Principais Arquiteturas de Software do mercado

Bootcamp Arquiteto de Software

Albert Tanure

2021

Principais Arquiteturas de Software do Mercado

Bootcamp Arquiteto de Software

Albert Tanure

Sumário

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 trabalham com diversos conceitos e todo este apanhado de necessidades faz com que o Arquiteto de Software se mantenha em estudo constante. 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. No entanto, ter uma boa base de fundamentos será o seu maior trunfo. Para além disso, como é arquitetar uma solução? Quais são os primeiros passos? Quais são as habilidades necessárias?

- Antes da tecnologia, nós atuamos com pessoas. Nesse 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 a seu favor:
  - Pessoas:
    - Esse é um dos pontos mais importantes. Todos os projetos têm, 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 arquiteto ouve mais, serve as pessoas.
    - Obtenha feedbacks constantes sobre as soluções propostas. Tenha empatia e busque entender se os artefatos que foram gerados, sejam quais 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 ótimas tecnologias que surgem todos os dias. Não dá para usar tudo a todo momento, 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 conquistar. A complexidade é feita de muitas simplicidades.

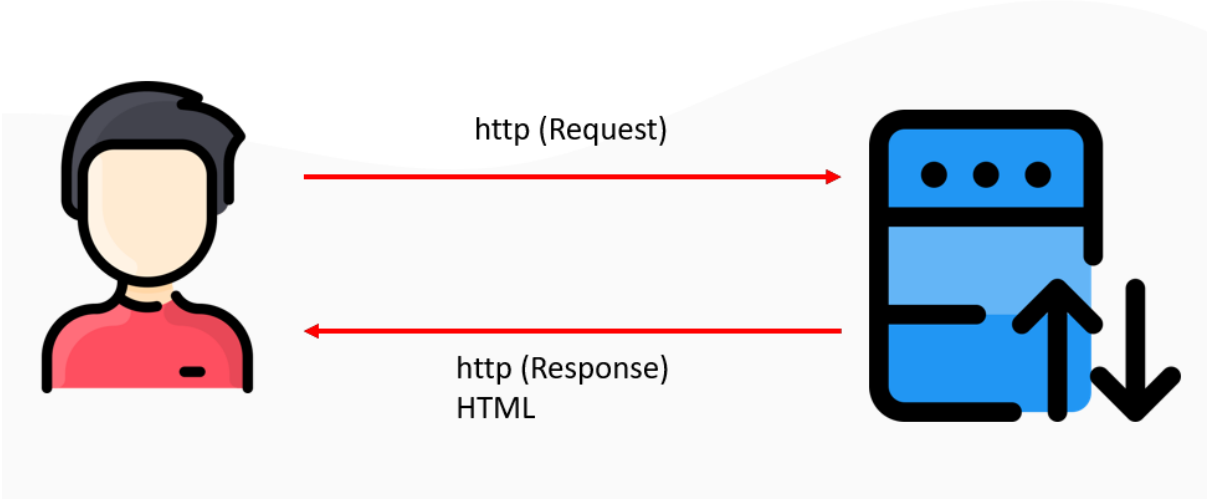
– Papel e lápis:

- Essas 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 estão presentes no dia a dia das pessoas, seja um website ou algum sistema com cadastro. A maioria dessas aplicações estão concentradas em um contexto único e publicadas em um web server. Dessa forma, o usuário de uma aplicação atua com uma requisição (request) para esses servidores, que retornam uma resposta (response).

Figura 1 – Comunicação Cliente-Servidor.



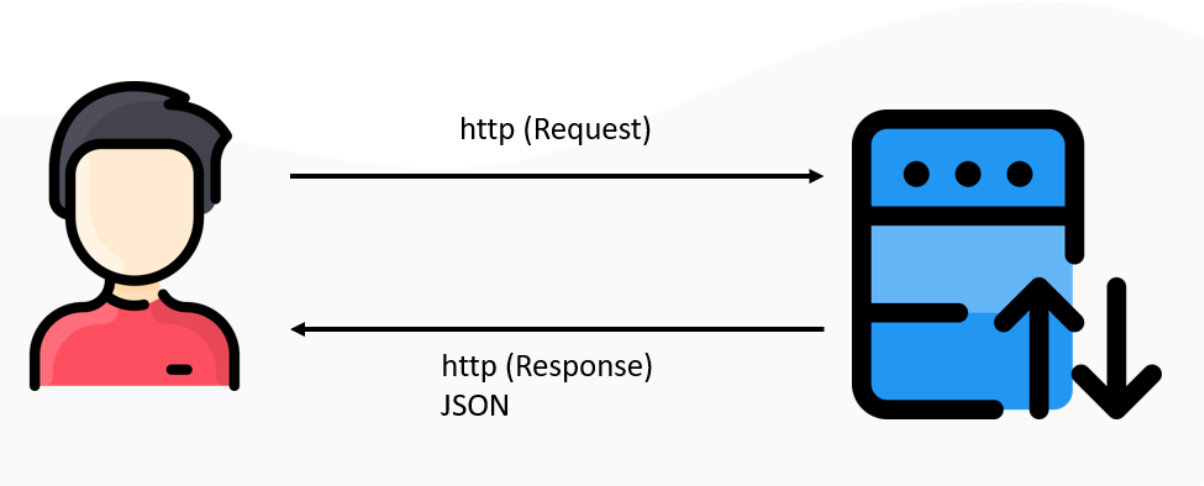
Como apresentado na Figura 1, todas as respostas das requisições nesse contexto retornam um documento HTML válido que será exibido pelo navegador.

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 desse modelo arquitetural, apesar de ser muito poderoso e funcional, pode criar algumas dificuldades principalmente para o lado dos usuários, já que não oferece 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.

Nesse caso, existe um modelo arquitetural denominado Single Page Application (SAP). Essa 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 nesse quadro é que não há a necessidade de tráfego de todo documento HTML, apenas dos 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 que possuem vários mecanismos, componentes e implementações de padrões que são extensíveis, o que traz 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 esses 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 utiliza conceitos como tipos, interfaces, classes, entre outros. Essa tecnologia não é uma exclusividade do Angular, podendo ser adotada em outras soluções.

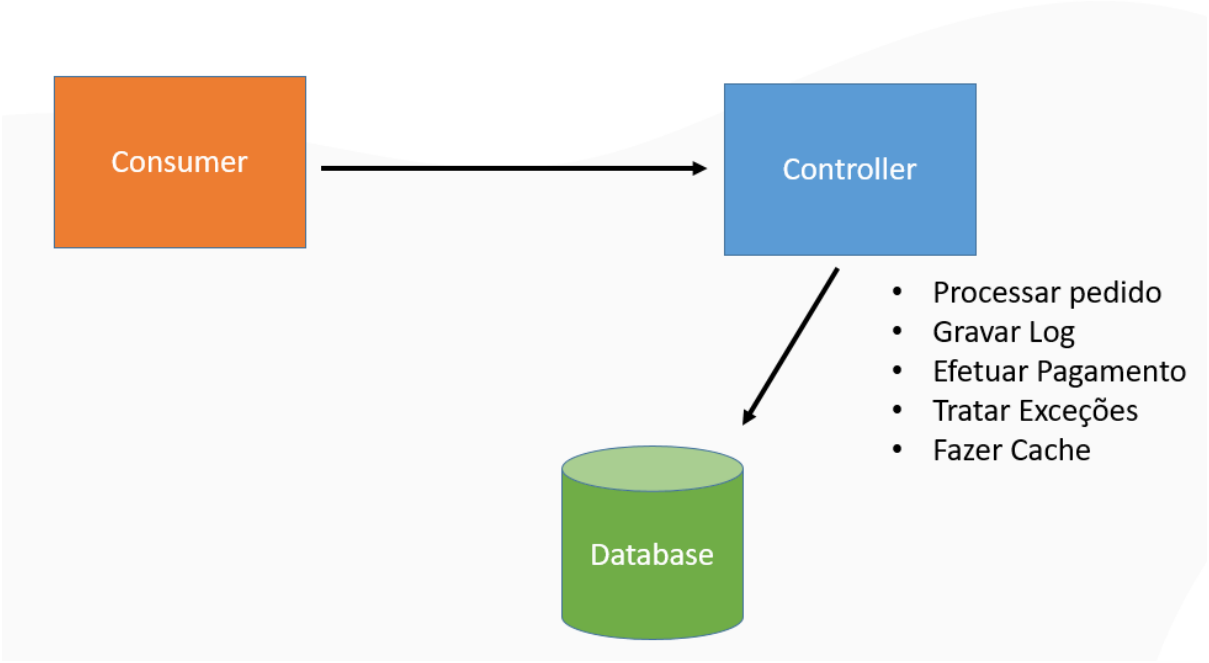
Outras técnicas muito interessantes que se tornaram possíveis com a adoção de uma solução SPA são: a utilização de teste de unidade para componentes, a separação em camadas, a utilização de containers, Continuous Integration e Continuous Deployment, entre outros princípios arquiteturais e técnicas.

