Arquiteturas comuns de aplicativo Web

Artigo15/12/202111 minutos para o fim da leitura2 colaboradores

Neste artigo

- O que é um aplicativo monolítico?
- Aplicativos todos-em-um
- O que são camadas?
- Aplicativos tradicionais da arquitetura de "N Camadas"
- Arquitetura limpa
- Contêineres e aplicativos monolíticos
- Suporte ao Docker

> "Se você acha que uma boa arquitetura é cara, tente usar uma arquitetura ruim." - *Brian Foote e Joseph Yoder*

A maioria dos aplicativos .NET tradicionais é implantada como unidades individuais correspondentes a um executável ou a um único aplicativo Web em execução em um único AppDomain do IIS. Essa abordagem é o modelo de implantação mais simples e fornece muitos aplicativos públicos internos e menores muito bem. No entanto, mesmo considerando essa única unidade de implantação, a maioria dos aplicativos de negócios não triviais se beneficia de uma separação lógica em várias camadas.

O que é um aplicativo monolítico?

Um aplicativo monolítico é aquele que é totalmente autossuficiente, em termos de comportamento. Ele pode interagir com outros serviços ou armazenamentos de dados durante a execução de suas operações, mas o núcleo de seu comportamento é executado em seu próprio processo e o aplicativo inteiro normalmente é implantado como uma única unidade. Se um aplicativo desse tipo precisar ser dimensionado horizontalmente, em geral, o aplicativo inteiro será duplicado em vários servidores ou máquinas virtuais.

Aplicativos todos-em-um

O menor número possível de projetos para uma arquitetura de aplicativo é um. Nessa arquitetura, toda a lógica do aplicativo está contida em um único projeto, compilada em um único assembly e implantada como uma única unidade.

Um novo projeto ASP.NET Core, seja ele criado no Visual Studio ou por meio da linha de comando, começa como um simples monólito "todos-em-um". Ele contém todo o comportamento do aplicativo, incluindo a lógica de apresentação, de negócios e de acesso a dados. A Figura 5-1 mostra a estrutura de arquivos de um aplicativo de projeto único.

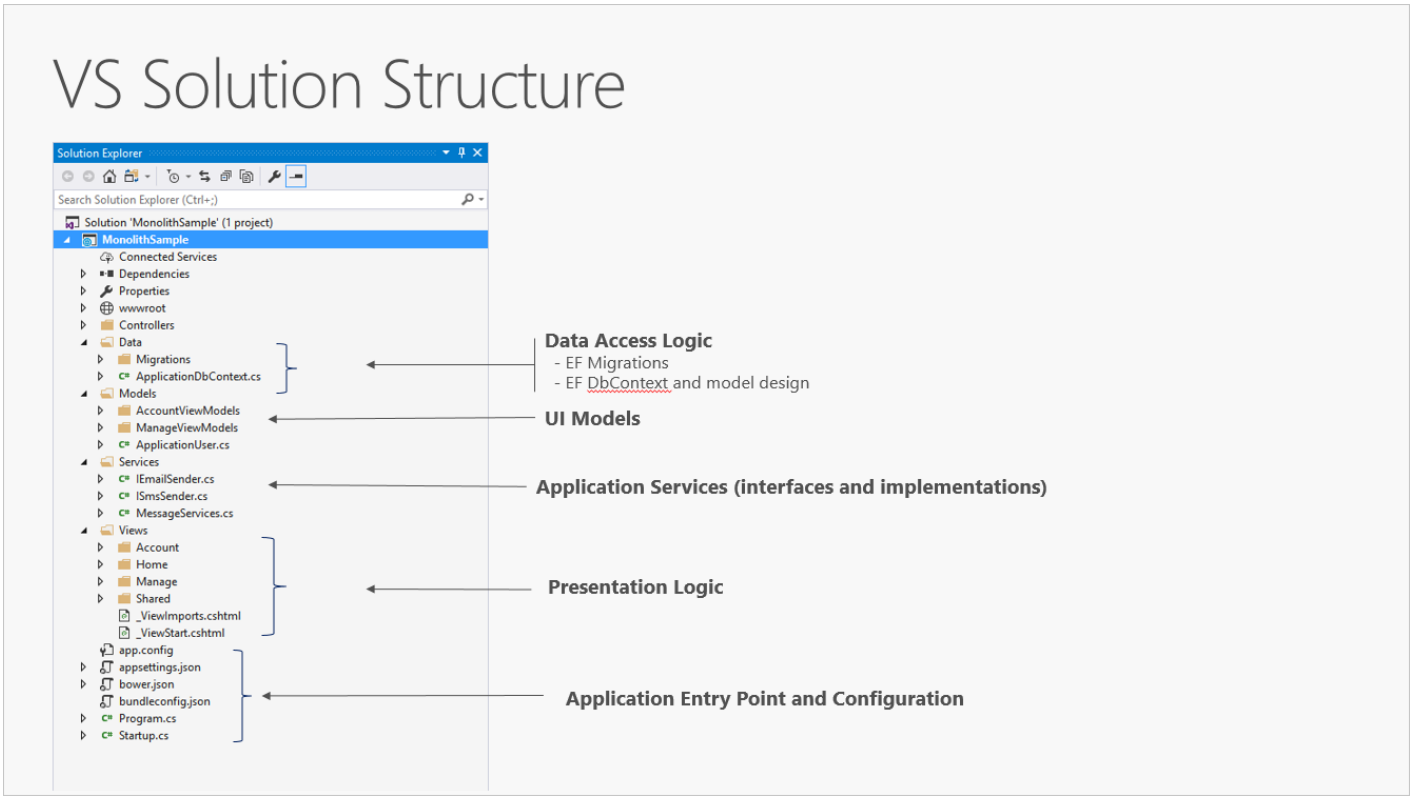


Figura 5-1. Um único projeto de aplicativo ASP.NET Core.

Em um cenário de projeto único, a separação de interesses é obtida com o uso de pastas. O modelo padrão inclui pastas separadas para as responsabilidades do padrão MVC de Modelos, Exibições e Controladores, bem como pastas adicionais para Dados e Serviços. Nessa disposição, os detalhes de apresentação devem ser limitados tanto quanto possível à pasta Views e os detalhes de implementação de acesso a dados devem ser limitados às classes mantidas na pasta Data. A lógica de negócios deve residir nos serviços e nas classes dentro da pasta Models.

Embora simples, a solução monolítica de projeto único traz algumas desvantagens. À medida que o tamanho e a complexidade do projeto aumentam, o número de arquivos e pastas continuará crescendo também. Os interesses de UI (interface do usuário) (modelos, exibições, controladores) residem em várias pastas, que não são agrupadas em ordem alfabética. Esse problema só fica pior quando constructos adicionais no nível da interface do usuário, como Filters ou ModelBinders, são adicionados em suas próprias pastas. A lógica de negócios é distribuída entre as pastas Modelos e Serviços e não há nenhuma indicação clara de quais classes em quais pastas devem depender de qual delas. Essa falta de organização no nível do projeto costuma levar ao código espaguete.

Para resolver esses problemas, os aplicativos geralmente evoluem para soluções de vários projetos, em que cada projeto é considerado um residente de determinada *camada* do aplicativo.

O que são camadas?

Conforme a complexidade dos aplicativos aumenta, uma maneira de gerenciar essa complexidade é dividir o aplicativo de acordo com suas responsabilidades ou interesses. Essa abordagem segue a separação do princípio de preocupações e pode ajudar a manter uma base de código crescente organizada para que os desenvolvedores possam encontrar facilmente onde determinadas funcionalidades são implementadas. Apesar disso, a arquitetura em camadas oferece inúmeras vantagens, além de apenas a organização do código.

Com a organização do código em camadas, a funcionalidade comum de baixo nível pode ser reutilizada em todo o aplicativo. Essa reutilização é útil porque significa que menos código precisa

ser escrito e ainda pode permitir que o aplicativo seja padronizado em uma única implementação, seguindo o princípio DRY (Don't Repeat Yourself).

Com uma arquitetura em camadas, os aplicativos podem impor restrições sobre quais camadas podem se comunicar com outras camadas. Essa arquitetura ajuda a atingir o encapsulamento. Quando uma camada é alterada ou substituída, somente as camadas que trabalham com ela devem ser afetadas. Ao limitar quais camadas dependem de outras camadas, o impacto das alterações pode ser reduzido, de modo que uma única alteração não afete todo o aplicativo.

As camadas (e o encapsulamento) facilitam grande parte da substituição da funcionalidade dentro do aplicativo. Por exemplo, um aplicativo inicialmente pode usar seu próprio banco de dados do SQL Server para persistência, mas, posteriormente, pode optar por usar uma estratégia de persistência baseada em nuvem ou uma protegida por uma API Web. se o aplicativo encapsulasse corretamente sua implementação de persistência dentro de uma camada lógica, essa camada específica de SQL Server poderia ser substituída por uma nova implementação da mesma interface pública.

Além do potencial de alternância de implementações em resposta a alterações futuras nos requisitos, as camadas de aplicativo também podem facilitar a alternância de implementações para fins de teste. Em vez da necessidade de gravar testes que operam na camada de dados reais ou na camada de interface do usuário do aplicativo, essas camadas podem ser substituídas em tempo de teste com implementações fictícias que fornecem respostas conhecidas a solicitações. Essa abordagem geralmente torna os testes muito mais fáceis de escrever e muito mais rápidos para serem executados em comparação com os testes em execução na infraestrutura real do aplicativo.

A disposição em camadas lógicas é uma técnica comum para melhorar a organização do código em aplicativos de software empresariais, e há várias maneiras pelas quais o código pode ser organizado em camadas.

Observação

As *camadas* representam uma separação lógica dentro do aplicativo. Caso a lógica do aplicativo seja fisicamente distribuída em servidores ou processos separados, esses destinos de implantação física separados são chamados de *camadas*. É possível, e bastante comum, ter um aplicativo de N-Camadas que é implantado em uma única camada.

