Principais Arquiteturas de Software do mercado

Bootcamp Arquiteto de Software

Albert Tanure

2020

Principais Arquiteturas de Software do Mercado

Bootcamp Arquiteto de Software

Albert Tanure

Sumário

Capítulo 1. Arquitetura de Sistemas Web 5
Características de aplicações Web modernas 7
Princípios arquiteturais 9
Separação de responsabilidades 10
Encapsulamento 13
Inversão de dependência 13
Dependências explícitas 15
Don’t repeat yourself (DRY) 16
Responsabilidade única 16
Principais arquiteturas 17
Capítulo 2. API 24
REST 24
GRPC 30
GgraphQL 32
The Twelve-factor app 35
Capítulo 3. Arquitetura de Sistemas Mobile 37
Arquitetura nativa 37
Android 38
IOS 38
Arquitetura Cross-Platform 39
Arquitetura Híbrida 40
Capítulo 4. Arquitetura de Microsserviços 42
O que são Microsserviços? 42
Benefícios 44

Desafios 45
Boas práticas 47
API Gateway 48
Capítulo 5. Arquitetura Serverless 49
Full Backend Serverless 50
Aplicações Web 50
Backend para aplicações móveis 51
IOT 52
Considerações 53
Capítulo 6. Arquitetura Cloud Native 56
Documentação 60
Referências 62

Capítulo 1. Arquitetura de Sistemas Web

Definir soluções arquiteturais é muito mais do que escolher tecnologias e criar um repositório. O mercado tem exigido aplicações cada vez mais complexas que demandam um alto grau de processamento e trabalha com diversos conceitos, e todo este apanhado de necessidades faz com que o Arquiteto de Software se mantenha em estudo continuamente. Estudar padrões arquiteturais é parte do skill do arquiteto que necessita de boas referências para servirem de base na provisão de soluções aderentes aos contextos negociais.

Existem diversos modelos arquiteturais existentes, no entanto, ter uma boa base de fundamentos será o seu maior trunfo. Mas além disso, como é arquitetar uma solução? Quais são os primeiros passos? Quais são as habilidades necessárias?

- Antes da tecnologia, nós atuamos com pessoas. Neste contexto, o Arquiteto deve ter uma visão mais generalista sobre o seu ambiente, apesar de ser especialista em algumas outras áreas ou tecnologias. Contudo, antes de inicar o desenho de uma solução, utilize algumas ferramentas poderosas ao seu favor:
  - Pessoas:
    - Este é um dos pontos mais importantes. Todos os projetos tem, com certeza, a participação de pessoas. Gerentes, desenvolvedores, Scrum Masters, Product Owners, equipe de QA, parceiros, clientes, entre outros. Você vai interagir com todos!
    - Esteja apto para ouvir. Um bom arquito ouve mais, serve as pessoas.
    - Obtenha feedbacks constantes sobre as soluções propostas. Tenha empatia e busque entender se os artefatos que foram gerados, sejam qual forem, atendem às necessidades de quem os utiliza.

- Promova um ambiente colaborativo e evolutivo. Capacite as pessoas do seu time e dê oportunidades.
- Um arquiteto não é um Deus!

– Decisões arquiteturais:

- Não se baseie na moda. É muito fácil ser atraído para novas e otimas tecnologias que surgem todos os dias. Não dá para usar tudo a todo momento, e nem mesmo estar concentrado para aprender todas as coisas. Seja prudente!
- Os padrões arquiteturais são uma base. Devem ser estudados, criticados e adaptados às necessidades da sua solução. Eles são um direcionador, não um script de como você deve fazer tudo.
- Não existe arquitetura padrão. Cada caso é um caso.
- Paute suas decisões em dados.
- As suas decisões hoje podem afetar suas soluções no futuro.

– Trabalhe de forma evolutiva:

- É muito comum, até pela experiência e vivência em diversos contextos, que ao definir uma solução, optamos por utilizar as melhores tecnologias e criar diversos mecanismos complexos, ou seja, criamos um “canhão para matar uma formiga”. Calma!
- Direcione suas estratégias com uma visão que entregue valor e atenda às necessidades negociais. Não quer dizer que não deve haver qualidade, mas faça por etapas.
- Foque na entrega.
- Adicione recursos à solução quando eles realmente são necessários.

- Trabalhe de forma evolutiva.
- Divida para conquiestar. A complexidade é feita de muitas simplicidades.

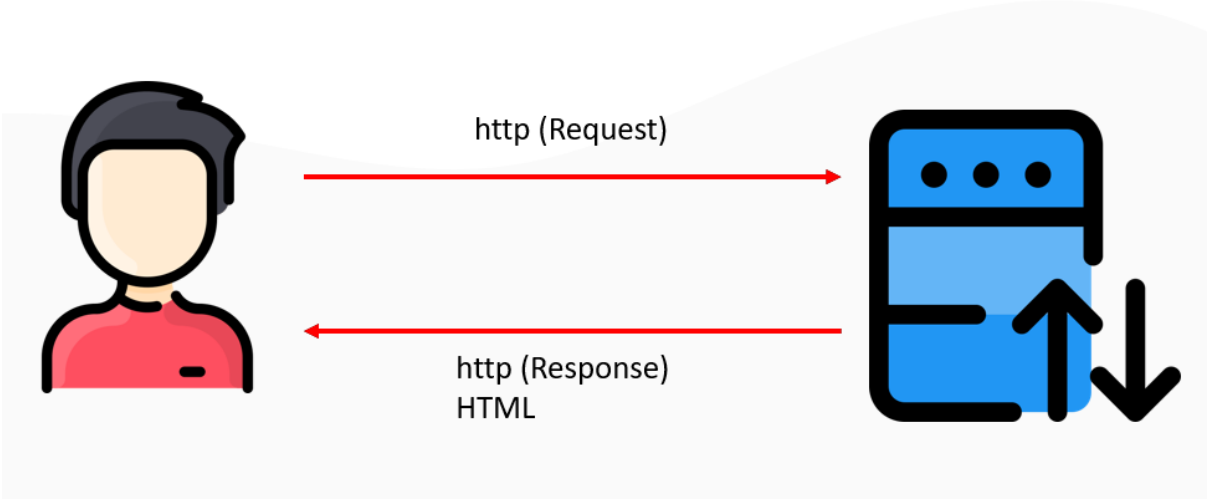
– Papel e lápis:

- Estes são as melhores ferramentas que possuímos. Descarregue suas ideias, desenhe, rabisque, apague e começe novamente. Não mantenha a complexidade na cabeça.

Características de aplicações Web modernas

Os sistemas Web são presentes no dia a dia das pessoas, seja um website ou algum sistema com um cadastro. A maioria destas aplicações estão concentradas, muitas vezes, em um contexto único e publicadas em um web server. Desta forma, o usuário de uma aplicação atua com uma requisição (request) para estes servidores que retornam uma resposta (response).

Figura 1 – Comunicação Cliente-Servidor.



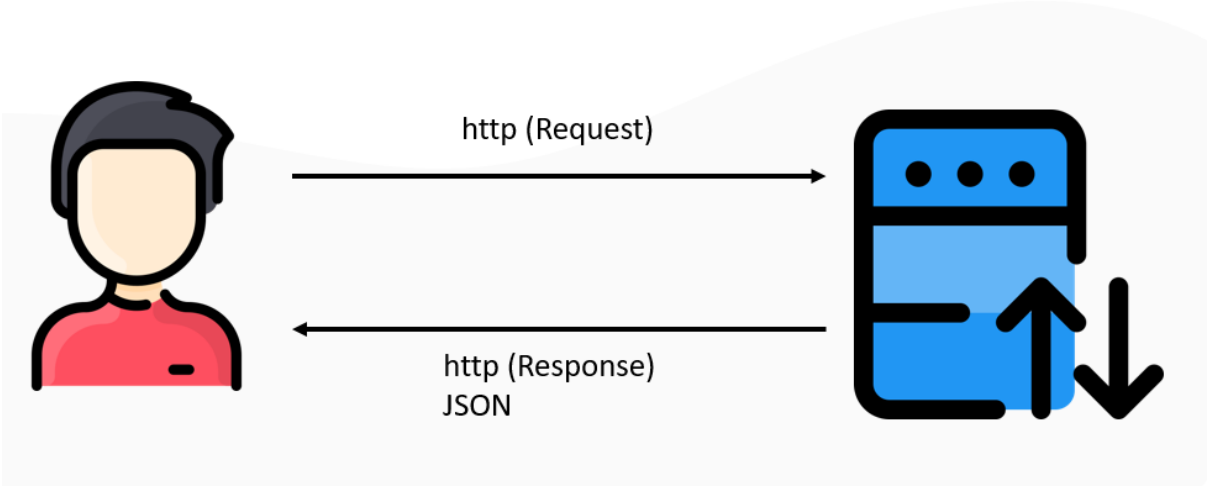
Como apresentado na Figura 1, todas as respostas das requisições, neste contexto, retornam um documento HTML válido que será exibido pelo navegador.

É um modelo arquitetural muito bom e com características que várias tecnologias utilizam, como por exemplo, o PHP, Classic Asp, Asp.Net MVC, entre muitas outras.

Contudo, trafegar um documento HTML pode ser custoso e, além disso, a utilização deste modelo arquitetural, apesar de ser muito poderoso e funcional, pode criar algumas dificuldades principalmente para o lado dos usuários, já que não oferecem tantos recursos de usabilidade. E mesmo com a possiblidade de se trabalhar parte com as requisições processadas, parte no servidor e parte no cliente através de JavaScript, não é uma boa abordagem.

Neste caso, existe um modelo arquitetural denominado Single Page Application (SAP). Esta abordagem cria um novo conceito de desenvolvimento de aplicações Web, já que se baseiam em JavaScript, ou seja, sua execução é do lado do cliente. Observe a Figura 2.

Figura 2 – Comunicação SPA com o Servidor.



Aparentemente a requisição é muito parecida com a figura anterior, se não fosse o fato de que a resposta, exibida na Figura 3, está em um formato denominado JSON, que é, basicamente, uma notação do tipo texto, legível aos humanos e bastante leve. O que é melhor neste contexto é que não há a necessidade de tráfego de todo documento HTML, mas apenas os dados de um contexto. Além de criar uma aplicação com uma ótima usabilidade, aproveitando os melhores recursos disponíveis

nas APIS dos navegadores, o SPA facilita a adoção de modelos arquiteturais mais robustos, aplicando as melhores práticas e princípios.

Um SAP pode ser implementado utilizando HTML, CSS e JavaScript. No entanto, existem vários frameworks disponíveis no mercado possuindo vários mecanismos, componentes e implementações de padrões que são extensíveis, o que trás maior ganho e agilidade no desenvolvimento das soluções.

Alguns dos principais frameworks existentes são o Angular, React e o VueJs. Claro que existem infinitas soluções, mas estes três são as principais tecnologias utilizadas por grandes empresas. Cada um possui sua organização, ferramentas e abordagem de desenvolvimento, mas são baseadas em HTML, CSS e JavaScript. No caso do Angular, há a utilização do TypeScript. Uma linguagem de scripts, tipada, criada pela Microsoft. O TypeScript fornece um modelo de desenvolvimento bem próximo do que temos no “backend”, já que utilizam conceitos como tipos, interfaces, classes, entre outros. Esta tecnologia não é uma exclusividade do Angular, podendo ser adotada em outras soluções.

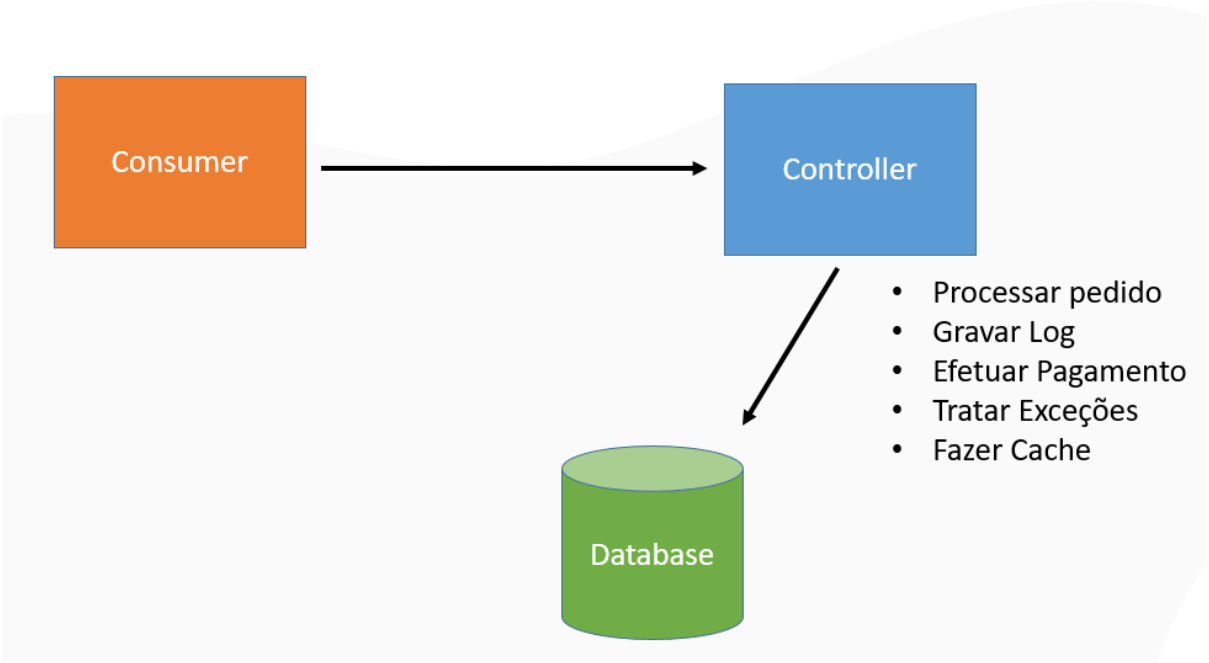
Outras técnicas muito interessantes se tornaram possíveis ao adotar uma solução SPA, são a utilização de teste de unidade para componentes, separação em camadas, utilização de containers, Continuous Integration e Continuous Deployment, entre outros princípios arquiteturais e técnicas.