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 esses objetivos, a orientação baseada em fundamentos e princípios é fundamental.

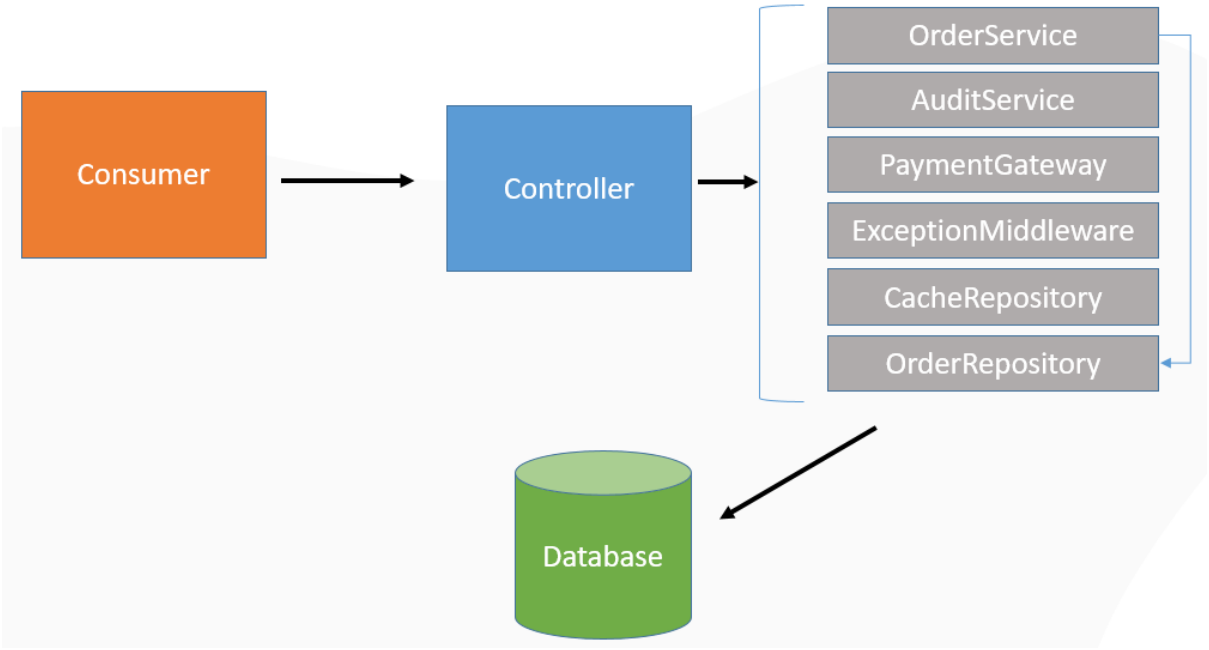
A Figura 3 representa um contexto de uma classe que possui mais de uma responsabilidade.

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 essa funcionalidade. Agora, observe a Figura 4:

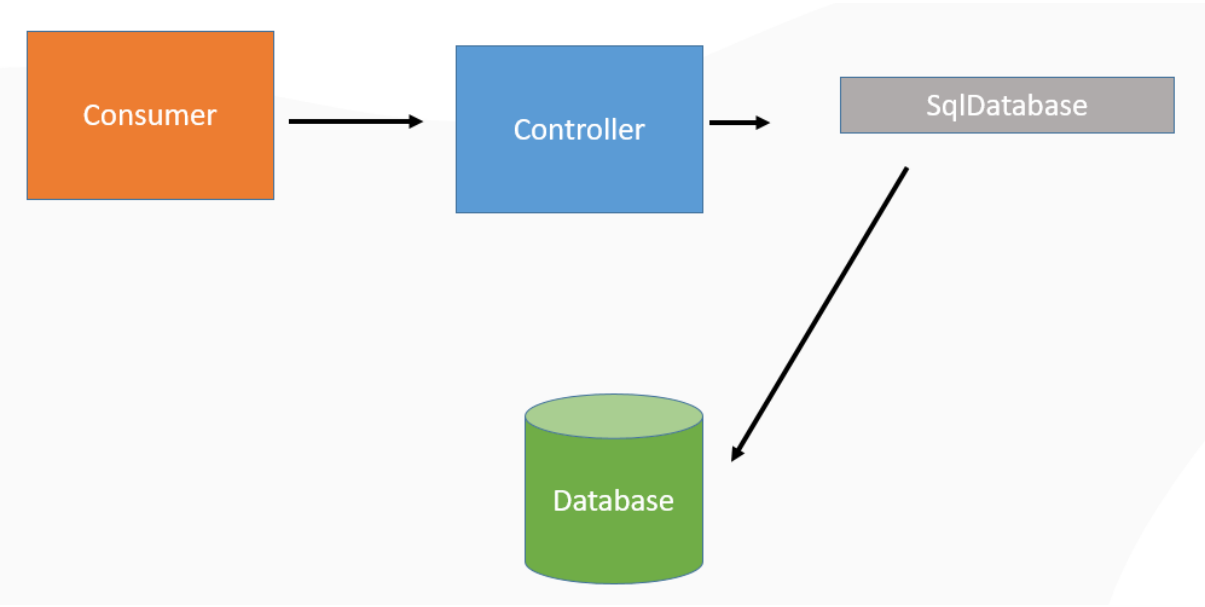
Figura 4 – Separação de responsabilidades.



