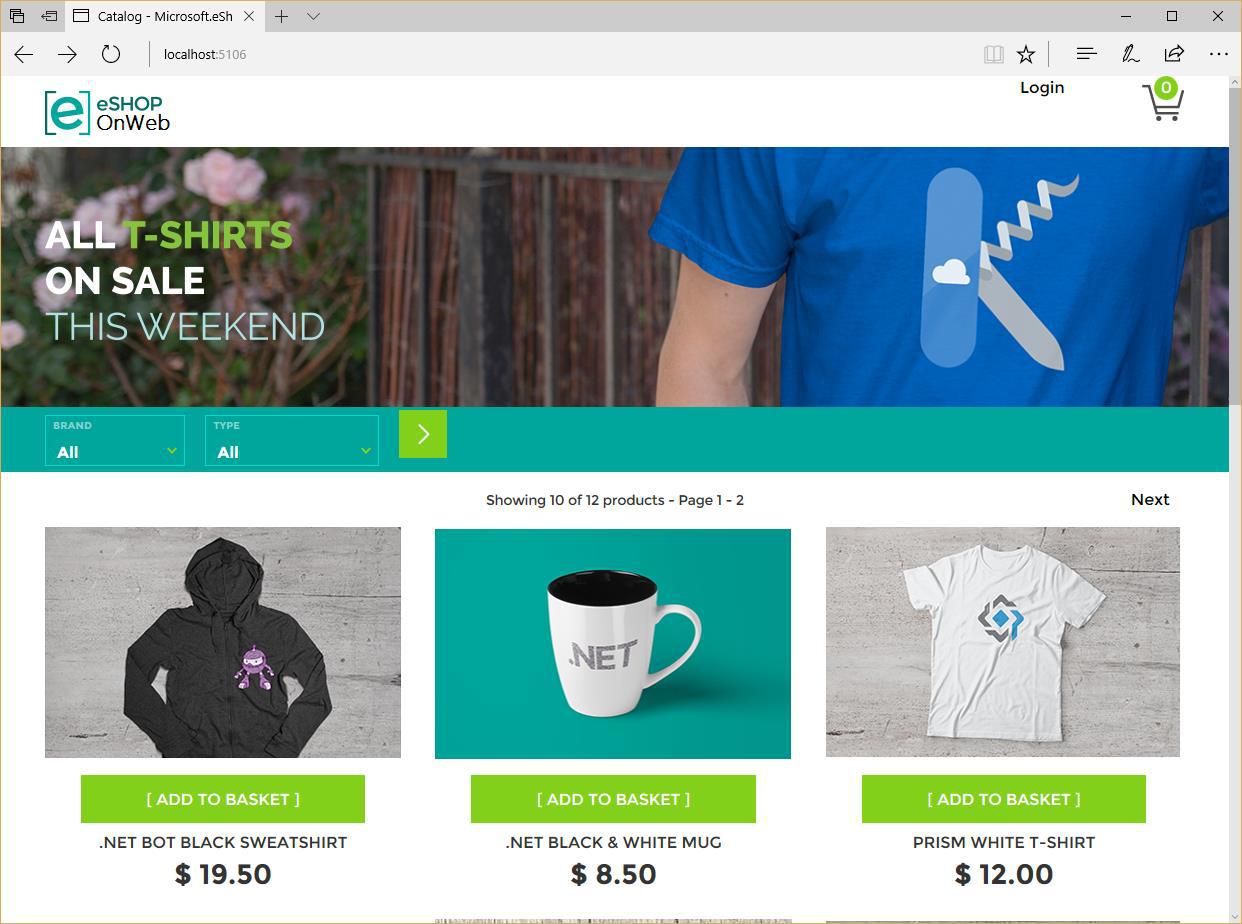
aplicaciones web modernas tienen altas expectativas de los usuarios y una mayor demanda que nunca. Se espera que las aplicaciones web de hoy en día para estar disponible 24/7 desde cualquier parte del mundo, y utilizable a partir de prácticamente cualquier tamaño o dispositivo de pantalla. Las aplicaciones web deben ser seguras, flexibles y escalables para satisfacer los picos de demanda. Cada vez más, escenarios complejos deben ser manejados por las experiencias de usuario ricas construidas en el cliente utilizando JavaScript y comunicar de manera eficiente a través de APIs web.

ASP.NET núcleo está optimizado para aplicaciones web modernas y escenarios de hosting basados ​​en la nube. Su diseño modular permite que las aplicaciones dependen sólo aquellas características que utilizan, la mejora de la seguridad de aplicaciones y el rendimiento al tiempo que reduce los requisitos de recursos de alojamiento.

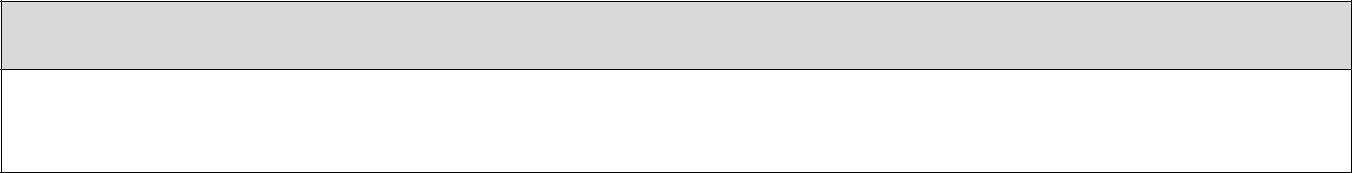
Aplicación Referencia: eShopOnWeb

Esta guía incluye una aplicación de referencia, eShopOnWeb, que demuestra algunos de los principios y recomendaciones. La aplicación es un simple tienda en línea que soporta la navegación a través de un catálogo de camisas, tazas, y otros artículos de marketing. La aplicación de referencia es deliberadamente sencilla para que sea fácil de entender.

|  |  |
| --- | --- |
| 3 | Capitulo 2 |
|  | Características de las aplicaciones web modernas |



La Figura 2-1. eShopOnWeb



**Aplicación referencia**

**eShopOnWeb**

https://github.com/dotnet-architecture/eShopOnWeb

Nube-Alojado y escalable

ASP.NET núcleo está optimizado para la nube (nube pública, nube privada, ninguna nube) porque es de poca memoria y de alto rendimiento. La huella más pequeña de las aplicaciones ASP.NET Core significa que puede albergar más de ellos en el mismo hardware, y pagar por un menor número de recursos cuando se utiliza de pago por uso que vaya servicios de alojamiento en la nube. Cuanto más alto rendimiento significa que puede servir a más clientes a partir de una aplicación dada el mismo hardware, reduciendo aún más la necesidad de invertir en servidores e infraestructura de alojamiento.

multiplataforma

ASP.NET Core es multiplataforma y se puede ejecutar en Linux y MacOS, así como Windows. Esto abre muchas nuevas opciones para el desarrollo y despliegue de aplicaciones integradas con ASP.NET Core. Estibador

|  |  |
| --- | --- |
| 4 | Capitulo 2 |
|  | Características de las aplicaciones web modernas |

contenedores, que normalmente se ejecutan Linux hoy en día, pueden alojar aplicaciones ASP.NET Core, lo que les permite aprovechar las ventajas de los contenedores y microservicios (enlace a microservicios y contenedores de libros electrónicos).

Modular y de estructura flexible

NuGet paquetes son ciudadanos de primera clase en .NET Core y aplicaciones ASP.NET básicas se componen de muchas bibliotecas a través de NuGet. Esta granularidad de funcionalidad ayuda a asegurar aplicaciones sólo dependen de e implementar la funcionalidad que requieren, reduciendo su superficie la huella y la vulnerabilidad de seguridad.

ASP.NET subyacente también es totalmente compatible con la inyección de dependencias, tanto a nivel interno como a nivel de aplicación. Las interfaces pueden tener varias implementaciones que se pueden intercambiar según sea necesario. La inyección de dependencias Permite que las aplicaciones sin apretar pareja a esas interfaces, haciéndolos más fáciles de ampliar, mantener y prueba.

Probado con facilidad las pruebas automatizadas

ASP.NET Core pruebas de apoyo a las aplicaciones de la unidad, y su acoplamiento débil y el apoyo a las inyecciones de dependencia hace que sea fácil de intercambiar inquietudes de infraestructura con implementaciones falsos con fines de prueba. ASP.NET Core también envía un TESTSERVER que se puede utilizar para alojar aplicaciones en la memoria. Las pruebas funcionales pueden entonces hacer peticiones a este servidor en memoria, el ejercicio de la pila de aplicación completa (incluyendo middleware, enrutamiento, la unión modelo, filtros, etc.) y recibir una respuesta, todo en una fracción del tiempo que sería necesario para acoger la aplicación en el servidor real y hacer peticiones a través de la capa de red. Estas pruebas son especialmente fáciles de escribir, y valioso, para las API, que son cada vez más importante en las aplicaciones web modernas.

Los comportamientos tradicionales y compatibles SPA

aplicaciones web tradicionales han implicado poco el comportamiento del lado del cliente, pero en su lugar se han basado en el servidor para todas las consultas de navegación, y actualiza la aplicación que tenga que hacer. Cada nueva operación realizada por el usuario se traduciría en una nueva solicitud web, con el resultado de una recarga de página completa en el navegador del usuario final. Modelo-Vista-Controlador (MVC) marcos suelen seguir este enfoque, con cada nueva solicitud correspondiente a una acción de un controlador diferente, que a su vez trabajar con un modelo y devolver una vista. Algunas operaciones individuales en una página determinada podrían ser mejorados con AJAX (JavaScript asíncrono y XML) funcionalidad, pero la arquitectura global de la aplicación utilizan muchos puntos de vista diferentes MVC y los puntos finales de URL.

Aplicaciones sola página (ZEPA), por el contrario, implican muy pocas cargas de la página del lado del servidor generadas dinámicamente (en su caso). Muchos balnearios se inicializan dentro de un archivo HTML estático que carga las bibliotecas de JavaScript necesarios para iniciar y ejecutar la aplicación. Estas aplicaciones hacen un uso intensivo de las API web para sus necesidades de datos y pueden proporcionar una experiencia de usuario mucho más rica.

Muchas aplicaciones web implican una combinación de comportamiento tradicional de aplicaciones web (normalmente de contenido) y ZEPA (para la interactividad). ASP.NET es compatible con Core API Web MVC y en la misma aplicación, utilizando el mismo conjunto de herramientas y bibliotecas marco subyacentes.

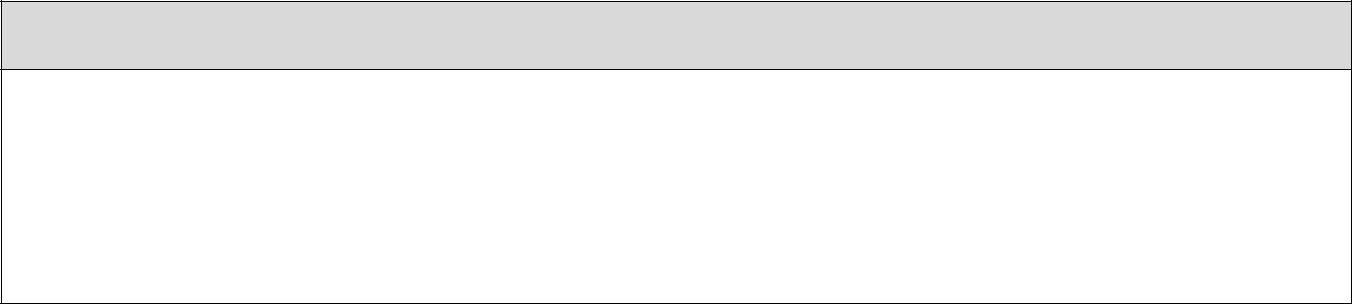
|  |  |
| --- | --- |
| 5 | Capitulo 2 |
|  | Características de las aplicaciones web modernas |

Desarrollo e implementación sencilla

aplicaciones ASP.NET Core se pueden escribir con editores de texto simples e interfaces de línea de comandos, o entornos de desarrollo con todas las funciones como Visual Studio. aplicaciones monolíticas se implementan a un solo punto final. Los despliegues se puede automatizar fácilmente a ocurrir como parte de una integración continua (CI) y la tubería de suministro continuo (CD). Además de las herramientas tradicionales CI / CD, Windows Azure ha integrado soporte para repositorios Git y se puede desplegar de forma automática actualizaciones a medida que se hacen a una rama Git o etiqueta especificada.

ASP.NET Web y las formas tradicionales

Además de ASP.NET Core, 4.x ASP.NET tradicional sigue siendo una plataforma robusta y fiable para la creación de aplicaciones web. ASP.NET MVC es compatible con los modelos de desarrollo y API Web, así como los formularios Web, lo que es muy adecuado para el desarrollo de aplicaciones basado en páginas y cuenta con un componente rico ecosistema de terceros. Windows Azure tiene un gran apoyo de muchos años para aplicaciones ASP.NET 4.x, y muchos desarrolladores están familiarizados con esta plataforma.



**Referencias - aplicaciones web modernas**

**Introducción a ASP.NET Core**

<https://docs.microsoft.com/en-us/aspnet/core/>

**Seis Principales ventajas de ASP.NET Core que la hacen diferente y mejor**

<http://blog.trigent.com/six-key-benefits-of-asp-net-core-1-0-which-make-it-different-better/>

**Las pruebas en ASP.NET Core**

<https://docs.microsoft.com/en-us/aspnet/core/testing/>

6 Capítulo 2 Características de las aplicaciones web modernas

Seccion 3

La elección entre las aplicaciones Web tradicionales y Aplicaciones Una sola página (ZEPA)

*“Ley de Atwood: Cualquier aplicación que puede ser escrito en JavaScript, con el tiempo será*

*escrito en JavaScript “.*

Jeff Atwood

Resumen

Hay dos enfoques generales para la construcción de aplicaciones web hoy en día: las aplicaciones tradicionales de Internet que realizan la mayor parte de la lógica de la aplicación en el servidor y las aplicaciones de una sola página (ZEPA) que realizan la mayor parte de la lógica de la interfaz de usuario en un navegador web, la comunicación con el servidor web utilizando principalmente las API web. Un enfoque híbrido también es posible, siendo el más simple el anfitrión uno o más ricos sub-aplicaciones SPA-como dentro de una aplicación web tradicional más grande.

|  |  |
| --- | --- |
| 7 | Capítulo 3 |
|  | La elección entre las aplicaciones web tradicionales y spas |

Debe utilizar las aplicaciones web tradicionales cuando:

* requisitos del lado del cliente de su aplicación son simples o incluso de sólo lectura.
* Su aplicación necesita para funcionar en navegadores sin soporte JavaScript.
* Su equipo no está familiarizado con las técnicas de desarrollo mecanografiado JavaScript o.

Debe utilizar un SPA cuando:

* Su aplicación debe exponer una interfaz de usuario rica con muchas características.
* Su equipo está familiarizado con JavaScript y / o desarrollo mecanografiado.
* Su aplicación ya debe exponer una API para otros (internas o públicas) clientes.

Además, los marcos de SPA requieren una mayor experiencia arquitectónica y la seguridad. Experimentan una mayor rotación, debido a las frecuentes actualizaciones y nuevos marcos que las aplicaciones web tradicionales. Configuración de los procesos de construcción y despliegue automáticos y que utilizan opciones de implementación como contenedores pueden ser más difíciles con las aplicaciones de SPA que las aplicaciones web tradicionales.

Las mejoras en la experiencia de usuario hechas posibles por el modelo SPA deben sopesarse contra estas consideraciones.

Plantillas de proyecto

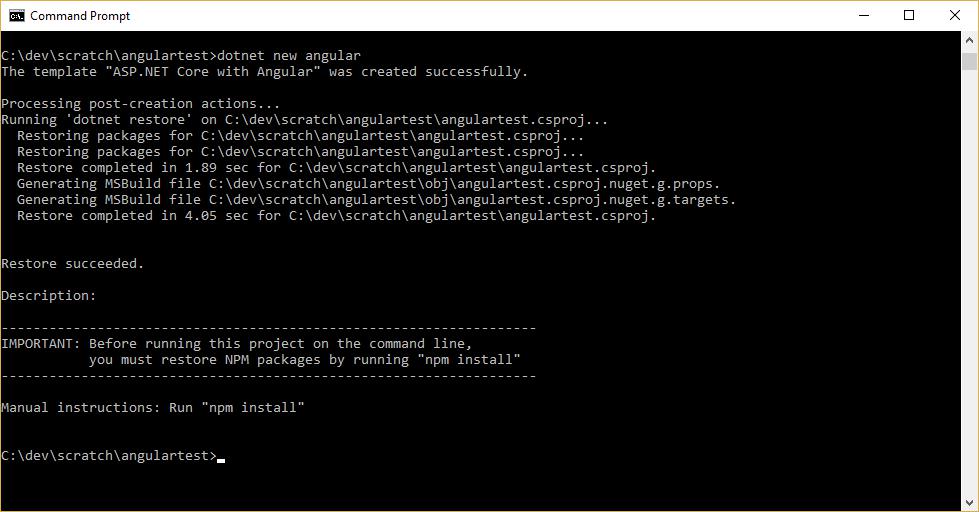
ASP.NET Core (2.0+) los buques varias plantillas de aplicaciones web que están disponibles en Visual Studio o la CLI dotnet que se puede utilizar para empezar a crear aplicaciones tradicionales o de tipo spa. Estas son sólo algunas de las plantillas que se envían y estos sólo son puntos de partida. Usted puede encontrar útil 3rd plantillas del partido y de código abierto o crear su propia:

* Aplicación Web ASP.NET Core (Razor Páginas)
* ASP.NET Web App Core (Modelo-Vista-Controlador)
* API Web ASP.NET Core (sin interfaz de usuario)
* ASP.NET Core con Angular (SPA)
* ASP.NET Core con React.js (SPA)
* ASP.NET Core con React.js y Redux (SPA)

El método recomendado para muchas aplicaciones tradicionales es Razor páginas, que ofrecen las mismas ventajas arquitectónicas de ASP.NET MVC Core, pero con un menor número de archivos y carpetas. También puede elegir la plantilla MVC, que será muy familiar para los desarrolladores con experiencia utilizando ASP.NET MVC o 1.x de ASP.NET MVC Core Si no se requiere ninguna representación del lado del servidor de páginas o puntos de vista, se puede utilizar la plantilla Web API, y crear su aplicación utilizando los archivos estáticos que utilizan JavaScript para llamar a sus métodos de la API. La aplicación de ejemplo eShopOnWeb incluye dos proyectos web, lo que demuestra tanto MVC y la maquinilla de afeitar Páginas estrategias para la organización del proyecto de interfaz de usuario. Es importante señalar que la maquinilla de afeitar Pages es un modelo basado en páginas que se acumula en la parte superior del modelo MVC. Se pueden añadir más páginas o más controladores para las API HTML o Web y mezclar y combinar a su gusto.

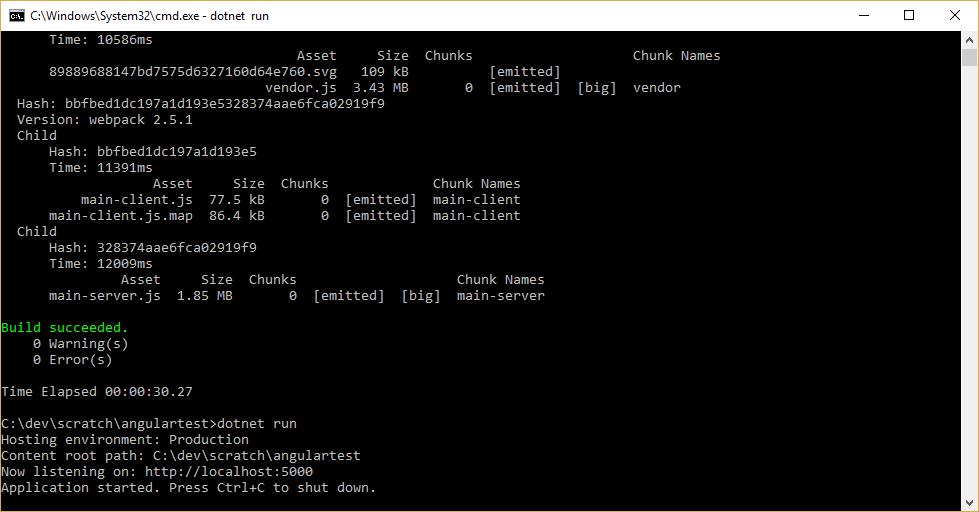
Encontrará plantillas incorporadas para angular, así como reaccionar y Redux para ayudar a comenzar la construcción de un SPA. Estas plantillas creados los archivos necesarios y del lado del cliente construir herramientas para que pueda empezar a trabajar rápidamente. Estas plantillas utilizan NPM (nodo gestor de paquetes) para instalar sus dependencias del lado del cliente.

|  |  |
| --- | --- |
| 8 | Capítulo 3 |
|  | La elección entre las aplicaciones web tradicionales y spas |



La Figura 3-1. Instalación de la plantilla de proyecto angular de la CLI dotnet.

Asegúrate de que tienes la última versión del nodejs instalados, a continuación, generar y ejecutar la aplicación (pista dotnet desde la línea de comandos).



La Figura 3-2. Ejecución de la plantilla de proyecto angular de la CLI dotnet.

La plantilla angular ofrece un SPA simple que incluye la navegación y el enrutamiento y ejemplos de implementación del lado del cliente y el comportamiento del lado del servidor.

|  |  |
| --- | --- |
| 9 | Capítulo 3 |
|  | La elección entre las aplicaciones web tradicionales y spas |

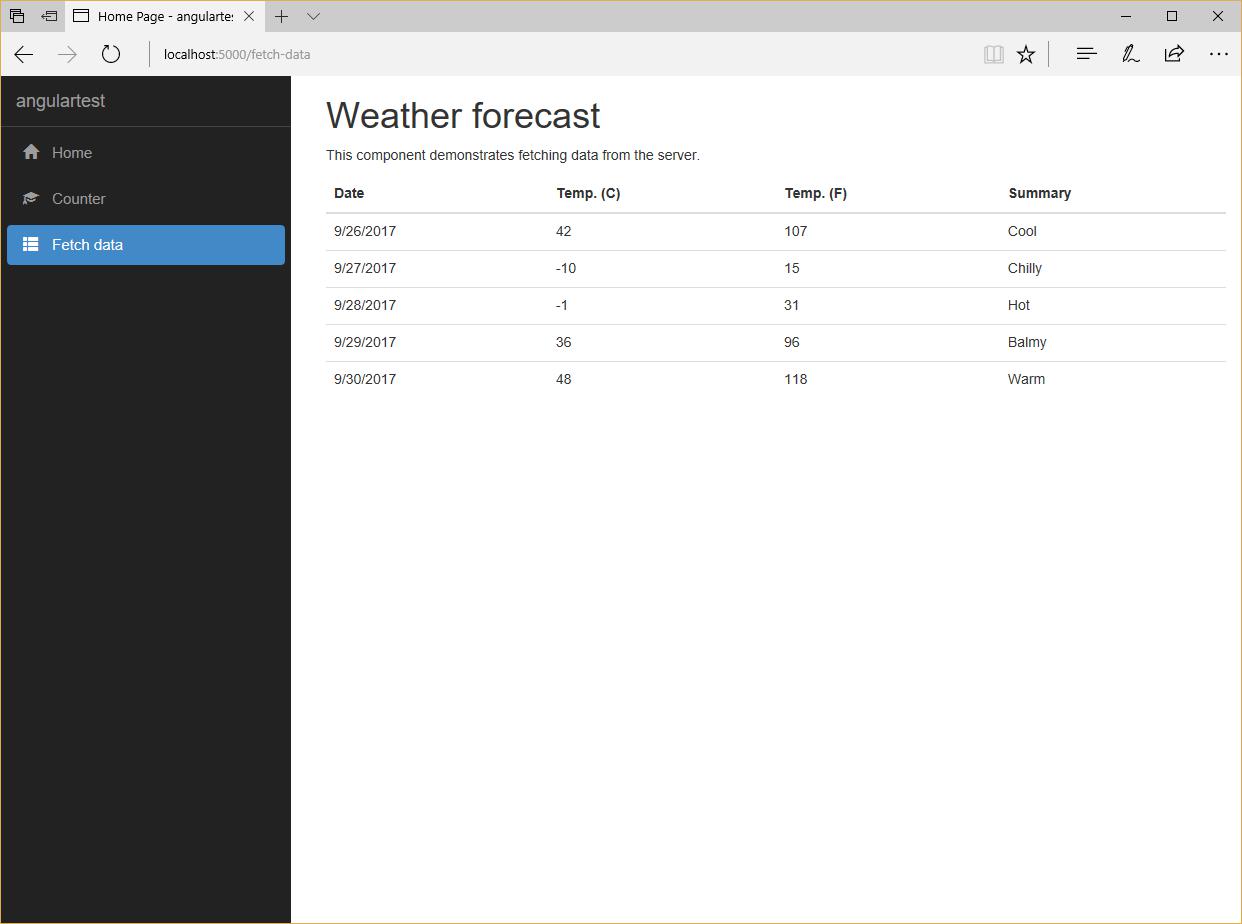


Figura 3-3. El SPA angular en el navegador.

Cuando de elegir las aplicaciones web tradicionales

La siguiente es una explicación más detallada para recoger las aplicaciones web tradicionales.

**Su aplicación tiene, posiblemente, de sólo lectura, requisitos del cliente simples**

Muchas aplicaciones web se consumen principalmente en forma de sólo lectura por la gran mayoría de sus usuarios. Sólo lectura (o lectura en su mayoría) las aplicaciones tienden a ser mucho más simples que los que mantener y manipular una gran cantidad de estado. Por ejemplo, un motor de búsqueda podría consistir en un solo punto de entrada con una caja de texto y una segunda página para mostrar los resultados de búsqueda. Los usuarios anónimos pueden fácilmente hacer peticiones, y hay poca necesidad de lógica de cliente. Del mismo modo, la aplicación de cara al público de un sistema de gestión de contenido blog o generalmente se compone principalmente de contenido con poco comportamiento del lado del cliente. Este tipo de aplicaciones se construyen fácilmente como las aplicaciones tradicionales basadas en servidores web que se mostrará en el navegador que realizan la lógica en el servidor web y representar HTML.

**Su aplicación necesita para funcionar en navegadores sin soporte JavaScript**

|  |  |
| --- | --- |
| 10 | Capítulo 3 |
|  | La elección entre las aplicaciones web tradicionales y spas |

aplicaciones web que necesitan para funcionar en navegadores con soporte limitado o ningún JavaScript debe ser escrito utilizando flujos de trabajo tradicionales de aplicaciones web (o al menos ser capaces de caer de nuevo a este tipo de comportamiento). SPAs requieren del lado del cliente JavaScript para funcionar; si no está disponible, ZEPA no son una buena opción.

**Su equipo no está familiarizado con las técnicas de desarrollo mecanografiado JavaScript o**

Si su equipo no está familiarizado con JavaScript o mecanografiado, pero está familiarizado con el desarrollo de aplicaciones web del lado del servidor, entonces probablemente será capaz de entregar una aplicación web tradicional más rápidamente que un SPA. A menos que aprender a programar ZEPA es una meta, o la experiencia del usuario proporcionada por un SPA que se requiere, las aplicaciones web tradicionales son una opción más productivo para los equipos que ya están familiarizados con la construcción de ellos.

Cuando elegir ZEPA

La siguiente es una explicación más detallada de cuándo debe elegir un estilo individual página Aplicaciones de desarrollo para su aplicación web.

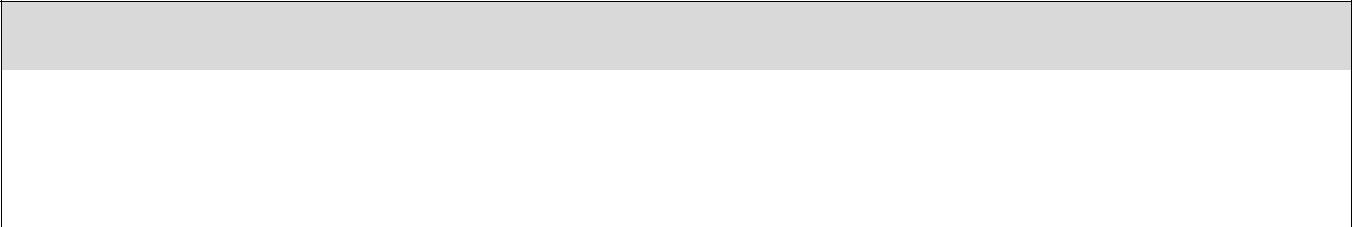
**La aplicación debe exponer una interfaz de usuario rica con muchas características**

SPAs pueden soportar la funcionalidad del lado del cliente rico que no requiere volver a cargar la página, los usuarios tomar acciones o navegar entre las áreas de la aplicación. SPAs se cargan más rápido, ir a buscar los datos en segundo plano, y las acciones de usuarios individuales son más sensibles ya cargar la página completa son raros. SPAs pueden soportar actualizaciones incrementales, el ahorro de los formularios o documentos parcialmente terminados y sin que el usuario tenga que hacer clic en un botón para enviar un formulario. SPAs pueden apoyar comportamientos ricos del lado del cliente, tales como arrastrar y soltar, mucho más fácilmente que las aplicaciones tradicionales. SPAs pueden ser diseñados para funcionar en un modo desconectado, haciendo cambios a un modelo de cliente que eventualmente se vuelve a sincronizar con el servidor una vez que se restablezca la conexión.

Tenga en cuenta que con frecuencia ZEPA necesitan para implementar características que están integrados en las aplicaciones web tradicionales, como mostrar una URL significativa en la barra de direcciones que refleja la operación actual (y permitiendo a los usuarios marcador o enlace profundo a este URL para volver a ella). SPAs también deberían permitir a los usuarios utilizar los botones Atrás y Adelante del navegador con resultados que no sorprenderá a ellos.

**Su equipo está familiarizado con JavaScript y / o desarrollo mecanografiado**

Escribiendo ZEPA requiere estar familiarizado con las técnicas de programación del lado del cliente JavaScript y / o mecanografiado e y bibliotecas. Su equipo debe ser competente en la escritura moderna JavaScript utilizando un marco de spa como angular.



**Referencias - Marcos SPA**

**Angular**

<https://angular.io/>

**Comparación de los 4 Marcos populares de JavaScript**

<https://www.developereconomics.com/feature-comparison-of-4-popular-js-mv-frameworks>

**Su aplicación ya debe exponer una API para otros (internas o públicas) clientes**

|  |  |
| --- | --- |
| 11 | Capítulo 3 |
|  | La elección entre las aplicaciones web tradicionales y spas |

Si ya está apoyando una API web para su uso por otros clientes, se puede requerir menos esfuerzo para crear una aplicación que aprovecha la SPA estas API en lugar de reproducir la lógica en la forma del lado del servidor. SPAs hacen un amplio uso de las API de Internet para consultar y actualizar los datos ya que los usuarios interactúan con la aplicación.

|  |  |
| --- | --- |
| 12 | Capítulo 3 |
|  | La elección entre las aplicaciones web tradicionales y spas |

Tabla de decisiones - Web tradicional o SPA

La siguiente tabla de decisión se resumen algunos de los factores básicos a considerar al elegir entre una aplicación web tradicional y un spa.

|  |  |  |
| --- | --- | --- |
| **Factor** | **Aplicación Web tradicional** | **Aplicación de una sola página** |
|  |  |  |
| La familiaridad con el equipo requerido | **Mínimo** | **Necesario** |
| JavaScript / Letra de imprenta |  |  |
|  |  |  |
| Los navegadores de apoyo sin | **Soportado** | **No soportado** |
| scripting |  |  |
|  |  |  |
| Mínima del lado del cliente | **Muy adecuado** | **Overkill** |
| comportamiento de las aplicaciones |  |  |
|  |  |  |
| , La interfaz de usuario rica Complejo | **Limitado** | **Muy adecuado** |
| requisitos |  |  |
|  |  |  |

|  |  |
| --- | --- |
| 13 | Capítulo 3 |
|  | La elección entre las aplicaciones web tradicionales y spas |

CAPÍTULO 4

Principios de arquitectura

*“Si los constructores construyen edificios de la manera en que los programadores escribieron los programas, entonces el primer*

*carpintero que llegó destruiría la civilización “.*

Gerald Weinberg

Resumen

Usted debe arquitecto de soluciones de software de diseño y con capacidad de mantenimiento en mente. Los principios descritos en esta sección pueden ayudar a guiarlo hacia las decisiones arquitectónicas que darán lugar a aplicaciones limpias, fáciles de mantener. En general, estos principios le guiarán hacia la construcción de aplicaciones a partir de componentes discretos que no están estrechamente unidas a otras partes de la aplicación, sino más bien comunicarse a través de interfaces explícitas o sistemas de mensajería.

principios de diseño comunes

**Separación de intereses**

Un principio rector en el desarrollo es la separación de intereses. Este principio afirma que el software debe ser separado basado en el tipo de trabajo que realiza. Por ejemplo, considere una aplicación que incluye la lógica para identificar elementos destacables para mostrar al usuario, el formato de esos artículos, de una forma particular, para destacarlos. El comportamiento responsable de la elección de qué artículos a formato deben mantenerse separados del comportamiento responsable para dar formato a los elementos, ya que éstas son preocupaciones separadas que están solamente casualmente relacionados entre sí.

Arquitectónicamente, las aplicaciones pueden ser construidas lógicamente seguir este principio mediante la separación de la conducta actividad principal de la lógica de la infraestructura y la interfaz de usuario. Idealmente, las reglas de negocio y la lógica deben residir en un proyecto independiente, que no debe depender de otros proyectos en la aplicación. Esto ayuda a asegurar que el modelo de negocio es fácil de probar y que pueda evolucionar sin ser fuertemente acoplado al bajo nivel de detalles de implementación. Separación de intereses es un factor clave detrás del uso de capas en las arquitecturas de aplicaciones.

14 Capítulo 4 Principios de Arquitectura

**La encapsulación**

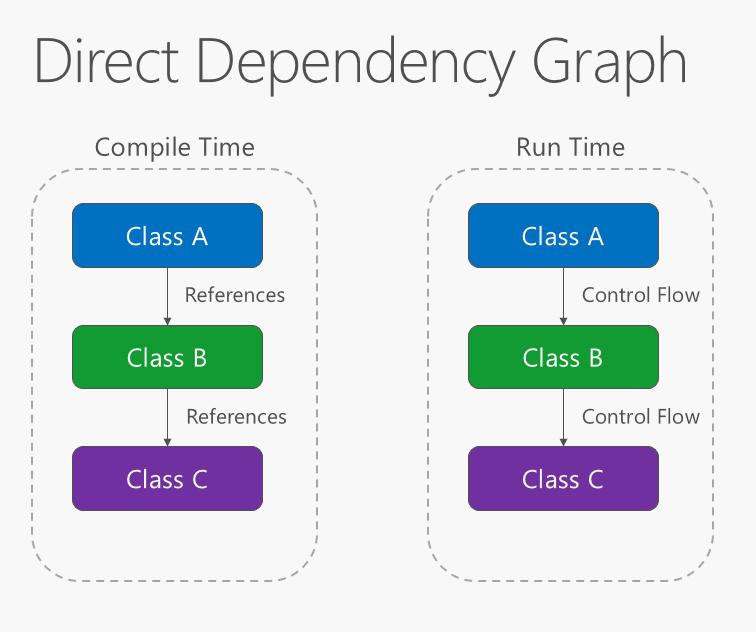
Diferentes partes de una aplicación debe utilizar la encapsulación para aislarlos de otras partes de la aplicación. componentes de la aplicación y las capas deben ser capaces de ajustar su implementación interna sin romper sus colaboradores si no se violan los contratos externos. El uso adecuado de encapsulación ayuda a conseguir el acoplamiento flojo y modularidad en diseños de aplicación, ya que los objetos y paquetes pueden ser reemplazados con implementaciones alternativas siempre que se mantiene la misma interfaz.

En las clases, la encapsulación se consigue limitando el acceso externo a estado interno de la clase. Si fuera un actor quiere manipular el estado del objeto, debe hacerlo a través de una función bien definida (o colocador propiedad), en lugar de tener acceso directo al estado privado del objeto. Del mismo modo, los componentes de aplicación y aplicaciones mismos deben exponer interfaces bien definidas por sus colaboradores a utilizar, en lugar de permitir su estado para ser modificados directamente. Esto libera diseño interno de la aplicación para evolucionar con el tiempo sin tener que preocuparse de que al hacerlo se romperá colaboradores, siempre y cuando se mantengan los contratos públicos.

**dependencia de Inversion**

La dirección de la dependencia dentro de la aplicación debe estar en la dirección de la abstracción, no los detalles de implementación. La mayoría de las aplicaciones se escriben tales que los flujos de dependencia en tiempo de compilación en la dirección de la ejecución en tiempo de ejecución. Esto produce una gráfica de dependencia directa. Es decir, si el módulo A llama a una función en el módulo B, que llama a una función en el módulo C, a continuación, en tiempo de compilación A dependerá de B que dependerá de C, como se muestra en la Figura 4-1.