Princípios arquiteturais

O desafio de se projetar software não está somente na definição dos seus mecanismos, mas também na criação de soluções extensíveis e de fácil manutenção. Para atingir estes objetivos, a orientação baseada em fundamentos e princípios é fundamental.

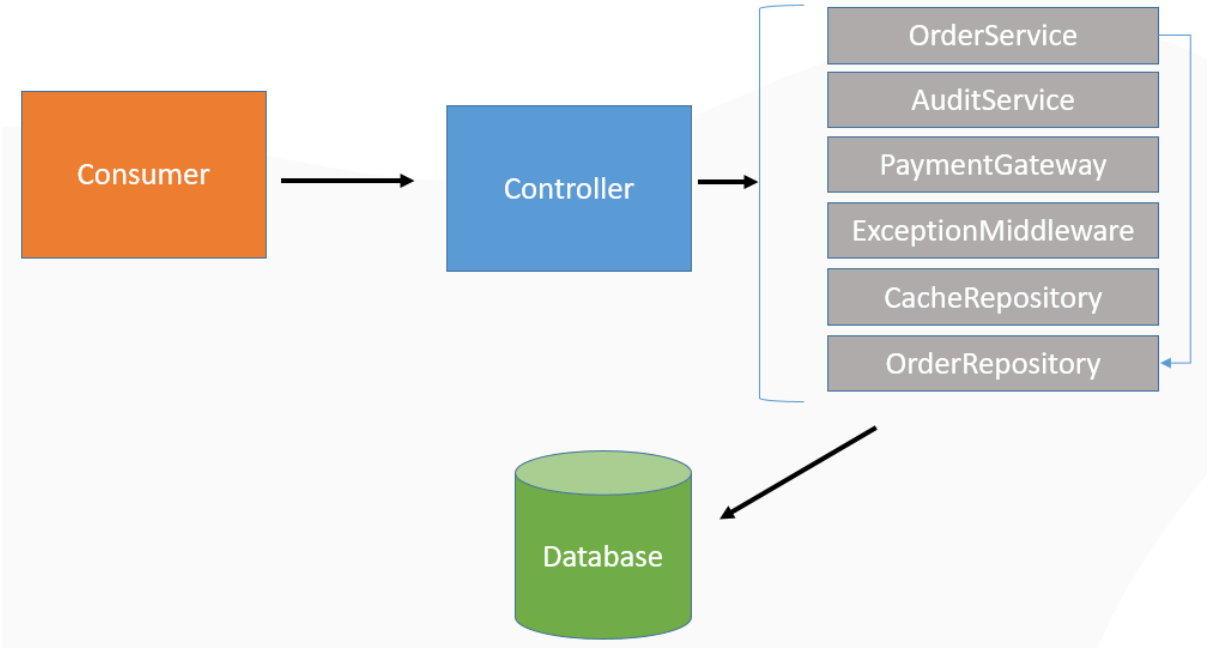
A Figura 3 representa um contexto de uma classe que possui mais de uma responsabilidade.

Figura 3 – Objeto com muitas responsabilidades.



A classe *Controller* possui a responsabilidade de orquestrar todos os fluxos relacionados a um pedido, concentrando toda a responsabilidade sobre esta funcionalidade. Agora observe a Figura 4:

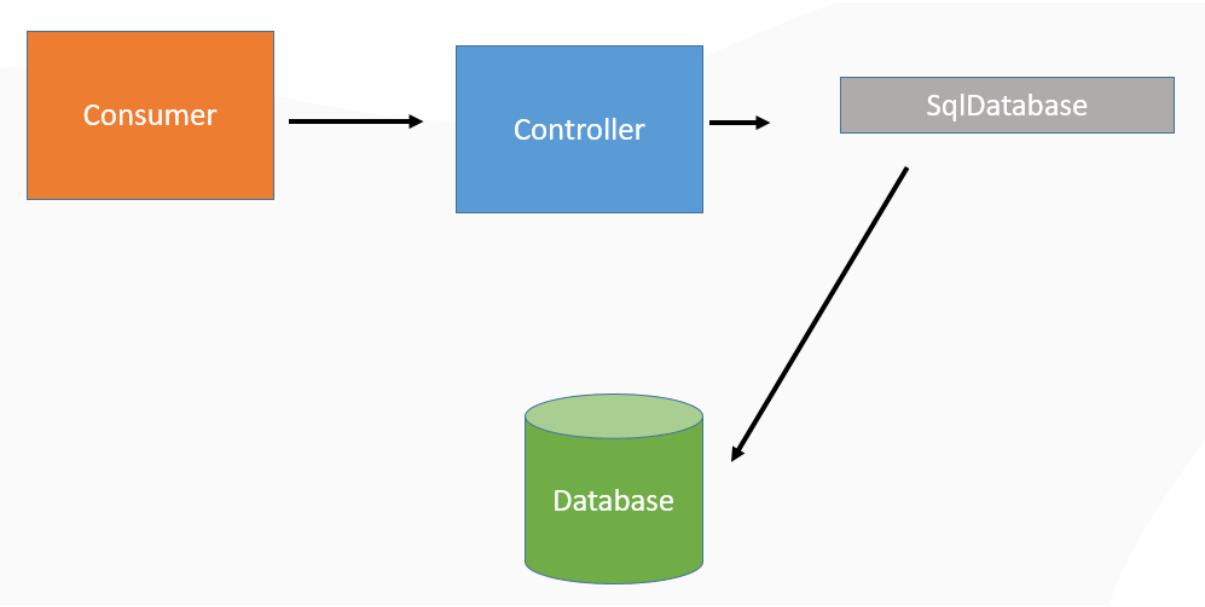
Figura 4 – Separação de responsabilidades.



A classe *Controller* apenas delega as responsabilidades para outras casses, separando corretamente os contextos ou responsabilidades.

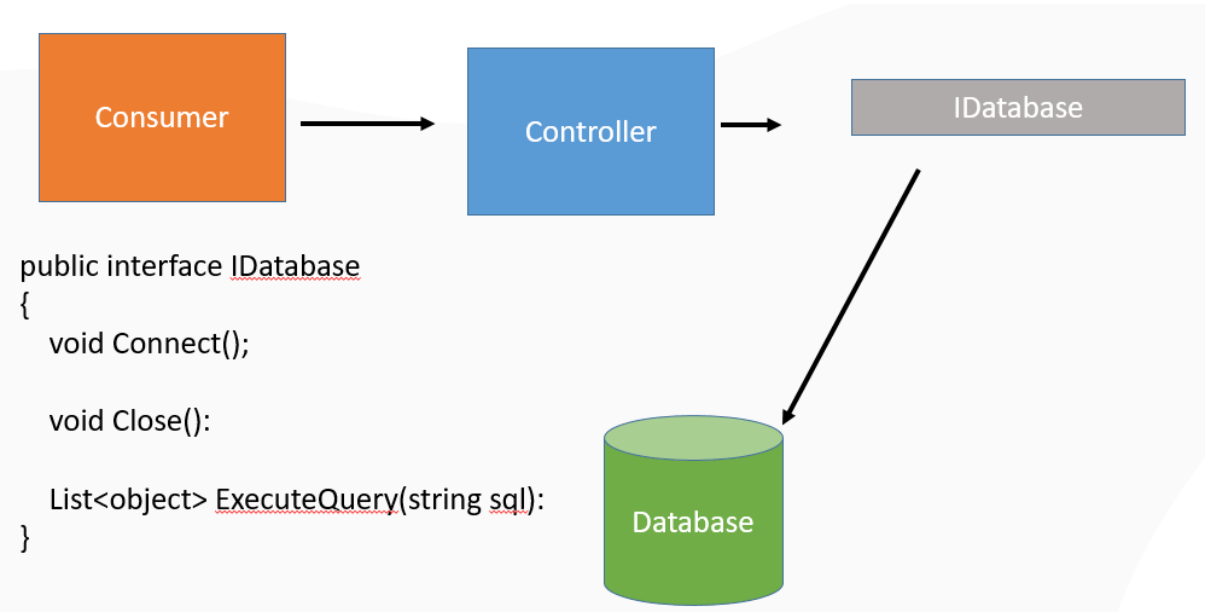
Em termos de arquitetura, os aplicativos podem ser logicamente criados para seguir esse princípio, separando o comportamento principal de negócios da infraestrutura e da lógica da interface do usuário. O ideal é que a lógica e as regras de negócio residam em um projeto separado, que não deve depender de outros projetos no aplicativo. Essa separação ajuda a garantir que o modelo de negócios seja fácil de testar e possa evoluir sem estar rigidamente acoplado a detalhes de implementação de baixo nível. Para ilustrar este exemplo, observe a seguinte figura:

Figura 5 – Dependência de implementação,



A Figura 5 apresenta um contexto onde a classe *Controller* depende diretamente da implementação da classe *SqlDatabase*, dificultando a manutenção, evolução e os testes na aplicação. Da mesma forma, este conceito deve ser aplicado no desenho de arquitetura de soluções, fazendo com que as dependências sejam por interfaces e não por implementações.

Figura 6 – Dependência por interfaces (contratos).



Encapsulamento

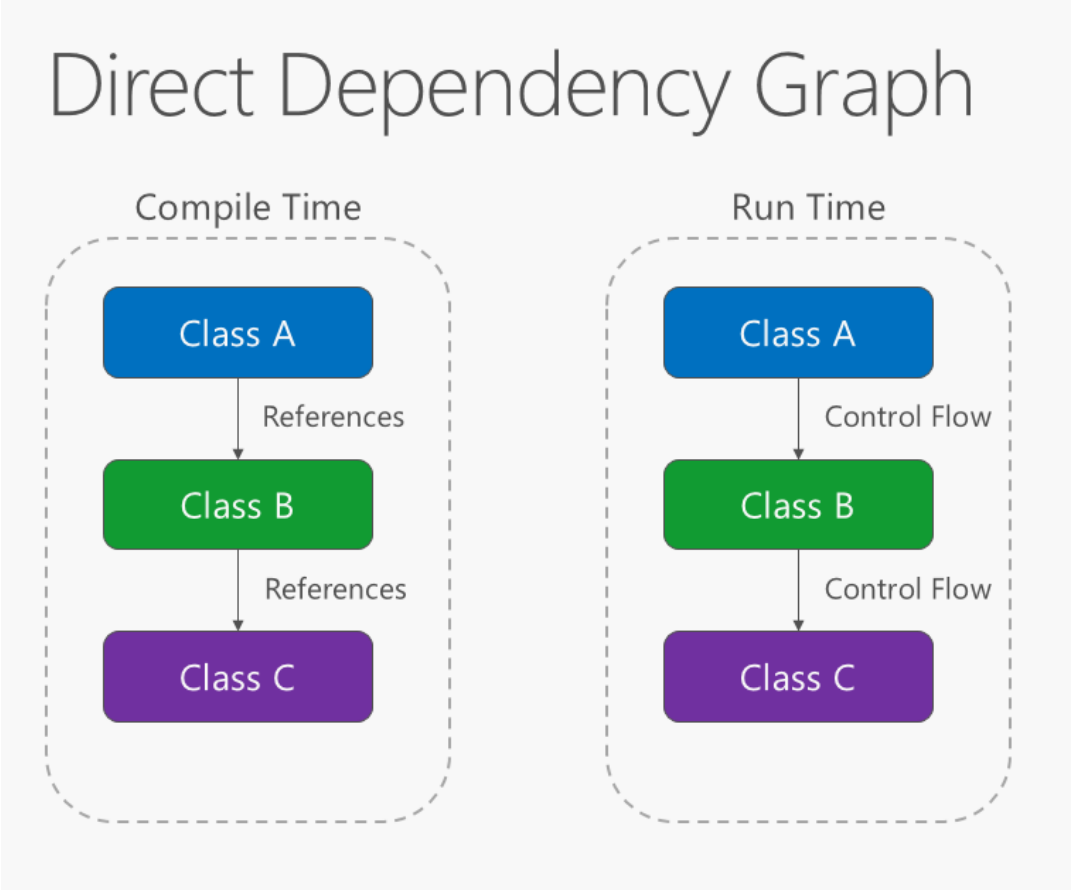
O princípio do **encapsulamento** tem como objetivo o isolamento de partes de uma aplicação. Basicamente, as camadas e seus componentes devem ser capaz de serem alterados sem depender dos seus colaboradores, desde que contratos externos não sejam violados. O uso adequado do encapsulamento ajuda a obter um acoplamento flexível e uma modularidade nos designs do aplicativo, pois os objetos e os pacotes podem ser substituídos por implementações alternativas, desde que a mesma interface seja mantida.

Nas classes, o encapsulamento é obtido por meio da limitação do acesso externo ao estado interno da classe. Se um ator externo desejar manipular o estado do objeto, ele deverá fazer isso por meio de uma função bem definida (ou um *setter* de propriedade), em vez de ter acesso direto ao estado particular do objeto. Da mesma forma, os componentes do aplicativo e os próprios aplicativos devem expor interfaces bem definidas para uso de seus colaboradores, em vez de permitir que seu estado seja modificado diretamente. Isso libera o design interno do aplicativo para evoluir ao longo do tempo, sem a preocupação de que fazendo isso dividirá os colaboradores, desde que os contratos públicos sejam mantidos.

