

# Contents

## 介绍

新式 Web 应用程序的特征

在传统 Web 应用和单页应用之间选择

体系结构原则

常用 Web 应用程序体系结构

常用客户端 Web 技术

开发 ASP.NET Core MVC 应用

在 ASP.NET Core 中使用数据

测试 ASP.NET Core MVC 应用

Azure 的开发过程

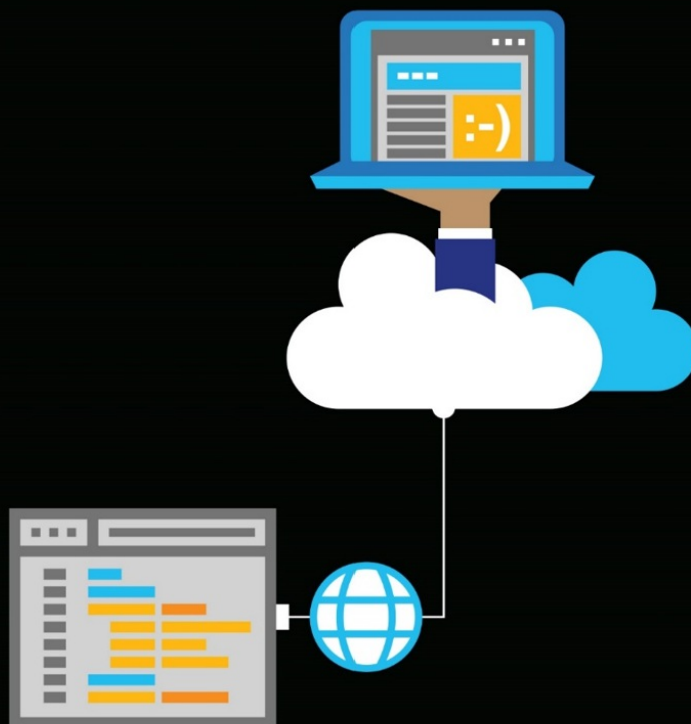
ASP.NET Web 应用的 Azure 托管建议

# 使用 ASP.NET Core 和 Azure 构建新式 Web 应用程序

2020/2/20 • [Edit Online](#)



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



Steve “ardalis” Smith

发布者

Microsoft 开发人员部门、.NET 和 Visual Studio 产品团队

Microsoft Corporation 的一个部门

One Microsoft Way

Redmond, Washington 98052-6399

版权所有 © 2020 Microsoft Corporation

保留所有权利。未经发布者书面许可，不得以任何形式或任何方式复制或传播本书中的任何内容。

本书“按原样”提供，表达作者的观点和看法。本书中表达的观点、看法和信息（包括 URL 和其他 Internet 网站引用）如有更改，恕不另行通知。

本书中提及的一些示例仅用于说明，纯属虚构。不存在任何实际关联或联系，请勿妄加推断。

Microsoft 和 <https://www.microsoft.com> 上“商标”网页列出的商标是 Microsoft 集团公司的商标。

Mac 和 macOS 是 Apple Inc. 的商标

Docker 的鲸鱼徽标是 Docker Inc. 的注册商标经许可方可使用。

所有其他标记和徽标均为其各自所有者的财产。

作者：

**Steve "ardalis" Smith** - 软件设计师及培训师 - [Ardalis.com](https://ardalis.com)

编辑：

Maira Wenzel

## 介绍

相比传统 .NET 开发，.NET Core 和 ASP.NET Core 具有一系列优势。如果以下所有方面或一些方面对于你的应用程序成功至关重要，应将 .NET Core 用于服务器应用程序：

- 跨平台支持。
- 微服务的使用。
- Docker 容器的使用。
- 高性能和可伸缩性需求。
- 在同一服务器上通过应用程序对 .NET 版本进行并行版本控制。

传统 .NET 应用程序虽然支持许多以上要求，但是 ASP.NET Core 和 .NET Core 已经过优化，可为以上方案提供完善的支持。

越来越多的组织选择使用 Microsoft Azure 等服务，在云中托管 web 应用程序。如果以下方面对你的应用程序或组织至关重要，应该考虑在云中托管应用程序：

- 减少对数据中心的成本投入（硬件、软件、空间、实用工具、服务器管理等）
- 灵活的定价（基于使用情况付费，无需为空闲容量付费）。
- 高可靠性。
- 改善应用移动性；可轻松更改部署应用的位置和方式。

- 灵活的容量; 基于实际需要增加或减少。

使用 Azure 中托管的 ASP.NET Core 生成 Web 应用, 与传统替代方法相比, 这能提供许多竞争优势。ASP.NET Core 针对新式 web 应用程序开发做法和云托管方案进行了优化。本指南介绍如何构建 ASP.NET Core 应用程序以充分利用这些功能。

## 目标

本指南提供了使用 ASP.NET Core 和 Azure 构建单片 Web 应用程序的端到端指导。在此上下文中, “单片”是指这一事实, 即这些应用程序会作为单个单元部署, 而不是作为交互服务和应用程序的集合。

本指南是“[.NET 微服务 - 容器化 .NET 应用程序体系结构](#)”的补充, 该文章更侧重于介绍 Docker、微服务和部署容器以托管企业应用程序。

### .NET 微服务。适用于容器化 .NET 应用程序的体系结构

- 电子书  
<https://aka.ms/MicroservicesEbook>
- 示例应用程序  
<https://aka.ms/microservicesarchitecture>

## 本指南的目标读者

本指南的受众主要是对使用云中的 Microsoft 技术和服务生成新式 web 应用程序感兴趣的开发者、开发潜在顾客和架构师。

次要受众是技术决策者, 他们已经熟悉 ASP.NET 或 Azure, 并想要了解是否需要为新项目或现有项目升级到 ASP.NET Core。

## 如何使用本指南

本指南已精简为较小的文档, 侧重介绍如何使用新式 .NET 技术和 Windows Azure 生成 web 应用程序。可通读本指南, 了解有关此类应用程序及其技术注意事项的基本信息。本指南及其示例应用程序还可作为操作起点或参考。可将相关示例应用程序作为你自己的应用程序的模板, 或者了解如何组织应用程序的组件部件。在对自己的应用程序进行选择权衡时, 请参考指南的原则、体系结构的范围以及技术选项和决策注意事项。

请将本指南转发到团队中, 这有助于确保对这些注意事项和机会的共同理解。确保每个人使用共同的术语和基础原则工作, 这有助于构建模式和做法的一致应用。

## reference

- 为服务器应用选择 .NET Core 或 .NET Framework  
<https://docs.microsoft.com/dotnet/standard/choosing-core-framework-server>

下一篇

# 新式 Web 应用程序的特征

2020/2/27 • • [Edit Online](#)

"... 恰当合理的设计，会使功能的价格变便宜。这种方式有难度，但是却总有成效。"

- Dennis Ritchie

新式 Web 应用程序比以往承载着更高的用户期望和要求。当今的 Web 应用需要能在全球任何地方任意时刻可用，需要可在任何设备或屏幕尺寸上使用。Web 应用程序必须具有安全性、灵活性和可缩放性，以便满足高峰需求。如今复杂方案日益需要由丰富的用户体验来处理，这建立在使用 JavaScript 并通过 Web API 进行有效通信的客户端的基础之上。

ASP.NET Core 针对新式 Web 应用程序和基于云的托管方案进行了优化。其模块化设计使应用程序仅依赖于其实际使用的功能，从而提升应用程序安全性和性能，降低托管资源要求。

## 参考应用程序: eShopOnWeb

本指南包含一个参考应用程序 eShopOnWeb，该应用程序演示了一些原则和建议。该应用程序是一个简单在线商店，支持浏览衬衫、咖啡杯和其他市场产品名录。特意选择该简单的参考应用，方便理解。

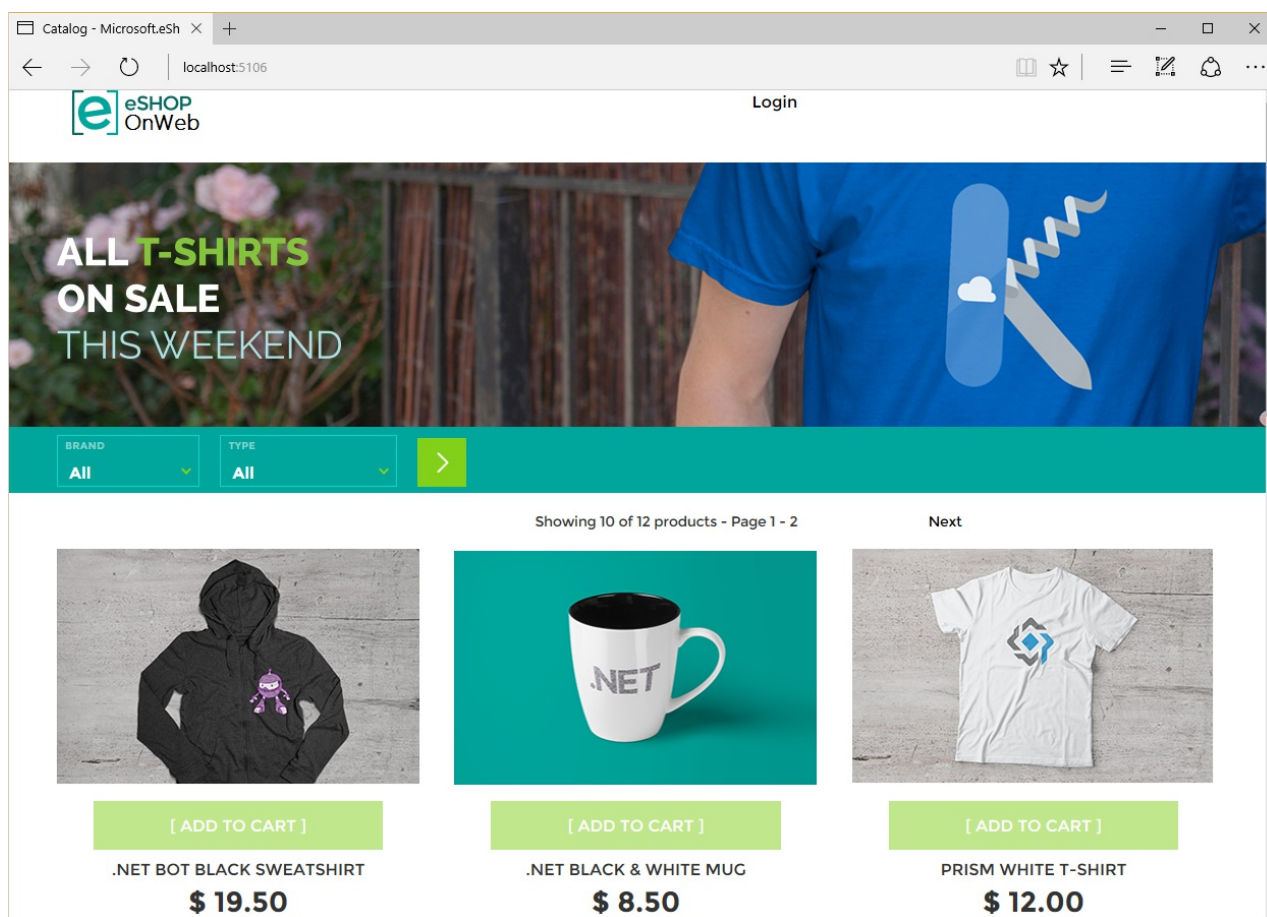


图 2-1. eShopOnWeb

### 参考应用程序

- eShopOnWeb

<https://github.com/dotnet/eShopOnWeb>

## 云托管和可缩放

由于低内存、高吞吐量的特点，ASP.NET Core 针对云（公共云、私有云以及任何云）进行了优化。ASP.NET Core 应用程序占用空间更小，因此在相同硬件上可托管更多此类应用。并且，使用即用即付云托管服务时仅需为极少数资源付费。更高的高吞吐量意味着，通过相同硬件上的一个应用，你可服务更多客户，进而降低服务器和托管基础设施所需的投入。

## 跨平台

ASP.NET Core 具有跨平台性，可在 Linux、macOS 以及 Windows 上运行。这为开发和部署通过 ASP.NET Core 构建的应用程序带来了新的选择。Docker 容器（包括 Linux 和 Windows）可托管 ASP.NET Core 应用程序，从而使应用程序可利用[容器和微服务](#)的优势。

## 模块化和松散耦合

NuGet 包在 .NET Core 中处于第一等级，而 ASP.NET Core 应用经 NuGet 由众多库构成。此功能的粒度有助于确保应用仅依赖和部署其实际所需的功能，从而降低占用空间和安全漏洞外围应用。

ASP.NET Core 还完全支持内部和应用程序级别的[依赖项注入](#)。接口可具有多个能按需换出的实现。依赖项注入可使应用松散耦合到此类接口，而不是特定的实现，从而使其扩展、维护和测试更加容易。

## 通过自动测试轻松实现测试

ASP.NET Core 应用程序支持单元测试，并且通过其松散耦合和依赖项注入支持，使得出于测试目的很容易将基础设施顾虑转换为虚假实现。ASP.NET Core 还附带可用于托管内存中应用的 TestServer。功能测试然后可向此内存中服务器发出请求，从而能在真实服务器上托管应用和通过网络层作出请求所需时间的短暂时间内，执行完整的应用程序堆栈（包括中间件、路由、模型绑定、筛选器等）和接收响应。对于在新式 Web 应用程序中愈发重要的 API 而言，这些测试非常易于编写，具有很大作用。

## 支持传统和 SPA 行为

传统 Web 应用程序涉及极少的客户端行为，而是依赖服务器执行该应用可能需要执行的所有导航、查询和更新。用户执行的每个新操作会被转换为新的 Web 请求，其结果是最终用户浏览器中的完整页面重载。经典模型视图控制器（MVC）框架通常采用此方式，其中每个新请求对应一个不同的控制器操作，这些操作会反过来作用于模型并返回一个视图。特定页面上的一些单独操作可使用 AJAX（异步 JavaScript 和 XML）功能改进，但该应用的总体体系结构使用了多个不同的 MVC 视图和 URL 终结点。此外，ASP.NET Core MVC 还支持 Razor Pages，这是一种组织 MVC 样式页的较为简单的方法。

与之相反，单页应用程序（SPAs）涉及极少动态生成的服务器端页面负载（如有）。许多 SPA 在一个静态 HTML 文件中进行初始化，此文件加载该应用启动和运行所需的 JavaScript 库。这些应用大量使用 Web API 处理数据需求，并提供更为丰富的用户体验。

许多 Web 应用程序同时涉及传统 Web 应用程序行为（尤其是对于内容）和 SPA（对于交互）。ASP.NET Core 通过使用相同工具集和基础框架库，在同一应用程序中同时支持 MVC（基于视图或页面）和 Web API。

## 简单的开发和部署

可使用简单的文本编辑器、命令行接口或者全功能开发环境（例如 Visual Studio）编写 ASP.NET Core 应用程序。整体应用程序通常部署到单个终结点。可轻松将部署自动化，作为持续集成（CI）和持续交付（CD）管道的一部分。除传统的 CI/CD 工具外，Microsoft Azure 还集成了 git 存储库支持，并可在对特定 git 分支或标记作出更新时自动部署更新。Azure DevOps 提供了功能完备的 CI/CD 生成和部署管道，而 GitHub Actions 为在那里托管的项目提供了另一个选择。

## 传统的 ASP.NET 和 Web 窗体

除 ASP.NET Core 外, 传统的 ASP.NET 4.x 依然是一个用于构建 Web 应用程序的可靠强大的平台。ASP.NET 支持 MVC 和 Web API 开发模型以及 Web 窗体。Web 窗体非常适合用于基于页面的丰富应用程序开发, 具有丰富的第三方组件系统。Microsoft Azure 长期以来支持 ASP.NET 4.x 应用程序, 而且许多开发人员非常熟悉该平台。

## Blazor

Blazor 包含在 ASP.NET Core 3.0 和更高版本中。它提供了一种新机制, 可使用 Razor、C# 和 ASP.NET Core 生成丰富的交互式 Web 客户端应用程序。它提供了在开发新式 Web 应用程序时要考虑的另一种解决方案。有两个版本的 Blazor 可供考虑: 服务器端和客户端。

服务器端 Blazor 于 2019 年随 ASP.NET Core 3.0 一起发布。顾名思义, 它在服务器上运行, 通过网络将对客户端文档的更改重新呈现到浏览器。服务器端 Blazor 提供了丰富的客户端体验, 而无需客户端 JavaScript, 也不需要为每个客户端页面交互单独加载页面。服务器请求并处理已加载页面中的更改, 然后使用 SignalR 将其发送回客户端。

客户端 Blazor 将于 2020 年发布, 将不需要在服务器上进行更改。相反, 它将利用 WebAssembly 在客户端中运行 .NET 代码。如果需要请求数据, 客户端仍然可以对服务器进行 API 调用, 但是所有客户端行为都通过 WebAssembly 在客户端中运行, WebAssembly 在所有主流浏览器中受支持, 并且只是一个 Javascript 库。

### 参考 - 新式 Web 应用程序

- **ASP.NET Core 简介**  
<https://docs.microsoft.com/aspnet/core/>
- **在 ASP.NET Core 进行测试**  
<https://docs.microsoft.com/aspnet/core/testing/>
- **Blazor - 入门**  
<https://blazor.net/docs/get-started.html>

[上一页](#)[下一页](#)



# 在传统 Web 应用和单页应用 (SPA) 之间选择

2020/2/27 • • [Edit Online](#)

“Atwood 定律:任何能够用 JavaScript 编写的应用程序, 最终必将用 JavaScript 编写。”

- Jeff Atwood

目前可通过两种通用方法来构建 Web 应用程序:在服务器上执行大部分应用程序逻辑的传统 Web 应用程序, 以及在 Web 浏览器中执行大部分用户界面逻辑的单页应用程序 (SPA), 后者主要使用 Web API 与 Web 服务器通信。也可以将两种方法混合使用, 最简单的方法是在更大型的传统 Web 应用程序中托管一个或多个丰富 SPA 类子应用程序。

何时使用传统 Web 应用程序:

- 应用程序的客户端要求简单, 甚至要求只读。
- 应用程序需在不支持 JavaScript 的浏览器中工作。
- 团队不熟悉 JavaScript 或 TypeScript 开发技术。

何时使用 SPA:

- 应用程序必须公开具有许多功能的丰富用户界面。
- 团队熟悉 JavaScript 和/或 TypeScript 开发。
- 应用程序已为其他(内部或公共)客户端公开 API。

此外, SPA 框架还需要更强的体系结构和安全专业知识。相较于传统 Web 应用程序, SPA 框架需要进行频繁的更新和使用新框架, 因此改动更大。相较于传统 Web 应用, SPA 应用程序在配置自动化生成和部署过程以及利用部署选项(如容器)方面的难度更大。

使用 SPA 方法改进用户体验时必须权衡这些注意事项。

## Blazor

ASP.NET Core 3.0 引入了一种新模型, 用于构建称为 Blazor 的丰富的、交互式 and 可组合的 UI。Blazor 服务器端允许开发人员在服务器上使用 Razor 构建 UI, 还使用 [WebAssembly](#) 将此代码传递到浏览器和执行客户端。现在 ASP.NET Core 3.0 或更高版本中提供 Blazor 服务器端。Blazor 客户端应该于 2020 年推出。

Blazor 提供了一个全新的第三个选项, 可用于评估是否生成纯服务器呈现的 Web 应用程序或 SPA。可以使用 Blazor 生成类似于 SPA 的丰富客户端行为, 而无需进行大量 JavaScript 开发。Blazor 应用程序可以调用 API 来请求数据或执行服务器端操作。

在以下情况下考虑使用 Blazor 生成 Web 应用程序:

- 应用程序必须公开丰富用户界面
- 与使用 JavaScript 或 TypeScript 开发相比, 团队更喜欢使用 .NET 开发

有关 Blazor 的详细信息, 请参阅 [Blazor 入门](#)。

## 何时选择传统 Web 应用

以下内容详细介绍前面提到的选择传统 Web 应用程序的原因。



## 应用程序的客户端要求简单，可能要求只读

对许多 Web 应用程序而言，其大部分用户的主要使用方式是只读。只读(或以读取为主)应用程序往往比那些维护和操作大量状态的应用程序简单得多。例如，搜索引擎可能由一个带有文本框的入口点和用于显示搜索结果的第二页组成。匿名用户可以轻松提出请求，并且很少需要使用客户端逻辑。同样，一般而言，博客或内容管理系统中面向公众的应用程序主要包含的内容与客户端行为关系不大。此类应用程序容易构建为基于服务器的传统 Web 应用程序，在 Web 服务器上执行逻辑，并呈现要在浏览器中显示的 HTML。事实上，网站的每个独特页面都有自己的 URL，搜索引擎可以将其存为书签和编入索引(默认设置，无需将其添加为应用程序的单独功能)，这也是此类情况的一个明显优势。

## 应用程序需在不支持 JavaScript 的浏览器中工作

如需在有限或不支持 JavaScript 的浏览器中工作的 Web 应用程序，则应使用传统的 Web 应用 workflows 编写(或至少可以回退到此类行为)。SPA 需要客户端 JavaScript 才能正常工作；如果没有客户端 JavaScript，SPA 不是好的选择。

## 团队不熟悉 JavaScript 或 TypeScript 开发技术

如果团队不熟悉 JavaScript 或 TypeScript，但熟悉服务器端 Web 应用程序开发，那相较于 SPA，他们交付传统 Web 应用的速度可能更快。除非以学习 SPA 编程为目的，或需要 SPA 提供用户体验，否则对已经熟悉构建传统 Web 应用的团队而言，选择传统 Web 应用的工作效率更高。

# 何时选择 SPA

以下内容详细介绍何时为 Web 应用选择单页应用程序开发样式。

## 应用程序必须公开具有许多功能的丰富用户界面

SPA 可支持丰富客户端功能，当用户执行操作或在应用的各区域间导航时无需重新加载页面。SPA 很少需要重新加载整个页面，因此加载速度更快，可在后台提取数据，并且对单个用户操作的响应更快。SPA 支持增量更新，可保存尚未完成的窗体或文档，而无需用户单击按钮提交窗体。SPA 支持丰富的客户端行为，例如拖放，比传统应用程序更容易操作。可以将 SPA 设计为在断开连接的模式下运行，对客户端模型进行更新，并在重新建立连接后将更新最终同步回服务器。如果应用要求包括丰富的功能，且超出了典型 HTML 窗体提供的功能，则选择 SPA 样式应用程序。

通常，SPA 需要实现内置于传统 Web 应用中的功能，例如在反映当前操作的地址栏中显示有意义的 URL(并允许用户将此 URL 存为书签或对其进行深层链接以便返回此 URL)。SPA 还应允许用户使用浏览器的后退和前进按钮寻找用户意料之中的结果。

## 团队熟悉 JavaScript 和/或 TypeScript 开发

编写 SPA 需要熟悉 JavaScript 和/或 TypeScript 以及客户端编程技术和库。团队应有能力像使用 Angular 一样使用 SPA 框架编写新式 JavaScript。

### 参考 - SPA 框架

- **Angular**  
<https://angular.io>
- **React** <https://reactjs.org/>
- **JavaScript 框架的比较**  
<https://jsreport.io/the-ultimate-guide-to-javascript-frameworks/>

## 应用程序已为其他(内部或公共)客户端公开 API

如果已提供一个 Web API 供其他客户端使用，则相较于在服务器端窗体中复制逻辑，创建一个利用这些 API 的 SPA 实现更加容易。用户与应用程序交互时，SPA 广泛使用 Web API 来查询和更新数据。

# 何时选择 Blazor

以下是有关何时为 Web 应用选择 Blazor 的详尽说明。

## 应用程序必须公开丰富用户界面

与基于 JavaScript 的 SPA 一样，Blazor 应用程序可以支持丰富客户端行为，而无需重载页面。这些应用程序对用户的响应更快，仅获取响应给定用户交互所需的数据(或 HTML)。如果设计得当，则服务器端 Blazor 应用可以配置为以客户端 Blazor 应用的形式运行，并且在支持该功能后只需进行最小的更改。

## 与使用 JavaScript 或 TypeScript 开发相比，团队更喜欢使用 .NET 开发

与使用 JavaScript 或 TypeScript 等客户端语言相比，许多使用 .NET 和 Razor 的开发人员的工作效率更高。由于已经使用 .NET 开发了应用程序的服务器端，因此，使用 Blazor 可以确保团队中的每个 .NET 开发人员都可以理解并且可能会生成应用程序前端的行为。

# 决策表

下面的决策表总结了在传统 Web 应用程序、SPA 或 Blazor 应用之间进行选择时要考虑的一些基本因素。

因素	传统 WEB 应用	单页面应用程序	BLAZOR 应用
需要团队熟悉 JavaScript/TypeScript	最低	必需	最低
支持不带脚本的浏览器	支持	不支持	支持
客户端应用程序行为极少	适合	不必要	可行
丰富而复杂的用户界面要求	受限	适合	适合

上一页

下一页

# 体系结构原则

2020/2/27 • [Edit Online](#)

“如果建筑师按照程序员编写程序的方式建造建筑物，那么第一只到来的啄木鸟(找 Bug)就将摧毁文明。”

- Gerald Weinberg

构建和设计软件解决方案时应考虑到可维护性。本部分概述的原则可帮助指导你作出体系结构决策，生成简洁、可维护的应用程序。一般而言，在这些原则的指导下构建的应用程序各部分间可通过显式接口或消息传送系统进行通信，并非松散耦合的离散组件。

## 通用设计原则

### 分离关注点

分离关注点是开发时的指导原则。此原则主张应根据软件执行的工作类型将软件分离。例如，假设应用程序中包含两个逻辑，其中一个逻辑标识要显示给用户的注意事项，另一个以特定方式设置这些注意事项的格式，使其更加显眼。负责选择要为哪些事项设置格式的行为应与负责设置事项格式的行为区分开，因为这两种行为只是碰巧彼此相关联的分离关注点。

从体系结构上来说，按此原则有逻辑地构建应用程序应将核心业务行为与基础结构及用户界面逻辑区分开。理想情况下，业务规则和逻辑应单独位于一个项目中，且该项目不依赖于应用程序中的其他项目。这样可帮助确保该业务模型易于测试，且可在不与低级别实现详细信息紧密耦合的情况下逐步改进。在应用程序体系结构的使用层背后，关注点分离是核心设计思想。

### 封装

应用程序的不同部分应通过封装与应用程序中的其他部分隔离开。只要不违反外部协定，应用程序组件和层应能在不中断其协作者的情况下调整其内部实现。正确使用封装有助于在应用程序设计中实现松散耦合及模块化，因为只要维持相同的接口，就可以用替代实现来替代对象和包。

在类中实现封装的方式是限制对该类的内部状态的外部访问权限。如果外部参与者想操作对象的状态，则应通过明确定义的函数(或属性 setter)来进行操作，而非直接访问该对象的私有状态。同样，应用程序组件和应用程序本身应公开明确定义的接口供协作者使用，而非让协作者直接修改其状态。这样一来，只要公共协定得到维护，你就可以不断改进应用程序的内部设计，而无需担心会中断协作者。

### 依赖关系反转

应用程序中的依赖关系方向应该是抽象的方向，而不是实现详细信息的方向。大部分应用程序都是这样编写的，以便编译时依赖关系顺着运行时执行的方向流动。这将产生一个直接依赖项关系图。也就是说，如果模块 A 调用模块 B 中的函数，而模块 B 又调用模块 C 中的函数，则编译时 A 取决于 B，而 B 又取决于 C，如图 4-1 中所示。

# Direct Dependency Graph

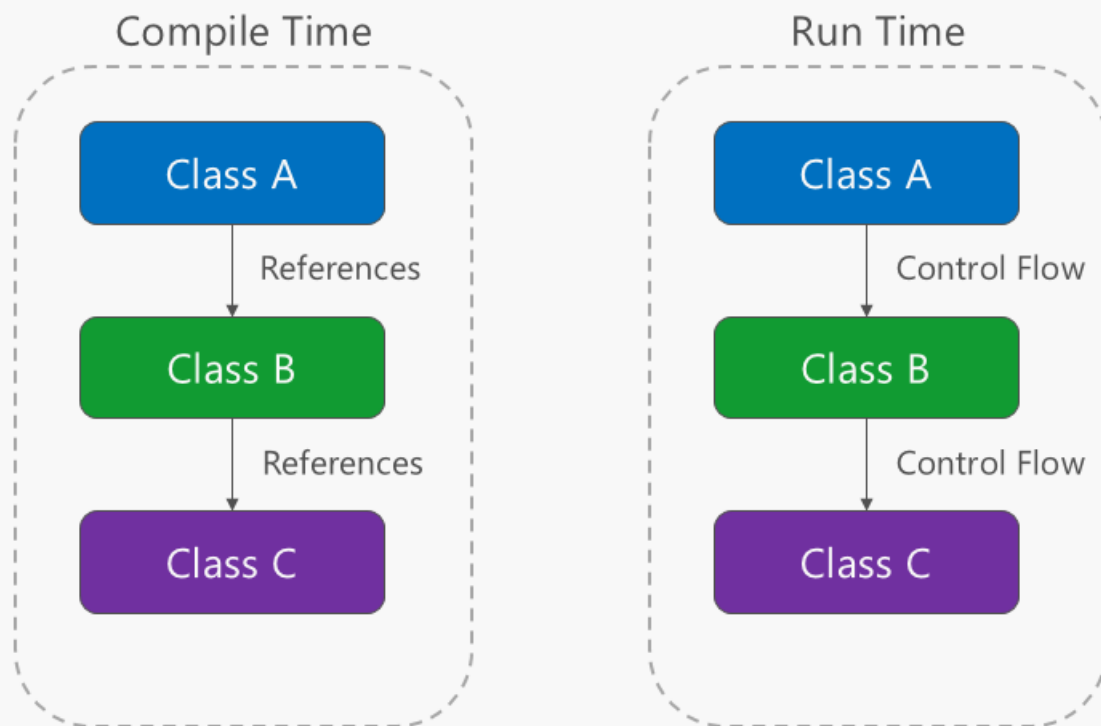


图 4-1。直接依赖项关系图。

应用依赖关系反转原则后，A 可以调用 B 实现的抽象上的方法，让 A 可以在运行时调用 B，而 B 又在编译时依赖于 A 控制的接口（因此，典型的编译时依赖项发生反转）。运行时，程序执行的流程保持不变，但接口引入意味着可以轻松插入这些接口的不同实现。