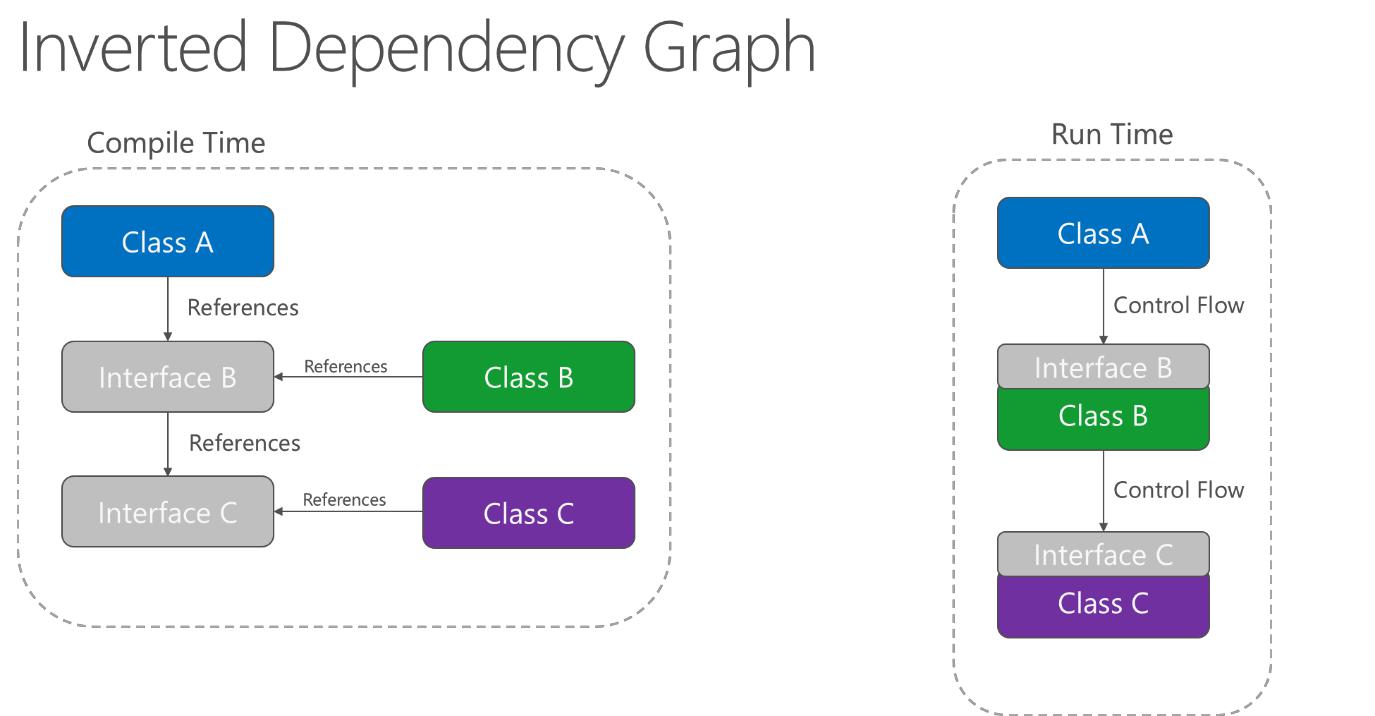
|  |  |
| --- | --- |
| 15 | Capítulo 4 |
|  | Principios de arquitectura |



***La Figura 4-1.*** *gráfico de la dependencia directa. flujo de control Espejos referencias.*

Aplicando el principio de la dependencia inversión permite A a llamar a métodos en una abstracción que B implementos, por lo que es posible para un para llamar a B en tiempo de ejecución, pero para B depender de una interfaz controlada por A en tiempo de compilación (por lo tanto, invirtiendo el de compilación típica dependencia del tiempo). En tiempo de ejecución, el flujo de la ejecución del programa se mantiene sin cambios, pero la introducción de interfaces significa que diferentes implementaciones de estas interfaces pueden ser fácilmente conectados. Figura 4-2 muestra este diseño.

|  |  |
| --- | --- |
| dieciséis | Capítulo 4 |
|  | Principios de arquitectura |



***La Figura 4-2.*** *gráfica de dependencia invertida.*

Dependencia de la inversión es una parte clave de la creación de aplicaciones débilmente acoplados, ya que los detalles de implementación pueden ser escritos a depender e implementar abstracciones de nivel superior, en lugar de al revés. Las aplicaciones resultantes son más comprobable, modular, y mantenible como resultado. La práctica de la inyección de dependencia es posible siguiendo el principio de la dependencia de la inversión.

**dependencias explícitas**

**Los métodos y las clases deben exigir explícitamente cualquier objeto colaboradoras que necesitan para funcionar correctamente.** constructores de clase proporcionan una oportunidad para las clases para identificar las cosas que necesitan para estar en un estado válido y que funcione correctamente. Si define las clases que se pueden construir y llamados, y que sólo funcionará correctamente si ciertos componentes globales o de infraestructura están en su lugar, estas clases son ser deshonesto con sus clientes. El contrato constructor está diciendo el cliente que sólo necesita las cosas especificadas (posiblemente nada si la clase es sólo usar un constructor por defecto), pero en tiempo de ejecución resulta que el objeto realmente se necesita algo más.

Siguiendo el principio dependencias explícita, sus clases y métodos están siendo honestos con sus clientes acerca de lo que necesitan para funcionar. Esto hace que el código sea más auto-documentado y sus contratos de codificación más fácil de usar, ya que los usuarios llegarán a confiar en que si proporcionan lo que se requiere en la forma de método o constructor parámetros, los objetos que están trabajando con la voluntad de comportarse correctamente en tiempo de ejecución .

**Responsabilidad individual**

El principio responsabilidad solo se aplica al diseño orientado a objetos, sino también puede ser considerado como un principio de arquitectura similar a la separación de las preocupaciones. Afirma que los objetos deben tener sólo una responsabilidad y que deberían tener una sola razón para cambiar. En concreto, la única situación en la que es necesario cambiar el objeto es si la forma en que lleva a cabo su responsabilidad debe ser uno

|  |  |
| --- | --- |
| 17 | Capítulo 4 |
|  | Principios de arquitectura |

actualizado. Siguiendo este principio ayuda a producir sistemas más débilmente acoplados y modulares, ya que muchos tipos de nuevos comportamientos pueden implementarse como nuevas clases, en lugar de mediante la adición de una responsabilidad adicional a las clases existentes. Añadiendo nuevas clases siempre es más seguro que el cambio de clases existentes, ya que ningún código todavía depende de las nuevas clases.

En una aplicación monolítica, podemos aplicar el principio de la responsabilidad individual en un alto nivel de las capas de la aplicación. Presentación responsabilidad debe permanecer en el proyecto de interfaz de usuario, mientras que la responsabilidad de acceso de datos debe mantenerse dentro de un proyecto de infraestructura. La lógica de negocio debe mantenerse en el proyecto núcleo de la aplicación, donde se puede comprobar fácilmente y puede evolucionar independientemente de otras responsabilidades.

Cuando este principio se aplica a la arquitectura de la aplicación, y llevado a su punto final lógico, se obtiene microservicios. Un microService dado debe tener una única responsabilidad. Si necesita extender el comportamiento de un sistema, por lo general es mejor hacerlo mediante la adición de microservicios adicionales, en lugar de mediante la adición de la responsabilidad a una ya existente.

[Más información sobre la arquitectura microservicios](http://aka.ms/MicroservicesEbook)

**No te repitas (DRY)**

La aplicación debe evitar especificar el comportamiento relacionado con un concepto particular en varios lugares, ya que es una fuente frecuente de errores. En algún momento, un cambio en los requisitos requerirá cambiar este comportamiento y la probabilidad de que al menos una instancia de la conducta dejará de ser actualizado dará lugar a un comportamiento inconsistente del sistema.

En lugar de duplicar la lógica, encapsular en una construcción de programación. Convertir esta construcción de la autoridad única sobre este comportamiento y tener cualquier otra parte de la aplicación que requiere este comportamiento utilizar la nueva construcción.

**Nota:** Evitar el comportamiento en conjunto de unión que es única coincidencia repetitivo. Por ejemplo, sólo porque dos constantes diferentes ambos tienen el mismo valor, eso no significa que usted debe tener sólo una constante, si conceptualmente se están refiriendo a cosas diferentes.

|  |  |
| --- | --- |
| 18 | Capítulo 4 |
|  | Principios de arquitectura |

**La ignorancia de persistencia**

**ignorancia de la persistencia** (PI) se refiere a los tipos que necesitan ser persistido, pero cuyo código no se ve afectada por la elección de la tecnología de persistencia. Tales tipos de .NET se refieren a veces como Plain Old CLR Objects (POÇOS), ya que no tienen que heredar de una clase base concreta o implementar una interfaz en particular. ignorancia de la persistencia es valiosa porque permite que el mismo modelo de negocio que se persistió en múltiples formas, que ofrece flexibilidad adicional para la aplicación. opciones de persistencia pueden cambiar con el tiempo, de una tecnología de base de datos a otro, o de otras formas de persistencia podrían ser requerida además de la que se inició la aplicación con (por ejemplo, utilizando una caché Redis o Azure Cosmos DB, además de una base de datos relacional).

Algunos ejemplos de violaciónes de este principio incluye:

* Una clase base requerida
* Una aplicación de interfaz necesario
* Clases responsables de salvarse a sí mismos (tales como el patrón Active Record)
* Requerido constructor por defecto
* propiedades que requieren virtual palabra clave
* Propiedades obligados a utilizar ciertos tipos (por ejemplo, propiedades de colección deben exponer

ICollection, No solo IEnumerable)

* atributos necesarios de persistencia específica

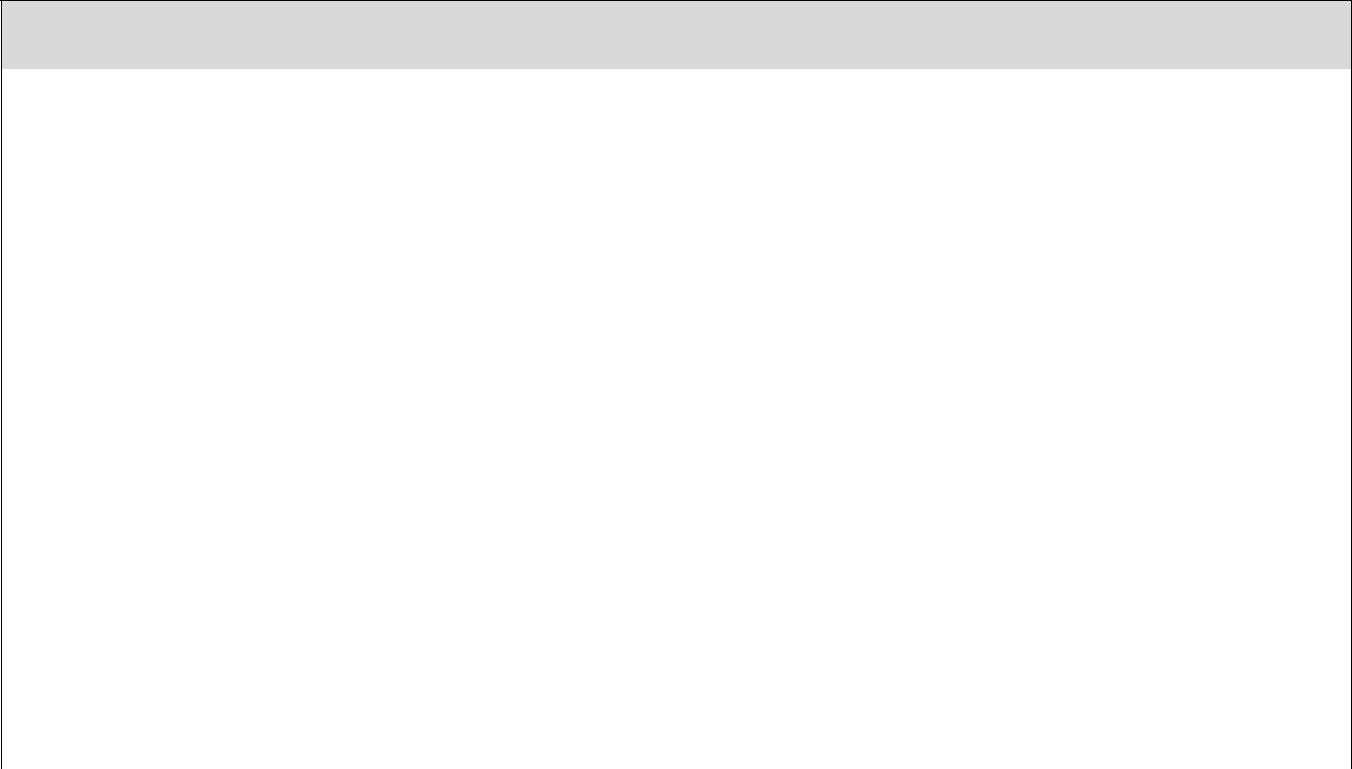
El requisito de que las clases tienen ninguna de las características o comportamientos anteriores añade acoplamiento entre los tipos que se persistido y la elección de la tecnología de persistencia, lo que hace más difícil la adopción de nuevas estrategias de acceso a datos en el futuro.

|  |  |
| --- | --- |
| 19 | Capítulo 4 |
|  | Principios de arquitectura |

**Contextos acotadas**

**contextos acotadas** son un patrón central en Driven Design-dominio. Proporcionan una manera de hacer frente a la complejidad de las grandes aplicaciones u organizaciones por dividirlo en módulos conceptuales separadas. Cada módulo conceptual representa entonces un contexto que se separa de otros contextos (por lo tanto, limitada), y puede evolucionar de manera independiente. Cada contexto delimitado ideal debería ser libre de elegir sus propios nombres para los conceptos dentro de ella y debe tener acceso exclusivo a su propia tienda de persistencia.

Como mínimo, las aplicaciones web individuales deben esforzarse por ser su propio contexto acotado, con su propia tienda de persistencia para su modelo de negocio, en lugar de compartir una base de datos con otras aplicaciones. La comunicación entre los contextos acotados se produce a través de interfaces de programación, en lugar de a través de una base de datos compartida, que permite la lógica de negocio y eventos que tendrá lugar en respuesta a los cambios que tienen lugar. contextos acotados se corresponden estrechamente a microservicios, que también son idealmente implementadas como propio individuo limitado contextos.



**Referencias - aplicaciones web modernas**

**Separación de intereses**

<http://deviq.com/separation-of-concerns/>

**La encapsulación**

<http://deviq.com/encapsulation/>

**Principio de Inversión de Dependencia**

<http://deviq.com/dependency-inversion-principle/>

**Explícita Dependencias Principio**

<http://deviq.com/explicit-dependencies-principle/>

**No te repitas**

<http://deviq.com/don-t-repeat-yourself/>

**La ignorancia de persistencia**

<http://deviq.com/persistence-ignorance/>

**Contexto acotada**

<https://martinfowler.com/bliki/BoundedContext.html>

**Fundamentos de Diseño de dominio-Driven**

<http://bit.ly/ddd-fundamentals>

**Sólidos principios de diseño orientado a objetos**

<http://bit.ly/SOLID-OOP>

|  |  |
| --- | --- |
| 20 | Capítulo 4 |
|  | Principios de arquitectura |

SECCIÓN 5

Arquitecturas de aplicaciones Web comunes

*“Si cree que la buena arquitectura es cara, prueba con mala arquitectura.”*

Brian Foote y José Yoder

Resumen

La mayoría de las aplicaciones tradicionales de .NET se despliegan como unidades individuales correspondientes a un una única aplicación web que se ejecuta dentro de un mismo dominio de aplicación ejecutable o IIS. Este es el modelo de implementación más sencilla y sirve para muchas aplicaciones públicas internas y pequeñas muy bien. Sin embargo, incluso teniendo en cuenta esta sola unidad de despliegue, la mayoría de las aplicaciones de negocio no triviales se benefician de una cierta separación lógica en varias capas.

¿Qué es una aplicación monolítica?

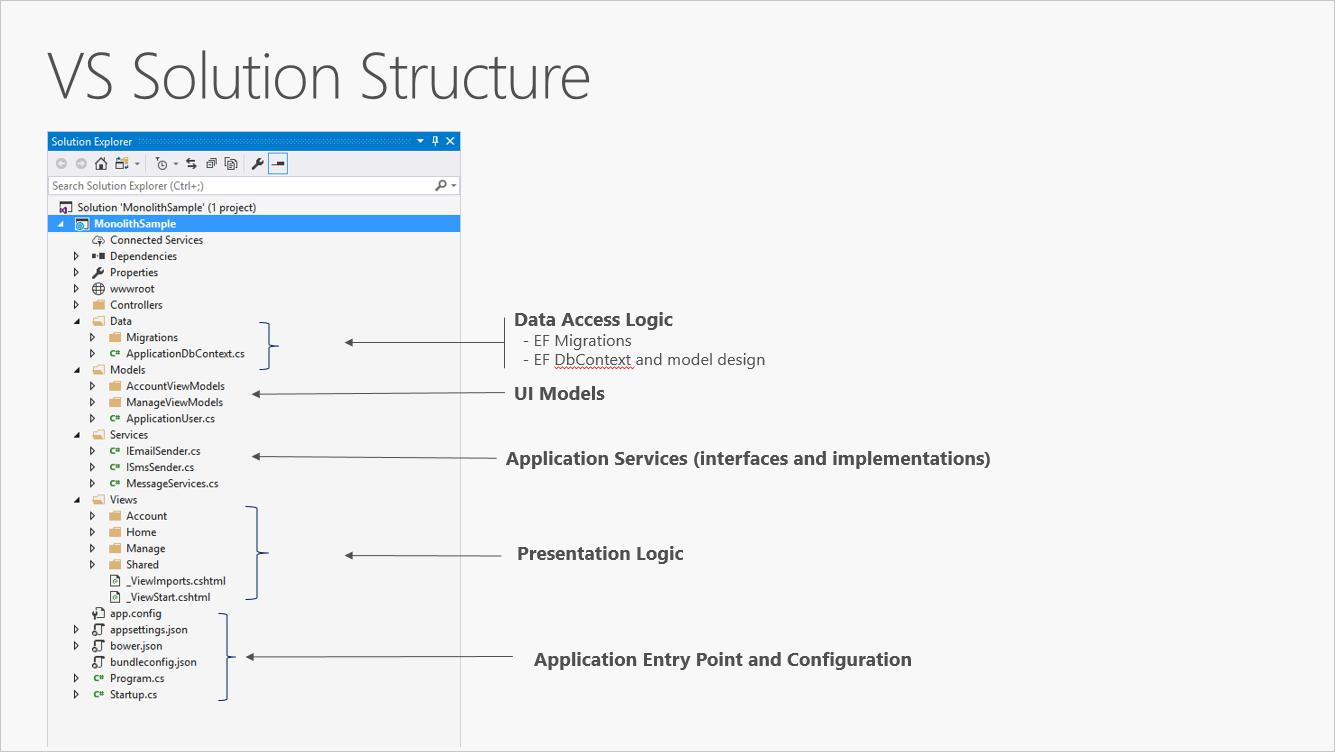
Una aplicación monolítica es uno que es totalmente autónomo, en términos de su comportamiento. Puede interactuar con otros servicios o almacenes de datos, mientras que la realización de sus operaciones, pero el núcleo de su comportamiento se ejecuta dentro de su propio proceso y toda la aplicación se implementa típicamente como una sola unidad. Si una aplicación de este tipo tiene que escalar horizontalmente, por lo general toda la aplicación se duplica a través de múltiples servidores o máquinas virtuales.

Todo-en-Uno aplicaciones

El menor número posible de proyectos para una arquitectura de aplicaciones es uno. En esta arquitectura, toda la lógica de la aplicación está contenido en un solo proyecto, compilado para un solo conjunto, y desplegado como una sola unidad.

Un nuevo proyecto de ASP.NET Core, ya sea creada en Visual Studio o desde la línea de comandos, que empieza como un simple “todo-en-uno” monolito. Contiene toda el comportamiento de la aplicación, incluyendo presentación, negocios, y la lógica de acceso a datos. La figura 5-1 muestra la estructura de archivos de una aplicación de un solo proyecto.

|  |  |
| --- | --- |
| 21 | Capítulo 5 |
|  | Arquitecturas de aplicaciones Web comunes |



**La Figura 5-1.** Un solo proyecto ASP.NET aplicación Core

En un único escenario de proyecto, la separación de las preocupaciones se logra mediante el uso de carpetas. La plantilla por defecto incluye carpetas separadas para las responsabilidades del patrón MVC de modelos, vistas y controladores, así como las carpetas adicionales para datos y servicios. En este arreglo, detalles de la presentación se deben limitar lo más posible a la carpeta Vistas y detalles de la implementación de acceso a datos deberían limitarse a clases mantienen en la carpeta de datos. La lógica de negocio debe residir en los servicios y las clases dentro de la carpeta Modelos.

Aunque simple, la solución monolítica de un solo proyecto tiene algunas desventajas. Como el tamaño y la complejidad del proyecto crece, el número de archivos y carpetas seguirá creciendo también. preocupaciones de interfaz de usuario (modelos, vistas, controladores) residen en varias carpetas, que no están agrupados en orden alfabético. Este problema sólo empeora cuando las construcciones a nivel de interfaz de usuario adicionales, tales como filtros o ModelBinders, se añaden en sus propias carpetas. La lógica de negocio se reparte entre los modelos y las carpetas Servicios, y no hay ninguna indicación clara de qué clases en las carpetas que dependerá de lo que los demás. Esta falta de organización a nivel de proyecto con frecuencia conduce a[código espagueti.](http://deviq.com/spaghetti-code/)

Para abordar estas cuestiones, las aplicaciones se desarrollan a menudo en soluciones multi-proyecto, cuando se estime que cada proyecto de residir en una capa particular de la aplicación.

¿Cuáles son las capas?

Dado que las aplicaciones crecen en complejidad, una manera de manejar esa complejidad es romper la aplicación de acuerdo a sus responsabilidades o preocupaciones. Esto sigue a la separación de las preocupaciones principales y puede ayudar a mantener una base de código crecimiento organizado de tal manera que los desarrolladores pueden encontrar fácilmente en donde se implementa cierta funcionalidad. arquitectura en capas ofrece muchas ventajas más allá de la organización del código, sin embargo.

|  |  |
| --- | --- |
| 22 | Capítulo 5 |
|  | Arquitecturas de aplicaciones Web comunes |

Mediante la organización de código en capas, la funcionalidad común de bajo nivel puede ser reutilizado en toda la aplicación. Esta reutilización es beneficioso porque significa menos código debe ser escrito y porque puede permitir que la aplicación estandarizar en una sola aplicación, siguiendo el principio DRY.

Con una arquitectura en capas, las aplicaciones pueden hacer cumplir las restricciones sobre la que las capas se pueden comunicar con otras capas. Esto ayuda a lograr la encapsulación. Cuando una capa se cambia o se sustituye, sólo aquellas capas que trabajan con ella deben ser impactados. Al limitar el cual las capas dependen de qué otras capas, el impacto de los cambios se puede mitigar de manera que un solo cambio no afecta a toda la aplicación.

Y capas (encapsulación) hacen que sea mucho más fácil para reemplazar la funcionalidad de la aplicación. Por ejemplo, una aplicación podría utilizar inicialmente su propia base de datos de SQL Server para la persistencia, pero más tarde se podría optar por utilizar una estrategia de persistencia basada en la nube, o uno detrás de una API web. Si la aplicación ha encapsulado adecuadamente su aplicación persistencia dentro de una capa lógica, esa capa específica de SQL Server podría ser reemplazado por uno nuevo la aplicación de la misma interfaz pública.

Además de la posibilidad de cambiar de implementaciones en respuesta a los cambios futuros en los requisitos, las capas de aplicación también puede hacer que sea más fácil de extraerse implementaciones para propósitos de prueba. En lugar de tener que escribir pruebas que operan contra la capa de datos real o capa de interfaz de usuario de la aplicación, estas capas pueden ser reemplazados en el tiempo de prueba con las implementaciones falsos que proporcionan respuestas conocidas a las solicitudes. Esto normalmente hace pruebas mucho más fácil escribir y mucho más rápido de ejecutar si se compara con la ejecución de pruebas de nuevo muy infraestructura de la aplicación.

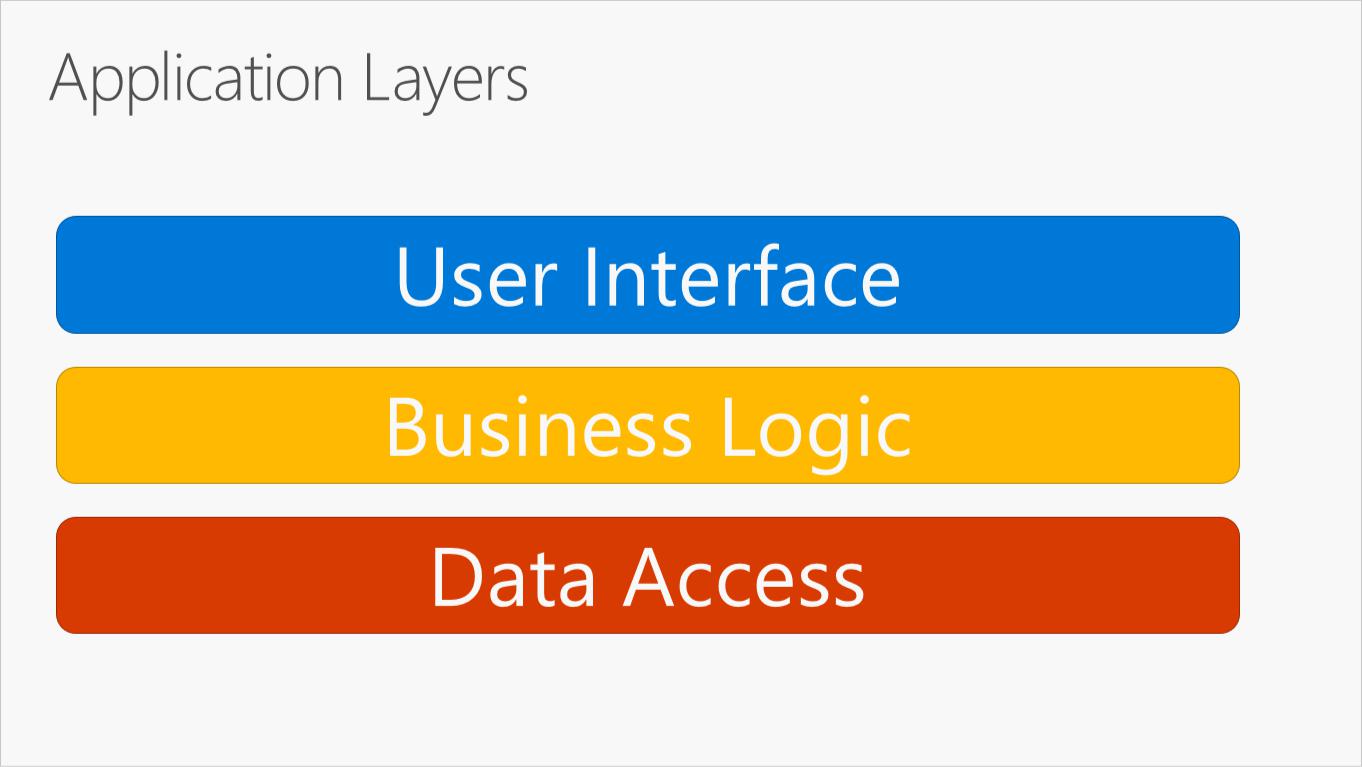
estratificación lógica es una técnica común para mejorar la organización de código en aplicaciones de software empresarial, y hay varias formas en las que el código se pueden organizar en capas.

**Nota:** *Capas*representar separación lógica dentro de la aplicación. Si la lógica de aplicación se distribuye físicamente a servidores o procesos separados, estos objetivos de implementación física separadas se denominan niveles. Es posible, y bastante común, de tener una aplicación N-capa que se implementa en un solo nivel.

|  |  |
| --- | --- |
| 23 | Capítulo 5 |
|  | Arquitecturas de aplicaciones Web comunes |

Las aplicaciones tradicionales “Arquitectura N-Layer”

La organización más común de la lógica de aplicación en capas se muestra en la Figura 5-2.



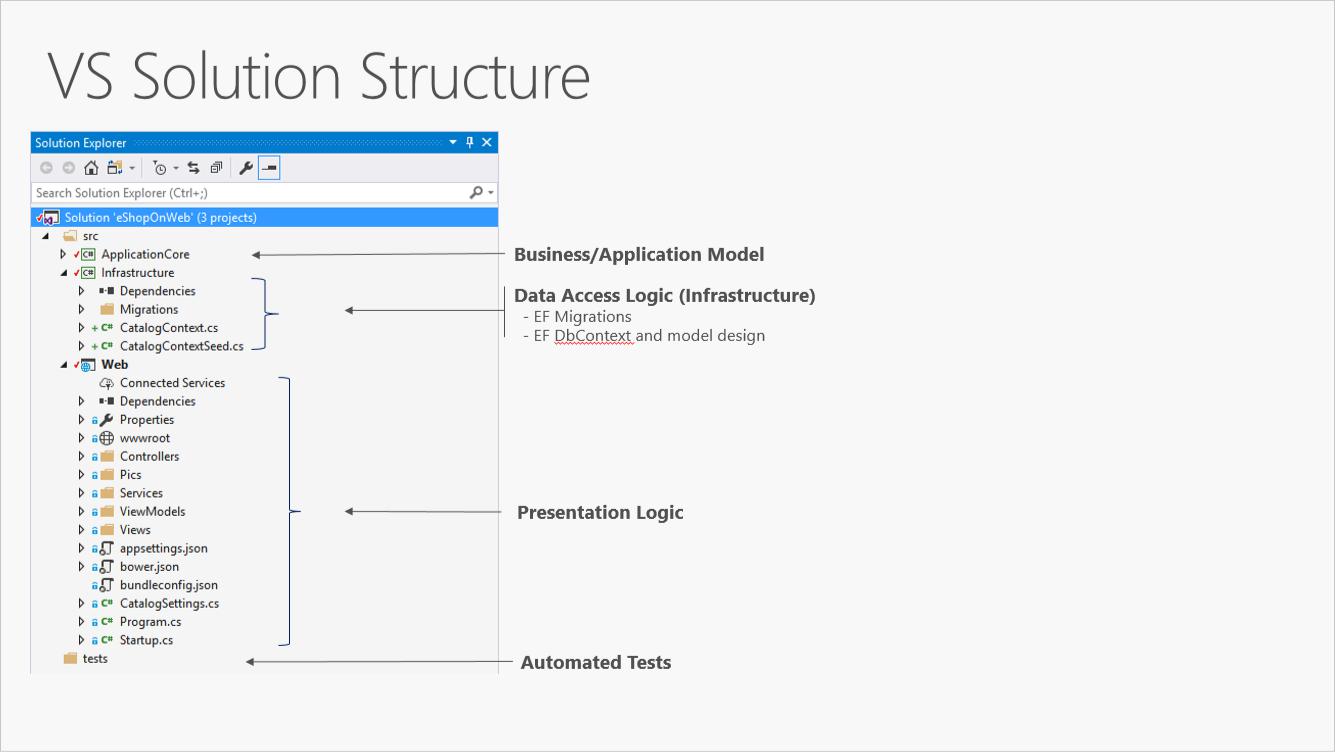
**La Figura 5-2.** capas de aplicación típicos.

Estas capas son frecuentemente abreviado como interfaz de usuario, BLL (la lógica de negocios de capa), y DAL (Data Access Layer). Usando esta arquitectura, los usuarios hacen peticiones a través de la capa de interfaz de usuario, que interactúa sólo con el BLL. El BLL, a su vez, puede llamar a la DAL para las solicitudes de acceso a datos. La capa de interfaz de usuario no debe hacer cualquier solicitud a la DAL directamente, ni debe interactuar con persistencia directamente a través de otros medios. Del mismo modo, el BLL sólo debe interactuar con la persistencia pasando a través de la DAL. De esta manera, cada capa tiene su propia responsabilidad conocido.

Una desventaja de este método de estratificación tradicional es que el tiempo de compilación dependencias corren desde la parte superior a la parte inferior. Es decir, la capa de interfaz de usuario depende de la BLL, que depende de la DAL. Esto significa que el BLL, que por lo general tiene la lógica más importante en la aplicación, depende de los detalles de implementación de acceso a datos (y, a menudo en la existencia de una base de datos). Prueba de lógica de negocio en este tipo de arquitectura es a menudo difícil, que requiere una base de datos de prueba. El principio de la dependencia de inversión se puede utilizar para hacer frente a este problema, como se verá en la siguiente sección.

|  |  |
| --- | --- |
| 24 | Capítulo 5 |
|  | Arquitecturas de aplicaciones Web comunes |

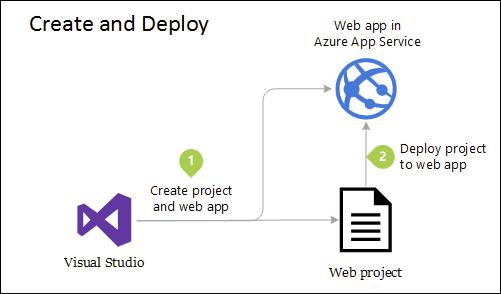
Figura 5-3 muestra un ejemplo de solución, rompiendo la aplicación en tres proyectos por responsabilidad (o capa).



***La Figura 5-3.*** *Una simple aplicación monolítica con tres proyectos.*

Aunque esta aplicación utiliza varios proyectos para fines de organización, todavía se implementa como una sola unidad y sus clientes van a interactuar con él como una sola aplicación web. Esto permite un proceso de implementación muy simple. La figura 5-4 muestra cómo una aplicación como podría ser alojado el uso de Windows Azure.

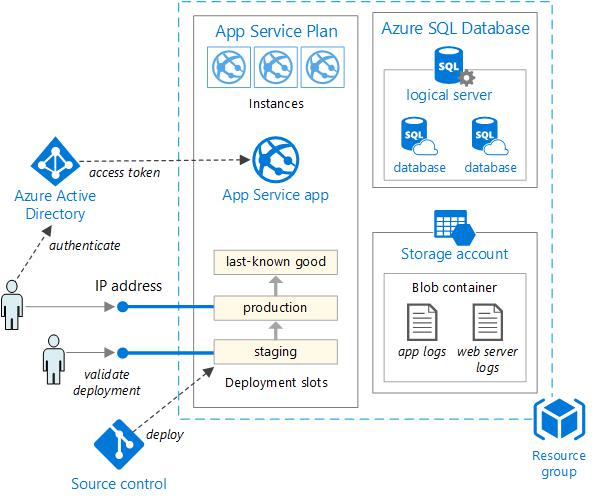
|  |  |
| --- | --- |
| 25 | Capítulo 5 |
|  | Arquitecturas de aplicaciones Web comunes |



***La Figura 5-4.*** *implementación simple de aplicación web de Azure*

A medida que crecen las necesidades de aplicación, se pueden requerir soluciones de implementación más complejos y robustos. Figura 5-5 muestra un ejemplo de un plan de despliegue más complejo que soporta capacidades adicionales.

|  |  |
| --- | --- |
| 26 | Capítulo 5 |
|  | Arquitecturas de aplicaciones Web comunes |



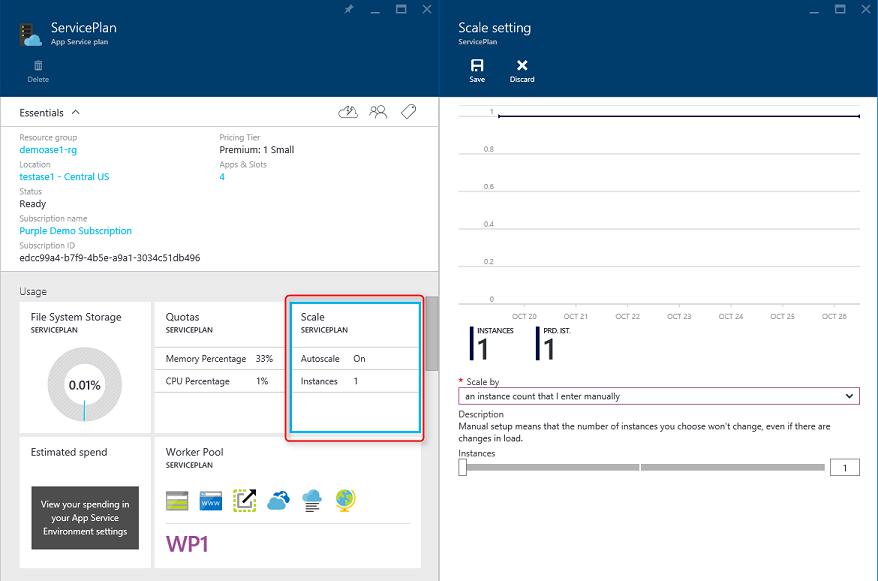
***Figura 5-5.*** *La implementación de una aplicación web para una aplicación de servicio Azure*

Internamente, la organización de este proyecto en varios proyectos basados ​​en la responsabilidad mejora la capacidad de mantenimiento de la aplicación.

Esta unidad se puede escalar hacia arriba o hacia fuera para tomar ventaja de la nube basada en la escalabilidad bajo demanda. La ampliación de medios de adición adicional de la CPU, memoria, espacio en disco, u otros recursos para el servidor (s) de alojamiento de su aplicación. Escalar hacia fuera significa añadir instancias adicionales de tales servidores, si se trata de servidores físicos o máquinas virtuales. Cuando su aplicación se encuentra alojado en varias instancias, un equilibrador de carga se utiliza para asignar solicitudes a las instancias de aplicaciones individuales.