Aplicativos tradicionais da arquitetura de "N Camadas"

A organização mais comum da lógica do aplicativo em camadas é mostrada na Figura 5-2.

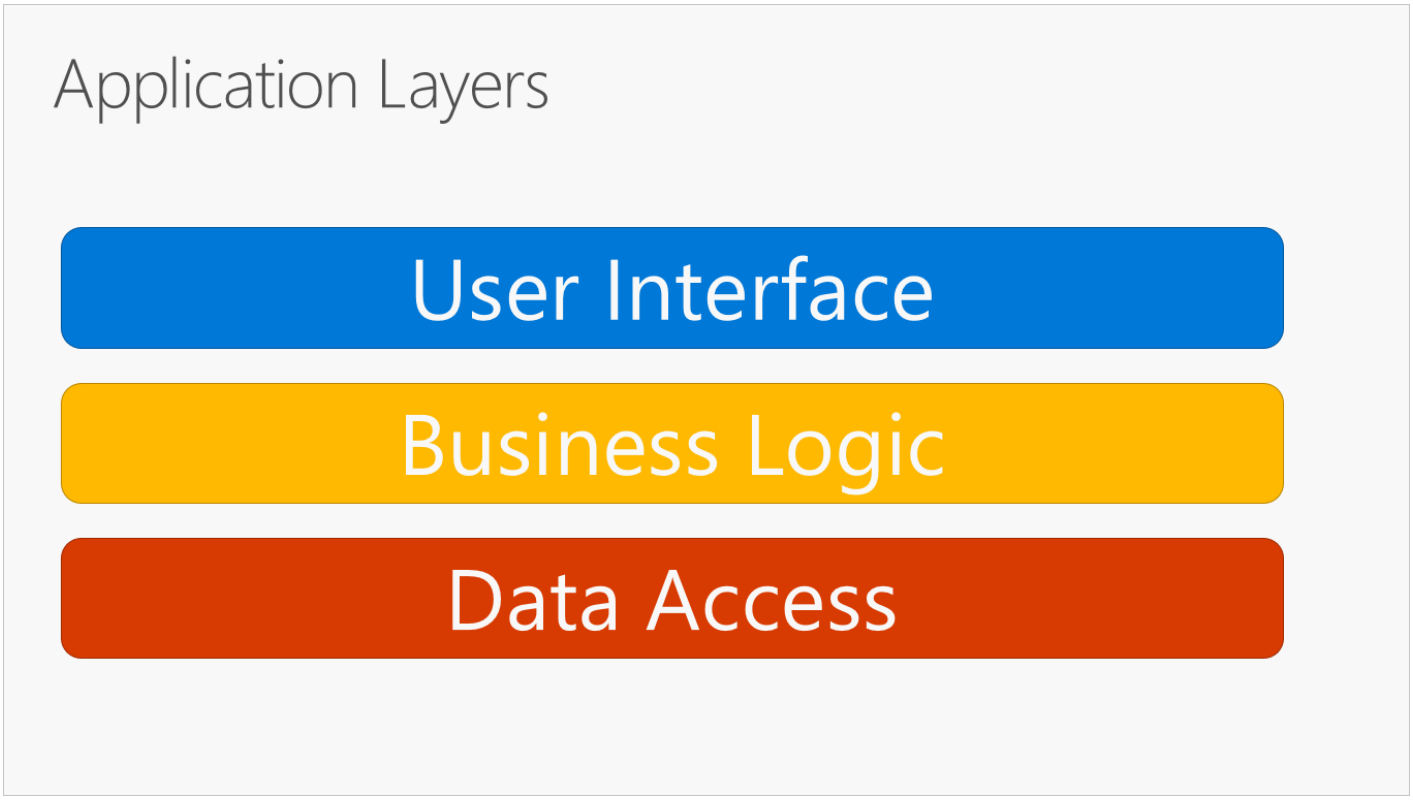


Figura 5-2. Camadas de aplicativo típicas.

Essas camadas são frequentemente abreviadas como interface do usuário, BLL (Camada de Lógica de Negócios) e DAL (Camada de Acesso a Dados). Usando essa arquitetura, os usuários fazem solicitações por meio da camada de interface do usuário, que interage com a BLL. A BLL, por sua vez, pode chamar a DAL para solicitações de acesso a dados. A camada de interface do usuário não deve fazer solicitações à DAL diretamente nem interagir com persistência diretamente por outros meios. Da mesma forma, a BLL só deve interagir com persistência por meio da DAL. Assim, cada camada tem sua própria responsabilidade conhecida.

Uma desvantagem dessa abordagem tradicional de disposição em camadas é que as dependências em tempo de compilação são executadas de cima para baixo. Ou seja, a camada de interface do usuário depende da BLL, que depende da DAL. Isso significa que a BLL, que normalmente contém a lógica mais importante no aplicativo, depende dos detalhes de implementação de acesso a dados (e geralmente da existência de um banco de dados). O teste da lógica de negócios em uma arquitetura como essa costuma ser difícil, exigindo um banco de dados de teste. O princípio da inversão de dependência pode ser usado para resolver esse problema, como você verá na próxima seção.

A Figura 5-3 mostra uma solução de exemplo, que divide o aplicativo em três projetos por responsabilidade (ou camada).

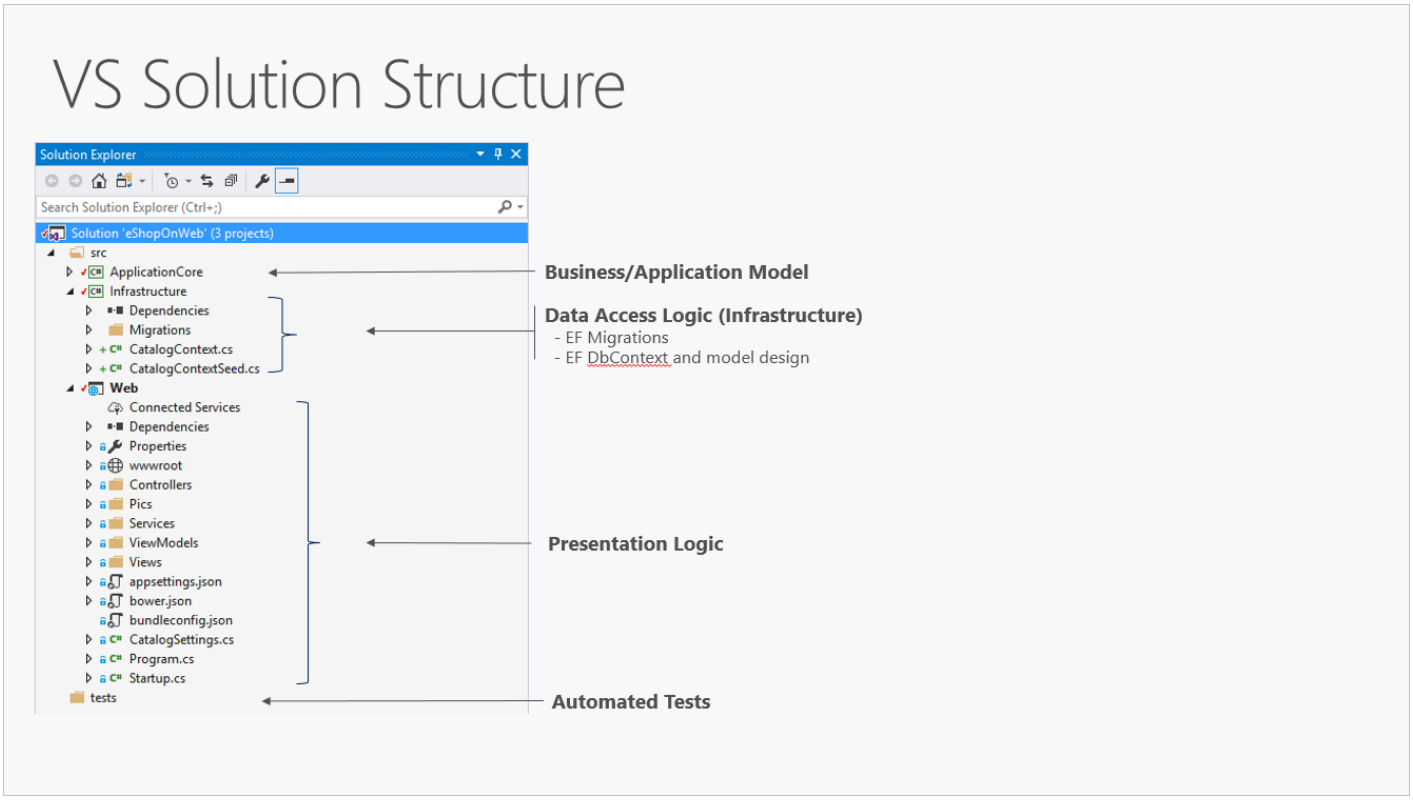


Figura 5-3. Um aplicativo monolítico simples com três projetos.

Embora esse aplicativo use vários projetos para fins de organização, ele ainda é implantado como uma única unidade e seus clientes interagirão com ele como um único aplicativo Web. Isso possibilita um processo de implantação muito simples. A Figura 5-4 mostra como um aplicativo desse tipo pode ser hospedado usando o Azure.