Inversão de dependência

O princípio da inversão de dependência reforça o que foi falado sobre depende de interfaces. Dependa de abstrações. Observe a seguinte imagem:

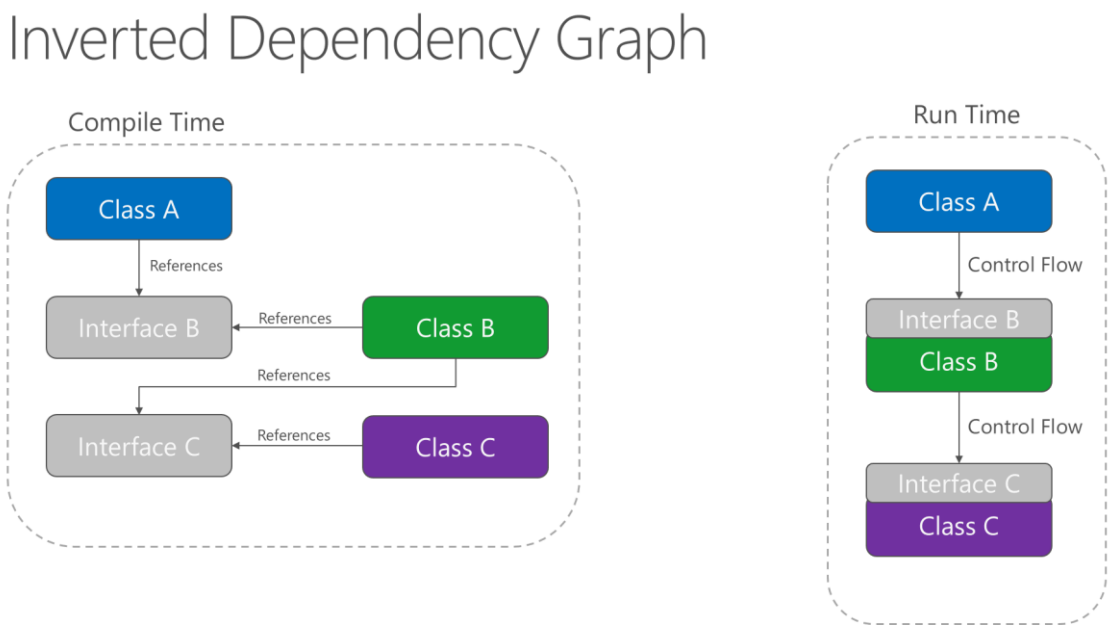
Figura 7 – Dependência de implementação.



Fonte: https://docs.microsoft.com.

A classe A depende de B que depende de C. Este comportamento se mantém tanto em tempo de compilação quanto em *run time*. Por outro lado, vejamos outro exemplo:

Figura 8 – Dependência de abstrações.



Fonte: https://docs.microsoft.com.

A dependência entre as classes A, B e C agora são através de implementações. Como resultado, os aplicativos resultantes são mais testáveis, modulares e de manutenção mais fácil. Esta prática, além disso, tornou possível a utilização do princípio de inversão de dependência.

Dependências explícitas

Métodos e classes devem exigir explicitamente suas dependências para funcionarem corretamente. Se você definir classes que podem ser construídas e chamadas, mas que só funcionarão corretamente se determinados componentes globais ou de infraestrutura estiverem em vigor, essas classes serão *desonestas* com seus consumidores. Isso pode gerar problemas em *runtime*.

Seguindo o princípio da dependência explícita, os métodos e as classes estão sendo coerentes com seus clientes sobre o que precisam para funcionar. Este princípio torna seu código mais autodocumentado e seus contratos de codificação mais amigáveis.

Don't repeat yourself (DRY)

É comum em que no desenvolvimento das aplicações existam códigos repetidos, utilizados em várias partes da solução. Contudo essa prática é uma fonte de erros frequente se tais débitos técnicos não forem pagos. Em algum momento, uma alteração nos requisitos exigirá a alteração desse comportamento. É provável que pelo menos uma instância do comportamento não seja atualizada, e o sistema se comportará de forma inconsistente.

Uma ótima abordagem para detectar tal inconformidade está relacionada ao uso de técnicas como o *Code Review*, que deve ser feito por todo o time.

Responsabilidade única

O princípio da responsabilidade única se aplica ao design orientado a objeto, mas também pode ser considerado um princípio de arquitetura semelhante à separação de responsabilidades. Ele informa que os objetos devem ter apenas uma responsabilidade e que devem ter apenas uma única razão para serem alterados. Especificamente, a única situação na qual o objeto deve ser alterado, é se a maneira na qual ele executa sua responsabilidade única precisa ser atualizada. Seguir esse princípio ajuda a produzir sistemas mais rígidos e modulares, já que muitos tipos de novos comportamentos podem ser implementados como novas classes, em vez de adicionar responsabilidade adicional às classes existentes. A adição de novas classes sempre é mais segura do que a alteração das classes existentes, pois nenhum código ainda depende das novas classes.

Em um aplicativo monolítico, podemos aplicar o princípio da responsabilidade única em um alto nível, às camadas do aplicativo. A responsabilidade de apresentação deve permanecer no projeto de interface do usuário, enquanto a responsabilidade de acesso a dados deve ser mantida em um projeto de infraestrutura. A lógica de negócios deve ser mantida no projeto de núcleo do aplicativo, no qual ela pode ser testada com facilidade e pode evoluir de maneira independente das outras responsabilidades.

Os modelos arquiteturais servem de **referência** para auxiliar na construção de arquiteturas robustas. A cada contexto teremos a necessidade de adaptar estes designers, utilizar suas melhores práticas e produzir novos estilos arquiteturais.

Cada proposta de design tem seus prós e contras. Não existe o melhor ou o certo ou o errado. Existe aquele que se adapta melhor a um determinado cenário.

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.

O que são layers?

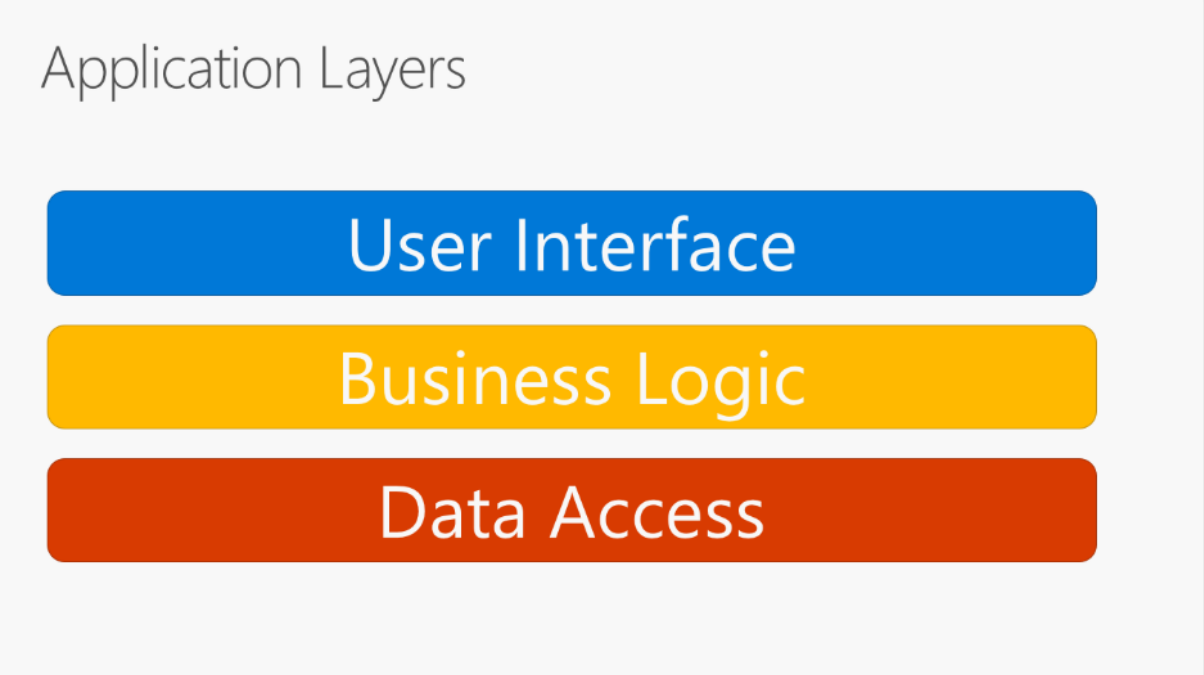
Conforme a complexidade dos aplicativos aumenta, uma maneira de gerenciar essa complexidade é dividir o aplicativo de acordo com suas responsabilidades ou interesses. Isso ajuda a manter uma base de código organizada para que os desenvolvedores possam encontrar facilmente onde determinadas funcionalidades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

Uma arquitetura em camadas impõe restrições sobre quais camadas podem se comunicar com outras camadas de acordo com o princípio do 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.

Imagine uma camada de acesso a dados que expõe abstrações para outras camadas. Havendo uma necessidade de alterar o modelo de persistência, as camadas superiores não sofrerão impactos, desde que as interfaces não sofram alterações.

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.

Figura 9 – Modelo de arquitetura em camadas.



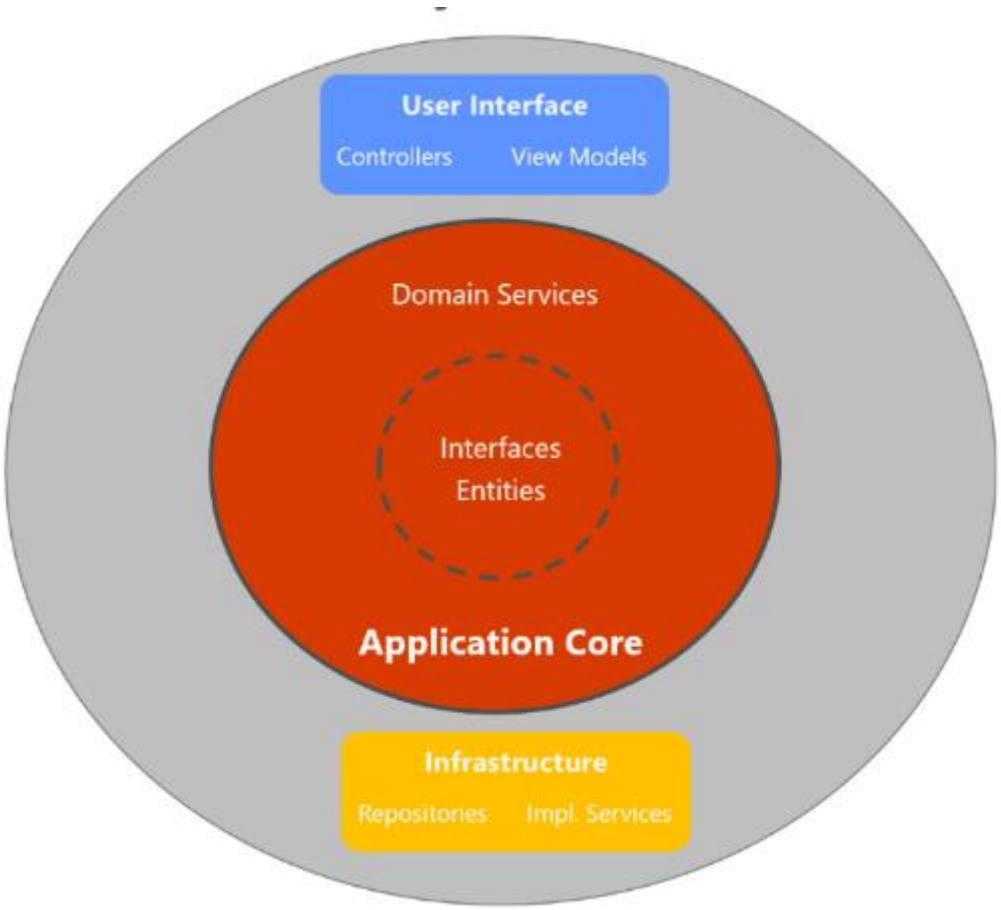
Fonte: https://docs.microsoft.com.

Clean Architecture (Onion Architecture):

Os aplicativos que seguem o Princípio da Inversão de Dependência, bem como os princípios de DDD (Domain Driven Design), 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 Ports e Adapters, mas é conhecida como Clean Architecture ou Onion Architecture.

Nesta arquitetura, toda a regra de negócio é localizada no centro do aplicativo. Sendo assim, a camada mais ao centro não conhece nenhuma camada superior. Isso é feito pela definição de abstrações, ou interfaces, no *core* da aplicação.

Figura 10 – Clean Architecture.

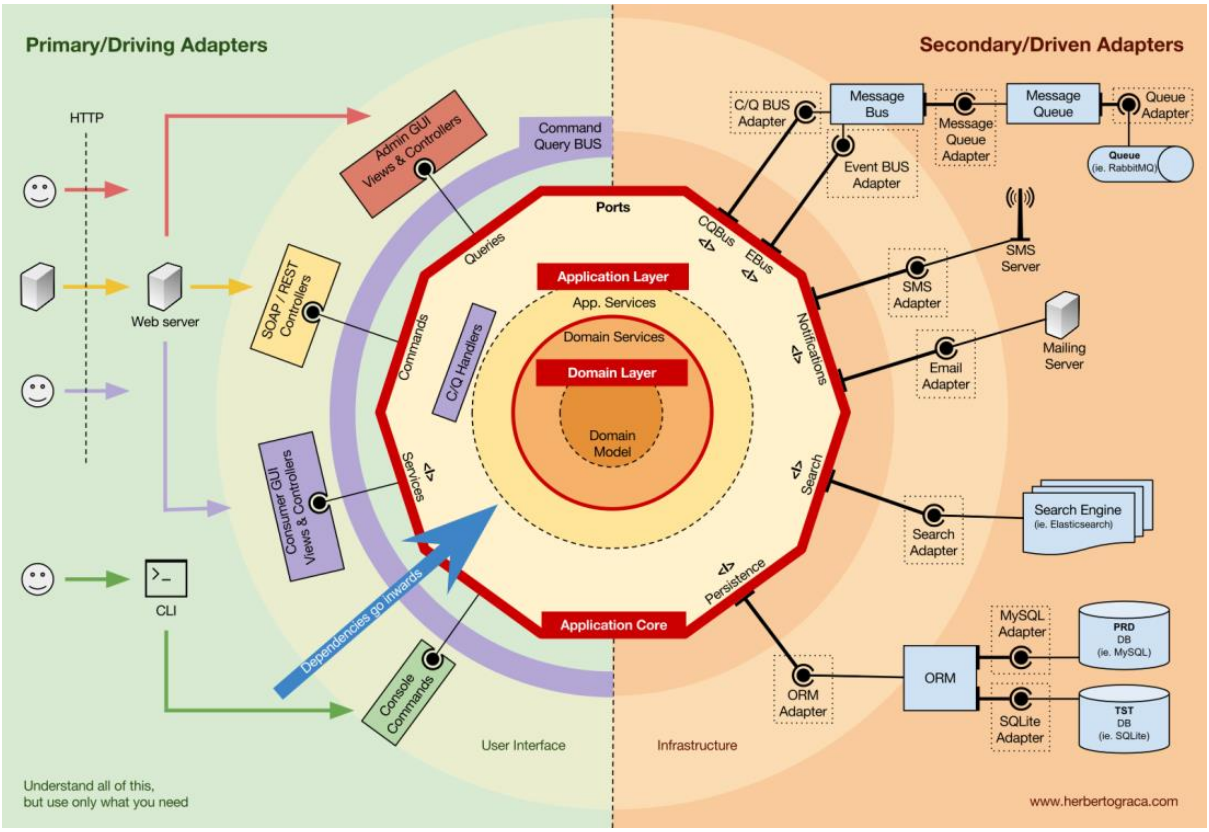


Fonte: https://docs.microsoft.com.