El método más sencillo para escalar una aplicación web en Azure es configurar manualmente la escala en el Plan de Aplicación de servicio de la aplicación. Figura 5-6 muestra la pantalla de tablero de mandos Azure apropiado para configurar cómo muchos casos están sirviendo a una aplicación.

|  |  |
| --- | --- |
| 27 | Capítulo 5 |
|  | Arquitecturas de aplicaciones Web comunes |



***Figura 5-6.*** *Plan de Servicio de Aplicación de escala en Azure.*

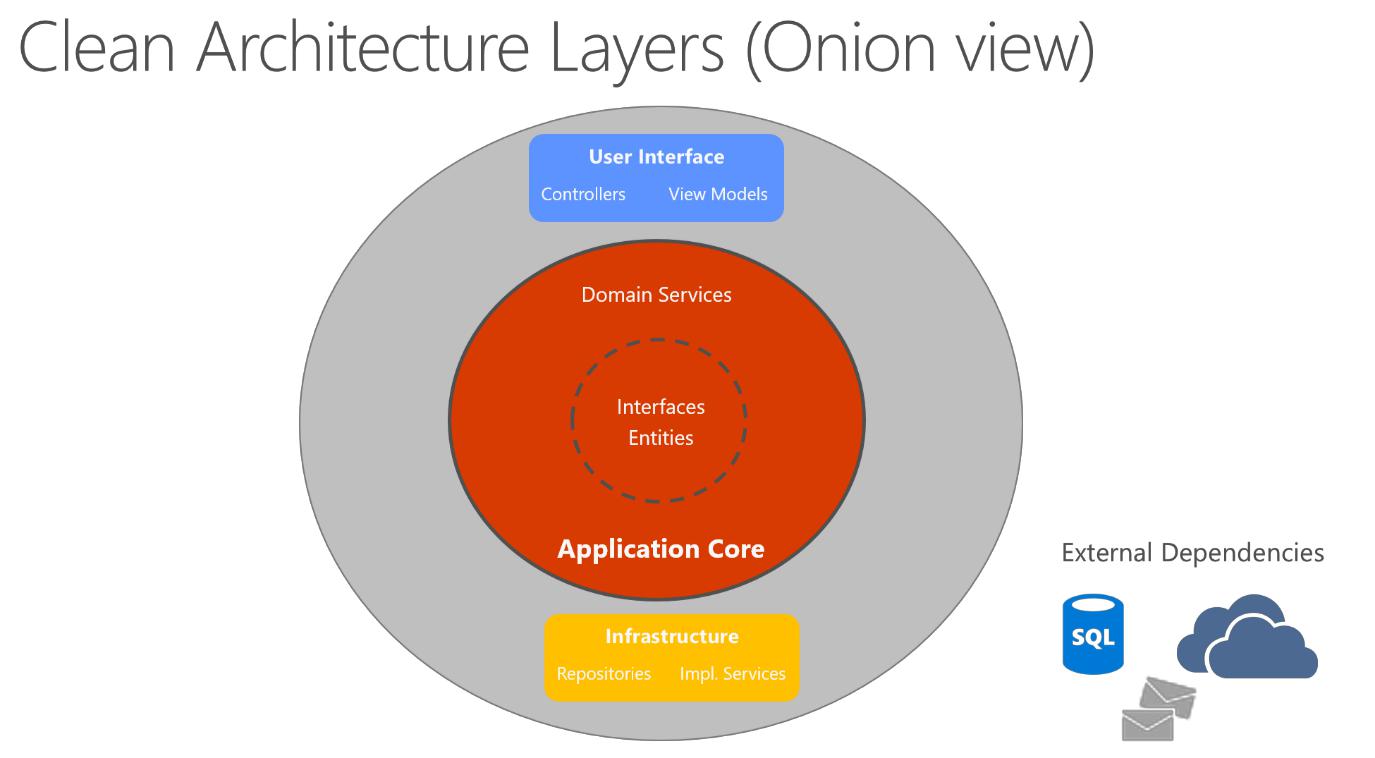
arquitectura limpia

Las aplicaciones que siguen a la Dependencia Inversión Principio, así como los principios de diseño-Driven dominio (DDD) tienden a llegar a una arquitectura similar. Esta arquitectura ha pasado por muchos nombres en los últimos años. Uno de los primeros nombres fue hexagonal Arquitectura, seguido de Puertos-y-adaptadores. Más recientemente, se ha citado como el[arquitectura de cebolla](http://jeffreypalermo.com/blog/the-onion-architecture-part-1/) o [Arquitectura limpio.](https://8thlight.com/blog/uncle-bob/2012/08/13/the-clean-architecture.html) Es este último nombre, arquitectura limpia, que se utiliza como base para la descripción de la arquitectura en este libro electrónico.

**Nota:** El término arquitectura limpia se puede aplicar a las aplicaciones que se construyen utilizando los principios de DDD, así como a aquellos que no están construidos con DDD. En el caso de la primera, esta combinación puede ser referido como “Arquitectura DDD limpio”.

arquitectura limpia pone la lógica de negocio y el modelo de aplicación en el centro de la aplicación. En lugar de tener la lógica de negocio dependen de acceso a datos u otros problemas de infraestructura, esta dependencia se invierte: la infraestructura y los detalles de aplicación dependen del núcleo de la aplicación. Esto se consigue mediante abstracciones que definen, o interfaces, en el núcleo de la aplicación, que se implementan a continuación, por los tipos definidos en la capa de infraestructura. Una forma común de visualizar esta arquitectura es utilizar una serie de círculos concéntricos, similares a una cebolla. Figura 5-7 muestra un ejemplo de este estilo de representación arquitectónica.

|  |  |
| --- | --- |
| 28 | Capítulo 5 |
|  | Arquitecturas de aplicaciones Web comunes |

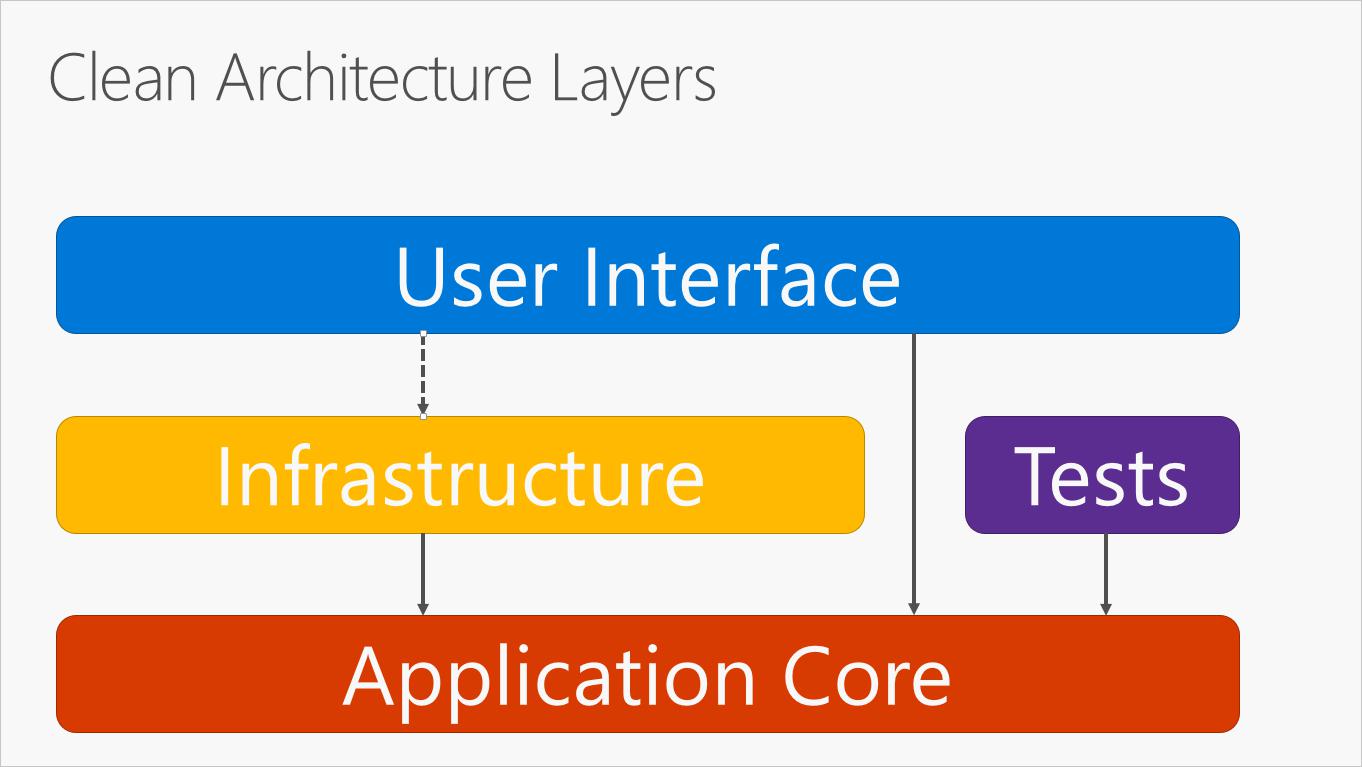


**La Figura 5-7.** Arquitectura limpio; Ver la cebolla

En este diagrama, dependencias fluyen hacia el círculo más interior. Por lo tanto, se puede ver que el núcleo de la aplicación (que toma su nombre de su posición en el núcleo de este diagrama) no tiene ninguna dependencia de otras capas de aplicación. En el mismo centro son entidades e interfaces de la aplicación. Justo fuera, pero aún en el núcleo de la aplicación, son servicios de dominio, que típicamente implementan interfaces definidas en el círculo interno. Fuera del núcleo de la aplicación, tanto la interfaz de usuario y las capas de infraestructura dependen del núcleo de la aplicación, pero no en otros (necesariamente).

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

|  |  |
| --- | --- |
| 29 | Capítulo 5 |
|  | Arquitecturas de aplicaciones Web comunes |

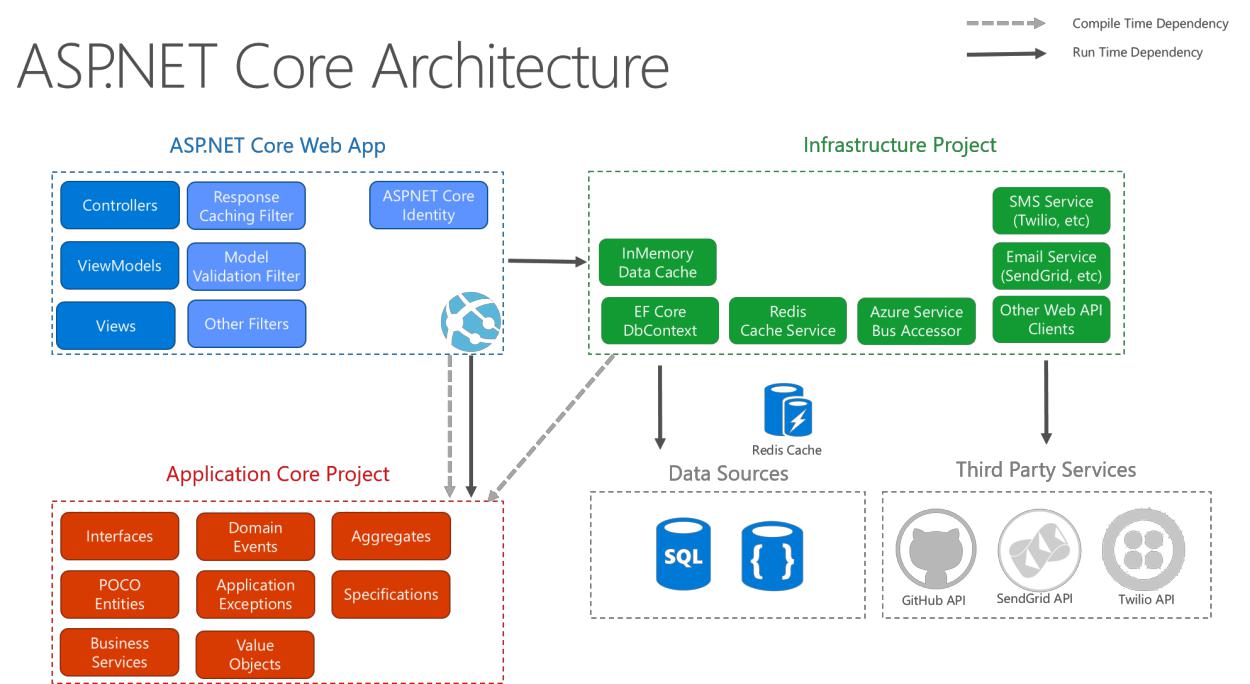


***Figura 5-8.*** *Arquitectura limpio; vista de capa horizontal*

Tenga en cuenta que las flechas continuas representan las dependencias de tiempo de compilación, mientras que la flecha de trazos representa una dependencia de tiempo de ejecución de sólo. Utilizando la arquitectura limpia, la capa de interfaz de usuario funciona con interfaces definidas en el núcleo de la aplicación en tiempo de compilación, e idealmente no debería tener ningún conocimiento de los tipos de implantación definidos en la capa de Infraestructura. En tiempo de ejecución, sin embargo, se requerirá que estos tipos de implementación para la aplicación a ejecutar, por lo que tendrán que estar presentes y por cable hasta las interfaces núcleo de la aplicación a través de la inyección de dependencia.

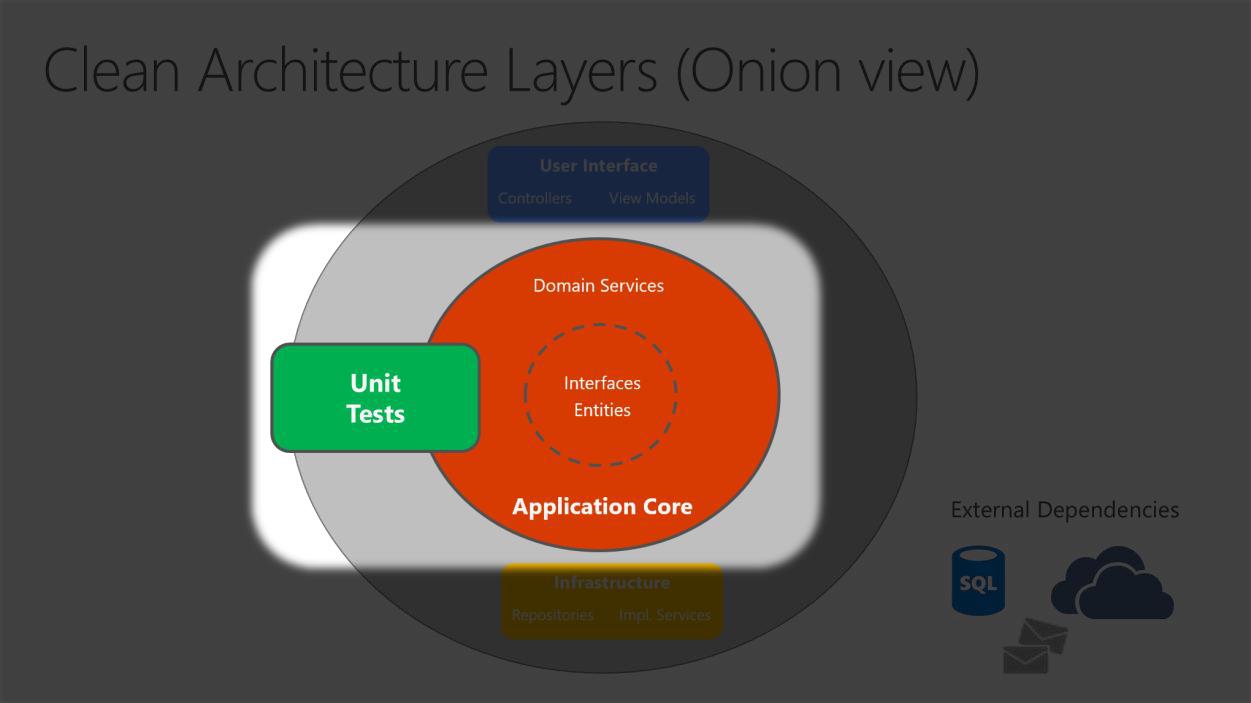
La Figura 5-9 muestra una vista más detallada de la arquitectura de una aplicación ASP.NET Core cuando construido siguiendo estas recomendaciones.

|  |  |
| --- | --- |
| 30 | Capítulo 5 |
|  | Arquitecturas de aplicaciones Web comunes |



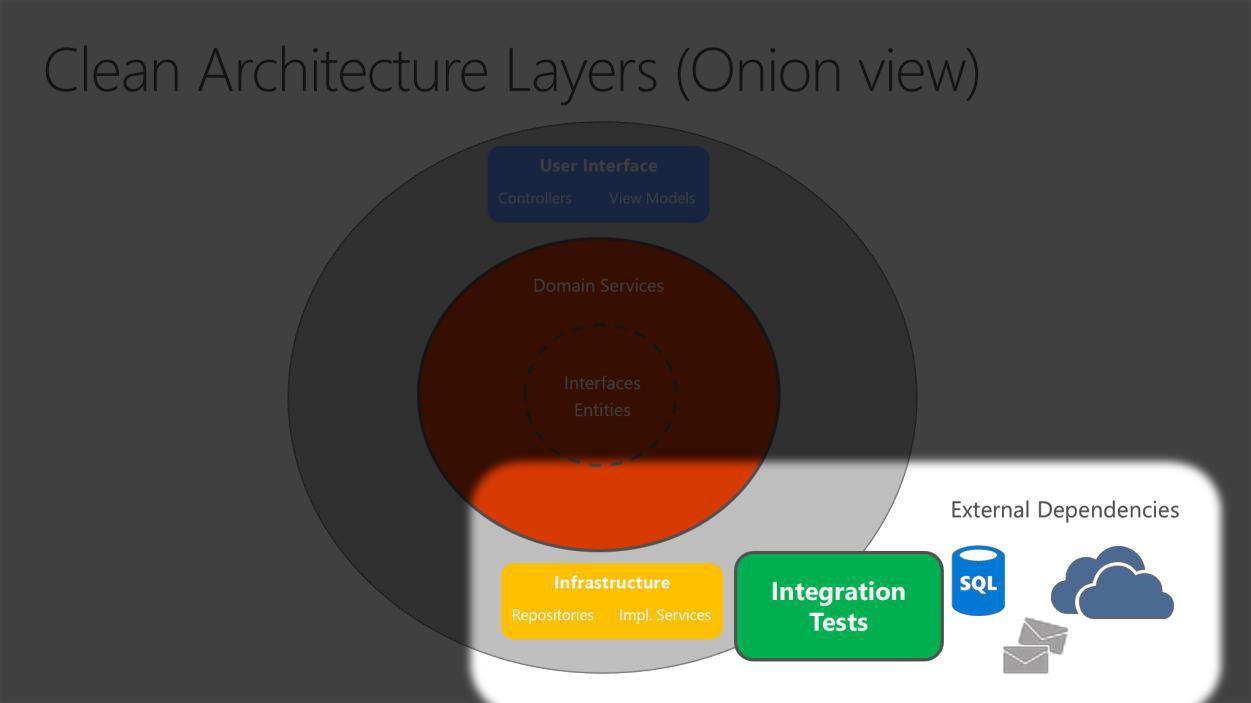
***Figura 5-9.*** *ASP.NET Arquitectura del núcleo siguiente diagrama arquitectura limpia.*

Debido a que el núcleo de la aplicación no depende de Infraestructura, es muy fácil de escribir pruebas unitarias automatizadas para esta capa. Las figuras 5-10 y 5-11 muestran cómo pruebas encajan en esta arquitectura.



***Figura 5-10.*** *Prueba de la unidad de aplicaciones Core en forma aislada.*

|  |  |
| --- | --- |
| 31 | Capítulo 5 |
|  | Arquitecturas de aplicaciones Web comunes |

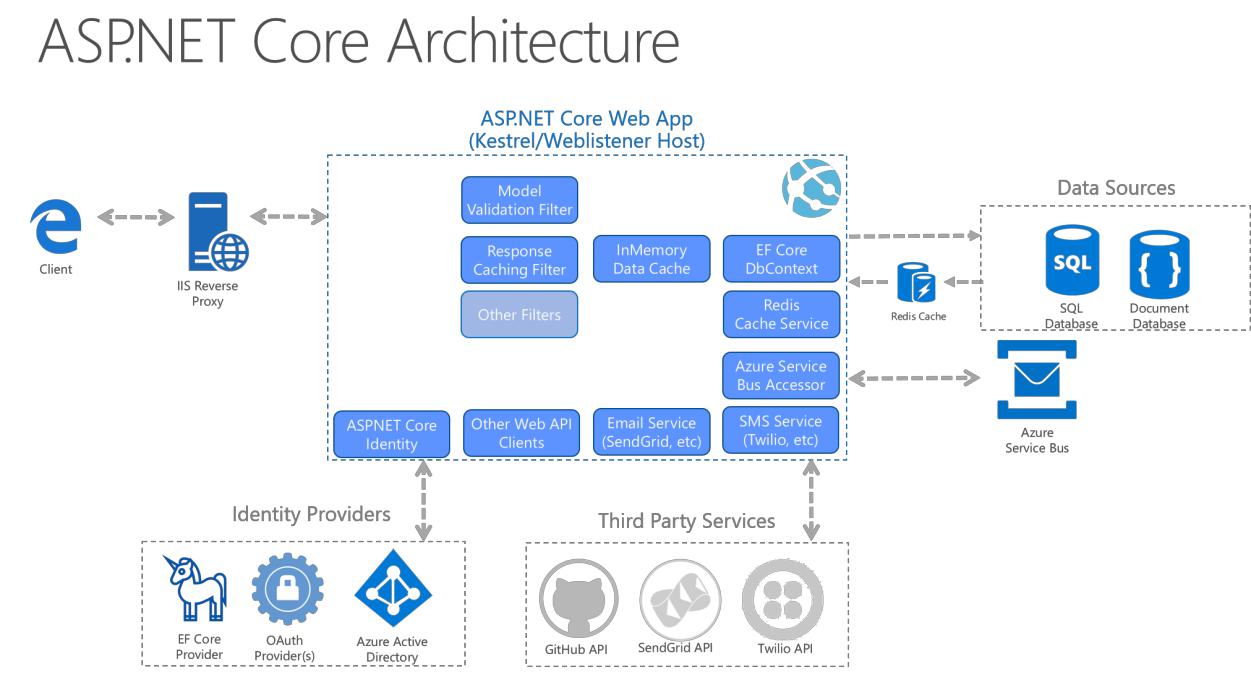


***Figura 5-11.*** *Integración de las implementaciones de infraestructura de pruebas con dependencias externas.*

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

|  |  |
| --- | --- |
| 32 | Capítulo 5 |
|  | Arquitecturas de aplicaciones Web comunes |

Para aplicaciones monolíticas el núcleo de la aplicación, la interfaz de proyectos de infraestructura, y los usuarios están todos ejecuta como una sola aplicación. La arquitectura de la aplicación en tiempo de ejecución podría parecerse a la Figura 5-12.

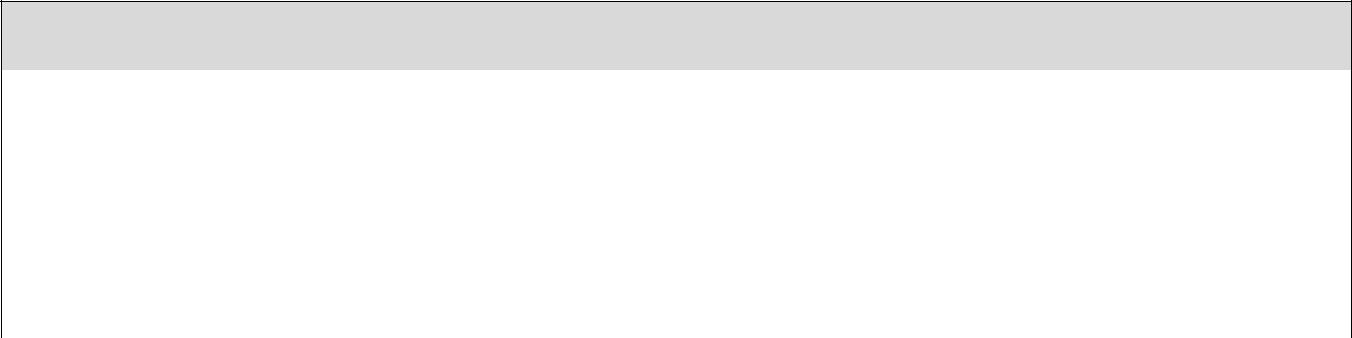


***Figura 5-12.*** *arquitectura de tiempo de ejecución de una aplicación Core muestra de ASP.NET.*

**Código organizar en arquitectura limpia**

En una solución de arquitectura limpia, cada proyecto tiene responsabilidades claras. Como tal, ciertos tipos pertenecerán en cada proyecto y que con frecuencia van a encontrar las carpetas correspondientes a estos tipos en el proyecto correspondiente.

El núcleo de la aplicación mantiene el modelo de negocio, que incluye a las entidades, servicios e interfaces. Estas interfaces incluyen abstracciones para las operaciones que se pueden realizar utilizando la infraestructura, tales como acceso a los datos, el acceso al sistema de archivos, llamadas de red, etc. A veces los servicios o interfaces definidas en esta capa tendrán que trabajar con tipos no son de entidad que no tienen dependencias en la interfaz de usuario o Infraestructura. Estos pueden ser definidos como simples objetos de transferencia de datos (DTO).



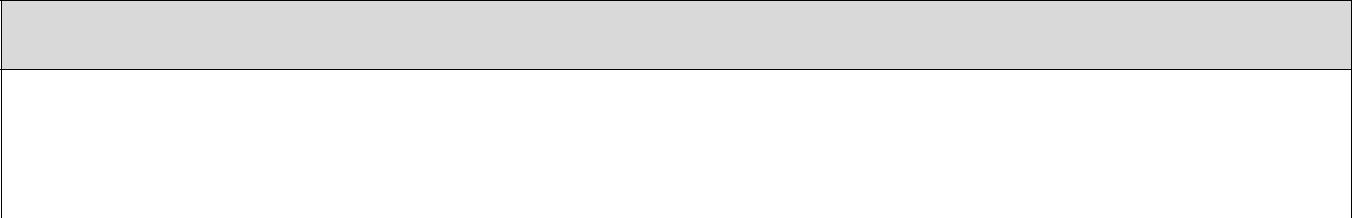
**Tipos de núcleo Aplicación**

* Entidades (clases del modelo de negocio que se conservan) y Áridos
* Interfaces
* Servicios
* DTO
* Presupuesto
* excepciones

El proyecto de infraestructura incluirá normalmente las implementaciones de acceso a datos. En una aplicación web ASP.NET típica Core, esto incluirá la Entidad marco DbContext, cualquier Core migraciones EF que han sido definidos, y clases de implementación de acceso a datos. La forma más común de código abstracto aplicación de acceso a datos es con el[patrón de diseño del repositorio.](http://deviq.com/repository-pattern/)

|  |  |
| --- | --- |
| 33 | Capítulo 5 |
|  | Arquitecturas de aplicaciones Web comunes |

Además de las implementaciones de acceso a datos, el proyecto de infraestructura debe contener implementaciones de los servicios que deben interactuar con las preocupaciones de infraestructura. Estos servicios deben implementar interfaces definidas en el núcleo de la aplicación, y así Infraestructura deben tener una referencia al proyecto de aplicación Core. Si usted encuentra que usted tiene servicios definidos en infraestructura que no dependen de ningún tipo relacionados con la infraestructura, a ver si puede moverlos en su proyecto núcleo de la aplicación. En general, si se puede mover en servicios de aplicaciones Core (sin añadir dependencias de este proyecto), debe hacerlo.



**Tipos de infraestructura**

* tipos EF núcleo (DbContext, migraciones)
* tipos de implementación de acceso a datos (repositorios)
* servicios de infraestructura específica (FileLogger, SmtpNotifier, etc.)

La capa de interfaz de usuario en una aplicación ASP.NET Core MVC será el punto de entrada para la aplicación y será un proyecto Core MVC ASP.NET (Razor Páginas o). Este proyecto deberá hacer referencia al proyecto de núcleo de la aplicación, y sus tipos debe interactuar con la infraestructura estrictamente a través de interfaces definidas en la Solicitud Core. Sin instanciación directa de la (o las llamadas estáticas a) tipos de capas de infraestructura deben ser permitidos en la capa de interfaz de usuario. Su aplicación puede[evitar hacer referencia al proyecto de infraestructura,](https://ardalis.com/avoid-referencing-infrastructure-in-visual-studio-solutions)sin dejar de utilizar sus tipos en tiempo de ejecución. Sólo los servicios que funcionan con tipos de capa de interfaz de usuario (tales como tipos ViewModel) deben ser definidas en esta capa. Mueva cualquier otro servicio en capas de infraestructura o núcleo de la aplicación, y evitar definir la lógica de negocio en la capa de interfaz de usuario.



**Tipos de capa de interfaz de usuario**

* controladores
* filtros
* páginas
* Servicios
* Puntos de vista
* ViewModels
* Puesta en marcha

La clase de inicio es responsable de la configuración de la aplicación, y para el cableado de tipos de implementación de las interfaces, lo que permite la inyección de dependencias para que funcione correctamente en tiempo de ejecución.

**Nota:** Para cablear la inyección de dependencia en ConfigureServices en el archivo del proyecto Startup.cs interfaz de usuario, el proyecto puede necesitar para hacer referencia al proyecto de infraestructura. Esta dependencia se puede eliminar, lo más fácilmente mediante el uso de un contenedor DI personalizado. A los efectos de esta muestra, el método más sencillo es permitir que el proyecto de interfaz de usuario para hacer referencia al proyecto de infraestructura.

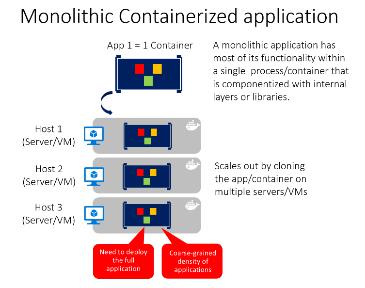
Las aplicaciones monolíticas y Contenedores

Se puede construir una aplicación web basada única y monolítica despliegue o servicio y desplegarlo como un contenedor. Dentro de la aplicación, puede que no sea monolítico, sino organizada en varias bibliotecas,

|  |  |
| --- | --- |
| 34 | Capítulo 5 |
|  | Arquitecturas de aplicaciones Web comunes |

componentes o capas. Externamente se trata de un solo recipiente como un solo proceso, aplicación web única o servicio único.

Para gestionar este modelo, se implementa un único contenedor para representar la aplicación. A escala, simplemente añadir más copias con un equilibrador de carga en el frente. La simplicidad viene de la gestión de un solo despliegue en un único contenedor o VM.



***Figura 5-13.*** *arquitectura de la aplicación monolítica*

Puede incluir múltiples componentes / bibliotecas o capas internas dentro de cada contenedor, como se ilustra en la figura 5-13. Pero, siguiendo el principio de contenedores de “un recipiente hace una cosa, y lo hace en un solo proceso”, el patrón monolítica podría ser un conflicto.

La desventaja de este enfoque viene si / cuando la aplicación crece, lo que requiere que a escala. Si toda la aplicación a escala, no es realmente un problema. Sin embargo, en la mayoría de los casos, algunas partes de la aplicación son los cuellos de botella que requieren de escala, mientras que otros componentes se utilizan menos.

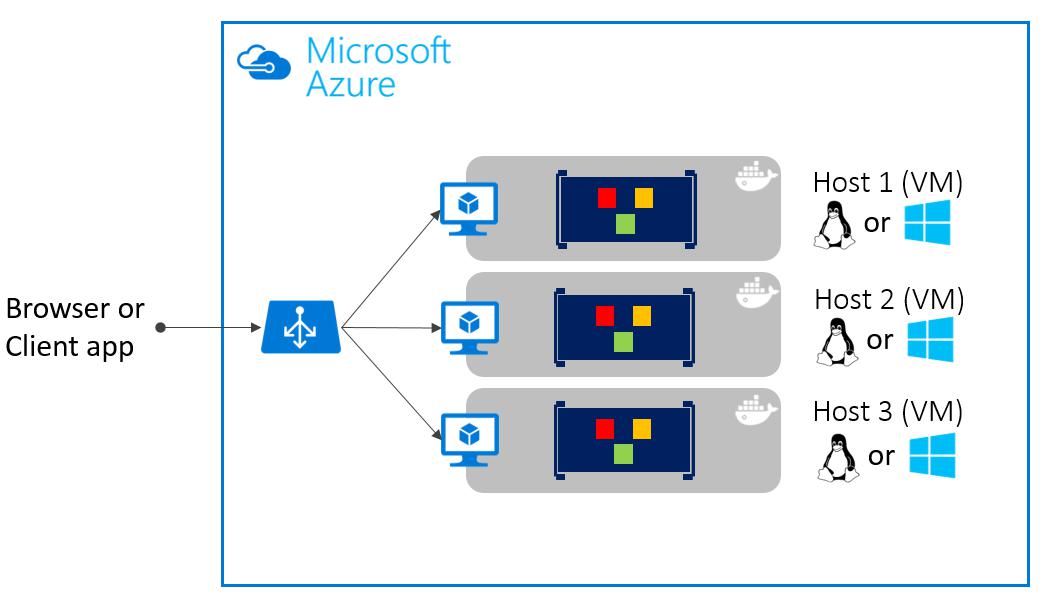
Usando el ejemplo típico de comercio electrónico; lo que es probable que necesite escala es el componente de información del producto. Muchos más clientes navegar por productos que comprarlos. Más clientes utilizan su cesta de usar la tubería de pago. Menos clientes añadir comentarios o ver su historial de compras. Y es probable que sólo tiene un puñado de empleados, en una sola región, que necesidad de gestionar los contenidos y campañas de marketing. Escalando el diseño monolítico, todo el código se despliega en múltiples ocasiones.

Además del problema de la escala de todo, los cambios en un solo componente requieren nuevas pruebas completa de toda la aplicación, y una redistribución completa de todos los casos.

|  |  |
| --- | --- |
| 35 | Capítulo 5 |
|  | Arquitecturas de aplicaciones Web comunes |

El enfoque monolítico es común, y muchas organizaciones están desarrollando con este enfoque arquitectónico. Muchos están teniendo buenos resultados suficientes, mientras que otros están golpeando límites. Muchos diseñado sus aplicaciones en este modelo, ya que las herramientas y la infraestructura eran demasiado difíciles de construir arquitecturas orientadas a servicios (SOA), y ellos no vieron la necesidad - hasta que la aplicación creció. Si encuentra que está golpeando los límites del enfoque monolítico, rompiendo la aplicación hasta que pueda mejores envases de apalancamiento y microservicios puede ser el siguiente paso lógico.

Despliegue de aplicaciones monolíticas en Microsoft Azure se puede lograr utilizando máquinas virtuales dedicados para cada instancia. Utilizando[Scale Sets Azure VM,](https://azure.microsoft.com/en-us/documentation/services/virtual-machine-scale-sets/) se puede escalar fácilmente las máquinas virtuales. [Azure Servicios App](https://azure.microsoft.com/en-us/services/app-service/) puede ejecutar aplicaciones monolíticas y escalar fácilmente casos sin tener que gestionar las máquinas virtuales. Azure App Servicios puede ejecutar instancias únicas de recipientes acoplables, así, lo que simplifica la implementación. El uso del estibador, se puede implementar una única máquina virtual como anfitrión del estibador, y ejecutar múltiples instancias. Utilizando el equilibrador Azure, como se muestra en la Figura 5-14, puede administrar escala.



***Figura 5-14.*** *Varios hosts ampliación a cabo una sola aplicación acoplable*

El despliegue de los diversos anfitriones se puede controlar con técnicas de despliegue tradicionales. Los anfitriones acoplables pueden ser manejados con comandos como**carrera ventana acoplable** realizado de forma manual o mediante la automatización tales como la entrega continua (CD) tuberías.

**aplicación monolítica desplegado como un contenedor**

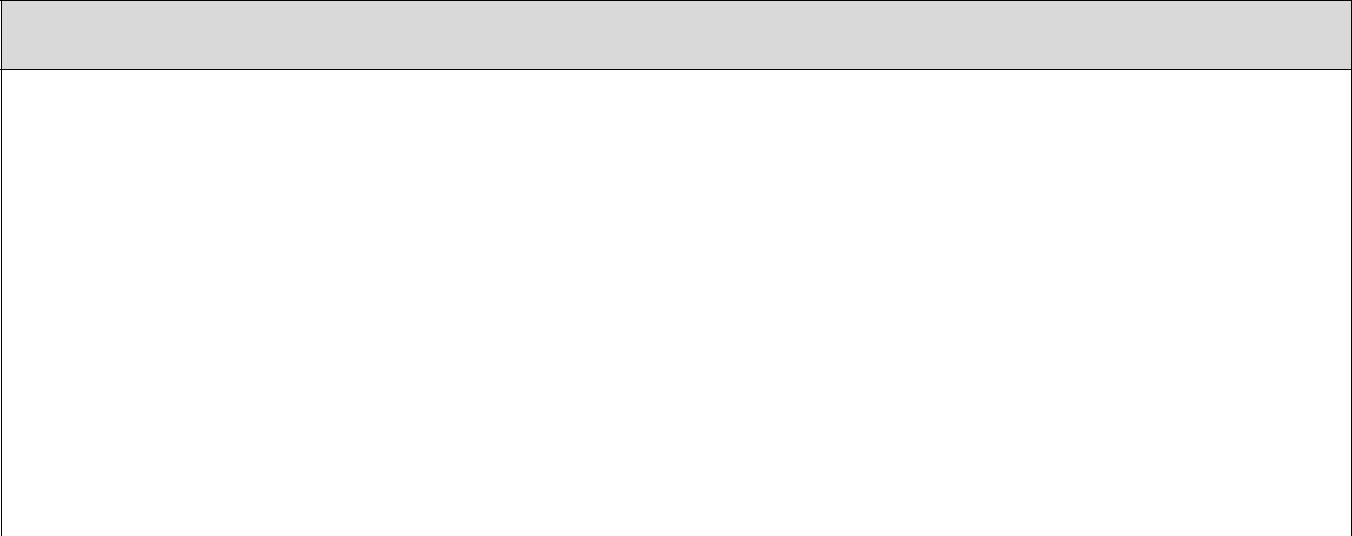
Hay beneficios del uso de contenedores para gestionar implementaciones de aplicaciones monolíticas. Escalado las instancias de contenedores es mucho más rápido y más fácil que el despliegue de máquinas virtuales adicional. Incluso cuando se utiliza la escala VM Establece para escalar las máquinas virtuales, se toman el tiempo a la instancia. Cuando se implementa como instancias de aplicaciones, la configuración de la aplicación es administrado como parte de la máquina virtual.

La implementación de actualizaciones como imágenes Docker es mucho más rápido y eficiente de la red. Docker Imágenes suelen comenzar en cuestión de segundos, lo que acelera los despliegues. Derribar un ejemplo acoplable es tan fácil como la emisión de una**parada ventana acoplable** de comandos, por lo general completar en menos de un segundo.

Como contenedores son intrínsecamente inmutables por su diseño, que no tendrá que preocuparse por las máquinas virtuales corruptos, mientras que las secuencias de comandos de actualización podrían olvidarse de dar cuenta de una cierta configuración o archivo específico a la izquierda en el disco.

|  |  |
| --- | --- |
| 36 | Capítulo 5 |
|  | Arquitecturas de aplicaciones Web comunes |

Mientras que las aplicaciones monolíticas pueden beneficiarse de estibador, rompiendo la aplicación monolítica en subsistemas que pueden ser ampliados, desarrollado y desplegado de forma individual puede ser su punto de entrada en el reino de microservicios.