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

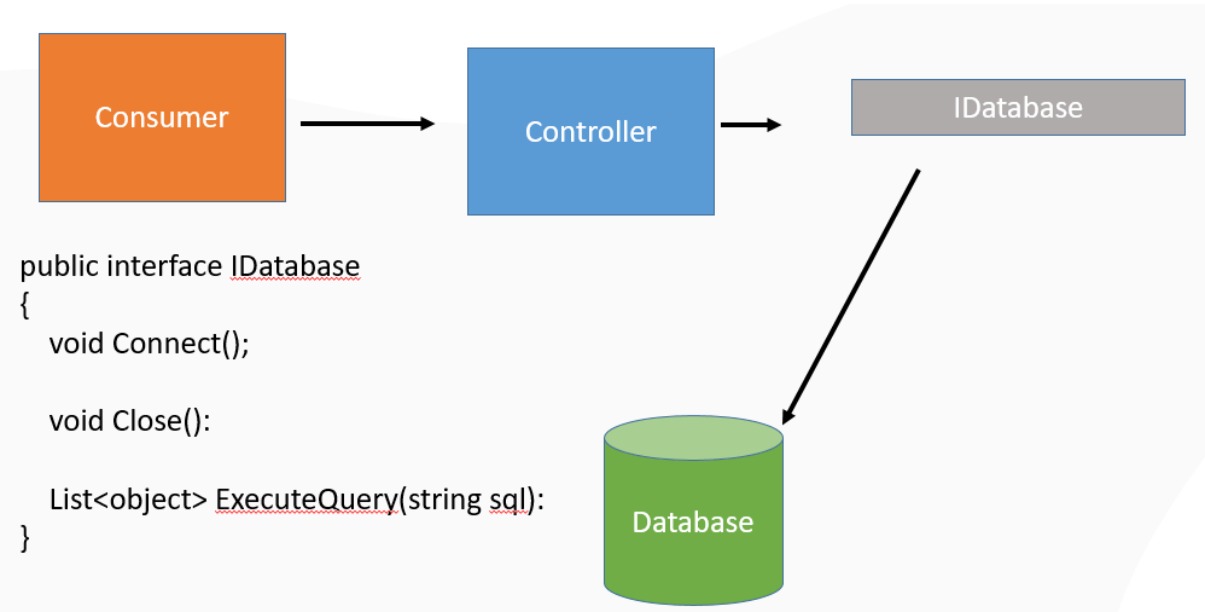
Seguindo o exemplo anterior, além da separação de responsabilidades, o ideal é que a lógica de negócio, implementações de mecanismos arquiteturais ou até a mesmo a implementação de mecanismos concretos baseados em interfaces (contratos) residam em projetos separados com a correta dependência entre eles. Esse modelo de desenvolvimento proporciona mais qualidade no código, além de garantir a *testabilidade* e rápidas evoluções. Para ilustrar esse exemplo, observe a seguinte figura:

Figura 5 – Dependência de implementação.



A Figura 5 apresenta um contexto no qual a classe *Controller* depende diretamente da implementação da classe *SqlDatabase*, dificultando a manutenção, a evolução e os testes na aplicação. Da mesma forma, esse 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

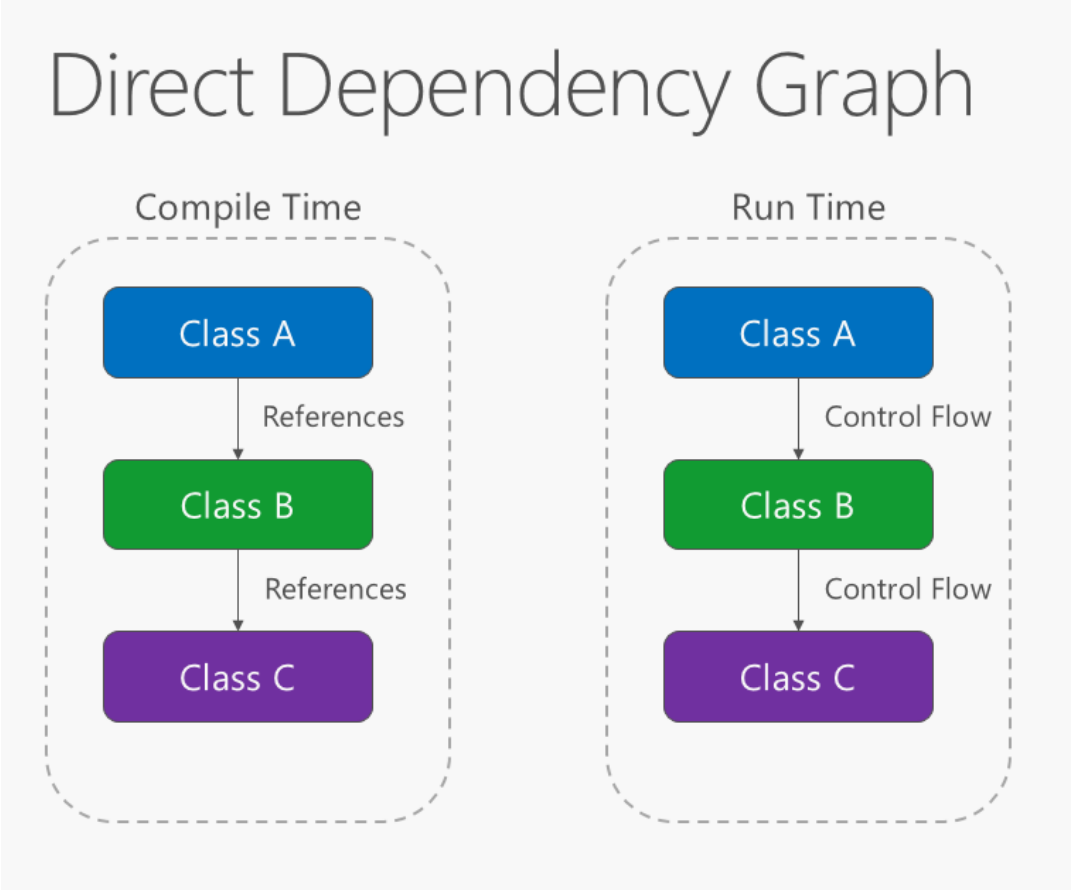
O princípio do **encapsulamento** tem como objetivo o isolamento de partes de uma aplicação. Basicamente, as camadas e seus componentes devem ser capazes de serem alterados sem depender dos seus colaboradores, desde que contratos externos não sejam violados.

O encapsulamento é um princípio da Orientação a Objetos. Ao definir uma classe, por exemplo, os modificadores de acesso encapsulam o estado de um objeto, mantendo coerente e limitando o acesso a detalhes de implementação ou integridade do objeto por parte dos colaboradores da classe. Por essa mesma razão, as aplicações e suas camadas devem abstrair a complexidade através de contratos, encapsulando seu estado e mantendo coerente o seu contexto negocial. Essa interação, através das trocas de mensagens entre os mecanismos baseados em contratos, possibilita que o aplicativo tenha rápida evolução.

Inversão de dependência

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

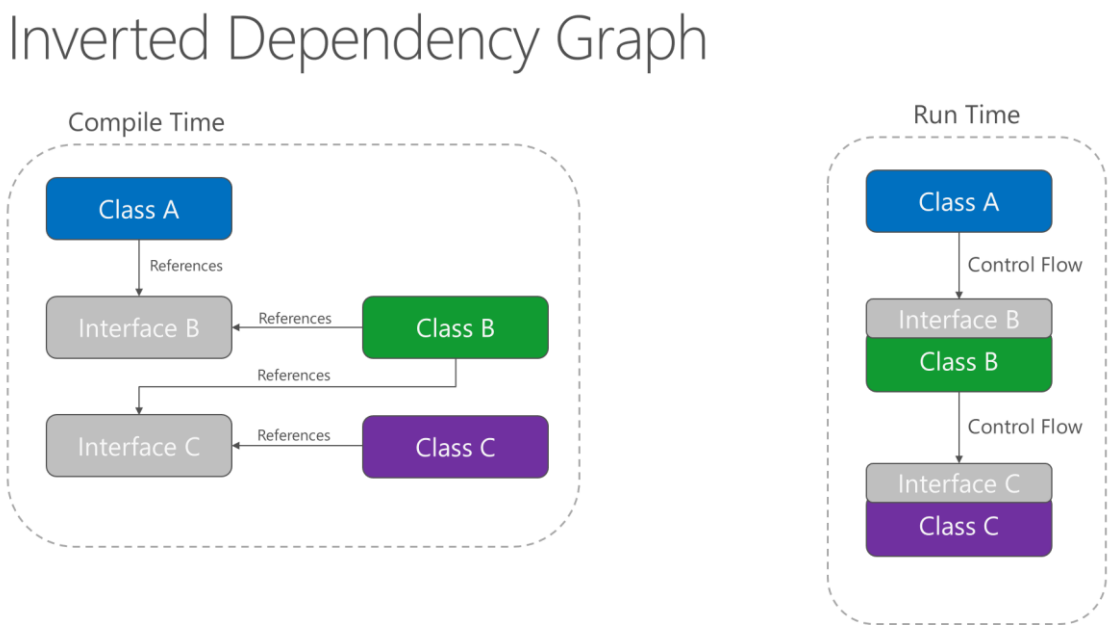
Figura 7 – Dependência de implementação.