## Inverted Dependency Graph

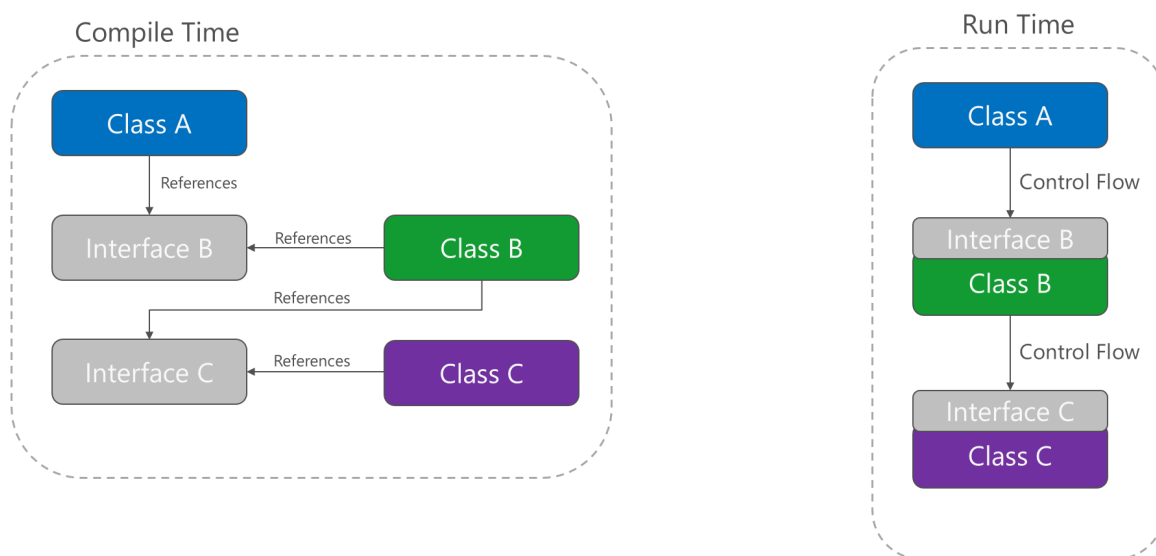


图 4-2。反转依赖项关系图。

依赖项反转是生成松散耦合应用程序的关键一环，因为可以将实现详细信息编写为依赖并实现更高级别的抽象，而不是相反。因此，生成的应用程序的可测试性、模块化程度以及可维护性更高。遵循依赖关系反转原则可实现依

赖关系注入。

## 显式依赖关系

方法和类应显式要求正常工作所需的任何协作对象。通过类构造函数，类可以标识其实现有效状态和正常工作所需的内容。如果定义的类可供构造和调用，但仅在具备特定全局组件或基础结构组件时正常工作，则这些类对其客户端而言就不诚实。构造函数协定将告知客户端，它只需要指定的内容(如果类只使用无参数构造函数，则可能不需要任何内容)，但随后在运行时，结果发现对象确实需要某些其他内容。

若遵循显式依赖关系原则，类和方法就会诚实地告知客户端其需要哪些内容才能工作。这就可以让代码更好地自我记录，并让代码协定更有利于用户，因为用户相信只要他们以方法或构造函数参数的形式提供所需的内容，他们使用的对象在运行时就能正常工作。

## 单一责任

单一责任原则适用于面向对象的设计，但也可被视为类似于分离关注点的体系结构原则。它指出对象只应有一个责任，并且只能因为一个原因更改对象。具体而言，只在必须更新对象执行其唯一责任的方式时才应更改对象。遵循这一原则有助于生成更松散耦合和模块化的系统，因为许多类型的新行为可以作为新类实现，而不是通过向现有类添加其他责任。添加新类始终比更改现有类安全，因为还没有任何代码依赖于新类。

在整体应用程序中，可以在高级别将单一责任原则应用于应用程序中的层。显示责任应位于 UI 项目中，而数据访问责任应位于基础结构项目中。业务逻辑应位于应用程序核心项目中，该项目易于测试，并且可以独立于其他责任进行逐步改进。

将此原则应用到应用程序体系结构及其逻辑终结点时，你将获得微服务。给定的微服务应具有单一责任。一般而言，如果需要扩展系统的行为，最好通过添加其他微服务来实现，而不要向现有微服务添加责任。

## 详细了解微服务体系结构

## 不要自我重复 (DRY)

应用程序应避免在多个位置指定与特定概念相关的行为，因为这样经常会导致出错。有些时候，对要求中的某处进行更改需要更改此行为，并且该行为可能至少有一个实例无法更新，这种可能性将导致出现不一致的系统行为。

请将逻辑封装在编程构造中，而不要重复该逻辑。让此构造成为针对此行为的单一权限，并让应用程序中需要此行为的任何其他部分都使用新的构造。

### NOTE

避免将恰巧重复的行为绑定在一起。例如，只因为两个不同的常数具有相同的值，如果从概念上讲两个常数是指不同的内容，这并不意味着只应使用一个常数。

## 持久性无感知

持久性无感知 (PI) 是指需要保持不变的类型，但其代码不受所选择的持久性技术的影响。.NET 中的这种类型有时被称为普通旧 CLR 对象 (POCO)，因为这种类型无需继承特定的基类或实现特定的接口。持久性无感知非常有用，因为它可以让相同的业务模型以多种方式保持不变，让应用程序更加灵活。持久性选择可能会随着时间的推移而发生变化，从一种数据库技术变为另一种数据库技术，或除应用程序一开始具备的持久性形式之外还需要其他形式的持久性(例如，除相关数据库之外还需使用 Redis 缓存或 Azure Cosmos DB)。

违反此原则的一些示例包括：

- 必需的基类。
- 必需的接口实现。
- 负责保存其自身的类(例如活动记录模式)。
- 所需的无参数构造函数。
- 需要 virtual 关键字的属性。

- 特定于持久性的必需特性。

要求类具有上述任何特性或行为会增加要保持不变的类型和持久性技术的选择之间的耦合，从而增加将来采用新的数据访问策略的难度。

## 有界上下文

有界上下文是领域驱动设计中的中心模式。它们可以将大型应用程序或组织分解为独立的概念模块，通过这种方式来解决复杂性问题。每个概念模块表示各自独立的上下文(因此有界)，并且可以独立改进。理想情况下，每个有界上下文都应该能够为其中的概念自由选择它自己的名称，并对其自己的持久性存储具有独占访问权限。

至少，各 Web 应用程序应努力成为自己的有界上下文，为其业务模型提供自己的持久性存储，而不是与其他应用程序共享数据库。有界上下文之间的通信通过编程接口进行，而不是通过共享数据库进行，这样可以引发业务逻辑和事件来响应发生的更改。有界上下文会紧密映射到微服务，后者在理想情况下也作为其自己的单独有界上下文实现。

## 其他资源

- [JAVA 设计模式:原则](#)
- [有界上下文](#)

[上一页](#)[下一页](#)

# 常用 Web 应用程序体系结构

2020/2/27 • [Edit Online](#)

“如果你认为好的体系结构很昂贵，试试糟糕的体系结构吧。”

- Brian Foote 和 Joseph Yoder

大多数传统 .NET 应用程序都部署为单一单位，对应于单一 IIS 应用域中运行的可执行文件或单个 Web 应用程序。这是最简单的部署模型，能很好地为众多内部和小型公共应用程序提供服务。然而，即使提供此单一单位部署，大多数重要的商业应用程序仍受益于逻辑分层。

## 什么是整体式应用程序？

就其行为而言，整体式应用程序是完全独立的应用程序。在执行操作的过程中，该应用程序可能与其他服务或数据存储发生交互，但其核心业务在其自身进程中运转，且整个应用程序通常作为单个单位部署。如果此类应用程序需要横向扩展，通常需要在多个服务器或虚拟机中复制整个应用程序。

## 一体式应用程序

应用程序体系结构项目可能的最小数量是一。在这种体系结构中，应用程序的完整逻辑包含在单一项目中，编译为单一程序集并作为单个单元进行部署。

一个新的 ASP.NET Core 项目，不管是在 Visual Studio 中还是通过命令行创建，最初都是简单的“一体式”整体应用程序。它包含应用程序的所有行为，包括展现、业务和数据访问逻辑。图 5-1 展示了单项目应用的文件结构。

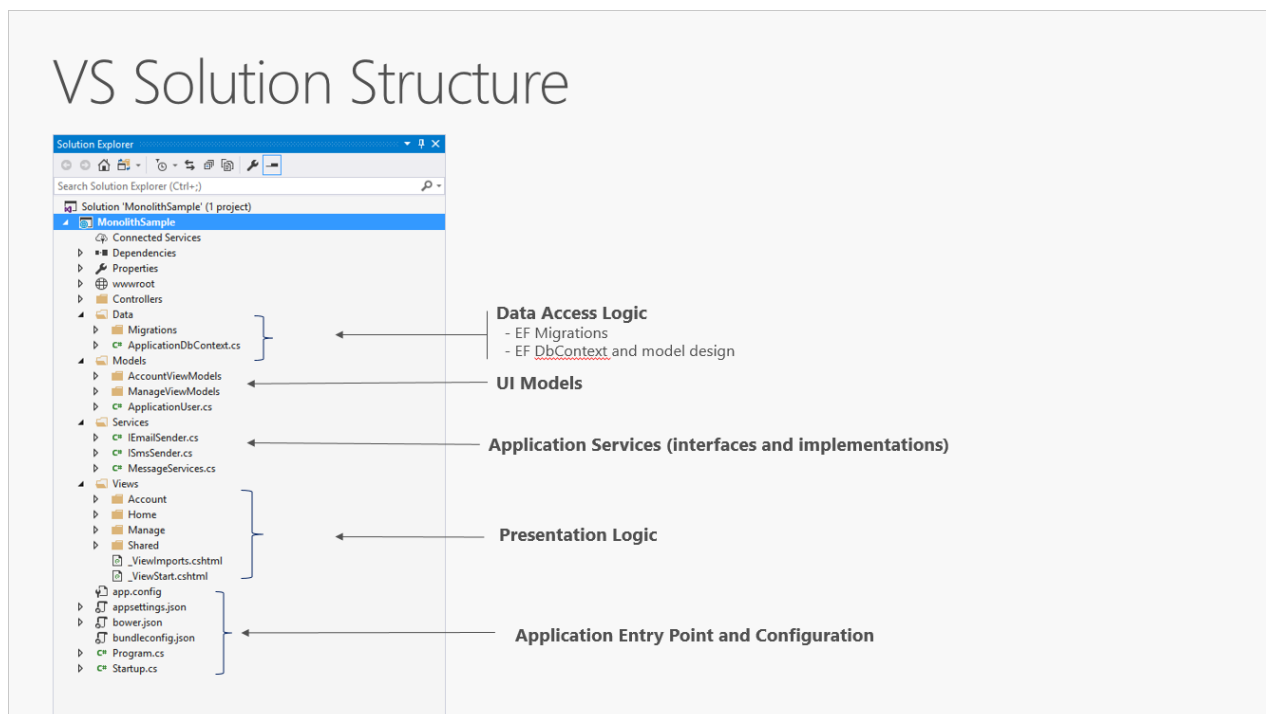


图 5-1。单项目 ASP.NET Core 应用。

在单项目方案中，通过使用文件夹实现关注点分离。默认模板包括单独的文件夹，对应于 MVC 模式中的模型、视图和控制器，以及其他数据和服务文件夹。在这种结构安排中，应尽可能地将展现逻辑限制在“Views”文件夹，将数据访问逻辑限制在“Data”文件夹中保存的类。业务逻辑应位于“Models”文件夹内的服务和类中。

尽管简单，但单项目整体解决方案也有一些缺点。随着项目的大小和复杂性增加，文件和文件夹数量也会继续随之增加。用户界面 (UI) 问题 (模型、视图和控制器) 驻留于多个文件夹中，这些文件夹未按字母顺序组合在一起。将其



他 UI 级别的构造(例如筛选器或 ModelBinder)添加到它们自己的文件夹时,问题只会变得更糟。业务逻辑分散于“Models”和“Services”文件夹之间,没有明确地指示哪些文件中的哪些类应当依赖其他类。这种项目级别缺少组织的情况通常会导致[面条式代码](#)。

为解决这些问题,应用程序通常演变为多项目解决方案,其中将每个项目视为位于应用程序的特定层。

## 什么是层次?

随着应用程序的复杂性增加,管理复杂性的方式之一是根据职责或问题分解应用程序。这遵循关注点分离原则,有助于使基本代码井然有序,以便开发人员可轻松找到实现特定功能的位置。然而,分层体系结构提供的好处远远不止于组织代码结构。

通过将代码分层排列,常见的低级功能可在整个应用程序中重复使用。这种重复使用很有用,因为它意味着需要编写的代码变少,还因为它可以让应用程序能够对某个实现进行标准化,从而遵循[不要自我重复 \(DRY\)](#) 原则。

借助分层体系结构,应用程序可以强制实施有关哪些层可以与其他层通信的限制。这有助于实现封装。某层发生更改或更换时,只有与它一起工作的那些层会受到影响。通过限制哪些层依赖其他层,可缓解更改的影响,使一项更改不会影响整个应用程序。

分层(和封装)让替换应用程序内的功能变得更加轻松。例如,应用程序最初可能使用自己的 SQL Server 数据库来实现持久性,但稍后可能选择使用基于云的持久性策略,或 Web API 后的策略。如果应用程序将其持久性实现正确封装于逻辑层中,则可使用实现相同公共接口的新的 SQL Server 特定层替换它。

除可能的交换实现,以应对将来的要求更改之外,应用程序层还能让测试用途的交换实现变得更加轻松。无需编写针对应用程序的真实数据层或 UI 层操作的测试,可在测试时使用提供请求的已知响应的假实现来替换这些层。通常情况下,与针对应用的实际基础结构运行测试相比,这可以降低测试的编写难度,并提高测试的运行速度。

逻辑分层是用于改进企业软件应用程序代码的常用技术,可通过多种方式将代码分层排列。

### NOTE

层次表示应用程序内的逻辑分隔。如果应用程序逻辑以物理方式分布到单独的服务器或进程中,这些单独的物理部署目标就称为“层级”。具有部署到单一层级的 N 层应用程序是可能的,也很常见。

## 传统“N 层”体系结构应用程序

图 5-2 展示了应用程序逻辑分层最常用的组织结构。

# Application Layers

User Interface

Business Logic

Data Access

图 5-2。典型的应用程序层次。

这些层经常简称为 UI、BLL (业务逻辑层) 和 DAL (数据访问层)。使用此体系结构, 用户可通过 UI 层 (仅与 BLL 交互) 提出请求。反过来, BLL 可为数据访问请求调用 DAL。UI 层不应直接向 DAL 提出任何请求, 也不得通过其他途径直接与持久性发生交互。同样, BLL 应仅通过 DAL 与持久性发生交互。通过这种方式, 每层都有自己熟知的职责。

这种传统分层方法的缺点之一是编译时依赖关系由上而下运行。即, UI 层依赖于 BLL, 而 BLL 依赖于 DAL。这意味着, 通常保存应用程序中最重要逻辑的 BLL 层, 必须依赖于数据访问的实现方式 (且通常依赖于数据库的存在)。在这样的体系结构中很难测试业务逻辑, 需要一个测试数据库。如下节中所述, 依赖倒置原则可以用来解决此问题。

图 5-3 展示了一个示例解决方案, 其中按职责 (层次) 将应用程序分解为三个项目。

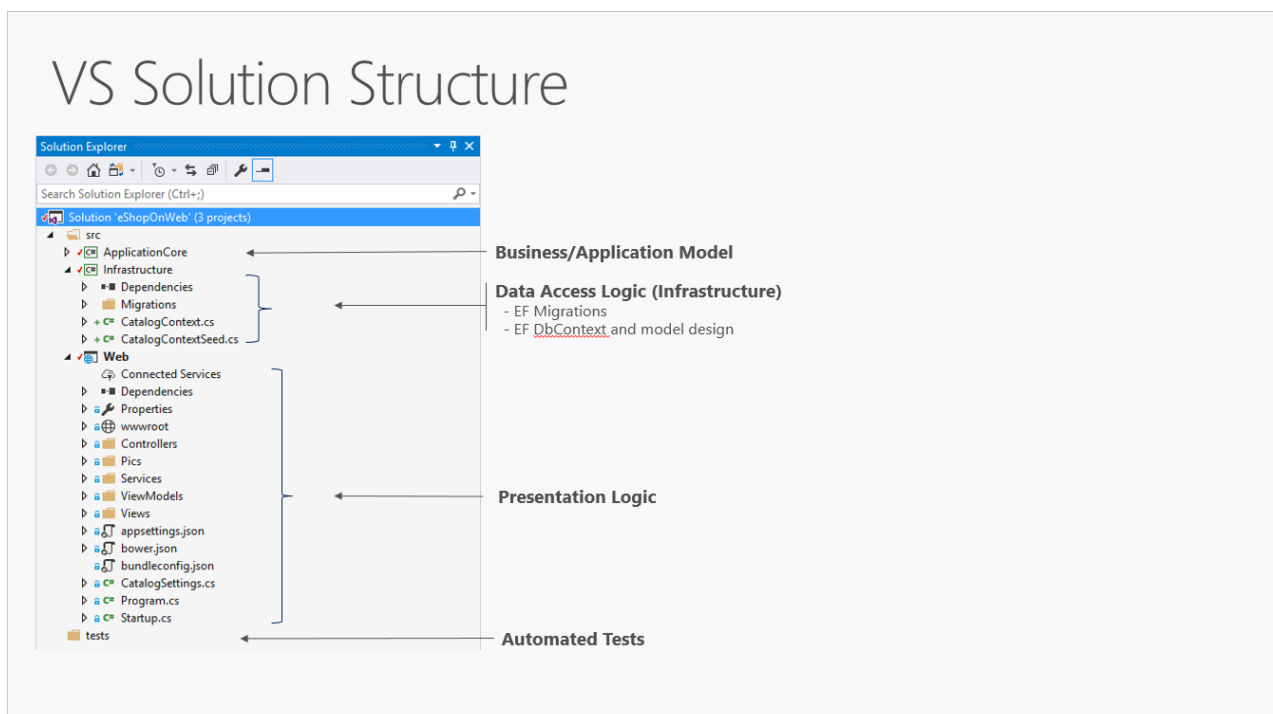


图 5-3。具有三个项目的简单整体式应用程序。

尽管出于组织架构目的, 此应用程序使用多个项目, 但它仍作为单一单位进行部署, 且其客户端以单一 Web 应用

的形式与其交互。这使部署过程变得非常简单。图 5-4 展示了如何使用 Azure 托管此类应用。

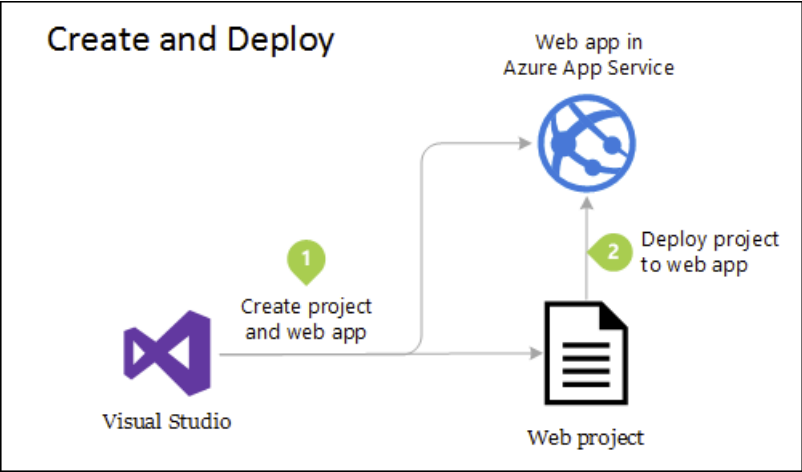


图 5-4。Azure Web 应用简单部署

随着应用程序需求增长，可能需要更复杂、更可靠的部署解决方案。图 5-5 展示了支持其他功能、更复杂的部署计划示例。

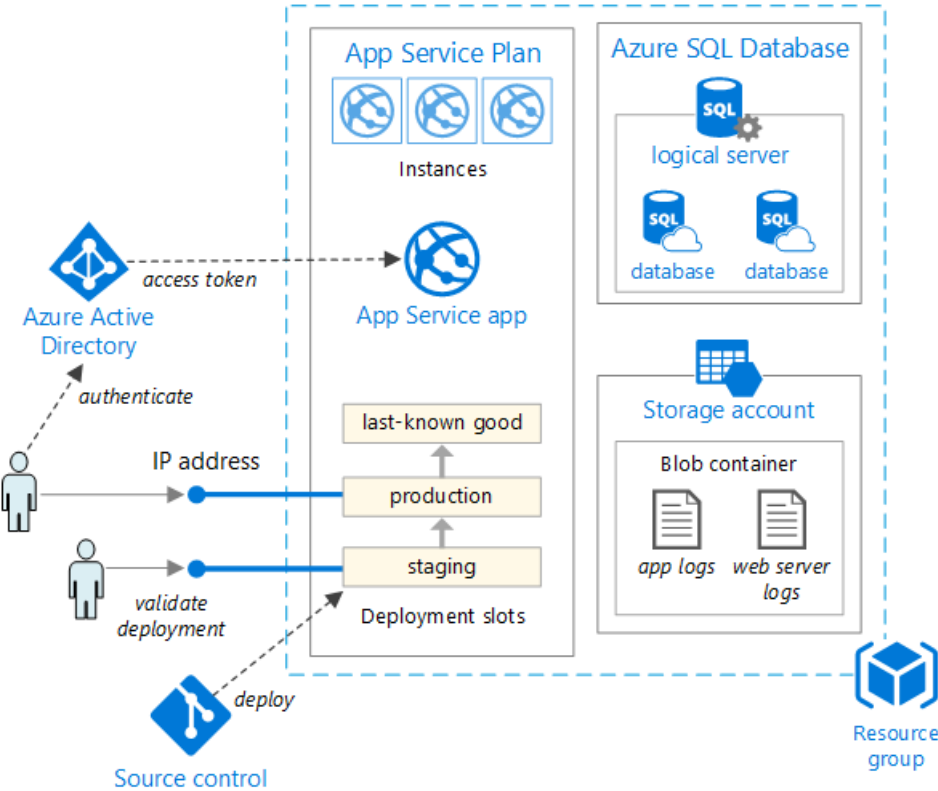
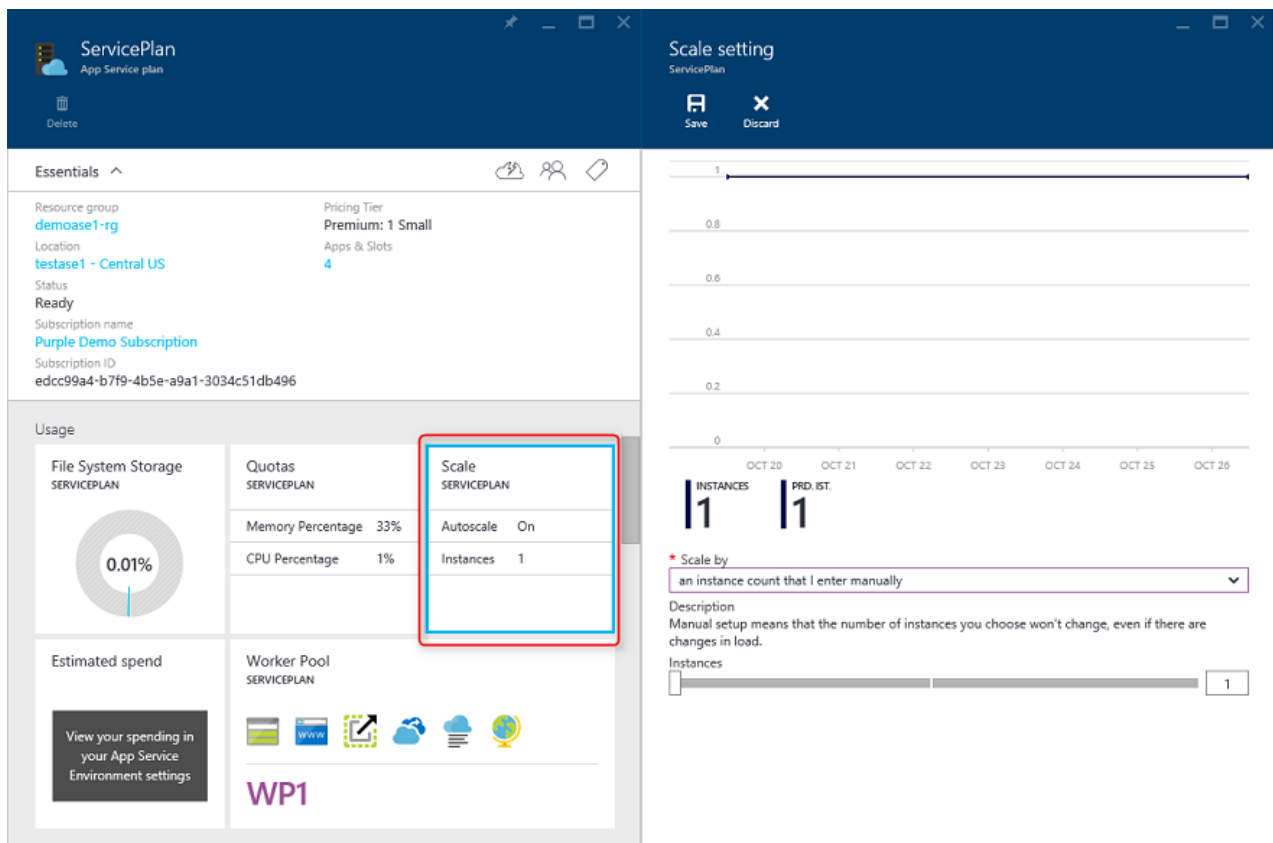


图 5-5。将 Web 应用部署到 Azure 应用服务

在内部，此项目的组织根据职责分为多个项目，提高了应用程序的可维护性。

可纵向或横向扩展此单位，以利用基于云的按需可伸缩性。纵向扩展指的是向承载应用的服务器添加额外的 CPU、内存、磁盘空间或其他资源。横向扩展指的是添加此类服务器的其他实例，无论它们属于物理服务器、虚拟机还是容器。在多个实例中承载应用时，可以使用负载均衡器来将请求分配给各个应用实例。

在 Azure 中缩放 Web 应用程序最简单的方法是在应用程序的应用服务计划中手动配置缩放。图 5-6 展示用于配置为应用提供服务的实例数量的相应 Azure 仪表板屏幕。



如 5-6。Azure 中的应用服务计划缩放。

## 干净体系结构

遵循依赖倒置原则以及域驱动设计原则 (DDD) 的应用程序倾向于达到类似的体系结构。多年来, 这种体系结构有多种名称。最初名称之一是六边形体系结构, 然后是端口 - 适配器。最近, 它被称为**洋葱体系结构**或**干净体系结构**。此电子书中将后一种名称“干净体系结构”用作此体系结构的名称。

eShopOnWeb 参考应用程序使用“干净体系结构”方法将其代码组织到项目中。可以在 [ardalis/cleanarchitecture](https://github.com/ardalis/cleanarchitecture) GitHub 存储库上找到一个解决方案模板, 用作自己的 ASP.NET Core 的起点。

干净体系结构将业务逻辑和应用程序模型置于应用程序的中心。而不是让业务逻辑依赖于数据访问或其他基础设施, 此依赖关系被倒置: 基础结构和实现细节依赖于应用程序内核。这是通过在应用程序核心中定义抽象或接口来实现的, 然后通过基础设施层中定义的类型实现。将此体系结构可视化的常用方法是使用一系列同心圆, 类似于洋葱。图 5-7 展示这种样式的体系结构表示形式的示例。

# Clean Architecture Layers (Onion view)

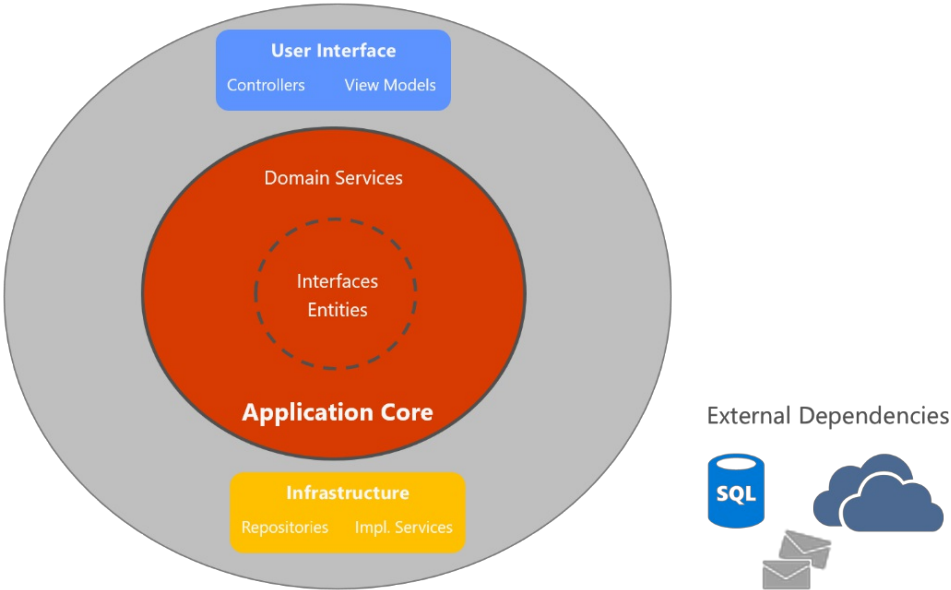


图 5-7。干净体系结构，洋葱视图

在此关系图中，依赖关系流向最里面的圆。“应用程序内核”因其位于此关系图的核心位置而得名。从关系图上可见，该应用程序内核在其他应用程序层上没有任何依赖项。应用程序的实体和接口位于正中心。在外圈但仍在应用程序核心的是域服务，它通常实现内圈中定义的接口。在应用程序内核外面，UI 和基础结构层均依赖于应用程序内核，但不一定彼此依赖。