**Referencias - Arquitecturas Web Común**

**Creación de aplicaciones de N-Capas en C #**

<http://bit.ly/PS-NTier1>

**La arquitectura limpia**

<https://8thlight.com/blog/uncle-bob/2012/08/13/the-clean-architecture.html>

**La arquitectura de cebolla**

<http://jeffreypalermo.com/blog/the-onion-architecture-part-1/>

**El patrón Repositorio**

<http://deviq.com/repository-pattern/>

**Arquitectura muestra de solución limpia**

<https://github.com/ardalis/cleanarchitecture>

**Architecting microservicios libro electrónico**

<http://aka.ms/MicroservicesEbook>

|  |  |
| --- | --- |
| 37 | Capítulo 5 |
|  | Arquitecturas de aplicaciones Web comunes |

SECCIÓN 6

Tecnologías Web del lado del cliente común

*“Los sitios web deben verse bien desde el interior y hacia fuera.”*

Paul Cookson

Resumen

aplicaciones ASP.NET centrales son las aplicaciones web y por lo general se basan en tecnologías web del lado del cliente como HTML, CSS y JavaScript. Al separar el contenido de la página (HTML) desde su diseño y estilo (CSS), y su comportamiento (a través de JavaScript), aplicaciones web complejas pueden aprovechar la separación de las preocupaciones principales. Los cambios futuros en la estructura, diseño, o el comportamiento de la aplicación se pueden hacer más fácilmente cuando estas preocupaciones no están entrelazados.

Mientras HTML y CSS son relativamente estables, JavaScript, por medio de los marcos de aplicaciones y utilidades desarrolladores trabajan con la construcción de aplicaciones basadas en web, está evolucionando a una velocidad vertiginosa. Este capítulo se centra en algunas de las formas JavaScript se utiliza por los desarrolladores web como parte de las aplicaciones en desarrollo, ya que proporciona una visión general de alto nivel de las bibliotecas del lado del cliente y reaccionar angular.

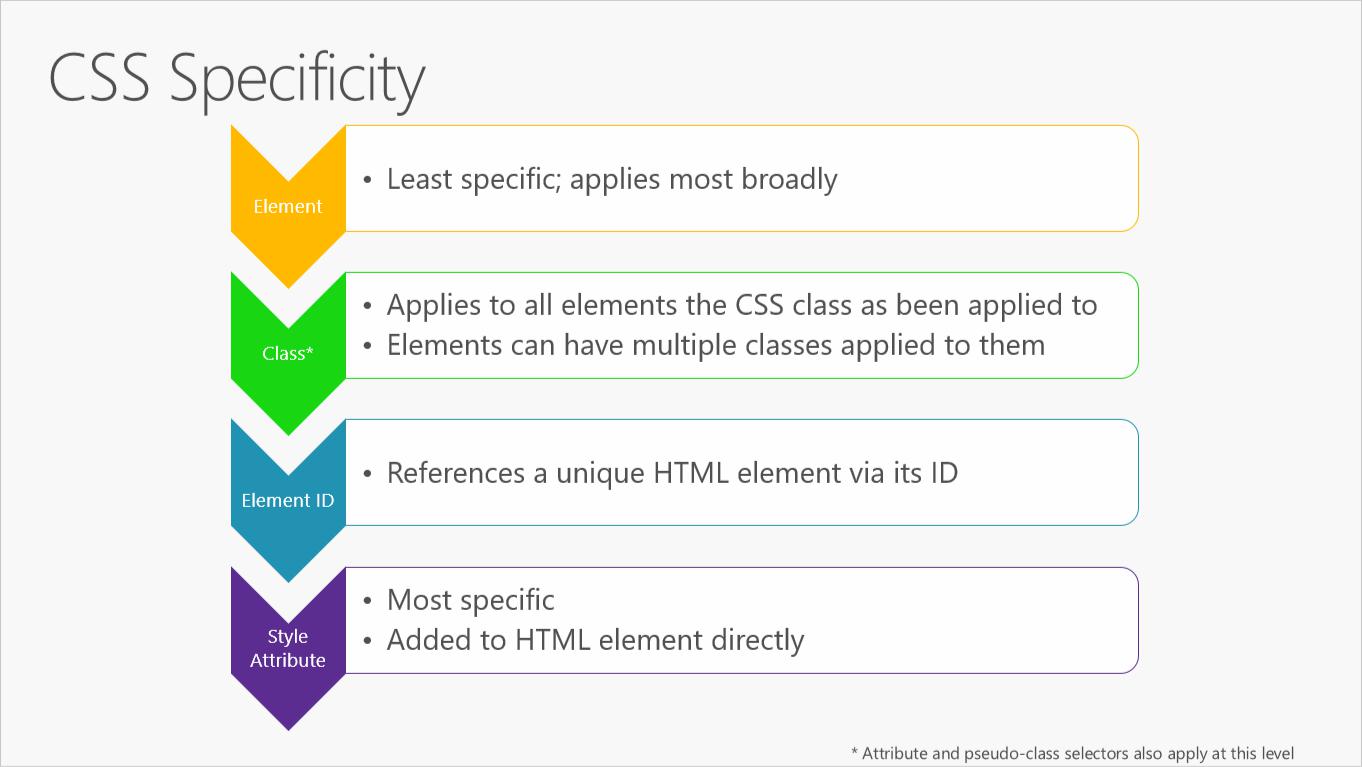
HTML

HTML (HyperText Markup Language) es el lenguaje de marcado estándar utilizado para crear páginas web y aplicaciones web. Sus elementos forman los bloques de construcción de páginas, lo que representa el texto con formato, imágenes, entradas de formulario, y otras estructuras. Cuando un navegador hace una petición a una URL, ya sea ir a buscar una página o una aplicación, lo primero que se devuelve es un documento HTML. Este documento HTML puede hacer referencia o incluir información adicional acerca de su aspecto y el diseño en forma de CSS, o el comportamiento en forma de JavaScript.

|  |  |
| --- | --- |
| 38 | Capítulo 6 |
|  | Tecnologías Web del lado del cliente común |

CSS

CSS (Cascading Style Sheets) se utiliza para controlar el aspecto y el diseño de los elementos HTML. estilos CSS se pueden aplicar directamente a un elemento HTML, definido por separado en la misma página, o se definen en un archivo separado y referenciados por la página. Estilos en cascada basan en la forma en que se utilizan para seleccionar un elemento HTML dado. Por ejemplo, un estilo podría aplicarse a un documento completo, pero sería anulado por un estilo que se aplica a un elemento en particular. Del mismo modo, un estilo específico del elemento sería anulado por un estilo que se aplica a una clase CSS que se aplica al elemento, que a su vez ser anulado por un estilo de orientación una instancia específica de ese elemento (a través de su ID). La Figura 7-1.



***La Figura 7-1.*** *reglas CSS especificidad, en orden.*

Lo mejor es mantener estilos en sus propios archivos de estilo separadas, y para usar en cascada a base de selección para aplicar estilos consistentes y reutilizables dentro de la aplicación. La colocación de las reglas de estilo dentro de HTML deben ser evitados y aplicar estilos a los elementos individuales específicos (en lugar de clases enteras de elementos, o elementos que han tenido una clase CSS particular, se les aplica) debe ser la excepción y no la regla.

**CSS Preprocesadores**

hojas de estilo CSS carecen de apoyo a la lógica condicional, variables y otras características del lenguaje de programación. Por lo tanto, las grandes hojas de estilo a menudo incluyen una gran cantidad de repetición, como el mismo color, tipo de letra, u otra configuración se aplica a muchas variaciones diferentes de los elementos HTML y CSS clases. preprocesadores CSS pueden ayudar a sus hojas de estilo siguen el[principio DRY](http://deviq.com/don-t-repeat-yourself/) mediante la adición de apoyo para las variables y la lógica.

Los preprocesadores CSS más populares son Sass y menos. Ambos se extienden CSS y son compatibles con ella, lo que significa que un archivo CSS es una llanura Sass válida o archivo MENOS. Sass se basa Rubí-y menos se basa JavaScript, y ambos funcionan típicamente como parte de su proceso de desarrollo local. Ambos tienen

|  |  |
| --- | --- |
| 39 | Capítulo 6 |
|  | Tecnologías Web del lado del cliente común |

herramientas de línea de comandos disponibles, así como soporte integrado en Visual Studio para ejecutar usando Gulp o tareas ronco.

JavaScript

JavaScript es un lenguaje de programación dinámico, interpretado que ha sido estandarizado en la especificación del lenguaje ECMAScript. Es el lenguaje de programación de la web. Como CSS, JavaScript puede ser definido como atributos dentro de los elementos HTML, como bloques de secuencia de comandos dentro de una página, o en archivos separados. Al igual CSS, generalmente se recomienda para organizar JavaScript en archivos separados, manteniéndolo separado tanto como sea posible del HTML encontrado en las páginas web individuales o puntos de vista de aplicación.

Cuando se trabaja con JavaScript en su aplicación web, hay algunas tareas que comúnmente se le debe realizar:

* Selección de un elemento HTML y recuperar y / o actualización de su valor
* Consultar una Web API para los datos
* El envío de un comando a un API de Web (y respondiendo a una llamada de retorno con su resultado)
* la validación de realizar

Puede realizar estas tareas con JavaScript solo, sino que existen muchas bibliotecas para hacer estas tareas más fáciles. Uno de los primeros y más exitosa de estas bibliotecas es jQuery, que sigue siendo una opción popular para simplificar estas tareas en las páginas web. Para aplicaciones de una sola página (ZEPA), jQuery no proporciona muchas de las características deseadas que Angular y reaccionar oferta.

**Aplicaciones Web legado con jQuery**

Aunque antigua para los estándares marco de JavaScript, jQuery sigue siendo una biblioteca muy comúnmente utilizado para trabajar con aplicaciones HTML / CSS y AJAX de construcción que hacen llamadas a las API web. Sin embargo, jQuery opera en el nivel del modelo de objetos de documento navegador (DOM), y por defecto sólo ofrece un modelo imprescindible, más que declarativo.

Por ejemplo, imaginar que si el valor de un cuadro de texto excede 10, un elemento de la página debe hacerse visible. En jQuery, esto normalmente se llevaría a cabo escribiendo un controlador de eventos con código a inspeccionar el valor del cuadro de texto y establecer la visibilidad del elemento de destino en función de ese valor. Este es un enfoque fundamental, basada en código. Otro marco podría utilizar en su lugar el enlace de datos para enlazar la visibilidad del elemento con el valor del cuadro de texto de forma declarativa. Esto no sería necesario escribir ningún código, sino que sólo requiere la decoración de los elementos que intervienen con los datos de atributos vinculante. Como comportamientos del lado del cliente se vuelven más complejos, el enlace de datos enfoques con frecuencia dan lugar a soluciones más simples con menos código y complejidad condicional.

**jQuery vs un Marco SPA**

|  |  |  |  |
| --- | --- | --- | --- |
| **Factor** | **jQuery** |  | **Angular** |
|  |  |  |  |
| Abstrae el DOM | **Sí** |  | **Sí** |
|  |  |  |  |
| Soporte AJAX | **Sí** |  | **Sí** |
|  |  |  |  |
| Enlace de datos declarativa | **No** |  | **Sí** |
|  |  |  |  |
| 40 |  |  | Capítulo 6 |
|  |  | Tecnologías Web del lado del cliente común | |

|  |  |  |
| --- | --- | --- |
| Enrutamiento de estilo MVC | **No** | **Sí** |
|  |  |  |
| de plantillas | **No** | **Sí** |
|  |  |  |
| Profundo vínculo de enrutamiento | **No** | **Sí** |
|  |  |  |

La mayor parte de las características jQuery carece intrínsecamente se puede añadir con la adición de otras bibliotecas. Sin embargo, un marco de spa como angular proporciona estas características de una manera más integrada, ya que ha sido diseñado con todos ellos en mente desde el principio. Además, jQuery es una biblioteca muy imprescindible, lo que significa que es necesario llamar a las funciones de jQuery para hacer cualquier cosa con jQuery. Gran parte del trabajo y la funcionalidad que proporcionan marcos de SPA se puede hacer de forma declarativa, que no requiere código real a escribirse.

El enlace de datos es un gran ejemplo de esto. En jQuery, por lo general sólo se necesita una línea de código para obtener el valor de un elemento DOM, o para ajustar el valor de un elemento. Sin embargo, se debe escribir el código cada vez que necesite cambiar el valor del elemento, ya veces esto va a ocurrir en múltiples funciones en una página. Otro ejemplo común es la visibilidad del elemento. En jQuery, puede haber muchos lugares diferentes donde se escribiría código para controlar si ciertos elementos eran visibles. En cada uno de estos casos, cuando se utiliza el enlace de datos, tendría que ser escrito ningún código. Sólo tendría que unir el valor o la visibilidad del elemento (s) en cuestión a un modelo de vista en la página, y los cambios en viewmodel que automáticamente se reflejaría en los elementos unidos.

**SPAs angulares**

AngularJS (1.x) se convirtió rápidamente en uno de los marcos de JavaScript más populares del mundo. Con angular, el equipo reconstruyó el marco desde la base (utilizando[Mecanografiado)](https://www.typescriptlang.org/) y renombrado de AngularJS simplemente angular. Ahora versiones actualizar más rápidamente, angular sigue siendo un marco sólido para la construcción de aplicaciones de una sola página.

aplicaciones angulares se construyen a partir de componentes. Componentes se combinan plantillas HTML con objetos especiales y controlan una parte de la página. Un componente simple a partir de docs angular se muestra aquí:

importación {Componente} de '@ angular / núcleo'; @Componente({

Selector: 'mi aplicación',

plantilla: `<h1> Hello {{nombre}} </ h1>`

})

clase de exportación AppComponent {name = 'angular'; }

Los componentes se definen utilizando la función decorador @Component, que toma en metadatos sobre el componente. La propiedad de selección identifica el identificador del elemento de la página en la que se visualiza este componente. La propiedad plantilla es una plantilla HTML simple que incluye un marcador de posición que corresponde a la propiedad el nombre del componente, que se define en la última línea.

Al trabajar con componentes y plantillas, en lugar de los elementos DOM, aplicaciones angular pueden funcionar a un nivel más alto de abstracción y con menos código en general que las aplicaciones escritas usando sólo JavaScript (también llamado “vainilla JS”) o con jQuery. Angular también impone un cierto orden en la forma de organizar los archivos de script del lado del cliente. Por convención, las aplicaciones angulares utilizan una estructura de carpetas comunes, con módulo y

|  |  |
| --- | --- |
| 41 | Capítulo 6 |
|  | Tecnologías Web del lado del cliente común |

archivos de comandos de componentes ubicados en una carpeta de aplicación. guiones angulares que se ocupan de la construcción, desplegar y probar la aplicación se encuentran normalmente en una carpeta de nivel superior.

Angular también hace un gran uso de la interfaz de línea de comandos de herramientas (CLI). Iniciación al desarrollo angular localmente (suponiendo que ya tiene instalado Git y NPM) consiste simplemente en la clonación de un acuerdo de recompra de GitHub y ejecutando `NPM instalar`Y`inicio NPM`. Más allá de esto, los barcos angular de su propia herramienta de CLI que puede crear proyectos, añadir archivos, y ayudar con la prueba, la agrupación, y las tareas de implementación. Esta facilidad de herramientas CLI hace angular especialmente compatible con ASP.NET Core, que también cuenta con un gran apoyo de la CLI.

Microsoft ha desarrollado una aplicación de referencia, [eShopOnContainers,](http://aka.ms/MicroservicesArchitecture) que incluye una implementación SPA angular. Esta aplicación incluye módulos angular para gestionar artículos de la cesta de compras, carga y visualización de la tienda en línea de su catálogo, y gastos de creación de la orden. Usted puede ver y descargar la aplicación de ejemplo de[GitHub.](https://github.com/dotnet-architecture/eShopOnContainers/tree/master/src/Web/WebSPA)

**Reaccionar**

A diferencia angular, que ofrece una implementación patrón Modelo-Vista-Controlador completa, reaccionan sólo se ocupa de vistas. No es un marco, sólo una biblioteca, por lo que la construcción de un SPA que necesita para aprovechar las bibliotecas adicionales.

Uno de los Reaccionar características más importantes es el uso de un DOM virtual. El DOM virtual proporciona reaccionar con varias ventajas, incluyendo el rendimiento (el DOM virtual puede optimizar qué partes del DOM real necesitan ser actualizados) y de la capacidad de prueba (sin necesidad de disponer de un navegador para probar reaccionar y sus interacciones con el DOM virtual).

Reaccionar también es inusual en la forma en que trabaja con HTML. En lugar de tener una separación estricta entre el código y marcado (con referencias a JavaScript que aparecen en atributos HTML tal vez), Reaccionar añade HTML directamente dentro de su código JavaScript como JSX. JSX es la sintaxis HTML similar que puede compilar a puro JavaScript. Por ejemplo:

<Ul>

{Authors.map (autor =>

<Li clave = {author.id}> {author.name} </ li>

)}

</ Ul>

Si ya conoce JavaScript, aprendiendo Reaccionar debe ser fácil. No hay curva de aprendizaje casi tanto o sintaxis especial involucrados como con las bibliotecas populares angular u otros.

Debido Reaccionar no es un marco completo, por lo general va desea que otras bibliotecas para manejar cosas como enrutamiento, llamadas a la API web, y gestión de la dependencia. Lo bueno es que, usted puede escoger el mejor biblioteca para cada uno de estos, pero la desventaja es que se necesita para hacer todas estas decisiones y verificar todas sus bibliotecas elegidos trabajan bien juntos cuando haya terminado. Si quieres un buen punto de partida, puede utilizar un kit de iniciación como Reaccionar tirachinas, que los envases preparados un conjunto de bibliotecas compatibles junto con React.

**La elección de un Marco SPA**

Al considerar qué marco de JavaScript funcionará mejor para apoyar su SPA, tenga en cuenta las siguientes consideraciones:

|  |  |
| --- | --- |
| 42 | Capítulo 6 |
|  | Tecnologías Web del lado del cliente común |

* Es el equipo familiarizado con el marco y sus dependencias (incluyendo mecanografiado en algunos casos)?
* Cómo obstinado es el marco, y está de acuerdo con su manera de hacer las cosas de forma predeterminada?
* Qué (o una biblioteca compañera) incluye todas las características de su aplicación requiere?
* ¿Está bien documentado?
* ¿Qué tan activo es su comunidad? Están nuevo edificio construido proyectos con ella?
* ¿Qué tan activo es su equipo central? Son cuestiones que se resuelven y nuevas versiones envían regularmente?

frameworks de JavaScript continúan evolucionando a una velocidad vertiginosa. Utilice las consideraciones mencionadas anteriormente para ayudar a mitigar el riesgo de elegir un marco que va a lamentar más tarde siendo dependiente. Si usted es particularmente adverso al riesgo, considere un marco que ofrece soporte comercial y / o está siendo desarrollado por una gran empresa.



**Referencias - Tecnologías Web Client**

**HTML y CSS**

<https://www.w3.org/standards/webdesign/htmlcss>

**Sass frente a menos**

https://www.keycdn.com/blog/sass-vs-less/

**Styling ASP.NET Aplicaciones Core con menos, Sass, y Font impresionante**

<https://docs.microsoft.com/en-us/aspnet/core/client-side/less-sass-fa>

**Desarrollo del lado del cliente en ASP.NET Core**

<https://docs.microsoft.com/en-us/aspnet/core/client-side/>

**jQuery**

https://jquery.com/

**jQuery vs AngularJS**

https://www.airpair.com/angularjs/posts/jquery-angularjs-comparison-migration-walkthrough

**Angular**

https://angular.io/

**Reaccionar**

<https://facebook.github.io/react/>

**reaccionar Slingshot**

<https://github.com/coryhouse/react-slingshot>

**Reaccionar vs Angular 2 Comparación**

<https://www.codementor.io/codementorteam/react-vs-angular-2-comparison-beginners-guide-lvz5710ha>

**5 Mejor JavaScript Frameworks de 2017**

https://hackernoon.com/5-best-javascript-frameworks-in-2017-7a63b3870282

|  |  |
| --- | --- |
| 43 | Capítulo 6 |
|  | Tecnologías Web del lado del cliente común |

SECCIÓN 7

El desarrollo de ASP.NET Core Aplicaciones

*“No es importante hacer las cosas bien la primera vez. Es de vital importancia para hacerlo bien la*

*ultima vez."*

Andrew Hunt y David Thomas

Resumen

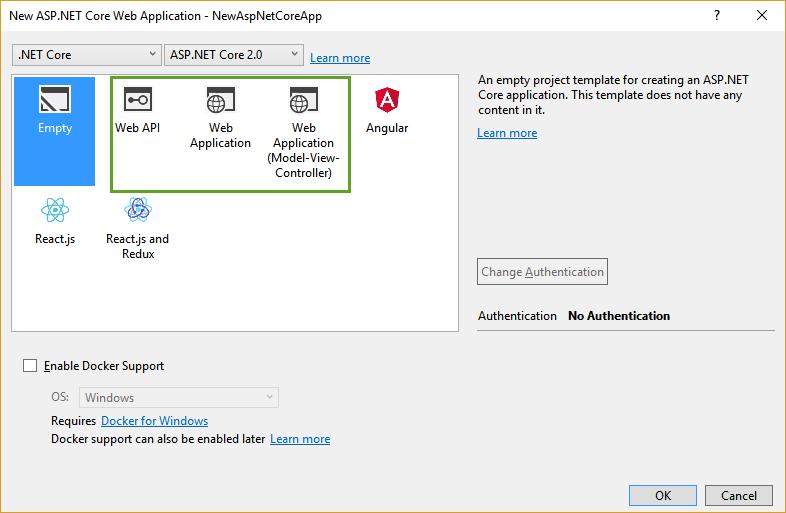
ASP.NET Core es una multi-plataforma, marco de código abierto para la construcción de modernas aplicaciones web optimizado en la nube. aplicaciones ASP.NET centrales son de peso ligero y modular, con soporte incorporado para la inyección de dependencia, lo que permite una mayor capacidad de prueba y facilidad de mantenimiento. En combinación con el marco ASP.NET MVC Core, que apoya la construcción de las API web modernos, además de ambas basadas en vistas y aplicaciones basados ​​en páginas, ASP.NET Core es un marco de gran alcance con el que construir aplicaciones web empresariales.

MVC y Páginas de la maquinilla de afeitar

ASP.NET MVC Core ofrece muchas características que son útiles para la construcción de API basadas en web y aplicaciones. El término MVC significa “Model-View-Controller”, un patrón de interfaz de usuario que rompe las responsabilidades de responder a las peticiones de los usuarios en varias partes. Además de seguir este patrón, también se puede implementar características en sus aplicaciones ASP.NET Core como Razor páginas. Páginas de afeitar están integrados en ASP.NET MVC Core, y utilizar las mismas funciones de enrutamiento, la unión modelo, etc. enrutamiento Sin embargo, en lugar de tener carpetas separadas y los archivos de controladores, vistas, etc., y el uso de atributos basados ​​en la maquinilla de afeitar páginas se colocado en una sola carpeta ( “/ Páginas”), ruta en función de su ubicación relativa en esta carpeta, y manejar las solicitudes con los programas pilotos en lugar de acciones del controlador.

Cuando se crea un nuevo núcleo de la aplicación ASP.NET, debe tener un plan en mente para el tipo de aplicación que desea construir. En Visual Studio, que se elija entre varias plantillas como las que se muestran en la Figura 7-1.

|  |  |
| --- | --- |
| 44 | Capítulo 7 |
|  | El desarrollo de ASP.NET MVC Core Aplicaciones |



*La Figura 7-1. Plantillas de proyecto ASP.NET en Visual Studio Core*

Las tres plantillas de proyectos más comunes son API web de aplicaciones Web y de aplicaciones Web (Modelo-Vista-Controlador). A pesar de que sólo se puede tomar esta decisión cuando se crea un proyecto, no es una decisión irrevocable. El proyecto de API Web utiliza controladores Modelo-Vista-Controlador estándar - sólo le falta Vistas por defecto. Del mismo modo, la plantilla de aplicación web por defecto usa la maquinilla de afeitar de páginas, y así también carece de una carpeta Vistas. Puede añadir una carpeta Vistas a estos proyectos adelante para apoyar el comportamiento basado en la vista. Ni el API Web ni el proyecto de Modelo-Vista-Controlador incluyen una carpeta de páginas por defecto, pero se puede añadir una tarde para apoyar el comportamiento basado en Páginas de la maquinilla de afeitar. Se puede pensar en estas tres plantillas como el apoyo a tres tipos diferentes de interacción con el usuario por defecto: datos (API web), la página-Based y basadas en vistas. Sin embargo,

**¿Por Razor páginas?**

Como se vio en la Figura 7-1, la maquinilla de afeitar de páginas es ahora el método por defecto para las nuevas aplicaciones Web en Visual Studio. Afeitar páginas ofrecen un medio significativamente más simples de construir funciones de la aplicación basada en la página, como las formas no-SPA. El uso de controladores y vistas, era común para que las aplicaciones tienen muy grandes controladores que trabajaron con muchas dependencias y diferentes modelos de vista y devueltos muchos puntos de vista diferentes. Esto dio lugar a una gran complejidad y, a menudo resultó en controladores que no siguen el principio de responsabilidad individual o Principios abierto / cerrado de manera efectiva. Páginas de afeitar aborda esta cuestión mediante la encapsulación de la lógica del lado del servidor para una “página” lógico dado en una aplicación web con su margen de beneficio de afeitar. Una maquinilla de afeitar de página que no tiene ninguna lógica del lado del servidor simplemente puede consistir en un archivo de la maquinilla de afeitar (por ejemplo, “Index.cshtml”). Sin embargo,

|  |  |
| --- | --- |
| 45 | Capítulo 7 |
|  | El desarrollo de ASP.NET MVC Core Aplicaciones |

Un modelo de página de la maquinilla de afeitar Página combina las responsabilidades de un controlador MVC y un modelo de vista. En vez de manejar peticiones con métodos acción del controlador, controladores página modelo como “OnGet ()” se ejecutan, lo que hace su página asociada de forma predeterminada. páginas de afeitar simplifican el proceso de creación de páginas individuales en una aplicación ASP.NET Core, sin dejar de ofrecer todas las características arquitectónicas de ASP.NET MVC Core. Son una buena opción por defecto para una nueva funcionalidad basada en la página.

**Cuándo utilizar MVC**

Si usted está construyendo las API web, el patrón MVC tiene más sentido que tratar de utilizar la maquinilla de afeitar páginas. Si su proyecto sólo expondrá los puntos finales de API web, lo ideal sería comenzar a partir de la plantilla de proyecto Web API, pero por lo demás es fácil de añadir controladores y puntos finales de API asociadas a cualquier aplicación ASP.NET Core. También debe utilizar el enfoque MVC basado en la vista si está migrando una aplicación existente de ASP.NET MVC 5 o anterior a ASP.NET MVC Core y desea hacerlo con la menor cantidad de esfuerzo. Una vez que haya tomado la migración inicial, puede evaluar si tiene sentido adoptar la maquinilla de afeitar Páginas para las nuevas características o incluso como una migración a gran escala.

Ya sea que elija para construir su aplicación web utilizando la maquinilla de afeitar o Páginas vistas de MVC, su aplicación tendrá un rendimiento similar, e incluirá soporte para la inyección de dependencias, filtros, enlace de modelos y validación, etc.

Las solicitudes de mapeo a las respuestas

En el fondo, ASP.NET aplicaciones Core mapa respuestas a las peticiones entrantes salientes. En un nivel bajo, esto se hace con el middleware y aplicaciones ASP.NET simples y básicas microservicios puede estar compuesta exclusivamente de middleware personalizado. Al utilizar ASP.NET MVC Core, se puede trabajar a un nivel algo más alto, pensar en términos de páginas y manipuladores o rutas, controladores y acciones. Cada solicitud entrante se compara con la tabla de enrutamiento de la aplicación, y si se encuentra una ruta coincidente, el controlador asociado página (perteneciente a una maquinilla de afeitar de página) o método de acción (que pertenece a un controlador) se llama para manejar la petición. Si no se encuentra ninguna ruta coincidente, un controlador de errores (en este caso, devolviendo un resultado NotFound) se llama.

Afeitar página enlaza a un sistema de enrutamiento basado en la convención (basado en la ubicación del archivo dentro de la carpeta de páginas) y métodos de controlador-verbo HTTP específica. Por ejemplo, una página llamada Index.cshtml situada en la raíz de la carpeta Páginas se dirigirá a la raíz de la aplicación web ( “/”) o su nombre ( “/ Índice”). Una página llamada “Customers.cshtml” encaminaría a “/ Clientes”. Puede especificar rutas personalizadas para Razor Páginas al configurar MVC en el inicio, si se desea.

aplicaciones ASP.NET Core MVC pueden utilizar rutas convencionales, rutas de atributo, o ambos. rutas convencionales están definidas en el Código, especificando las convenciones de enrutamiento utilizando sintaxis como en el ejemplo siguiente:

app.UseMvc (rutas =>

{

routes.MapRoute ( "default", "controlador = {Inicio} / {action = Índice} / {id}?");

});

En este ejemplo, una ruta denominada “default” se ha añadido a la tabla de enrutamiento. Define una plantilla de ruta con marcadores de posición para el regulador, la acción y de identificación. Los marcadores de posición controlador y la acción tienen por defecto especificado ( “Inicio” y “Índice”, respectivamente), y el marcador de posición ID es opcional (en virtud de una “?” Que se le aplica). La convención define aquí afirma que la primera parte de una solicitud debe corresponder

|  |  |
| --- | --- |
| 46 | Capítulo 7 |
|  | El desarrollo de ASP.NET MVC Core Aplicaciones |

al nombre del controlador, la segunda parte de la acción, y luego si es necesario tercera parte representará un parámetro id. rutas convencionales se definen típicamente en un lugar para la aplicación, como en el método Configurar en la clase de arranque.

rutas de atributos se aplican a los controladores y acciones directamente, en lugar de especificar a nivel mundial. Esto tiene la ventaja de hacerlos mucho más visible cuando se está buscando a un método en particular, pero sí significa que la información de enrutamiento no se mantiene en un solo lugar en la aplicación. Con rutas de atributos, puede especificar fácilmente múltiples rutas para una acción determinada, así como combinar rutas entre controladores y acciones. Por ejemplo:

[Ruta ( "Inicio")]

HomeController clase pública: Controlador

{

[Ruta ( "")] // Combina para definir la plantilla ruta "Inicio" [Ruta ( "Índice")] // combina para definir la plantilla ruta "Inicio / Índice" [Ruta ( "/")] // NO combinan, define la plantilla ruta "" Índice pública IActionResult () {}

plantillas de ruta pueden ser especificadas en [HttpGet] y atributos similares, evitando la necesidad de añadir por separado [Ruta]atributos. Attribute rutas también pueden utilizar tokens para reducir la necesidad de repetir los nombres del controlador o de acción, como se muestra a continuación:

[Ruta ( "[controlador]")]

ProductsController clase pública: Controlador

{

[Ruta ( "")] // 'Productos' Partidos [Ruta ( "Índice")] 'Productos / Índice' // Partidos Índice IActionResult pública ()

}

Páginas de afeitar no utilizan enrutamiento atributo. Puede especificar información adicional para una plantilla de ruta

Página de afeitar como parte de su @página directiva:

@ página “{id: int}”

En el ejemplo anterior, la página en cuestión se correspondería con una ruta con un número entero carné de identidadparámetro. Por ejemplo:

“/ Productos / 123”

Una vez que una petición dada ha sido igualado a una ruta, pero antes de que se llama el método de acción (o controlador), ASP.NET Core MVC realizará [enlace de modelos](https://docs.microsoft.com/en-us/aspnet/core/mvc/models/model-binding) y [Modelo de validación](https://docs.microsoft.com/en-us/aspnet/core/mvc/models/validation) sobre la solicitud. es responsable de la conversión de datos HTTP entrantes en los tipos de .NET especificados como parámetros del método de acción a ser llamados enlace de modelos. Por ejemplo, si el método de acción espera unaint idparámetro, la unión modelo intentará proporcionar este parámetro a partir de un valor proporcionado como parte de la solicitud. Para ello, se ve vinculante para los valores en una forma publicado modelo, los valores en la ruta en sí, y los valores de cadena de consulta. Suponiendo se encuentra un valor de ID, se convierte en un número entero antes de ser pasado al método de acción.

|  |  |
| --- | --- |
| 47 | Capítulo 7 |
|  | El desarrollo de ASP.NET MVC Core Aplicaciones |

Después de la unión del modelo, pero antes de llamar al método de acción, se produce la validación del modelo. La validación del modelo utiliza atributos opcionales en el tipo de modelo, y puede ayudar a asegurar que el objeto del modelo proporcionada se ajusta a ciertos requisitos de datos. Ciertos valores pueden ser especificados como se requiere, o limitadas a una cierta longitud o rango numérico, etc. Si se especifican atributos de validación pero el modelo no se ajusta a sus necesidades, la propiedadModelState.IsValid será falsa, y el conjunto de reglas de validación en su defecto estará disponible para enviar al cliente que realiza la solicitud.

Si está utilizando la validación del modelo, usted debe estar seguro de comprobar siempre que el modelo es válido antes de hacer cualquier comando que alteran el estado, para asegurar que su aplicación no está corrompida por los datos no válidos. Puede utilizar una[filtrar](https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/filters) para evitar la necesidad de añadir código para esto en cada acción. ASP.NET MVC filtros Core ofrecen una forma de grupos de solicitudes que intercepta, por lo que las políticas comunes y preocupaciones transversales se pueden aplicar de forma selectiva. Los filtros se pueden aplicar a las acciones individuales, controladores enteros, o globalmente para una aplicación.

Para las API web, ASP.NET MVC soporta Core [*negociación de contenido*,](https://docs.microsoft.com/en-us/aspnet/core/mvc/models/formatting) permitiendo que las solicitudes para especificar cómo deben ser formateados respuestas. Sobre la base de los encabezados proporcionados en la solicitud, las acciones que regresan datos formatear la respuesta en XML, JSON, u otro formato compatible. Esta característica permite que la misma API para ser utilizado por varios clientes con diferentes requisitos de formato de datos.



**Referencias - Las solicitudes de asignación a las respuestas**

**Enrutamiento a acciones del controlador**

<https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/routing>

**modelo Encuadernación**

https://docs.microsoft.com/en-us/aspnet/core/mvc/models/model-binding

**Modelo de validación**

<https://docs.microsoft.com/en-us/aspnet/core/mvc/models/validation>

**filtros**

https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/filters

Trabajar con dependencias

ASP.NET Core ha incorporado soporte para e internamente hace uso de una técnica conocida como [dependencia](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection) [inyección.](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection) Dependencia inyección es una técnica que activar la articulación flexible entre diferentes partes de Una aplicación. acoplamiento más flojo es deseable ya que hace que sea más fácil para aislar partes de la aplicación, lo que permite para las pruebas o reemplazo. También hace que sea menos probable que un cambio en una parte de la aplicación tendrá un impacto inesperado en otra parte de la aplicación. La inyección de dependencia se basa en la[principio de inversión de la dependencia,](http://deviq.com/dependency-inversion-principle/) y es clave para lograr la frecuencia [principio abierto / cerrado.](http://deviq.com/open-closed-principle/) Cuando la evaluación de cómo funciona su aplicación con sus dependencias, tenga cuidado de el [estática](http://deviq.com/static-cling/) código de olor, y recordar el aforismo “[nuevo es pegamento.](http://ardalis.com/new-is-glue)”

electricidad estática se produce cuando las clases hacen llamadas a métodos estáticos, o propiedades estáticas de acceso, que tienen efectos secundarios, o dependencias de la infraestructura. Por ejemplo, si usted tiene un método que llama a un método estático, que a su vez escribe a una base de datos, el método está estrechamente unida a la base de datos. Cualquier cosa que rompe esa llamada base de datos de romper su método. Prueba de tales métodos es muy difícil, ya que tales pruebas o bien requieren bibliotecas burlones comerciales para burlarse de las llamadas estáticas, o sólo pueden ser probados con una base de datos de prueba en su sitio. llamadas estáticas que no tienen ninguna dependencia

|  |  |
| --- | --- |
| 48 | Capítulo 7 |
|  | El desarrollo de ASP.NET MVC Core Aplicaciones |

infraestructura, especialmente aquellos que son completamente sin estado, están muy bien para llamar y no tienen ningún impacto en el acoplamiento o la capacidad de prueba (más allá de código de acoplamiento a la llamada estática en sí).

Muchos desarrolladores comprendan los riesgos de la electricidad estática y el estado global pero todavía firmemente par su código para implementaciones específicas a través de instancias directa. “Nuevo es pegamento” está destinado a ser un recordatorio de este acoplamiento, y no una condena general de la utilización de lanuevopalabra clave. Al igual que con las llamadas a métodos estáticos, nuevas instancias de tipos que no tienen dependencias externas normalmente no lo hacen fuertemente código par de detalles de implementación o hacer la prueba más difícil. Pero cada vez que se crea una instancia de una clase, tome un momento para considerar si tiene sentido para codificar esa instancia específica en ese lugar en particular, o si sería un mejor diseño para solicitar esa instancia como una dependencia.

|  |  |
| --- | --- |
| 49 | Capítulo 7 |
|  | El desarrollo de ASP.NET MVC Core Aplicaciones |

**Declarar las dependencias**

ASP.NET núcleo se construye en torno a tener métodos y clases declaran sus dependencias, solicitando como argumentos. Las aplicaciones ASP.NET se establecen normalmente en una clase de inicio, que a su vez está configurado para soportar la inyección de dependencias en varios puntos. Si su clase tiene un constructor de inicio, se puede solicitar a través del constructor dependencias, así:

public class plantilla 1.x de inicio // ASP.NET Core {

De inicio pública (env IHostingEnvironment)

{

var constructor = new ConfigurationBuilder ()

.SetBasePath (env.ContentRootPath)

.AddJsonFile ( "appsettings.json", opcional: falso,

reloadOnChange: true)

.AddJsonFile ($ "appsettings {} env.EnvironmentName .json.", Opcional: true);

// código adicional omite

}

}

La clase de arranque es interesante ya que no hay requisitos de tipo explícitas para ello. No hereda de una clase base de inicio especial, ni implementar ninguna interfaz en particular. Se le puede dar un constructor, o no, y puede especificar todos los parámetros en el constructor como desee. Cuando el proveedor de alojamiento web que ha configurado para su aplicación se inicia, se llamará la clase de arranque le has dicho a utilizar, y utilizará la inyección de dependencias para rellenar cualquier dependencia de la clase de arranque requiere. Por supuesto, si usted solicita los parámetros que no están configuradas en el recipiente servicios utilizados por ASP.NET Core, obtendrá una excepción, pero si usted se pega a las dependencias del contenedor conoce, puede solicitar cualquier cosa que desee.