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

A classe A depende de B, que depende de C. Esse 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, garantido maior *testabilidade*. Essa 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. Não devemos delegar o correto funcionamento dos nossos mecanismos delegando a responsabilidade de uso para os consumidores.

Por exemplo, se ao definir uma classe de repositório para interagir com o banco de dados, não definirmos uma dependência explícita de um drive de acesso ao banco ou a algum mecanismo como um ORM, delegamos a responsabilidade de funcionamento desse objeto para o seu consumidor.

Assim, criar uma instância dessa classe sem definir suas dependências pode ocasionar erros, que muitas vezes, são descobertos em *runtime.* Se o seu modelo de

classe se propõe a resolver um mecanismo, deve encapsular as funcionalidades e expor suas dependências como regra para uso correto desse objeto. Isso é uma definição de objetos coerentes.

Don't repeat yourself (DRY)

É comum 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. Havendo alguma alteração no fluxo negocial e de requisitos, não conformidades podem ser geradas, ainda mais se a cobertura de testes não abrange todo o contexto. Sendo assim, essa propagação de código repetidos podem gerar grande retrabalho.

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

No princípio da orientação a objetos, a responsabilidade única é parte do modelo de design em que um objeto deve ser coerente em sua implementação e possuir exclusivamente 1 objetivo. É a granularização de objetos com suas devidas responsabilidades que fornecerá um contexto rico de iteração entre os objetos e sua composição, além de que proporcionará mecanismos que funcionam em conjunto. Isso facilitará, também, a manutenabilidade e a testabilidade.

Imagine a abstração de um motor de um veículo. O motor é uma composição de vários componentes para o seu correto funcionamento. Contudo, tendo como exemplo uma **vela** como peça do motor, sua responsabilidade é, basicamente, gerar uma faísca para a combustão. A vela não precisa saber se a combustão ocorreu ou não. Sua responsabilidade é de responder a um estímulo de um outro mecanismo, onde ele executará sua responsabilidade.

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

Cada proposta de design tem seus prós e contras. Não existe a melhor ou a certa ou a errada; existe aquela 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 uma aplicação desse tipo precisar ser dimensionada horizontalmente, em geral, o aplicativo inteiro será duplicado em vários servidores ou máquinas virtuais.

O que são layers?

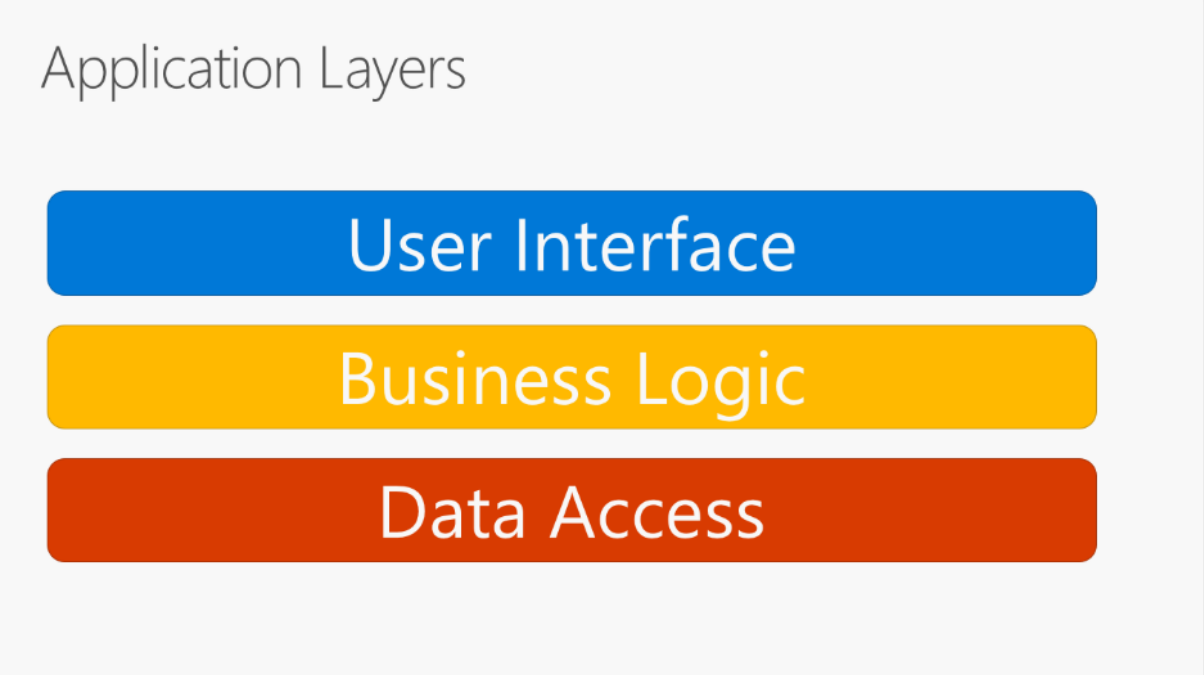
Conforme a complexidade dos aplicativos aumenta, uma maneira de gerenciá-lo é dividindo-o de acordo com suas responsabilidades ou interesses. Isso ajuda a manter uma base de código organizada para que os desenvolvedores possam encontrar facilmente onde determinadas funcionalidades são implementadas. Ainda, 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 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, 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.



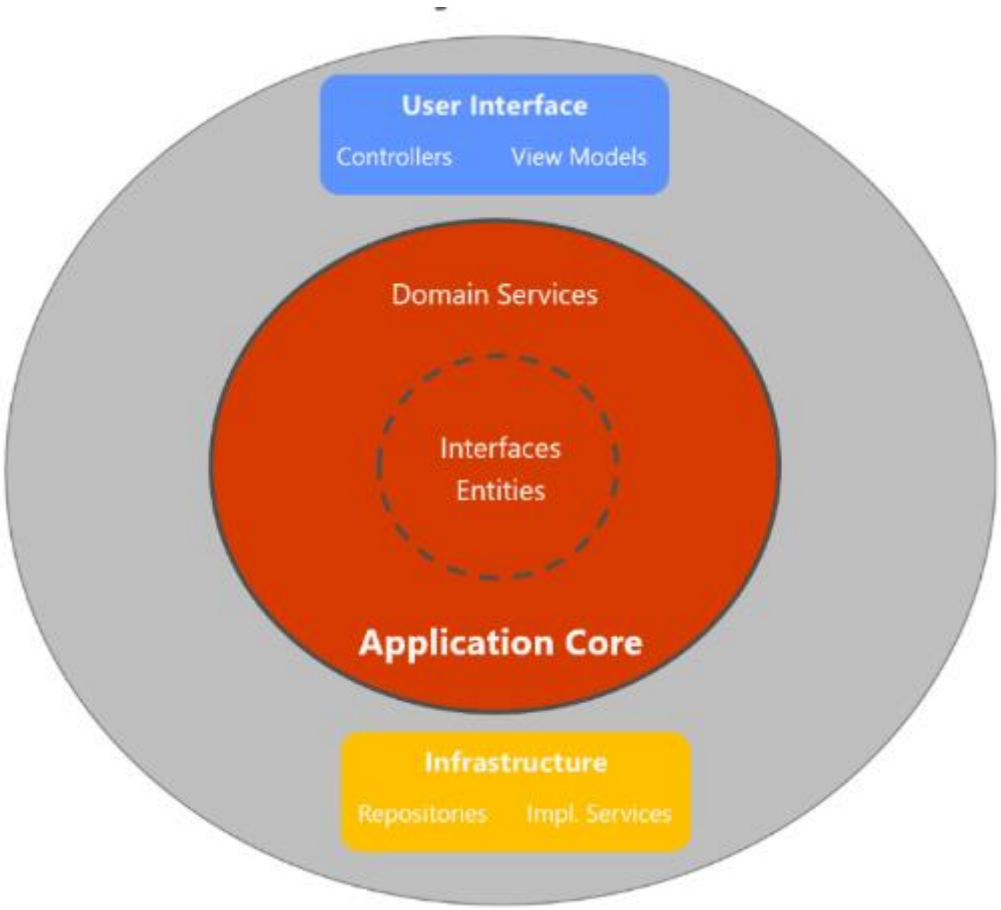
Fonte: https://docs.microsoft.com.