Arquitetura Hexagonal:

A arquitetura hexagonal, também conhecida como Ports and Adapters, tem os seus princípios bastante parecidos com a Clean Architecture. Possui 3 pilares – User Interface, Application Core, Infraestructure.

Figura 11 – Arquitetura Hexagonal.

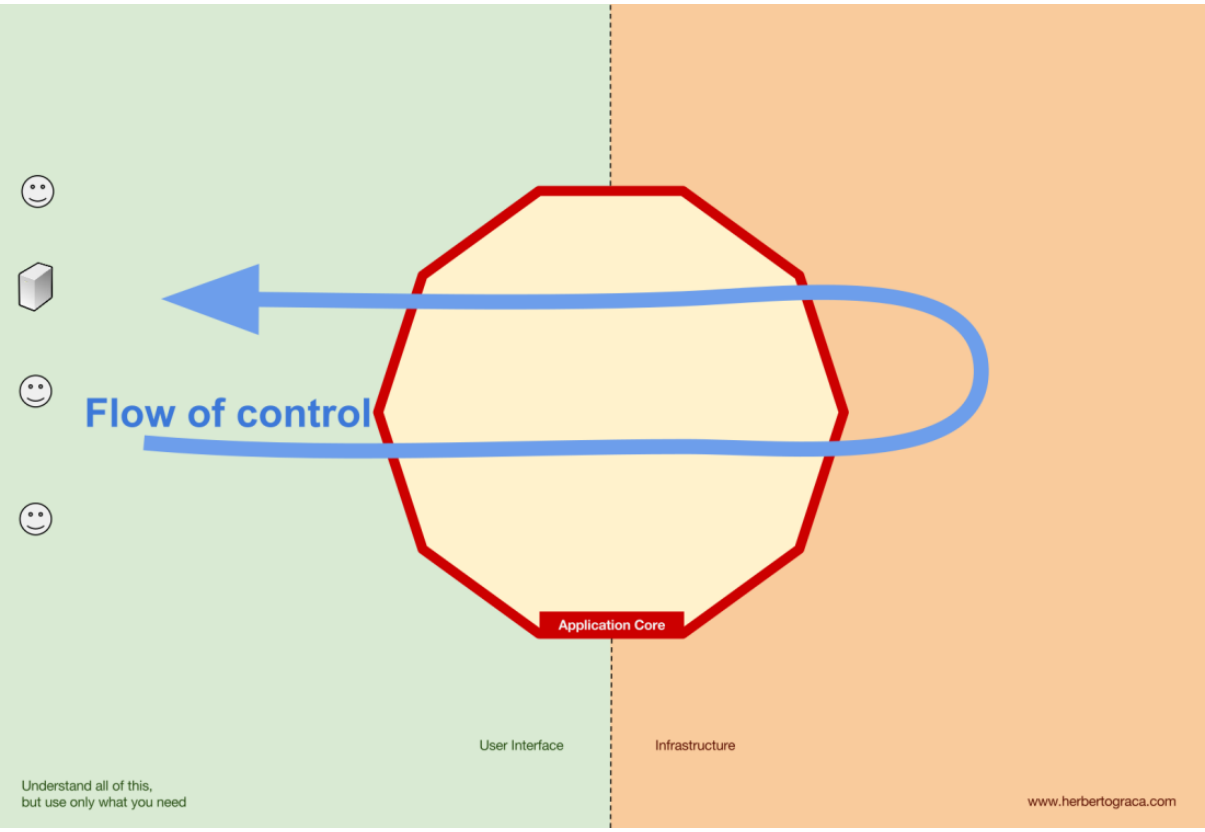


Fonte: https://herbertograca.com/2017/11/16/explicit-architecture-01-ddd-hexagonal-onion-clean-cqrs-how-i-put-it-all-together/.

Assim como na Clean Architecture, o centro da aplicação, application core, concentra toda a lógica de negócios e não há dependências das camadas superiores. Toda a comunicação das camadas superiores com as camadas inferiores é feita através de interfaces(Ports). Desta forma, a camada inferior expõe interfaces de saída e de entrada, enquanto que as camadas superiores, dependente das camadas inferiores, resolvem esses contratos com implementações concretas (Adapters).

O fluxo de interação entre as camadas se dá através de um fluxo iniciado em *User interface,* passando pelo *application core* até a camada de *infraestrutura* e finalmente faz o caminho inverso.

Figura 12 – Flow of control.



Fonte: https://herbertograca.com/2017/11/16/explicit-architecture-01-ddd-hexagonal-onion-clean-cqrs-how-i-put-it-all-together/**.**

É um modelo que implementa vários princípios arquiteturais, trás uma enorme flexibilidade, mas não se aplica em todos os contextos.

Arquitetura orientada a eventos:

Uma arquitetura orientada a eventos possui **produtores de eventos** que geram um fluxo de eventos, e **consumidores dos eventos** que escutam eventos.

Figura 13 – Pub / Sub.

De acordo com a Figura 12, o objeto A publica uma mensagem em um canal e o objeto B aguarda uma mensagem; e quando ela é consumida, não poderá ser reutilizada por outro objeto.

Os eventos são entregues quase em tempo real, para que os consumidores possam responder imediatamente conforme os eventos ocorrem. Os produtores são separados dos consumidores e não possuem conhecimento sobre os consumidores.

Esta arquitetura pode utilizar um modelo baseado em pub/sub ou um streaming de eventos.

Pub/sub: a infraestrutura de mensagens acompanha o controle de assinaturas. Quando um evento é publicado, ele envia o evento para cada assinante. Depois que um evento é recebido, ele não pode ser reproduzido e não será exibido para assinantes novos.

Streaming de eventos: eventos são gravados em um registro. Os eventos são estritamente ordenados (dentro de uma partição) e duráveis. Os clientes não assinam o fluxo, em vez disso, um cliente pode ler a partir de qualquer parte do

fluxo. O cliente é responsável por avançar a sua posição no fluxo. Isso significa que um cliente pode participar a qualquer momento e pode reproduzir eventos.

No lado do consumidor há algumas variações comuns:

Processamento de eventos simples: um evento dispara imediatamente uma ação no consumidor. Por exemplo, você pode usar as Azure Functions com um gatilho de Barramento de Serviço, para que uma função seja executada sempre que uma mensagem é publicada em um tópico do Barramento de Serviço.

Processamento de eventos complexos: um consumidor processa uma série de eventos, procurando padrões nos dados de eventos usando uma tecnologia, como o Azure Stream Analytics ou o Apache Storm. Por exemplo, você pode agregar as leituras de um dispositivo incorporado em uma janela de tempo e gerar uma notificação se a média móvel ultrapassar um certo limite.

Processamento de fluxo de eventos: use uma plataforma de fluxo de dados, como o Hub IoT do Azure ou o Apache Kafka, como um pipeline para ingestão de eventos e os encaminhe para os processadores de fluxo. Os processadores de fluxo agem para processar ou transformar o fluxo. Pode haver vários processadores de fluxo para subsistemas diferentes do aplicativo. Esta abordagem é uma boa opção para cargas de trabalho de IoT.

É uma abordagem arquitetural poderosa, sendo altamente escalada e distribuída, proporciona performance para a comunicação entre serviços mas possui desafios como, por exemplo, garantir da entrega de mensagens.

Capítulo 2. API

As APIs Web são um meio de expor funcionalidades que podem ser consumidas através do protocolo *HTTP* proporcionando independência de plataformas.

REST

É uma abordagem de arquitetura para criar serviços Web. REST é um estilo arquitetural, **independente do protocolo de comunicação**, para a criação de sistemas distribuídos com base em hipermídia. No entanto, as implementações mais comuns de REST usam HTTP como o protocolo de aplicativo.

Uma vantagem principal do REST sobre HTTP é que ele usa padrões abertos e não vincula a implementação da API ou os aplicativos cliente a nenhuma implementação específica. Por exemplo, um serviço Web REST poderia ser escrito em ASP.NET, e aplicativos cliente podem usar qualquer linguagem ou o conjunto de ferramentas que possa gerar solicitações HTTP e analisar respostas HTTP.

APIs REST são projetadas para *recursos*, que se tratam de qualquer tipo de objeto, dados ou serviço que possa ser acessado pelo cliente.

Figura 14 – Recursos REST.

Recurso	POST	GET	PUT	DELETE
/clientes	Criar um novo cliente	Obter todos os clientes	Atualização em massa de clientes	Remover todos os clientes
/clientes/1	Erro	Obter os detalhes do cliente 1	Atualizar os detalhes do cliente 1 se ele existir	Remover cliente 1
/clientes/1/pedidos	Criar um novo pedido para o cliente 1	Obter todos os pedidos do cliente 1	Atualização em massa de pedidos do cliente 1	Remover todos os pedidos do cliente 1

Fonte: https://docs.microsoft.com.

Conforme demonstrado na Figura 14, os objetos são recursos. Neste exemplo o recurso é **Cliente**, um serviço, que de acordo com o verbo HTTP, se comporta para retornar dados, remover, atualizar etc.

O protocolo HTTP define vários métodos que atribuem significado semântico a uma solicitação. Os métodos HTTP comuns, usados pelas APIs da Web mais RESTful, são:

- **GET**, que recupera uma representação do recurso no URI especificado. O corpo da mensagem de resposta contém os detalhes do recurso solicitado.
- **POST**, que cria um novo recurso no URI especificado. O corpo da mensagem de solicitação fornece os detalhes do novo recurso. Observe que POST também pode ser usado para disparar operações que, na verdade, não criam recursos.
- **PUT**, que cria ou substitui o recurso no URI especificado. O corpo da mensagem de solicitação especifica o recurso a ser criado ou atualizado.
- **PATCH**, que realiza uma atualização parcial de um recurso. O corpo da solicitação especifica o conjunto de alterações a ser aplicado ao recurso.
- **DELETE**, que remove o recurso do URI especificado.

Os objetos trafegados através do REST são, em sua maioria, serializados no formato JSON. Há a possibilidade de trafegar *strings*, *inteiros*, *boleanos*, *xml*. Mas os objetos JSON são mais recomendados por serem leves.

O padrão REST utiliza a semântica do protocolo HTTP, sendo assim, em seu fluxo, utiliza-se informações de cabeçalho (headers), status code, mime types para definições dos tipos trafegados etc. A utilização desta semântica facilita o modelo de comunicação entre as aplicações que utilizam recursos de sua API, podendo facilmente interagir e tomar decisões com base no response de um recurso, e até mesmo em um request.

Figura 15 – Requisição REST.

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "status":"In progress",
    "link": { "rel":"cancel", "method":"delete", "href":"/api/status/12345" }
}
```

Fonte: http://docs.microsoft.com.

A Figura 15 representa uma requisição do tipo GET, onde a resposta de retorno teve o **Status Code** como 200 OK (Sucesso) e, como conteúdo da resposta, um objeto JSON. O tipo do conteúdo é especificado pelo valor de *header* **Content-Type**.

Versionamento de API:

Ao trabalhar com a exposição de APIs, o gerenciamento dos recursos é sempre uma preocupação, visto que apesar de saber quais clientes consomem sua API, manter a compatibilidade com os consumidores em evoluções e alterações de contrato é de extrema importância. Para isso, o versionamento de APIS é uma ótima prática no desenho de APIs.

O versionamento de APIs pode ser feito de várias formas:

- Através do URL: http://www.minhaapi.com.br/v1/Clientes.
- Através de *Query String*: http://www.minhaapi.com.br/Clientes?version=v1.
- No cabeçalho:

Figura 16 – Versionamento por cabeçalho.

```
GET https://adventure-works.com/customers/3 HTTP/1.1
Custom-Header: api-version=1
```

Fonte: https://docs.microsoft.com.

- Por *media type*:

Figura 17 – Versionamento por media type.

```
GET https://adventure-works.com/customers/3 HTTP/1.1
Accept: application/vnd.adventure-works.v1+json
```

Fonte: https://docs.microsoft.com.

HATEOAS:

Uma das principais motivações por trás de REST, é a que deve ser possível navegar por todo o conjunto de recursos sem exigir conhecimento prévio do esquema de URI. Esse conceito é conhecido como HATEOAS. Cada solicitação HTTP GET, deve retornar as informações necessárias para localizar os recursos relacionados diretamente ao objeto solicitado por hiperlinks incluídos na resposta, e também deve ser provida de informações, descrevendo as operações disponíveis em cada um desses recursos. Não é um recurso tão comum de ser implementado e não existe um padrão completamente definido, de forma a direcionar como deve ser definida esta rastreabilidade por parte da resposta dos recursos. No entanto, algumas APIS como a do GitHub, utilizam parte destes recursos no cabeçalho de respostas das requisições para direcionar os consumidores da api.

Figura 18 – HATEOAS.