En ASP.NET Core 2,0, valores predeterminados de configuración se establecen típicamente en Program.cs y luego pasado en el

clase de arranque. Por lo tanto, la clase de inicio en las plantillas por defecto es el siguiente:

public class plantilla 2.x de inicio // ASP.NET Core {

Inicio público (configuración IConfiguration)

{

Configuración = configuración;

}

}

La inyección de dependencia es parte integral del núcleo ASP.NET aplicaciones desde el principio, cuando se crea la instancia de inicio. No se detiene allí por la clase de arranque. También se puede solicitar en las dependencias del método Configurar:

|  |  |
| --- | --- |
|  | public void Configurar (IApplicationBuilder aplicación, |
|  | env IHostingEnvironment, |
| 50 | Capítulo 7 |
|  | El desarrollo de ASP.NET MVC Core Aplicaciones |

ILoggerFactory loggerFactory)

{

// código omite

}

los ConfigureServicesmétodo es la excepción a este comportamiento; se debe tomar sólo un parámetro de tipoIServiceCollection. En realidad, no tenga que soportar la inyección de dependencia, ya que por un lado es responsable de añadir objetos al contenedor de servicios, y por el otro se tiene acceso a todos los servicios configurados actualmente a través de laIServiceCollectionparámetro. Por lo tanto, se puede trabajar con las dependencias definidas en la colección de servicios de ASP.NET Core en cada parte de la clase de arranque, ya sea mediante la solicitud del servicio que se necesita como parámetro o trabajando con elIServiceCollection en ConfigureServices.

**Nota:** Si es necesario asegurarse de determinados servicios están disponibles para su clase de arranque, puede configurarlas usando WebHostBuilder y su método ConfigureServices.

La clase de inicio es un modelo de cómo se debe estructurar otras partes de su aplicación ASP.NET Core, desde controladores de middleware para Filtros de sus propios servicios. En cada caso, se debe seguir la[Explícita Dependencias Principio,](http://deviq.com/explicit-dependencies-principle/) solicitando sus dependencias en lugar de crear directamente, y el aprovechamiento de la inyección de dependencia a través de su aplicación. Tenga cuidado de dónde y cómo se crea directamente implementaciones, especialmente los servicios y objetos que trabajar con la infraestructura o de efectos secundarios. Prefiero trabajar con abstracciones definidas en el núcleo de la aplicación y pasa como argumentos para codificar las referencias a los tipos específicos de implementación.

La estructuración de la Aplicación

aplicaciones monolíticas suelen tener un único punto de entrada. En el caso de una web ASP.NET Core

aplicación, el punto de entrada será el proyecto web ASP.NET Core. Sin embargo, eso no significa que la solución debe consistir en un solo proyecto. Es útil para romper la aplicación en diferentes capas de seguir separación de preocupaciones. Una vez dividido en capas, es útil ir más allá de las carpetas a proyectos separados, que pueden ayudar a lograr una mejor encapsulación. El mejor enfoque para lograr estos objetivos con una aplicación ASP.NET Core es una variación de la arquitectura limpia discutido en el capítulo 5. Siguiendo este enfoque, la solución de la aplicación se compone de bibliotecas separadas para la interfaz de usuario, Infraestructura y ApplicationCore.

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

modelo y las interfaces de objetos de la aplicación se deben colocar en el proyecto ApplicationCore. Este proyecto tendrá el menor número de dependencias como sea posible, y los otros proyectos de la solución va a hacer referencia a ella. Las entidades empresariales que necesitan ser persistido se definen en el proyecto ApplicationCore, como son los servicios que no dependen directamente de la infraestructura.

Los detalles de implementación, por ejemplo, cómo se realiza la persistencia o cómo las notificaciones podrían ser enviados a un usuario, se mantienen en el proyecto de infraestructura. Este proyecto hará referencia a paquetes específicos de la implementación como Entidad Marco de base, pero no debería exponer detalles acerca de estas implementaciones fuera del proyecto. Los servicios de infraestructura y repositorios deben implementar las interfaces que se definen en el proyecto ApplicationCore, y sus implementaciones de persistencia son responsables de recuperar y almacenar entidades definidas en ApplicationCore.

|  |  |
| --- | --- |
| 51 | Capítulo 7 |
|  | El desarrollo de ASP.NET MVC Core Aplicaciones |

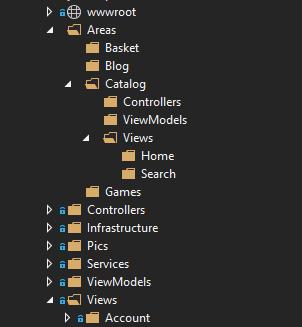
El proyecto ASP.NET Core sí es responsable de cualquier preocupación a nivel de interfaz de usuario, pero no debe incluir la lógica de negocio o los detalles de infraestructura. De hecho, lo ideal es que ni siquiera tienen una dependencia en el proyecto de infraestructura, lo que ayudará a asegurar que no haya dependencia entre los dos proyectos se introduce accidentalmente. Esto se puede lograr usando un contenedor DI de terceros, como StructureMap, lo que le permite definir reglas DI en las clases de registro en cada proyecto.

Otro enfoque para desacoplar la aplicación de los detalles de implementación es tener las llamadas microservicios aplicación, tal vez desplegados en recipientes individuales acoplables. Esto proporciona una mayor separación de las preocupaciones y desacoplamiento de aprovechamiento de DI entre dos proyectos, pero tiene una complejidad adicional.

**Organización función**

Por defecto, las aplicaciones basadas en vistas ASP.NET MVC Core organizan su estructura de carpetas para incluir controladores y vistas, y con frecuencia ViewModels (ver más abajo para la organización de la maquinilla de afeitar de páginas). código de cliente para apoyar estas estructuras del lado del servidor es normalmente almacenada por separado en la carpeta wwwroot. Sin embargo, las grandes aplicaciones pueden encontrar problemas con esta organización, ya que trabajar en cualquier característica dada a menudo requiere saltar entre estas carpetas. Esto se vuelve más y más difícil a medida que el número de archivos y subcarpetas de cada carpeta crece, lo que resulta en una gran cantidad de desplazamiento a través de Explorador de soluciones. Una solución a este problema consiste en organizar el código de aplicación de función en lugar de por tipo de archivo. Este estilo de organización se denomina típicamente como carpetas de características o rodajas de características (véase también:[Las rebanadas verticales)](http://bit.ly/2abpJ7t).

ASP.NET MVC Core soporta Áreas para este propósito. El uso de áreas, puede crear conjuntos separados de controladores y vistas carpetas (así como cualquiera de los modelos asociados) en cada carpeta de Área. Figura 7-2 muestra un ejemplo de estructura de carpetas, el uso de áreas.



*Figura 7-2 de la organización Muestra Área*

Al utilizar zonas, debe utilizar atributos para decorar sus controladores con el nombre de la zona a la que pertenecen:

|  |  |
| --- | --- |
| 52 | Capítulo 7 |
|  | El desarrollo de ASP.NET MVC Core Aplicaciones |

[Área ( “Catálogo”)]

HomeController clase pública

{}

También es necesario añadir soporte de área a tus rutas:

app.UseMvc (rutas =>

{

* Áreas apoyan routes.MapRoute (

nombre: "areaRoute", la plantilla:

"{Zona: existe} / {controller = Inicio} / {action = Índice} / {id}?");

routes.MapRoute (

nombre: "por defecto",

plantilla: "{controller = Inicio} / {action = Índice} / {? id}");

});

Además del apoyo integrado para las zonas, también puede utilizar su propia estructura de carpetas, y los convenios en lugar de atributos y rutas personalizadas. Esto permitirá tener carpetas de características que no incluían carpetas separadas para las vistas, controladores, etc., manteniendo el plano jerarquía y lo que es más fácil de ver todos los archivos relacionados en un solo lugar para cada función.

ASP.NET central utiliza una función de los tipos de convenciones para controlar su comportamiento. Puede modificar o sustituir estas convenciones. Por ejemplo, puede crear una convención que conseguirá automáticamente el nombre de la función de un controlador determinado basado en su espacio de nombres (que se correlaciona normalmente a la carpeta en la que se encuentra el controlador):

FeatureConvention: {IControllerModelConvention

public void Aplicar (controlador ControllerModel)

{

controller.Properties.Add ( "característica",

GetFeatureName (controller.ControllerType));

}

GetFeatureName cadena privada (TypeInfo controllerType)

{

string [] tokens = controllerType.FullName.Split ( ''); if (! tokens.Any (t => t == "Características")) return ""; cadena FeatureName = fichas

.SkipWhile (t =>! T.Equals ( "características", StringComparison.CurrentCultureIgnoreCase))

Saltee (1)

.Toma 1)

.FirstOrDefault ();

|  |  |
| --- | --- |
|  | volver FeatureName; |
| 53 | Capítulo 7 |
|  | El desarrollo de ASP.NET MVC Core Aplicaciones |

}

}

A continuación, especifica esta convención como una opción cuando se agrega soporte para MVC para su aplicación en

ConfigureServices:

services.AddMvc (o => o.Conventions.Add (nueva FeatureConvention ()));

ASP.NET MVC Core también utiliza una convención para localizar puntos de vista. Puede anular con una convención personalizado para que los puntos de vista estarán en sus carpetas de características (utilizando el nombre de la función proporcionada por el FeatureConvention, arriba). Usted puede aprender más acerca de este enfoque y descargar una muestra de trabajo del artículo de MSDN,[Característica de segmentos para ASP.NET MVC Core.](https://msdn.microsoft.com/en-us/magazine/mt763233.aspx)

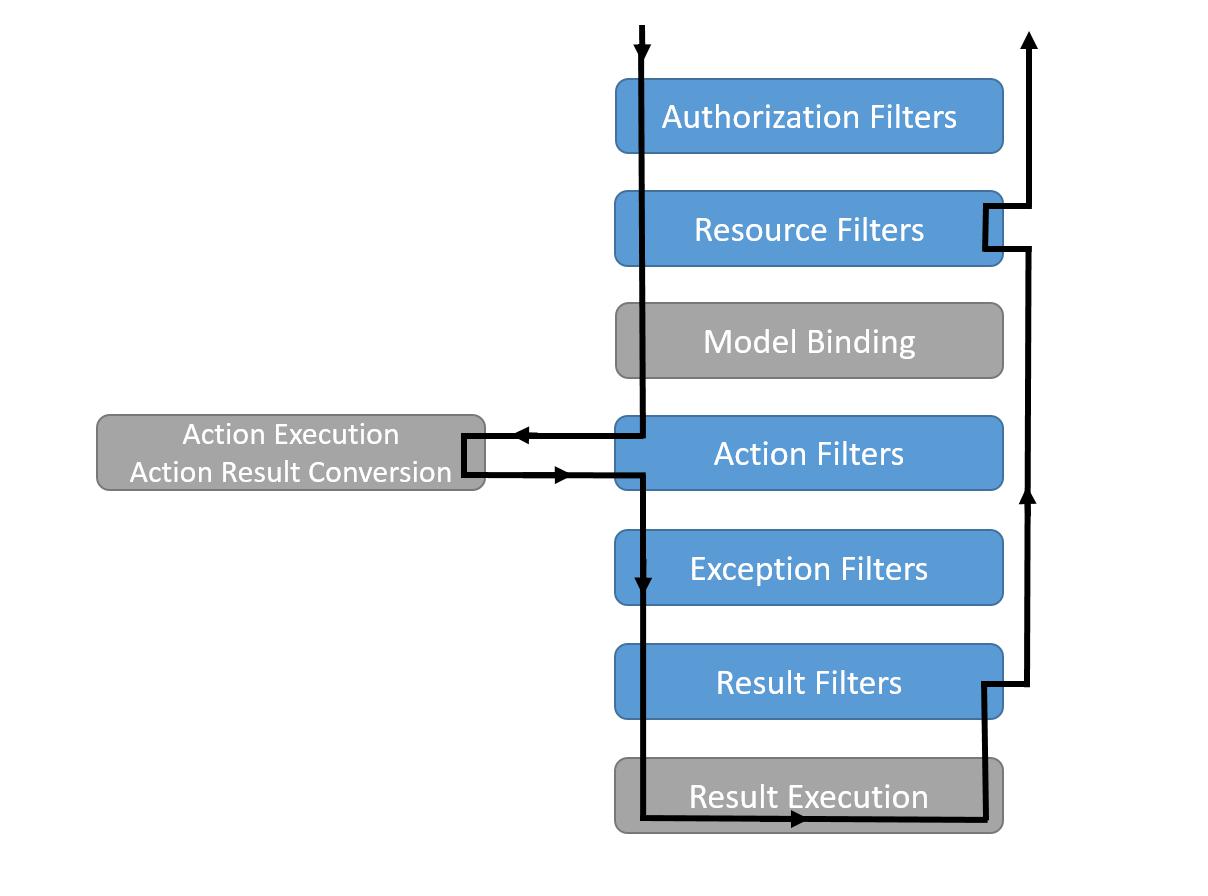
**Páginas de afeitar**

Afeitar páginas ofrecen muchos de los mismos beneficios de la organización como carpetas de características o áreas. Al utilizar la maquinilla de afeitar de páginas, ya no es necesario carpetas separadas y los archivos de controladores, vistas y ViewModels. Usted también no tendrá muchos puntos de vista diferentes para un controlador determinado. En su lugar, va a crear un archivo de una sola página para cada página lógica que el usuario interactúa. Cualquier lógica asociada con esta página se especificará en su página del modelo, junto con cualquier dato opinión de que la página requiere (a menudo como una clase anidada dentro del modelo de página). Páginas de afeitar y sus modelos de página se agrupan por Visual Studio, así que la navegación entre las marcas de afeitar y el código correspondiente y modelo de datos no requiere cambiar de una carpeta dentro de su proyecto. A medida que el número de páginas de la aplicación crece, se pueden organizar fácilmente dentro de la carpeta de páginas mediante el uso de subcarpetas que corresponden a la estructura de URL de su aplicación. Más información acerca de la construcción[Más simple ASP.NET MVC Core Aplicaciones con la maquinilla de afeitar Artículos en la revista de MSDN.](https://msdn.microsoft.com/en-us/magazine/mt842512.aspx)

**Problemas transversales**

Dado que las aplicaciones crecen, cada vez es más importante tener en cuenta las preocupaciones transversales para eliminar la duplicación y mantener la consistencia. Algunos ejemplos de las preocupaciones transversales en aplicaciones ASP.NET centrales son la autenticación, reglas de validación de modelo, el almacenamiento en caché de salida, y el control de errores, aunque hay muchos otros. ASP.NET MVC Core[filtros](https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/filters) permite llevar a cabo código antes o después de ciertos pasos en la canalización de procesamiento de la solicitud. Por ejemplo, un filtro se puede ejecutar la unión antes y después de modelo, antes y después de una acción, o antes y después el resultado de una acción. También puede utilizar un filtro de autorización para controlar el acceso al resto de la tubería. Las figuras 7-1 muestran cómo la ejecución petición pasa a través de filtros, si está configurado.

|  |  |
| --- | --- |
| 54 | Capítulo 7 |
|  | El desarrollo de ASP.NET MVC Core Aplicaciones |



*Figura 7-1 Solicitud de ejecución a través de filtros y canalización de solicitudes.*

Los filtros se implementan normalmente como atributos, para que ellos puedan aplicar los controladores o acciones. Cuando se añade de esta manera, los filtros especificados en la anulación de nivel de acción o se basan en filtros especificados a nivel del controlador, que a su vez invalidar los filtros globales. Por ejemplo, el atributo [Ruta] se puede utilizar para construir las rutas entre controladores y acciones. Del mismo modo, la autorización se puede configurar en el nivel de controlador, y luego anulado por acciones individuales, como el siguiente ejemplo se muestra:

[Autorizar]

AccountController clase pública: Controlador

{

[Permitir anónimos]

pública asíncrono de tareas <IActionResult> Inicio de sesión () {}

pública asíncrono de tareas <IActionResult> ForgotPassword () {}

}

El primer método, Login, utiliza el filtro AllowAnonymous (atributo) para anular el filtro Autorizar establecido en el nivel de controlador. La acción ForgotPassword (y cualquier otra acción en la clase que no tiene un atributo AllowAnonymous) requerirán una solicitud autenticada.

|  |  |
| --- | --- |
| 55 | Capítulo 7 |
|  | El desarrollo de ASP.NET MVC Core Aplicaciones |

Los filtros se pueden utilizar para eliminar la duplicación en forma de políticas de manejo de errores comunes para las API. Por ejemplo, una política típica API es devolver una respuesta NotFound a las solicitudes que hacen referencia a las teclas que no existen, y una respuesta BadRequest si falla la validación del modelo. El siguiente ejemplo demuestra estas dos políticas en acción:

[HttpPut ( "{id}")]

pública asíncrono de tareas <IActionResult> Ponga (int id, [FromBody] Autor autor)

{

if ((esperar \_authorRepository.ListAsync ()). Todo (a => a.Id! = id)) {

volver NotFound (id);

}

if (! ModelState.IsValid)

{

volver BadRequest (ModelState);

}

author.Id = id;

esperar \_authorRepository.UpdateAsync (autor); Ok volver ();

}

No permita que sus métodos de acción a llenarse con código condicional como este. En su lugar, tire de las políticas en los filtros que se pueden aplicar en una medida que sea necesario. En este ejemplo, la comprobación de validación de modelo, lo que debería ocurrir en cualquier momento se envía un comando a la API, puede ser reemplazado por el siguiente atributo:

public class ValidateModelAttribute: {ActionFilterAttribute

override public void OnActionExecuting (contexto ActionExecutingContext)

{

if (! context.ModelState.IsValid)

{

context.Result = new

BadRequestObjectResult (context.ModelState);

}

}

}

Puede añadir el ValidateModelAttribute a su proyecto como una dependencia Nuget mediante la inclusión de la [paquete Ardalis.ValidateModel.](https://www.nuget.org/packages/Ardalis.ValidateModel)

Asimismo, un filtro se puede utilizar para comprobar si existe un registro y devolver un 404 antes de que se ejecuta la acción, lo que elimina la necesidad de realizar estas comprobaciones en la acción. Una vez que ha tirado a cabo convenciones común y organizada su solución de código de infraestructura separada y la lógica de negocio de su interfaz de usuario, sus métodos de acción MVC deben ser extremadamente delgada:

|  |  |
| --- | --- |
| 56 | Capítulo 7 |
|  | El desarrollo de ASP.NET MVC Core Aplicaciones |

* PUT api / autores2 / 5 [HttpPut ( "{id}")] [ValidateAuthorExists]

pública asíncrono de tareas <IActionResult> Ponga (int id, [FromBody] Autor

autor)

{

esperar \_authorRepository.UpdateAsync (autor); Ok volver ();

}

Puede leer más sobre la implementación de filtros y descargar una muestra de trabajo del artículo de MSDN, [Real World ASP.NET MVC Core Filtros.](https://msdn.microsoft.com/en-us/magazine/mt767699.aspx)



**Referencias - Aplicaciones Estructuración**

**Rebanadas de características para ASP.NET MVC Core - MSDN**

<https://msdn.microsoft.com/en-us/magazine/mt763233.aspx>

**MSDN - Simplificación de ASP.NET MVC Apps con la maquinilla de afeitar Páginas**

<https://msdn.microsoft.com/en-us/magazine/mt842512.aspx>

**Nuevo es el pegamento**

<http://www.weeklydevtips.com/005>

**MSDN - Real World ASP.NET MVC Core Filtros**

<https://msdn.microsoft.com/en-us/magazine/mt767699.aspx>

**áreas**

https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/areas

**filtros**

<https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/filters>

**ValidateModel Nuget Paquete**

https://www.nuget.org/packages/Ardalis.ValidateModel

Seguridad

Protección de aplicaciones web es un tema muy amplio, con muchas consideraciones. En su nivel más básico, la seguridad implica asegurar que sabe que una determinada solicitud está viniendo, y luego asegurarse de que esta solicitud sólo tiene acceso a los recursos que debería. La autenticación es el proceso de comparar las credenciales proporcionadas con una petición a los de un almacén de datos de confianza, para ver si la petición debe ser tratado como procedente de una entidad conocida. La autorización es el proceso de restringir el acceso a ciertos recursos basados ​​en la identidad del usuario. Una preocupación tercer seguridad es la protección de las solicitudes de espionaje por parte de terceros, sobre las que debería, al menos,[Asegúrese de que SSL es utilizada por su aplicación.](https://docs.microsoft.com/en-us/aspnet/core/security/enforcing-ssl)

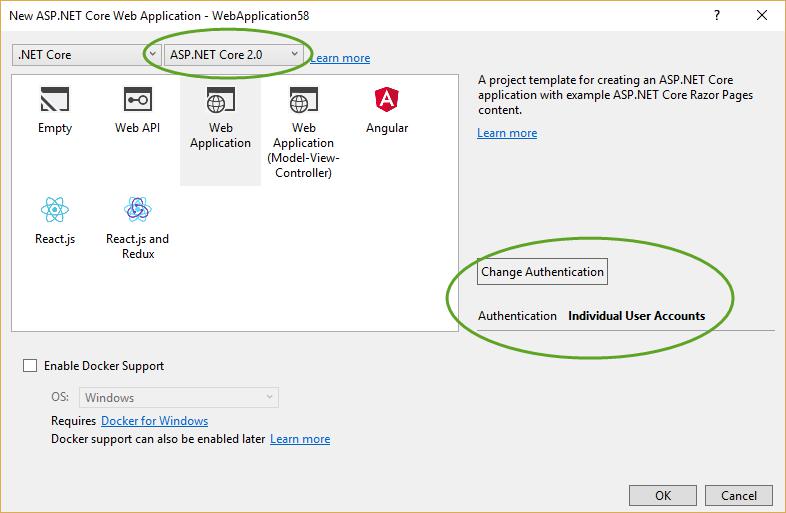
**Autenticación**

ASP.NET Núcleo de identidad es un sistema de afiliación puede utilizar para soportar la funcionalidad de inicio de sesión para su aplicación. Tiene soporte para cuentas de usuarios locales, así como el soporte del proveedor de acceso externa de proveedores como Microsoft cuenta, Twitter, Facebook, Google, y mucho más. Además de ASP.NET Core

|  |  |
| --- | --- |
| 57 | Capítulo 7 |
|  | El desarrollo de ASP.NET MVC Core Aplicaciones |

La identidad, la aplicación puede utilizar la autenticación de Windows, o un proveedor de identidad de terceros, como [Identity Server.](https://github.com/IdentityServer/IdentityServer4)

ASP.NET Core Identidad está incluido en las nuevas plantillas de proyecto si la opción Cuentas de usuario individual se selecciona. Esta plantilla incluye soporte para el registro, inicio de sesión, los inicios de sesión externos, contraseñas olvidadas, y la funcionalidad adicional.



***Figura 7-2.*** *Seleccionar cuentas de usuario individuales que han preconfigurado de identidad.*

Identidad de soporte está configurado en inicio, tanto en ConfigureServices y configurar:

ConfigureServices public void (servicios) IServiceCollection

{

services.AddDbContext <ApplicationDbContext> (Opciones =>

options.UseSqlServer (Configuration.GetConnectionString ( "cción DefaultConne")));

services.AddIdentity <ApplicationUser, IdentityRole> ()

.AddEntityFrameworkStores <ApplicationDbContext> ()

.AddDefaultTokenProviders ();

|  |  |
| --- | --- |
|  | services.AddMvc () |
|  | } |
| 58 | Capítulo 7 |
|  | El desarrollo de ASP.NET MVC Core Aplicaciones |

public void Configurar (IApplicationBuilder aplicación, IHostingEnvironment

env)

{

si (env.IsDevelopment ())

{

app.UseDeveloperExceptionPage ();

app.UseBrowserLink ();

app.UseDatabaseErrorPage ();

}

más

{

app.UseExceptionHandler ( "/ Error");

}

app.UseStaticFiles ();

app.UseAuthentication ();

app.UseMvc (rutas =>

{

routes.MapRoute (

nombre: "por defecto",

plantilla: "{controlador} / {action = Índice} / {id}?");

});

}

Es importante que UseAuthentication presentarse ante UseMvcen el método Configurar. Al configurar la identidad enConfigureServices, Se dará cuenta de una llamada a AddDefaultTokenProviders. Esto no tiene nada que ver con fichas que se pueden utilizar para proteger las comunicaciones de Internet, sino que se refiere a los proveedores que crean mensajes que se pueden enviar a los usuarios a través de SMS o correo electrónico para ellos para confirmar su identidad.

Usted puede aprender más acerca [configuración de la autenticación de dos factores](https://docs.microsoft.com/en-us/aspnet/core/security/authentication/2fa) y [permitiendo a los proveedores de acceso externos](https://docs.microsoft.com/en-us/aspnet/core/security/authentication/social/) a partir de los documentos oficiales ASP.NET Core.

**Autorización**

La forma más simple de la autorización implica restringir el acceso a usuarios anónimos. Al utilizar la maquinilla de afeitar de páginas, el control de acceso a las páginas individuales o carpetas que se hace al configurar MVC en el inicio:

services.AddMvc ()

.AddRazorPagesOptions (opciones =>

{

|  |  |
| --- | --- |
|  | options.Conventions.AuthorizeFolder ( "/ Orden"); |
|  | }); |
| 59 | Capítulo 7 |
|  | El desarrollo de ASP.NET MVC Core Aplicaciones |

Al utilizar controladores y puntos de vista, aplicar el atributo [Autorizar] a ciertos controladores o acciones. Si se utilizan papeles, el atributo puede ser extendido para restringir el acceso a usuarios que pertenecen a determinadas funciones, como se muestra:

[autorice (Roles = "HRManager, Finanzas")] public class SalaryController: Controller {

}

En este caso, los usuarios que pertenecen a cualquiera de los HRManager o papeles financieros (o ambos) tendrían acceso a la SalaryController. Para solicitar que un usuario pertenece a varios papeles (no sólo uno de varios), se puede aplicar el atributo varias veces, especificando un papel cada vez requerida.

Especificación de ciertos conjuntos de roles como cadenas en muchos controladores y acciones diferentes puede conducir a la repetición indeseable. Puede configurar las políticas de autorización, que encapsulan las reglas de autorización y especifique la política en lugar de papeles individuales cuando se aplica el atributo [Autorizar]:

[Autorizar (Política = "CanViewPrivateReport")] pública IActionResult ExecutiveSalaryReport () {

Ver regresar ();

}

El uso de las políticas de esta manera, puede separar los tipos de acciones que se están restringidos de los roles o normas específicas que le sean aplicables. Más tarde, si se crea un nuevo papel que debe tener acceso a ciertos recursos, sólo puede actualizar una política, en lugar de actualizar todas las listas de papeles en todos los atributos [Autorizar].

**Reclamaciones**

Reivindicaciones son pares de nombre y valor que representan las propiedades de un usuario autenticado. Por ejemplo, es posible almacenar el número de empleado de los usuarios como un reclamo. Las reivindicaciones a continuación, se puede utilizar como parte de las políticas de autorización. Se puede crear una política llamada “EmployeeOnly” que requiere la existencia de una demanda llamada “EmployeeNumber”, como se muestra en este ejemplo:

ConfigureServices public void (servicios) IServiceCollection

{

services.AddMvc ();

services.AddAuthorization (opciones =>

{

options.AddPolicy ( "EmployeeOnly", la política => policy.RequireClaim ( "EmployeeNumber"));

});

}

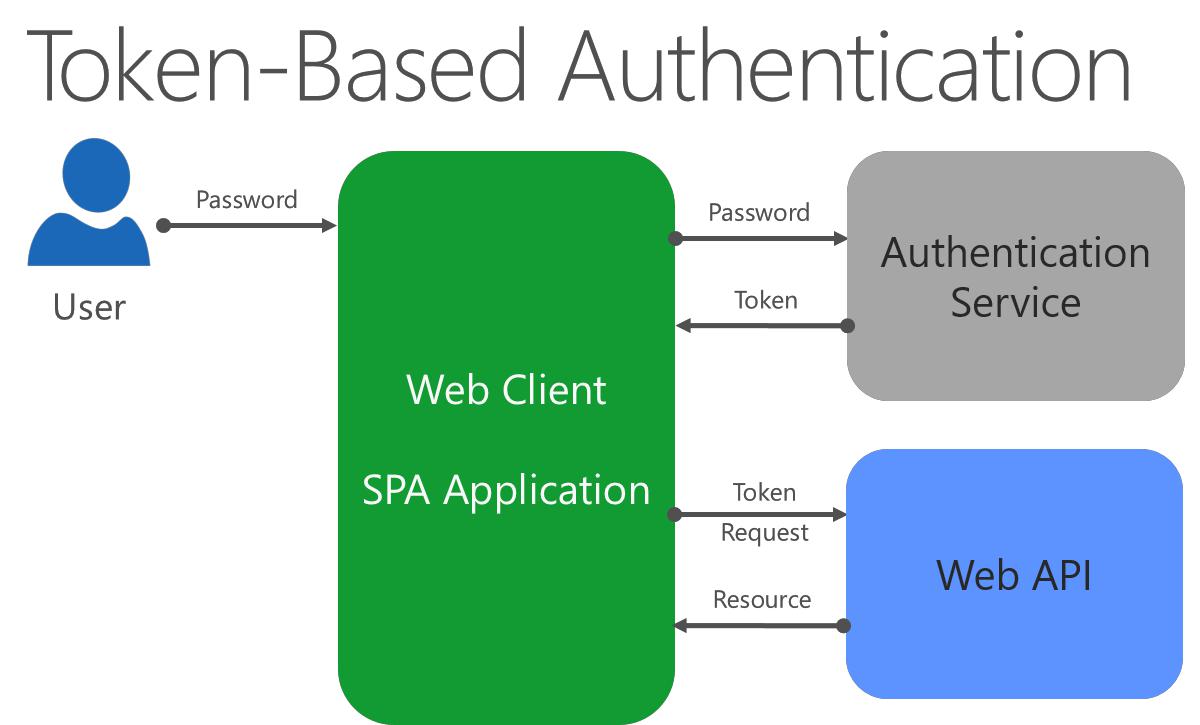
Esta política podría ser utilizado con el [Autorizar] atribuir a proteger cualquier controlador y / o acción, como se describe anteriormente.

|  |  |
| --- | --- |
| 60 | Capítulo 7 |
|  | El desarrollo de ASP.NET MVC Core Aplicaciones |

|  |  |
| --- | --- |
| 61 | Capítulo 7 |
|  | El desarrollo de ASP.NET MVC Core Aplicaciones |

**Asegurar Web APIs**

La mayoría de las API web deben implementar un sistema de autenticación basada en token. token de autenticación no tiene estado y diseñado para ser escalable. En un sistema de autenticación basada en token, el cliente debe autenticarse primero con el proveedor de autenticación. Si tiene éxito, el cliente se emite una señal, la cual es simplemente una cadena de caracteres criptográficamente significativa. Cuando el cliente se tiene que emitir una solicitud a una API, añade este token en el encabezamiento de la solicitud. Después, el servidor valida el token encontrado en el encabezado de la solicitud antes de completar la solicitud. Figura 7-3 muestra este proceso.



***Figura 7-3.*** *autenticación basada en token para las API Web.*



**Referencias - Seguridad**

**Seguridad general Docs**

https://docs.microsoft.com/en-us/aspnet/core/security/

**La aplicación de SSL en un núcleo de aplicaciones ASP.NET**

<https://docs.microsoft.com/en-us/aspnet/core/security/enforcing-ssl>

**Introducción a la Identidad**

<https://docs.microsoft.com/en-us/aspnet/core/security/authentication/identity>

**Introducción a la autorización**

<https://docs.microsoft.com/en-us/aspnet/core/security/authorization/introduction>

**Autentificación y autorización para aplicaciones de API en servicio Azure App**

<https://docs.microsoft.com/en-us/azure/app-service-api/app-service-api-authentication>

|  |  |
| --- | --- |
| 62 | Capítulo 7 |
|  | El desarrollo de ASP.NET MVC Core Aplicaciones |

Comunicación cliente

Además de servir páginas y respondiendo a las solicitudes de datos a través de las API web, aplicaciones ASP.NET núcleo puede comunicarse directamente con los clientes conectados. Esta comunicación de salida puede utilizar una variedad de tecnologías de transporte, siendo los más comunes WebSockets. ASP.NET Core SignalR es una biblioteca que hace que sea sencillo de tipo de funcionalidad de comunicación de servidor a cliente en tiempo real a sus aplicaciones. SignalR es compatible con una variedad de tecnologías de transporte, incluyendo WebSockets y abstrae muchos de los detalles de implementación del desarrollador.

ASP.NET Core SignalR se libera actualmente como una [alfa.](https://blogs.msdn.microsoft.com/webdev/2017/09/14/announcing-signalr-for-asp-net-core-2-0/) Además, otro [WebSockets de código abierto](https://github.com/radu-matei/websocket-manager) [bibliotecas](https://github.com/radu-matei/websocket-manager) son actualmente disponible.

la comunicación del cliente en tiempo real, ya sea utilizando WebSockets directamente u otras técnicas, son útiles en una variedad de escenarios de aplicación. Algunos ejemplos incluyen:

* aplicaciones en vivo salas de chat
* aplicaciones de monitoreo
* actualizaciones del progreso de Trabajo
* Notificaciones
* aplicaciones de formularios interactivos

Cuando la construcción de la comunicación del cliente en sus aplicaciones, normalmente hay dos componentes:

* conexión del lado servidor del gestor (SignalR Hub, WebSocketManager WebSocketHandler)
* biblioteca del lado del cliente

Los clientes no se limitan a los navegadores - aplicaciones móviles, aplicaciones de consola, y otras aplicaciones nativas también puede comunicarse mediante SignalR / WebSockets. El siguiente sencillo programa se hace eco de todo el contenido enviado a una aplicación de chat a la consola, como parte de una aplicación de ejemplo WebSocketManager:

Programa public class

{

\_Connection conexión estática privada;

public void main (String [] args)

{

StartConnectionAsync ();

\_connection.On ( "receiveMessage", (argumentos) => {

Console.WriteLine ($ "argumentos {[0]} dijo: {argumentos [1]}");

});

Console.ReadLine ();

StopConnectionAsync ();

}

public static asíncrono de tareas StartConnectionAsync ()

{

|  |  |
| --- | --- |
|  | \_Connection = conexión nueva (); |
| 63 | Capítulo 7 |
|  | El desarrollo de ASP.NET MVC Core Aplicaciones |

esperar

\_connection.StartConnectionAsync ( "WS: // localhost: 65110 / chat");

}

public static asíncrono de tareas StopConnectionAsync ()

{

esperar \_connection.StopConnectionAsync ();

}

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



**Referencias - comunicación de los clientes**

**ASP.NET Core SignalR**

<https://github.com/aspnet/SignalR>

**WebSocket Director**

https://github.com/radu-matei/websocket-manager

Dominio-Driven Design - En caso de aplicarlo?

Diseño de dominio-Driven (DDD) es un enfoque ágil a la creación de software que hace hincapié en centrarse en el dominio del negocio. Se pone un fuerte énfasis en la comunicación y la interacción con el experto (s) dominio de negocio que pueden relacionarse con los desarrolladores de cómo funciona el sistema en el mundo real. Por ejemplo, si usted está construyendo un sistema que se encarga de operaciones de bolsa, su experto de dominio podría ser un corredor de bolsa con experiencia. DDD está diseñado para hacer frente a los problemas de negocios grandes y complejos, y con frecuencia no es apropiado para aplicaciones más pequeñas y simples, como la inversión en la comprensión y la modelización del dominio no vale la pena.

Cuando la construcción de software siguiendo un enfoque DDD, su equipo (incluidos los interesados ​​no técnicos y colaboradores) debe desarrollar un lenguaje ubicuo para el espacio del problema. Es decir, la misma terminología se debe utilizar para el concepto del mundo real que está siendo modelado, el equivalente en software, y cualquier estructura que pudiera existir para persistir el concepto (por ejemplo, tablas de bases de datos). Por lo tanto, los conceptos descritos en el idioma en todas partes deben constituir la base para su modelo de dominio.

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

* [entidades,](http://deviq.com/entity/) cual representar objetos con un hilo de identidad. Entidades se almacenan típicamente en la persistencia con una clave por los que más tarde se pueden recuperar.
* [agregados,](http://deviq.com/aggregate-pattern/) cual representar grupos de objetos que deben ser persistieron como una unidad.
* [objetos de valor,](http://deviq.com/value-object/) cual representar conceptos que pueden ser comparados basa en la suma de su valores de propiedad. Por ejemplo, DateRange que consta de una fecha de inicio y fin.
* [acontecimientos de dominio,](https://martinfowler.com/eaaDev/DomainEvent.html) cual representar las cosas que suceden dentro del sistema que son de interés para otras partes del sistema.

Tenga en cuenta que un modelo de dominio DDD debe encapsular comportamiento complejo dentro del modelo. Entidades, en particular, deben ser no sólo colecciones de propiedades. Cuando el modelo de dominio carece de comportamiento y

|  |  |
| --- | --- |
| 64 | Capítulo 7 |
|  | El desarrollo de ASP.NET MVC Core Aplicaciones |

simplemente representa el estado del sistema, se dice que es una [modelo anémica,](http://deviq.com/anemic-model/) que es indeseable en DDD.