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 foi Arquitetura Hexagonal, seguido por Ports e Adapters, mas atualmente é conhecida como Clean Architecture ou Onion Architecture.

Nessa 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.

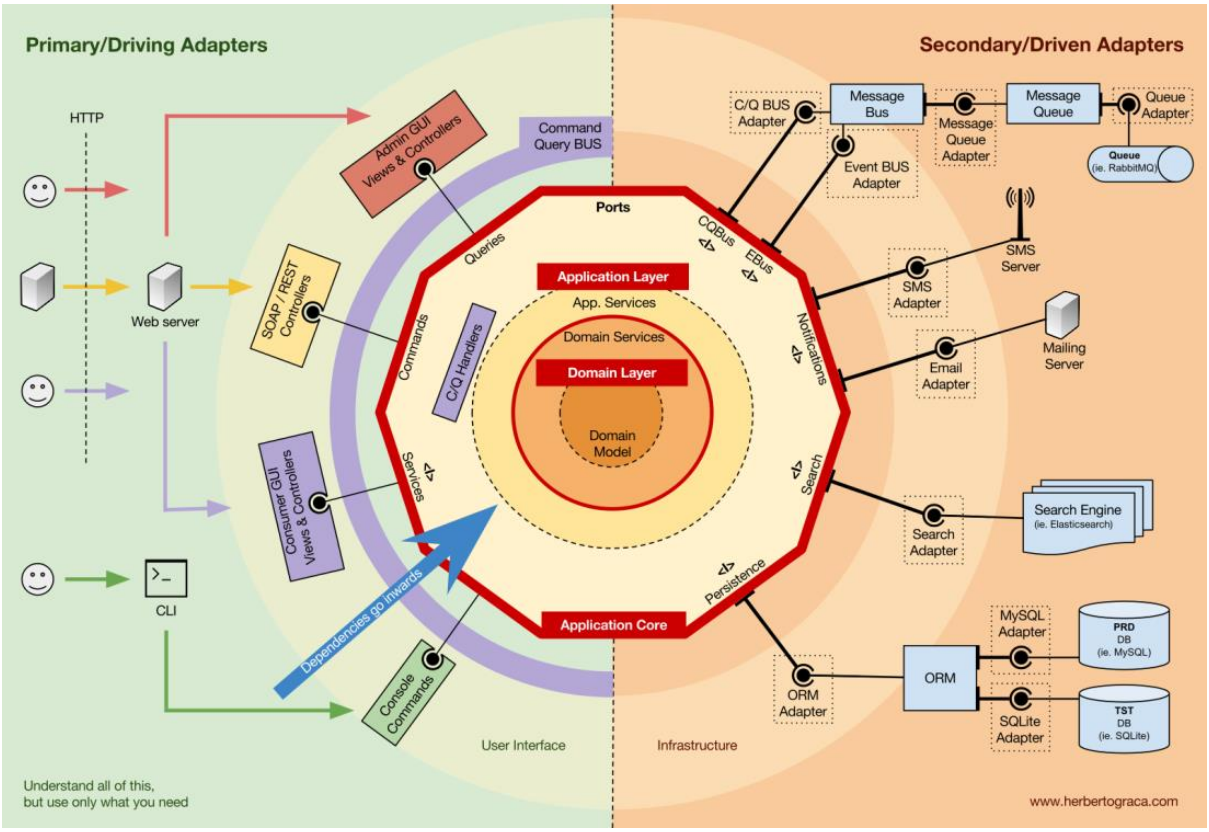


Fonte: https://docs.microsoft.com.

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.

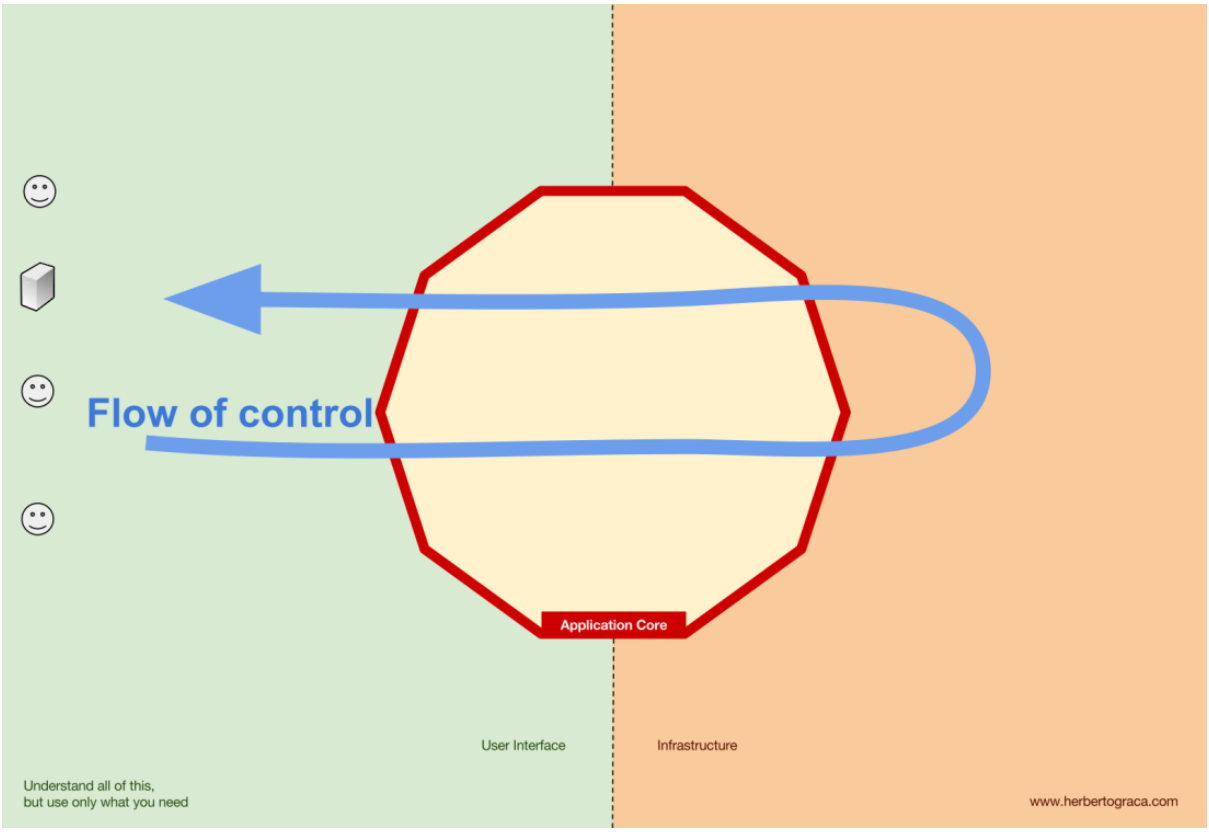


Fonte: https://herbertograca.com/2017/11/16/explicit-architecture-01-ddd-hexagonal-onion-clean-cqrs-how-i-put-it-all-together/.

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). Dessa forma, a camada inferior expõe interfaces de saída e de entrada, enquanto 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 fazendo 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 e traz uma enorme flexibilidade, mas não se aplica a 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; quando ela for 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 eles.

Essa 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, é enviado 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 e 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, o que significa que ele 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 ao 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 encaminhe-os 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.