图 5-8 展示了可更好地反映 UI 和其他层之间的依赖关系的更传统的水平层次关系图。

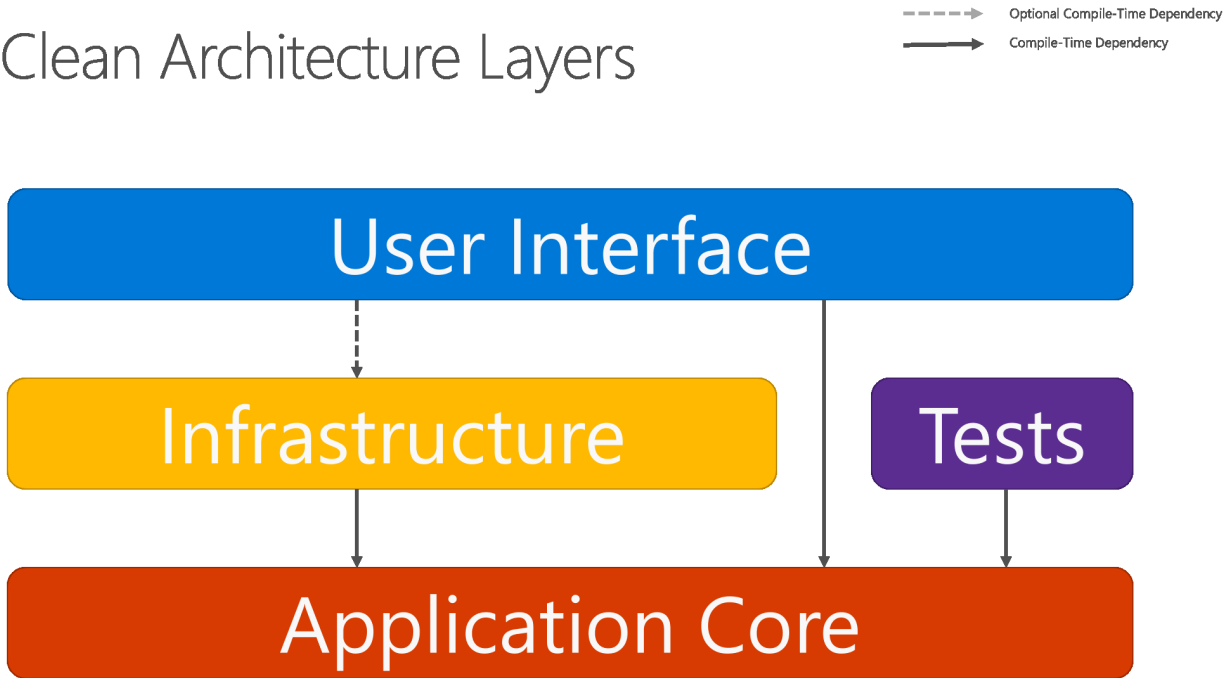


图 5-8。干净体系结构，水平层次视图

注意，实线箭头表示编译时依赖关系，而虚线箭头表示仅运行时依赖关系。使用干净体系结构，UI 层可使用编译时应用程序内核中定义的接口，理想情况下不应知道体系结构层中实现的实现类型。但是在运行时，这些实现类型是应用执行所必需的，因此它们需要存在并通过依赖关系注入接通应用程序内核接口。

图 5-9 展示了遵循这些建议生成 ASP.NET Core 应用程序体系结构时的更详细的视图。

# ASP.NET Core Architecture

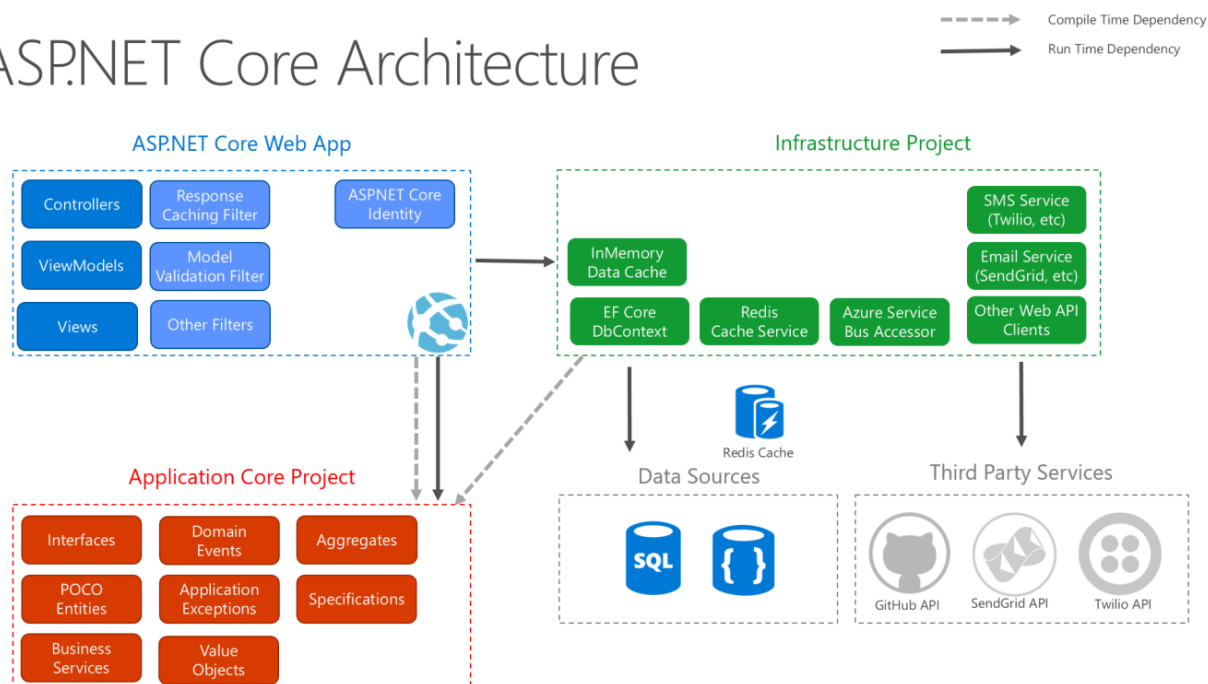


图 5-9。遵循干净体系结构的 ASP.NET Core 体系结构关系图。

由于应用程序内核不依赖于基础结构，可轻松为此层次编写自动化单元测试。图 5-10 和 5-11 展示了测试如何适应此体系结构。

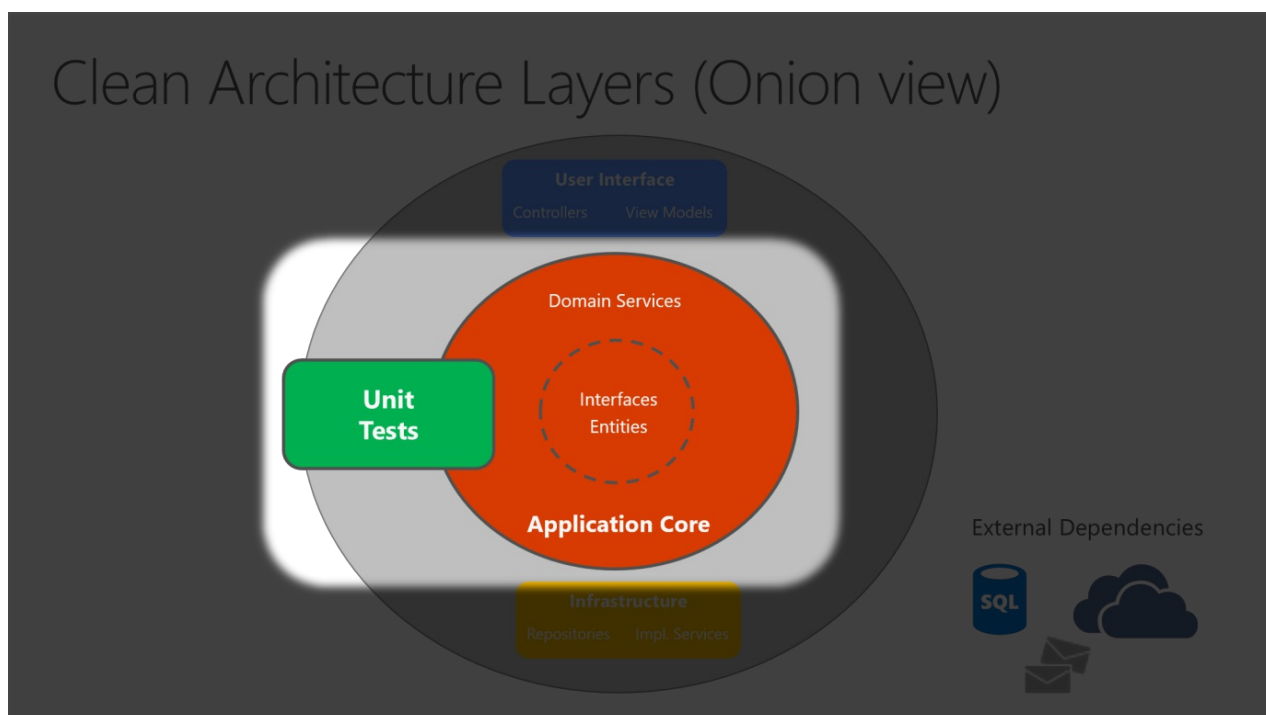


图 5-10。隔离状态下的单元测试应用程序内核。

# Clean Architecture Layers (Onion view)

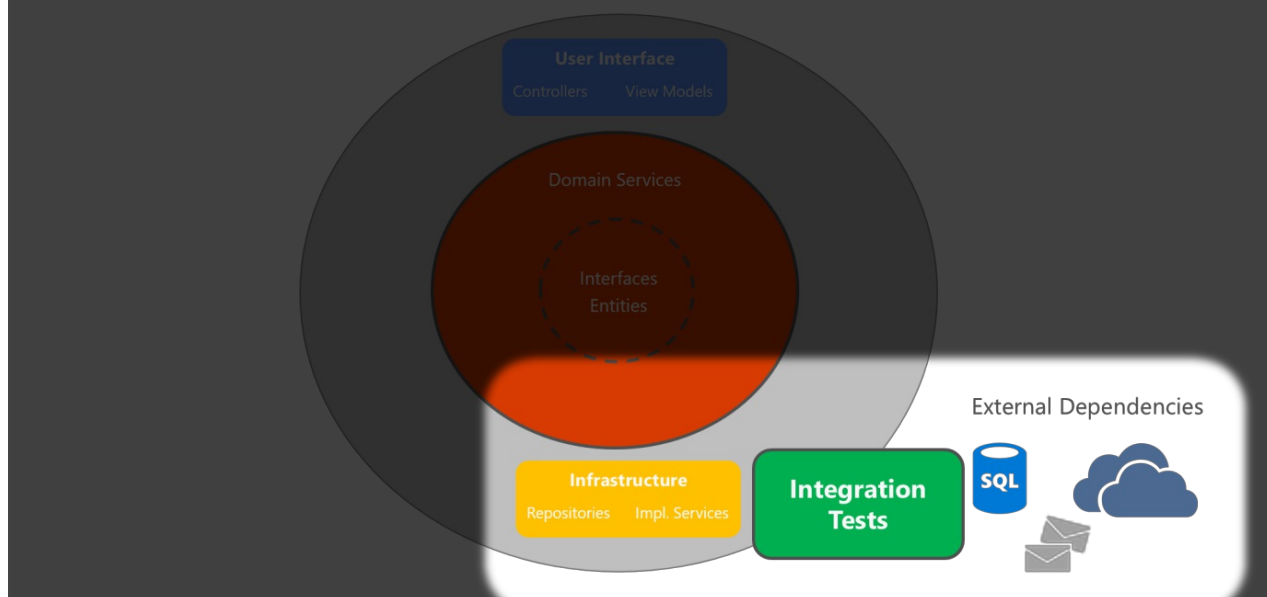


图 5-11。使用外部依赖关系的集成测试基础结构实现。

由于 UI 层对基础结构项目中定义的类型没有任何直接依赖关系，同样，可轻松交换实现，无论是为便于测试还是为应对不断变化的应用程序要求。ASP.NET Core 对内置依赖关系注入的使用及相关支持使此体系结构最适合用于构造重要的整体式应用程序。

对于整体式应用程序，应用程序内核、基础结构和 UI 项目均作为单一应用程序运行。运行时应用程序体系结构可能类似于图 5-12。

## ASP.NET Core Architecture

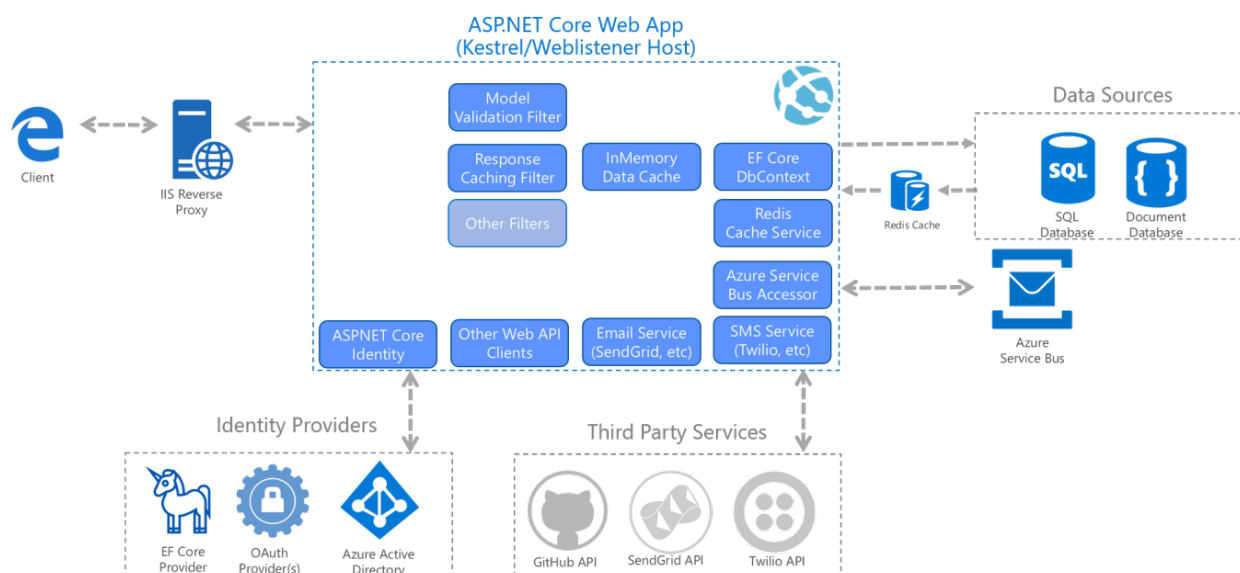


图 5-12。示例 ASP.NET Core 应用的运行时体系结构。

### 采用干净体系结构排列代码

在干净体系结构解决方案中，每个项目都有明确的职责。在这种情况下，某些类型将属于每个项目，你会经常相应的项目中找到与这些类型相应的文件夹。

应用程序内核包含业务模型，后者包括实体、服务和接口。这些接口包括使用基础结构执行的操作（如数据访问、文件系统访问和网络调用等）的抽象。有时，在此层定义的服务或接口需要使用与 UI 或基础结构没有任何依赖关系



的非实体类型。这些类型可定义为简单的数据传输对象 (DTO)。

### 应用程序内核类型

- 实体(保存的业务模型类)
- 接口
- Services
- DTO

基础结构项目通常包括数据访问实现。在典型的 ASP.NET Core Web 应用程序中, 这些实现包括 Entity Framework (EF) DbContext、任何已定义的 EF Core `Migration` 对象以及数据访问实现类。提取数据访问实现代码最常用的方式是通过使用[存储库设计模式](#)。

除数据访问实现外, 基础结构项目还应包含必须与基础结构问题交互的服务的实现。这些服务应实现应用程序内核中定义的接口, 因此基础结构应包含对应用程序内核项目的引用。

### 基础结构类型

- EF Core 类型( `DbContext`、`Migration` )
- 数据访问实现类型(存储库)
- 特定于基础结构的服务(如 `FileLogger` 或 `SmtpNotifier` )

ASP.NET Core MVC 应用程序中的用户界面层是应用程序的入口点。此项目应引用应用程序内核项目, 且其类型应严格通过应用程序内核中定义的接口与基础结构进行交互。UI 层中不允许基础结构层类型的直接实例化(或静态调用)。

### UI 层类型

- Controllers
- 筛选器
- 视图
- ViewModels
- 启动

启动类负责配置应用程序, 并将实现类型与接口接通, 使依赖关系在运行时可正常工作。

#### NOTE

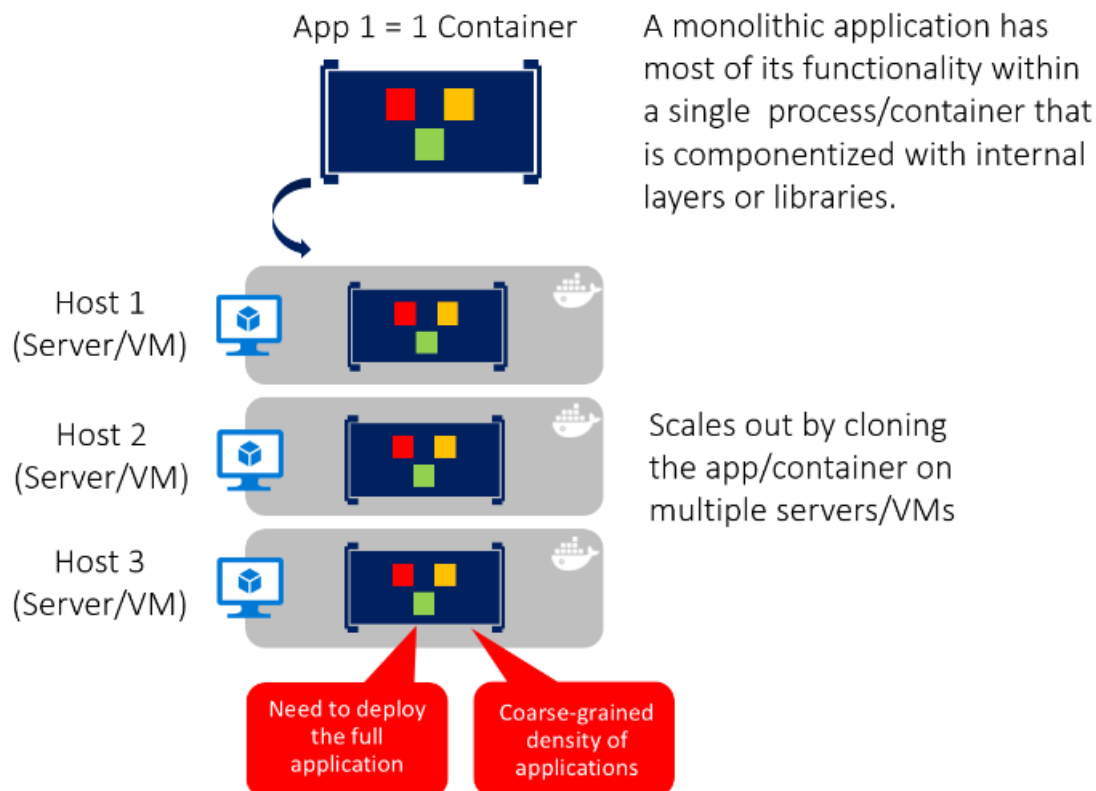
为在 UI 项目的 Startup.cs 文件的 ConfigureServices 中接通依赖关系注入, 项目可能需要引用基础结构项目。可通过使用自定义 DI 容器(最轻松的方式)消除此依赖关系。对于本示例, 最简单的方式是允许 UI 项目引用基础结构项目。

## 整体式应用程序和容器

可以构建基于单个和整体部署的 Web 应用程序或服务, 并将其部署为容器。在应用程序内, 它可能不是一个整体, 而是排列在若干个库、组件或层中。但在外部, 它是单个容器, 如单个进程、单个 Web 应用程序或单个服务。

若要管理此模型, 可部署单个容器来表示应用程序。若要进行缩放, 只需添加更多副本, 并将负载均衡器置于前面即可。为了简单起见, 在单个容器或 VM 中管理单个部署。

# Monolithic Containerized application



如图 5-13 中所示，可以在每个容器内添加多个组件/库或内部层。但是，遵循容器原则（“一个容器在一个进程中做一件事”），整体模式可能成为冲突。

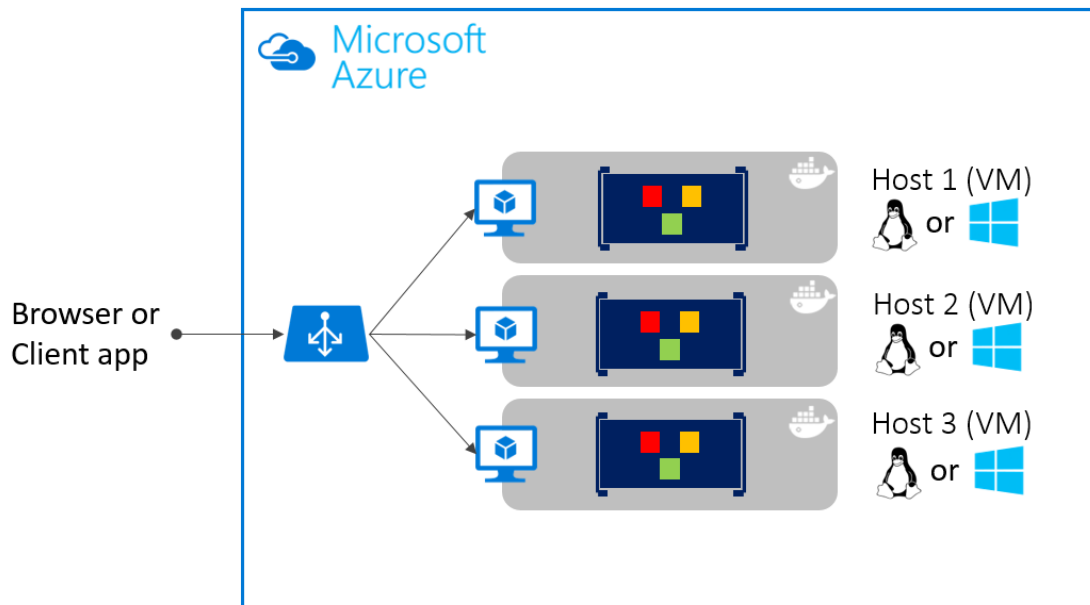
这种方法的缺点是应用程序增长时，需要将它进行缩放。如果整个应用程序都已缩放，这就不是问题了。但在大多数情况下，应用程序中只有一小部分是瓶颈，需要进行缩放，而其他组件使用较少。

在典型的电子商务示例中，可能需要缩放产品信息组件。众多客户浏览产品，但并不购买它们。使用购物车的顾客比使用付款管道的多。较少的顾客会评论或查看购买记录。而且你可能只需要少量的员工（在一个区域内）管理货物和营销活动。通过缩放整体式设计，可多次部署所有代码。

除了“缩放所有组件”问题外，更改单个组件还需要完全重新测试整个应用程序，以及完全重新部署所有实例。

整体式方法很常见，并且许多组织均使用此体系结构方法进行开发。其中许多组织取得了足够好的成果，而其他组织已达到极限。许多组织使用这种模型设计应用程序，因为工具和基础结构难以构建面向服务的体系结构（SOA），而且在应用程序增长之前他们也没有发现这种需要。如果发现已达到整体式方法的极限，请分解应用，使其可更好地利用可能作为下一个逻辑步骤的容器和微服务。

# Architecture in Docker infrastructure for monolithic applications



在 Microsoft Azure 中部署整体式应用程序可以通过使用每个实例的专用 VM 实现。使用 [Azure 虚拟机规模集](#) 可轻松缩放 VM。[Azure 应用服务](#) 可运行整体式应用程序并轻松缩放实例，无需管理 VM。Azure 应用服务还可运行 Docker 容器的单个实例，从而简化部署。通过使用 Docker，可将单个 VM 部署为 Docker 主机，并运行多个实例。如图 5-14 所示，使用 Azure 均衡器可管理缩放。

使用传统的部署技术可以管理各种主机的部署。通过 `docker run` 等命令可以手动管理 Docker 主机，也可以通过持续交付 (CD) 管道等自动化管理。

## 部署为容器的整体式应用程序

使用容器管理整体式应用程序部署有很多好处。缩放容器实例比部署额外的 VM 要快得多，也容易得多。即便使用虚拟机规模集缩放 VM，也需要时间才能到达实例。部署为应用实例时，应用的配置将作为 VM 的一部分进行管理。

将更新部署为 Docker 映像会快得多，并且网络效率更高。Docker 映像通常会在几秒内启动，加快了推出速度。拆除 Docker 实例与发出 `docker stop` 命令一样简单，通常在一秒钟以内便可完成。

正如容器从设计上来说，它的本质就是不可变的，因此你无需担心 VM 损坏，而更新脚本可能忘记考虑磁盘上剩下的某些特定配置或文件。

对于简单 Web 应用程序的单片式部署，可以使用 Docker 容器。这样可以改进持续集成和持续部署管道，并有助于成功实现部署到生产环境。不再出现“为什么可以在我的计算机上正常运行，却不能在生产环境中正常运行？”的问题

基于微服务的体系结构具有许多优势，但以增加复杂性为代价。在某些情况下，付出的代价会比得到的优势更为重大，因此在单个或少量容器中运行的单片式部署应用程序是更好的选择。

单片式应用程序可能不易分解到独立性良好的微服务。微服务应彼此独立地运行，以提供恢复能力更强的应用程序。如果无法实现应用程序的独立功能切片，将其分离只会增加复杂性。

应用程序可能尚不需要独立地扩展功能。许多应用程序在需要扩展到超出单个实例时，可以通过克隆该整个实例这一相对简单的过程来实现此目的。将应用程序分离成各自分散的服务不仅需增加额外工作量，且收效甚微，相比之下，缩放应用程序的完整实例则既简单又节约成本。

在应用程序开发前期，可能并不确定自然功能边界。开发最小可独立产品时，自然分离可能尚未出现。其中一些条件可能是临时的。可首先创建单片式应用程序，稍后再分离要以微服务的形式开发和部署的某些功能。而另一些条件可能对应用程序的容错空间至关重要，这意味着应用程序可能永远无法分解到多个微服务。

将应用程序分离到多个离散进程还会带来开销。将功能分离到不同的进程则更复杂。通信协议会变得更加复杂。在服务之间必须使用异步通信，而不得使用方法调用。移动到微服务体系结构时，需要添加许多在 eShopOnContainers 应用程序的微服务版本中实现的构建基块：事件总线处理、消息恢复和重试、最终一致性等。

较为简单的 [eShopOnWeb 参考应用程序](#) 支持单容器整体化容器应用。该应用程序包括一个 Web 应用程序，其中包括传统的 MVC 视图、Web API 和 Razor Pages。可使用 `docker-compose build` 和 `docker-compose up` 命令从解决方案根目录启动该应用程序。此命令使用 web 项目根目录中的 `Dockerfile` 为 Web 实例配置容器，并在指定端口上运行容器。可从 GitHub 下载此应用程序的源代码，并在本地运行。即使是这一单片式应用程序，在容器环境中部署也是有益的。

其一，容器化部署意味着应用程序的每个实例都在同一环境中运行。这包括用于前期测试和开发的开发人员环境。开发团队可在与生产环境完全相同的容器化环境中运行应用程序。

此外，容器化应用程序横向扩展成本较低。使用容器环境比使用传统 VM 环境更有利于资源共享。

最后，容器化应用程序会强制分离业务逻辑和存储服务器。应用程序横向扩展时，多个容器将全部依赖于单个物理存储介质。此存储介质通常是运行 SQL Server 数据库的高可用性服务器。

## Docker 支持

`eShopOnWeb` 项目在 .NET Core 上运行。因此，该项目可以在基于 Linux 或 Windows 的容器中运行。请注意，在 Docker 部署中，请对 SQL Server 使用相同的主机类型。基于 Linux 的容器占用较小，是首选方案。

可使用 Visual Studio 2017 或更高版本向现有应用程序添加 Docker 支持：右键单击“解决方案资源管理器”中的一个项目，然后选择“添加”>“Docker 支持”。此操作可添加所需文件并修改项目以使用这些文件。当前的 `eShopOnWeb` 示例中已具有这些文件。

解决方案级别 `docker-compose.yml` 文件包含有关要生成的映像和要启动的容器的信息。该文件可让你使用 `docker-compose` 命令来同时启动多个应用程序。在本示例中，它只启动 Web 项目。还可使用该文件配置依赖项，例如单独的数据库容器。

```
version: '3'

services:
  eshopwebmvc:
    image: eshopwebmvc
    build:
      context: .
      dockerfile: src/Web/Dockerfile
    environment:
      - ASPNETCORE_ENVIRONMENT=Development
    ports:
      - "5106:5106"

networks:
  default:
    external:
      name: nat
```

`docker-compose.yml` 文件引用 `Web` 项目中的 `Dockerfile`。`Dockerfile` 用于指定将要使用的基容器以及在该容器上配置应用程序的方式。`Web` `Dockerfile`：

```
FROM mcr.microsoft.com/dotnet/core/sdk:3.1 AS build
WORKDIR /app

COPY *.sln .
COPY . .
WORKDIR /app/src/Web
RUN dotnet restore

RUN dotnet publish -c Release -o out

FROM mcr.microsoft.com/dotnet/core/aspnet:3.1 AS runtime
WORKDIR /app
COPY --from=build /app/src/Web/out ./

ENTRYPOINT ["dotnet", "Web.dll"]
```

## Docker 问题疑难解答

运行容器化应用程序后，它会持续运行，直到将其停止。可使用 `docker ps` 命令查看正在运行的容器。可通过使用 `docker stop` 命令并指定容器 ID 来停止正在运行的容器。

请注意，正在运行的 Docker 容器可能已绑定到其他可能在开发环境中尝试使用的端口。如果尝试使用与运行 Docker 容器相同的端口来运行或调试应用程序，将收到指示服务器无法绑定到该端口的错误。再次强调，停止容器应可解决该问题。

如果要使用 Visual Studio 向应用程序添加 Docker 支持，请确保执行此操作时 Docker Desktop 处于运行状态。如果启动向导时 Docker Desktop 未处于运行状态，则向导无法正常运行。此外，向导会检查当前的容器选择，添加正确的 Docker 支持。若要为 Windows 容器添加支持，需在配置了 Windows 容器的 Docker Desktop 运行时运行向导。若要为 Linux 容器添加支持，则运行向导，同时运行配置有 Linux 容器的 Docker。

## 参考 - 常见 Web 体系结构

- 干净体系结构  
<https://8thlight.com/blog/uncle-bob/2012/08/13/the-clean-architecture.html>
- 洋葱体系结构  
<https://jeffreypalermo.com/blog/the-onion-architecture-part-1/>
- 存储库模式  
<https://deviq.com/repository-pattern/>
- 干净体系结构解决方案模板  
<https://github.com/ardalis/cleanarchitecture>
- 构建微服务电子书  
<https://aka.ms/MicroservicesEbook>
- DDD (域驱动设计)  
<https://docs.microsoft.com/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/>