Además de estos tipos de modelo, DDD emplea típicamente una variedad de patrones:

* + [Repositorio,](http://deviq.com/repository-pattern/) para abstraer detalles de persistencia.
  + [Fábrica,](https://en.wikipedia.org/wiki/Factory_method_pattern) para encapsular la creación de objetos complejos.
  + acontecimientos de dominio, para desacoplar el comportamiento dependiente de la activación comportamiento.
  + [Servicios,](http://gorodinski.com/blog/2012/04/14/services-in-domain-driven-design-ddd/) para encapsular complejas de comportamiento y / o implementación de infraestructura detalles.
  + [Mando,](https://en.wikipedia.org/wiki/Command_pattern) para desacoplamiento dando órdenes y ejecutar el comando en sí.
  + [Especificación,](http://deviq.com/specification-pattern/) para encapsular detalles de la consulta.

DDD también recomienda el uso de la arquitectura limpia discutido previamente, lo que permite para el acoplamiento flojo, encapsulación, y el código que puede ser fácilmente verificada utilizando pruebas de unidad.

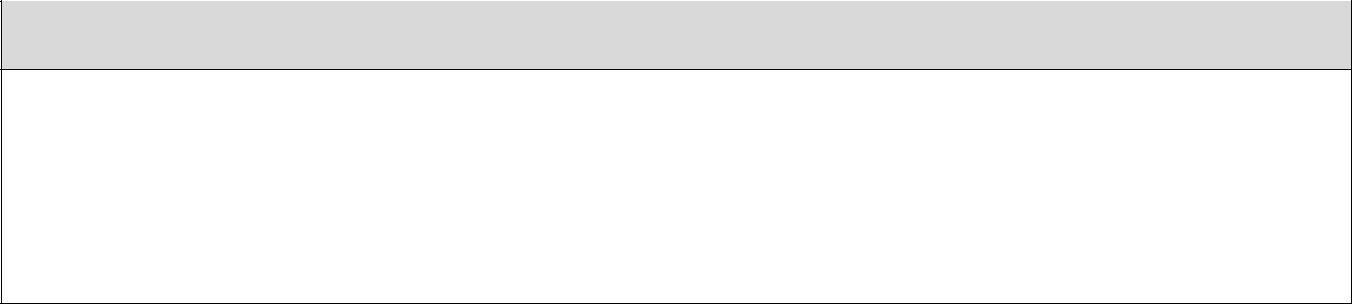
**¿Cuándo se debe aplicar DDD**

1. se adapta bien a las grandes aplicaciones de negocios importante con la complejidad (no sólo técnica). La aplicación debería requerir el conocimiento de los expertos de dominio. Debe haber un comportamiento significativo en el propio modelo de dominio, que representa las reglas de negocio y las interacciones más allá de simplemente almacenar y recuperar el estado actual de varios registros de almacenes de datos.

**Cuando no se debería solicitar DDD**

1. implica inversiones en el modelado, la arquitectura y la comunicación que pueden no estar garantizada por pequeñas aplicaciones o aplicaciones que son esencialmente sólo CRUD (crear / leer / actualizar / borrar). Si decide acercarse a su aplicación siguiendo DDD, pero encuentra que su dominio tiene un modelo anémicos con ningún comportamiento, puede que tenga que reconsiderar su enfoque. Puede ser que su aplicación puede no necesitar DDD, o puede que necesite asistencia refactorización su aplicación para encapsular la lógica de negocio en el modelo de dominio, en lugar de en su base de datos o la interfaz de usuario.

Un enfoque híbrido sería utilizar sólo DDD para las áreas transaccionales o más complejos de la aplicación, pero no para CRUD simple o partes de la aplicación de sólo lectura. Por ejemplo, no es necesario tener las limitaciones de un agregado si está consultando los datos para visualizar un informe o para visualizar los datos de un cuadro de mando. Es perfectamente aceptable tener un modelo de lectura independiente, más simple para tales requisitos.



**Diseño de dominio-Driven - Referencias**

**Fundamentos de Diseño de dominio-Driven (curso)**

<http://bit.ly/ddd-fundamentals>

**Patrones de diseño Biblioteca (por supuesto)**

<http://bit.ly/PS-design-patterns>

**DDD en Inglés Simple (Stackoverflow respuesta)**

<http://bit.ly/2pmVgK2>

|  |  |
| --- | --- |
| sesenta y cinco | Capítulo 7 |
|  | El desarrollo de ASP.NET MVC Core Aplicaciones |

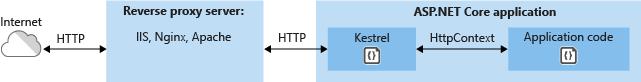
|  |  |
| --- | --- |
| 66 | Capítulo 7 |
|  | El desarrollo de ASP.NET MVC Core Aplicaciones |

Despliegue

Hay algunos pasos que intervienen en el proceso de despliegue de la aplicación ASP.NET Core, independientemente de donde se encuentra alojado. El primer paso es la publicación de la solicitud, que se puede hacer utilizando eldotNET publicar comando CLI. Esto compilará la aplicación y colocar todos los archivos necesarios para ejecutar la aplicación en una carpeta designada. Cuando se implementa desde Visual Studio, este paso se realiza de forma automática. La carpeta contiene publicar archivos .exe y .dll para la aplicación y sus dependencias. Una aplicación autónomo incluirá también una versión del tiempo de ejecución de .NET. aplicaciones ASP.NET Core también incluirán los archivos de configuración, activos de clientes estáticos y vistas MVC.

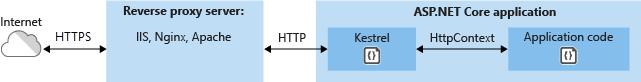
aplicaciones ASP.NET centrales son aplicaciones de consola que se debe iniciar cuando se inicia el servidor y se reinicia si la aplicación (o servidor) se bloquea. Un administrador de procesos se puede utilizar para automatizar este proceso. Los responsables de los procesos más comunes para ASP.NET Core son Nginx y Apache en Linux y IIS o servicio de Windows en Windows.

Además de un gestor de procesos, aplicaciones ASP.NET Core alojados en el servidor web Kestrel pueden beneficiarse del uso de un servidor proxy inverso (esto era un requisito para ASP.NET 1.x Core). Un servidor proxy inverso recibe peticiones HTTP de Internet y los reenvía al cernícalo después de un tratamiento preliminar. servidores proxy inverso proporcionan una capa de seguridad para la aplicación, y se utilizan para los despliegues de borde (expuestos al tráfico de internet). Como de ASP.NET Core 2.0, Kestrel ahora es compatible con las implementaciones de borde directamente. Sin embargo, Kestrel aún no soporta alojar múltiples aplicaciones en el mismo puerto, así como técnicas de encabezados de host no se pueden utilizar con ella para permitir alojar múltiples aplicaciones en el mismo puerto y la dirección IP (un escenario de IIS admite).



***Figura 7-4.*** *ASP.NET Core alojado en Cernícalo detrás de un servidor proxy inverso*

Otro escenario en el que un proxy inverso puede ser útil es asegurar múltiples aplicaciones que utilizan SSL / HTTPS. En este caso, sólo tendría que haber SSL configurado el proxy inverso. La comunicación entre el servidor proxy inverso y Kestrel podría tener lugar a través de HTTP, como se muestra en la Figura 7-5.



***Figura 7-5.*** *ASP.NET Core alojada detrás de un servidor proxy inverso HTTPS-asegurado*

Un enfoque cada vez más popular es la de alojar su aplicación ASP.NET Core en un recipiente acoplable, que luego pueden ser alojados localmente o desplegado en Azure para el alojamiento basado en la nube. El envase acoplable podría contener el código de aplicación, que se ejecuta en Kestrel, y sería desplegado detrás de un servidor proxy inverso, como se muestra arriba.

|  |  |
| --- | --- |
| 67 | Capítulo 7 |
|  | El desarrollo de ASP.NET MVC Core Aplicaciones |

Si usted es anfitrión de la aplicación en Azure, puede utilizar Microsoft Azure Application Gateway como un dispositivo virtual dedicada a proporcionar varios servicios. Además de actuar como un proxy inverso para aplicaciones individuales, Application Gateway también puede ofrecer las siguientes características:

* balanceo de carga HTTP
* descarga SSL (SSL sólo para Internet)
* De extremo a extremo SSL
* enrutamiento multi-sitio (consolidar hasta 20 sitios en una sola pasarela de aplicaciones)
* firewall de aplicaciones Web
* apoyo WebSocket
* El diagnóstico avanzado

*Más información sobre las opciones de implementación Azure en el capítulo 10.*



**Referencias - Despliegue**

**Hospedaje y de despliegue general**

<https://docs.microsoft.com/en-us/aspnet/core/publishing/>

**Cuándo utilizar Cernícalo con un proxy inverso**

<https://docs.microsoft.com/en-us/aspnet/core/fundamentals/servers/kestrel#when-to-use-kestrel-with-a-reverse-proxy>

**Alojar aplicaciones ASP.NET Core en acoplable**

https://docs.microsoft.com/en-us/aspnet/core/publishing/docker

**Presentación de Azure Application Gateway**

<https://docs.microsoft.com/en-us/azure/application-gateway/application-gateway-introduction>

|  |  |
| --- | --- |
| 68 | Capítulo 7 |
|  | El desarrollo de ASP.NET MVC Core Aplicaciones |

SECCIÓN 8

Trabajo con datos en ASP.NET Aplicaciones Core

*“Los datos son una cosa preciosa y durará más tiempo que los propios sistemas.”*

Tim Berners-Lee

Resumen

acceso a datos es una parte importante de casi cualquier aplicación de software. ASP.NET Core es compatible con una variedad de opciones de acceso a datos, incluyendo Entidad Marco de base (Entity Framework y 6 también), y puede trabajar con cualquier marco de acceso a datos .NET. La elección del marco de acceso de datos a utilizar depende de las necesidades de la aplicación. Abstraer estas opciones de los proyectos ApplicationCore y la interfaz de usuario, y encapsular los detalles de implementación en infraestructura, ayuda a producir débilmente acoplado, software comprobable.

Entidad Marco de base (para bases de datos relacionales)

Si va a escribir una nueva aplicación ASP.NET núcleo que tiene que trabajar con datos relacionales, a continuación, Entity Framework Core (Core EF) es la forma recomendada para su aplicación a acceder a sus datos. EF Core es un mapeador objeto-relacional (O / RM) que permite a los desarrolladores de .NET a persistir objetos hacia y desde una fuente de datos. Se elimina la necesidad de que la mayor parte de los desarrolladores de código de acceso a datos que suelen tener que escribir. Al igual que ASP.NET Core, Core EF ha sido reescrito desde cero para soportar aplicaciones modulares y multiplataforma. Se agrega a su aplicación como un paquete NuGet, lo configura en Inicio, y solicitar a través de la inyección de dependencia siempre que lo necesite.

Si está utilizando ASP.NET 2.0 Core y tiene una referencia a la Microsoft.AspNetCore.Allmetapaquete, que ya tiene referencias a EF Core y sus paquetes de SQL Server y Inmemory. Si usted no está usando este metapaquete, puede agregar referencias a la entidad Marco de base usando las instrucciones siguientes:

Capítulo 8

Para utilizar EF Core con una base de datos SQL Server, ejecute el siguiente comando CLI dotnet:

dotnet paquete adicional Microsoft.EntityFrameworkCore.SqlServer

Para añadir soporte para una fuente de datos InMemory, para la prueba:

dotnet paquete adicional Microsoft.EntityFrameworkCore.InMemory

**el DbContext**

Para trabajar con EF Core, necesita una subclase de DbContext. Esta clase contiene propiedades que representan colecciones de las entidades la aplicación va a trabajar. La muestra eShopOnWeb incluye unaCatalogContext con colecciones de artículos, marcas y tipos:

CatalogContext clase pública: DbContext

{

CatalogContext pública (DbContextOptions <> CatalogContext opciones)

* base (opciones)

{

}

públicas DbSet <> CatalogItem CataloGitems {get; conjunto; } Públicas DbSet <> CatalogBrand CatalogBrands {get; conjunto; }

públicas DbSet <> CatalogType CatalogTypes {get; conjunto; }

}

Tu DbContext debe tener un constructor que acepta DbContextOptionsy pasar este argumento para el constructor DbContext base. Tenga en cuenta que si usted tiene sólo un DbContext en su aplicación, puede pasar una instancia deDbContextOptions, Pero si usted tiene más de una debe utilizar el genérico DbContextOptions <T> escribir, pasando en su tipo DbContext como el parámetro genérico.

**Configuración de EF Core**

En su aplicación ASP.NET Core, por lo general va configurar EF Core en su ConfigureServicesmétodo. EF Core utiliza unaDbContextOptionsBuilder, Que soporta varios métodos de extensión útil para simplificar su configuración. tp configureCatalogContext utilizar una base de datos SQL Server con una cadena de conexión se define en la configuración, deberá añadir el siguiente código a

ConfigureServices:

services.AddDbContext <CatalogContext> (opciones => options.UseSqlServer (Configuration.GetConnectionString ( "cción DefaultConne")));

Para utilizar la base de datos en memoria:

services.AddDbContext <CatalogContext> (Opciones =>

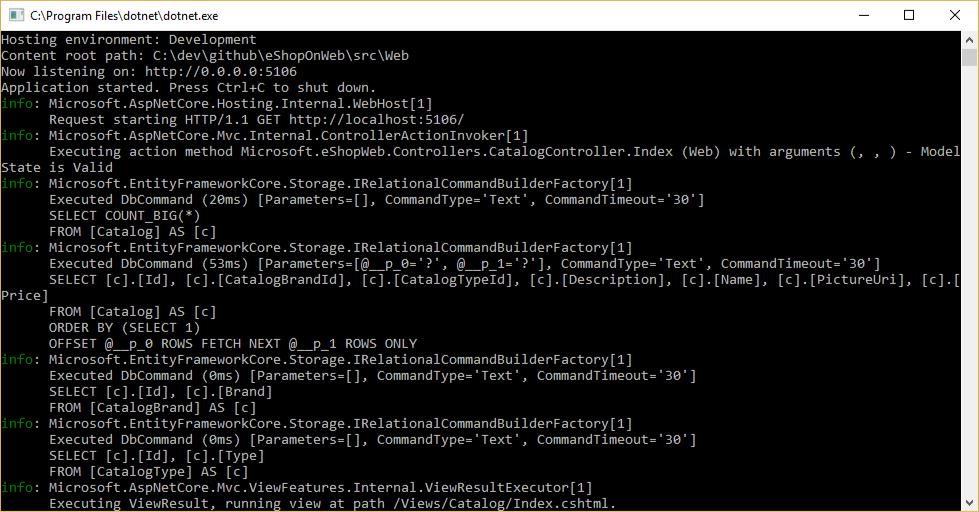
options.UseInMemoryDatabase ( “Catálogo”));

|  |  |
| --- | --- |
| 70 | Capítulo 8 |
|  | Trabajar con datos |

Debe especificar un nombre para su base de datos en memoria. Este nombre debe ser único para su DbContext asociado.

Una vez que haya instalado EF Core, creó un tipo DbContext niño, y configurado en ConfigureServices, ya está listo para usar EF Core. Puede solicitar una instancia de su tipo DbContext en cualquier servicio que necesita y empezar a trabajar con sus entidades persistido usando LINQ como si fueran simplemente en una colección. EF Core hace el trabajo de la traducción de sus expresiones LINQ en consultas SQL para almacenar y recuperar sus datos.

Puede ver las consultas de EF Core está ejecutando mediante la configuración de nivel de registro de su aplicación para ser fijado a al menos la información, como se muestra en la Figura 8-1.



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

Puede configurar el registro durante el desarrollo de una plantilla de aplicación por defecto editando el appsettings-

Development.json archivo. Por ejemplo:

{

"Registro": {

"IncludeScopes": false,

"Nivel de registro": {

"Defecto": "Depuración",

"Información del sistema",

"Microsoft": "Información"

}

}

}

|  |  |
| --- | --- |
| 71 | Capítulo 8 |
|  | Trabajar con datos |

**Obtención y almacenamiento de datos**

Para recuperar datos de EF Core, se accede a la propiedad adecuada y el uso de LINQ para filtrar el resultado. También puede utilizar LINQ para llevar a cabo la proyección, transformando el resultado de un tipo a otro. En el siguiente ejemplo se CatalogBrands recuperar, ordenar por nombre, filtrada por su propiedad Enabled, y proyecta sobre un tipo SelectListItem:

brandItems var = esperan \_context.CatalogBrands

.Where (b => b.Enabled)

.OrderBy (b => b.Name)

.Seleccionar (b => new SelectListItem {

Valor = b.Id, Text = b.Name})

.ToListAsync ();

Es importante en el ejemplo anterior para añadir la llamada a ToListAsynccon el fin de ejecutar la consulta inmediatamente. De lo contrario, la declaración asignará unaIQueryable <SelectListItem> a brandItems, Que no se ejecutará hasta que se enumera. Hay pros y los contras de volverIQueryableresultados de métodos. Permite la consulta EF Core construirá para ser modificado adicionalmente, pero también puede dar lugar a errores que sólo se producen en tiempo de ejecución, si se añaden las operaciones para la consulta que EF Core no puede traducir. Por lo general es más seguro para pasar los filtros en el método de realizar el acceso a los datos, y volver una colección en memoria (por ejemplo,Lista <T>) como el resultado. Estos filtros se pueden encapsular adicionalmente usando el[patrón de especificación.](http://deviq.com/specification-pattern/)

EF Core seguimiento de los cambios en las entidades que descarge de persistencia. Para guardar los cambios en una entidad de seguimiento, que acaba de llamar el método SaveChanges en el DbContext, asegurándose de que es la misma instancia DbContext que se utilizó para buscar a la entidad. Adición y eliminación de entidades se realiza directamente en la propiedad DbSet adecuada, de nuevo con una llamada a SaveChanges para ejecutar los comandos de base de datos. El siguiente ejemplo demuestra la adición, actualización y eliminación de entidades de persistencia.

// crear

var = new Newbrand CatalogBrand () {Marca = "Acme"}; \_context.Add (Newbrand);

esperar \_context.SaveChangesAsync ();

// leer y actualizar

var existingBrand = \_context.CatalogBrands.Find (1); existingBrand.Brand = "Marca actualización"; esperar \_context.SaveChangesAsync ();

// leer y borrar (alternativa Find sintaxis)

var brandToDelete = \_context.Find <CatalogBrand> (2); \_context.CatalogBrands.Remove (brandToDelete); esperar \_context.SaveChangesAsync ();

EF Core soporta ambos métodos sincrónicos y asincrónicos para ir a buscar y salvar. En las aplicaciones web, se recomienda utilizar el patrón esperan ser asíncrono / con los métodos asincrónicos, de modo que los hilos del servidor web no están bloqueados a la espera de las operaciones de acceso a datos para completar.

|  |  |
| --- | --- |
| 72 | Capítulo 8 |
|  | Trabajar con datos |

**Ir a buscar datos relacionados**

Cuando EF Core recupera entidades, rellena todas las propiedades que se almacenan directamente con esa entidad en la base de datos. propiedades de navegación, tales como listas de entidades relacionadas, no son pobladas y pueden tener su valor establecido en null. Esto asegura EF Core no es ir a buscar más datos que los que se necesita. Esto es especialmente importante para las aplicaciones web, que deben procesar rápidamente las solicitudes y las respuestas de retorno de una manera eficiente. Para incluir relaciones con una entidad utilizando la carga ansiosa, se especifica el establecimiento mediante el método de extensión Incluir en la consulta, como se muestra:

* .include requiere el uso de brandsWithItems Microsoft.EntityFrameworkCore var = esperan \_context.CatalogBrands

.include (b => b.Items)

.ToListAsync ();

Puede incluir varias relaciones, y también se puede incluir el uso de sub-relaciones ThenInclude. EF Core ejecutará una sola consulta para recuperar el conjunto resultante de las entidades.

Además de la encapsulación de lógica de filtrado, la especificación puede especificar la forma de los datos a ser devuelto, incluyendo las propiedades para poblar. La muestra eShopOnWeb incluye varias especificaciones que demuestran que encapsula información de carga ansiosos dentro de la especificación. Se puede ver cómo se utiliza la memoria descriptiva como parte de una consulta aquí:

pública IEnumerable <T> lista (ISpecification <T> especificación)

{

* buscar una consultables que incluye todos incluye basada en la expresión var queryableResultWithIncludes = spec.Includes

.Aggregate (. \_dbContext.Set <T> () AsQueryable (), (corriente, incluir) => current.Include (incluir));

* modificar el IQueryable incluir cualquier sentencias de inclusión basados ​​en cadenas

var secondaryResult = spec.IncludeStrings

.Aggregate (queryableResultWithIncludes,

(Corriente, incluir) => current.Include (incluir));

* devolver el resultado de la consulta utilizando criterios de la expresión de la especificación

volver secondaryResult

.Where (spec.Criteria)

.AsEnumerable ();

}

Otra opción para los datos relacionados con la carga es utilizar la carga explícita. explícita carga le permite cargar datos adicionales en una entidad que ya se ha recuperado. Dado que se trata de una solicitud separada de la base de datos, no se recomienda para aplicaciones web, lo que debería reducir al mínimo el número de idas y vueltas de bases de datos hechas por encargo.

|  |  |
| --- | --- |
| 73 | Capítulo 8 |
|  | Trabajar con datos |

*Carga lenta* es una característica que se carga automáticamente los datos relacionados a medida que se hace referencia en la solicitud. No es compatible actualmente con EF Core, pero como con carga explícita que por lo general se debe desactivar para aplicaciones web.

**datos de encapsulación**

Entity Framework Core soporta varias características que permiten su modelo para encapsular adecuadamente su estado. Un problema común en los modelos de dominio es que exponen propiedades recogida de navegación como tipos de listas de acceso público. Esto permite que cualquier colaborador para manipular el contenido de estos tipos de colección, que pueden eludir las reglas de negocio importantes relacionados con la recolección, posiblemente dejando el objeto en un estado no válido. La solución a esto es exponer acceso de sólo lectura a las colecciones relacionadas, y explícitamente proporcionar métodos que definen las formas en que los clientes puedan manipularlos, como en este ejemplo:

Cesta clase pública: BaseEntity

{

public string ID\_COMPRADOR {get; conjunto; }

Lista de sólo lectura privada <BasketItem> \_items = new List <BasketItem> ();

públicas IReadOnlyCollection <BasketItem> Artículos => \_items.AsReadOnly ();

AddItem pública vacío (int cataloGitemId, precioUnitario decimal, int cantidad = 1)

{

if (! Items.Any (i => i.CataloGitemId == cataloGitemId))

{

\_items.Add (nuevo BasketItem ()

{

CataloGitemId = cataloGitemId,

Cantidad = cantidad,

PrecioUnidad = precioUnitario

});

regreso;

}

var existinGitem = Items.FirstOrDefault (i => i.CataloGitemId == cataloGitemId);

existinGitem.Quantity + = cantidad;

}

}

Tenga en cuenta que este tipo de entidad no expone una lista pública o la propiedad ICollection, sino que expone un tipo IReadOnlyCollection que ajusta el tipo de lista subyacente. Al utilizar este patrón, se puede indicar a la entidad Marco de base a utilizar el campo de respaldo de este modo:

ConfigureBasket private void (EntityTypeBuilder <Basket> constructor)

|  |  |
| --- | --- |
|  | { |
| 74 | Capítulo 8 |
|  | Trabajar con datos |

var = navegación

builder.Metadata.FindNavigation (nombredel (Basket.Items));

navigation.SetPropertyAccessMode (PropertyAccessMode.Field);

}

Otra forma en que usted puede mejorar su modelo de dominio es a través del uso de objetos de valor para los tipos que carecen de identidad y sólo se distinguen por sus propiedades. El uso de tipos tales como las propiedades de sus entidades puede ayudar a mantener la lógica específica al objeto de valor que le corresponde, y puede evitar la lógica duplicado entre varias entidades que utilizan el mismo concepto. En Entidad Marco de base, puede persistir objetos de valor en la misma mesa que su entidad propietaria mediante la configuración del tipo como una entidad de propiedad, así:

ConfigureOrder private void (EntityTypeBuilder <Order> constructor)

{

builder.OwnsOne (o => o.ShipToAddress);

}

En este ejemplo, la propiedad es de tipo ShipToAddress Dirección. La dirección es un objeto de valor con varias propiedades (calle, ciudad, etc.). EF Core mapear el objeto Order a su tabla con una columna por propiedad Dirección, anteponiendo el nombre de cada columna con el nombre de la propiedad. Por ejemplo, la tabla Order en este caso incluiría columnas “ShipToAddress\_City” “ShipToAddress\_Street”, etc.

**Las conexiones flexibles**

Los recursos externos, como bases de datos SQL en ocasiones pueden no estar disponibles. En los casos de indisponibilidad temporal, las aplicaciones pueden usar la lógica de reintento para evitar lanzar una excepción. Esta técnica se conoce comúnmente como la resistencia de conexión. Puede implementar su[propia de reintento con exponencial](https://docs.microsoft.com/en-us/azure/architecture/patterns/retry) [Apártate](https://docs.microsoft.com/en-us/azure/architecture/patterns/retry) técnica al tratar de volver a intentar con un tiempo de espera de forma exponencial cada vez mayor, hasta un máximo el número de intentos que se ha alcanzado. Esta técnica abarca el hecho de que los recursos de la nube podría no estar disponible intermitentemente durante cortos períodos de tiempo, lo que resulta en un fracaso de algunas peticiones.

Para SQL Azure DB, Entity Framework Core ya proporciona la resistencia de conexión de base de datos interna y lógica de reintento. Pero es necesario tener la estrategia de ejecución de Entity Framework para cada conexión DbContext si usted quiere tener conexiones EF núcleo elástico.

Por ejemplo, el código siguiente en el nivel de conexión EF Core permite conexiones SQL resilientes que se vuelve a intentar si falla la conexión.

|  |  |
| --- | --- |
| 75 | Capítulo 8 |
|  | Trabajar con datos |

* Startup.cs de cualquier clase pública de inicio de la API Web ASP.NET Core

{

pública IServiceProvider ConfigureServices (servicios IServiceCollection)

{

// ...

services.AddDbContext <OrderingContext> (Opciones =>

{

options.UseSqlServer (configuración [ "ConnectionString"], sqlServerOptionsAction: sqloptions =>

{

sqlOptions.EnableRetryOnFailure (maxRetryCount: 5,

maxRetryDelay: TimeSpan.FromSeconds (30),

errorNumbersToAdd: null); });

});

}

// ...

**estrategias de ejecución y transacciones explícitas utilizando BeginTransaction y múltiples DbContexts**

Cuando reintentos están habilitadas en las conexiones básicas EF, cada operación que realice utilizando EF núcleo se convierte en su propio funcionamiento retriable. Cada consulta y cada llamada aGuardar cambios se volverá a intentar como una unidad si se produce un error transitorio.

Sin embargo, si su código inicia una transacción utilizando BeginTransaction, Que está definiendo su propio grupo de operaciones que deben ser tratados como una unidad de todo dentro de la transacción se pueden revertir si se produce un fallo. Verá una excepción como la siguiente si intenta ejecutar la transacción cuando se utiliza una estrategia de ejecución EF (reintentar la política) y que incluyen variosGuardar cambios desde múltiples DbContexts en ella.

System.InvalidOperationException: La estrategia de ejecución configurado 'SqlServerRetryingExecutionStrategy' no admite transacciones iniciadas por el usuario. Usar la estrategia de ejecución que devuelve 'DbContext.Database.CreateExecutionStrategy ()' para ejecutar todas las operaciones en la transacción como una unidad retriable.

La solución es invocar manualmente la estrategia de ejecución EF con un delegado en representación de todo lo que necesita ser ejecutado. Si se produce un error transitorio, la estrategia de ejecución invocará el delegado de nuevo. El código siguiente muestra cómo implementar este enfoque:

* El uso de una estrategia de resiliencia EF Core cuando se utilizan múltiples DbContexts
* dentro de una transacción explícita

|  |  |
| --- | --- |
| 76 | Capítulo 8 |
|  | Trabajar con datos |

* Ver:
* https://docs.microsoft.com/en-us/ef/core/miscellaneous/connection-resiliency

estrategia var = \_catalogContext.Database.CreateExecutionStrategy (); esperar strategy.ExecuteAsync (asíncrono () =>

{

* El logro de la atomicidad entre la operación original de la base de datos del catálogo y la
* IntegrationEventLog gracias a una transacción local

usando (transacción var =

\_catalogContext.Database.BeginTransaction ())

{

\_catalogContext.CataloGitems.Update (CatalogItem); esperar \_catalogContext.SaveChangesAsync ();

* Enviar al EventLog sólo si cambia el precio del producto si (raiseProductPriceChangedEvent) {

esperar \_integrationEventLogService.SaveEventAsync (priceChangedEvent);

}

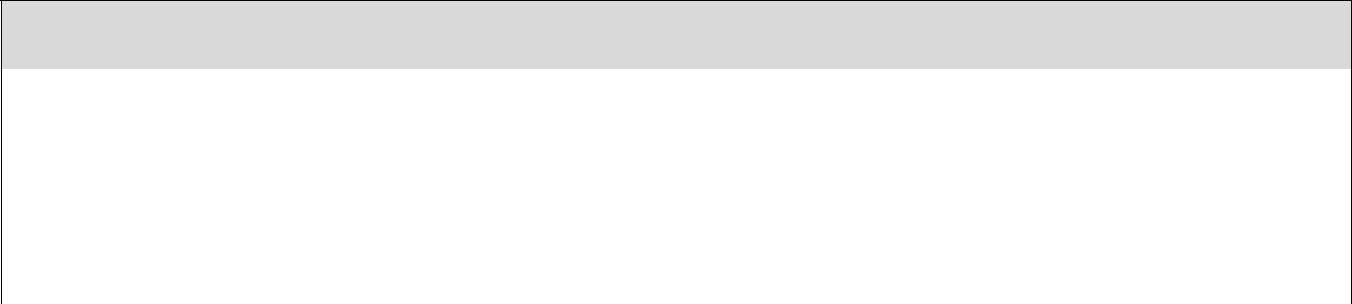
transaction.Commit ();

}

});

El primero es el DbContext \_catalogContext y la segunda DbContext está dentro de la

objeto \_integrationEventLogService. Finalmente, la acción Commit se llevaría a cabo múltiples DbContexts y utilizando una estrategia de ejecución EF.



**Referencias - Entidad marco básico**

**EF Core Docs**

<https://docs.microsoft.com/en-us/ef/>

**EF Core: Datos Relacionados**

<https://docs.microsoft.com/en-us/ef/core/querying/related-data>

**Evitar Lazy Loading Entidades en aplicaciones ASP.NET**

http://ardalis.com/avoid-lazy-loading-entities-in-asp-net-applications

EF Core o micro-ORM?

Mientras EF Core es una gran opción para la gestión de persistencia, y en su mayor parte encapsula datos de bases de datos de los desarrolladores de aplicaciones, no es la única opción. Otra alternativa de código abierto es muy popular[Apuesto,](https://github.com/StackExchange/Dapper) un denominado micro-ORM. Un micro-ORM es una herramienta ligera, menos con todas las funciones de mapeo de objetos a las estructuras de datos. En el caso de Dapper, sus objetivos de diseño se centran en el rendimiento, en lugar de encapsular totalmente las consultas subyacentes que utiliza para recuperar y actualizar datos. Porque hace SQL no abstracta del promotor, Dapper es “más cercano al metal” y permite a los desarrolladores escribir las consultas exactas que desea utilizar para una operación de acceso de datos dado.

EF Core tiene dos características significativas que proporciona que la separan de Dapper pero también se suman a su sobrecarga de rendimiento. La primera es la traducción de expresiones LINQ a SQL. Estas son traducciones

|  |  |
| --- | --- |
| 77 | Capítulo 8 |
|  | Trabajar con datos |

almacena en caché, pero aún así existe sobrecarga en el desempeño de ellas por primera vez. El segundo es el seguimiento de cambios en las entidades (de modo que las instrucciones de actualización eficientes pueden ser generados). Este comportamiento puede ser[desactivado para consultas específicas](https://docs.microsoft.com/en-us/ef/core/querying/tracking) por utilizando el AsNoTracking extensión. EF Core también genera SQL consultas que por lo general son muy eficientes y en todo caso perfectamente aceptable desde un punto de vista de rendimiento, pero si necesita un control preciso sobre la consulta precisa para ser ejecutado, se pueden pasar en SQL personalizado (o ejecutar un procedimiento almacenado) usando EF Core, también. En este caso, apuesto todavía supera a EF Core, pero sólo un poco. Julie Lerman presenta algunos datos de rendimiento en su artículo MSDN estará el año 2016[Apuesto,](https://msdn.microsoft.com/en-us/magazine/mt703432.aspx) [Marco de la entidad, y aplicaciones híbridas.](https://msdn.microsoft.com/en-us/magazine/mt703432.aspx) Adicional datos de referencia de rendimiento para una variedad de datos métodos de acceso se pueden encontrar en [el sitio Dapper.](https://github.com/StackExchange/Dapper)

Para ver cómo la sintaxis para Dapper varía de EF Core, considere estas dos versiones de un mismo método para recuperar una lista de elementos:

// EF Core

\_context CatalogContext sólo lectura privada;

pública asíncrono de tareas <IEnumerable <CatalogType >> GetCatalogTypes ()

{

retorno esperan \_context.CatalogTypes.ToListAsync ();

}

// Dapper

\_conn SqlConnection sólo lectura privada; pública asíncrono de tareas <IEnumerable <CatalogType >> GetCatalogTypesWithDapper () {

esperan el retorno \_conn.QueryAsync <CatalogType> ( "SELECT \* FROM

CatalogType ");

}

Si usted necesita para construir gráficos de objetos más complejos con Dapper, tiene que escribir las consultas asociadas a ti mismo (a diferencia de la adición de una Incluircomo lo haría en EF Core). Esto es apoyado a través de una variedad de sintaxis, que incluye una característica llamada Multi mapeo que le permite asignar filas individuales a múltiples objetos asignada. Por ejemplo, dada una clase del poste con el dueño de una propiedad de tipo de usuario, el siguiente SQL devolvería todos los datos necesarios:

seleccionar \* de p #Posts

izquierda unirse a #usuarios u en u.Id = p.OwnerId

Ordenar por p.Id

Cada fila devuelta incluye tanto los datos de usuario y Pon. Dado que los datos de usuario deben estar unidos al

Colocar nuestros datos a través de su propiedad de propietario, se utiliza la siguiente función:

(Post, usuario) => {post.Owner = usuario; volver poste; }

El código completo lista para devolver una colección de mensajes con sus bienes propietario poblada con los datos de usuario asociados serían:

|  |  |
| --- | --- |
| 78 | Capítulo 8 |
|  | Trabajar con datos |

SQL var = @ "SELECT \* FROM #Posts p

izquierda unirse a #usuarios u en u.Id = p.OwnerId

Ordenar por p.Id ";

datos var = connection.Query <Post, usuario, Post> (sql, (post, usuario) => {post.Owner = usuario; vuelta de correo;});

Porque ofrece menos encapsulación, Dapper requiere que los desarrolladores saben más acerca de cómo se almacenan sus datos, cómo consultar de manera eficiente, y escribir más código a buscarla. Cuando los cambios de modelo, en lugar de la simple creación de una nueva migración (otra característica EF Core), y / o actualización de la información de mapeo en un solo lugar en un DbContext, cada consulta que se impactó deben actualizarse. Estas consultas no han compilar garantías de tiempo, por lo que se pueden romper en tiempo de ejecución en respuesta a cambios en el modelo o base de datos, por lo que los errores más difíciles de detectar con rapidez. A cambio de estas concesiones, Dapper ofrece un rendimiento extremadamente rápido.

Para la mayoría de las aplicaciones, y la mayor parte de casi todas las aplicaciones, EF Core ofrece un rendimiento aceptable. Por lo tanto, sus beneficios de productividad de desarrolladores son probablemente mayores que su sobrecarga de rendimiento. Para las consultas que pueden beneficiarse del almacenamiento en caché, la consulta real sólo se puede ejecutar un pequeño porcentaje de las veces, lo que hace relativamente pequeñas diferencias de rendimiento de consulta discutible.

SQL o NoSQL

Tradicionalmente, las bases de datos relacionales como SQL Server han dominado el mercado para el almacenamiento de datos persistente, pero no son la única solución disponible. bases de datos NoSQL como[MongoDB](https://www.mongodb.com/what-is-mongodb) ofrecer un enfoque diferente para almacenar objetos. En lugar de mapeo de objetos a las tablas y filas, otra opción es para serializar todo el gráfico de objeto, y almacenar el resultado. Los beneficios de este enfoque, al menos inicialmente, son la simplicidad y rendimiento. Es ciertamente más fácil de almacenar un único objeto serializado con una tecla que no sea para descomponer el objeto en muchas mesas con las relaciones y la actualización y filas que pueden haber cambiado desde que el objeto fue recuperado de la base de datos anterior. Del mismo modo, ir a buscar y deserializar un objeto en una tienda basada en clave es típicamente mucho más rápido y más fácil que se une a complejos o múltiples consultas de bases de datos necesarios para componer totalmente el mismo objeto a partir de una base de datos relacional. La falta de cerraduras o transacciones o un esquema fijo también hace que las bases de datos NoSQL muy susceptibles a la ampliación a través de muchas máquinas,

Por otro lado, las bases de datos NoSQL (como se les suele llamar) tienen sus inconvenientes. Bases de datos relacionales utilizan normalización para hacer cumplir la consistencia y evitar la duplicación de datos. Esto reduce el tamaño total de la base de datos y asegura que los cambios a los datos compartidos están disponibles de inmediato en toda la base de datos. En una base de datos relacional, una tabla de direcciones podría hacer referencia a una tabla de País por ID, de manera que si el nombre de un país se han cambiado, los registros de direcciones se beneficiarían de la actualización sin que tienen que ser actualizados. Sin embargo, en una base de datos NoSQL, la dirección y el país asociado podría ser serializado como parte de muchos objetos almacenados. Una actualización de un nombre de país requeriría que todos esos objetos que se actualizan, en lugar de una sola fila. bases de datos relacionales también pueden garantizar la integridad relacional mediante la aplicación de reglas como las claves externas.