Essa abordagem é uma boa opção para cargas de trabalho de IoT. É uma abordagem arquitetural poderosa, sendo altamente escalada e distribuída, além de proporcionar performance para a comunicação entre serviços. No entanto, possui desafios, como garantir a 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 clientes a nenhuma implementação específica. Por exemplo, um serviço Web REST poderia ser escrito em ASP.NET e aplicativos clientes podem usar qualquer linguagem ou o conjunto de ferramentas que possam gerar solicitações HTTP, além de analisar respostas HTTP.

APIs REST são projetadas para *recursos*, que se tratam de qualquer tipo de objeto, dados ou serviço que possam ser acessados 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. Nesse 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*. No entanto, 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 dessa 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 de 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. Além disso, 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 que direcione como deve ser definida esta rastreabilidade por parte da resposta dos recursos. No entanto, algumas APIS, como a do GitHub, utilizam parte desses 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":[]
    }]
}
```

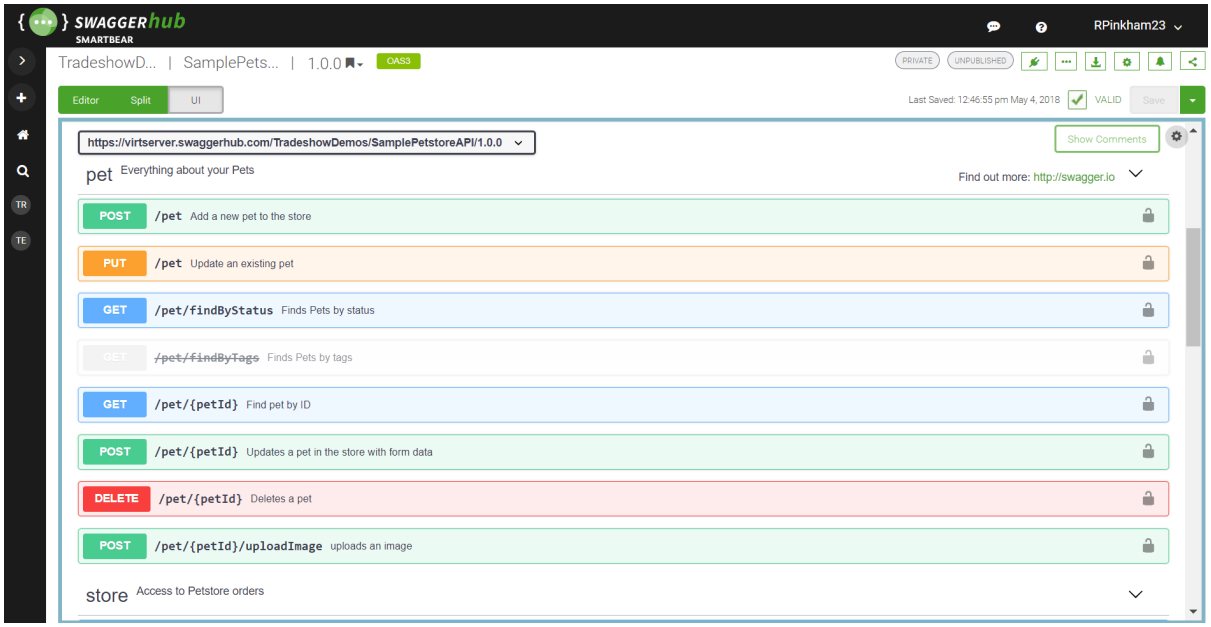
Fonte: https://docs.microsoft.com.

Documentação:

Apesar das técnicas disponibilizadas pela especificação REST para a disponibilização de recursos de forma facilitada, uma boa documentação é de extrema importância e isso não é 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, que possui uma interface simples e 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 e 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 independentes 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. Essa 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.

Esse arquivo especifica quais são as chamadas disponíveis (métodos), além dos tipos de entrada e saída. Ele 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#. Essa implementação pode variar de acordo com a tecnologia. Após definidos os contratos, suas implementações e as 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 adicional para fornecer suporte a esse 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, essa tecnologia é um *Query language* criada pelo Facebook. Portanto, não depende de um database ou de uma tecnologia específica.

Essa 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 a 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 nesse 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#. Porém, para a adaptação do modelo do GraphQL, utilizando essa 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.

Nesse 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 esse 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 práticas. 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 pode fazer grande diferença na tomada decisões e entregar valor mais rapidamente.

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

- Nativo.
- Cross-Platform.
- Híbrido.

Cada modelo possui suas especificações e seus prós e contras. 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 a performance. Aplicações nativas têm 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.

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. Nesse 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 essa tecnologia, porém uma nova linguagem vem ganhando força: o **Kotlin**.

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 obtê-la. 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

Essa 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 essa 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 à entrega de soluções para as duas principais plataformas de mercado, utilizando o mesmo code base. Nesse modelo, o desenvolvimento da lógica de negócio e das interfaces é reutilizado e "traduzido" para cada plataforma.

Um outro fator muito importante é a performance. Sendo assim, essas 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, ela 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 que é 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 aberto, assim como as outras três plataformas, utiliza Dart como linguagem de programação e também pode ser desenvolvida em qualquer sistema operacional. É uma tecnologia muito poderosa e 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 contexto, utilizar esta abordagem pode ser uma ótima estratégia para a entrega de valor.

Os aplicativos desenvolvidos nessa abordagem podem ser instalados através das Stores de cada plataforma e possuem acesso facilitado às APIs. Se baseiam em JavaScript, HTML e CSS, mas 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 percebendo cada vez mais a economia de custo ao se resolver problemas de implantação e melhorar as operações de produção e de DevOps usando os containers.