[上一页](#)[下一页](#)

# 常用客户端 Web 技术

2020/2/27 • [Edit Online](#)

“优秀的网站应表里如一。”

- Paul Cookson

ASP.NET Core 应用程序属于 Web 应用程序，并且通常依赖于 HTML、CSS 和 JavaScript 等客户端 Web 技术。通过将页面 (HTML) 内容从其布局和样式 (CSS) 以及行为 (JavaScript) 中分离出来，复杂的 Web 应用也可以利用“关注分离”原则。将来，当这些关注点不相互交织时，可以更轻松地对应用程序的结构、设计或行为进行更改。

尽管 HTML 和 CSS 相对稳定，但应用程序框架和实用程序开发人员用于生成基于 Web 的应用程序的 JavaScript，正以惊人的速度发展。本章介绍 Web 开发人员使用 JavaScript 的几种方式，并提供 Angular 和 React 客户端库的简要概述。

## NOTE

Blazor 提供了 JavaScript 框架的替代方法，用于生成丰富的交互式客户端用户界面。客户端 Blazor 支持仍处于预览状态，因此目前不在本章的范围之内。

## HTML

HTML 是标准标记语言，用于创建网页和 Web 应用程序。它的元素组成了网页的构建基块，表示格式化文本、图像、窗体输入和其他结构。浏览器发出 URL 请求时，无论提取网页还是应用程序，首先需要返回 HTML 文档。这个 HTML 文档可能引用或包括关于其外观、CSS 形式的布局或 JavaScript 形式的行为的详细信息。

## CSS

CSS (级联样式表) 用于控制 HTML 元素的外观和布局。CSS 样式可直接应用于 HTML 元素，单独对相同页面进行定义，或在单独文件中定义或由页面引用。样式根据用于选择给定 HTML 元素的方式进行级联。例如，样式可应用于整个文档，但会由应用于特定元素的样式覆盖。同样，特定于元素的样式会由应用于 CSS 类的样式 (曾应用于该元素) 覆盖，后者反过来会由指向该元素的特定实例 (通过其 ID) 的样式覆盖。图 6-1

# CSS Specificity

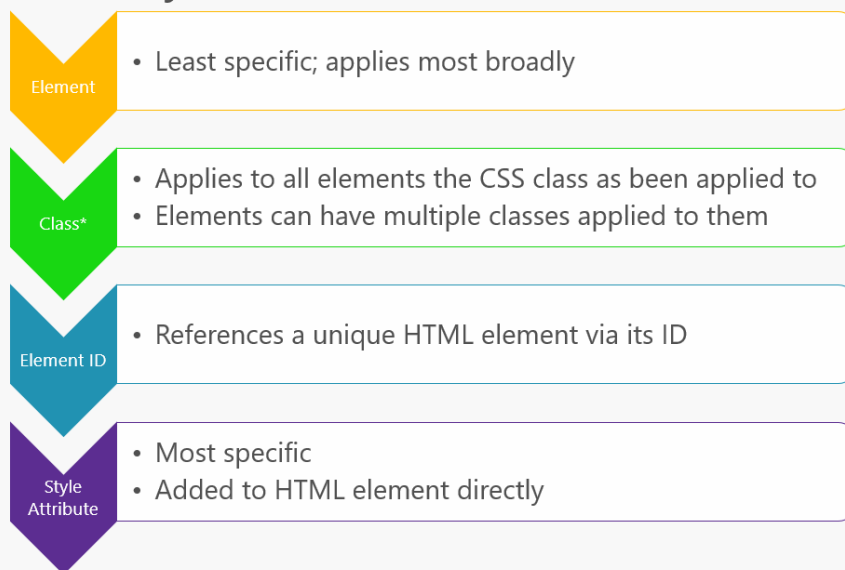


图 6-1。按顺序排列的 CSS 特殊性规则。

最好将样式保存在单独的样式表文件中，并使用基于选择的级联在应用程序内实现一致且可重用的样式。应避免将样式规则置于 HTML 中，但将样式（而不是规则）应用于特定的各个元素（不是元素的整个类，也不是应用特定 CSS 类的元素）除外。

## CSS 预处理器

CSS 样式表缺少对条件逻辑、变量以及其他编程语言功能的支持。因此，大型样式表通常包含很多重复，如将相同的颜色、字体或其他设置应用于 HTML 元素和 CSS 类的多个不同的变体。通过添加对变量和逻辑的支持，CSS 预处理器可帮助样式表遵循 **DRY 原则**。

最常用的 CSS 预处理器是 Sass 和 LESS。两者均可扩展 CSS 并与其向后兼容，表示无格式 CSS 文件即为有效的 Sass 或 LESS 文件。Sass 基于 Ruby，而 LESS 基于 JavaScript，二者通常作为本地部署过程的一部分运行。两者都提供命令行工具，以及 Visual Studio 中的内置支持，可使用 Gulp 或 Grunt 任务运行它们。

## JavaScript

JavaScript 是动态的解释性编程语言，已在 ECMAScript 语言规范中进行标准化。它是 Web 的编程语言。与 CSS 一样，JavaScript 可定义为 HTML 元素中的属性，作为网页或单独文件中的脚本基块。正如 CSS，建议将 JavaScript 组织到单独的文件中，并尽可能将其与各个网页或应用程序视图中找到的 HTML 分开保存。

在 Web 应用程序中使用 JavaScript 时，通常需要执行以下几项任务：

- 选择 HTML 元素并检索和/或更新其值。
- 查询 Web API 以获取数据。
- 向 Web API 发送命令（并使用其结果响应回调）。
- 执行验证。

可以仅使用 JavaScript 执行上述所有任务，但存在的许多库可简化这些任务。其中一个，也是最成功的库是 jQuery，仍是在网页上简化这些任务的热门选择。对于单页应用程序 (SPA)，jQuery 不会提供 Angular 和 React 提供的众多所需功能。

## 使用 jQuery 的旧版 Web 应用

尽管 jQuery 基于古老的 JavaScript 框架标准，但它仍然是常用的库，用于处理 HTML/CSS 和生成对 Web API 进



行 AJAX 调用的应用程序。但是, jQuery 按浏览器文档对象模型 (DOM) 级别操作, 并默认只提供命令性, 而不是声明性的模型。

例如, 假设一个文本框的值超过 10, 则应使页面上的元素可见。在 jQuery 中, 此操作通常通过使用检查文本框的值, 并根据该值设置目标元素可见性的代码编写事件处理程序来实现。这是一种基于代码的命令性方法。另一种框架可能转而使用数据绑定, 以声明的方式将元素的可见性与文本框值绑定。这种方法不需要编写任何代码, 改为只需要修饰数据绑定属性涉及的元素。随着客户端行为变得更加复杂, 数据绑定方法经常成为更简单的解决方案, 其包含的代码和条件复杂性也较少。

jQuery 与 SPA Framework

因素	JQUERY	ANGULAR
抽象 DOM	是	是
AJAX 支持	是	是
声明性数据绑定	否	是
MVC 样式路由	否	是
模板化	否	是
深层链接路由	否	是

本质上, jQuery 缺少的大多数功能均可通过添加其他库进行添加。但是, SPA 框架(如 Angular)以更集中的方式提供这些功能, 因为它从一开始设计的时候就考虑到了这一点。此外, jQuery 是一种命令性库, 这意味着你需要调用 jQuery 函数才能使用 jQuery 执行任何任务。SPA 框架提供的很多工作和功能都可以声明的方式完成, 无需编写任何实际代码。

数据绑定便是一个很好的示例。在 jQuery 中, 通常只需要一个代码行就能获得 DOM 元素的值或设置某元素的值。但是, 一旦需要更改元素值就需要编写此行代码, 有时这会出现一个页面上的多个函数中。另一常见示例是元素可见性。在 jQuery 中, 可能需要在很多位置编写代码来控制特定元素是否可见。每这两种情况中, 如果使用数据绑定, 便无需编写任何代码。只需将相关元素的值或可见性绑定到页面上的 viewmodel 即可, 对该 viewmodel 的任何更改都会自动反映在绑定的元素中。

Angular SPA

Angular 仍是世界上最常用的一种 JavaScript 框架。从 Angular 2 开始, 团队彻底重建了框架(使用 TypeScript), 并从最初的 AngularJS 名称重新命名为简单的 Angular。经过数年的发展, 重新设计的 Angular 仍是用于生成单页应用程序的可靠框架。

Angular 应用程序基于组件构建。组件通过特殊对象与 HTML 模板进行组合, 并控制页面的一部分。下面是 Angular 文档中的简单组件：

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

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

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

组件使用 @Component 修饰器函数定义, 可接收有关组件的元数据。选择器属性可在显示此组件的页面上标识元素的 ID。模板属性是一个简单的 HTML 模板, 包括最后一行上定义的对应用于组件名称属性的占位符。

通过使用组件和模板，而不是 DOM 元素，Angular 应用可以更高的抽象级别操作，且比单独使用 JavaScript(也称为“vanilla JS”)或 jQuery 编写的应用具有更少的总体代码。Angular 还强加了有关如何组织客户端脚本文件的特定顺序。按照约定，Angular 应用使用常用文件夹结构，同时模块和组件脚本文件位于应用文件夹中。与生成、部署和测试应用相关的 Angular 脚本通常位于更高级别的文件夹中。

可以使用 CLI 开发 Angular 应用。从本地入门 Angular 开发(假设已安装 git 和 npm)包括简单地从 GitHub 中克隆存储库以及运行 `npm install` 和 `npm start`。除此之外，Angular 附带自己的 CLI，可用于创建项目、添加工具，并协助测试、打包和部署任务。这种 CLI 友好性也使 Angular 特别兼容 ASP.NET Core，后者也可提供很好的 CLI 支持。

Microsoft 开发了一个参考应用程序，[eShopOnContainers](#)，其中包含 Angular SPA 实现。此应用包含 Angular 模块，可用于管理在线商店的购物篮，加载和显示其目录中的商品并处理订单创建。可以在 [GitHub](#) 中查看和下载该示例应用程序。

## React

与 Angular 不同，Angular 提供完整的模型 - 视图 - 控制器模式实现，而 React 仅关注视图。它不是一个框架，只是一个库，因此需要利用其他库才能生成 SPA。有许多库旨在与 React 一起使用，以生成丰富的单页应用程序。

React 最重要的特征之一是使用虚拟 DOM。虚拟 DOM 可向 React 提供多项好处，包括性能(虚拟 DOM 可优化实际 DOM 的哪部分需要更新)和可测试性(无需使用浏览器即可测试 React 和它与虚拟 DOM 的交互)。

React 很少使用与 HTML 配合的方式。React 直接在其 JavaScript 代码中以 JSX 的形式添加 HTML，而没有代码和标记(可能引用 HTML 属性中出现的 JavaScript)之间的严格分离。JSX 是类似 HTML 的语法，可向下编译为纯 JavaScript。例如：

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

如果已了解 JavaScript，学习 React 应该很简单。与 Angular 或其他常用库相比，它几乎不涉及众多的学习曲线或特殊语法。

由于 React 不是完整的框架，因此通常需要其他库来处理路由、Web API 调用和依赖关系管理等任务。好处是可为每个任务选择最合适的库，但坏处时需要进行所有决策，并在完成后验证所有选定库是否可以很好地协同工作。如果想要一个好的开端，可使用初学者工具包(如 React Slingshot)，它可使用 React 预先打包一组可兼容库。

## Vue

在其入门指南中，“Vue 是用于生成用户界面的渐进式框架。与其他单一框架不同，Vue 旨在从头开始逐渐采用。核心库仅集中在视图层，易于提取并与其他库或现有项目集成。另一方面，与新式工具和支持库结合使用时，Vue 完全有能力为复杂的单页应用程序提供支持。”

开始使用 Vue 只需要在 HTML 文件中包含其脚本：

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

添加了框架之后，你就可以使用 Vue 的简单模板语法以声明方式将数据呈现到 DOM：

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

然后添加以下脚本：

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

这足以在页面上呈现“Hello Vue!”。但是，请注意，Vue 并不只是将消息呈现给 div 一次。它支持数据绑定和动态更新，因此如果 `message` 的值发生更改，`<div>` 中的值将立即更新以反映更改。

当然，这里仅介绍 Vue 的简单功能。在过去的几年中，它非常受欢迎，并且拥有一个庞大的社区。有一个**庞大且不断增长的支持组件和库列表**，它们可以与 Vue 一起扩展。如果希望将客户端行为添加到 Web 应用程序中，或者正在考虑生成完整的 SPA，那么值得对 Vue 进行一番研究。

### 选择 SPA 框架

考虑哪个 JavaScript 框架最适合支持 SPA 时，请注意以下几项：

- 团队是否熟悉该框架及其依赖关系(某些情况下还包括 TypeScript)？
- 框架的“固执”程度如何，以及是否同意它处理事情的默认方式？
- 它(或伴随库)是否包括应用需要的所有功能？
- 是否有详细的文档记录？
- 它在社区中的活跃程度？新项目是否使用它生成？
- 它的核心团队活跃程度如何？是否定期解决问题和发布新版本？

JavaScript 框架仍以极快的速度发展。使用上面列出的注意事项，可帮助减轻选择之后会后悔依赖的框架的风险。如果你特别不愿意承担风险，请考虑提供商业支持和/或大型企业开发的框架。

### 参考 - 客户端 Web 技术

- **HTML 和 CSS**  
<https://www.w3.org/standards/webdesign/htmlcss>
- **Sass 与 LESS**  
<https://www.keycdn.com/blog/sass-vs-less/>
- **使用 LESS、Sass 和 Font Awesome 为 ASP.NET Core 应用设置样式**  
<https://docs.microsoft.com/aspnet/core/client-side/less-sass-fa>
- **ASP.NET Core 中的客户端开发**  
<https://docs.microsoft.com/aspnet/core/client-side/>
- **jQuery**  
<https://jquery.com/>
- **jQuery 与 AngularJS**  
<https://www.airpair.com/angularjs/posts/jquery-angularjs-comparison-migration-walkthrough>
- **Angular**  
<https://angular.io/>
- **React**  
<https://reactjs.org/>
- **Vue**  
<https://vuejs.org/>
- **Angular、React 与 Vue: 2020 年要选择哪种框架** <https://www.codeinwp.com/blog/angular-vs-vue-vs-react/>
- **2020 年用于前端开发的顶级 JavaScript 框架**  
<https://www.freecodecamp.org/news/complete-guide-for-front-end-developers-javascript-frameworks->

上一頁

下一頁

# 开发 ASP.NET Core MVC 应用

2020/2/27 • [Edit Online](#)

“第一次是否正确完成并不重要。最后一次正确完成才至关重要。”

— Andrew Hunt 和 David Thomas

ASP.NET Core 是一个跨平台的开源框架，用于构建新式云优化 Web 应用程序。ASP.NET Core 具有轻量级和模块化的特点，并且内置了对依赖关系注入的支持，因此具有更好的可测试性和可维护性。而 MVC 支持构建新式 Web API 和基于视图的应用，ASP.NET Core 与之结合后将成为一个功能强大的框架，用于构建企业 Web 应用程序。

## MVC 和 Razor Pages

ASP.NET Core MVC 提供了许多对构建基于 Web 的 API 和应用有用的功能。术语 MVC 代表“模型 - 视图 - 控制器”，这是一种 UI 模式，它将响应用户请求的职责分为几个部分。除了遵循此模式之外，还可以将 ASP.NET Core 应用中的功能实现为 Razor Pages。Razor Pages 内置于 ASP.NET Core MVC 中，并使用相同的功能进行路由、模型绑定等。但是，Razor Pages 不会为控制器、视图等提供单独的文件夹和文件，也不会使用基于属性的路由，而是将它们放置在一个文件夹（“/Pages”）中，根据它们在此文件夹中的相对位置进行路由，并使用处理程序而非控制器操作处理请求。

在创建新的 ASP.NET Core 应用时，应考虑好要构建的应用类型。在 Visual Studio 中，你可以从多个模板中进行选择。三个最常见的项目模板是 Web API、Web 应用程序和 Web 应用程序（模型 - 视图 - 控制器）。虽然只能在首次创建项目时做出此决定，但此决定可以撤销。Web API 项目使用标准的“模型 - 视图 - 控制器”控制器（默认情况下，它只缺少视图）。同样，默认的 Web 应用程序模板使用 Razor Pages，因此也缺少 Views 文件夹。可以稍后向这些项目添加 Views 文件夹以支持基于视图的行为。默认情况下，Web API 和模型 - 视图 - 控制器项目不包含 Pages 文件夹，但可以稍后添加一个以支持基于 Razor Pages 的行为。可以将这三个模板视为支持三种不同类型的默认用户交互：数据（Web API）、基于页面和基于视图。但是，如果你愿意，可以在单个项目中混合和匹配任何或所有这些模板。

### 为何选择 Razor Pages？

Razor Pages 是 Visual Studio 中新 Web 应用程序的默认方法。Razor Pages 提供了一种较为简单的方法来构建基于页面的应用程序功能，例如非 SPA 表单。通过使用控制器和视图，应用程序通常拥有非常大的控制器，这些控制器处理许多不同的依赖项和视图模型，并返回许多不同的视图。这大大增加了复杂性，并且经常导致控制器不能有效地遵循单一责任原则或打开/关闭原则。Razor Pages 通过使用其 Razor 标记在 Web 应用程序中封装给定逻辑“页面”的服务器端逻辑来解决此问题。没有服务器端逻辑的 Razor Page 可以只包含一个 Razor 文件（例如，“Index.cshtml”）。但是，大多数重要的 Razor Pages 都有关联的页面模型类，按照惯例，它的名称与带有“.cs”扩展名的 Razor 文件相同（例如，“Index.cshtml.cs”）。

Razor Page 的页面模型结合了 MVC 控制器和视图模型的职责。不通过控制器操作方法执行请求，而是执行“OnGet()”等页面模型处理程序，默认情况下，呈现其关联页面。Razor Pages 简化了在 ASP.NET Core 应用中构建单个页面的过程，同时仍然提供了 ASP.NET Core MVC 的所有体系结构功能。对于基于页面的新功能，它们是很好的默认选择。

### 何时使用 MVC

如果正在构建 Web API，则 MVC 模式比尝试使用 Razor Pages 更有意义。如果项目将只公开 Web API 终结点，则最好从 Web API 项目模板开始。否则，很容易将控制器和关联的 API 终结点添加到任何 ASP.NET Core 应用中。如果要将现有应用程序从 ASP.NET MVC 5 或更早版本迁移到 ASP.NET Core MVC，并且需要以最少的工作量完成此操作，可使用基于视图的 MVC 方法。完成初始迁移后，可以评估针对新功能甚至批量迁移采用 Razor Pages 是否有意义。

无论是选择使用 Razor Pages 还是 MVC 视图生成 Web 应用，应用都将具有类似的性能，并且都支持依赖项注入、筛选器、模型绑定、验证等。

## 将请求映射到响应

ASP.NET Core 应用的核心在于将传入请求映射到传出响应。较低级别的实现方式是使用中间件，简单的 ASP.NET Core 应用和微服务可能只包含自定义中间件。在某种程度上，使用 ASP.NET Core MVC 可以实现更高级别的操作，需要考虑路由、控制器和操作。每个传入请求都会和应用程序的路由表进行对比，如果找到匹配的路由，则调用关联的操作方法(属于控制器)来处理该请求。如果未找到匹配的路由，则调用错误处理程序(此时返回 NotFound 结果)。

ASP.NET Core MVC 应用可以使用传统路由或属性路由，或二者同时使用。传统路由在代码中定义，使用类似以下示例中的语法指定路由约定：

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

此示例向路由表添加了一个名为“default”的路由。它定义了一个具有 controller、action 和 id 占位符的路由模板。controller 和 action 占位符具有指定的默认值(分别为“Home”和“Index”)，id 占位符则为可选项(通过应用“?”来实现)。此处定义的约定规定，请求的第一部分应与控制器的名称对应，第二部分与操作对应，第三部分(如有)表示 id 参数。通常在同一个位置定义应用程序的传统路由，例如在 Startup 类的 Configure 方法中。

属性路由直接应用到控制器和操作，而不是在全局范围内指定。其优势在于，查看特定方法时，属性路由更容易发现，但也意味着路由信息不会保存在应用程序中的同一个地方。通过属性路由可以为给定操作轻松指定多个路由，并将控制器和操作之间的路由合并在一起。例如：

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

可以在 [HttpGet] 和类似属性上指定路由，而无需添加单独的 [Route] 属性。属性路由还可以通过使用令牌来减少重复控制器或操作名称的次数，如下所示：

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

Razor Pages 不使用属性路由。可以作为 Razor Pages 的 `@page` 指令的一部分为其指定其他路由模板信息：

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

在前面的示例中，问题中的页面将匹配具有整数 `id` 参数的路由。例如，位于 `/Pages` 根目录中的“Products.cshhtml”页面将具有以下路由：



给定请求与路由匹配之后, ASP.NET Core MVC 会对该请求执行**模型绑定**和**模型验证**, 然后才调用操作方法。模型绑定负责将传入到指定 .NET 类型的 HTTP 数据转换为要调用的操作方法的参数。例如, 如果操作方法需要一个 `int id` 参数, 模型绑定将尝试根据请求中提供的值来提供此参数。为此, 模型绑定会查找已发表单中的值、路由中的值和查询字符串值。假设找到了 `id` 值, 该值会被转换为整数, 然后传入操作方法。

模型验证发生在绑定模型之后, 调用操作方法之前。模型验证对模型类型使用可选属性, 且有助于确保提供的模型对象符合特定数据要求。可以将某些值指定为必需项, 将其限制为特定长度, 或将其限制在一定数值范围内, 等等。如果指定了验证特性, 但该模型不符合其要求, 则属性 `ModelState.IsValid` 将为 `false`, 并且失败的验证规则集将可被发送到发出请求的客户端。

使用模型验证时, 执行任何状态更改命令之前, 务必确保模型有效, 以防无效数据损坏应用。使用**筛选器**可避免在每个操作中都需要为此添加代码。ASP.NET Core MVC 筛选器提供了一种拦截成组请求的方法, 因此可以有针对性地应用通用策略和横切关注点。筛选器可应用于单个操作、整个控制器或应用程序全局。

对于 Web API, ASP.NET Core MVC 支持**内容协商**, 允许对指定如何设置响应格式进行请求。根据请求中提供的标头, 操作返回的数据将采用 XML、JSON 或其他支持格式作为响应的格式。借助此功能, 同一个 API 可供数据格式要求不同的多个客户端使用。

Web API 项目应考虑使用 `[ApiController]` 属性, 该属性可应用于单个控制器、基本控制器类或整个程序集。此属性添加自动模型验证检查, 任何具有无效模型的操作都将返回 `BadRequest` 以及验证错误的详细信息。该属性还要求所有操作都具有属性路由, 而不是使用传统路由, 并返回更详细的 `ProblemDetails` 信息以响应错误。

#### 参考 - 将请求映射到响应

- 路由到控制器操作 <https://docs.microsoft.com/aspnet/core/mvc/controllers/routing>
- 模型绑定 <https://docs.microsoft.com/aspnet/core/mvc/models/model-binding>
- 模型验证 <https://docs.microsoft.com/aspnet/core/mvc/models/validation>
- 筛选器 <https://docs.microsoft.com/aspnet/core/mvc/controllers/filters>
- **ApiController** 属性 <https://docs.microsoft.com/aspnet/core/web-api/>

## 处理依赖关系

ASP.NET Core 内置了对**依赖关系注入**技术的支持, 并且在内部使用这一技术。依赖关系注入技术可以在应用程序的不同部分之间实现松散耦合。比较松散的耦合更符合需要, 因为它可以更轻松地将应用程序的某些部分隔离开, 便于进行测试或替换。它还可以降低对应用程序某个部分进行更改会对应用程序中的其他位置产生意外影响的可能性。依赖关系注入的基础是依赖关系反转原则, 并且通常是实现开放/闭合原则的关键。评估应用程序对其依赖关系的处理方式时, 请注意 **static cling** (静态粘附) 这一代码味, 并请记住这句格言: **新增即粘附**。

类调用静态方法或访问静态属性时, 会对基础结构造成负面影响或产生依赖关系, 此时会发生静态粘附。例如, 如果一个方法调用静态方法, 静态方法反过来又写入数据库, 则该方法与该数据库紧密耦合。破坏该数据库调用的任何内容都会破坏该方法。测试此类方法非常困难, 因为此类测试要么需要使用商业模拟库来模拟静态调用, 要么只能使用已有测试数据库进行测试。不依赖于任何基础结构的静态调用, 尤其是完全无状态的静态调用可以进行正常调用, 并且对耦合或可测试性没有任何影响(超越了将代码耦合到静态调用本身)。

许多开发人员知道静态粘附和全局状态的风险, 但仍会通过直接实例化将其代码与具体实现紧密耦合。“新增即粘附”旨在提醒注意这种耦合, 并非一律谴责使用 `new` 关键字。和静态方法调用一样, 没有外部依赖关系的类型的新实例通常不会将代码紧密耦合到具体的实现, 这会增加测试的难度。但每次将类实例化时, 应花一点时间来考虑在该特定位置硬编码该特定实例是否有意义, 或者说如果将该实例作为依赖项进行请求会不会更好。

#### 声明依赖关系

ASP.NET Core 的构建原理是让方法和类声明依赖关系, 并将其作为参数进行请求。ASP.NET 应用程序通常在 `Startup` 类中进行设置, 而该类本身就配置为在多处支持依赖关系注入。如果 `Startup` 类具有构造函数, 则可以通过



构造函数请求依赖关系，如下所示：

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

Startup 类很有意思，因为它没有显式的类型要求。它不继承特殊的 Startup 基类，也不实现任何特定的接口。可为其提供构造函数，也可不提供，并且可以为构造函数指定任意所需数量的参数。为应用程序配置的 Web 主机启动时，该主机将调用你命令其使用的 Startup 类，并将使用依赖关系注入来填充 Startup 类请求的任何依赖关系。当然，如果 ASP.NET Core 使用的服务容器中未配置请求的参数，则会引发异常，但只要是粘附到容器知晓的依赖项，则可以请求任何所需内容。

依赖关系注入从一开始创建 Startup 实例时就内置在 ASP.NET Core 应用中。它不会为 Startup 类在此停留。也可以在 Configure 方法中请求依赖关系：

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

ConfigureServices 方法是此行为的例外情况，它必须使用 IServiceCollection 类型的一个参数。实际上它并不需要支持依赖注入，因为一方面它负责向服务容器添加对象，另一方面它有权通过 IServiceCollection 参数访问所有当前已配置的服务。因此在 Startup 类的每个部分均可使用 ASP.NET Core 服务集中定义的依赖关系，方法是以参数形式请求所需服务，或在 ConfigureServices 中使用 IServiceCollection。

#### NOTE

如果需要确保某些服务可供 Startup 类使用，可以使用 IWebHostBuilder 及其 ConfigureServices 方法在 CreateDefaultBuilder 调用中对其进行配置。

Startup 类是一个范例，应照此构建 ASP.NET Core 应用程序的其他部分 - 从控制器到中间件到过滤器再到自己的服务。在任何情况下都应遵守[显式依赖关系原则](#)，请求依赖关系，而不要直接创建依赖关系，在整个应用程序中充分利用依赖关系注入。注意对实现进行直接实例化的位置和方式，特别是使用基础结构或会产生负面影响的服务和对象。相较于对针对特定实现类型的引用进行硬编码，最好是使用在应用程序核心中定义并作为参数传递的抽象元素。

## 构建应用程序

整体式应用程序通常只有一个入口点。对 ASP.NET Core Web 应用程序而言，入口点是 ASP.NET Core Web 项目。但是，这并不意味着解决方案只应包含一个项目。按照分离关注点的原则，将应用程序分解到不同层中非常有用。分解到不同层，有助于脱离文件夹的局限来分离项目，可帮助实现更好的封装。通过 ASP.NET Core 应用程序实现这些目标的最佳方法是第 5 章中所述的整洁架构的变体。按照此方法，应用程序的解决方案将包含 UI、基础结构和 ApplicationCore 各自单独的库。

除这些项目之外，还包含一个单独的测试项目（第 9 章中对测试进行介绍）。

应用程序的对象模型和接口应放在 ApplicationCore 项目中。此项目应具有尽可能少的依赖关系，可供解决方案中

的其他项目引用。需要保留的业务实体以及不直接依赖基础结构的服务在 ApplicationCore 项目中进行定义。

实现的详细信息(例如如何执行保留或如何将通知发送给用户)保存在 Infrastructure 项目中。此项目将引用特定于实现的包, 例如 Entity Framework Core, 但不应在此项目之外公开这些实现的详细信息。基础结构服务和存储库应实现 ApplicationCore 项目中定义的接口, 其持久性实现负责检索和存储 ApplicationCore 中定义的实体。

ASP.NET Core UI 项目负责所有 UI 级问题, 但不得包含业务逻辑或基础结构详细信息。实际上, 最理想的情况是它甚至没有对 Infrastructure 项目的依赖关系, 这样可确保不意外引入两个项目之间的依赖关系。第三方 DI 容器(例如 Autofac)可用于定于每个项目中模块类中的 DI 规则, 从而帮助实现这一目的。

将应用程序与实现详细信息分离开的另一种方法是让应用程序调用微服务, 微服务可能部署在各 Docker 容器中。相较于在两个项目之间利用 DI, 这种方法更好地分离关注点, 且解耦效果更好, 但也更复杂一些。

## 功能整理

默认情况下, ASP.NET Core 应用程序将其文件夹结构整理为包含 Controllers 和 Views, 还经常包含 ViewModels。支持这些服务器端结构的客户端代码通常单独存放在 wwwroot 文件夹中。但是对于大型应用程序而言, 这种整理方式可能会出现问題, 因为处理任何给定功能通常会要求在这些文件夹之间跳转。每个文件夹中的文件和子文件夹数量越多, 通过解决方案资源管理器的滚动就越多, 这种整理方式实现起来也就越难。解决此问题的其中一种办法是按功能, 而不要按文件类型来整理应用程序代码。这种整理方式通常被称为功能文件夹或[功能切片](#)(另请参阅:[垂直切片](#))。

ASP.NET Core MVC 支持使用 Areas 实现此目的。使用区域可以在每个 Area 文件夹中创建单独的 Controllers 和 Views 文件夹集(以及任何关联的模型)。图 7-1 显示了一个使用 Areas 的示例文件夹结构。