```
{
  "orderID":3,
  "productID":2,
  "quantity":4,
  "orderValue":16.60,
  "links":[
    {
      "rel":"customer",
      "href":"https://adventure-works.com/customers/3",
      "action":"GET",
      "types":["text/xml","application/json"]
    },
    {
      "rel":"customer",
      "href":"https://adventure-works.com/customers/3",
      "action":"PUT",
      "types":["application/x-www-form-urlencoded"]
    },
    {
      "rel":"customer",
      "href":"https://adventure-works.com/customers/3",
      "action":"DELETE",
      "types":[]
    },
    {
      "rel":"self",
      "href":"https://adventure-works.com/orders/3",
      "action":"GET",
      "types":["text/xml","application/json"]
    },
    {
      "rel":"self",
      "href":"https://adventure-works.com/orders/3",
      "action":"PUT",
      "types":["application/x-www-form-urlencoded"]
    },
    {
      "rel":"self",
      "href":"https://adventure-works.com/orders/3",
      "action":"DELETE",
      "types":[]
    }]
}
```

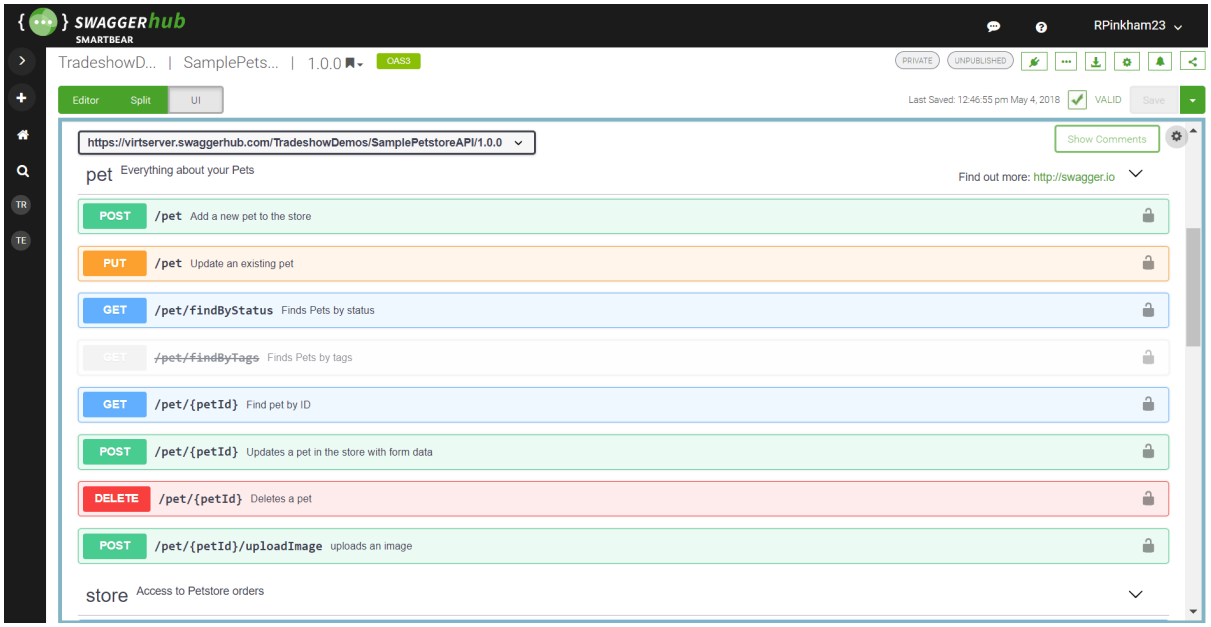
Fonte: https://docs.microsoft.com.

Documentação:

Apesar das técnicas disponibilizadas pela especificação REST para a disponibilização de recursos de forma facilitada, uma boa documentação é de extrema importância e não seria diferente em APIs. O uso de documentações é recomendado para que equipes possam interagir diretamente com a API, entendendo os seus contratos, status code e até mesmo testar.

As tecnologias possuem vários pacotes, que auxiliam na disponibilização de documentações. Uma ótima opção é o Swagger. Possui uma interface simples, trabalha com versionamento e com OpenId.

Figura 19 – Swagger.



Fonte: https://swagger.io.

GRPC

Uma outra abordagem de comunicação entre APIs é o GRPC, uma tecnologia criada pelo Google a fim de obter maior performance na comunicação entre microsserviços.

O GRPC é uma estrutura de RPC (Remote Procedure call) de alto desempenho independente de linguagem.

Os principais benefícios de gRPC, são:

- Estrutura de RPC leve, moderna e de alto desempenho.
- Desenvolvimento da API baseada em contratos, usando buffers de protocolo por padrão, permitindo implementações independente de linguagem.
- As ferramentas disponíveis para várias linguagens gerarem clientes e servidores fortemente tipados.
- Dá suporte ao cliente, servidor e chamadas bi-direcionais de streaming.
- Uso de rede reduzida com a serialização binária Protobuf.

Esses benefícios tornam o gRPC ideal para:

- Microsserviços leves em que a eficiência é crítica.
- Sistemas poliglotas, nos quais múltiplas linguagens são necessárias para o desenvolvimento.
- Serviços ponto a ponto em tempo real, que precisam lidar com solicitações ou respostas de streaming.

Enquanto o REST diz respeito a recursos, o gRPC está relacionado a procedimentos. É uma evolução do padrão RPC.

O gRPC trabalha com contratos. Esta especificação é feita através de um arquivo denominado .protobuf.

Figura 20 – Arquivo protobuf.

```
syntax = "proto3";

service Greeter {
  rpc SayHello (HelloRequest) returns (HelloReply);
}

message HelloRequest {
  string name = 1;
}

message HelloReply {
  string message = 1;
}
```

Fonte: https://docs.microsoft.com.

Este arquivo especifica quais são as chamadas disponíveis (métodos), além da especificação dos tipos de entrada e saída. Este arquivo permite a geração de clientes e posteriormente a implementação dos contratos na tecnologia de preferência.

Figura 21 – Implementação C#.

```
public class GreeterService : Greeter.GreeterBase
{
    private readonly ILogger<GreeterService> _logger;

    public GreeterService(ILogger<GreeterService> logger)
    {
        _logger = logger;
    }

    public override Task<HelloReply> SayHello(HelloRequest request,
        ServerCallContext context)
    {
        _logger.LogInformation("Saying hello to {Name}", request.Name);
        return Task.FromResult(new HelloReply
        {
            Message = "Hello " + request.Name
        });
    }
}
```

Fonte: https://docs.microsoft.com.

A Figura 21 representa uma implementação de um serviço que será consumido no padrão gRPC, utilizando C#. Esta implementação pode variar de acordo com a tecnologia. Após definidos os contratos, suas implementações e particularidades de cada tecnologia, é possível consumir o recurso em um canal:

Figura 22 – Consumindo através do gRPC.

```
var channel = GrpcChannel.ForAddress("https://localhost:5001");
var client = new Greeter.GreeterClient(channel);

var response = await client.SayHelloAsync(
    new HelloRequest { Name = "World" });

Console.WriteLine(response.Message);
```

Fonte: https://docs.microsoft.com.

A Figura 22 apresenta a criação de um canal através do endereço *https://localhost.5001*, utilizando um cliente gerado com base no arquivo *protobuf* e o consumo do método *SayHelloAsync* especificado no contrato.

É uma ótima tecnologia, com muitos benefícios mas que não se aplica em todos os casos. Ainda existem limitações para utilizá-la através dos navegadores, mas existe a possibilidade de utilizar um recurso como o gRPC-Web, uma tecnologia adicionar para fornecer suporte a este cenário.

GgraphQL

Uma outra tecnologia disponível que, no primeiro momento, causa muita confusão, é o GraphQL. O GraphQL não é uma tecnologia de banco de dados exclusiva para APIs e não é um ORM. De uma forma bem simples, esta tecnologia é um *Query language* criada pelo Facebook. Portanto não depende de um database específico ou tecnologia.

Esta tecnologia, baseada em grafos, facilita a definição de tipos e queries e schemas. Seu objetivo é performance, eliminar buscas excessivas de dados.

Suas características auxiliam o backend para ser mais estável por garantir tempos de respostas mais rápidos (menos dados a serem trafegados), mas não deve ser utilizada em todos os contextos. Uma das dificuldades neste padrão é a curva de aprendizado, um modelo diferente de se trabalhar em relação ao REST; o armazenamento em cache é mais difícil e as consultas sempre retornam um status code 200.

Existem diversas implementações do GraphQL para diversas tecnologias. Uma delas é uma extensão para se trabalhar com *IQueryable* no .net core. Vejamos um exemplo:

Figura 23 – Definião de classes em C#.

```
public class User
{
    public int Id { get; set; }
    public string Name { get; set; }

    public int AccountId { get; set; }
    public Account Account { get; set; }
}

public class Account
{
    public int Id { get; set; }
    public string Name { get; set; }
    public bool Paid { get; set; }
}
```

Fonte: https://github.com/chkimes/graphql-net.

A Figura 22 apenas define duas classes utilizando C#. Mas para a adaptação do modelo do GraphQL, utilizando esta implementação, é necessário definir um schema.

Figura 24 – Definindo um schema no DbContext.

```
var schema = GraphQL<TestContext>.CreateDefaultSchema(() => new TestContext());
```

Fonte: https://github.com/chkimes/graphql-net.

Nest Código, exibido na Figura 24, temos a criação de um schema associado a um DbContext, responsável pela interação com a base de dados.

Figura 25 – Adicionando tipo User ao schema do GraphQL.

```
var user = schema.AddType<User>();
user.AddField(u => u.Id);
user.AddField(u => u.Name);
user.AddField(u => u.Account);
user.AddField("totalUsers", (db, u) => db.Users.Count());
user.AddField("accountPaid", (db, u) => u.Account.Paid);
```

Fonte: https://github.com/chkimes/graphql-net.

Figura 26 – Adicionando tipo Account ao schema do GraphQL.

```
schema.AddType<Account>().AddAllFields();
```

Fonte: https://github.com/chkimes/graphql-net.

Figura 27 – Adicionando queries ao schema do GraphQL.

```
schema.AddListField("users", db => db.Users);
schema.AddField("user", new { id = 0 }, (db, args) => db.Users.Where(u => u.Id == args.id).FirstOrDefault());
```

Fonte: https://github.com/chkimes/graphql-net.

As Figuras 25, 26 e 27 demonstram a criação de schemas no GraphQL com base nas classes em C#. Espeficiamente na Figura 27, temos a adição de duas queries:

- **Users**: que retorna todos os usuários e suas propriedades, diretamente da base de dados.
- **User**: que define uma consulta parametrizada pelo **Id** para a obtenção de um usuário na base de dados.

Após completar o schema, a aplicação está pronta para receber consultas:

Figura 28 – Consulta em GraphQL utilizando o IQueryable.

```
var query = @"{
user(id:1) {
    userId : id
    userName : name
    account {
        id
        paid
    }
    totalUsers
}}";

var gql = new GraphQL<TestContext>(schema);
var dict = gql.ExecuteQuery(query);
Console.WriteLine(JsonConvert.SerializeObject(dict, Formatting.Indented));