A Microsoft tem lançando inovações de container 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 container 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 containers, 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?

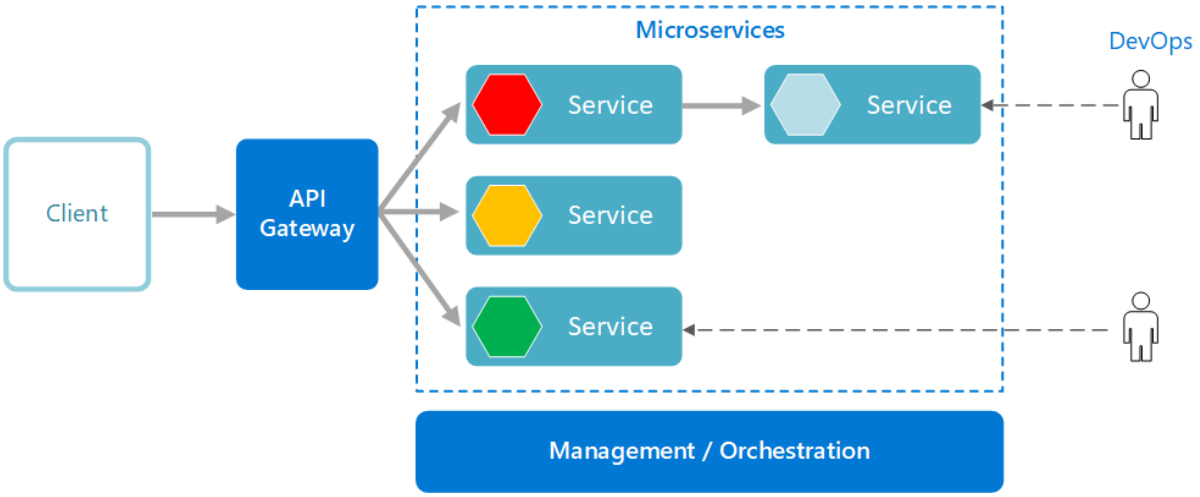
- Os microsserviços são um aplicação contextualizada para resolução de um contexto de domínio.
- Cada serviço é um projeto separado, gerido por uma equipe e focado em um objetivo.
- Os serviços podem ser implementados de forma independente.

- Possui seu próprio modelo de persistência.
- Os serviços comunicam entre si através de APIs bem definidas.
- Agnóstico a tecnologias.

Outros componentes presentes na estratégia de microsserviços, são:

- **Gestão/orquestração**: os serviços podem ser facilmente escalados. Geralmente, se baseando em uma estratégia de containers, o Kubernetes gerencia a necessidade de escala e também de acesso às instâncias de um mesmo serviço. Além disso, garante e auxilia no correto funcionamento de suas instâncias, fluxos de requisições e atualizações. Essa delegação para um orquestrador é de extrema importância para utilizar o benefício desse modelo arquitetural.
- **API Gateway**: o API gateway é a porta de entrada para os consumidores de serviços abstraindo ou encapsulando o backend e sua complexidade, além de possuir outros benefícios como segurança.

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, a independência de promoções aos ambientes produtivos, a escalabilidade, entre outros benefícios:

- **Agilidade:** a característica de contextualização de um microsserviço e sua independência de outros serviços fornece uma abordagem muito ágil de acordo com o contexto – como escalar em uma necessidade de alto nível de acesso; ou até mesmo na publicação de novas versões ou correções, já que o contexto de atualização está voltando apenas para o serviço em questão e não para toda a aplicação.

- **Equipes pequenas e direcionadas**: por possuir um contexto limitado de atuação, não há necessidade de ter grandes equipes para manter um microsserviço. Uma equipe poderá manter um ou mais microsserviços. Mas a complexidade é dividida entre os contextos, facilitando a manutenção, os testes, o monitoramento etc .

- **Base de código pequena**: a arquitetura é mais simples e direcionada para atender um determinado contexto de negócio. Consequentemente, serão menos mecanismos e códigos para gerir em um microsserviço em específico.

- **Misto de tecnologias**: as equipes podem escolher a tecnologia mais adequada ao respetivo serviço.

- **Isolamento de falhas**: a contextualização proposta por esse modelo arquitetural proporciona maior controle em diversos níveis. Nesse caso, utilizar padrões para aplicar resiliência ou também a implementação de Circuit-Breakeres é mais fácil e direcionada.

- **Escalabilidade**: os serviços podem ser facilmente escalados de acordo com a necessidade e a plataforma que provê o serviço.

- **Isolamento de dados**: cada serviço gerencia seu próprio modelo de dados, facilitando alguma atualização de schemas ou adição de tabelas, atributos etc.

Desafios

É claro que apensar da contextualização do “problema”, a abordagem de microsserviços também nos impõem vários desafios:

- **Complexidade**: gerir microsserviços é mais complexo do que uma aplicação monolítica onde tudo era contextualizado em um binário apenas. Dependendo do contexto de negócio, haverão muitos serviços a serem gerenciados.

- **Desenvolvimento e teste**: apesar de possuir um contexto mais delimitado, havendo a necessidade de efetuar testes de integração em alguma dependência do serviço, esse trabalho deve ser muito bem definido e elaborado, já que as interações podem ser muito além do que uma simples requisição. Nesse caso, vai envolver mecanismos de dados, mensageria ou até mesmo outros serviços.

- **Falta de governança**: é importante definir padrões de uso da arquitetura de microsserviços em seu contexto. Por serem agnósticos à tecnologia, a existência de vários padrões de implementações e também de monitoramento, diagnóstico, cache etc., podem gerar problemas de gerenciamento de seus contextos negociais.

- **Latência e congestionamento de rede**: muitos serviços sendo executados e comunicando constantemente podem gerar problemas de performance, tempos de respostas etc. Defina corretamente o modelo de comunicação para seus contextos, como por exemplo o uso de gRPC em determinados processos além de padrões de comunicação assíncronas.

- **Integridade de dados**: cada microsserviço tem seu próprio modelo de persistência que é abstraído totalmente de quaisquer aplicações. Assim, a

consistência dos dados pode ser um desafio. Adote consistência eventual quando possível.

- **Gerenciamento**: utilizar os benefícios da cultura DevOps é uma boa prática em tudo do cliclo de vida de um microsserviço, tanto para gerenciar sua manutenção e correção de bugs, quanto para seu monitoramento, diagnósticos e feedbacks constantes.