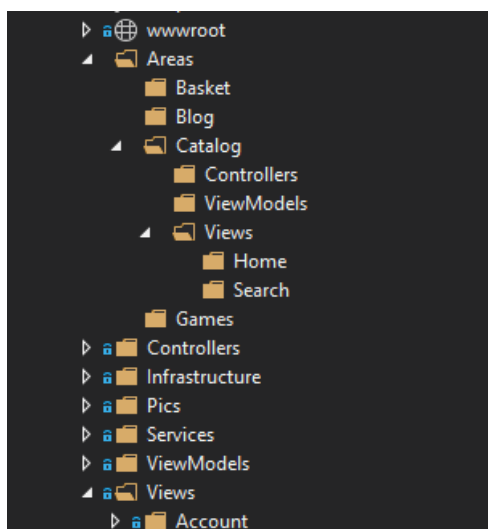


图 7-1。示例 Area 整理

使用 Areas 时, 必须使用属性通过控制器所属的区域名称来修饰控制器:

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

还需要向路由添加区域支持:

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

除了对 Areas 的内置支持, 还可以使用你自己的文件夹结构和约定来代替属性和自定义路由。这样可以让功能文

件夹中不包含单独的 Views 和 Controllers 等文件夹, 让层次结构更简单, 并且可以更轻松地在一个地方查看每个功能的所有相关文件。

ASP.NET Core 使用内置的约定类型来控制其行为。可以修改或替换这些约定。例如, 可以创建一个约定, 自动基于给定控制器的命名空间获取其功能名称(通常对应于控制器所在的文件夹):

```
public class FeatureConvention : IControllerModelConvention
{
    public void Apply(ControllerModel controller)
    {
        controller.Properties.Add("feature",
            GetFeatureName(controller.ControllerType));
    }

    private string GetFeatureName(TypeInfo controllerType)
    {
        string[] tokens = controllerType.FullName.Split('.');
        if (!tokens.Any(t => t == "Features")) return "";
        string featureName = tokens
            .SkipWhile(t => !t.Equals("features",
                StringComparison.CurrentCultureIgnoreCase))
            .Skip(1)
            .Take(1)
            .FirstOrDefault();
        return featureName;
    }
}
```

然后在 ConfigureServices 中向应用程序添加 MVC 支持时将此约定指定为一个选项:

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

ASP.NET Core MVC 还使用约定来确定视图的位置。可以使用自定义约定取而代之, 使视图位于功能文件夹中(使用上述 FeatureConvention 提供的功能名称)。可在 MSDN 杂志文章 [ASP.NET Core MVC 的功能切片](#) 中详细了解此方法并下载工作示例。

## 横切关注点

随着应用程序的发展, 将横切关注点分解出来以消除重复和保持一致性变得越来越重要。ASP.NET Core 应用程序中的横切关注点非常多, 例如身份验证、模型验证规则、输出缓存和错误处理等。ASP.NET Core MVC [过滤器](#) 允许在执行请求处理管道中的特定步骤之前或之后运行代码。例如, 可以在模型绑定之前/之后、某个操作之前/之后或某个操作结果之前/之后运行过滤器。还可以使用授权过滤器来控制对管道其余部分的访问权限。图 7-2 显示了请求执行操作是如何通过一系列过滤器(如果已配置)的。

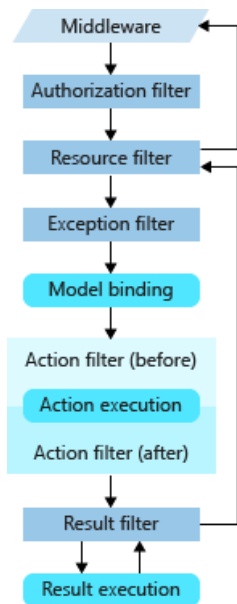


图 7-2。请求执行通过各过滤器和请求管道。

筛选器通常作为属性实现，因此可应用于控制器或操作（甚至全局）。以这种方式添加时，在操作级别指定的过滤器会覆盖在控制器级别指定的过滤器（会覆盖全局过滤器）或在其基础之上生成。例如，[Route] 属性可用来生成控制器和操作之间的路由。同样，可以在控制器级别配置授权，然后被各操作覆盖，如下所示：

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

第一个方法 Login 使用 AllowAnonymous 筛选器（属性）来覆盖在控制器级别设置的 Authorize 筛选器。ForgotPassword 操作（以及类中没有 AllowAnonymous 属性的任何其他操作）需要经过身份验证的请求。

筛选器可作为 API 的常见错误处理策略，用于消除重复。例如，如果请求引用的关键字不存在，典型的 API 策略会返回 NotFound 响应，如果模型验证失败，则返回 BadRequest 响应。下面的示例通过操作演示了这两种策略：

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

切勿让操作方法因类似于此的条件代码变得杂乱。应将策略放在可按需应用的过滤器中。此示例中，可使用以下属性替换（每当向 API 发送命令就会触发的）模型验证检查：

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

可以通过包含 `Ardalis.ValidateModel` 包将 `ValidateModelAttribute` 作为 NuGet 依赖项添加到项目中。对于 API，可以使用 `ApiController` 属性强制执行此行为，而无需单独的 `ValidateModel` 筛选器。

同样，可以使用过滤器来检查某条记录是否存在，并在执行操作前返回 404，而无需在操作中执行这些检查。在将常见约定提取出来、整理解决方案、将基础结构代码和业务逻辑与 UI 分离开后，MVC 操作方法会变得极其精简：

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

你可阅读 MSDN 杂志文章[真实的 ASP.NET Core MVC 过滤器](#)，了解有关实现过滤器的详细信息并下载工作示例。

#### 参考 - 构建应用程序

- **Areas**  
<https://docs.microsoft.com/aspnet/core/mvc/controllers/areas>
- **MSDN 杂志 - ASP.NET Core MVC 的功能切分**  
<https://docs.microsoft.com/archive/msdn-magazine/2016/september/asp-net-core-feature-slices-for-asp-net-core-mvc>
- **筛选器**  
<https://docs.microsoft.com/aspnet/core/mvc/controllers/filters>
- **MSDN 杂志 - 真实的 ASP.NET Core MVC 筛选器**  
<https://docs.microsoft.com/archive/msdn-magazine/2016/august/asp-net-core-real-world-asp-net-core-mvc-filters>

## 安全性

保护 Web 应用程序是一个非常大的主题，涉及到许多问题。安全性涉及的最基本问题是，确保你知道是谁发出的给定请求，然后确保该请求只对它应访问的资源具有访问权限。身份验证是将请求提供的凭据与受信任数据存储中的凭据进行对比的过程，目的在于确定该请求是否应被视为来源于已知实体。授权是根据用户标识限制对某些资源的访问权限的过程。第三个安全问题是保护请求免遭第三方窃听，对于此问题至少应[确保 SSL 由你的应用程序使用](#)。

### 身份验证

ASP.NET Core 标识是一个成员身份系统，可用于支持应用程序的登录功能。它支持本地用户帐户，也支持来自 Microsoft Account、Twitter、Facebook 和 Google 等提供者的外部登录。除 ASP.NET Core 标识之外，应用程序还可以使用 Windows 身份验证或 [Identity Server](#) 等第三方标识提供者。

如果选择了“个人用户帐户”选项，ASP.NET Core 标识将包含在新的项目模板中。此模板包括对注册、登录名、外部登录名、忘记密码和其他功能的支持。

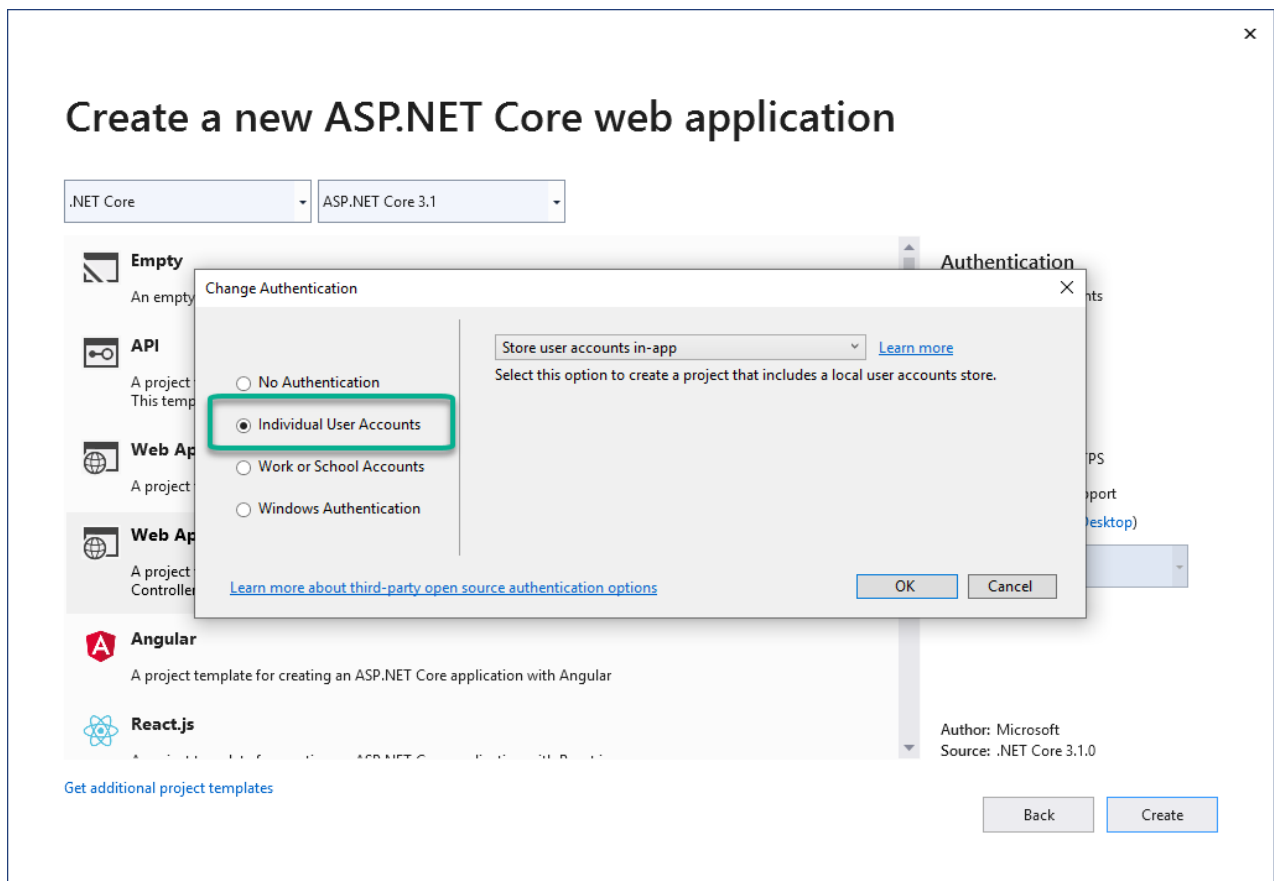


图 7-3。选择“个人用户帐户”以预配标识。

在 ConfigureServices 和 Configure 中，标识支持都在 Startup 中配置：

```
public void ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));
    services.AddIdentity<ApplicationUser, IdentityRole>()
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddDefaultTokenProviders();
    services.AddMvc();
}

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

在 Configure 方法中，UseIdentity 应出现在 UseMvc 之前，这一点很重要。在 ConfigureServices 中配置标识时，请注意对 AddDefaultTokenProviders 的调用。这与用于保护 Web 通信的令牌无关，它引用的是创建提示的提供者，该提示会通过短信或电子邮件发送给用户让其确认身份。

可从官方 ASP.NET Core 文档详细了解有关 [配置双重身份验证](#) 和 [启用外部登录提供者](#) 的信息。

## 授权

最简单的授权方式包括限制匿名用户的访问权限。只需对某些控制器和操作应用 [Authorize] 属性即可实现此目的。如果正在使用角色，可进一步扩展该属性，用于限制属于特定角色的用户的访问权限，如下所示：

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

}
```

此示例中, 属于 HRManager 或 Finance 角色 (或同时属于这两个角色) 的用户将有权访问 SalaryController。如要求用户属于多个角色, 而不是属于多个角色中的某一个角色, 可多次应用该属性, 每次指定一个所需角色。

在许多不同的控制器和操作中以字符串形式指定特定角色集可能会导致不必要的重复。可配置封装授权规则的授权策略, 然后在应用 [Authorize] 属性时指定该策略, 而不是各个角色:

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

以这种方式使用策略可将受限制的操作与适用于该操作的特定角色或规则区分开。之后创建需访问特定资源的新角色时, 只需更新策略即可, 而无需更新每个 [Authorize] 属性上的每个角色列表。

#### 声明

声明是名称值对, 代表已通过身份验证的用户的属性。例如, 可以将用户的员工编号存储为声明。该声明随后可用作授权策略的一部分。可以创建一个名为“EmployeeOnly”的策略, 该策略要求存在名为“EmployeeNumber”的声明, 如下所示:

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

此策略随后可与 [Authorize] 属性一起用于保护任何控制器和/或操作, 如上所述。

#### 保护 Web API

大多数 Web API 应实现基于令牌的身份验证系统。令牌身份验证无状态, 且可缩放。在基于令牌的身份验证系统中, 客户端必须首先使用身份验证提供程序进行身份验证。如果成功, 则向该客户端颁发一个令牌, 该令牌即是字符经过加密的有意义的字符串。然后当客户端需要向 API 发出请求时, 会将此令牌添加为请求的标题。继而, 服务器会在完成请求之前验证请求标头中的令牌。图 7-4 展示了此过程。



# Token-Based Authentication

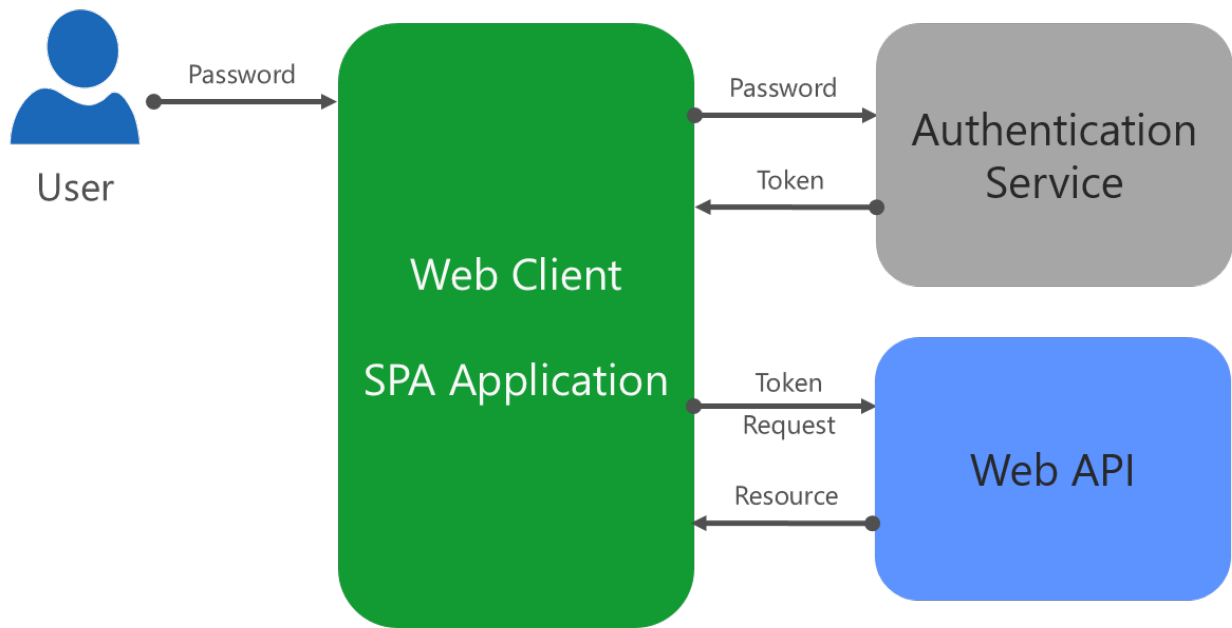


图 7-4. Web API 基于令牌的身份验证。

可以创建自己的身份验证服务，与 Azure AD 和 OAuth 集成，或使用开源工具（如 [IdentityServer](#)）实现服务。

## 自定义安全

要特别注意加密、用户成员身份或令牌生成系统的“自己回滚”实现。有许多商业和开源替代方案可供使用，几乎可以肯定，它们比自定义实现更具安全性。

### 参考 - 安全

- 安全性文档概述  
<https://docs.microsoft.com/aspnet/core/security/>
- 在 ASP.NET Core 应用中强制实施 SSL  
<https://docs.microsoft.com/aspnet/core/security/enforcing-ssl>
- 标识简介  
<https://docs.microsoft.com/aspnet/core/security/authentication/identity>
- 授权简介  
<https://docs.microsoft.com/aspnet/core/security/authorization/introduction>
- Azure 应用服务中 API 应用的身份验证和授权  
<https://docs.microsoft.com/azure/app-service-api/app-service-api-authentication>
- 标识服务器  
<https://github.com/IdentityServer>

## 客户端通信

除了通过 Web API 提供页面和响应数据请求之外，ASP.NET Core 应用还能与已连接的客户端直接通信。这种出站通信可以使用多种传输技术，其中最常见的是 WebSocket。ASP.NET Core SignalR 是一个库，它简化了向应用程序添加某种实时服务器到客户端的通信功能的过程。SignalR 支持多种传输技术，包括 WebSocket，并从开发人员处抽象出许多实现细节。

无论是直接使用 WebSocket 还是使用其他技术，实时客户端通信在许多应用程序方案中都很有用。一些示例包括：

- 实时聊天室应用程序

- 监视应用程序
- 作业进度更新
- 通知
- 交互式窗体应用程序

在应用程序中构建客户端通信时，通常有两个组件：

- 服务器端连接管理器 (SignalR Hub、WebSocketManager WebSocketHandler)
- 客户端库

客户端不限于浏览器 - 移动应用、控制台应用和其他本地应用也可以使用 SignalR/WebSocket 进行通信。下面的简单程序是 WebSocketManager 示例应用程序的一部分，它向控制台回显发送给聊天应用程序的所有内容：

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

    public static async Task StartConnectionAsync()
    {
        _connection = new Connection();
        await _connection.StartConnectionAsync("ws://localhost:65110/chat");
    }

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

请思考应用程序可通过哪些方式与客户端应用程序直接通信，以及实时通信是否会改善应用的用户体验。

#### 参考 - 客户端通信

- **ASP.NET Core SignalR**  
<https://github.com/dotnet/aspnetcore/tree/master/src/SignalR>
- **WebSocket 管理器**  
<https://github.com/radu-matei/websocket-manager>

## 领域驱动设计 - 是否该使用？

领域驱动设计 (DDD) 是一种敏捷方法，用于构建强调业务领域的软件。它非常注重与业务领域专家的沟通和互动，这些专家可以让开发人员了解真实世界中的系统是如何工作的。例如，如果你在构建处理股票交易的系统，那么领域专家可能是一位经验丰富的股票经纪人。DDD 旨在解决大型、复杂的业务问题，通常不适合小型简单的应用程序，因为在理解领域和为领域建模上所花费的投入是不必要的。

采用 DDD 方法构建软件时，团队（包括非技术型利益干系人和参与者）应为问题空间开发一种通用语言。即，要进行建模的实际概念、软件同义词以及可能存在以维持该概念的任何结构（例如数据库表）应使用相同的术语。因此，通用语言中所述的概念应该形成域模型的基础。

组成域模型的对象彼此交互，以表现系统行为。这些对象可分为以下几类：

- **实体**，表示具有一系列标识的对象。实体通常使用键永久存储，并且之后可使用该键进行检索。
- **聚合**，表示应作为单元保留的一组对象。
- **值对象**，表示可以根据其属性值的总和进行比较的概念。例如，包含开始日期和结束日期的 `DateRange`。
- **领域事件**，表示系统中发生的与系统其他部分相关的事件。

DDD 领域模型应在模型中包含复杂行为。尤其是实体，它不应该仅仅是属性的集合。域模型缺少行为，并且仅表示系统状态时，就是所谓的**贫乏性模型**，DDD 中应避免此类模型。

除这些模型类型之外，DDD 通常还采用多种模式：

- **存储库**，用于提取持久保留详细信息。
- **工厂**，用于封装复杂对象创建。
- **域事件**，用于从触发行为中分离依赖性行为。
- **服务**，用于封装复杂行为和/或基础结构实现细节。
- **命令模式**，用于分离发出的命令并执行命令本身。
- **规约模式**，用于封装查询细节。

DDD 还建议使用之前介绍过的整洁架构，以实现松散耦合、封装和使用单元测试即可轻松验证的代码。

### 该何时使用 DDD

DDD 非常适合业务（不仅仅是技术）非常复杂的大型应用程序。这种应用程序需要借助领域专家的知识。领域模型本身应包含有某种意义的行为，应体现出业务规则和交互，而不仅仅是存储和检索数据存储中各种记录的当前状态。

### 何时不该使用 DDD

DDD 需要在建模、体系结构和通信方面进行投资，这对于较小型的应用程序或本质只是 CRUD（创建/读取/更新/删除）的应用程序来说可能并不值得。如果选择采用 DDD 处理应用程序，但发现域中有一个没有任何行为的贫乏性模型，则可能需要重新考虑处理方法。可能是该应用程序不需要 DDD，也可能是你需要别人帮助你重构应用程序，将业务逻辑封装在域模型中，而不是数据库或用户界面中。

可以使用混合方法，只对应用程序中的事务性区域或比较复杂的区域使用 DDD，而不对应用程序中比较简单的 CRUD 或只读部分使用 DDD。例如，如果是为显示报表或将仪表板数据可视化而查询数据，则无需具有聚合约束。使用单独的、更简单的读取模型处理这类要求是完全可以接受的。

#### 参考 - 域驱动设计

- 简明 DDD (StackOverflow 答案)

<https://stackoverflow.com/questions/1222392/can-someone-explain-domain-driven-design-ddd-in-plain-english-please/1222488#1222488>

## 部署

无论在哪里托管 ASP.NET Core 应用，部署过程都包含以下几个步骤。第一步，发布应用程序，这可以使用

`dotnet publish` CLI 命令来完成。此操作将编译应用程序，并将运行应用程序所需的所有文件放到指定的文件夹中。从 Visual Studio 部署时，系统将自动执行此步骤。发布文件夹中包含应用程序的 .exe 和 .dll 文件及其依赖项。自包含应用程序中还包含一个 .NET 运行时版本。ASP.NET Core 应用程序还将包含配置文件、静态客户端资产和 MVC 视图。

ASP.NET Core 应用程序是控制台应用程序，服务器启动时必须启动，应用程序（或服务器）崩溃时必须重新启动。

可以使用流程管理器自动执行此过程。适用于 ASP.NET Core 的最常见的进程管理器是 Linux 上的 Nginx 和 Apache, 以及 Windows 上的 IIS 或 Windows Service。

除流程管理器之外, ASP.NET Core 应用程序可以使用反向代理服务器。反向代理服务器接收到来自 Internet 的 HTTP 请求, 并在进行一些初步处理后将这些请求转发到 Kestrel。反向代理服务器为应用程序提供了一层安全性。Kestrel 也不支持在同一端口上承载多个应用程序, 因此不能将其用于主机头之类的技术以实现在同一端口和 IP 地址上承载多个应用程序。

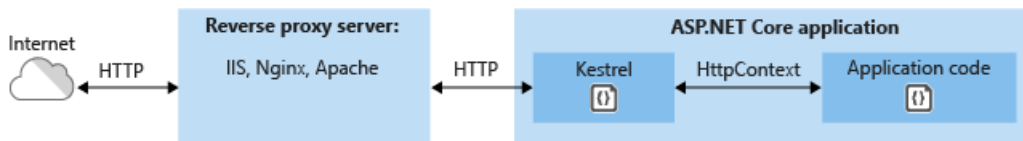


图 7-5。反向代理服务器背后托管在 Kestrel 中的 ASP.NET

反向代理发挥作用的其他情况包括使用 SSL/HTTPS 保护多个应用程序。在这种情况下, 只需要为反向代理服务器配置 SSL。反向代理服务器和 Kestrel 之间的通信可以通过 HTTP 进行, 如图 7-6 所示。

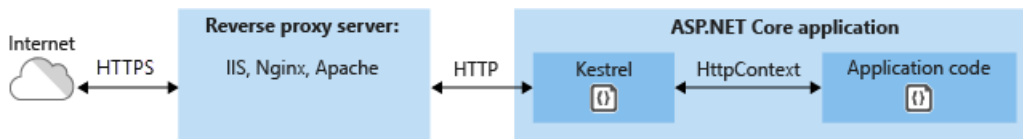


图 7-6。HTTPS 保护的反向代理服务器后托管的 ASP.NET

可以将 ASP.NET Core 应用程序托管在 Docker 容器中, 然后该应用程序即可本地托管或部署到 Azure 进行基于云的托管, 这种方法正日益普及。Docker 容器可以包含应用程序代码(在 Kestrel 上运行), 并部署在反向代理服务器后, 如上所述。

如果在 Azure 上托管应用程序, 则可以使用 Microsoft Azure 应用程序网关作为专用虚拟设备来提供多项服务。除了充当单个应用程序的反向代理之外, 应用程序网关还可以提供以下功能:

- HTTP 负载均衡
- SSL 卸载(仅到 Internet 的 SSL)
- 端到端 SSL
- 多站点路由(在单个应用程序网关上整合最多 20 个站点)
- Web 应用程序防火墙
- Websocket 支持
- 高级诊断

请在第 10 章中了解有关 Azure 部署选项的详细信息。

#### 参考 - 部署

- 托管和部署概述  
<https://docs.microsoft.com/aspnet/core/publishing/>
- 何时结合使用 Kestrel 和反向代理  
<https://docs.microsoft.com/aspnet/core/fundamentals/servers/kestrel#when-to-use-kestrel-with-a-reverse-proxy>
- 在 Docker 容器中托管 ASP.NET Core 应用  
<https://docs.microsoft.com/aspnet/core/publishing/docker>
- 应用程序网关简介  
<https://docs.microsoft.com/azure/application-gateway/application-gateway-introduction>



# 在 ASP.NET Core 应用中使用数据

2020/2/27 • [Edit Online](#)

“数据很宝贵，它的持续时间长于系统本身。”

Tim Berners-Lee

对于几乎任何软件应用程序，数据访问都是重要的部分。ASP.NET Core 支持各种数据访问选项，包括 Entity Framework Core (以及 Entity Framework 6)，并且可使用任何 .NET 数据访问框架。选择使用哪种数据访问框架，具体取决于应用程序的需求。从 ApplicationCore 和 UI 项目中提取这些选项，并在基础结构中封装实现详细信息，这有助于生成松散耦合的可测试软件。

## Entity Framework Core (适用于关系数据库)

如果要编写需要使用关系数据的新的 ASP.NET Core 应用程序，则 Entity Framework Core (EF Core) 是应用程序访问数据的建议方式。EF Core 是一种支持 .NET 开发人员将对象保存到数据源或从数据源中保存的对象关系映射程序 (O/RM)。它不要求提供开发人员通常需要编写的大部分数据访问代码。与 ASP.NET Core 一样，EF Core 经过完全重新编写以支持模块化跨平台应用程序。将其添加到应用程序作为 NuGet 包，在启动中配置它，并在需要的任何位置通过依赖关系注入请求它。

若要将 EF Core 用于 SQL Server，请运行以下 dotnet CLI 命令：

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

若要添加对 InMemory 数据源的支持用于测试，请使用：

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

### DbContext

要使用 EF Core，需要 [DbContext](#) 的子类。此类可保留表示应用程序将使用的实体集合的属性。eShopOnWeb 示例包括项目、品牌和类型集合的 CatalogContext：

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

    public DbSet<CatalogItem> CatalogItems { get; set; }

    public DbSet<CatalogBrand> CatalogBrands { get; set; }

    public DbSet<CatalogType> CatalogTypes { get; set; }
}
```

DbContext 必须包含可接受 DbContextOptions 的构造函数，可将此参数传递给基础 DbContext 构造函数。如果应用程序中只有一个 DbContext，可传递 DbContextOptions 的实例，但如果有多于一个 DbContext，则必须使用泛型 DbContextOptions<T>，在 DbContext 类型中作为泛型参数传递。

### 配置 EF Core

在 ASP.NET Core 应用程序中，通常在 `ConfigureServices` 方法配置 EF Core。EF Core 使用 `DbContextOptionsBuilder`，可支持多个有用的扩展方法，以简化其配置。若要将 `CatalogContext` 配置为与配置中定义的连接字符串一起使用 SQL Server 数据库，需要将以下代码添加到 `ConfigureServices`：

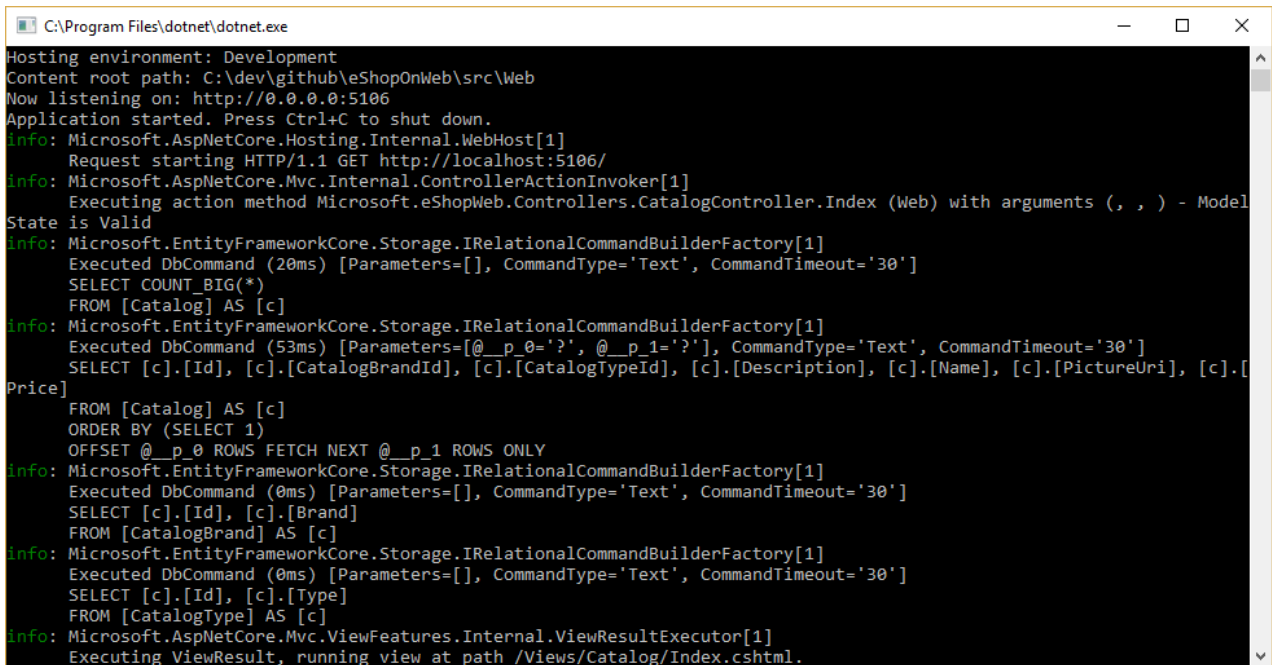
```
services.AddDbContext<CatalogContext>(options => options.UseSqlServer(
    Configuration.GetConnectionString("DefaultConnection")));
```

使用内存中数据库：

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

安装 EF Core 后，创建 `DbContext` 子类型，并在 `ConfigureServices` 中进行配置，然后便可使用 EF Core。可在需要 `DbContext` 类型实例的任何服务中进行请求，并通过 LINQ 开始使用持久化实体，就像它们位于集合中一样。EF Core 负责将 LINQ 表达式转换成 SQL 查询，以存储和检索数据。

可通过配置记录器并确保其级别至少设置为“信息”，查看 EF Core 要执行的查询，如图 8-1 所示。