// {
//   "user": {
//     "userId": 1,
//     "userName": "Joe User",
//     "account": {
//       "id": 1,
//       "paid": true
//     },
//     "totalUsers": 2
//   }
// }
```

Fonte: https://github.com/chkimes/graphql-net.

Como apresentado na Figura 28, a consulta efetuada deseja apenas o usuário de código 1 e, consequentemente, algumas propriedades que são necessárias para este caso. Não temos a necessidade de retornar todas as propriedades de um tipo, trazendo mais flexibilidade e diminuindo o tráfego.

The Twelve-factor app

É sempre bom ter referências para nos direcionar nas melhores decisões e boas prática. Neste caso, um ótimo recurso que possui boas práticas para a criação

de APIs é o site The Twelve-factor app, que disponibiliza 12 princípios a serem seguidos, para se balizar no desenvolvimento de soluções com a melhor qualidade em diversos níveis. Disponível em: http://12factor.net.

Capítulo 3. Arquitetura de Sistemas Mobile

Assim como qualquer outra tecnologia, o desenvolvimento mobile traz grandes desafios e muitas possibilidades.

Com a necessidade do mercado na entrega de aplicações para dispositivos móveis, ter a melhor estratégia e as melhores tecnologias podem fazer grande diferença na tomada decisões e entregar valor o mais rapidamente.

A arquitetura mobile se divide, basicamente, em três modelos:

- Nativo.
- Cross-Platform.
- Híbrido.

Cada modelo possui suas especificações, prós, contras e o uso de uma estratégia ou outra vai depender do contexto.

O que vai diferenciar em relação ao modelo nativo para os demais é, justamente, o suporte e performance. Aplicações nativas tem mais poder de uso dos recursos de suas respectivas plataformas, acessando diretamente os recursos de suas APIs. Além disso, com o avanço da tecnologia, todas as abordagens arquiteturais mobile possuem a capacidade de aplicação de princípios arquiteturais de mercado, como os citados nesta apostila. Mas conhecer as plataformas é muito importante para determinar o melhor desenho. Sendo assim, é possível e recomendado utilizar recursos como separação de camadas, Clean Architecture, entre outros modelos.

Arquitetura nativa

Este modelo arquitetural está associado, especificamente, ao desenvolvimento de soluções móveis utilizando os SDKs disponíveis em cada

plataforma. Neste caso, para os principais players do mercado, temos as plataformas Android e IOS.

Android

É uma ótima plataforma criada pelo Google e que está presente em grande parte dos dispositivos mundiais.

O desenvolvimento Android iniciou-se baseado no uso da tecnologia **Java**. Hoje ainda suporta esta tecnologia, porém uma nova linguagem, **Kotlin**, vem ganhando força.

Para iniciar o desenvolvimento em Android, você vai precisar de uma IDE, e hoje a melhor IDE para este caso é o Android Studio, além do SDK e uma conta de desenvolvedor na Google Play. A conta não é mandatória para o início do desenvolvimento, porém, para a publicação dos aplicativos, será necessário obter uma conta. O pagamento é feito anualmente.

O desenvolvimento poderá ser feito em qualquer sistema operacional, Windows, Linux ou Mac, o que faz com que a plataforma seja muito popular.

O SDK do Android é muito poderoso, tendo várias ferramentas para auxiliar no desenvolvimento, além de um ótimo emulador que possibilita executar a maioria dos recursos que são oferecidos em dispositivos reais.

IOS

Esta é uma plataforma criada pela Apple. Possui muito mais restrições em relação ao desenvolvimento Android, porém é extremamente poderosa.

Inicialmente a linguagem oficial do IOS era o **Objective-C**. No entanto, houve uma grande evolução e a linguagem oficial passou a ser o **Swift**.

Para iniciar o desenvolvimento para IOS, será necessário um ambiente da Apple, portanto será necessário possuir um IOS com a IDE Xcode instalada. Ao instalar o XCode, todo o ambiente estará disponível.

Um ponto importante é que é possível desenvolver na plataforma e testar suas aplicações no emulador. Entretanto, faz-se necessário possuir uma conta de desenvolvedor na Apple Store, também paga anualmente. Sem esta conta não será possível publicar aplicativos na loja e nem os testar em seus dispositivos da Apple.

Arquitetura Cross-Platform

O grande benefício de arquiteturas cross-platform está associado na entrega de soluções para as duas principais plataformas de mercado, utilizando o mesmo code base. Neste modelo, o desenvolvimento da lógica de negócio e também das interfaces é reutilizado e "traduzido" para cada plataforma.

Um outro fator muito importante é a performance. Sendo assim, estas tecnologias atuam em uma camada antes da camada nativa. Então, no processo de compilação de um aplicativo, há uma pré-compilação e posteriormente é entregue aos SDKs das plataformas específicas, para proceder com a compilação nativa. Isso é uma grande vantagem. Entregar soluções para plataformas diferentes, utilizando uma linguagem, uma IDE e ainda ter performance nativa. Entretanto, em alguns casos há uma pequena perda de performance, já que na maioria das vezes as plataformas necessitam de algum componente, como é o caso do Xamarin, que depende do Mono instalado no dispositivo. Isso acaba gerando uma perda. Mas dependendo do tipo de aplicação, esta perda não faz tanta diferença.

As principais tecnologias cross-platfform existentes no mercado, são:

- **Xamarin –** Mantida pela Microsoft, utiliza toda a plataforma .Net e seus recursos para criar aplicativos para as duas plataformas. Em seu processo de compilação, é gerado uma linguagem intermediária e direcionada ao compilador nativo de cada plataforma para compilação. Tem a dependência da

plataforma MONO no dispositivo. Atualmente é possível desenvolver em Xamarin através do Windows e Mac.

- **Reactive Native** – Criada pelo Facebook, utiliza JavaScript, HTML e CSS. É uma plataforma muito produtiva e também entrega soluções para IOS e Android. O seu modelo de desenvolvimento facilita a adoção em qualquer sistema operacional.

- **Flutter** - É uma plataforma criada e mantida pelo Google. Tem o código fonte aberta, assim como as outras três plataformas, utiliza Dart como linguagem de programação e também pode ser desenvolvido em qualquer sistema operacional. É uma tecnologia muito poderosa, inclusive tem a melhor performance para aplicativos móveis.

Todas estas plataformas dependem, no caso de aplicações iOS, de um computador MAC com todo o ambiente configurado para compilar a aplicação. Contudo, a performance de cada uma delas é perfeitamente aceitável e o acesso aos APIs são extremamente comuns, como nativo.

Arquitetura Híbrida

As arquiteturas híbridas são mais uma opção no desenvolvimento de aplicativos móveis e, dependendo do seu contexto, utilizar esta abordagem pode ser uma ótima estratégia para a entrega de valor.

Os aplicativos desenvolvidos nesta abordagem podem ser instalados através das Stores de cada plataforma e possuem acesso facilitado às APIs. Se baseiam em JavaScript, HTML e CSS, contudo, após os aplicativos serem gerados, eles são executados em uma WebView Full Screen. Apesar disso as arquiteturas híbridas são extremamente poderosas.

Uma das plataformas mais famosas é o IONIC. Inicialmente utilizava o framework Angular como base para o desenvolvimento, mas atualmente é possível utilizar Angular, VueJs, JavaScript e React.

O Ionic possui vários componentes e abstrações para acesso às APIs nativas de cada plataforma e, além disso, pode ser utilizada também em ambientes Web, facilitando o desenvolvimento do mesmo código para entrega em diversas plataformas.

Capítulo 4. Arquitetura de Microsserviços

As empresas estão cada vez mais percebendo a economia de custo ao resolver problemas de implantação e melhorar as operações de produção e de DevOps usando os contêineres.

A Microsoft tem lançando inovações de contêiner para Windows e Linux com a criação de produtos como o Serviço de Kubernetes do Azure e o Azure Service Fabric por meio de parcerias com líderes do setor, como a Docker, a Mesosphere e a Kubernetes. Esses produtos oferecem soluções de contêiner que ajudam as empresas a criar e implantar aplicativos com a velocidade e a escala da nuvem, seja qual for a escolha de plataformas ou de ferramentas.

O Docker está se tornando o verdadeiro padrão no setor de contêineres, com suporte dos fornecedores mais significativos nos ecossistemas do Windows e do Linux (a Microsoft é um dos principais fornecedores de nuvem que suportam o Docker.). No futuro, o Docker provavelmente estará onipresente em qualquer datacenter na nuvem ou no local.

Além disso, a arquitetura de microsserviços está despontando como uma abordagem importante para aplicativos críticos distribuídos. Em uma arquitetura baseada em microsserviços, o aplicativo é criado em uma coleção de serviços que podem ser desenvolvidos, testados, implantados e ter as versões controladas de forma independente.

O que são Microsserviços?

- Os microsserviços são serviços pequenos, independentes e relativamente interligados.
- Cada serviço é um código base separado, que pode ser gerido por uma equipe de desenvolvimento pequena.

- Os serviços podem ser implementados de forma independente. São responsáveis pela persistência dos seus próprios dados ou de seu próprio estado externo.

- Os serviços comunicam entre si através de APIs bem definidas.

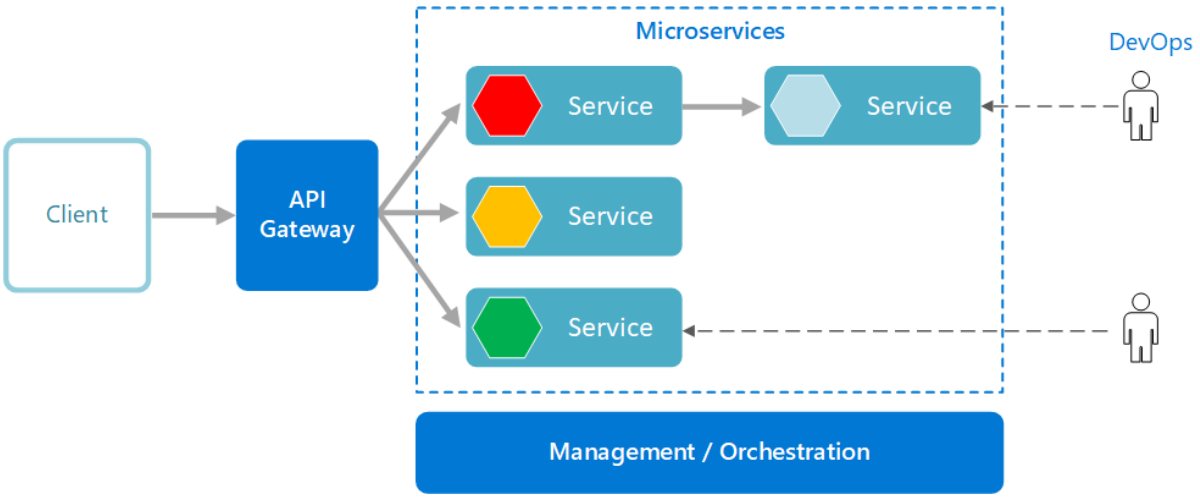
- Os serviços não precisam de partilhar a mesma pilha de tecnologia, bibliotecas ou estruturas.

Além dos próprios serviços, alguns outros componentes estão presentes numa arquitetura de microsserviços típica:

Gestão/orquestração. Este componente é responsável por colocar os serviços em nós, identificando falhas, reequilibrando os serviços nos nós, e assim sucessivamente. Normalmente, este componente é uma tecnologia pronta para ser utilizada, como o Kubernetes, em vez de ser algo personalizado.

API Gateway. O gateway da API é o ponto de entrada para os clientes. Em vez de chamarem serviços diretamente, os clientes chamam o gateway de API, o qual reencaminha a chamada para os serviços adequados no back-end.

Figura 29 – Exemplo de arquitetura de Microsserviços.



Fonte: https://docs.microsoft.com.

Benefícios

A arquitetura baseada em microsserviços traz muitos desafios, mas também muitos benefícios. É uma estratégia na qual há uma contextualização ou divisão de responsabilidades. Sendo assim, podemos ter times distintos para manter os serviços, independência de promoções aos ambientes produtivos, a escalabilidade, entre outros benefícios:

- **Agilidade.** Como os microsserviços são implementados de forma independente, pode-se gerir a correção de erros e o lançamento de funcionalidades com mais facilidade. Pode-se atualizar um serviço sem implementar novamente toda a aplicação, e reverter uma atualização caso algo corra mal. Em muitas aplicações tradicionais, quando se encontra um erro numa parte da aplicação, este pode bloquear todo o processo de lançamento. As novas funcionalidades podem ficar retidas enquanto a correção do erro é integrada, testada e publicada.

- **Equipes pequenas e direcionadas**: um microsserviço deve ser pequeno ao ponto de ser possível que uma única equipe consiga criá-lo, testá-lo e implementá-lo. Equipes menores promovem uma maior agilidade. Equipes grandes tendem a ser menos produtivas, porque a comunicação é mais demorada, a despesa de gestão aumenta e a agilidade diminui.

- **Base de código pequena**: numa aplicação monolítica, há a tendência de, ao longo do tempo, as dependências de código ficarem emaranhadas umas nas outras, e para se adicionar uma nova funcionalidade é necessário mexer em código em diferentes locais. Como os microsserviços não partilham código nem arquivos de dados, a arquitetura de microsserviços minimiza as dependências, o que torna mais fácil adicionar novas funcionalidades.

- **Misto de tecnologias**: as equipes podem escolher a tecnologia mais adequada ao respectivo serviço através da utilização de várias tecnologias diferentes, conforme as necessidades.