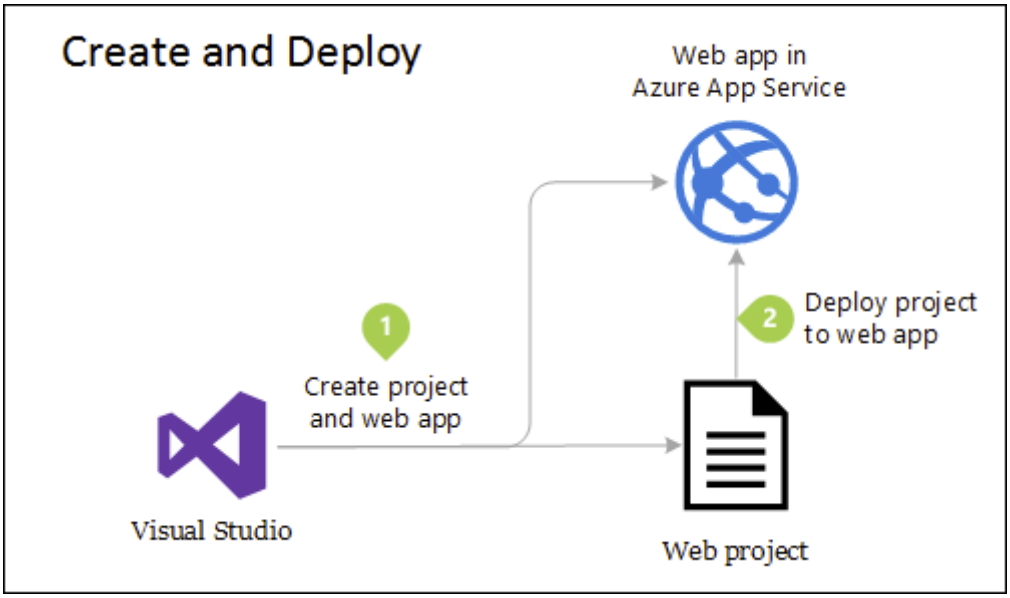


Figura 5-4. Implantação simples do Aplicativo Web do Azure

Conforme o aplicativo precisar ser aumentado, soluções de implantação mais robustas e complexas poderão ser necessárias. A Figura 5-5 mostra um exemplo de um plano de implantação mais complexo compatível com funcionalidades adicionais.

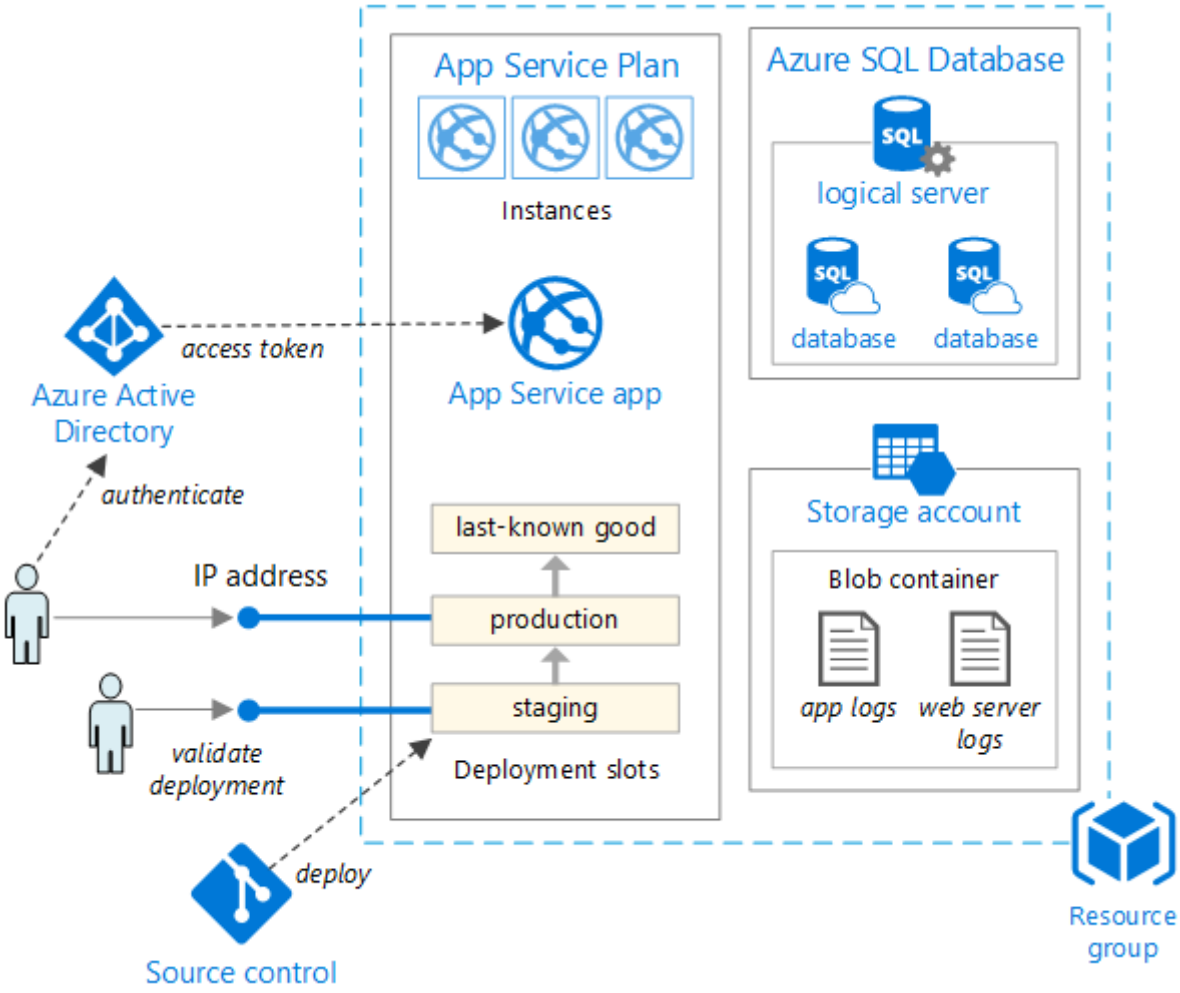


Figura 5-5. Implantando um aplicativo Web em um Serviço de Aplicativo do Azure

Internamente, a organização desse projeto em vários projetos com base na responsabilidade melhora a facilidade de manutenção do aplicativo.

Essa unidade pode ser escalada verticalmente ou expandida para aproveitar a escalabilidade sob demanda baseada em nuvem. Escalar verticalmente significa adicionar mais CPU, memória, espaço em disco ou outros recursos aos servidores que hospedam o aplicativo. Escalar horizontalmente significa adicionar mais instâncias desses servidores, sejam eles servidores físicos, máquinas virtuais ou contêineres. Quando o aplicativo é hospedado em várias instâncias, um balanceador de carga é usado para atribuir solicitações a instâncias individuais do aplicativo.

A abordagem mais simples para dimensionar um aplicativo Web no Azure é configurar o dimensionamento manualmente no Plano do Serviço de Aplicativo do aplicativo. A Figura 5-6 mostra a tela apropriada do painel do Azure para configurar a quantidade de instâncias que atendem um aplicativo.

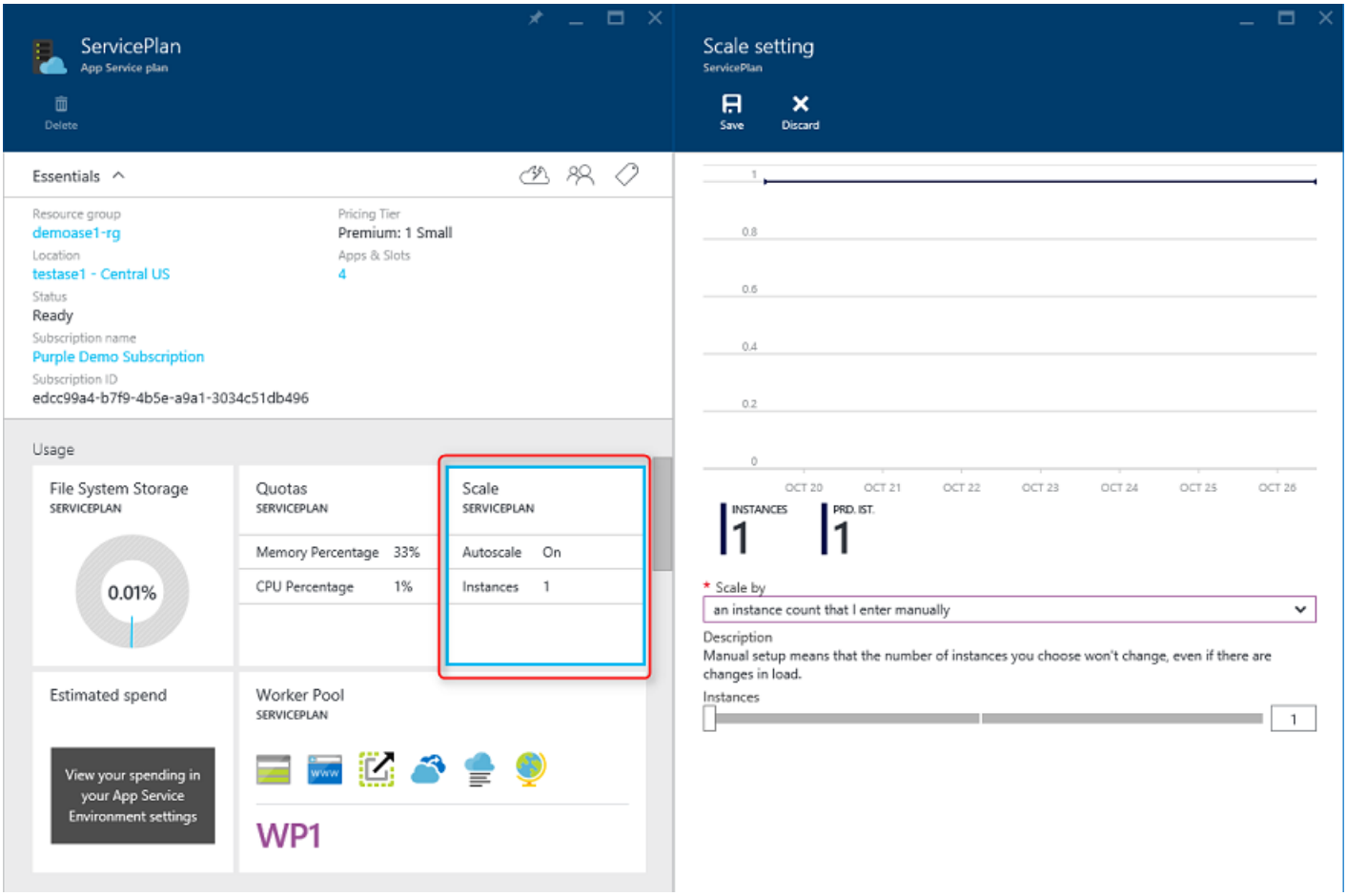


Figura 5-6. Dimensionamento do Plano do Serviço de Aplicativo no Azure.