```
C:\Program Files\dotnet\dotnet.exe
Hosting environment: Development
Content root path: C:\dev\github\eshoponweb\src\Web
Now listening on: http://0.0.0.0:5106
Application started. Press Ctrl+C to shut down.
Info: Microsoft.AspNetCore.Hosting.Internal.WebHost[1]
      Request starting HTTP/1.1 GET http://localhost:5106/
Info: Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker[1]
      Executing action method Microsoft.eShopWeb.Controllers.CatalogController.Index (Web) with arguments (, , ) - ModelState is Valid
Info: Microsoft.EntityFrameworkCore.Storage.IRelationalCommandBuilderFactory[1]
      Executed DbCommand (20ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      SELECT COUNT_BIG(*)
      FROM [Catalog] AS [c]
Info: Microsoft.EntityFrameworkCore.Storage.IRelationalCommandBuilderFactory[1]
      Executed DbCommand (53ms) [Parameters=@__p_0='?', @__p_1='?'], CommandType='Text', CommandTimeout='30']
      SELECT [c].[Id], [c].[CatalogBrandId], [c].[CatalogTypeId], [c].[Description], [c].[Name], [c].[PictureUri], [c].[Price]
      FROM [Catalog] AS [c]
      ORDER BY (SELECT 1)
      OFFSET @__p_0 ROWS FETCH NEXT @__p_1 ROWS ONLY
Info: Microsoft.EntityFrameworkCore.Storage.IRelationalCommandBuilderFactory[1]
      Executed DbCommand (0ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      SELECT [c].[Id], [c].[Brand]
      FROM [CatalogBrand] AS [c]
Info: Microsoft.EntityFrameworkCore.Storage.IRelationalCommandBuilderFactory[1]
      Executed DbCommand (0ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      SELECT [c].[Id], [c].[Type]
      FROM [CatalogType] AS [c]
Info: Microsoft.AspNetCore.Mvc.ViewFeatures.Internal.ViewResultExecutor[1]
      Executing ViewResult, running view at path /Views/Catalog/Index.cshtml.
```

图 8-1。将 EF Core 查询记录到控制台

## 提取和存储数据

要从 EF Core 中检索数据，请访问相应的属性，并使用 LINQ 筛选结果。还可以使用 LINQ 执行投影，将结果从一种类型转换为另一种。下面的示例可检索 `CatalogBrands`，按名称排序，按 `Enabled` 属性进行筛选，并投影到 `SelectListItem` 类型：

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

请务必在上述示例中添加对 `ToListAsync` 的调用，以立即执行查询。否则，语句会将 `IQueryable<SelectListItem>` 分配给 `brandItems`，`brandItems` 在被枚举前不会执行。从方法中返回 `IQueryable` 结果有优点，也有缺点。如果将操作添加到 EF Core 无法转换的查询中，它仍允许对 EF Core 将构造的查询进行进一步修改，但会导致出现仅在运行时发生的错误。将任何筛选器传递给执行数据访问的方法，并返回内存中集合（例如，`List<T>`）作为结果，这通



常会更安全。

EF Core 可跟踪它从持久性提取的实体上的更改。要将更改保存到被跟踪实体，只需对 DbContext 调用 SaveChanges 方法，确保它是用于提取实体的同一 DbContext 实例。直接对相应的 DbSet 属性添加和删除实体，再次通过调用 SaveChanges 执行数据库命令。下面的示例演示如何从持久性中添加、更新和删除实体。

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

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

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

EF Core 同时支持用于提取和保存的同步和异步方法。在 Web 应用程序中，建议将 async/await 模式与异步方法结合使用，以便在等待数据访问操作完成时不会阻止 Web 服务器线程。

### 提取相关数据

EF Core 检索实体时，将填充数据库中直接使用该实体存储的所有属性。不会填充导航属性（如相关实体列表），且它们的值可能设置为 Null。这可确保 EF Core 仅提取所需的数据，这对 Web 应用程序格外重要，Web 应用程序必须快速处理请求，并以有效的方式返回响应。要使用预先加载添加与实体的关系，请指定查询上使用 Include 扩展方法的属性，如下所示：

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

可添加多种关系，也可使用 ThenInclude 添加子关系。EF Core 将执行单一查询，检索生成的实体集。或者，可以通过将“”分隔的字符串传递给 `.Include()` 扩展方法来包含导航属性的导航属性，如下所示：

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

除了封装筛选逻辑，规范还可指定要返回的数据的形状，包括要填充的属性。eShopOnWeb 示例包含几个规范，用于演示如何在规范内封装预先加载信息。在此处可以查看如何将规范用作查询的一部分：

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

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

加载相关数据的另一个选项是使用显式加载。通过显式加载，可以将其他数据加载到已检索的实体中。由于这涉及单独的数据请求，因此不建议用于 Web 应用程序，Web 应用程序应尽量减少每个请求的数据往返次数。

延迟加载是可在应用程序引用相关数据时自动对其进行加载的一项功能。EF Core 2.1 版本中添加了延迟加载支持。延迟加载默认为不启用，且需要安装 `Microsoft.EntityFrameworkCore.Proxies`。与显式加载相同，通常应对 Web 应用禁用延迟加载，因为使用延迟加载将导致在每个 Web 请求内进行额外的数据库查询。遗憾的是，当延迟

较小并且用于测试的数据集通常也较小时，在开发时常常会难以发现延迟加载所产生的开销。但是，在生产中（涉及更多用户、更多数据和更多延迟），额外的数据库请求常常会导致大量使用延迟加载的 Web 应用性能不佳。

## 避免延迟加载 Web 应用中的实体

### 封装数据

EF Core 支持多种功能，使模型能够正确封装其状态。域模型中的一个常见问题是，它们将集合导航属性公开为可公开访问的列表类型。这使得任何协作方都能操作这些集合类型的内容，这可能会绕过与集合相关的重要业务规则，从而可能使对象处于无效状态。若要解决该问题，可向相关集合公开只读访问权限，并显式提供一些方法来定义客户端操作这些集合的方式，如本例中所示：

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

    public void AddItem(int catalogItemId, decimal unitPrice, int quantity = 1)
    {
        if (!Items.Any(i => i.CatalogItemId == catalogItemId))
        {
            _items.Add(new BasketItem()
            {
                CatalogItemId = catalogItemId,
                Quantity = quantity,
                UnitPrice = unitPrice
            });
            return;
        }
        var existingItem = Items.FirstOrDefault(i => i.CatalogItemId == catalogItemId);
        existingItem.Quantity += quantity;
    }
}
```

此实体类型不公开公共 `List` 或 `ICollection` 属性，而是公开包装有基础列表类型的 `IReadOnlyCollection` 类型。使用此模式时，可以指示 Entity Framework Core 使用支持字段，如下所示：

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

    navigation.SetPropertyAccessMode(PropertyAccessMode.Field);
}
```

另一种可以改进域模型的方法是通过缺少标识的类型使用值对象，并仅通过其属性进行区分。使用这些类型作为实体的属性有助于将逻辑特定于它所属的值对象，并且可以避免使用相同概念的多个实体之间的逻辑重复。在 Entity Framework Core 中，可以通过将类型配置为从属实体来将值对象保存在与其所属实体相同的表中，如下所示：

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

在此示例中，`ShipToAddress` 属性的类型为 `Address`。`Address` 是一个具有多个属性的值对象，例如 `Street` 和 `City`。EF Core 将 `Order` 对象映射到其表中，每个 `Address` 属性有一列，每个列名前面都加有该属性的名称。在此示例中，`Order` 表将包含 `ShipToAddress_Street` 和 `ShipToAddress_City` 等列。如果需要，也可以将从属类型存储在单独的表中。

详细了解 [EF Core 中的从属实体支持](#)。

## 弹性连接

外部资源(如 SQL 数据库)偶尔可能不可用。如果暂时不可用,应用程序可使用重试逻辑避免引发异常。此方法通常称为连接弹性。可以实现[指数退避算法的重试技术](#),方法是尝试重试,等待时间呈指数级增长,直到达到最大重试次数。该技术接受以下事实:云资源可能出现短时间间歇性不可用情况,导致某些请求失败。

对于 Azure SQL DB, Entity Framework Core 早已提供了内部数据库连接复原和重试逻辑。但如果想要复原 EF Core 连接,则需要为每个 DbContext 连接启用 Entity Framework 执行策略。

例如, EF Core 连接级别的下列代码可启用复原 SQL 连接,此连接在连接失败时会重试。

```
// Startup.cs from any ASP.NET Core Web API
public class Startup
{
    public IServiceCollection ConfigureServices(IServiceCollection services)
    {
        //...
        services.AddDbContext<OrderingContext>(options =>
        {
            options.UseSqlServer(Configuration["ConnectionString"],
                sqlServerOptionsAction: sqlOptions =>
                {
                    sqlOptions.EnableRetryOnFailure(
                        maxRetryCount: 5,
                        maxRetryDelay: TimeSpan.FromSeconds(30),
                        errorNumbersToAdd: null);
                });
        });
    }
    //...
```

### 使用 BeginTransaction 和多个 DbContext 的执行策略和显式事务

在 EF Core 连接中启用重试时,使用 EF Core 执行的每项操作都会成为其自己的可重试操作。如果发生暂时性故障,每个查询和 SaveChanges 的每次调用都会作为一个单元进行重试。

但是,如果代码使用 BeginTransaction 启动事务,这表示在定义一组自己的操作,这些操作需要被视为一个单元;如果发生故障,事务内的所有内容都会回退。如果在使用 EF 执行策略(重试策略)时尝试执行该事务,并且事务中包含一些来自多个 DbContext 的 SaveChanges,则会看到与下列情况类似的异常。

System.InvalidOperationException: 已配置的执行策略“SqlServerRetryingExecutionStrategy”不支持用户启动的事务。使用“DbContext.Database.CreateExecutionStrategy()”返回的执行策略执行事务(作为一个可重试单元)中的所有操作。

该解决方案通过代表所有需要执行的委托来手动调用 EF 执行策略。如果发生暂时性故障,执行策略会再次调用委托。下面的代码演示如何实现此方法:

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

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

第一个 DbContext 是 \_catalogContext, 第二个 DbContext 位于 \_integrationEventLogService 对象内。最后, “提交”操作将执行多个 DbContext, 并使用 EF 执行策略。

#### 引用 - Entity Framework Core

- **EF Core 文档** <https://docs.microsoft.com/ef/>
- **EF Core: 关联数据** <https://docs.microsoft.com/ef/core/querying/related-data>
- **避免延迟加载 ASP.NET 应用程序中的实体** <https://ardalis.com/avoid-lazy-loading-entities-in-asp-net-applications>

## 选择 EF Core 还是微型 ORM?

尽管对于管理持久性以及在大多数情况下封装应用程序开发者提供的数据库详细信息而言, EF Core 是绝佳选择, 但它不是唯一的选择。另一个热门的开源替代选择是一种所谓的微型 ORM, 即 [Dapper](#)。微型 ORM 是不具有完整功能的轻量级工具, 用于将对象映射到数据结构。Dapper 在设计上主要关注性能, 而不是完全封装用于检索和更新数据的基础查询。因为它不提取开发人员的 SQL, Dapper“更接近于原型”, 使开发人员可以编写要用于给定数据访问操作的确切查询。

EF Core 具有两个重要功能, 使其有别于 Dapper, 并且增加其性能开销。第一个功能是从 LINQ 表达式转换为 SQL。将缓存这些转换, 但即便如此, 首次执行它们时仍会产生开销。第二个功能是对实体进行更改跟踪(以便生成高效的更新语句)。通过使用 AsNotTracking 扩展, 可对特定查询关闭此行为。EF Core 还会生成通常非常高效的 SQL 查询, 并且从性能角度看, 任何情况下都能完全接受, 但如果需要执行对精确查询的精细化控制, 也可使用 EF Core 传入自定义 SQL(或执行存储过程)。在这种情况下, Dapper 的性能仍然优于 EF Core, 但只有略微优势。Julie Lerman 在其 2016 年 5 月的 MSDN 文章 [Dapper、Entity Framework 和混合应用](#) 中展示了部分性能数据。可访问 [Dapper 网站](#), 查看各种数据访问方法的其他性能基准数据。

要查看 Dapper 与 EF Core 语法之间的差异, 请考虑用于检索一系列项目的相同方法的两个版本:

```
// EF Core
private readonly CatalogContext _context;
public async Task<IEnumerable<CatalogType>> GetCatalogTypes()
{
    return await _context.CatalogTypes.ToListAsync();
}

// Dapper
private readonly SqlConnection _conn;
public async Task<IEnumerable<CatalogType>> GetCatalogTypesWithDapper()
{
    return await _conn.QueryAsync<CatalogType>("SELECT * FROM CatalogType");
}
```

要使用 Dapper 生成更复杂的对象图，需要自行编写相关的查询（这与在 EF Core 中添加 Include 不同）。此功能受到多种语法支持，包括称为“多映射”的功能，该功能允许将各个行映射到多个映射对象中。例如，给定一个类 Post，其中“所有者”属性是“用户”类型，以下 SQL 将返回所有必要的数据库：

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

返回的每行同时包括“用户”和“Post”数据。由于“用户”数据应通过其“所有者”属性附加到 Post 数据，因此使用下面的函数：

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

返回 post 集合的完整代码列表（其中使用相关的用户数据填充其“所有者”属性）为：

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

因为它提供较少封装，Dapper 要求开发人员深入了解如何存储其数据，如何对其进行高效查询，以及如何编写更多代码来提取它。模型发生更改时，不是单纯地创建新的迁移（另一种 EF Core 功能），并/或在 DbContext 的某一位置更新映射信息，而是必须更新受影响的所有查询。这些查询没有编译时间保证，因此它们可能在运行时中断，以应对模型或数据库的更改，增加快速检测错误的难度。Dapper 可提供极快的性能，以补偿这些方面付出的代价。

对于大多数应用程序和几乎所有应用程序的大多数组件而言，EF Core 提供可接受的性能。因此，其开发人员工作效率优势很可能大于性能开销这一点上的劣势。对于可受益于缓存的查询，实际查询执行的时间可能只占很小的百分比，因此可以忽略较小的查询性能差异。

## 选择 SQL 还是 NoSQL

传统上，SQL Server 等关系数据库占领了持久性数据存储的市场，但它们不是唯一可用的解决方案。NoSQL 数据库（如 MongoDB）可提供用于存储对象的不同方法。另一种选择是序列化整个对象图并存储结果，而不是将对象映射到表格和行。此方法的优点（至少在起初阶段）是简单和高性能。使用密钥存储单个序列化对象当然比将对象分解为多个表格（其中的关系以及更新和行可能自上次从数据库中检索该对象以来已更改）更简单。同样，从基于密钥的存储中提取和反序列化单个对象通常比复杂联接或多个数据库查询（完全编写关系数据库中的相同对象所需）更快速、更简单。此外，由于缺少锁定、事务或固定的架构，这使得 NoSQL 数据库适合在多台计算机中缩放，以支持大型数据集。

但是, NoSQL 数据库(人们通常这么称呼该数据库)也有其缺点。关系数据库利用规范化强制实施一致性并避免数据重复。这可减少数据库的总大小, 确保共享数据更新在整个数据库中立即可用。在关系数据库中, “地址”表可能按 ID 引用“国家/地区”表, 这样一来, 如果国家/地区名称发生更改, 地址记录将受益于更新, 而无需自行更新。但是, 对于 NoSQL 数据库, 众多存储对象中的“地址”及其相关的“国家/地区”可能会被序列化。如果对国家/地区或区域名称进行更新, 将需要更新所有此类对象, 而不是单独的某行。关系数据库还可通过强制实施外键等规则, 确保关系完整性。NoSQL 数据库通常不提供此类数据约束。

NoSQL 数据库需要处理的另一复杂性是版本控制。对象的属性发生更改时, 可能无法从存储的过去版本进行反序列化。因此, 需要更新具有对象的序列化(以前)版本的所有现有对象, 才能符合其新架构。从概念上讲, 这与关系数据库没有什么差异, 架构更改有时需要更新脚本或映射更新。但是, 需要修改的条目数量通常多于 NoSQL 方法, 因为存在更多的重复数据。

可以在 NoSQL 数据库中存储多个对象版本, 而这是固定架构关系数据库通常不支持的功能。但是, 在这种情况下, 应用程序代码需要考虑以前对象版本的存在, 这增加了额外的复杂性。

NoSQL 数据库通常不会强制实施 **ACID**, 这意味着它在性能和可伸缩性方面优于关系数据库。它们非常适合特别大型的数据集和对象, 不适合规范化表格结构中的存储。根据最适合的情景分别加以利用, 单个应用程序能同时获得关系数据库和 NoSQL 数据库带来的好处。

## Azure Cosmos DB

Azure Cosmos DB 是一项完全托管的 NoSQL 数据库服务, 可提供基于云的无架构数据存储。Azure Cosmos DB 旨在实现快速且可预测性能、高可用性、弹性缩放和全球分发。尽管属于 NoSQL 数据库, 但开发人员可对 JSON 数据使用熟悉的一系列 SQL 查询功能。Azure Cosmos DB 中的所有资源均存储为 JSON 文档。资源作为“项目”(包含元数据的文档)和“源”(项目集合)管理。图 8-2 显示了不同 Azure Cosmos DB 资源之间的关系。

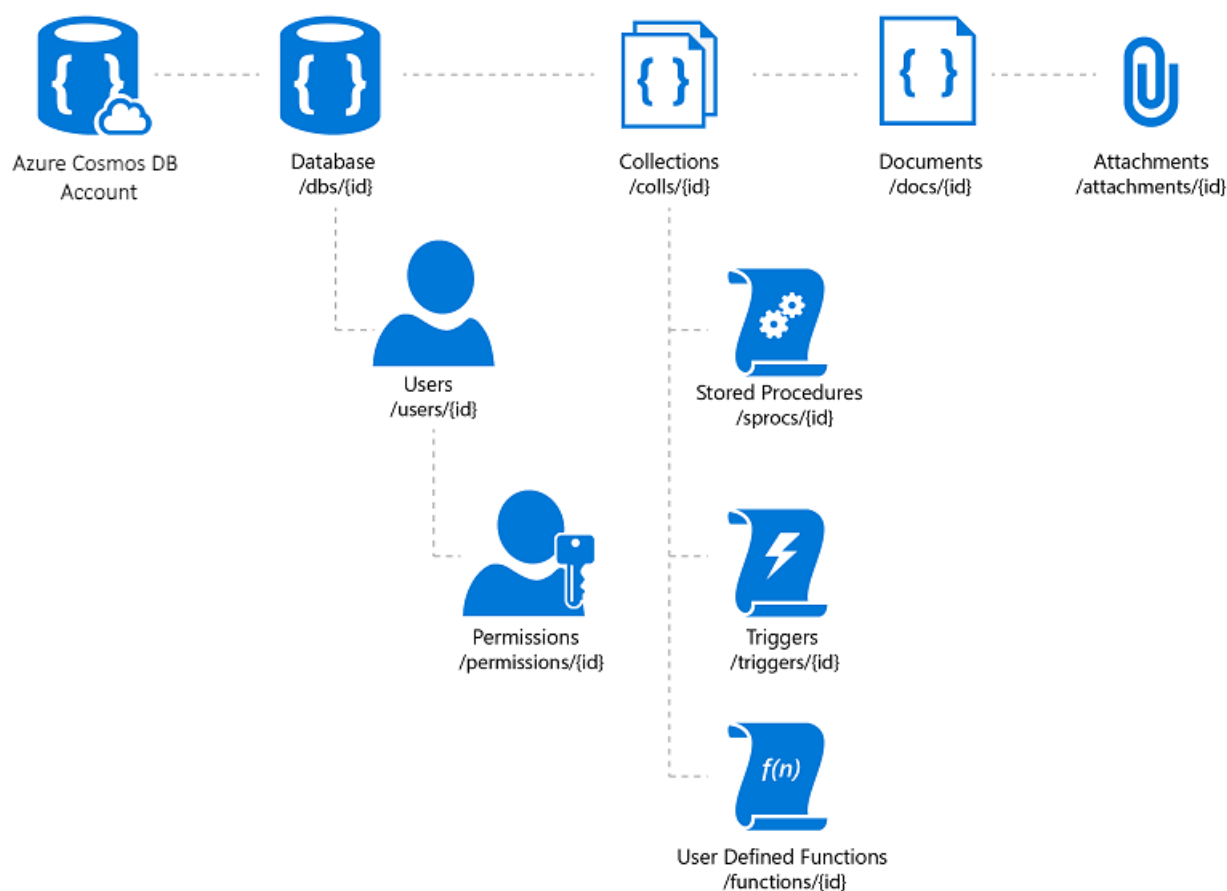


图 8-2。Azure Cosmos DB 资源组织。

Azure Cosmos DB 查询语言是一种用于查询 JSON 文档的简单而强大的接口。该语言支持 ANSI SQL 语法的子集, 并添加了对 JavaScript 对象、数组、对象结构和函数调用的深度集成。



- Azure Cosmos DB 简介 <https://docs.microsoft.com/azure/cosmos-db/introduction>

## 其他持久性选项

除关系数据库和 NoSQL 数据库存储选项外，ASP.NET Core 应用程序还可使用 Azure 存储，以基于云的可缩放方式存储各种数据格式和文件。Azure 存储可大规模缩放，因此如果应用程序需要存储少量数据并纵向扩展到存储数百或 TB 级数据，可采用 Azure 存储。Azure 存储支持四种数据：

- 适用于非结构化文本或二进制储存的 Blob 存储，也称为对象存储。
- 适用于结构化数据集的表存储，可通过行键进行访问。
- 适用于可靠且基于队列的消息传送的队列存储。
- 适用于 Azure 虚拟机和本地应用程序之间的共享文件访问的文件存储。

### 参考 – Azure 存储

- Azure 存储简介 <https://docs.microsoft.com/azure/storage/storage-introduction>

## 缓存

在 Web 应用程序中，应尽可能在最短的时间内完成每个 Web 请求。实现此目的的一种方法是限制服务器完成请求必须进行的外部调用次数。缓存涉及在服务器上存储数据副本（或比数据源更易于查询的其他数据存储）。Web 应用程序，特别是非 SPA 传统 Web 应用程序，需要对每个请求生成整个用户界面。这通常需要一个用户请求到下一个用户请求，重复进行多个相同的数据库查询。在大多数情况下，此类数据很少更改，因此没有必要不断地从数据库进行请求。ASP.NET Core 支持响应缓存，用于缓存整个页面，以及数据缓存（可支持更精细的缓存行为）。

实现缓存时，请务必记住将问题分离。避免在数据访问逻辑或用户界面中实现缓存逻辑。应将缓存封装于其自己的类中，并使用配置管理其行为。这遵循打开/关闭和单一责任原则，以使用户更轻松地随着应用程序发展管理应用程序的缓存使用方式。

### ASP.NET Core 响应缓存

ASP.NET Core 支持两种级别的响应缓存。第一种级别不会在服务器上缓存任何内容，但会向缓存响应添加指示客户端和代理服务器的 HTTP 标头。实现方式是通过向各个控制器或操作添加 ResponseCache 属性：

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

上述示例将导致将以下标头添加到响应，同时指示客户端缓存结果（最长 60 秒）。

Cache-Control: public,max-age=60

若要将服务器端内存中缓存添加到应用程序，则必须引用 Microsoft.AspNetCore.ResponseCaching NuGet 包，然后添加响应缓存中间件。ConfigureServices 和 Configure 中的 Startup 中均配置有此中间件：



```
public void ConfigureServices(IServiceCollection services)
{
    services.AddResponseCaching();
}

public void Configure(IApplicationBuilder app)
{
    app.UseResponseCaching();
}
```

响应缓存中间件将根据一组可自定义的条件自动缓存响应。默认情况下，仅缓存通过 GET 或 HEAD 方法请求的 200(正常)响应。此外，请求必须具有包含缓存控件(公共标头)的响应，且不能包含授权标头或 Set-Cookie。请参阅[响应缓存中间件所用缓存条件的完整列表](#)。

## 数据缓存

可以缓存各个数据查询的结果，而不是(或除了)缓存完整 web 响应。为此，可对 Web 服务器使用内存中缓存，或使用[分布式缓存](#)。本节将演示如何实现内存中缓存。

在 ConfigureServices 中添加内存(或分布式)缓存支持：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMemoryCache();
    services.AddMvc();
}
```

还需确保添加 Microsoft.Extensions.Caching.Memory NuGet 包。

添加服务后，可在需要访问缓存的任何位置，通过依赖关系注入请求 IMemoryCache。在此示例中，CachedCatalogService 将使用“代理”(或“修饰器”)设计模式，方法是通过提供控制对基础 CatalogService 实现的访问权限(或向其添加行为)的 ICatalogService 的备用实现。

```

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

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

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

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

```

若要配置应用程序，以使用服务的缓存版本，但仍允许服务获取其构造函数中需要的 `CatalogService` 的实例，请在 `ConfigureServices` 中添加以下内容：

```

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

```

完成以上操作后，将每分钟执行一次提取目录数据的数据库调用，而不是对每个请求执行。这可能对向数据库提出的查询数，以及当前依赖于此服务公开的所有三个查询的主页平均加载时间产生非常大的影响，具体取决于网站的流量。

缓存实现时将引发的问题是“数据过时”。也就是说，源中的数据已经更改，但缓存中保留的是过期版本的数据。缓解此问题的简单方法是使用较短的缓存持续时间，因此对于一个繁忙的应用程序而言，延长数据缓存时长的其他好处非常有限。例如，假设有一个进行单一数据库查询的网页，每秒对其请求 10 次。如果此网页缓存一分钟，每分钟执行的数据库查询数将从 600 降为 1，减少了 99.8%。如果缓存持续时间改为 1 小时，将整体减少 99.997%，但此时过时数据的可能性和潜在期限都可能大幅增加。

另一种方法是在包含的数据更新时主动删除缓存条目。如果其键已知，可删除任何条目：

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

如果应用程序公开用于更新其缓存的条目的功能，可在执行更新的代码中删除相应的缓存条目。有时可能存在众多不同的条目，具体取决于特定数据集。在这种情况下，使用 `CancellationToken` 创建缓存条目之间的依赖关系可能非常有用。使用 `CancellationToken`，可通过取消令牌同时使多个缓存条目过期。

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

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

缓存可以显著提高从数据库重复请求相同值的网页的性能。请确保在应用缓存前测量数据访问和页面性能，并且仅在发现需要改进性能时才应用缓存。缓存使用 Web 服务器内存资源并增加应用程序的复杂性，因此，不要过早使用此技术进行优化。

[上一 页](#)[下一 页](#)

# 测试 ASP.NET Core MVC 应用

2020/2/27 • [Edit Online](#)

“如果你不喜欢对产品进行单元测试，很可能你的客户也不喜欢这样做。” - 匿名 -

任何复杂程度的软件在响应更改方面皆可能意外失败。因此，更改后，除最普通(或关键性最低)的应用程序外，其他所有应用程序均需测试。手动测试是最慢、最不可靠且最昂贵的软件测试方式。遗憾的是，如果应用程序在设计上不具备可测试性，这可能是唯一可行的方式。为了遵循第 4 章中列出的体系结构原则而编写的应用程序应为单元可测试的。ASP.NET Core 应用程序支持自动集成和功能测试。

## 自动测试类型

软件应用程序自动测试具有多种类型。最简单最低级别的测试是单元测试。级别稍高的测试包括集成测试和功能测试。其他类型的测试不在本文讨论范围之内，例如 UI 测试、负载测试、压力测试和版本验收测试。

### 单元测试

单元测试可测试应用程序逻辑的单个部分。可通过否定列举方式对其进行进一步描述。单元测试并不测试代码如何处理依赖项或基础结构 - 这是集成测试的用途。单元测试不测试编写代码所用的框架，你应假定其适用，如果不适用，请报告 bug 并编写代码解决。单元测试完全在内存或进程中运行。它不会与文件系统、网络或数据库通信。单元测试仅测试代码。

因为单元测试仅测试单个代码单元，且无外部依赖项，所以执行速度应该非常快。因此，可在数秒内运行数百个单元测试的测试套件。请经常运行单元测试，最好是在每次推送到共享源代码管理存储库前运行，当然还要在生成服务器上的每个自动生成时运行它们。

### 集成测试

尽管建议封装与数据库和文件系统等基础结构交互的代码，但是仍会剩下一些此类代码，你可能需要对其进行测试。应用程序依赖项完全解析时，还应验证代码层是否按预期方式交互。这是集成测试的目的。与单元测试相比，由于集成测试通常依赖外部依赖项和基础结构，因此其设置速度较慢，难度较大。因此，应避免测试可以通过集成测试中的单元测试进行测试的项。如果可通过单元测试测试给定方案，请使用单元测试进行测试。如果不能，再考虑使用集成测试。

与单元测试相比，集成测试通常具有更复杂的设置和拆卸过程。例如，针对实际数据库运行的集成测试，在每次运行测试前，需要通过一种方式将数据库返回到已知状态。随着不断添加新测试以及生产数据库架构不断发展，这些测试脚本的大小和复杂性会逐渐增加。对于许多大型系统，将签入共享源代码管理更改前，在开发人员工作站上运行一整套集成测试并不实际。这种情况下，可在生成服务器上运行集成测试。

### 功能测试

集成测试从开发者角度编写，用于验证系统的一些组件是否能共同正常运行。功能测试从用户角度编写，用于基于其要求验证系统的正确性。下面的节选提供了一个有用的类比，有助于理解功能测试与单元测试的区别：