Otra bases de datos NoSQL complejidad deben tratar es de versiones. Cuando las propiedades de un objeto cambian, puede que no sea capaz de ser deserializado de las versiones anteriores que estaban almacenados. Por lo tanto, todos los objetos existentes que se debe actualizarse un (anterior) versión serializada del objeto que se ajustan a su nuevo esquema. Esto no es conceptualmente diferente de una base de datos relacional, donde los cambios de esquema

|  |  |
| --- | --- |
| 79 | Capítulo 8 |
|  | Trabajar con datos |

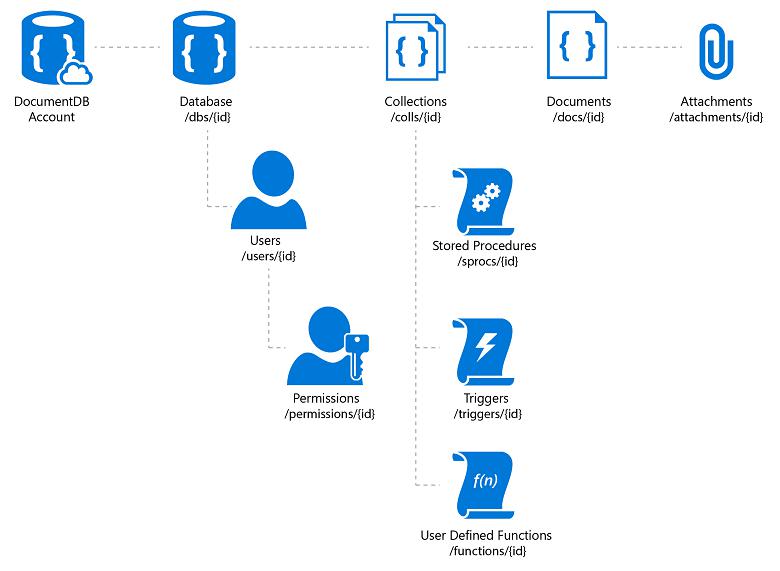
a veces requieren scripts de actualización o actualizaciones de mapas. Sin embargo, el número de entradas que debe ser modificado a menudo es mucho mayor en el enfoque de NoSQL, porque hay más duplicación de datos.

Es posible en bases de datos NoSQL para almacenar varias versiones de objetos, algo fijo esquema de bases de datos relacionales normalmente no son compatibles. Sin embargo, en este caso el código de aplicación tendrá que dar cuenta de la existencia de versiones anteriores de objetos, añadiendo una complejidad adicional.

bases de datos NoSQL normalmente no hacen cumplir [ÁCIDO,](http://en.wikipedia.org/wiki/ACID) que significa que tienen dos ventajas de rendimiento y escalabilidad más de las bases de datos relacionales. Están bien adaptados a muy grandes conjuntos de datos y objetos que no están bien adaptados para el almacenamiento en las estructuras de tablas normalizadas. No hay ninguna razón por la que una sola aplicación no puede tomar ventaja de las dos bases de datos relacionales y NoSQL, utilizando cada uno en el que es el más adecuado.

Azure Cosmos DB

Azure Cosmos DB es el almacenamiento distribuido a nivel mundial, la base de datos multi-modelo. Cosmos DB está construido para una rápida y predecible rendimiento, alta disponibilidad, la escala elástica, y la distribución global. Incluye soporte para múltiples modelos, incluyendo el motor JSON sin esquema DocumentDB. A pesar de ser una base de datos NoSQL, los desarrolladores pueden utilizar las capacidades ricas y familiares consulta SQL en los datos JSON con la API DocumentDB. Todos los recursos en DocumentDB se almacenan como documentos JSON. Los recursos se gestionan como artículos, que son documentos que contienen metadatos, y alimentos, que son colecciones de artículos. La Figura 8-2 muestra la relación entre diferentes recursos DocumentDB.



|  |  |
| --- | --- |
| 80 | Capítulo 8 |
|  | Trabajar con datos |

***Figura 8-2.*** *organización de recursos DocumentDB.*

El lenguaje de consulta DocumentDB es una interfaz sencilla pero potente para la consulta de documentos JSON. El lenguaje soporta un subconjunto de ANSI SQL y la gramática añade integración profunda de objeto JavaScript, matrices, construcción de objetos, y la invocación de la función.

Azure Cosmos DB es compatible con la API de MongoDB, que se puede utilizar con las bibliotecas y herramientas existentes MongoDB. También ofrece una API de la tabla, para el almacenamiento de clave-valor, y una API de gráficos construido siguiendo el Apache[Tinkerpop](http://tinkerpop.apache.org/) especificación.

**Referencias - Azure Cosmos DB**

* Cosmos DB Introducción https://docs.microsoft.com/en-us/azure/cosmos-db/introduction

Otras Opciones de persistencia

Además de las opciones de almacenamiento relacional y NoSQL, aplicaciones ASP.NET núcleo puede utilizar Azure de almacenamiento para almacenar una variedad de formatos y archivos de datos en una, de manera escalable basado en la nube. Azure Storage es masivamente escalable, para que pueda empezar a cabo almacenar pequeñas cantidades de datos y escalar hasta almacenar cientos de terabytes o si la aplicación lo requiere. Azure Storage soporta cuatro tipos de datos:

* Almacenamiento Blob para texto estructurado o almacenamiento binario, también referido como el almacenamiento de objetos.
* Almacenamiento de tablas de bases de datos estructuradas, accesible a través de teclas de la fila.
* Cola de almacenamiento para la mensajería basada en cola fiable.
* Almacenamiento de archivos para acceso a archivos compartidos entre las máquinas virtuales Azure y aplicaciones internas.

**Referencias - Azure Storage**

* Azure Storage Introducción <https://docs.microsoft.com/en-us/azure/storage/storage-introduction>

El almacenamiento en caché

En las aplicaciones web, cada solicitud de la tela debe ser completado en el menor tiempo posible. Una forma de lograrlo es limitar el número de llamadas externas del servidor debe realizar para completar la solicitud. El almacenamiento en caché implica almacenar una copia de los datos en el servidor (o en otro almacén de datos que es más fácil de consultar a la fuente de los datos). aplicaciones web y aplicaciones web especialmente los no tradicionales de spa, necesitan construir la interfaz de usuario con cada petición. Con frecuencia, esto implica hacer muchas de las mismas consultas de bases de datos en varias ocasiones de una solicitud de un usuario a otro. En la mayoría de los casos, estos datos rara vez se cambia, por lo que hay pocas razones para solicitar constantemente desde la base de datos. ASP.NET Core soporta el almacenamiento en caché la respuesta, para almacenar en caché páginas enteras, y los datos de almacenamiento en caché, que apoya el comportamiento de almacenamiento en caché más granular.

Al implementar el almacenamiento en caché, es importante tener en cuenta la separación de las preocupaciones. Evitar la aplicación de la lógica de almacenamiento en caché en su lógica de acceso a datos, o en la interfaz de usuario. En su lugar, encapsular el almacenamiento en caché en sus propias clases, y utilizar la configuración para gestionar su comportamiento. Esto sigue los principios de Responsabilidad abierto / cerrado e individual, y hará que sea más fácil para usted para manejar cómo se utiliza el almacenamiento en caché de la aplicación a medida que crece.

|  |  |
| --- | --- |
| 81 | Capítulo 8 |
|  | Trabajar con datos |

**ASP.NET Core Almacenamiento en caché de respuesta**

ASP.NET Core soporta dos niveles de almacenamiento en caché de respuesta. El primer nivel no almacena en caché nada en el servidor, pero añade las cabeceras HTTP que indican a los clientes y los servidores proxy para las respuestas de antememoria. Esto se implementa mediante la adición de laResponseCache atribuir a controladores o acciones:

[ResponseCache (Duración = 60)]

pública IActionResult Contacto ()

{}

ViewData [ "mensaje"] = "Su página de contacto."; Ver regresar ();

}

El ejemplo anterior dará lugar a la siguiente cabecera de ser añadido a la respuesta, instruir a los clientes para almacenar en caché el resultado para un máximo de 60 segundos.

Cache-Control:, max-age = 60 pública

Con el fin de añadir el lado del servidor de almacenamiento en caché en memoria a la aplicación, se debe hacer referencia al Microsoft.AspNetCore.ResponseCaching NuGet paquete y, a continuación, añadir el almacenamiento en caché de respuesta middleware. Este middleware se configura en tantoConfigureServices y configurar en el inicio:

ConfigureServices public void (servicios) IServiceCollection

{

services.AddResponseCaching ();

}

public void Configurar (IApplicationBuilder aplicación)

{

app.UseResponseCaching ();

}

El almacenamiento en caché de respuesta Middleware en caché automático de las respuestas en base a un conjunto de condiciones, que se pueden personalizar. Por defecto, sólo 200 (OK) respuestas deseados a través de métodos GET o la cabeza se almacenan en caché. Además, las solicitudes deben tener una respuesta con un Cache-Control: cabecera pública, y no pueden incluir encabezados de autorización o Set-Cookie. Ver un[Lista completa de las condiciones de almacenamiento en caché](https://docs.microsoft.com/en-us/aspnet/core/performance/caching/middleware#conditions-for-caching) [utilizado por el middleware almacenamiento en caché la respuesta.](https://docs.microsoft.com/en-us/aspnet/core/performance/caching/middleware#conditions-for-caching)

**El almacenamiento en caché de datos**

En lugar de (o además de) el almacenamiento en caché web respuestas completas, puede almacenar en caché los resultados de las consultas de datos individuales. Para esto, se puede utilizar en el almacenamiento en caché de memoria en el servidor web, o utilizar[una memoria caché distribuida.](https://docs.microsoft.com/en-us/aspnet/core/performance/caching/distributed) En esta sección se demostrará cómo implementar el almacenamiento en caché en memoria.

Agrega soporte para memoria (o distribuido) el almacenamiento en caché en ConfigureServices:

|  |  |
| --- | --- |
|  | ConfigureServices public void (servicios) IServiceCollection |
|  | { |
|  | services.AddMemoryCache (); |
| 82 | Capítulo 8 |
|  | Trabajar con datos |

services.AddMvc ();

}

Asegúrese de añadir el Microsoft.Extensions.Caching.Memory NuGet paquete también.

Una vez que haya añadido el servicio, se solicita IMemoryCachea través de la inyección de dependencias siempre que se necesite acceder a la caché. En este ejemplo, laCachedCatalogService está utilizando el patrón de diseño Proxy (o decorador), proporcionando una implementación alternativa de ICatalogService que controla el acceso a (o se suma a la conducta) el subyacente CatalogService implementación.

CachedCatalogService clase pública: {ICatalogService

\_cache IMemoryCache sólo lectura privada;

sólo lectura privada CatalogService \_catalogService; \_brandsKey privada estática de sólo lectura String = "marcas"; privado cadena de sólo lectura estática \_typesKey = "tipos";

privado cadena de sólo lectura estática \_itemsKeyTemplate = "artículos- {0} - {1} - {2} - {3}";

privadas estáticas de sólo lectura TimeSpan \_defaultCacheDuration = TimeSpan.FromSeconds (30);

CachedCatalogService pública (caché IMemoryCache, CatalogService CatalogService)

{

\_cache = caché;

\_catalogService = CatalogService;

}

pública asíncrono de tareas <IEnumerable <SelectListItem >> GetBrands ()

{

esperan el retorno \_cache.GetOrCreateAsync (\_brandsKey, entrada asíncrono

=>

{

entry.SlidingExpiration = \_defaultCacheDuration; retorno esperan \_catalogService.GetBrands ();

});

}

pública asíncrono de tareas <Catálogo> GetCatalogItems (int pageIndex, int itemsPage, int? brandID, int? TypeId)

{

cadena cacheKey = String.Format (\_itemsKeyTemplate, pageIndex, itemsPage, brandID, TypeId);

esperan el retorno \_cache.GetOrCreateAsync (cacheKey, entrada asíncrono

|  |  |
| --- | --- |
|  | => |
|  | { |
| 83 | Capítulo 8 |
|  | Trabajar con datos |

entry.SlidingExpiration = \_defaultCacheDuration;

retorno esperan \_catalogService.GetCatalogItems (pageIndex, itemsPage, brandID, TypeId);

});

}

pública asíncrono de tareas <IEnumerable <SelectListItem >> getTypes ()

{

esperan el retorno \_cache.GetOrCreateAsync (\_typesKey, entrada asíncrono

=>

{

entry.SlidingExpiration = \_defaultCacheDuration; retorno esperan \_catalogService.GetTypes ();

});

}

}

Para configurar la aplicación para utilizar la versión en caché del servicio, pero todavía permiten el servicio para obtener la instancia de CatalogService que necesita en su constructor, debe agregar lo siguiente en

ConfigureServices:

services.AddMemoryCache ();

services.AddScoped <ICatalogService, CachedCatalogService> ();

services.AddScoped <CatalogService> ();

Con esto en su lugar, la base de datos llama a buscar los datos del catálogo sólo se realizarán una vez por minuto, en lugar de en cada petición. Dependiendo del tráfico al sitio, esto puede tener un impacto muy significativo en el número de consultas realizadas a la base de datos, y el tiempo promedio de carga de página de la página de inicio que actualmente depende de los tres de las consultas expuestos por este servicio.

Un problema que surge cuando se implementa el almacenamiento en caché es rancio de datos - es decir, datos que han cambiado en la fuente, sino una versión desactualizada permanece en la memoria caché. Una forma sencilla para mitigar este problema es utilizar pequeñas duraciones de caché, ya que para una aplicación ocupado allí está limitado beneficio adicional para extender la longitud de datos se almacena en caché. Por ejemplo, considere una página que hace que una sola consulta de base de datos y se solicita 10 veces por segundo. Si esta página se almacena en caché durante un minuto, el resultado será el número de consultas de bases de datos hechas por minuto a caer de 600 a 1, una reducción del 99,8%. Si por el contrario la duración de la caché se hicieron de una hora, la reducción global sería 99,997%, pero ahora la probabilidad y la edad potencial de los datos antiguos son a la vez aumentado de manera espectacular.

Otro enfoque consiste en eliminar de forma proactiva las entradas de caché cuando los datos que contienen se actualiza. Cualquier entrada individual puede ser eliminado si su clave es conocida:

\_cache.Remove (cacheKey);

Si su aplicación contiene las funciones para la actualización de las entradas que se almacena en caché, puede eliminar las entradas de caché correspondientes en el código que realiza las actualizaciones. A veces puede haber muchas entradas diferentes que dependen de un conjunto particular de datos. En ese caso, puede ser útil para crear

|  |  |
| --- | --- |
| 84 | Capítulo 8 |
|  | Trabajar con datos |

dependencias entre las entradas de caché, mediante el uso de una CancellationChangeToken. Con unCancellationChangeToken, Puede expirar múltiples entradas de caché a la vez mediante la cancelación de la simbólico.

* CancellationToken configurar y añadir la entrada a cts var cache = nueva CancellationTokenSource (); \_cache.Set ( “cts”, CTS);

\_cache.Set (cacheKey, itemToCache,

nuevo CancellationChangeToken (cts.Token));

* en otros lugares, expirará la caché mediante la cancelación de la señal de \_cache.Get <CancellationTokenSource> ( “CTS”) Cancelar ().;

|  |  |
| --- | --- |
| 85 | Capítulo 8 |
|  | Trabajar con datos |

El almacenamiento en caché puede mejorar drásticamente el rendimiento de páginas web que repiten la solicitud de los mismos valores de su base de datos. Asegúrese de medir el acceso a datos y rendimiento de la página antes de aplicar el almacenamiento en caché, y el almacenamiento en caché sólo se aplican donde se ve una necesidad de mejora. El almacenamiento en caché consume recursos de memoria del servidor web y aumenta la complejidad de la aplicación, por lo que es importante que no prematuramente optimizar el uso de esta técnica.

|  |  |
| --- | --- |
| 86 | Capítulo 8 |
|  | Trabajar con datos |

SECCIÓN 9

Pruebas de ASP.NET MVC Core Aplicaciones

*“Si no lo hace como unidad de probar sus productos, más probable es que sus clientes no les gusta*

*probarlo, ya sea “.*

Anónimo

Resumen

Software de cualquier complejidad puede fallar de forma inesperada en respuesta a los cambios. Por lo tanto, se requiere una prueba después de realizar cambios para todos, pero las aplicaciones más triviales (o menos críticos). Las pruebas manuales es la manera más lenta, menos fiable, más caro probar el software. Por desgracia, si las aplicaciones no están diseñadas para ser comprobable, puede ser el único medio disponible. Las aplicaciones escritas siguiendo los principios arquitectónicos establecidos en el capítulo X debe ser probado con pruebas unitarias, y las aplicaciones ASP.NET núcleo de soporte integración automatizada y pruebas funcionales también.

Tipos de pruebas automatizadas

Hay muchos tipos de pruebas automatizadas para aplicaciones de software. El, prueba de nivel más bajo más simple es la prueba de la unidad. A un nivel ligeramente más alto hay pruebas de integración y pruebas funcionales. Otros tipos de pruebas, como las pruebas de interfaz de usuario, pruebas de carga, pruebas de esfuerzo, y pruebas de humo, están más allá del alcance de este documento.

**Las pruebas unitarias**

Una prueba de la unidad a prueba una sola parte de la lógica de la aplicación. Uno puede describir con más detalle por enumerar algunas de las cosas que no lo es. Una unidad de prueba no prueba cómo funciona su código con dependencias o infraestructuras - eso es lo que las pruebas de integración son para. Una unidad de prueba no prueba el marco de su código está escrito en - que debe asumir funciona o, si lo encuentra no, presente un error y el código de una solución. Una unidad de prueba se ejecuta por completo en la memoria y en el proceso. No se comunica con el sistema de archivos, la red, o una base de datos. Las pruebas unitarias sólo debe probar el código.

Las pruebas unitarias prueba sólo una única unidad de su código, sin dependencias externas, y deben ejecutar de forma extremadamente rápida. Por lo tanto, usted debe ser capaz de ejecutar conjuntos de pruebas de cientos de pruebas de unidad en unos pocos segundos. Ejecutarlos con frecuencia, idealmente antes de cada empujar a un repositorio de control de código fuente compartido, y ciertamente con cada construcción automatizado en su servidor de compilación.

87 Capítulo 9

Si estás métodos que se basan en otros servicios de pruebas, lo ideal es definido como interfaces y se inyecta como constructor o método argumentos, es probable que utilice implementaciones falsas o simuladas de estas interfaces en su unidad de prueba. Recuerde que su objetivo es sólo para probar el código en una sola unidad (lo ideal es un método único) de su aplicación, no codificar esta unidad referencias.

**Las pruebas de integración**

Aunque es una buena idea para encapsular el código que interactúa con la infraestructura, como bases de datos y sistemas de archivos, de todas maneras tendrá un poco de ese código, y es probable que desee para probarlo. Además, se debe verificar que las capas de su código interactúan como se puede esperar cuando las dependencias de su aplicación están completamente resueltos. Esta es la responsabilidad de las pruebas de integración. Pruebas de integración tienden a ser más lento y más difícil de instalar que las pruebas unitarias, ya que a menudo dependen de las dependencias externas e infraestructura. Por lo tanto, se debe evitar cosas de pruebas que podrían ser probadas con pruebas unitarias en las pruebas de integración. Si se puede probar un escenario dado con una prueba de unidad, debe probarlo con una prueba de unidad. Si no puede, entonces considerar el uso de una prueba de integración.

Debido a que el propósito de las pruebas de integración es verificar que los múltiples piezas trabajan juntos, incluyendo el código de la infraestructura de bienes, podrás rara vez desee utilizar implementaciones falsas o simuladas en las pruebas de integración. Si necesita implementaciones falsas o simuladas, asegúrese de que usted no está accidentalmente probar estas implementaciones, en lugar del comportamiento real que está tratando de probar.

Pruebas de integración a menudo tienen procedimientos de montaje y desmontaje más complejas que las pruebas unitarias. Por ejemplo, una prueba de integración que va en contra de una base de datos real se necesita una manera de devolver la base de datos a un estado conocido antes de cada ejecución de prueba. A medida que se añaden nuevas pruebas, y el esquema de base de datos de producción se desarrolla, estos scripts de prueba tienden a crecer en tamaño y complejidad. En muchos sistemas grandes, no es práctico para funcionar suites completas de pruebas de integración en estaciones de trabajo de desarrollador antes de registrarse cambios en control de código fuente compartido. En estos casos, las pruebas de integración se pueden ejecutar en un servidor de compilación.

La muestra incluye un eShopOnWeb OrderRepository que se encarga de ir a buscar y salvar

Datos de los pedidos. Se define un método GetById. Una prueba de ejemplo de integración para este método se muestra a continuación:

GetById clase pública

{

sólo lectura privada CatalogContext \_catalogContext; sólo lectura privada OrderRepository \_orderRepository;

OrderBuilder OrderBuilder privada {get; } = New OrderBuilder (); \_output ITestOutputHelper sólo lectura privada; GetById pública (salida ITestOutputHelper)

{

\_output = salida;

dbOptions var = new

DbContextOptionsBuilder <CatalogContext> ()

.UseInMemoryDatabase (de databaseName: "TestCatalog")

.OPTIONS;

\_catalogContext = new CatalogContext (dbOptions);

\_orderRepository = new OrderRepository (\_catalogContext);

|  |  |
| --- | --- |
|  | } |
| 88 | Capítulo 9 |
|  | Pruebas |

[Hecho]

GetsExistingOrder public void ()

{

var existingOrder = OrderBuilder.WithDefaultValues ​​(); \_catalogContext.Orders.Add (existingOrder); \_catalogContext.SaveChanges (); int orderId = existingOrder.Id;

\_output.WriteLine ($ "OrderId: {} orderId");

var orderFromRepo = \_orderRepository.GetById (orderId); Assert.Equal (OrderBuilder.TestBuyerId,

orderFromRepo.BuyerId);

* Nota: El uso de base de datos en memoria de pedido está disponible. Será nulo si se utiliza SQL DB.

var firstItem = orderFromRepo.OrderItems.FirstOrDefault (); Assert.Equal (OrderBuilder.TestUnits, firstItem.Units);

}

}

Cabe destacar que en la prueba anterior, una base de datos en memoria se utiliza. Entidad Marco de base es compatible con el uso de bases de datos en memoria para escenarios de pruebas de integración en la que son útiles debido a su rápida velocidad de ejecución y facilidad de instalación. No se requiere ninguna base de datos externa para ejecutar estas pruebas de integración. En el método de ensayo, el tipo CatalogContext se utiliza para configurar la base de datos con un buen orden existente conocido. A continuación, el tipo de implementación del repositorio es usado para obtener una orden por su id. Por último, la prueba afirma que el tipo de orden que devuelve el depósito era el esperado que se añadió.

Una nota de precaución al realizar pruebas con base de datos en memoria, sin embargo, es que no se validará que la carga ansiosa está funcionando correctamente. propiedades de navegación se creará una instancia cuando se utiliza base de datos en memoria, pero pueden ser nula cuando el mismo código se ejecuta en una base de datos relacional. Para probar este escenario, es necesario utilizar un verdadero ejemplo de base de datos relacional en sus pruebas de integración.

**Las pruebas funcionales**

Pruebas de integración están escritos desde la perspectiva del desarrollador, para verificar que algunos componentes del sistema funcione correctamente juntos. Las pruebas funcionales están escritos desde la perspectiva del usuario, y verificar la exactitud del sistema en función de sus necesidades. El siguiente extracto ofrece una analogía útil para la forma de pensar acerca de las pruebas de funcionamiento, en comparación con las pruebas de unidad:

*“Muchas veces el desarrollo de un sistema se asemeja a la construcción de una casa. Mientras que esta analogía no es del todo correcto, podemos extenderlo a los efectos de entender la diferencia entre la unidad y pruebas funcionales. Prueba de la unidad es análogo a un inspector de edificios visitando el sitio de la construcción de una casa. Él se centra en los diversos sistemas internos de la casa, la base, el encuadre, electricidad, fontanería, y así sucesivamente. Él asegura (pruebas) de que las partes de la casa funcionará correctamente y de forma segura, es decir, cumplir con el código de construcción. Las pruebas funcionales en este escenario son análogos a los dueños de casa visitando este mismo emplazamiento de la obra. Se supone que los sistemas internos se comportarán adecuadamente, que el inspector de construcción está llevando a cabo su tarea. El dueño de casa es*

|  |  |
| --- | --- |
| 89 | Capítulo 9 |
|  | Pruebas |

*enfocado en lo que será como vivir en esta casa. Él se refiere a cómo la casa se ve, son las distintas habitaciones de un tamaño cómodo, no la casa adaptarse a las necesidades de la familia, son las ventanas en un buen lugar para tomar el sol de la mañana. El dueño de casa está realizando pruebas de funcionamiento de la casa. Él tiene la perspectiva del usuario. El inspector de construcción está realizando pruebas de unidad en la casa. Él tiene la perspectiva de que el constructor “.*

*Fuente:* [*Prueba de la unidad frente a las pruebas funcionales*](http://www.softwaretestingtricks.com/2007/01/unit-testing-versus-functional-tests.html)

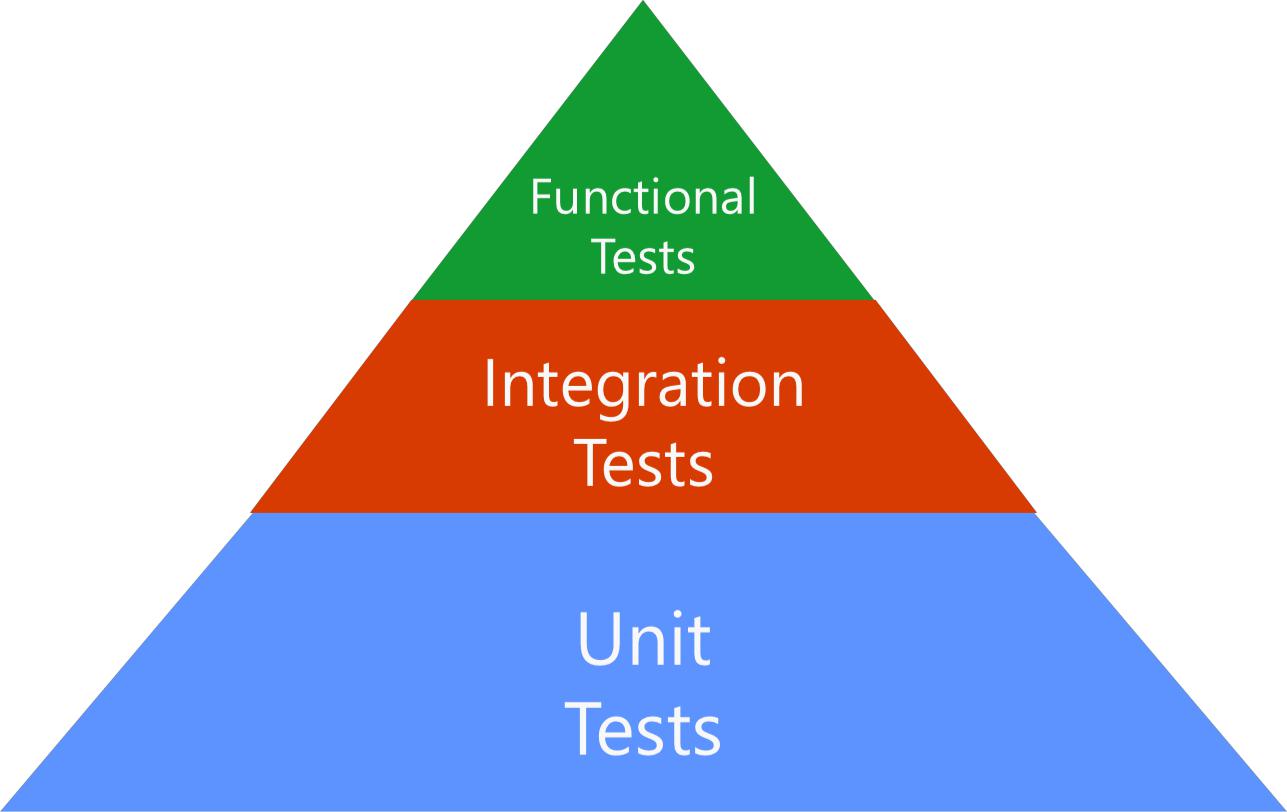
Soy aficionado a decir “Como desarrolladores, no somos capaces de dos maneras:. Construimos las cosas mal, o construimos las cosas mal” Las pruebas unitarias aseguramos que usted está construyendo lo correcto; pruebas funcionales aseguran que está construyendo lo correcto.

Dado que las pruebas funcionales operan a nivel del sistema, que pueden requerir algún grado de automatización de la interfaz de usuario. Al igual que las pruebas de integración, por lo general trabajan con infraestructura de pruebas también. Esto los hace más lento y más frágil que las pruebas unitarias y de integración. Usted debe tener sólo como muchas pruebas funcionales como sea necesario para estar seguro de que el sistema se comporta como los usuarios esperan.

En las aplicaciones ASP.NET Core, pruebas unitarias y de integración no suelen ejercer la pila completa MVC, y por lo tanto no se pueden probar las características como el enrutamiento, enlace de modelos, validación de modelos, etc. Las pruebas funcionales se pueden utilizar para verificar que estas características son marco de interfaz de usuario implementado correctamente. Un ejemplo de este tipo de ensayo se muestra al final de este capítulo.

**Pirámide de pruebas**

Martin Fowler escribió sobre la pirámide de prueba, un ejemplo del cual se muestra en la Figura 9-1.



|  |  |
| --- | --- |
|  | ***La Figura 9-1.*** *La pirámide de Pruebas* |
| 90 | Capítulo 9 |
|  | Pruebas |

Las diferentes capas de la pirámide, y sus tamaños relativos, representan diferentes tipos de pruebas y cuántas se deben escribir para su aplicación. Como se puede ver, la recomendación es tener una gran base de pruebas unitarias, apoyados por una capa pequeña de las pruebas de integración, con una capa aún más pequeño de pruebas funcionales. Cada capa debe idealmente sólo tienen pruebas en los mismos que no pueden ser ejecutadas adecuadamente a una capa inferior. Mantener la pirámide prueba en mente cuando usted está tratando de decidir qué tipo de prueba que necesita para un escenario particular.

**Lo que prueba**

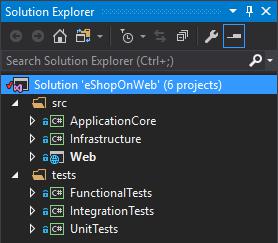
Un problema común para los desarrolladores que no tienen experiencia con pruebas de escritura automática es dar con lo que para probar. Un buen punto de partida es poner a prueba la lógica condicional. Cualquier lugar que tenga un método con el comportamiento que cambia según una sentencia condicional (if-else, switch, etc.), debe ser capaz de llegar al menos un par de pruebas que confirman el comportamiento correcto para ciertas condiciones. Si el código tiene condiciones de error, que es bueno para escribir al menos una prueba para el “camino feliz” a través del código (sin errores), y al menos una prueba para el “camino triste” (con errores o resultados atípicos) para confirmar su aplicación se comporta como se esperaba en la cara de los errores. Por último, trate de concentrarse en las cosas que pueden fallar prueba, en lugar de centrarse en las métricas como la cobertura de código. Más cobertura de código es mejor que menos, por lo general. Sin embargo,

La organización de Proyectos de Prueba

proyectos de prueba se pueden organizar sin embargo funciona mejor para usted. Es una buena idea para pruebas separadas por tipo (unidad de prueba, ensayo de la integración) y por lo que están poniendo a prueba (por proyecto, por espacio de nombres). Si esta separación consiste en carpetas dentro de un proyecto de una sola prueba, o múltiples proyectos de prueba, es una decisión de diseño. Uno de los proyectos es más simple, pero para grandes proyectos con muchas pruebas, o para ejecutar más fácilmente diferentes conjuntos de pruebas, es posible que desee tener varios proyectos de prueba diferentes. Muchos equipos organizan proyectos de prueba basados ​​en el proyecto que están poniendo a prueba, que para aplicaciones con más de unos pocos proyectos pueden dar lugar a muchos proyectos de prueba, especialmente si todavía no cumple estas en función de qué tipo de pruebas se encuentran en cada proyecto. Una solución de compromiso es tener un proyecto por tipo de prueba, por aplicación,

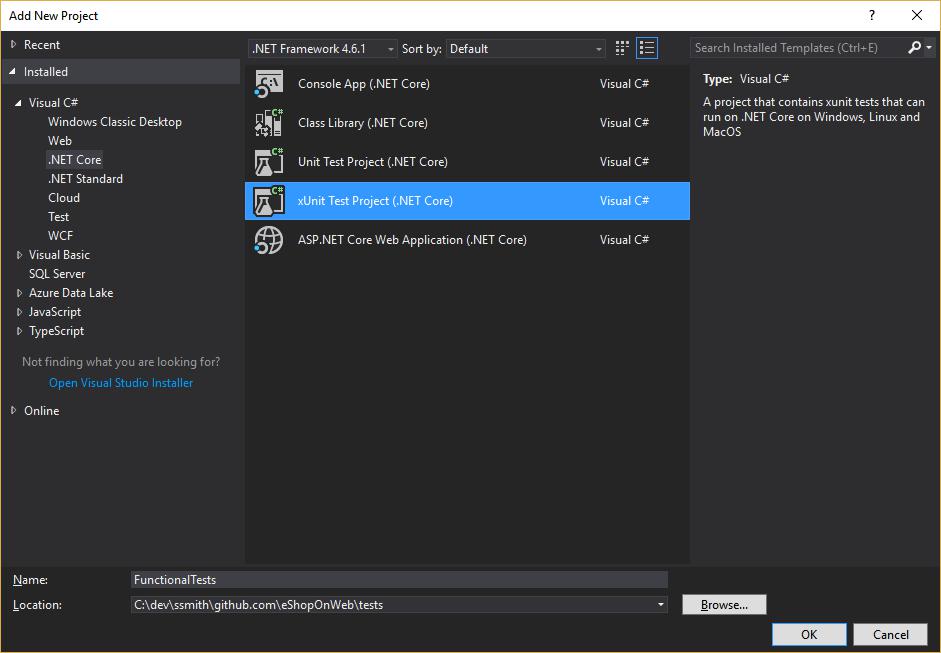
Un enfoque común es organizar los proyectos de aplicación en una carpeta 'src', y proyectos de prueba de la aplicación en una carpeta paralela 'pruebas'. Se pueden crear carpetas solución adecuada en Visual Studio, si usted encuentra esta organización útil.

|  |  |
| --- | --- |
| 91 | Capítulo 9 |
|  | Pruebas |



***Figura 9-2.*** *organización de pruebas en su solución*

Que puede elegirse el marco de pruebas así lo prefiere. El marco xUnit funciona bien y es lo que todas las pruebas Core Core ASP.NET y EF están escritos en. Se puede añadir un proyecto de prueba xUnit en Visual Studio utilizando la plantilla se muestra en la figura 9-X, o desde la CLI usandodotnet nueva xUnit.



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

|  |  |
| --- | --- |
| 92 | Capítulo 9 |
|  | Pruebas |

**Naming Test**

Debieras [asignar nombres a las pruebas de una manera coherente,](https://ardalis.com/unit-test-naming-convention) con nombres que indican lo que cada prueba. Uno de los enfoques que he tenido un gran éxito con es nombrar las clases de prueba de acuerdo con la clase y el método que se están probando. Esto da lugar a muchas clases de prueba pequeños, pero hace que sea muy claro lo que cada prueba es responsable. Con el nombre de la clase de prueba establecido para identificar la clase y el método a ensayar, el nombre del método de ensayo se puede utilizar para especificar el comportamiento que se está probando. Esto debe incluir el comportamiento esperado y cualquier entrada o suposiciones que deben producir este comportamiento. Algunos nombres de ejemplo de prueba:

* CatalogControllerGetImage.CallsImageServiceWithId
* CatalogControllerGetImage.LogsWarningGivenImageMissingException
* CatalogControllerGetImage.ReturnsFileResultWithBytesGivenSuccess
* CatalogControllerGetImage.ReturnsNotFoundResultGivenImageMissingException

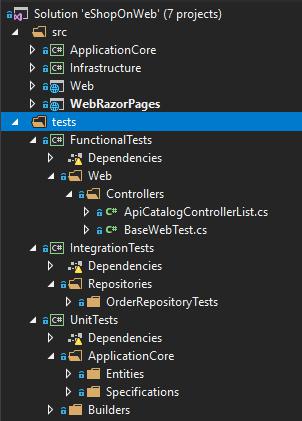
Una variación de este enfoque finaliza cada nombre de la clase de prueba con “debería” y modifica la tensa ligeramente:

* CatalogControllerGetImageShould.CallImageServiceWithId
* CatalogControllerGetImageShould.LogWarningGivenImageMissingException

Algunos equipos de encontrar el segundo enfoque de nombres más clara, aunque un poco más detallado. En cualquier caso, tratar de utilizar una convención de nombres que proporciona información sobre el comportamiento de prueba, de modo que cuando una o más pruebas fallan, es obvio a partir de sus nombres lo casos han fracasado. Evitar nombrar sus pruebas vagamente, comoControllerTests.Test1, Ya que estos ofrecen ningún valor cuando se les ve en los resultados de las pruebas.

Si usted sigue una convención de nomenclatura similar a la anterior que produce muchas pequeñas clases de prueba, que es una buena idea para organizar mejor sus pruebas con las carpetas y los espacios de nombres. La figura 9-4 muestra un enfoque para la organización de pruebas de carpeta dentro de varios proyectos de prueba.

|  |  |
| --- | --- |
| 93 | Capítulo 9 |
|  | Pruebas |



***Figura 9-4.*** *La organización de las clases de prueba de carpeta basada en la clase siendo probados.*

Por supuesto, si una clase de aplicación tiene muchos métodos que se está probando (y por lo tanto muchas clases de prueba), puede tener sentido para colocar estos en una carpeta correspondiente a la clase de aplicación. Esta organización no es diferente de cómo se puede organizar los archivos en carpetas en otro lugar. Si usted tiene más de tres o cuatro archivos relacionados en una carpeta que contiene muchos otros archivos, a menudo es útil para mover a su propia subcarpeta.