Arquitetura limpa

Os aplicativos que seguem o Princípio da Inversão de Dependência, bem como os princípios de DDD (Design Controlado por Domínio), tendem a chegar a uma arquitetura semelhante. Essa arquitetura foi conhecida por muitos nomes ao longo dos anos. Um dos primeiros nomes foi Arquitetura Hexagonal, seguido por Portas e Adaptadores. Mais recentemente, ela é citada como a Arquitetura Cebola ou Arquitetura Limpa. O último nome, Arquitetura Limpa, é usado como o nome dessa arquitetura neste livro eletrônico.

O aplicativo de referência eShopOnWeb usa a abordagem de arquitetura limpa para organizar seu código em projetos. você pode encontrar um modelo de solução que pode ser usado como um ponto de partida para suas próprias soluções de ASP.NET Core no repositório de GitHub ardalis/cleanarchitecture ou instalando o modelo do NuGet.

A arquitetura limpa coloca a lógica de negócios e o modelo de aplicativo no centro do aplicativo. Em vez de fazer com que a lógica de negócios dependa do acesso a dados ou de outros interesses da infraestrutura, essa dependência é invertida: os detalhes de implementação e a infraestrutura dependem do Núcleo do Aplicativo. Essa funcionalidade é obtida definindo-se abstrações, ou interfaces, no núcleo do aplicativo, que são implementadas por tipos definidos na camada de infraestrutura. Uma maneira comum de visualizar essa arquitetura é usar uma série de círculos concêntricos, semelhantes a uma cebola. A Figura 5-7 mostra um exemplo desse estilo de representação de arquitetura.

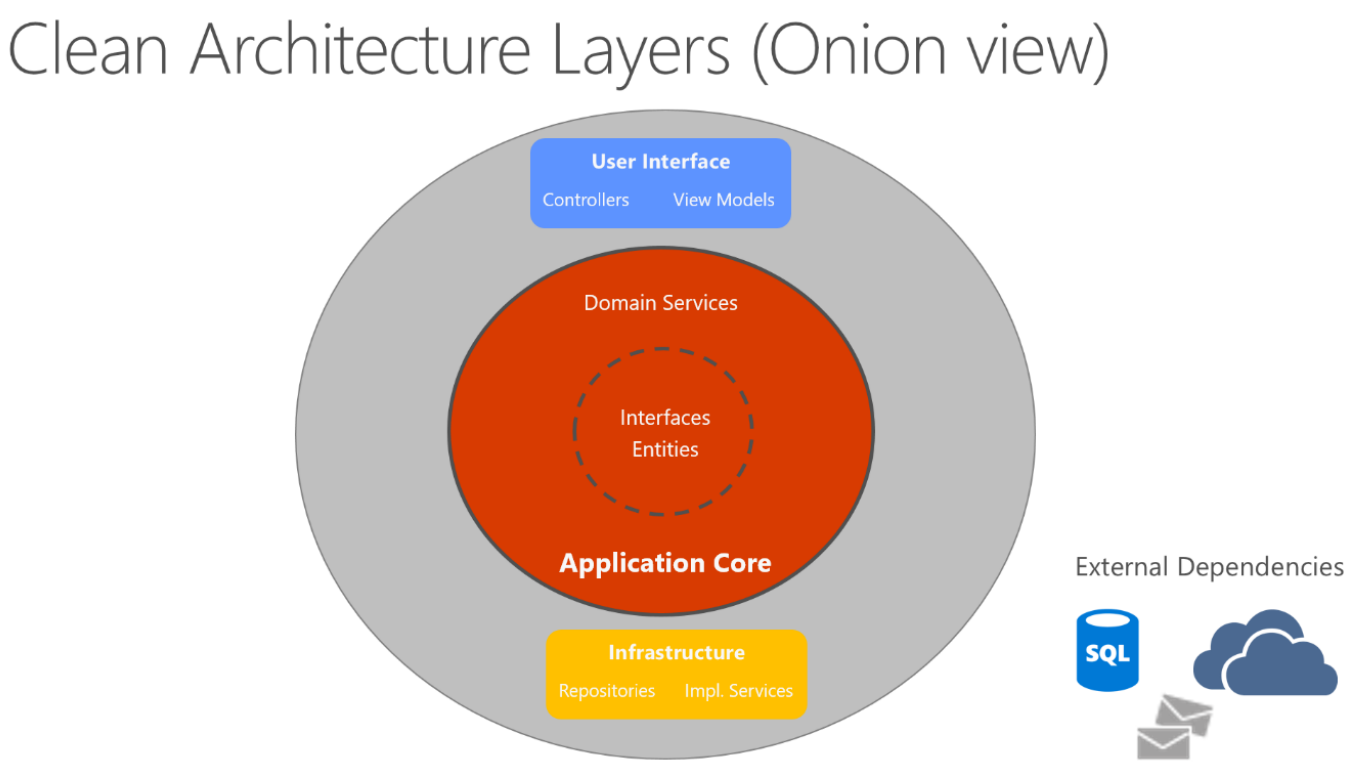


Figura 5-7. Arquitetura Limpa; exibição de cebola

Nesse diagrama, as dependências fluem para o círculo interno. O núcleo do aplicativo obtém seu nome por meio de sua posição no centro desse diagrama. E você pode ver no diagrama que o Núcleo do Aplicativo não tem dependências de outras camadas do aplicativo. As entidades e as interfaces do aplicativo ficam bem no centro. Fora dele, mas ainda no Núcleo do Aplicativo, estão os serviços de domínio, que normalmente implementam interfaces definidas no círculo interno. Fora do Núcleo do Aplicativo, as camadas de Interface do Usuário e de Infraestrutura dependem do Núcleo do Aplicativo, mas não (necessariamente) uma da outra.

A Figura 5-8 mostra um diagrama de camada horizontal mais tradicional que reflete melhor a dependência entre a interface do usuário e as outras camadas.

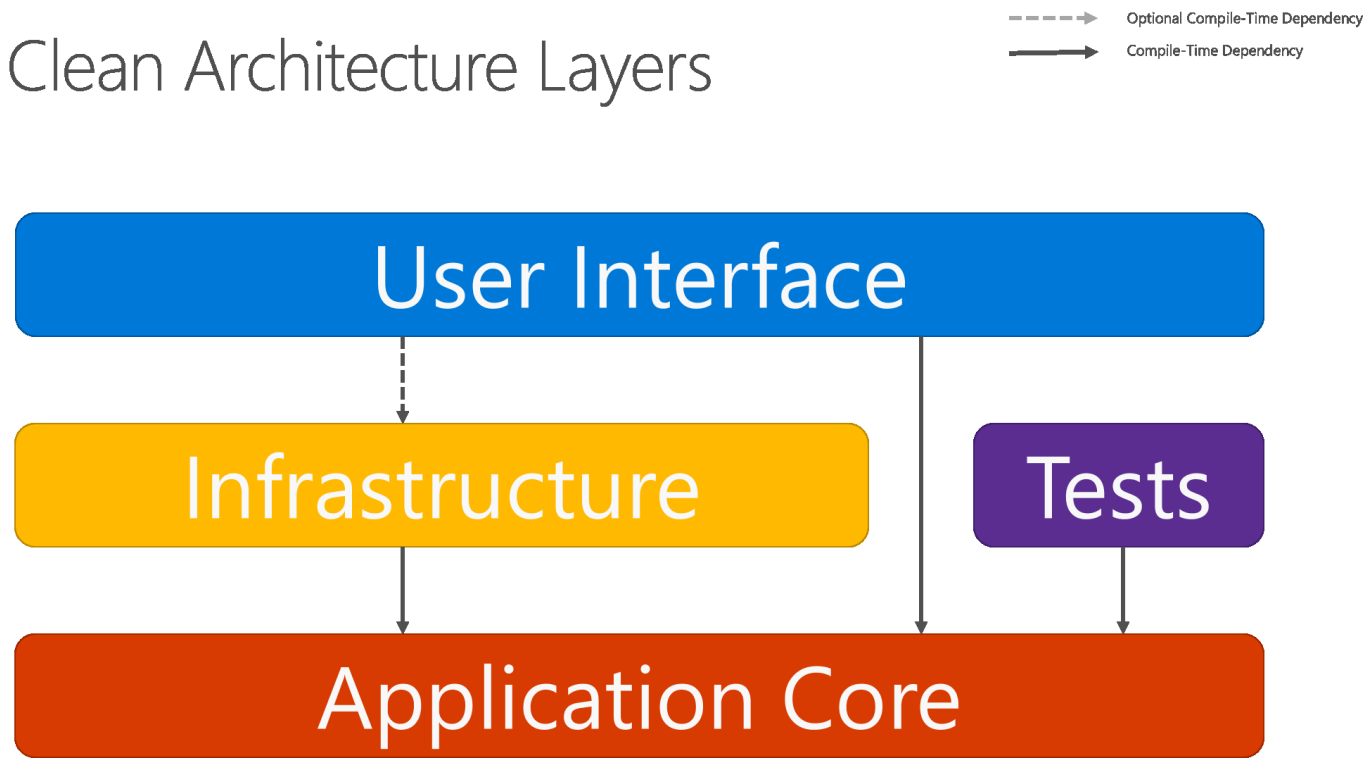


Figura 5-8. Arquitetura Limpa; exibição de camada horizontal

Observe que as setas sólidas representam as dependências em tempo de compilação, enquanto a seta tracejada representa uma dependência somente em runtime. Usando a arquitetura limpa, a camada de interface do usuário funciona com as interfaces definidas no Núcleo do Aplicativo no tempo de compilação. O ideal é que ela não tenha conhecimento dos tipos de implementação definidos na camada de Infraestrutura. No entanto, no tempo de execução, esses tipos de implementação serão necessários para a execução do aplicativo e, portanto, precisam estar presentes e conectados às interfaces do Núcleo do Aplicativo por meio da injeção de dependência.