“很多时候，开发系统类似于修建房屋。虽然这个类比并不完全准确，但是我们可以将其延伸，用以理解单元测试与功能测试的区别。单元测试类似于巡查房屋建筑工地的建筑检查员。建筑检查员专注于房屋的各种内部系统、地基、框架、电路、管道等。他确保(测试)房屋各部分功能正常且安全，即符合建筑规范。在这种情景下，功能测试类似于出现在同一建筑工地上的房主。他假定房屋内部系统一切正常，建筑检查员履行了其检查职责。房主关心的是住在这个房屋里的体验。他关心房屋外观如何、每个房间是否大小合适、房屋是否能满足家庭需要，以及窗外是否有好的风景。也就是说，房主对房屋执行功能测试。他是站在用户角度。建筑检查员则是对房屋进行单元测试。他站在建筑商角度。”

源：[单元测试与功能测试](#)

我认为：“作为开发人员，我们的失败可能体现在两方面：我们构建应用的方式错误，或者我们的应用不能满足客户需求。”单元测试可确保构建应用的方式正确；功能测试可确保我们的应用可满足客户需求。

由于功能测试在系统级别运行，所以可能需要一定程度的 UI 自动化。与集成测试一样，它们通常也适用于某种类型的测试基础结构。这使其比单元测试和集成测试更慢、更易发生故障。应仅运行确保系统按用户期望运行所需数量的功能测试。

## 测试金字塔

Martin Fowler 提出了测试金字塔概念，如图 9-1 所示。

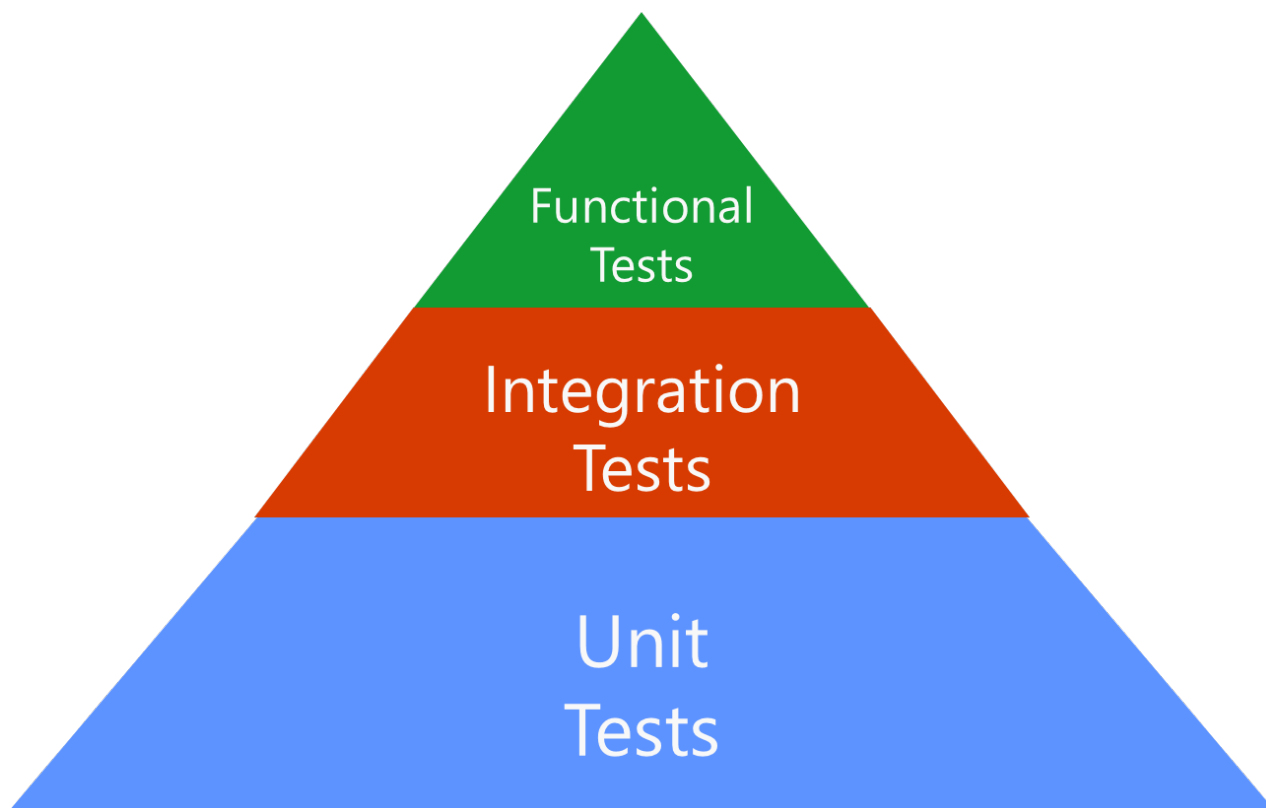


图 9-1。测试金字塔

该金字塔的不同层次及其相对大小表示不同的测试类别，以及应为应用程序编写的测试数量。如图所示，建议以大量单元测试作为基层，中间以较小的集成测试层作为支持，顶端为更小的功能测试层。理想情况下，每层应仅包含较低层中无法充分执行的测试。确定特定方案所需的测试类型时，请考虑此测试金字塔。

## 要测试的内容

对缺乏编写自动测试经验的开发人员而言，一个经常遇到的问题是确定测试内容。不妨从测试条件逻辑开始。如果方法中包含基于条件语句(if-else、switch 等)更改的行为，应能至少编写数个用于在某些条件下确定行为是否正确的测试。如果代码存在错误条件，建议通过代码为“主逻辑”编写至少一个测试(不含任何错误)，并为“副逻辑”编写至少一个测试(含错误或非典型结果)，以确保应用程序在错误情况下按预期运行。最后，重点测试可能失败的项，而不是关注代码覆盖率等指标。一般而言，高代码覆盖率比低代码覆盖率更好。但是，与仅仅为了改善测试代码覆盖率指标而编写自动属性测试相比，编写复杂的业务关键型方法的测试更有意义。

## 组织测试项目

可按最适合你的方式组织测试项目。最好按照类型(单元测试和集成测试)以及测试内容(按项目和按命名空间)分离测试。此分离由单个测试项目内的文件夹构成，还是由多个测试项目构成，这取决于设计决策。虽然一个项目最为简单，但是对于包含多个测试的大型项目，或者为了更轻松地运行不同测试集，可能需要具有一系列不同的测试项目。许多团队基于要测试的项目组织测试项目，对于具有很多项目的应用程序而言，这可能导致出现大量测试项目，在根据每个项目中的测试类型进行细分的情况下更是如此。一种折衷方式是让每个测试类型、每个应用程序有一个项目，测试项目内的文件夹指示要测试的项目(和类)。

一种常见方式是在“src”文件夹下组织应用程序项目，在平行的“tests”文件夹下组织应用程序的测试项目。如果你认

为这种组织方式有用，可以在 Visual Studio 中创建匹配的解决方案文件夹。

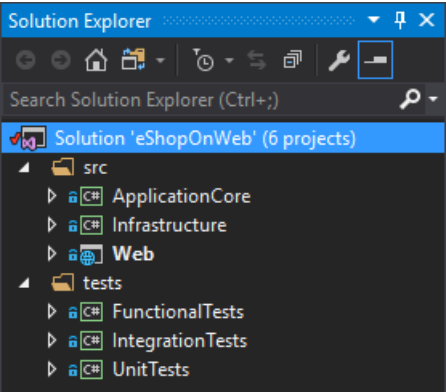


图 9-2。解决方案中的测试组织

可使用你喜欢的任何测试框架。xUnit 框架有效运行，编写所有 ASP.NET Core 和 EF Core 测试时皆使用此框架。可使用图 9-3 中所示的模板或从使用 `dotnet new xunit` 的 CLI，在 Visual Studio 中添加 xUnit 测试项目。

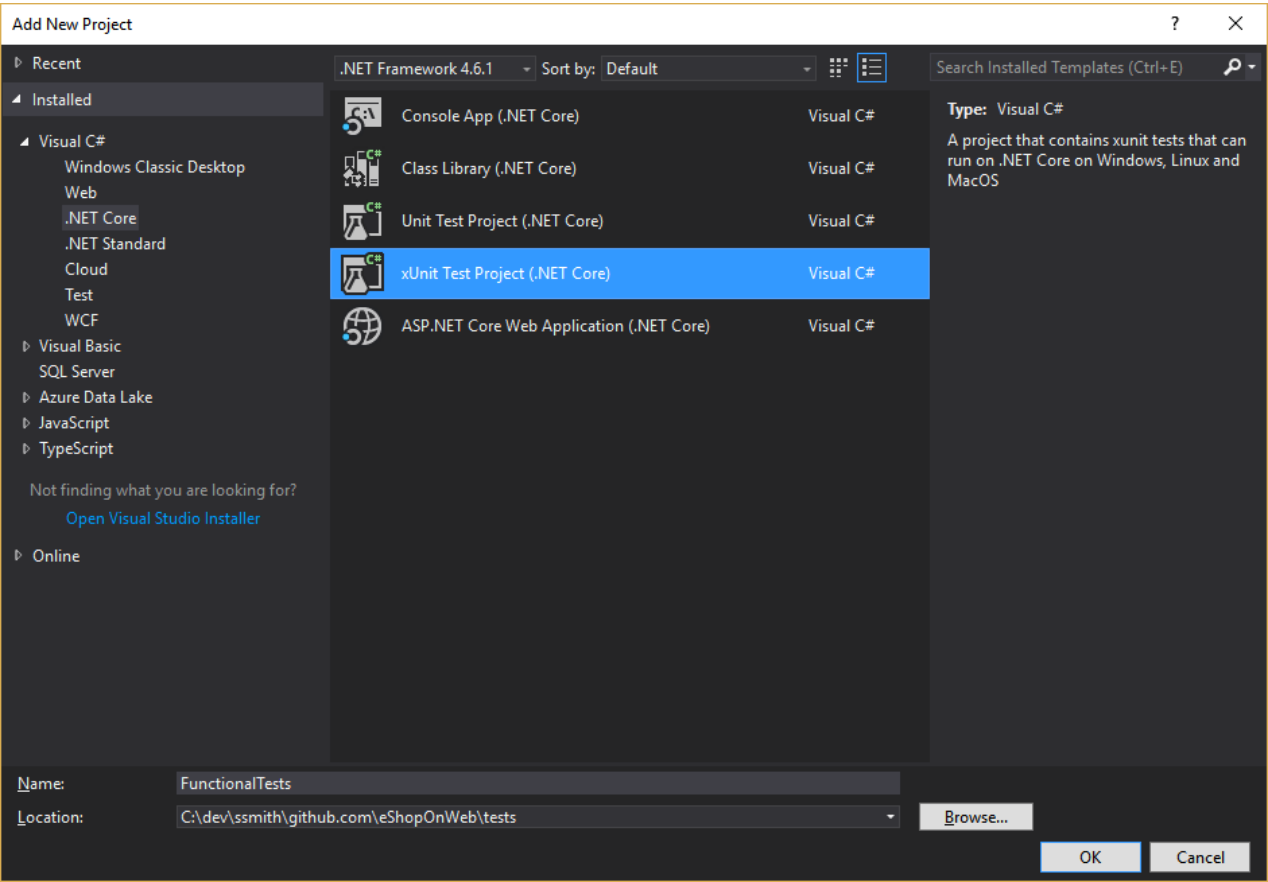


图 9-3。在 Visual Studio 中添加 xUnit 测试项目

### 测试命名

以一致方式命名测试，名称指示每个测试的功能。我使用的一种很成功的方式是根据其测试的类和方法命名测试类。虽然这会产生许多小测试类，但是可非常清楚地表明每个测试的作用。在设置测试类名称以标识待测试类和方法后，测试方法名称可用于指定要测试的行为。这应包括预期行为以及任何应导致该行为的输入或假设。部分示例测试名称：

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

此方式的一种变体是让每个测试类名称以“Should”结尾，并稍微修改其时态：

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

虽然第二种命名方式略为冗长，但一些团队发现这种命名方式更加清楚。任何情况下，请使用可提供测试行为见解的命名约定，以便一个或多个测试失败时，可通过其名称清楚了解已失败的事例。避免使用 `ControllerTests.Test1` 等模糊的测试名称，因为查看测试结果时，此类名称不能提供任何价值。

如果使用类似上述会产生众多小测试类的命名约定，建议使用文件夹和命名空间进一步组织测试。图 9-4 显示了一种在数个测试项目内按照文件夹组织测试的方式。

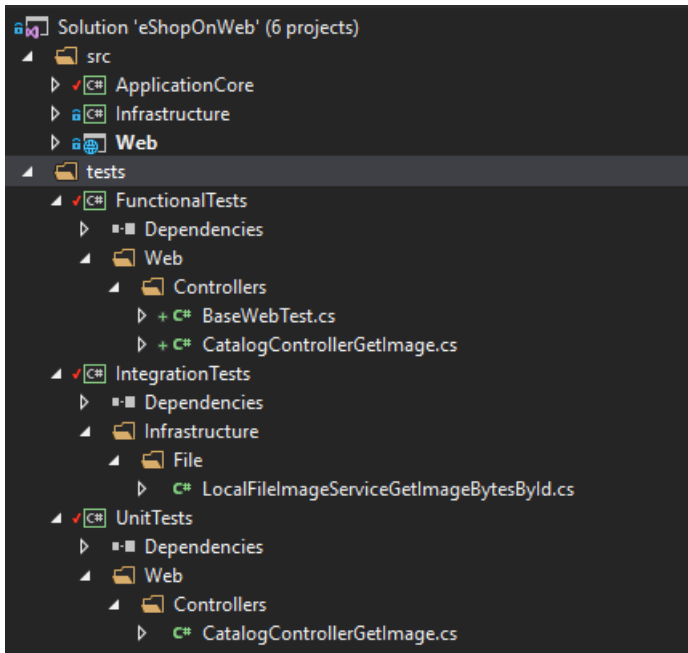


图 9-4。基于要测试的类按照文件夹组织测试类。

如果特定应用程序类具有多个待测试的方法（以及由此产生的众多测试类），最好将其放置于该应用程序类对应的文件夹中。该组织方式与在其他情况下将文件组织到文件夹并无差别。如果一个包含许多其他文件的文件夹中具有三个或四个以上相关文件，最好将其移动到它们自己的子文件夹中。

## 对 ASP.NET Core 应用执行单元测试

在具有出色设计的 ASP.NET Core 应用程序中，大多数复杂性和业务逻辑会封装在业务实体以及各种服务中。ASP.NET Core MVC 应用本身及其控制器、筛选器、视图模型和视图需要的单元测试极少。特定操作的很多功能体现在该操作方法之外。单元测试不能有效测试路由是否正常运行或测试全局错误。同样，无法使用针对控制器的操作方法的测试对任何筛选器（包括模型验证筛选器以及身份验证和授权筛选器）执行单元测试。如果没有这些行为源，大多数操作方法应非常小，这会将其大量工作委托至服务（可独立于使用这些服务的控制器对这些服务执行测试）。

有时为对代码执行单元测试，需要重构代码。通常，这涉及到确定抽象以及使用依赖项注入来访问待测试代码中的抽象，而不是直接针对基础结构编码。例如，请思考以下用于显示图像的简单操作方法：

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



通过 `System.IO.File` 上的直接依赖项难以对此方法执行单元测试。可测试此行为以确保其按预期方式运行，但对实际文件执行此操作属于集成测试。请注意，无法单元测试该方法的路由。稍后我们将了解如何通过功能测试快速执行此操作。

如果无法直接对文件系统行为执行单元测试，且无法测试路由，还能测试什么呢？通过重构确保单元测试的可行性后，可能会发现一些测试用例以及缺失行为，例如错误处理。如未找到文件，此方法会执行什么操作？它应执行什么操作？本示例中，重构方法如下：

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

`_logger` 和 `_imageService` 都作为依赖项注入。现在，可测试是否传递到操作方法的相同 ID 被传递到 `_imageService`，以及生成的字节是否作为 `FileResult` 的一部分返回。还可测试错误日志记录是否按预期发生，以及如果映像缺失，是否返回 `NotFound` 结果（假定这是重要的应用程序行为，即不只是开发人员为诊断问题而添加的临时代码）。已将实际文件逻辑移动到单独的实现服务，并已将其扩充，以在缺失文件的情况下返回特定于应用程序的异常。可使用集成测试独立测试该实现。

在大多数情况下，需要在控制器中使用全局异常处理程序，因此它们中的逻辑量应该是最小的，并且可能不值得进行单元测试。应该使用功能测试和下面介绍的 `TestServer` 类对控制器操作进行大部分测试。

## 对 ASP.NET Core 应用执行集成测试

ASP.NET Core 应用中的大多数集成测试应该是测试基础结构项目中定义的服务和其他实现类型。例如，可以[测试 EF Core 是否已成功更新并检索](#)希望从驻留在基础结构项目中的数据访问类中获得的数据。测试 ASP.NET Core MVC 项目是否正常运行的最佳方法是针对在测试主机中运行的应用运行的功能测试。

## 对 ASP.NET Core 应用执行功能测试

对于 ASP.NET Core 应用程序，`TestServer` 类让功能测试非常易于编写。可以直接使用 `WebHostBuilder`（或 `HostBuilder`）（针对应用程序的一般操作）或使用 `WebApplicationFactory` 类型（自 2.1 版开始提供）来配置 `TestServer`。应尝试将测试主机与生产主机进行尽可能密切的匹配，以便让测试执行与应用将在生产中进行的行作为类似的行为。`WebApplicationFactory` 类有助于配置 `TestServer` 的 `ContentRoot`，该 `ContentRoot` 由 ASP.NET Core 用于定位静态资源（例如视图）。

可以通过创建实现 `IClassFixture<WebApplicationFactory<TEntry>>`（其中 `TEntry` 为 Web 应用的启动类）的测试类来创建简单的功能测试。创建完成后，测试固定例程可使用中心的 `CreateClient` 方法来创建客户端：

```

public class BasicWebTests : IClassFixture<WebApplicationFactory<Startup>>
{
    protected readonly HttpClient _client;

    public BaseWebTest(WebApplicationFactory<Startup> factory)
    {
        _client = factory.CreateClient();
    }

    // write tests that use _client
}

```

用户常常想要在运行每个测试之前对站点执行一些其他配置，例如将应用程序配置为使用内存中数据存储，然后将测试数据植入应用程序。若要执行此操作，应创建自己的 `WebApplicationFactory<TEntry>` 子类并重写其 `ConfigureWebHost` 方法。以下示例来自 `eShopOnWeb FunctionalTests` 项目，并用作主要 Web 应用上的测试的一部分。

```

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

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

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

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

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

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

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

                // Create a scope to obtain a reference to the database
                // context (ApplicationDbContext).
                using (var scope = sp.CreateScope())

```

```

    {
        var scopedServices = scope.ServiceProvider;
        var db = scopedServices.GetRequiredService<CatalogContext>();
        var loggerFactory = scopedServices.GetRequiredService<ILoggerFactory>();

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

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

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

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

```

测试可以使用此自定义 `WebApplicationFactory` 来创建客户端，并使用此客户端实例向应用程序作出请求。该应用程序将具备植入的数据，这些数据可用作测试断言的一部分。以下测试验证 `eShopOnWeb` 应用程序的主页是否正确加载并包括已作为种子数据的一部分添加至应用程序的产品列表。

```

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

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

        public HttpClient Client { get; }

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

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

```

该功能测试执行完整的 ASP.NET Core MVC/Razor Pages 应用程序堆栈，包括所有中间件、筛选器、绑定器等。它验证给定的路由 ("/") 是否返回预期的成功状态代码和 HTML 输出。它无需设置实际 Web 服务器即可实现该操作，因此可避免使用实际 Web 服务器进行测试可能遇到的许多问题（例如防火墙设置问题）。虽然针对 TestServer 运行的功能测试通常比集成测试和单元测试更慢，但是比测试 Web 服务器的网络上运行的测试速度更快。应使用功能测试来确保应用程序的前端堆栈按预期运行。当在控制器或页面中发现了重复内容并通过添加筛选器找到了这些重复内容时，这些测试将尤为有用。理想情况下，此重构不会改变应用程序的行为，并且将有一套功能测试来验证确实如此。

#### 参考 - 测试 ASP.NET Core MVC 应用

- 在 ASP.NET Core 中进行测试  
<https://docs.microsoft.com/aspnet/core/testing/>
- 单元测试命名约定  
<https://ardalis.com/unit-test-naming-convention>
- 测试 EF Core  
<https://docs.microsoft.com/ef/core/miscellaneous/testing/>
- ASP.NET Core 中的集成测试  
<https://docs.microsoft.com/aspnet/core/test/integration-tests>

# Azure 的开发过程

2020/2/27 • [Edit Online](#)

“凭借云，个体和小型企业可轻松无忧地立即建立企业级服务。”

- Roy Stephan

## 愿景

使用 Visual Studio、dotnet CLI、Visual Studio Code 或所选用的编辑器，按照自己喜欢的方式开发出设计优良的 ASP.NET Core 应用程序。

## ASP.NET Core 应用的开发环境

### 开发工具选择：IDE 或编辑器

无论你更青睐内容丰富、功能强大的 IDE 还是灵活轻量的编辑器，Microsoft 都可为你提供用于开发 ASP.NET Core 应用程序的工具。

**Visual Studio 2019。**Visual Studio 2019 是用于开发适用于 ASP.NET Core 的应用程序的同类最佳 IDE。它提供了多种可提高开发人员生产率的功能。可以使用它来开发应用程序，然后分析其性能和其他特征。借助集成的调试程序，你可以暂停代码执行，并在运行时在代码中来回切换。借助内置的测试运行程序，你可以组织测试及其结果，甚至可以在编写代码时执行实时单元测试。使用 Live Share，可以与其他开发人员进行实时协作，并通过网络无缝共享代码会话。准备就绪后，Visual Studio 包括了将应用程序发布到 Azure 所需的所有内容或任何可托管它的位置。

[下载 Visual Studio 2019](#)

**Visual Studio Code 和 dotnet CLI** (适用于 Mac、Linux 和 Windows 的跨平台工具)。如果更青睐支持任何开发语言的轻量级跨平台编辑器，可以使用 Microsoft Visual Studio Code 和 the dotnet CLI。这些产品提供简单但可靠的体验，可以简化开发人员 workflow。此外，Visual Studio Code 支持适用于 C# 和 Web 开发的扩展，在该编辑器内提供智能感知和快捷任务。

[下载 .NET Core SDK](#)

[下载 Visual Studio Code](#)

## Azure 托管型 ASP.NET Core 应用的开发 workflow

应用程序开发生命周期始于每位开发人员的计算机，开发人员使用其首选语言对应用进行编码和本地测试。开发人员可选择其青睐的源代码管理系统，并可使用生成服务器或基于内置 Azure 功能配置持续集成 (CI) 和/或持续交付/部署 (CD)。

若要开始使用 CI/CD 开发 ASP.NET Core 应用程序，可使用 Azure DevOps Services 或组织自身的 Team Foundation Server (TFS)。

### 初始设置

若要为应用创建发布管道，需要具备源代码管理中的应用程序代码。设置一个本地存储库，并将其连接到团队项目中的远程存储库。请按照以下说明执行操作：

- [使用 Git 和 Visual Studio 分享代码](#)或
- [使用 TFVC 和 Visual Studio 分享代码](#)

创建要在其中部署应用程序的 Azure 应用服务。转至 Azure 门户上的“应用服务”选项卡，创建一个 Web 应用。单击“+ 添加”，选择 Web 应用模板，单击“创建”并提供名称或其他详细信息。该 Web 应用可通过 {name}.azurewebsites.net 进行访问。

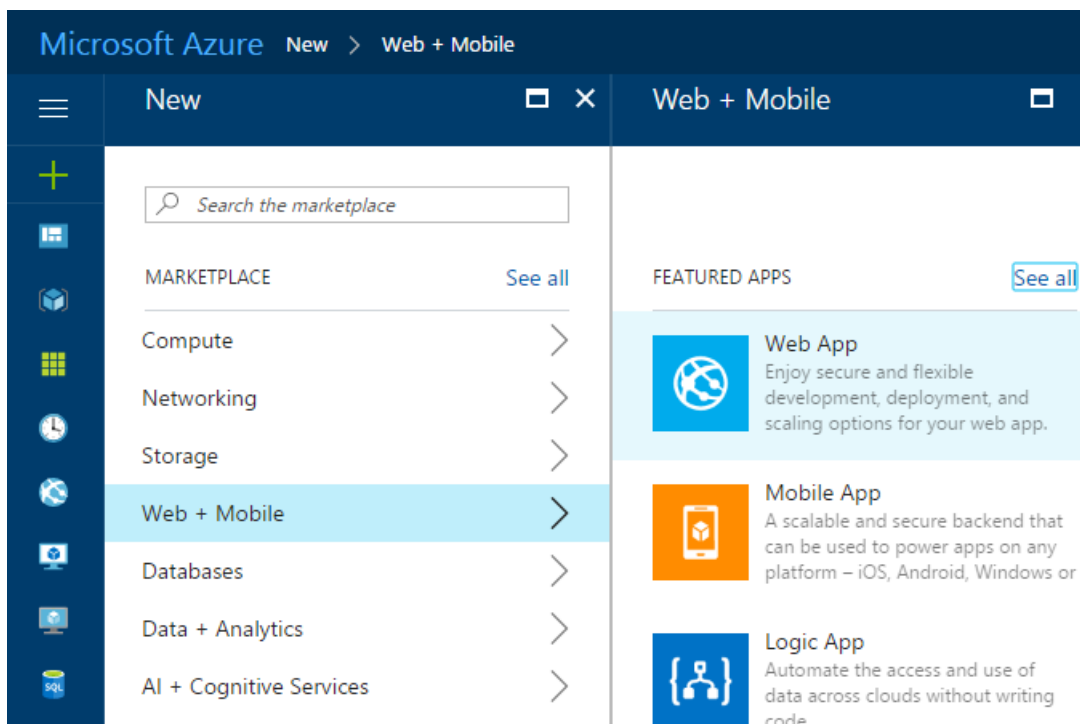


图 10-1. 在 Azure 门户中创建新的 Azure 应用服务 Web 应用。

无论新代码何时提交至项目的源代码管理存储库，CI 生成过程均会执行自动生成。由此你可获得即时反馈，知悉代码已生成（且理想情况下可通过自动测试），并且或许可进行部署。此 CI 生成将生成一个 Web 部署包项目，并将其发布，以供 CD 进程使用。

#### 定义 CI 生成过程

请务必启用持续集成，从而使得无论何时团队成员提交新代码，系统均可将生成排队。测试该生成，并验证其是否生成 Web 部署包作为其中一个项目。

生成成功后，CD 过程会将 CI 生成结果部署到 Azure Web 应用。如需对其配置，请创建并配置一个 Release（它将部署到 Azure 应用服务）。

#### 定义 CD 发布过程

配置 CI/CD 管道后，即可更新 Web 应用，并将更新提交至源代码管理以进行部署。

#### 开发 Azure 托管型 ASP.NET Core 应用程序的工作流

配置 Azure 帐户和 CI/CD 过程后，即可轻松开发 Azure 托管的 ASP.NET Core 应用程序。生成 ASP.NET Core 应用并将其托管在 Azure 应用服务作为 Web 应用时，通常会采用以下基本步骤，如图 10-2 所示。

## end-to-end development / deployment workflow

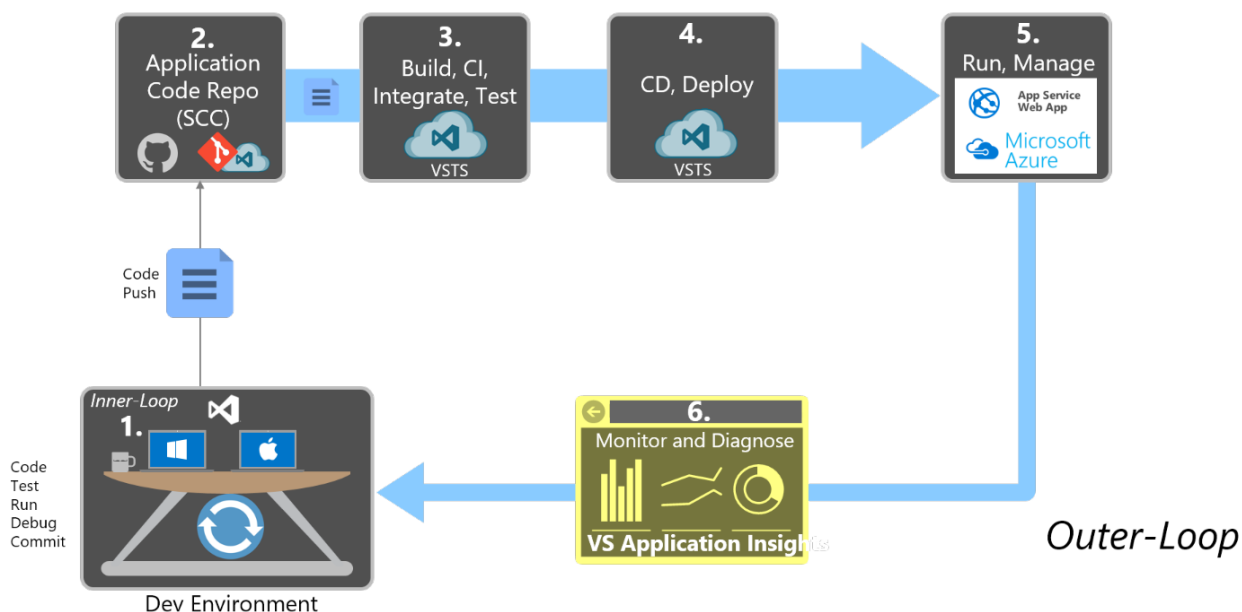


图 10-2. 构建 ASP.NET Core 应用以及将应用于托管于 Azure 的分步工作流

### 步骤 1. 本地开发环境内循环

开发要部署到 Azure 的 ASP.NET Core 应用程序与开发其他程序并无不同。使用惯用的本地开发环境，无论是 Visual Studio 2017、dotnet CLI、Visual Studio Code 还是个人首选的编辑器。在准备将更改推送到共享源代码存储库前，可以编写代码、运行并调试更改、运行自动测试以及本地提交到源代码管理。

### 步骤 2. 应用程序代码存储库

无论何时准备与团队共享代码，均应将更改从本地源存储库中推送到团队共享源存储库。如果一直在自定义分支中工作，此步骤通常涉及将代码合并到共享分支中（或许通过[拉取请求](#)方式）。