- **Isolamento de falhas**: se um microsserviço individual ficar indisponível, essa ocorrência não afetará toda a aplicação, desde que os microsserviços de origem estejam preparados para lidar corretamente com as falhas (por exemplo, através da implementação de um disjuntor de circuito).

- **Escalabilidade**: os serviços podem ser dimensionados independentemente, o que permite a ampliação de subsistemas que necessitem de mais recursos sem que, para isso, se tenha de aumentar horizontalmente toda a aplicação. Ao utilizar um orquestrador como o Kubernetes ou o Service Fabric, pode-se incluir uma maior densidade de serviços num único anfitrião, o que permite uma utilização mais eficiente dos recursos.

- **Isolamento de dados**: é muito mais fácil fazer atualizações de esquema porque apenas é afetado um único microsserviço. Numa aplicação monolítica, as atualizações de esquema podem ser muito difíceis de fazer, porque diferentes partes da aplicação podem partilhar os mesmos dados, o que faz com que seja arriscado proceder com as alterações.

Desafios

No entanto, é claro que a abordagem de microsserviços também nos impões vários desafios:

- **Complexidade**. Um aplicativo de microsserviços tem mais partes móveis que o aplicativo monolítico equivalente. Cada serviço é mais simples, mas o sistema como um todo é mais complexo.

- **Desenvolvimento e teste**. Escrever um pequeno serviço que precise de outros serviços dependentes exige uma abordagem diferente da escrita de um aplicativo monolítico ou em camadas tradicional. As ferramentas existentes nem sempre são projetadas para funcionar com dependências de serviço. Também pode ser um desafio testar as dependências de serviço, especialmente quando o aplicativo está evoluindo rapidamente.

- **Falta de governança**. A abordagem descentralizada para compilar microsserviços tem vantagens, mas também pode causar problemas. Você pode acabar com muitos idiomas e estruturas diferentes que tornem difícil a manutenção do aplicativo. Pode ser útil estabelecer alguns padrões para todo o projeto, sem restringir excessivamente a flexibilidade das equipes. Isso se aplica especialmente a funcionalidades abrangentes, como registro em log.

- **Latência e congestionamento de rede**. O uso de muitos serviços granulares pequenos pode resultar em mais comunicação entre serviços. Além disso, se a cadeia de dependências de serviço ficar muito longa (o serviço A chamada o B, que chama o C…), a latência adicional poderá se tornar um problema. Você precisará projetar APIs com cuidado. Evite APIs excessivamente prolixas, pense em formatos de serialização e procure locais para usar padrões de comunicação assíncrona.

- **Integridade de dados**. Cada microsserviço deve ser responsável pela própria persistência de dados. Assim, a consistência dos dados pode ser um desafio. Adote consistência eventual quando possível.

- **Gerenciamento**. Ter êxito com microsserviços requer uma cultura DevOps madura. Registro em log correlacionado entre serviços pode ser desafiador. Normalmente, o registro em log deve correlacionar várias chamadas de serviço para uma operação de um único usuário.

- **Controle de versão**. As atualizações de um serviço não devem interromper os serviços que dependerem delas. Vários serviços podem ser atualizados a qualquer momento, portanto, sem design cuidadoso, você pode ter problemas com compatibilidade com versões anteriores ou futuras.

- **Conjunto de qualificações**. Os microsserviços são sistemas altamente distribuídos. Avalie cuidadosamente se a equipe tem as habilidades e a experiência necessária para ser bem-sucedida.

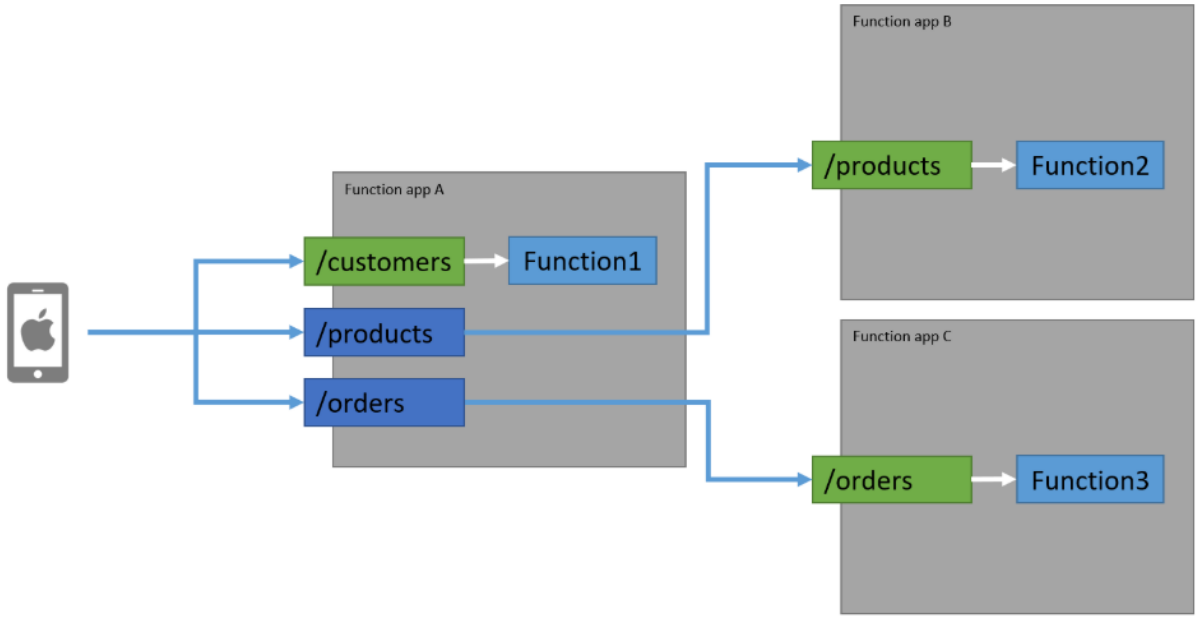
Boas práticas

- Modele os serviços em torno de domínio da empresa.
- Descentralize tudo. Equipes individuais são responsáveis por projetar e criar serviços. Evite compartilhar esquemas de dados ou códigos.
- O armazenamento de dados deve ser privado para o serviço que é o proprietário dos dados. Use o melhor armazenamento para cada serviço e tipo de dados.
- Os serviços comunicam-se por meio de APIs bem projetadas. Evite o vazamento de detalhes da implementação. As APIs devem modelar o domínio, não a implementação interna do serviço.
- Evite acoplamento entre serviços. Causas de acoplamento incluem protocolos de comunicação rígidos e esquemas de banco de dados compartilhados
- Descarregue preocupações abrangentes, como autenticação e terminação SSL, para o gateway.
- Mantenha o conhecimento de domínio fora do gateway. O gateway deve tratar e rotear solicitações de cliente sem qualquer conhecimento das regras de negócios ou da lógica do domínio. Caso contrário, o gateway se tornará uma dependência e poderá causar um acoplamento entre serviços.
- Os serviços devem ter um acoplamento flexível e uma alta coesão funcional. Funções que provavelmente mudarão juntas devem ser empacotadas e implantadas juntas. Se residirem em serviços separados, esses serviços acabarão sendo fortemente acoplados, porque uma alteração em um serviço exigirá atualizar outro. Uma comunicação excessivamente prolixa entre dois serviços pode ser um sintoma de acoplamento forte e coesão baixa.
- Isole falhas. Use estratégias de resiliência para impedir que falhas em um serviço distribuam-se em cascata.

API Gateway

Um gateway de API fornece um ponto único de entrada para clientes e, em seguida, roteia solicitações de forma inteligente para serviços de back-end. É útil gerenciar grandes conjuntos de serviços. Ele também pode lidar com controle de versão e simplificar o desenvolvimento, conectando facilmente clientes a ambientes diferentes. Pode lidar com o dimensionamento de back-end de microservices individuais ao mesmo tempo em que apresenta um único front-end por meio de um gateway de API.

Figura 29 – API Gateway.

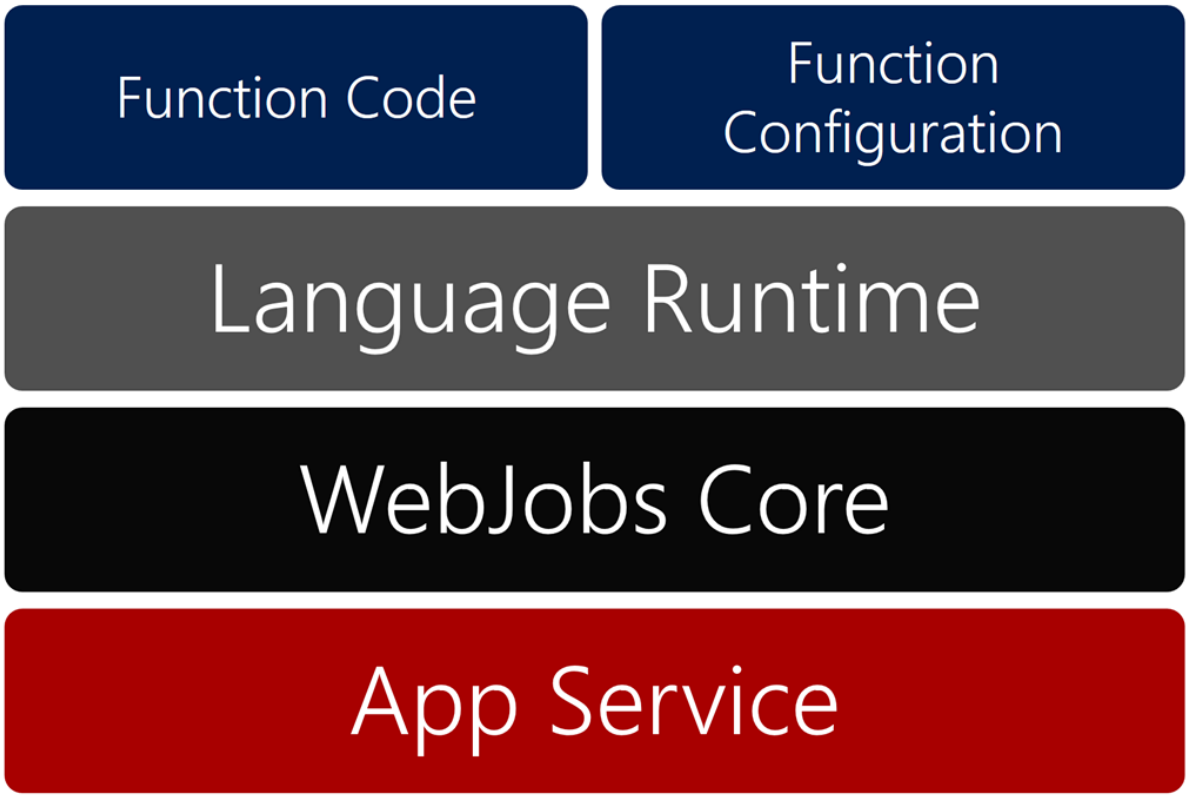


Font: https://docs.microsoft.com.

Capítulo 5. Arquitetura Serverless

Há muitas abordagens para o uso de arquiteturas serverless . Os hosts serverless geralmente usam uma camada de PaaS ou baseada em contêiner existente para gerenciar as instâncias sem servidor. Por exemplo, Azure Functions se baseia no serviço Azure app. O app service é usado para expandir instâncias e gerenciar o tempo de execução do código das Azure Functions. Para functions baseadas no Windows, o host é executado como PaaS e dimensiona o tempo de execução do .NET. Para funções baseadas em Linux, o host aproveita contêineres.

Figura 30 – Camadas da arquitetura de Azure Functions.



Fonte: https://docs.microsoft.com.

Full Backend Serverless

O Flull backend serverlers é ideal para vários tipos de cenários, especialmente ao criar aplicativos novos ou "green fields". Um aplicativo com uma grande área de superfície de APIs pode se beneficiar da implementação de cada API como uma função sem servidor. Aplicativos baseados na arquitetura de microsserviços são outro exemplo que podem ser implementados como um back-end completo sem servidor. Os microsserviços se comunicam por vários protocolos entre si. Os cenários específicos incluem:

- Produtos SaaS baseados em API (exemplo: processador de pagamentos financeiros).
- Aplicativos orientados a mensagens (exemplo: solução de monitoramento de dispositivo).
- Aplicativos focados na integração entre serviços (exemplo: aplicativo de reservas de viagens).
- Processos que são executados periodicamente (exemplo: limpeza de banco de dados baseada em temporizador).
- Aplicativos focados na transformação de dados (exemplo: importação disparada por upload de arquivo).
- Extrair processos de ETL (transformação e carregamento).

É uma estratégia poderosa por prover rapidamente uma solução de backend, e pode ser utilizadas em vários cenários interessantes:

Aplicações Web

As aplicações Web são excelentes candidatas para o uso de Serverless. Atualmente, há duas abordagens comuns para aplicativos Web: *server-driven* e cliente-driven (como um SPA). Aplicativos Web baseados em servidor

normalmente usam uma camada de middleware para emitir chamadas de API para renderizar a interface do usuário da Web.

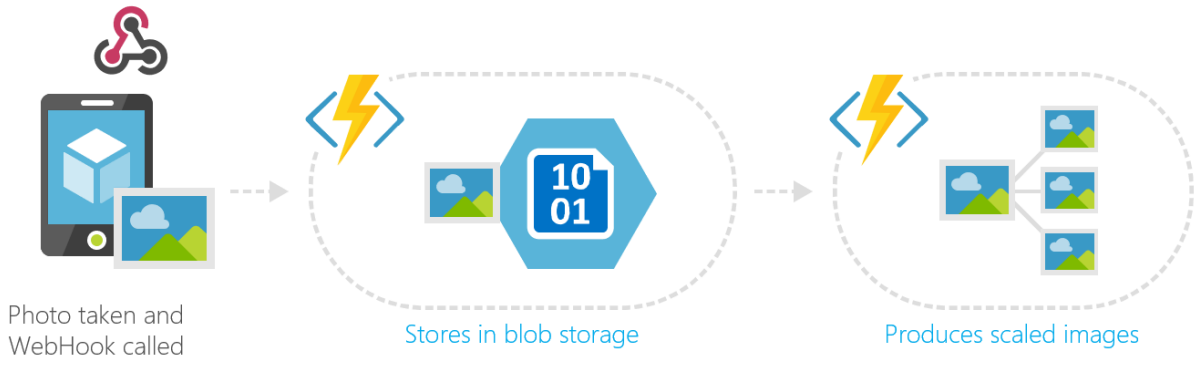
Os aplicativos SPA fazem chamadas à API REST diretamente do navegador. Em ambos os cenários, o servidor pode acomodar o middleware ou a solicitação da API REST fornecendo a lógica comercial necessária. Uma arquitetura comum é criar um servidor Web estático leve. O SPA serve HTML, CSS, JavaScript e outros ativos de navegador. O aplicativo Web, então, se conecta a um back-end de microsserviço.

Backend para aplicações móveis

O paradigma event-driven de aplicações Serverless as torna ideais como backends móveis. O dispositivo móvel dispara os eventos e o código sem servidor é executado para atender a solicitações. Tirar proveito de um modelo serverles permite que os desenvolvedores aprimorem a lógica de negócios sem precisar implantar uma atualização completa do aplicativo, e também permite que as equipes e *endpoints* trabalhem em paralelo.

O sererless abstrai as dependências do lado do servidor e permite que o desenvolvedor se concentre na lógica de negócios. Por exemplo, um desenvolvedor móvel que cria aplicativos usando uma estrutura JavaScript (arquitetura híbrida) também pode criar funções serverless com JavaScript em Node. O desenvolvedor recebe um conjunto simples de entradas e um modelo padrão para saídas. Em seguida, eles podem se concentrar na criação e no teste da lógica de negócios. Portanto, eles são capazes de usar habilidades existentes para criar a lógica de backend para o aplicativo móvel sem precisar aprender novas plataformas ou se tornar um "desenvolvedor backend".

Figura 31 – Serverless e aplicações móveis.



Fonte: https://docs.microsot.com.

IOT

O IoT refere-se a objetos físicos que estão conecetados. Às vezes, eles são chamados de "connected devices" ou "smart devices". Tudo, desde carros até máquinas de vendas, pode estar conectado e enviar informações que vão desde o inventário até dados de sensor, como temperatura e umidade. Na empresa, a IoT fornece melhorias de processos de negócios por meio de monitoramento e de automação. Os dados de IoT podem ser usados para regulamentar o clima em um grande depósito ou acompanhar o inventário por meio da cadeia de suprimentos. A IoT pode detectar derramamentos químicos e chamar o departamento de incêndio quando a fumaça for detectada.

O enorme volume de dispositivos e informações geralmente determina uma arquitetura orientada por eventos para rotear e processar mensagens. O serverless é uma solução ideal por vários motivos:

- Habilita o dimensionamento conforme o volume de dispositivos e dados aumenta.
- Acomoda a adição de novos endpoints para dar suporte a novos dispositivos e sensores.

- Facilita o controle de versão independente, para que os desenvolvedores possam atualizar a lógica de negócios de um dispositivo específico sem precisar implantar o sistema inteiro.

- Resiliência e menos tempo de inatividade.

A disseminação da IoT resultou em vários produtos serverless que se concentram especificamente em preocupações com a IoT, como o Hub IOT do Azure. O servidor automatiza tarefas como registro de dispositivos, imposição de políticas, acompanhamento e até mesmo implantação de código para *devices at the edge*. São dispositivos como sensores e atuadores que estão conectados à Internet, mas não a uma parte ativa.

Considerações

A adoção de uma arquitetura serverless vem com determinados desafios. Todos esses desafios têm soluções. Assim como acontece com todas as opções de arquitetura, a decisão de utilizar serverless deve ser feita somente após considerar atentamente os prós e contras. Dependendo das necessidades do seu contexto, não será a melhor escolha.

- **Gerenciamento de estados** – As functions Serverless, assim como com os microservices em geral, não têm estado por padrão. Evitar o estado permite que o serverless seja efêmero, para escalar horizontalmente e para fornecer resiliência sem um ponto central de falha.