A Figura 5-9 mostra uma exibição mais detalhada da arquitetura de um aplicativo ASP.NET Core quando criado seguindo essas recomendações.

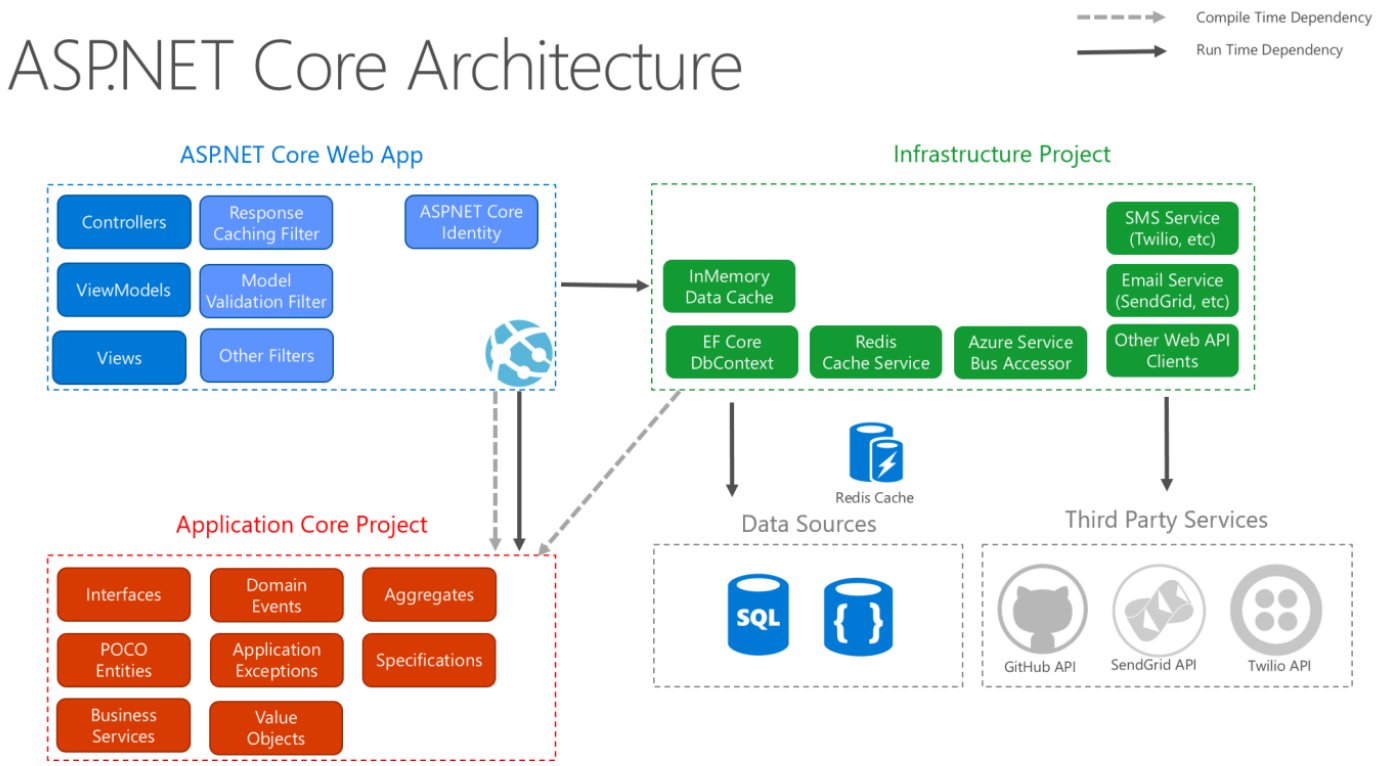


Figura 5-9. Diagrama da arquitetura do ASP.NET Core que segue a Arquitetura Limpa.

Como o Núcleo do Aplicativo não depende da Infraestrutura, é muito fácil escrever testes de unidade automatizados para essa camada. A Figuras 5-10 e 5-11 mostram como os testes se enquadram nessa arquitetura.

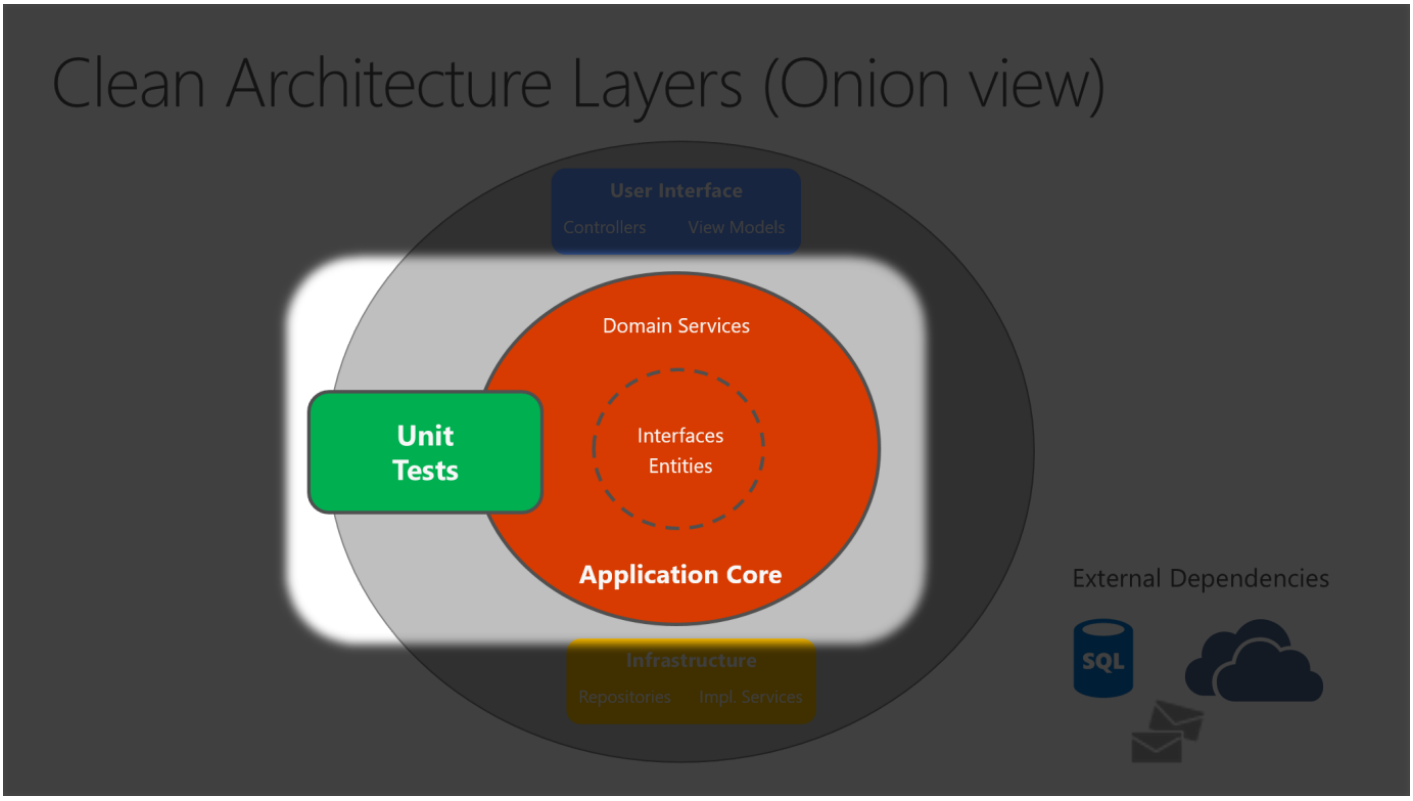


Figura 5-10. Realizando o teste de unidade do Núcleo do Aplicativo em isolamento.