### 步骤 3. 生成服务器：持续集成。生成、测试、打包

向共享应用程序代码存储库进行新的提交时，生成服务器上会触发新的生成。作为 CI 过程的一部分，该生成应充分编译应用程序，并运行自动测试以确定一切正常。CI 过程的最终结果应是已可部署的打包版 Web 应用。

### 步骤 4. 生成服务器：持续交付

生成成功后，CD 过程将选取产生的生成项目。其中包括一个 Web 部署包。生成服务器将此包部署到 Azure 应用服务，使用新创建的服务替换任何现有服务。通常该步骤面向过渡环境，但是部分应用程序通过 CD 过程直接部署到生产。

### 步骤 5. Azure 应用服务 Web 应用

部署后，ASP.NET Core 应用程序在 Azure 应用服务 Web 应用的上下文运行。可使用 Azure 门户监视以及进一步配置该 Web 应用。

### 步骤 6. 生产监视和诊断

该 Web 应用运行时，可监视该应用程序的运行状况，并收集诊断和用户行为数据。Application Insights 包含于 Visual Studio 中，为 ASP.NET 应用提供自动检测。它可提供有关使用情况、异常、请求、性能和日志的信息。

## 参考资料

### 构建 ASP.NET Core 应用并将其部署到 Azure

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





# 关于 ASP.NET Core Web 应用的 Azure 托管建议

2019/11/25 • [Edit Online](#)

“全球各地的业务线领导绕过 IT 部门，从云获取应用程序(又称 SaaS)并支付使用费用，就像订阅杂志一样。不再需要服务时，他们可取消订阅，不会出现设备闲置的情况。”

-Gartner 分析师 Daryl Plummer

无论面对应用程序的何种需求和体系结构，Microsoft Azure 均可提供支持。托管需求可以非常简单(例如静态网站)，也可以非常复杂(例如由多个服务组成的复杂的应用程序)。对于 ASP.NET Core 整体 Web 应用程序和支持服务，推荐以下几种常见配置。无论是完整应用程序、单个进程或数据，本文中的建议配置均按待托管资源的类型分组。

## Web 应用程序

可通过以下方式托管 Web 应用程序：

- 应用服务 Web 应用
- 容器(多个选项)
- 虚拟机 (VM)

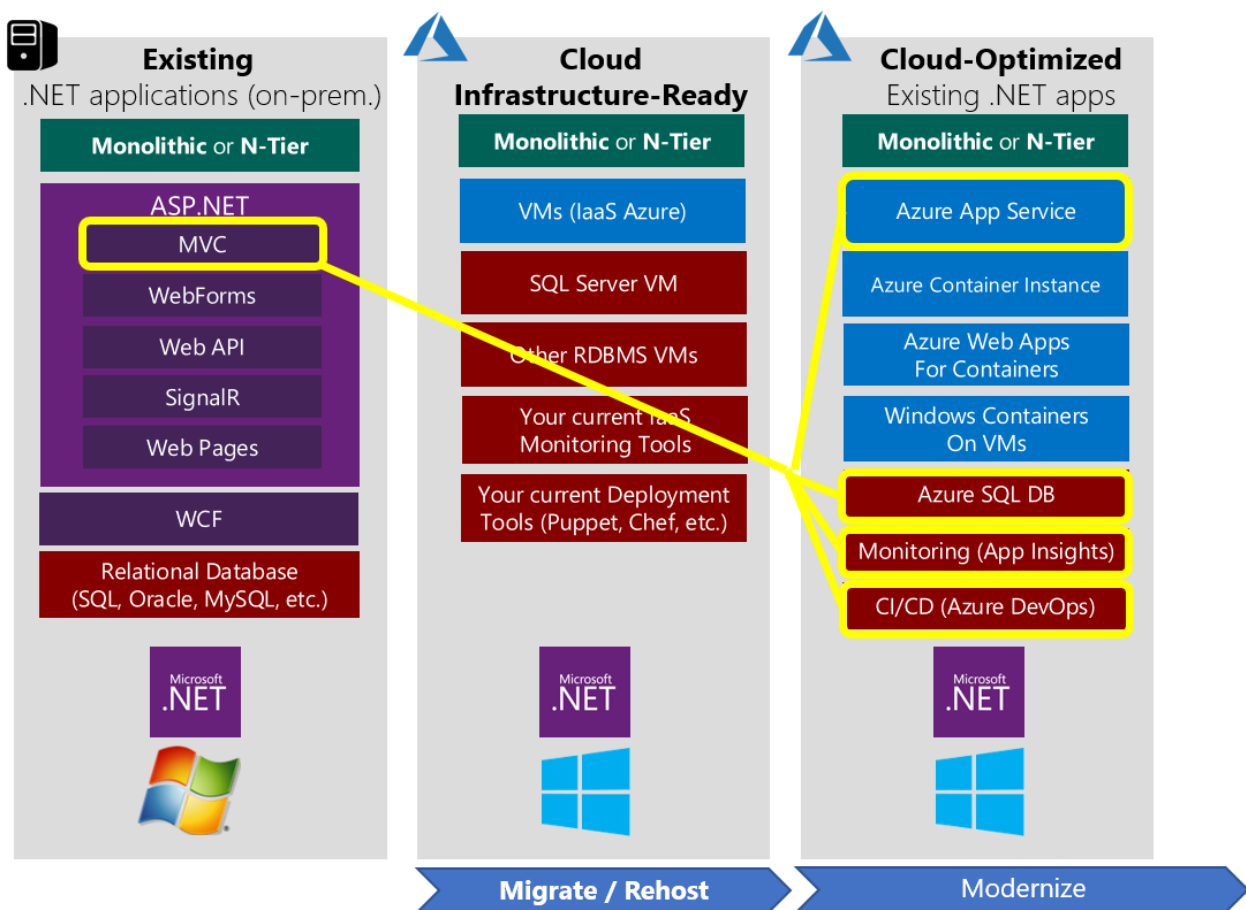
其中，对于大多数应用场景(包括简单的基于容器的应用)，推荐使用应用服务 Web 应用方法。对于微服务体系结构，请考虑使用基于容器的方法。如果需要更好地控制运行应用程序的计算机，请考虑使用 Azure 虚拟机。

### 应用服务 Web 应用

应用服务 Web 应用提供的完全托管平台针对托管 Web 应用程序进行了优化。它是一种平台即服务 (PaaS) 产品/服务，可使你专注于业务逻辑，而由 Azure 负责维护应用运行和缩放所需的基础结构。应用服务 Web 应用的一些关键功能：

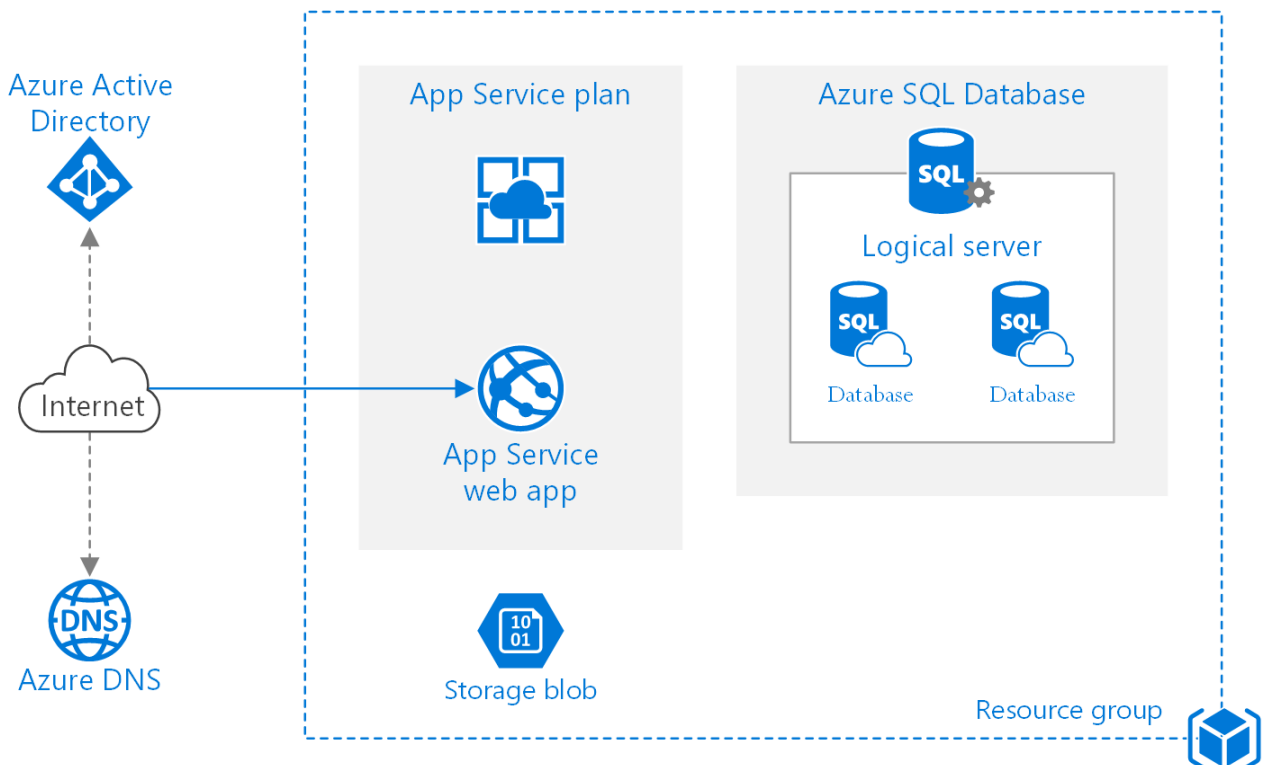
- DevOps 优化(持续集成和交付、多环境、A/B 测试和脚本支持)。
- 全球缩放和高可用性。
- 可连接至 SaaS 平台和本地数据。
- 安全性和符合性。
- Visual Studio 集成。

Azure 应用服务是适合大多数 Web 应用的最佳选择。该平台集成部署与管理，站点可快速缩放以处理高流量负载，内置负载均衡和流量管理器提供高可用性。可通过在线迁移工具将现有站点轻松移动到 Azure 应用服务、使用 Web 应用程序库中的开源应用或使用框架和你选择的工具创建新的站点。通过 WebJobs 功能可将后台作业处理轻松添加到应用服务 Web 应用。若已使用本地数据库将现有的 ASP.NET 应用程序托管在本地，则可以使用 Azure SQL 数据库将此应用顺利地迁移到应用服务 Web 应用(如果愿意，也可以通过安全访问本地数据库服务器完成迁移)。



多数情况下，将本地托管的 ASP.NET 应用迁移到应用服务 Web 应用的过程都很简单。只需对应用本身进行极少的修改甚至无需修改，它便可以快速开始利用 Azure 应用服务 Web 应用提供的众多功能。

除了未针对云进行优化的应用，Azure 应用服务 Web 应用还是适合许多简单的整体式（非分布式）应用程序（例如许多 ASP.NET Core 应用）的出色解决方案。此方法中的基本体系结构简单易懂，并且易于管理：



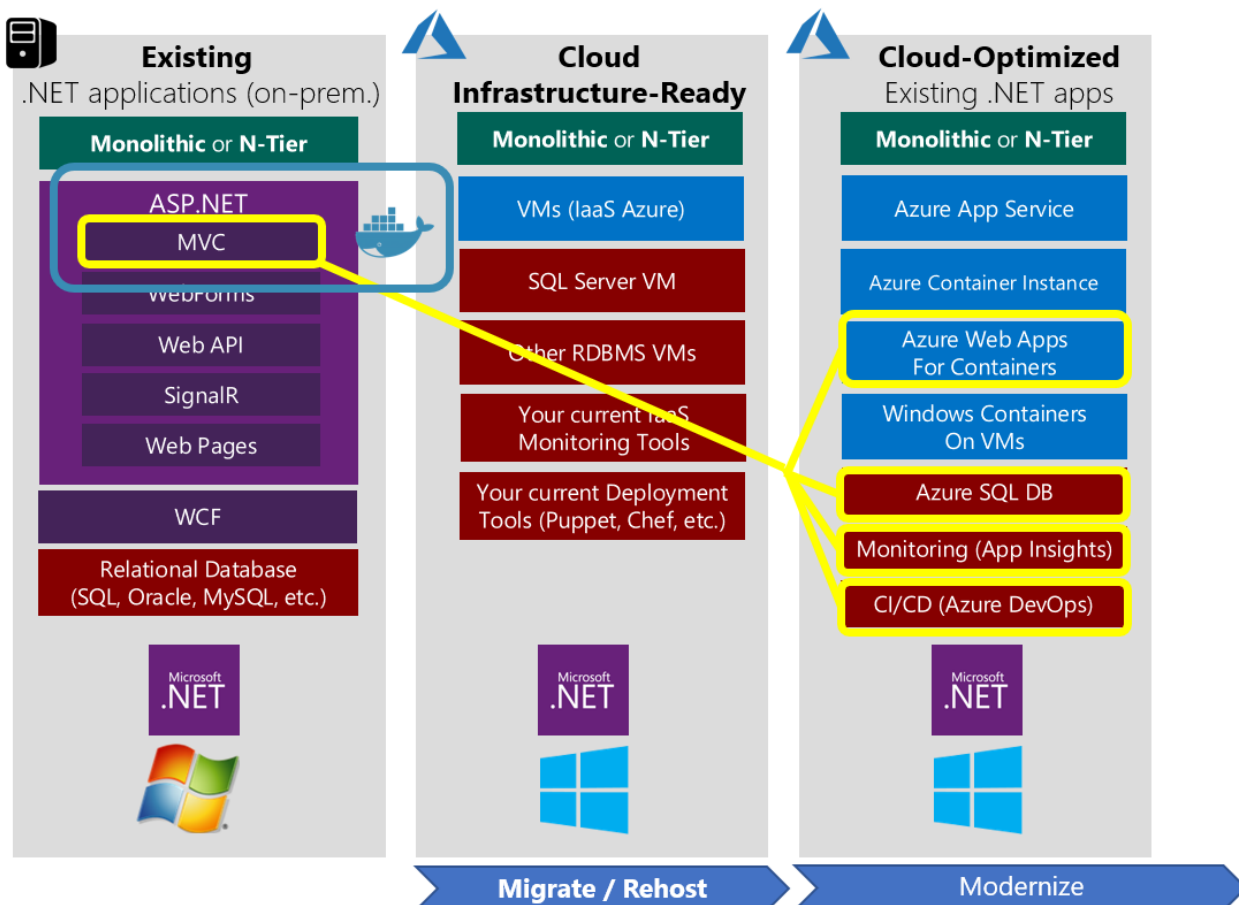
通常单个资源组中的少量资源即足以管理此类应用。这种基本体系结构方法非常适合通常部署为单个单元的应用，而不适合由许多独立进程构成的应用。尽管此方法的体系结构很简单，但仍允许托管应用纵向扩展（每个节点包含更多资源）和横向扩展（更多托管节点），来满足任何需求增长。借助自动缩放功能，可以将应用配置为根据需

求和节点上的平均负载自动调整托管应用的节点数量。

### 用于容器的应用服务 Web 应用

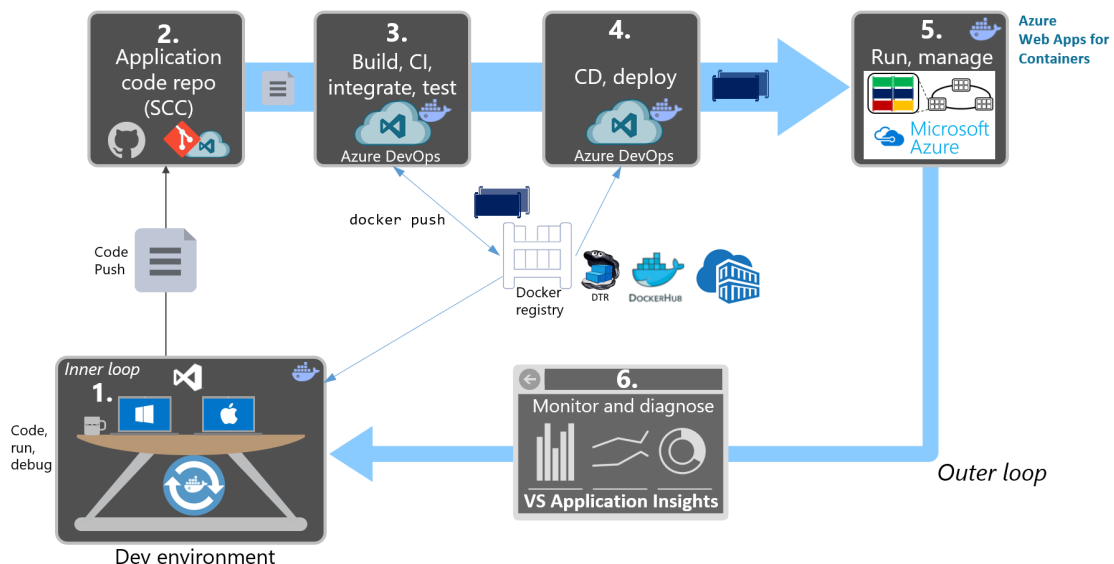
除了支持直接托管 Web 应用，[用于容器的应用服务 Web 应用](#)还可用于在 Windows 和 Linux 上运行容器化的应用程序。借助此服务，可以轻松部署和运行可随业务缩放的容器化的应用程序。此类应用具有上面列出的应用服务 Web 应用的所有功能。此外，用于容器的 Web 应用还支持使用 Docker Hub、Azure 容器注册表和 GitHub 来简化 CI/CD。你可以使用 Azure DevOps 定义将更改发布到注册表的生成和部署管道。然后可以在过渡环境中测试这些更改，并使用部署槽位自动将其部署到生产环境，无需停机即可完成升级。还可以同样轻松地回滚到以前的版本。

在某些应用场景中，用于容器的 Web 应用是最理想的选择。如果现有应用可以容器化，则无论是在 Windows 容器还是 Linux 容器中，都可以使用此工具集轻松托管这些应用。只需发布容器，然后将用于容器的 Web 应用配置为从所选注册表中拉取该映像的最新版本。这是从经典应用托管模型迁移到云优化模型的“直接迁移”方法。



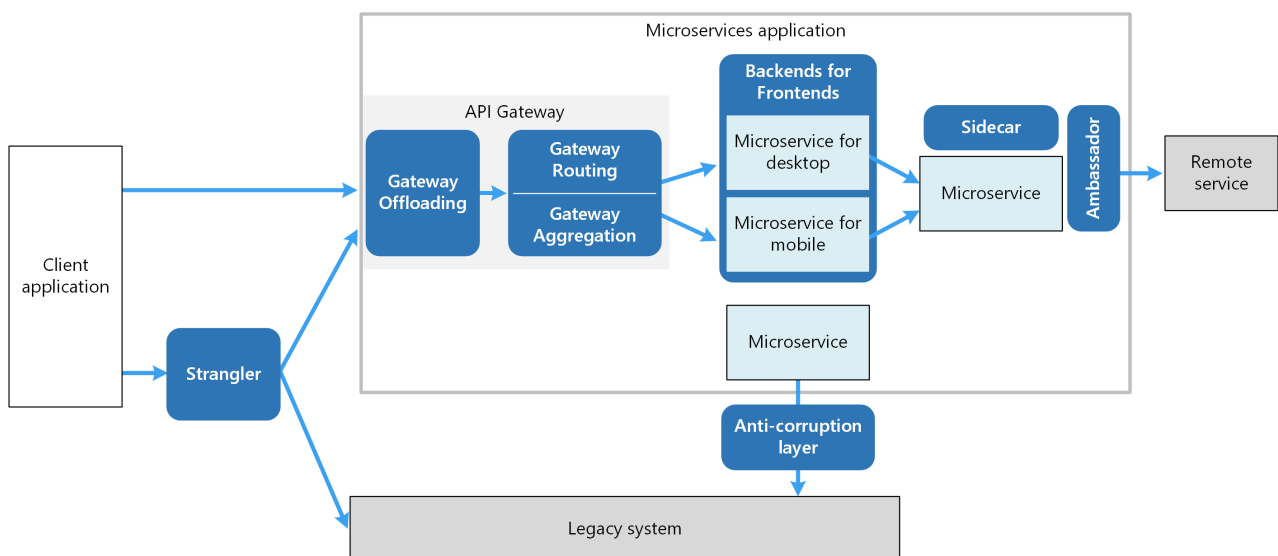
如果开发团队能够转为使用基于容器的开发过程，这种方法也很有效。使用容器开发应用的“内部循环”包括使用容器生成应用。对代码和容器配置所做的更改将推送到源代码管理，自动生成功能负责将新容器映像发布到 Docker Hub 或 Azure 容器注册表等注册表。然后，这些映像将用作其他开发以及生产部署的基础，如下图所示：

# End to End Docker DevOps Lifecycle Workflow



使用容器进行开发具有许多优点，特别是在生产环境中使用容器时。在运行应用的每个环境(从本地开发计算机到生成和测试系统再到生产)中，用于托管应用的容器配置是相同的。这大大降低了由于计算机配置或软件版本的差异而导致缺陷的可能性。开发人员还可以使用他们最有效的工具，包括操作系统，因为容器可以在任何 OS 上运行。在某些情况下，涉及多个容器的分布式应用程序在单个开发计算机上运行时可能非常耗费资源。在这种情况下，可能适合升级为使用 Kubernetes 和 Azure Dev Spaces，下一节将对此进行介绍。

随着较大型应用程序的各个部分被分解为其各自较小的、独立的微服务，可以使用其他设计模式来改善应用的行为。可以使用 API 网关来简化访问并将客户端与其后端分离，而不是直接使用各个服务。通过为不同的前端提供单独的服务后端，还可以让服务随其使用者而演化。可以通过单独的 *sidecar* 容器访问公共服务，该容器可能包含使用 *ambassador* 模式的公共客户端连接库。



详细了解在生成基于微服务的系统时要考虑的设计模式。

## Azure Kubernetes 服务

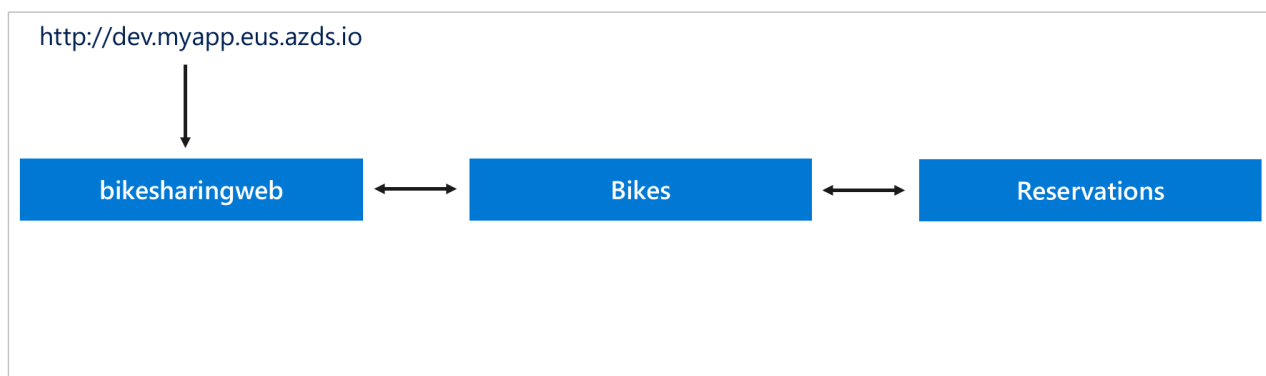
Azure Kubernetes 服务 (AKS) 管理托管的 Kubernetes 环境，即使没有容器业务流程专业知识，也可快速轻松地部署和管理容器化应用程序。此外，它还通过预配、升级和按需缩放资源，消除了持续操作和维护的负担，而无需应用程序脱机。

通过将大量责任转移至 Azure，AKS 降低了管理 Kubernetes 群集的复杂性和运营开销。作为一项托管的 Kubernetes 服务，Azure 将为你处理运行状况监视和维护等关键任务。并且，你仅需支付群集中的代理节点，而无需支付主节点。作为一项托管的 Kubernetes 服务，AKS 提供以下功能：

- 自动化的 Kubernetes 版本升级和修补。
- 轻松的群集缩放。
- 自我修复型托管控制平面(主节点)。
- 节省成本:仅支付运行的代理池节点。

由于 Azure 会负责管理 AKS 群集中的节点, 因此, 很多任务(例如群集升级)不再需要手动执行。由于 Azure 会处理这些关键维护任务, 因此, AKS 不提供对群集的直接访问(例如使用 SSH)。

使用 AKS 的团队也可以使用 Azure Dev Spaces。Azure Dev Spaces 允许团队直接使用他们在 AKS 中运行的整个微服务体系结构或应用程序, 帮助团队专注于微服务应用程序的开发和快速迭代。Azure Dev Spaces 还提供一种独立更新微服务体系结构各部分的方法, 而不会影响 AKS 群集的其余部分或其他开发人员。



Azure Dev Spaces:

- 最大限度地减少本地计算机设置时间和资源要求
- 允许团队更快地循环访问
- 减少团队所需的集成环境数量
- 进行开发/测试时, 无需在分布式系统中模拟某些服务

[详细了解 Azure Dev Spaces](#)

### Azure 虚拟机

如果现有应用程序需要经过大量修改才可在应用服务中运行, 为简化迁移到云的操作, 可选择虚拟机。然而, 与 Azure 应用服务相比, 正确配置、保护和维护 VM 需要更多时间和 IT 专业知识。如要使用 Azure 虚拟机, 请务必考虑到 VM 环境的修补、更新和管理所需的持续性维护工作。Azure 虚拟机属于基础结构即服务 (IaaS), 而应用服务属于 PaaS。还应考虑将应用作为 Windows 容器部署到用于容器的 Web 应用是否可能成为方案的可行选项。

## 逻辑进程

对于可从应用程序的剩余部分分离的单独逻辑进程, 可以“无服务器”方式将其独立部署到 Azure Functions。通过 Azure Functions 可编写解决特定问题所需的代码, 无需担心运行它的应用程序或基础结构。可选择各种编程语言, 包括 C#、F#、Node.js、Python 和 PHP, 以便你选择一种最高效语言来处理手头任务。与多数基于云的解决方案一样, 仅需为使用的时间量支付费用, 而且 Azure Functions 按需扩展非常可靠。

## 数据

Azure 提供多种数据存储选择, 以便应用程序可使用恰当的数据提供程序处理相关数据。

对于事务和关系数据, Azure SQL 数据库是最佳选择。对于以读取为主的高性能数据, 受 Azure SQL 数据库支持的 Redis 缓存是一个不错的解决方案。

非结构化 JSON 数据可用多种方式进行存储, 包括 SQL 数据库列、Blob、Azure 存储中的表, 以及 Azure Cosmos DB。其中, Azure Cosmos DB 可提供最佳的查询功能, 建议用于必须支持查询的大量基于 JSON 的文档。

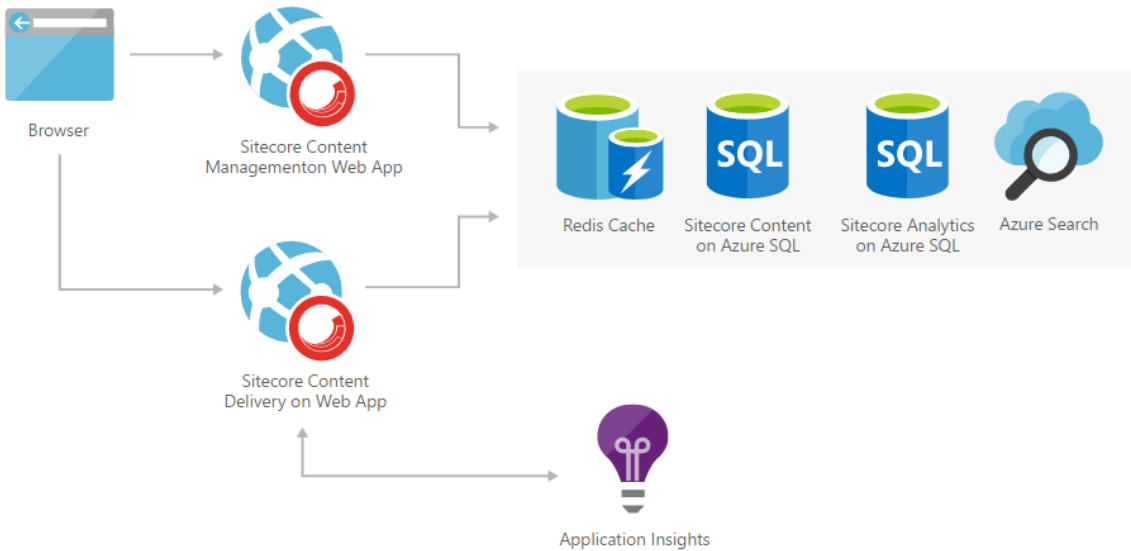
暂时命令或者用于安排应用程序行为的基于事件的数据可使用 Azure 服务总线或 Azure 存储队列。Azure 存储总

线具有更强的灵活性, 建议将该服务用于处理应用程序内和应用程序间的非普通消息传递。

## 体系结构建议

应用程序要求应决定其体系结构。有多种不同的 Azure 服务可供使用。选择正确的服务是一项重要决定。Microsoft 提供一系列参考体系结构, 帮助用户确定针对常见方案优化的典型体系结构。可找到一个与应用程序要求高度契合或至少提供一个出发点的参考体系结构。

图 11-1 显示了一个示例参考体系结构。该图介绍了一个建议用于为市场营销而优化的 Sitecore 内容管理系统网站的体系结构方式。



**Figure 11-1.** Sitecore 市场营销网站参考体系结构。

### 参考 : Azure 托管建议

- Azure 解决方案体系结构  
<https://azure.microsoft.com/solutions/architecture/>
- Azure 基本 Web 应用程序体系结构  
<https://docs.microsoft.com/azure/architecture/reference-architectures/app-service-web-app/basic-web-app>
- 微服务设计模式  
<https://docs.microsoft.com/azure/architecture/microservices/design/patterns>
- Azure 开发人员指南  
<https://azure.microsoft.com/campaigns/developer-guide/>
- Web 应用概述  
<https://docs.microsoft.com/azure/app-service/app-service-web-overview>
- 用于容器的 Web 应用  
<https://azure.microsoft.com/services/app-service/containers/>
- Azure Kubernetes 服务 (AKS) 简介  
<https://docs.microsoft.com/azure/aks/intro-kubernetes>