Unidad de prueba ASP.NET Aplicaciones Core

En una aplicación ASP.NET Core bien diseñado, la mayor parte de la lógica de la complejidad y de negocios se encapsula en entidades comerciales y una variedad de servicios. El ASP.NET MVC Core aplicación en sí, con sus controladores, filtros, ViewModels, y puntos de vista, debería requerir muy pocas pruebas unitarias. Gran parte de la funcionalidad de un método de acción determinada o manipulador se encuentra fuera del método en sí. Probando si el enrutamiento funciona correctamente o el tratamiento de errores global, no puede hacerse efectiva con una prueba de unidad. Del mismo modo, los filtros, incluyendo validación de modelos y filtros de autenticación y autorización, no pueden ser probados unidad. Sin estas fuentes de comportamiento, la mayoría de los manipuladores y los métodos de acción deben ser trivialmente pequeña, delegando la mayor parte de su trabajo a los servicios que pueden ser probados independientemente de los métodos que se utilicen.

A veces tendrás que refactorizar el código para unidad de prueba él. Con frecuencia, esto implica la identificación de las abstracciones y el uso de la inyección de dependencias para acceder a la abstracción en el código que desea probar, en lugar de codificar directamente en contra de la infraestructura. Por ejemplo, considere este método simple acción para la visualización de imágenes:

|  |  |
| --- | --- |
| 94 | Capítulo 9 |
|  | Pruebas |

[HttpGet ( "[controlador] / pic / {id}")]

pública IActionResult GetImage (int id)

{

var contentRoot = \_env.ContentRootPath + "// Pics"; var path = Path.Combine (contentRoot, ID + ".png"); Byte [] b = System.IO.File.ReadAllBytes (ruta de acceso); Archivo (b, "image / png") return;

}

Unidad de prueba de este método se hace difícil por su dependencia directa de System.IO.File, Que se utiliza para leer desde el sistema de archivos. Se puede comprobar este comportamiento para asegurarse de que funciona como se esperaba, pero al hacerlo con los archivos reales es una prueba de integración. Vale la pena señalar que no se puede probar la ruta de este método - verá cómo hacer esto con una prueba de funcionamiento en breve.

Si no puede unidad de probar el comportamiento del sistema de archivos directamente, y no se puede probar la ruta, ¿qué hay que probar? Bueno, después de refactorización para hacer posible la unidad de pruebas, es posible descubrir algunos casos de prueba y el comportamiento que falta, tales como el tratamiento de errores. ¿Qué hace el método cuando no se encuentra un archivo? ¿Qué debe hacer? En este ejemplo, el método refactorizado se ve así:

[HttpGet ( "[controlador] / pic / {id}")]

pública IActionResult GetImage (int id)

{

byte [] imageBytes;

tratar

{

imageBytes = \_imageService.GetImageBytesById (id);

}

captura (CatalogImageMissingException ex)

{

\_logger.LogWarning ($ "Sin imagen encontrado para id: {id}"); NotFound retorno ();

}

Archivo de retorno (imageBytes, "image / png");

}

Los \_registrador y \_imageServiceson a la vez inyectado como dependencias. Ahora puede probar que el mismo identificador que se pasa al método de acción se pasa a la \_imageServiceY que los bytes resultantes se devuelven como parte de la FileResult. También puede probar que el registro de errores está sucediendo como se esperaba, y que un resultado NotFound se devuelve si la imagen no está, suponiendo que esto es importante comportamiento de la aplicación (es decir, no sólo temporal código del desarrollador agrega a diagnosticar un problema). La lógica real del archivo se ha trasladado a un servicio de aplicación independiente y ha sido aumentada para devolver una excepción específica de la aplicación para el caso de un archivo que falta. Puede probar esta aplicación de forma independiente, utilizando una prueba de integración.

|  |  |
| --- | --- |
| 95 | Capítulo 9 |
|  | Pruebas |

Pruebas de integración ASP.NET Aplicaciones Core

La mayor parte de las pruebas de integración en sus aplicaciones ASP.NET núcleo debe ser servicios y otros tipos de implantación definidos en el proyecto de infraestructura de pruebas. La mejor manera de probar que su proyecto ASP.NET MVC Core está comportando correctamente es con las pruebas funcionales que se ejecutan en su aplicación que se ejecuta en una serie de pruebas. Un ejemplo de una prueba de integración de una clase de acceso a datos se muestra arriba.

Pruebas funcionales ASP.NET Aplicaciones Core

Para aplicaciones ASP.NET Core, la TestServerclase hace pruebas funcionales bastante fácil de escribir. Se configura unaTestServer usando un WebHostBuilder, Tal como lo hace normalmente para su aplicación. EstaWebHostBuilder debería configurarse como verdadero anfitrión de la aplicación, pero se puede modificar cualquier aspectos que hacen más fácil la prueba. La mayoría de las veces, se va a reutilizar la mismaTestServer para muchos casos de prueba, por lo que puede encapsular en un método reutilizable (tal vez en una clase base):

BaseWebTest clase abstracta pública

{

protegida de sólo lectura HttpClient \_client; protegida \_contentRoot cadena;

BaseWebTest pública ()

{

\_client = GetClient ();

}

protegida HttpClient GetClient ()

{

var startupAssembly = typeof (Inicio) .GetTypeInfo () Asamblea.; \_contentRoot = GetProjectPath ( "src", startupAssembly); var constructor = new WebHostBuilder ()

|  |  |
| --- | --- |
|  | .UseContentRoot (\_contentRoot) |
|  | .UseEnvironment ( "Prueba") |
|  | .UseStartup <inicio> (); |
|  | servidor var = new TestServer (builder); |
|  | // datos de semillas |
|  | usando (alcance var = server.Host.Services.CreateScope ()) |
| 96 | Capítulo 9 |
|  | Pruebas |

{

servicios var = scope.ServiceProvider;

var loggerFactory =

services.GetRequiredService <ILoggerFactory> ();

var catalogContext =

services.GetRequiredService <CatalogContext> ();

CatalogContextSeed.SeedAsync (catalogContext, loggerFactory) .Wait ();

var UserManager =

services.GetRequiredService <UserManager <ApplicationUser >> ();

AppIdentityDbContextSeed.SeedAsync (UserManager) .Wait ();

}

server.CreateClient retorno ();

}

}

los GetProjectPath método simplemente devuelve la ruta física al proyecto web [(muestra de descarga](https://github.com/dotnet-architecture/eShopOnWeb) [solución)](https://github.com/dotnet-architecture/eShopOnWeb). los WebHostBuilder en este caso se limita a señalar que la raíz de contenido para la web aplicación es, y hace referencia a la misma clase de inicio utiliza la aplicación web real. Para trabajar con elTestServer, Se utiliza la norma System.Net.HttpClient escribir para hacer peticiones a la misma. TestServer expone un método createClient útil que proporciona un cliente preconfigurado que está listo para hacer peticiones a la aplicación que se ejecuta en el TestServer. Se utiliza este cliente (fijado al miembro \_client protegida en la prueba anterior base) al escribir pruebas funcionales para su aplicación ASP.NET Core:

ApiCatalogControllerList clase pública: BaseWebTest

{

[Hecho]

pública asíncrono ReturnsFirst10CatalogItems de tareas ()

{

respuesta var = esperar \_client.GetAsync ( "/ api / catálogo / lista de"); response.EnsureSuccessStatusCode (); = Var stringResponse esperan

response.Content.ReadAsStringAsync ();

modelo var =

JsonConvert.DeserializeObject <CatalogIndexViewModel> (stringResponse)

;

|  |  |
| --- | --- |
|  | Assert.Equal (10, model.CatalogItems.Count ()); |
|  | } |
| 97 | Capítulo 9 |
|  | Pruebas |

[Hecho]

asíncrono pública de tareas ReturnsLast2CatalogItemsGivenPageIndex1 ()

{

respuesta var = esperar

\_client.GetAsync ( "? / api / catálogo / lista de páginas = 1");

response.EnsureSuccessStatusCode ();

= Var stringResponse esperan

response.Content.ReadAsStringAsync ();

modelo var =

JsonConvert.DeserializeObject <CatalogIndexViewModel> (stringResponse)

;

Assert.Equal (2, model.CatalogItems.Count ());

}

}

Esta prueba funcional ejerce la pila completa aplicación ASP.NET MVC Core, incluyendo todos

middleware, filtros, aglutinantes, etc., que puede ser en su lugar. Se verifica que una determinada ruta

( “/ Api / catálogo / lista”) devuelve el resultado de datos con formato JSON esperado. Lo hace sin

la creación de un servidor web real, y por lo tanto evita la fragilidad de que el uso de una web de bienes

servidor para la prueba puede experimentar (por ejemplo, problemas con la configuración del firewall). Funcional

pruebas que se ejecutan en contra TestServer son por lo general más lento que las pruebas de integración y unidad, pero

son mucho más rápidas que las pruebas que se ejecutan en la red a un servidor web de prueba.

|  |  |
| --- | --- |
| 98 | Capítulo 9 |
|  | Pruebas |

SECCIÓN 10

proceso de desarrollo de aplicaciones ASP.NET Core alojados en Azure

*“Con la nube, los individuos y las pequeñas empresas pueden chasquear los dedos y*

*al instante establecer servicios de clase empresarial “.*

Roy Stephan

Visión

*Desarrollar aplicaciones ASP .NET Core bien diseñados a su gusto, utilizando la CLI dotnet y código de Visual Studio o su editor de elección Visual Studio o.*

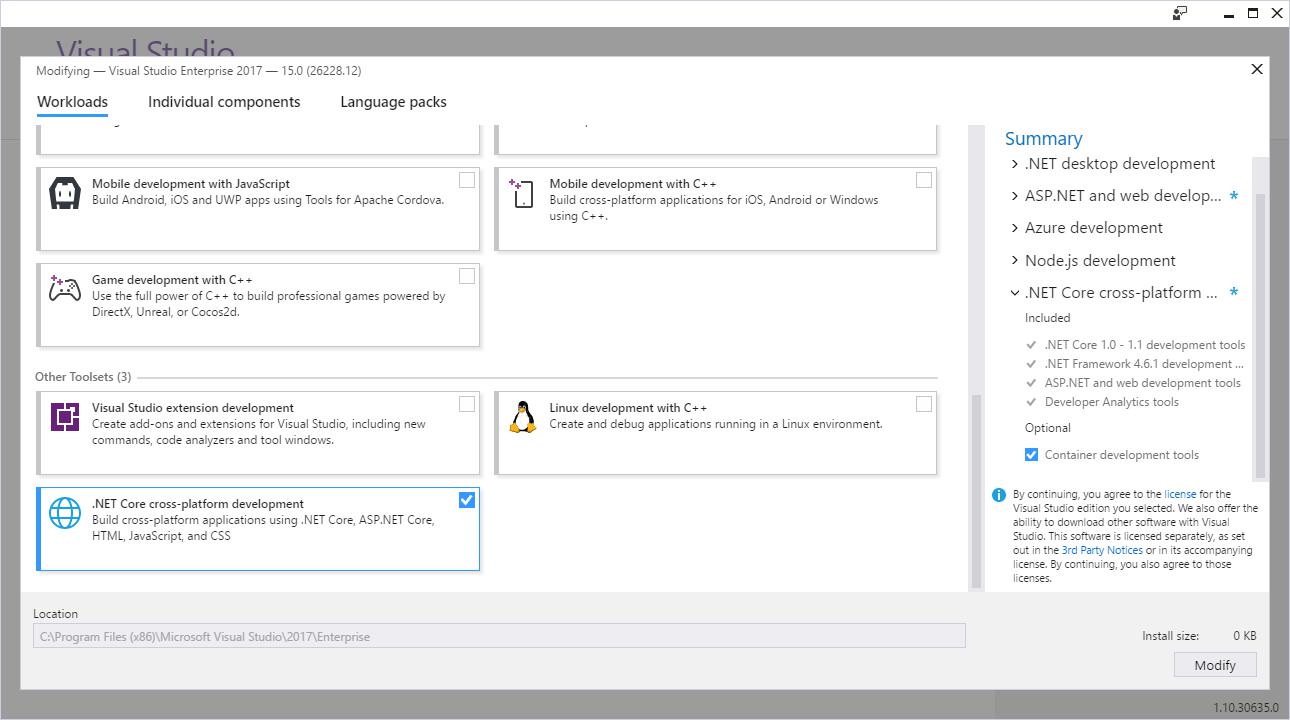
Entorno de desarrollo para aplicaciones ASP.NET Core

**Herramientas de desarrollo de opciones: IDE o editor**

Si usted prefiere un IDE completo y potente o un editor ligero y ágil, Microsoft tiene cubierto al desarrollar aplicaciones ASP.NET Core.

**Visual Studio 2017.** Si está utilizando Visual Studio 2017 puede crear aplicaciones ASP.NET Core, siempre y cuando tenga instalado el .NET Core multiplataforma carga de trabajo de desarrollo. La figura 10-1 muestra la carga de trabajo requerida en el diálogo de configuración de Visual Studio 2017.

99 Capítulo 10



***La Figura 10-1.*** *Instalación de la carga de trabajo .NET Core en Visual Studio 2017.*

[Descargar Visual Studio 2017](https://www.visualstudio.com/downloads/)

**Código Visual Studio y CLI dotnet** (herramientas multiplataforma para Mac, Linux y Windows). Si prefiere un editor ligero y multiplataforma apoyar cualquier lenguaje de desarrollo, puede utilizar código de Microsoft Visual Studio y elpunto netCLI. Estos productos proporcionan una experiencia sencilla y robusta que hace más eficiente el flujo de trabajo del desarrollador. Adicionalmente, código de Visual Studio es compatible con las extensiones para C # y desarrollo web, que proporciona acceso directo y intelisense-tareas dentro del editor.

[Descargar el SDK de .NET Core](https://www.microsoft.com/net/download/core)

[Descargar código de Visual Studio](https://code.visualstudio.com/download)

flujo de trabajo de desarrollo para aplicaciones ASP.NET Core alojados en Azure

El ciclo de vida de desarrollo de la aplicación se inicia desde la máquina de cada desarrollador, que codifica la aplicación utilizando su idioma preferido y probar localmente. Los desarrolladores pueden elegir su sistema de control de fuente preferida y pueden configurar de integración continua (CI) y / o la entrega continua / distribución (CD) utilizando un servidor de compilación o en base a características Azure incorporadas.

Para comenzar con el desarrollo de una aplicación ASP.NET utilizando Core CI / CD, puede utilizar los servicios de Estudio equipo de Visual o propia de Team Foundation Server de su organización (TFS).

**Configuración inicial**

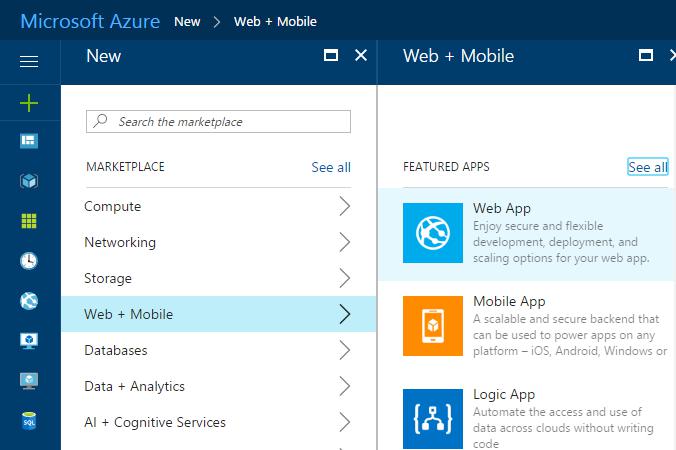
Para crear un canal de lanzamiento para su aplicación, es necesario tener el código de aplicación en el control de código fuente. Establecer un repositorio local y conectarlo a un repositorio remoto en un proyecto de equipo. Siga estas instrucciones:

|  |  |
| --- | --- |
| 100 | Capítulo 10 |
|  | Proceso de desarrollo de Azure |

* [Compartir su código con Git y Visual Studio](https://www.visualstudio.com/en-us/docs/git/share-your-code-in-git-vs) o
* [Compartir el código con TFVC y Visual Studio](https://www.visualstudio.com/en-us/docs/tfvc/share-your-code-in-tfvc-vs)

Crear un Servicio de Aplicación Azure donde implementar la aplicación. Crear una aplicación web a través de la hoja de Aplicación de Servicios en el portal Azure. Hacer clic+ Agregar, Selecciona el Aplicación Web plantilla, haga clic CrearY proporcionar un nombre y otros detalles. La aplicación web será accesible desde

{Nombre} .azurewebsites.net.



***Figura 10-2.*** *Creación de una nueva aplicación web de servicio de aplicaciones Azure en el Portal de Azure.*

Su proceso de construcción CI realizará una construcción automática cada vez que el nuevo código se ha comprometido a repositorio de control de código fuente del proyecto. Esto le da una retroalimentación inmediata que el código crea (y, a ser posible, pasa las pruebas automatizadas) y potencialmente puede ser desplegado. Esta acumulación CI producirá un paquete de despliegue de artefactos web y publicarla para el consumo de su proceso de DC.

[Definir su proceso de construcción CI](https://www.visualstudio.com/en-us/docs/build/apps/aspnet/aspnetcore-to-azure#ci)

Asegúrese de habilitar la integración continua para que el sistema pondrá en cola una acumulación cada vez que alguien en su equipo comete nuevo código. Prueba de la acumulación y verifique que se está produciendo un paquete de despliegue web como uno de sus artefactos.

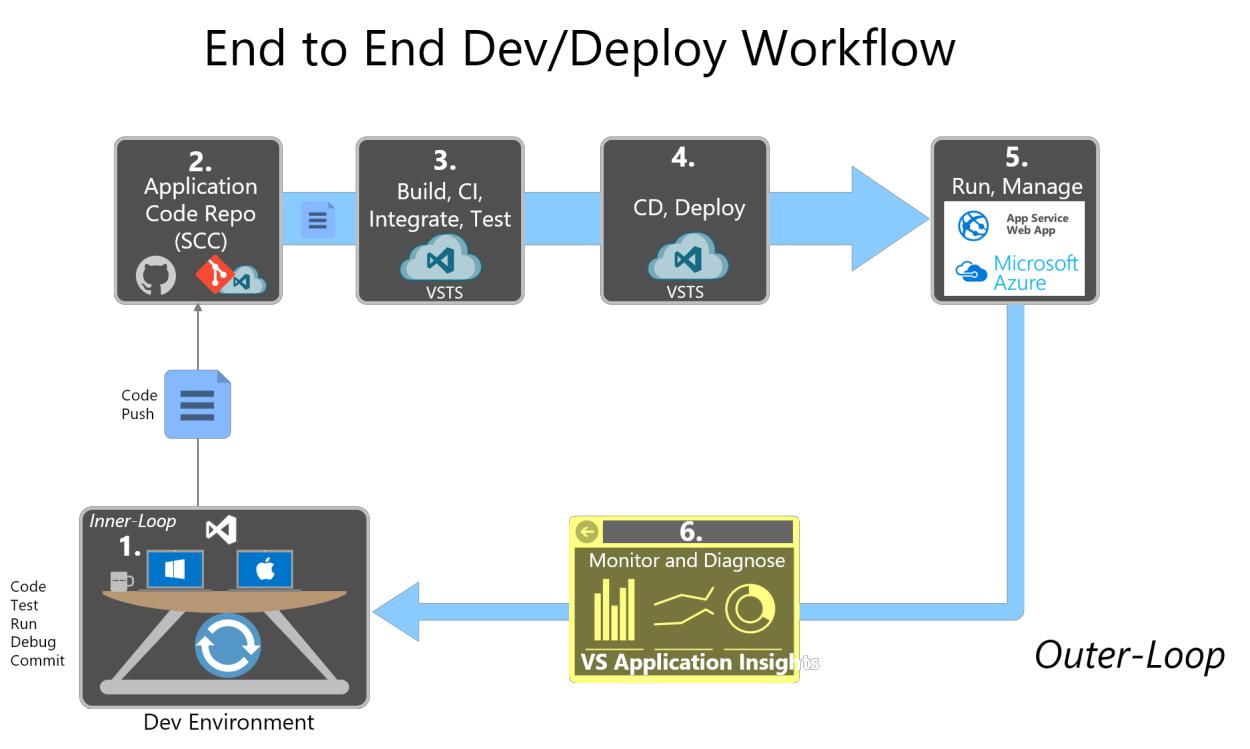
Cuando una acumulación tiene éxito, el proceso de DC desplegará los resultados de su CI construyen a su aplicación web Azure. Para configurar esto, crear y configurar un comunicado, que se desplegará a su aplicación Azure servicio.

|  |  |
| --- | --- |
| [Definir su proceso de lanzamiento del CD](https://www.visualstudio.com/en-us/docs/build/apps/aspnet/aspnetcore-to-azure#cd) |  |
| 101 | Capítulo 10 |
|  | Proceso de desarrollo de Azure |

Una vez que se ha configurado su tubería CI / CD, puede simplemente hacer cambios a su aplicación web y las aprenden de control de fuente para ellos han desplegado.

**Flujo de trabajo para el desarrollo de aplicaciones ASP.NET Core alojados en Azure**

Una vez que haya configurado su cuenta de Azure y su proceso de CI / CD, el desarrollo de aplicaciones ASP.NET alojada-Azure núcleo es simple. Los siguientes son los pasos básicos que toma habitualmente en la construcción de una aplicación ASP.NET Core, alojada en Azure Aplicación de servicio como una aplicación web, como se ilustra en la Figura 10-X.



***Figura 10-3.*** *Paso a paso del flujo de trabajo para la creación de aplicaciones ASP.NET Core y alojamiento en Azure.*

|  |  |
| --- | --- |
| 102 | Capítulo 10 |
|  | Proceso de desarrollo de Azure |

**Paso 1. Local Dev Medio Ambiente Inner Loop**

El desarrollo de su aplicación ASP.NET Core para su despliegue en Azure no es diferente de desarrollar la aplicación de otra manera. Utilizar el entorno de desarrollo local que se sienta cómodo, ya sea de Visual Studio 2017 o elpunto netCLI y código de Visual Studio o su editor preferido. Se puede escribir código, ejecutar y depurar sus cambios, ejecutar pruebas automatizadas, y hacer commits locales de control de código fuente hasta que esté listo para empujar los cambios a su repositorio de control de código fuente compartido.

**Step 2. Application Code Repository**

Cuando esté listo para compartir su código con su equipo, que debe empujar los cambios desde el repositorio de origen local para repositorio de código compartido de su equipo. Si usted ha estado trabajando en una rama de encargo, este paso por lo general implica la fusión de su código en una sucursal compartida (tal vez por medio de una[Halar](https://www.visualstudio.com/en-us/docs/git/pull-requests) [solicitud)](https://www.visualstudio.com/en-us/docs/git/pull-requests).

**Paso 3. Construir Servidor: integración continua. Construir, probar, empaquetar**

Una nueva construcción se inicia en el servidor de compilación siempre que un nuevo compromiso se hace al repositorio código de la aplicación compartida. Como parte del proceso de CI, esta formación debería compilar completamente la aplicación y ejecutar pruebas automáticas para confirmar que todo está funcionando como se esperaba. El resultado final del proceso de CI debe ser una versión empaquetada de la aplicación web, listo para su despliegue.

**Paso 4. Construir Servidor: Entrega Continua**

Una vez que una constitución que tuvo éxito, el proceso de DC va a recoger los artefactos producidos construcción. Esto incluirá un paquete de despliegue web. El servidor de compilación desplegará este paquete para Azure App de Servicio, reemplazando cualquier servicio existente por el recién creado. Normalmente este paso se dirige a un entorno de ensayo, pero algunas aplicaciones desplegar directamente a la producción a través de un proceso de DC.

**Paso 5. Aplicación de servicio Azure. Aplicación Web.**

Una vez desplegado, la aplicación ASP.NET Core se ejecuta dentro del contexto de una aplicación web de Azure servicio de aplicaciones. Esta aplicación web se puede supervisar y configurado, además, la utilización del Portal Azure.

**Paso 6. Monitoreo de la Producción y diagnóstico**

Mientras que la aplicación web se está ejecutando, puede supervisar el estado de la aplicación y recoger datos de diagnóstico y de comportamiento del usuario. Insights aplicación se incluye en Visual Studio, y ofrece la instrumentación automática para aplicaciones ASP.NET. Se le puede proporcionar información sobre el uso, excepciones, peticiones, el rendimiento y los registros.

**referencias**

**Construir y desplegar su aplicación ASP.NET Core a Azure**

<https://www.visualstudio.com/en-us/docs/build/apps/aspnet/aspnetcore-to-azure>

|  |  |
| --- | --- |
| 103 | Capítulo 10 |
|  | Proceso de desarrollo de Azure |

SECCIÓN 11

Azure de alojamiento Recomendaciones para ASP.NET Core Web Apps

*“Los líderes de línea de negocio de todo el mundo están pasando por alto los departamentos de TI para obtener aplicaciones de la nube (también conocido como SaaS) y pagar por ellos como lo harían con una suscripción a una revista. Y cuando ya no se requiere el servicio, pueden cancelar la suscripción sin el equipo no se utiliza en la esquina “.*

Daryl Plummer, analista de Gartner

Resumen

Sea cual sea las necesidades y la arquitectura de su aplicación, Windows Azure puede apoyarla. Sus necesidades de alojamiento pueden ser tan simples como un sitio web estático a una aplicación extremadamente sofisticada compuesta por decenas de servicios. Para ASP.NET núcleo monolítico aplicaciones web y servicios de apoyo, hay varias configuraciones bien conocidas que se recomiendan. Las siguientes recomendaciones están agrupadas según el tipo de recurso para ser alojados, si las aplicaciones de procesos completos, individuales o datos.

Aplicaciones web

Las aplicaciones web pueden alojarse con:

* Aplicaciones Web de servicios de aplicaciones

|  |  |
| --- | --- |
| 104 | Capítulo 11 |
|  | Azure de alojamiento Recomendaciones |

* contenedores
* Azure Servicio Tela
* Máquinas virtuales (VM)

De éstos, Servicio de Aplicación Las aplicaciones web son el método recomendado para la mayoría de los escenarios. Para arquitecturas MICROSERVICE, considere un enfoque basado en el recipiente, o tela de servicio. Si necesita más control sobre las máquinas que ejecutan su aplicación, considere Azure máquinas virtuales.

**Aplicaciones Web de servicios de aplicaciones**

Aplicación de servicio Web Apps ofrece una plataforma completamente gestionada optimizado para alojar aplicaciones web. Se trata de una plataforma como servicio (PaaS) oferta que le permite centrarse en su lógica de negocio, mientras que Azure se encarga de la infraestructura necesaria para correr y escalar la aplicación. Algunas características clave de Aplicaciones Web de servicios de aplicaciones:

* DevOps optimización (integración continua y la entrega, múltiples entornos, las pruebas A / B, el apoyo scripting)
* escala global y alta disponibilidad
* Las conexiones a las plataformas SaaS y sus datos de correo locales
* Seguridad y cumplimiento
* la integración de Visual Studio

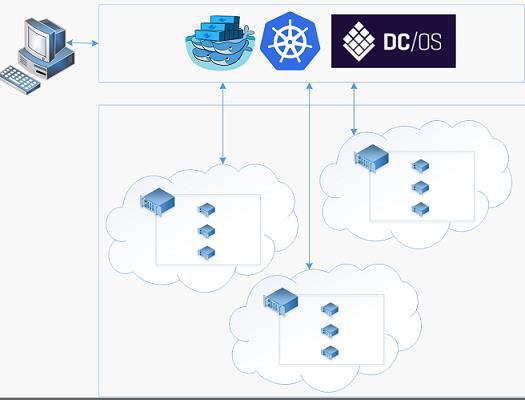
Azure App de servicio es la mejor opción para la mayoría de las aplicaciones web. Despliegue y gestión están integrados en la plataforma, los sitios pueden escalar rápidamente para manejar altas cargas de tráfico, y la incorporada en el Administrador de equilibrio de carga y tráfico de proporcionar una alta disponibilidad. Puede mover los sitios existentes en Azure App de servicio fácilmente con una herramienta de migración en línea, usar una aplicación de código abierto de la galería de aplicación web, o crear un nuevo sitio con el marco y los instrumentos de su elección. La característica WebJobs hace que sea fácil agregar el procesamiento de tareas de fondo para su aplicación web Servicio de Aplicación.

**Envases y Azure Container Service**

Azure servicio de contenedores hace que sea más fácil para que usted pueda crear, configurar y administrar un conjunto de máquinas virtuales que están preconfigurados para ejecutar aplicaciones en contenedores. Utiliza una configuración optimizada de programación y de orquestación herramientas de código abierto populares. Esto le permite utilizar sus habilidades existentes, o recurrir a un cuerpo grande y creciente de conocimientos de la comunidad, para desplegar y gestionar aplicaciones basadas en contenedor en Microsoft Azure.

Uno de los objetivos de Azure servicio de contenedores es proporcionar un entorno de alojamiento contenedor utilizando herramientas y tecnologías que son populares entre los clientes de Microsoft hoy de código abierto. Con este fin, Azure Container Service expone los puntos finales de API estándar de su orquestador elegido (CC / CA, acoplable Swarm, o Kubernetes). Mediante el uso de estos criterios de valoración, puede aprovechar cualquier software que es capaz de hablar con los criterios de valoración. Por ejemplo, en el caso del punto final del estibador Swarm, es posible optar por utilizar la interfaz de línea de comandos del estibador (CLI). Para DC / OS, es posible elegir el DCOS CLI. Para Kubernetes, puede elegir kubectl. La figura 11-1 muestra como se puede utilizar estos criterios de valoración para administrar los clústeres de contenedores.

|  |  |
| --- | --- |
| 105 | Capítulo 11 |
|  | Azure de alojamiento Recomendaciones |



***Figura 11-1.*** *gestión Azure Servicio Contenedor con estibador, Kubernetes, o CC / puntos finales OS.*

**Azure Servicio Tela**

Servicio de Tela es una buena opción si va a crear una nueva aplicación o re-escribir una aplicación existente para utilizar una arquitectura microService. Aplicaciones que se ejecutan en un conjunto compartido de máquinas, pueden empezar poco a poco y crecer a escala masiva con cientos o miles de máquinas, según sea necesario. servicios con estado que sea fácil de almacenar consistente y fiable estado de la aplicación, y el Servicio de Tela gestiona automáticamente la partición de servicio, la escala y la disponibilidad para usted. Tela servicio también es compatible con la Interfaz Web WebAPI Open for .NET (OWIN) y ASP.NET Core. En comparación con la aplicación de servicio, Servicio de Tejido también proporciona un mayor control sobre, o el acceso directo a la infraestructura subyacente. Puede remoto en sus servidores o configurar las tareas de inicio del servidor.

**Máquinas Virtuales Azure**

Si usted tiene una aplicación existente que requeriría modificaciones sustanciales para funcionar en aplicaciones de servicio o Service Tela, podría elegir máquinas virtuales con el fin de simplificar la migración a la nube. Sin embargo, la configuración correcta, asegurar y mantener las máquinas virtuales requiere mucho más tiempo y experiencia de TI en comparación con Azure Aplicación del Servicio y Tela. Si está pensando en máquinas virtuales Azure, asegúrese de tomar en cuenta el esfuerzo de mantenimiento en curso requerido para reparar, actualizar y administrar el entorno de máquina virtual. Máquinas Virtuales Azure es Infraestructura-as-a-Service (IaaS), mientras que la aplicación del Servicio y la tela son Platform-as-a-Service (PaaS).

|  |  |
| --- | --- |
| 106 | Capítulo 11 |
|  | Azure de alojamiento Recomendaciones |

**Comparación de funciones**

|  |  |  |  |
| --- | --- | --- | --- |
| Característica | Aplicación | Servicio | Virtual |
|  | Servicio | Tela | Máquina |
|  |  |  |  |
| Despliegue casi instantánea | X | X |  |
|  |  |  |  |
| Escalar hasta las máquinas más grandes sin redeploy | X | X |  |
|  |  |  |  |
| Las instancias compartir contenido y la configuración; no es necesario volver a implementar | X | X |  |
| o reconfigurar al escalar |  |  |  |
|  |  |  |  |
| entornos de despliegue múltiple (producción, puesta en escena) | X | X |  |
|  |  |  |  |
| Gestión automática de actualización del sistema operativo | X |  |  |
|  |  |  |  |
| la conmutación sin fisuras entre las plataformas de 32/64 bits | X |  |  |
|  |  |  |  |
| Implementar código con Git, FTP | X |  | X |
|  |  |  |  |
| Implementar código con WebDeploy | X |  | X |
|  |  |  |  |
| Implementar código con TFS | X | X | X |
|  |  |  |  |
| Proveedor de alojamiento web o servicio Web de etapa de la arquitectura de varios niveles | X | X | X |
|  |  |  |  |
| servicios de acceso Azure como servicio de autobuses, almacenamiento, base de datos SQL | X | X | X |
|  |  |  |  |
| Instalar ninguna costumbre MSI |  | X | X |
|  |  |  |  |

Los procesos lógicos

procesos lógicos individuales que pueden ser desacoplado del resto de la aplicación pueden ser desplegados de forma independiente a las funciones Azure de una manera “sin servidor”. Funciones Azure permite que acaba de escribir el código que necesita para un problema dado, sin tener que preocuparse acerca de la aplicación o infraestructura para ejecutarlo. Puede elegir entre una variedad de lenguajes de programación, incluyendo C #, C #, Node.js, Python y PHP, que le permite elegir el lenguaje más productivo para la tarea en cuestión. Como la mayoría de las soluciones basadas en la nube, usted sólo paga por la cantidad de tiempo que su uso, y se puede confiar en Funciones Azure para ampliar según sea necesario.

Datos

Azure ofrece una amplia variedad de opciones de almacenamiento de datos, por lo que su aplicación puede utilizar el proveedor de datos apropiada para los datos en cuestión.

Para transaccional, datos relacionales, bases de datos SQL Azure son la mejor opción. Para un alto rendimiento de lectura en su mayoría datos, un caché Redis el respaldo de una base de datos SQL Azure es una buena solución.

|  |  |
| --- | --- |
| 107 | Capítulo 11 |
|  | Azure de alojamiento Recomendaciones |

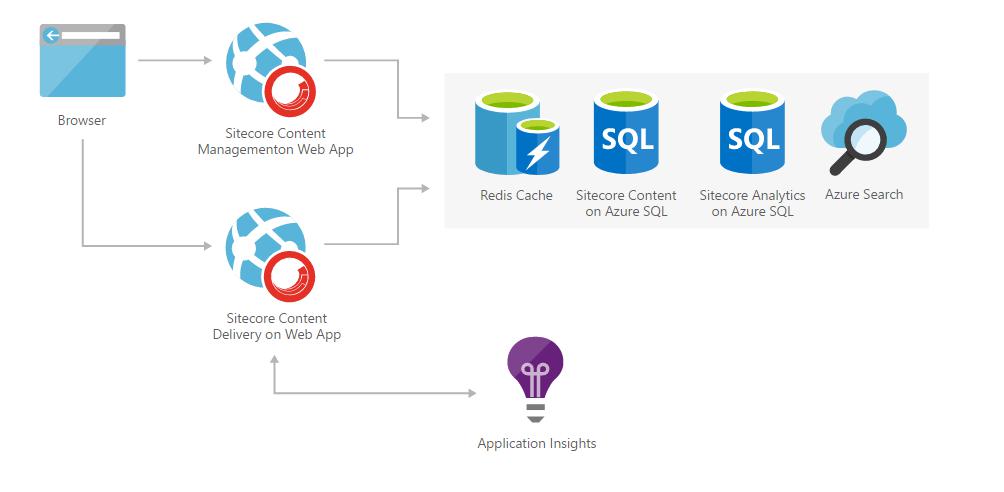
datos JSON no estructurados pueden ser almacenados en una variedad de formas, desde las columnas de base de datos SQL a gotas o de la mesa en Azure de almacenamiento, a DocumentDB. De éstos, DocumentDB ofrece la mejor funcionalidad de consulta, y es la opción recomendada para un gran número de documentos basados ​​en JSON que deben soportar consulta.

de comandos transitoria o datos basados ​​en eventos utilizados para orquestar comportamiento de la aplicación puede utilizar Azure Service Bus o Azure Storage colas. Azure Storage Bus ofrece una mayor flexibilidad y es el servicio recomendado para la mensajería no trivial dentro y entre las aplicaciones.

Recomendaciones Arquitectura

requisitos de su aplicación deben dictar su arquitectura. Hay muchos servicios de Azure diferentes disponibles, elegir el más adecuado es una decisión importante. Microsoft ofrece una galería de arquitecturas de referencia para ayudar a identificar las arquitecturas típicas optimizados para escenarios comunes. Es posible enlazar una arquitectura de referencia que eran similares con los requisitos de su aplicación, o al menos ofrece un punto de partida.

La Figura 11-2 muestra una arquitectura ejemplo de referencia. Este diagrama describe un enfoque de arquitectura recomendada para un sitio web gestor de contenidos Sitecore optimizado para su comercialización.



***Figura 11-2.*** *arquitectura de referencia marketing website Sitecore.*

**Referencias - Recomendaciones de Alojamiento Azure**

* Azure Arquitecturas de soluciones

<https://azure.microsoft.com/en-us/solutions/architecture/>

* Guía Azure desarrollador <https://azure.microsoft.com/en-us/campaigns/developer-guide/>
* ¿Cuál es Azure App de servicio? <https://docs.microsoft.com/en-us/azure/app-service/app-service-value-prop-what-is>

|  |  |
| --- | --- |
| 108 | Capítulo 11 |
|  | Azure de alojamiento Recomendaciones |

* Azure servicio de aplicaciones, máquinas virtuales, Servicio de Tela y servicios en la nube Comparación

<https://docs.microsoft.com/en-us/azure/app-service-web/choose-web-site-cloud-service-vm>

|  |  |
| --- | --- |
| 109 | Capítulo 11 |
|  | Azure de alojamiento Recomendaciones |