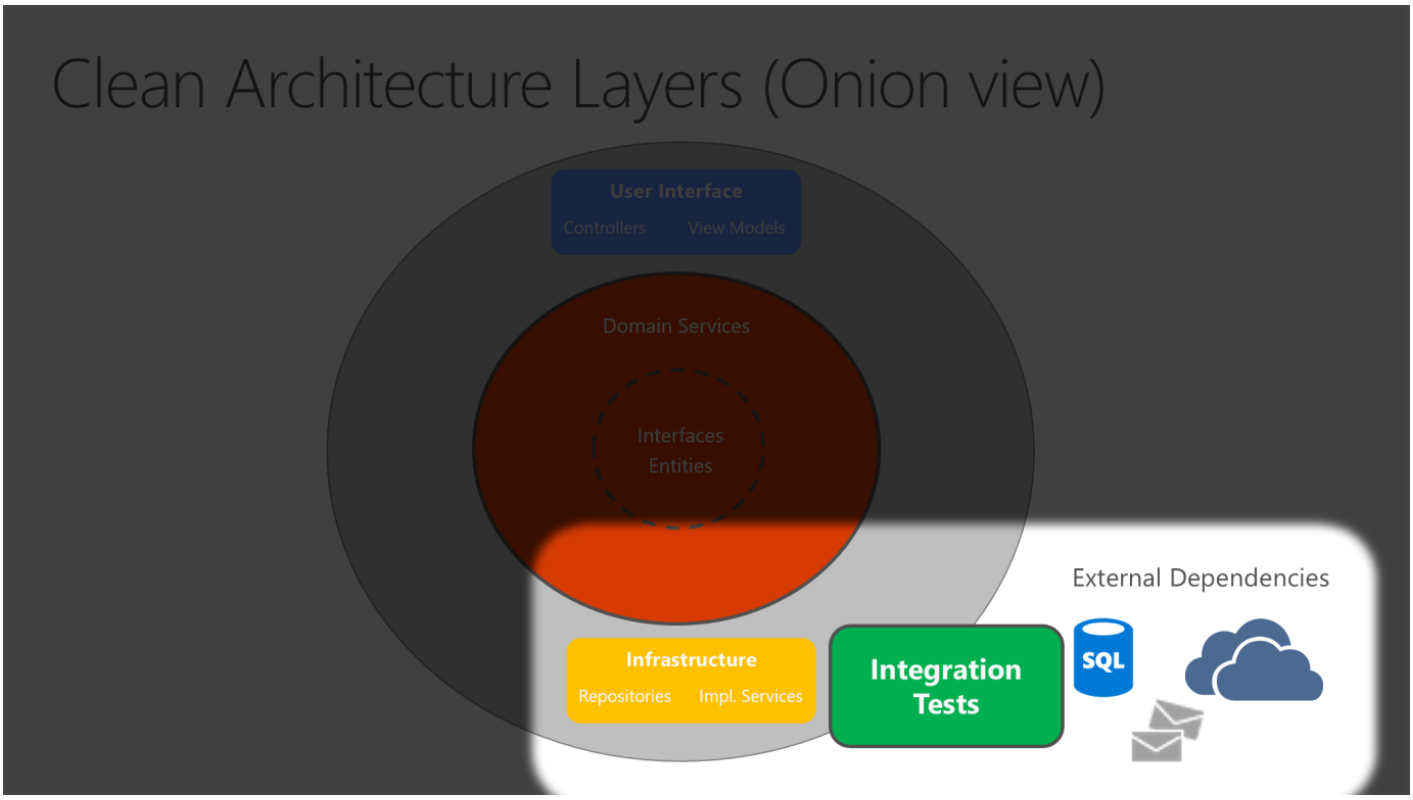


Figura 5-11. Realizando o teste de integração de implementações de Infraestrutura com dependências externas.

Como a camada de interface do usuário não tem nenhuma dependência direta dos tipos definidos no projeto de Infraestrutura, da mesma forma, é muito fácil alternar entre implementações, para facilitar o teste ou em resposta a alterações nos requisitos do aplicativo. O uso interno do ASP.NET Core e o suporte à injeção de dependência torna essa arquitetura a maneira mais apropriada de estruturar aplicativos monolíticos não triviais.

Para aplicativos monolíticos, o núcleo do aplicativo, a infraestrutura e os projetos de interface do usuário são executados como um único aplicativo. A arquitetura do aplicativo em runtime pode ser semelhante à Figura 5-12.

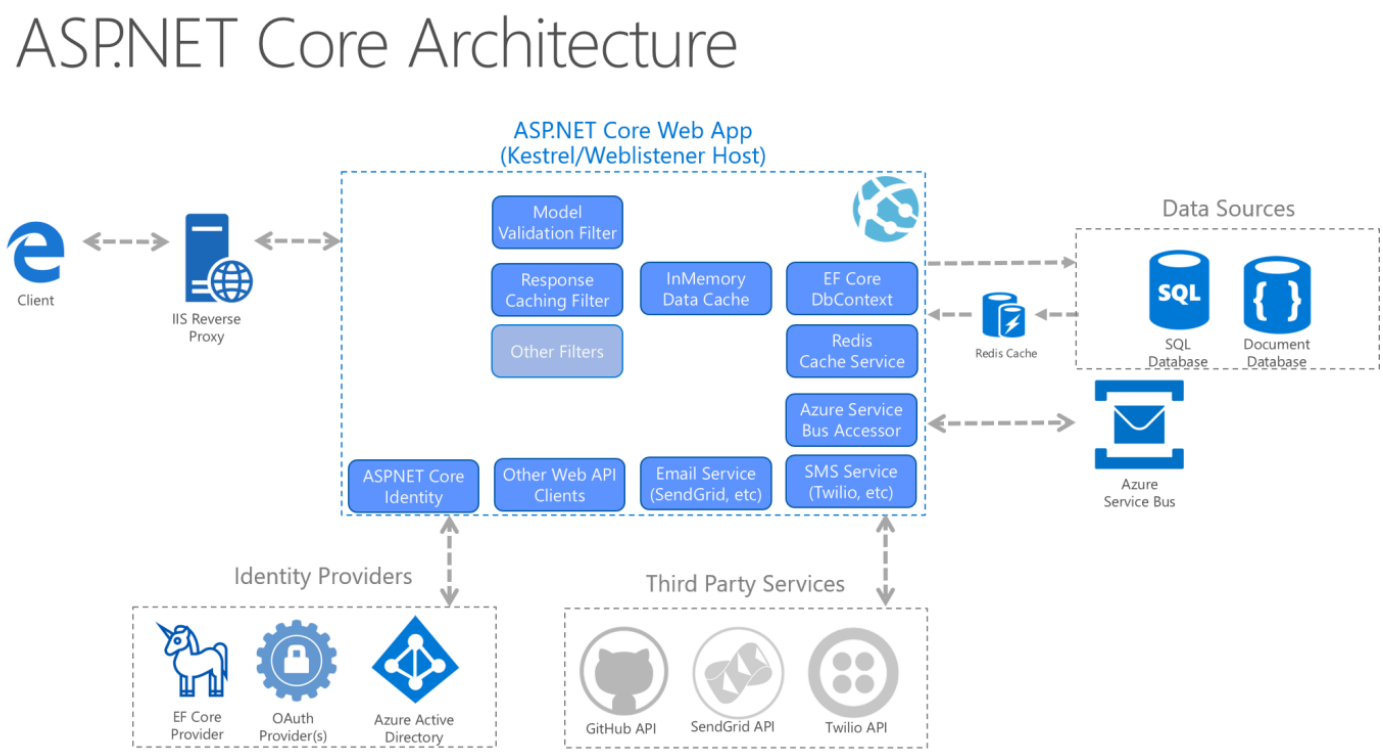


Figura 5-12. Arquitetura em runtime de um aplicativo ASP.NET Core de exemplo.

Organizando o código na arquitetura limpa

Em uma solução de Arquitetura Limpa, cada projeto tem responsabilidades bem-definidas. Assim, determinados tipos pertencem a cada projeto e, com frequência, haverá pastas correspondentes a esses tipos no projeto apropriado.

Núcleo do aplicativo

O Núcleo do Aplicativo contém o modelo de negócios, que inclui entidades, serviços e interfaces. Essas interfaces incluem abstrações para operações que serão executadas usando a infraestrutura, como acesso a dados, acesso ao sistema de arquivos, chamadas de rede, etc. Às vezes, serviços ou interfaces definidos nessa camada precisarão trabalhar com tipos que não sejam de entidade que não tenham dependências na interface do usuário ou na infraestrutura. Eles podem ser definidos como DTOs (Objetos de Transferência de Dados) simples.

Tipos de Núcleo do Aplicativo

- Entidades (classes de modelo de negócios que são persistidas)
- Agregações (grupos de entidades)
- Interfaces
- Serviços de Domínio
- Especificações
- Exceções personalizadas e cláusulas de proteção
- Manipuladores e eventos de domínio

Infraestrutura

O projeto de Infraestrutura normalmente incluirá implementações de acesso a dados. Em um aplicativo Web ASP.NET Core típico, essas implementações incluem o DbContext do EF (Entity Framework), todos os objetos `Migration` do EF Core que foram definidos e as classes de implementação de acesso a dados. A maneira mais comum de abstrair o código de implementação de acesso a dados é pelo uso do padrão de design de Repositório.

Além das implementações de acesso a dados, o projeto de Infraestrutura deve conter implementações de serviços que devem interagir com os interesses de infraestrutura. Esses serviços devem implementar as interfaces definidas no Núcleo do Aplicativo e, portanto, a Infraestrutura deve ter uma referência ao projeto de Núcleo do Aplicativo.

Tipos de infraestrutura

- Tipos do EF Core (`DbContext`, `Migration`)
- Tipos de implementação de acesso a dados (Repositórios)
- Serviços específicos de infraestrutura (por exemplo, `FileLogger` ou `SmtpNotifier`)

Camada de interface do usuário

A camada de interface do usuário em um aplicativo ASP.NET Core MVC é o ponto de entrada para o aplicativo. Esse projeto deve referenciar o projeto de Núcleo do Aplicativo e seus tipos devem interagir com a infraestrutura estritamente por meio das interfaces definidas no Núcleo do Aplicativo. Não deve ser permitida nenhuma criação de instância direta de tipos da camada de Infraestrutura nem nenhuma chamada estática a esses tipos na camada de interface do usuário.