- **Durable process** - Tempos de execução curtos facilitam para o provedor serverless liberar recursos à medida em que as funções terminam e compartilham funções entre os hosts. A maioria dos provedores de nuvem limita o tempo total que sua função pode executar a cerca de 10 minutos. Se o processo puder levar mais tempo, você poderá considerar uma implementação alternativa.

- **Tempo de inicialização** - Uma preocupação potencial com implementações serverless é o tempo de inicialização (Warm-up). Para conservar recursos, muitos provedores serverless criam infraestrutura "sob demanda". Quando uma function é disparada após um período de tempo, os recursos para hospedar a função podem precisar ser criados ou reiniciados.

- **Atualizações e migrações de dados** - Uma vantagem de um código serverless é que você pode liberar novas funções sem precisar reimplantar todo o aplicativo. Essa vantagem pode se tornar uma desvantagem quando há um banco de dados relacional envolvido. As alterações nos esquemas de banco de dados são difíceis de sincronizar com atualizações serverless.

- **Dimensionamento** - É um equívoco comum acher que serverless significa a "ausência de servidor". É, na verdade, "menos servidor". A maioria das plataformas sererless fornecem um conjunto de controles para manipular como a infraestrutura deve ser dimensionada quando a densidade do evento aumenta. Portanto, é necessário organizar e planejar quais funções são hospedadas juntas com a base nos requisitos de escala.

- **Monitoramento** - Um aspecto frequentemente ignorado do DevOps é o monitoramento de aplicativos depois da implantação. É importante ter uma estratégia para monitorar serverless functions. O maior desafio é geralmente correlação ou reconhecimento quando um usuário chama várias funções como parte da mesma interação. A maioria das plataformas serverless permitem o log de console que pode ser importado para ferramentas de terceiros. Também há opções para automatizar a coleta de telemetria, gerar e acompanhar IDs de correlação e monitorar ações específicas para fornecer informações detalhadas. O Azure fornece a plataforma de Application insights avançada para monitoramento e análise.

- **Dependências** - Uma arquitetura serverless pode incluir funções que dependem de outras funções, e não é incomum que uma arquitetura sem servidor tenha vários serviços chamados entre si como parte de uma transação

distribuída ou de interação. Para evitar um forte acoplamento, é recomendável que os serviços não referenciem uns aos outros diretamente.

- **Gerenciamento de falhas** - Também é importante considerar o padrão de Circuit Breaker. Se, por algum motivo, um serviço continuar a falhar, não é aconselhável chamar esse serviço repetidamente. Em vez disso, um serviço alternativo é chamado ou uma mensagem retornada até que a integridade do serviço dependente seja restabelecida. A arquitetura sem servidor precisa levar em conta a estratégia para resolver e gerenciar dependências entre serviços.

- **Implantações blue/green** - Para que as atualizações sejam bem-sucedidas, as funções devem ter controle de versão, para que os serviços que as chamam sejam roteados para a versão correta do código. Uma estratégia para implantar novas versões também é importante. Uma abordagem comum é usar "blue-green deployment". A implantação green é a função atual. Uma nova versão "blue" é implantada para produção e testada. Quando o teste passa, as versões green e blue são trocadas para que a nova versão venha ao vivo. Se forem encontrados problemas, eles poderão ser trocados novamente. O suporte à versão e aos deployments blue/green requerem uma combinação entre criar funções para acomodar alterações de versão e trabalhar com a plataforma sem servidor para lidar com as implantações.

Capítulo 6. Arquitetura Cloud Native

Arquiteturas Cloud Native capacitam as organizações para criar e executar aplicativos escalonáveis em ambientes modernos e dinâmicos, como nuvens públicas, privadas e híbridas. Contêineres, services meshes, microservices, immutable infraestruture e APIs declarativas exemplificam essa abordagem.

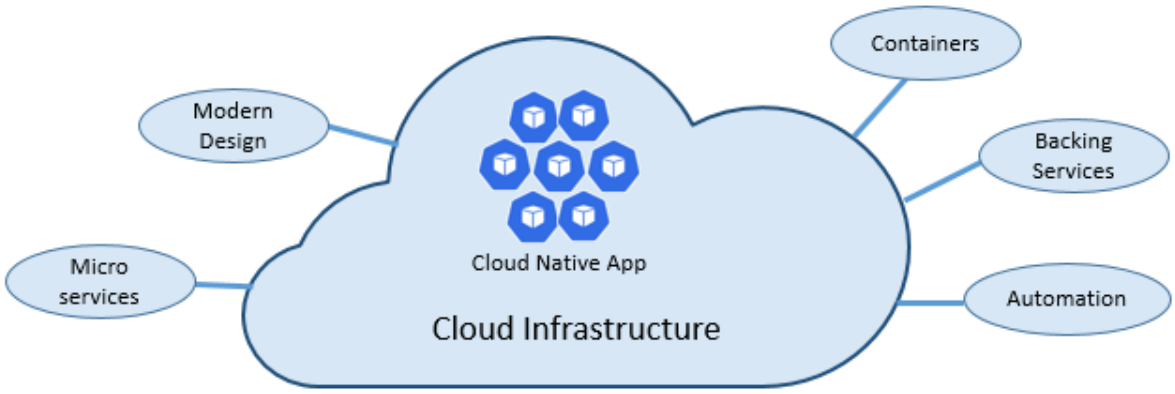
Essas técnicas permitem sistemas menos rígidos, resilientes, gerenciáveis e observáveis. Combinada com a automação robusta, elas permitem que os engenheiros façam alterações de alto impacto com frequência e previsibilidade.

Falar sobre Cloud Native é falar sobre *velocidade* e *agilidade*. Os sistemas de negócios estão evoluindo para permitir que os recursos de negócios sejam armas de transformação estratégica que aceleram a velocidade e o crescimento dos negócios. Empresas como Netlix, Uber e WeChat implementam estas técnicas.

Esse estilo de arquitetura permite que estas empresas respondam rapidamente às condições do mercado. Eles podem atualizar de forma instantânea áreas pequenas de um aplicativo dinâmico e complexo, além de dimensionar individualmente essas áreas conforme necessário.

A velocidade e a agilidade da nuvem nativa vem de vários fatores. A maioria é a infraestrutura de nuvem. Cinco pilares básicos adicionais são mostrados na Figura 32:

Figura 32 – pilares básicos de uma arquitetura Cloud Native.



Fonte: https://docs.microsoft.com.

- **Design Moderno**: utilizar referências como o The Twelve-factor app ou o livro Beyond the twelve-factor app é um ótimo direcionador para definir designs robustos. Além disso, outrs pontos importantes no design devem ser consierados:

  – Comunicação: Como serão feitas as comunicações entre suas aplicações e serviços? Quais os protocolos serão utilizados para tráfego das informações? Serão utilizados eventos? Todas essas perguntas devem ser respondidas na definição do seu modelo arquitetural.

  – Resiliência: Uma arquitetura de microsserviços move o sistema de dentro do processo para comunicação de rede fora do processo. Em uma arquitetura distribuída, o que acontece quando o serviço B não está respondendo a uma chamada de rede do serviço A? Ou então, o que acontece quando o serviço C fica temporariamente indisponível e outros serviços que o chamam são bloqueados?

  – Dados distribuídos: Por design, cada microsserviço encapsula seus próprios dados, expondo as operações por meio de sua interface pública. Em caso afirmativo, como você consulta dados ou implementa uma transação em vários serviços?

  – Identidade : Como seu serviço identificará quem está acessando e quais permissões eles têm?

- **Microsserviços**: os sistemas nativos de nuvem adotam os microsserviços, um estilo de arquitetura popular para construir aplicativos modernos.

Criado como um conjunto distribuído de serviços pequenos e independentes que interagem por meio de uma malha compartilhada, os microsserviços compartilham as seguintes características:

  – Cada um implementa um recurso comercial específico dentro de um contexto de domínio maior.

  - Cada um é desenvolvido de forma autônoma e pode ser implantado independentemente.

  - Cada um é um encapsulamento independente de sua própria tecnologia de armazenamento de dados (SQL, NoSQL) e plataforma de programação.

  - Cada é executado em seu próprio processo, e se comunica com outras pessoas usando protocolos de comunicação padrão, como HTTP/HTTPS, WebSockets ou AMQP.

  - Eles compõem juntos para formar um aplicativo.

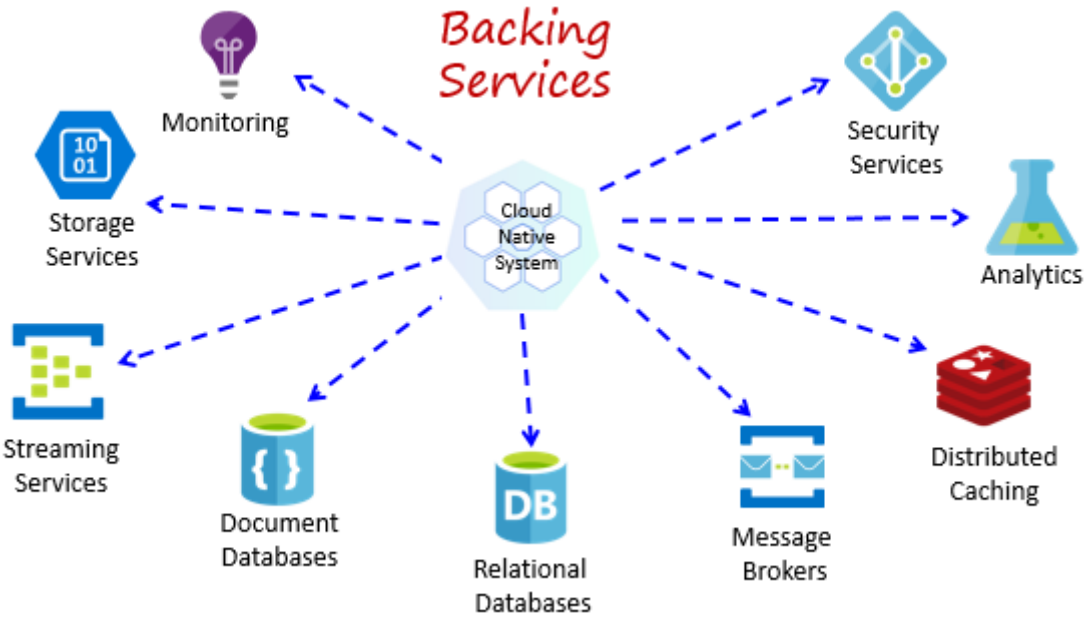
- **Containers:** hoje em dia, é natural ouvir o ***contêiner*** de termos mencionado em qualquer conversa relacionada à ***Cloud Native***..

O contêiner de um microsserviço é simples e direto. O código, suas dependências e tempo de execução são empacotados em um binário chamado container image. As imagens são armazenadas em um container registry, que atua como um repositório ou uma biblioteca para imagens. Um registry pode estar localizado em seu computador, em seu data center ou em uma nuvem pública. O Docker em si mantém um registro público por meio do Docker Hub.

Quando necessário, você transforma a imagem em um container (imagem em execução). A instância é executada em qualquer computador que tenha uma engine do Docker instalada. Você pode ter tantas instâncias do serviço em contêineres, conforme necessário.

- **Serviços de backup** - Os sistemas Cloud Native dependem de vários recursos auxiliares diferentes, como armazenamentos de dados, agentes de mensagens, monitoramento e serviços de identidade. Esses serviços são conhecidos como serviços de backup.

Figura 33 – Backing Services.



Fonte: https://docs.microsoft.com.

Estes serviços atendem aos princípios 3, 4, e 6 descritos em The Twelve-Factor app.

Com esse padrão, um serviço de backup pode ser anexado e desanexado sem alterações de código. Você pode promover um microsserviço de QA para um ambiente de preparo, atualizando a configuração de microsserviço para apontar para os serviços de backup em preparo e injetar as configurações em seu contêiner por meio de uma variável de ambiente.

Automação – Arquiteturas Cloud native adotam microsserviços, contêineres e design de sistema moderno para atingir velocidade e agilidade. No entanto, isso é apenas parte da história. Como provisionar os ambientes de nuvem nos quais esses sistemas são executados? Como você implanta rapidamente recursos e atualizações do aplicativo? Como você arredonda o panorama completo?

Com o IaC (Infraestruture as code), você automatiza o provisionamento de plataforma e a implantação de aplicativos. Essencialmente, você aplica práticas de engenharia de software, como teste e controle de versão, às suas práticas de

DevOps. Sua infraestrutura e implantações são automatizadas, consistentes e reproduzíveis.

A arquitetura Cloud Native vai muito além do desenvolvimento de soluções disponíveis em “várias clouds”. De acordo com as várias estratégias que estudamos nesta disciplina, a arquitetura de nuvem passa a ser uma composição de várias outras soluções arquiteturais, porém elevando o âmbito de tomadas de decisões, direcionadas por boas práticas e principalmente por dados.

Em alguns momentos, a escolha de uma estratégia Cloud Native diz respeito a utilizar o melhor recurso disponível em vários providers, às vezes por questões relacionadas a custos, outras por recursos necessários. E é importante avaliar as estratégias de uso de um provider direcionado por tecnologias para não haver os famosos Lock-in.

Ao mesmo tempo que possuímos aceleradores, que são recursos que facilitam uma determinada abordagem de resolução de problemas em um contexto, o Lock-in diz respeito à dependências sobre plataformas ou tecnologias. Este tipo de dependência pode causar grandes impactos em seu modelo arquitetural, dificultando alguns fatores, como os apresentados no item **Automação.**

Documentação

A documentação é uma ferramenta que faz parte de qualquer definição arquitetural. Um padrão muito interessante é o C4 model (https://c4model.com/), onde temos 4 visões, níveis de detalhes sobre o nosso desenho arquitetural.

Este modelo de documentação está divido em:

- Level 1 System Context: apresenta o contexto do Software de nível mais alto.
- Level 2 Container: É um zoom no primeiro nível, onde são apresentados mais detalhes e alguns componentes de software.

- Level 3 Componente: É um detalhe em alguma parte do diagrama de nível 2 e na apresentação dos componentes que são parte do contexto de interação.

- Level 4 Code – Pode ser utilizado para especificar, com mais detalhes, itens da arquitetura, utilizando, por exmeplo, UML.

Este padrão de documentação auxilia na apresentação de uma arquitetura e seus níveis, e neste padrão é possível até a utilização de documentação com código.

Referências

MICROSOFT. *Microsoft Docs*. 2020. Disponível em: <https://docs.microsoft.com>. Acesso em: 28 Out. 2020.

GRAPHQL. *Home*. 2020. Disponível em: <https://graphql.org/>. Acesso em: 28 Out. 2020.

C4 MODEL. *Home.* 2020. Disponível em: <https://graphql.org/>. Acesso em: 28 Out. 2020.

MICROSOFT. *Architecture Guid*. 2020. Disponível em <https://dotnet.microsoft.com/learn/dotnet/architecture-guides>. Acesso em: 28 Out. 2020.