Tipos de camada de interface do usuário

- Controladores
- Filtros personalizados
- Middleware personalizado
- Exibições
- ViewModels
- Inicialização

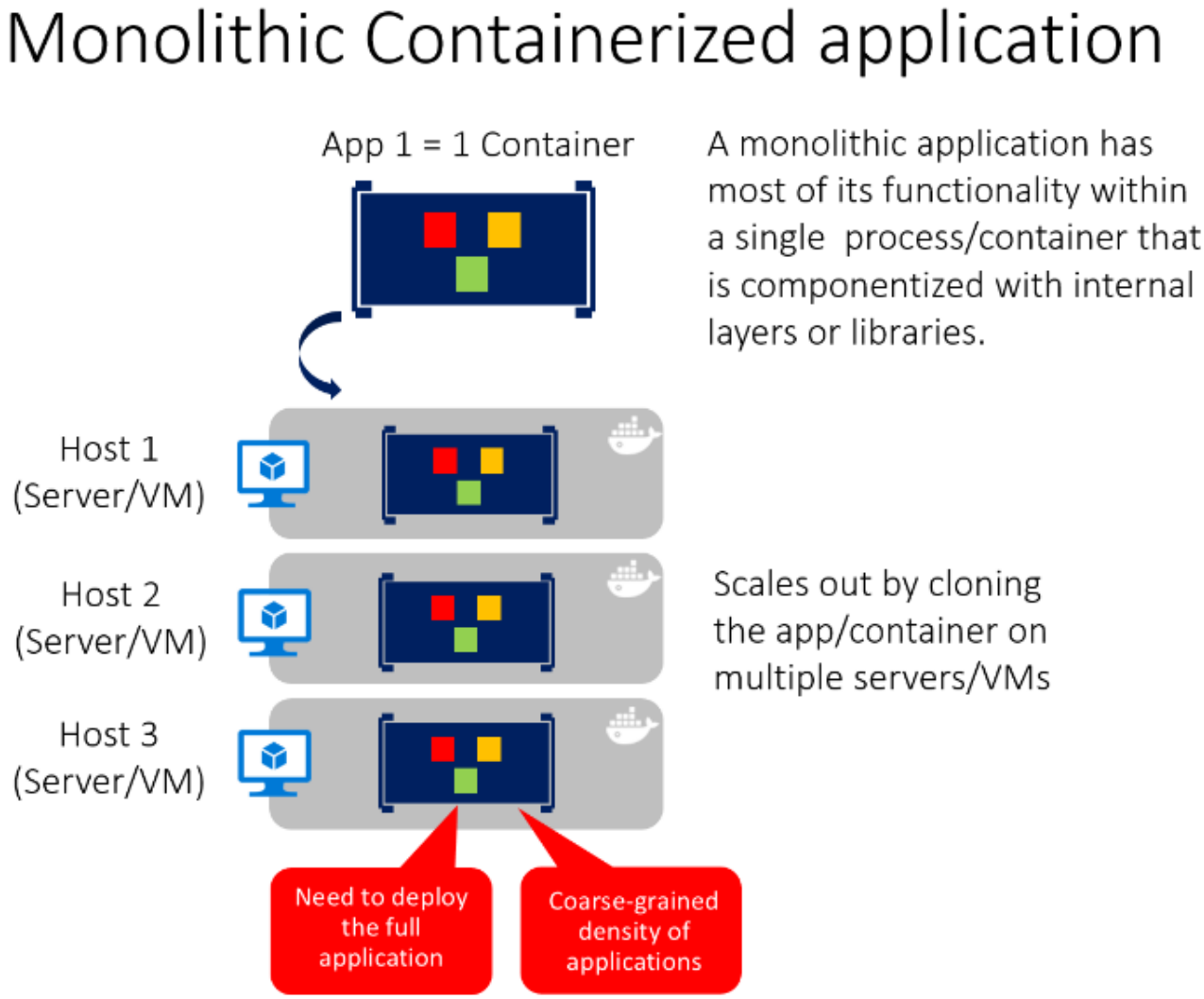
O `Startup` arquivo de classe ou *programa. cs* é responsável por configurar o aplicativo e para conectar tipos de implementação a interfaces. O local onde essa lógica é executada é conhecido como a raiz da *composição* do aplicativo e é o que permite que a injeção de dependência funcione corretamente no tempo de execução.

> **Observação**
>
> Para conectar a injeção de dependência durante a inicialização do aplicativo, o projeto da camada de interface do usuário pode precisar fazer referência ao projeto de infraestrutura. Essa dependência pode ser eliminada, mais facilmente usando um contêiner de DI personalizado que tem suporte interno para carregar tipos de assemblies. Para os fins deste exemplo, a abordagem mais simples é permitir que o projeto de interface do usuário referencie o projeto de infraestrutura (mas os desenvolvedores devem limitar as referências reais a tipos no projeto de infraestrutura para a raiz de composição do aplicativo).

Contêineres e aplicativos monolíticos

Você pode criar um Aplicativo ou Serviço Web baseado em uma implantação única e monolítica e implantá-lo como um contêiner. Dentro do aplicativo, ele pode não ser monolítico, mas organizado em várias bibliotecas, componentes ou camadas. Externamente, é um único contêiner com um único processo, um único aplicativo Web ou um único serviço.

Para gerenciar esse modelo, implante um contêiner único para representar o aplicativo. Para dimensionar, basta adicionar mais cópias com um balanceador de carga na frente. A simplicidade está em gerenciar um a implantação única em um contêiner ou VM único.



É possível incluir vários componentes/bibliotecas ou camadas internas em cada contêiner, conforme ilustrado na Figura 5-13. Mas, seguindo o princípio de contêiner de *"um contêiner executa uma ação e faz isso em um processo*", o padrão monolítico pode gerar um conflito.

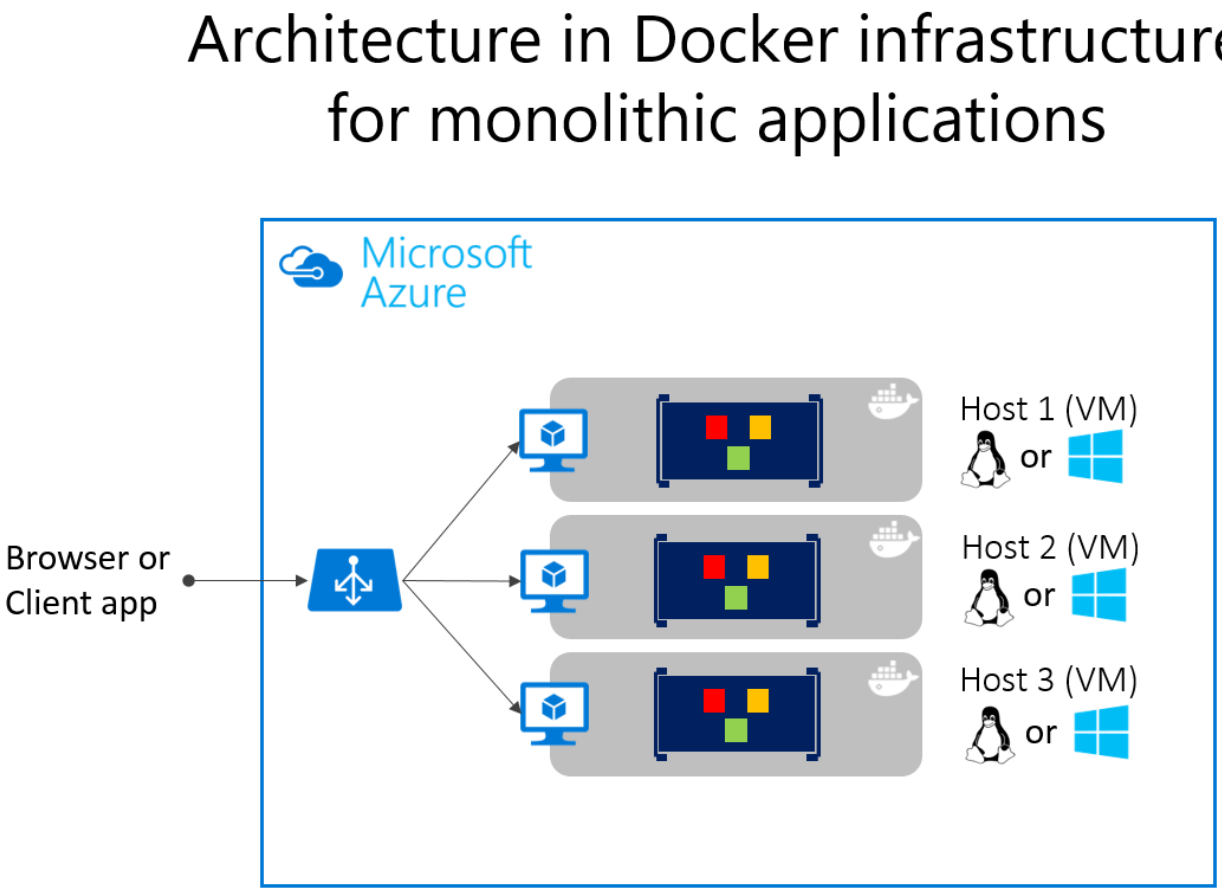
O ponto negativo dessa abordagem ficará evidente se ou quando o aplicativo crescer e for necessário dimensioná-lo. Não haverá problema se o aplicativo inteiro for dimensionado. Entretanto, na maioria dos casos, apenas algumas partes do aplicativo são os pontos de redução que exigem dimensionamento, enquanto outros componentes são menos utilizados.

Usando o exemplo de comércio eletrônico típico, o que você provavelmente precisa dimensionar é o componente de informações do produto. Uma quantidade muito maior de clientes procura produtos em vez de comprá-los. Mais clientes usam o carrinho em vez do pipeline de pagamento. Menos clientes fazem comentários ou exibem o histórico de compras. E você provavelmente tem apenas um grupo de funcionários, em uma única região, que precisa gerenciar o conteúdo e as

campanhas de marketing. Ao dimensionar o design monolítico, todo o código é implantado várias vezes.

Além do problema de “ter que dimensionar tudo”, a alteração de um único componente exige um novo teste completo de todo o aplicativo e uma reimplantação completa de todas as instâncias.

A abordagem monolítica é comum e muitas organizações estão desenvolvendo aplicativos com essa abordagem de arquitetura. Muitas estão tendo resultados bons o suficiente, enquanto outras estão atingindo os limites. Muitas criaram seus aplicativos nesse modelo porque as ferramentas e a infraestrutura eram muito difíceis para criar SOAs (arquiteturas orientadas a serviços) e elas não viam a necessidade até que o aplicativo crescesse. Se você achar que está chegando ao limite da abordagem monolítica, a divisão do aplicativo para permitir que ele aproveite melhor os contêineres e os microsserviços poderá ser a próxima etapa lógica.



A implantação de aplicativos monolíticos no Microsoft Azure pode ser feita por meio de VMs dedicadas para cada instância. Usando Conjuntos de Dimensionamento de Máquinas Virtuais do Azure, você pode dimensionar VMs facilmente. Os Serviços de Aplicativos do Azure podem executar aplicativos monolíticos e dimensionar instâncias com facilidade, sem a necessidade de gerenciar as VMs. Os Serviços de Aplicativos do Azure também podem executar instâncias únicas de contêineres do Docker, simplificando a implantação. Usando o Docker, você pode implantar uma única VM como um host do Docker e executar várias instâncias. Usando o balanceador do Azure, conforme mostrado na Figura 5-14, você pode gerenciar o dimensionamento.

A implantação em vários hosts pode ser gerenciada com técnicas de implantação tradicionais. Os hosts do Docker podem ser gerenciados manualmente com comandos como **docker run** executados manualmente ou por meio da automação, como os pipelines de CD (Entrega Contínua).

Aplicativo monolítico implantado como um contêiner

Há benefícios no uso de contêineres para gerenciar implantações de aplicativos monolíticos. O dimensionamento das instâncias de contêineres é muito mais rápido e fácil do que a implantação de VMs adicionais. Mesmo ao usar conjuntos de dimensionamento de máquinas virtuais para dimensionar as VMs, elas levam tempo para serem criadas. Quando implantada como instâncias de aplicativo, a configuração do aplicativo é gerenciada como parte da VM.

Implantar atualizações como imagens do Docker é muito mais rápido e eficiente em termos de rede. As Imagens do Docker costumam ser iniciadas em segundos, o que agiliza as distribuições. Desmontar uma instância do Docker é tão fácil quanto emitir um comando `docker stop` e geralmente leva menos de um segundo.

Como os contêineres são, de forma inerente, imutáveis por design, você nunca precisa se preocupar com VMs corrompidas, enquanto os scripts de atualização podem se esquecer de considerar uma configuração específica ou um arquivo deixado no disco.

Você pode usar contêineres do Docker para uma implantação monolítica de aplicativos Web mais simples. Essa abordagem melhora a integração contínua e os pipelines de implantação contínua e ajuda a atingir o sucesso da implantação para a produção. Não mais "ele funciona em meu computador, por que ele não funciona em produção?"

Uma arquitetura baseada em microsserviços tem muitos benefícios, mas esses benefícios são fornecidos ao custo do aumento da complexidade. Em alguns casos, os custos superam os benefícios, portanto, um aplicativo de implantação monolítica em execução em um único contêiner ou em alguns contêineres é uma opção melhor.

Um aplicativo monolítico não pode ser facilmente decomposto em microsserviços bem separados. Os microsserviços devem funcionar independentemente uns dos outros para fornecer um aplicativo mais resiliente. Se você não puder entregar fatias independentes de recurso do aplicativo, separá-las apenas aumentará a complexidade.

Talvez o aplicativo ainda não precise dimensionar recursos de modo independente. Muitos aplicativos, quando precisam ser dimensionados para mais de uma única instância, podem passar pelo processo relativamente simples de clonagem da instância inteira. O trabalho adicional para separar o aplicativo em serviços discretos fornece um benefício mínimo ao dimensionar as instâncias completas do aplicativo é simples e econômico.

No início do desenvolvimento de um aplicativo, você pode não ter uma ideia clara de onde estão os limites funcionais naturais. Mesmo depois de desenvolve rum produto mínimo viável, a separação natural pode ainda não ter surgido. Algumas dessas condições podem ser temporárias. Você pode começar criando um aplicativo monolítico e posteriormente separar alguns recursos para serem desenvolvidos e implantados como microsserviços. Outras condições podem ser essenciais para o espaço problemático do aplicativo, o que significa que o aplicativo pode nunca ser dividido em vários microserviços.

A separação de um aplicativo em vários processos distintos também apresenta sobrecarga. É mais complexo separar recursos em processos diferentes. Os protocolos de comunicação tornam-se mais complexos. Em vez de chamadas de método, você deve usar a comunicação assíncrona entre serviços. Ao passar para uma arquitetura de microsserviços, você precisa adicionar muitos dos blocos de construção implementados na versão de microsserviços do aplicativo eShopOnContainers: manipulação de barramento de eventos, resiliência e novas tentativas de mensagem, consistência eventual e muito mais.

O aplicativo de referência eShopOnWeb muito mais simples dá suporte ao uso de único contêiner monolítico. O aplicativo inclui um aplicativo Web com exibições de MVC, APIs Web e Razor Pages tradicionais. Opcionalmente, você pode executar o componente de administração baseado em mais alto do aplicativo, o que requer um projeto de API separado para ser executado também.

O aplicativo pode ser iniciado a partir da raiz da solução usando os `docker-compose build` `docker-compose up` comandos e. Esse comando configura um contêiner para a instância

da Web usando o `Dockerfile` encontrado na raiz do projeto Web e executa o contêiner em uma porta especificada. Você pode baixar o código-fonte desse aplicativo do GitHub e executá-lo localmente. Até mesmo esse aplicativo monolítico beneficia-se de ser implantado em um ambiente do contêiner.

Por exemplo, a implantação em contêineres significa que cada instância do aplicativo é executada no mesmo ambiente. Essa abordagem inclui o ambiente de desenvolvimento no qual ocorrem testes e desenvolvimento antecipados. A equipe de desenvolvimento pode executar o aplicativo em um ambiente em contêineres que corresponda ao ambiente de produção.

Além disso, os aplicativos em contêineres são expandidos a um custo mais baixo. O uso de um ambiente de contêiner permite um compartilhamento maior de recursos do que os ambientes de VM tradicionais.

Por fim, colocar o aplicativo em contêineres força uma separação entre a lógica de negócios e o servidor de armazenamento. À medida que o aplicativo for escalado horizontalmente, os vários contêineres dependerão de uma única mídia de armazenamento físico. A mídia de armazenamento seria normalmente um servidor de alta disponibilidade executando um banco de dados do SQL Server.

Suporte ao Docker

O `eShopOnWeb` projeto é executado no .net. Portanto, ele pode ser executado em contêineres baseados em Linux ou em Windows. Observe que, para a implantação do Docker, você deseja usar o mesmo tipo de host para o SQL Server. Os contêineres baseados em Linux permitem o uso de um espaço menor e são preferenciais.

Você pode usar o Visual Studio 2017 ou posterior para adicionar o suporte ao Docker a um aplicativo existente clicando com o botão direito do mouse em um projeto no **Gerenciador de Soluções** e escolhendo **Adicionar** > **Suporte ao Docker**. Esta etapa adiciona os arquivos necessários e modifica o projeto para usá-los. O exemplo `eShopOnWeb` atual já têm esses arquivos.

O arquivo `docker-compose.yml` no nível da solução contém informações sobre quais imagens devem ser criadas e quais contêineres devem ser iniciados. O arquivo permite que você use o comando `docker-compose` para iniciar vários aplicativos ao mesmo tempo. Nesse caso, ele está iniciando apenas o projeto Web. Você também pode usá-lo para configurar dependências, como um contêiner de banco de dados separado.

yml Copiar

```yml
version: '3'

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

networks:
```

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

O arquivo `docker-compose.yml` referencia o `Dockerfile` no projeto `Web`. O `Dockerfile` é usado para especificar qual contêiner base será usado e como o aplicativo será configurado nele.
O `Dockerfile` do `Web`:

Dockerfile

Copiar

```
FROM mcr.microsoft.com/dotnet/sdk:6.0 AS build
WORKDIR /app

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

RUN dotnet publish -c Release -o out

FROM mcr.microsoft.com/dotnet/aspnet:6.0 AS runtime
WORKDIR /app
COPY --from=build /app/src/Web/out ./

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

Solução de problemas do Docker

Depois que você iniciar a execução do aplicativo em contêineres, ele continuará sendo executado até que você o pare. É possível ver quais contêineres estão em execução com o comando `docker ps`. Você pode parar um contêiner em execução usando o comando `docker stop` e especificando a ID do contêiner.

Observe que a execução de contêineres do Docker pode ser associada a portas, que caso contrário, você tentaria usar em seu ambiente de desenvolvimento. Se você tentar executar ou depurar um aplicativo usando a mesma porta, como um contêiner do Docker em execução, ocorrerá um erro informando que o servidor não pôde ser associado a essa porta. Mais uma vez, parar o contêiner deverá resolver o problema.

Para adicionar o suporte ao Docker ao aplicativo usando o Visual Studio, verifique se o Docker Desktop está em execução. Quando você iniciar o assistente, ele não será executado corretamente se o Docker Desktop não estiver em execução. Além disso, o assistente examinará sua escolha de contêiner atual para adicionar o suporte ao Docker correto. se você quiser adicionar, suporte para contêineres de Windows, será necessário executar o assistente enquanto houver área de trabalho do docker em execução com contêineres de Windows configurados. Se você quiser adicionar o, suporte para contêineres do Linux, execute o assistente enquanto você tem o Docker em execução com contêineres do Linux configurados.

Referências – arquiteturas comuns da Web

- **A Arquitetura Limpa**
  https://8thlight.com/blog/uncle-bob/2012/08/13/the-clean-architecture.html

- **A Arquitetura Cebola**
  https://jeffreypalermo.com/blog/the-onion-architecture-part-1/
- **O padrão de repositório**
  https://deviq.com/repository-pattern/
- **Limpar modelo de solução de arquitetura**
  https://github.com/ardalis/cleanarchitecture
- **Livro eletrônico Architecting Microservices**
  https://aka.ms/MicroservicesEbook
- **DDD (design controlado por domínio)**
  https://docs.microsoft.com/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/