- **Controle de versão**: muitas vezes, os serviços são atualizados e novas versões são geradas. É muito importante controlar o versionamento de serviços para manter a compatibilidade com seus consumidores.

- **Conjunto de qualificações**: a arquitetura de microsserviços é uma abordagem para sistemas distribuídos, então um dos fatores para adoção é entender a capacidade do seu time para atuar em um cenário exponencialmente maior do que um contexto monolítico.

Boas práticas

- Modele os serviços contextualizados em seu negócio.

- Descentralizar tudo de cada serviço é um projeto a parte sob responsabilidade de uma equipe.

- O serviço que é o proprietário dos seus dados. Use o melhor armazenamento para o contexto desse serviço.

- Os serviços comunicam-se por meio de APIs ou mensagens. O encapsulamento é seu melhor amigo. Um serviço não deve conhecer detalhes de implementação out e responsabilidade sobre o funcionamento de outro serviço.

- Evite dependência entre serviços.

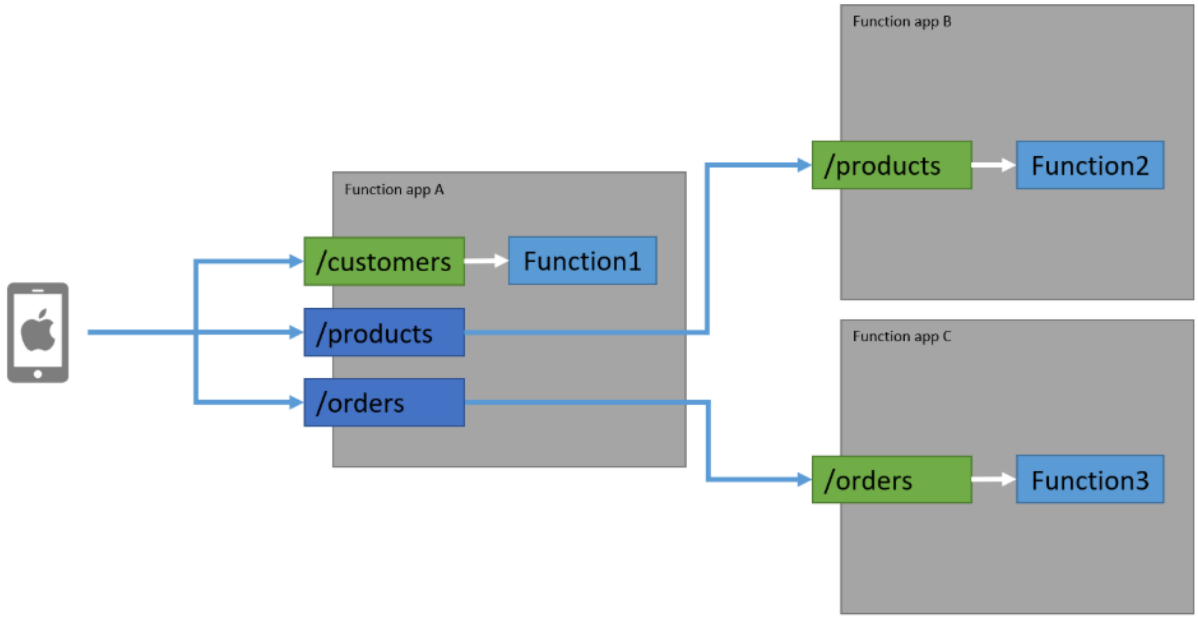
- APIs Gateways são apenas portas de entradas e roteadores.

- Isole falhas. Utilize padrões como resiliência e Circuit-breakers.

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.



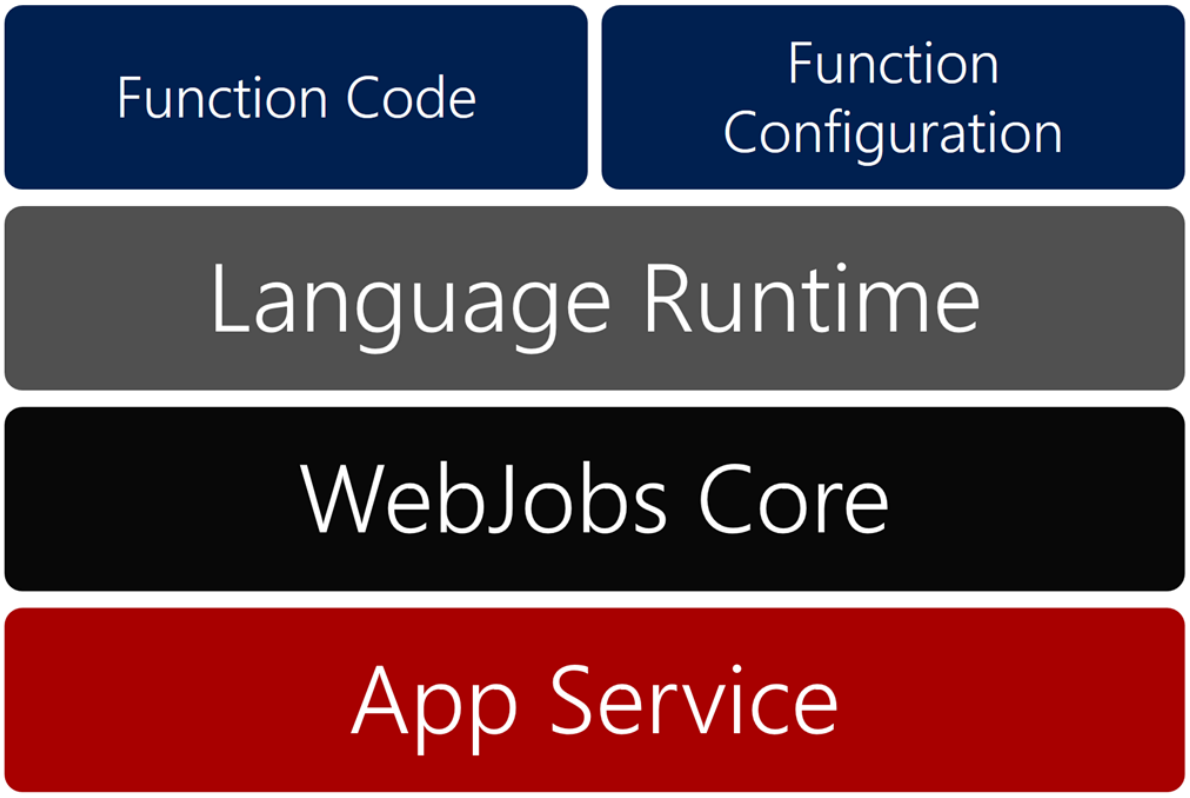
Font: https://docs.microsoft.com.

Capítulo 5. Arquitetura Serverless

De forma geral, a arquitetura serverless é uma ótima abordagem para vários contextos negociais. Apesar do termo **serverless**, não significa a ausência de servidores. Nesse caso, o foco é na funcionalidade e não na preocupação com a infraestrutura. Desse modo, as functions são basicamente métodos expostos como serviços que são consumidas ou utilizadas através de algum gatilho, evitando consumo desnecessário de recursos, mas ao mesmo tempo, provendo uma plataforma altamente escalável com base no nível de necessidade de uso para um recurso.

Em um modelo de cloud, os custos basicamente são relacionados por quantidade de requisições. As functions podem ser desenvolvidas por tecnologias como C#, Java, NodeJs, Powershell, dependendo do seu provider e da mesma forma como outros modelos arquiteturais, podem implementar as boas práticas como inversão de depdências, orientação a objetos etc.

Figura 30 – Camadas da arquitetura de Azure Functions.



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

Full Backend Serverless

O Full backend serverlerss é uma abordagem ideal para vários cenários de entrega de valor rapidamente, como a criação de novos recusos e a evolução de um sistema legado, bem como a criação um modelo de backend para aplicações mobile ou web. Assim como microsserviços se comunicam por vários protocolos, as functions também possuem cenários, que incluem:

- SaaS baseados em API (por exemplo: API de crédito pessoal).
- Mensageria (exemplo: solução de monitoramento de dispositivo).
- Integração entre serviços (exemplo: aplicativo de reservas de viagens).
- Background (exemplo: limpeza de banco de dados baseada em temporizador).
- Tradução ou adaptação de dados entre aplicações (exemplo: importação disparada por upload de arquivo).

É uma estratégia poderosa por prover rapidamente uma solução de backend e pode ser utilizada em vários cenários interessantes.

Aplicações Web

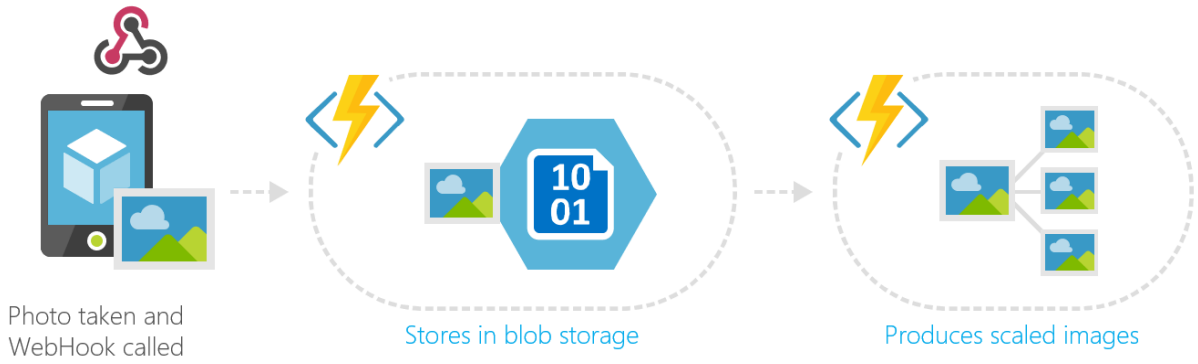
As aplicações Web são excelentes candidatas para o uso de Serverless. Na abordagem de desenvolvimento utilizando o modelo SPA (Single Page Application), a utilização de serverless pode ser uma ótima escolha, já que podem ser consumidos recursos através da aplicação Web pelo protocolo HTTP no padrão REST. Então, o backend poderá ser baseado em serverless, evitando desperdícios de recursos.

Backend para aplicações móveis

O paradigma event-riven de aplicações Serverless os torna ideal como backends para dispositivos móveis. Tirar proveito de um modelo serverles permite que os desenvolvedores foquem nos negócios e obtenham soluções rapidamente.

Por exemplo, um desenvolvedor móvel que utiliza a JavaScript (arquitetura híbrida) também pode criar funções serverless com JavaScript em NodeJs. Portanto, eles são capazes de usar habilidades existentes, como desenvolvedor, para criar a lógica de negócio.

Figura 31 – Serverless e aplicações móveis.



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

IOT

O IoT refere-se a objetos físicos que estão conectados, os “connected devices” ou “smart devices”. Tudo pode estar conectado e pode enviar informações através da internet. Existem muitas aplicações de IoT, como por exemplo para controlar se uma peça em um elevador precisa de alguma manutenção.

O enorme volume de dispositivos e informações geralmente determina uma arquitetura orientada pela utilização de serverless, que é ideal por vários motivos:

- Escalabilidade conforme a necessidade de consumo dos dados de dispositivos.

- Acomoda a adição de novos endpoints para dar suporte a novos dispositivos e sensores.
- Controle de versão independente.
- Resiliência e menos disperdício.

Considerações

A adoção de uma arquitetura serverless vem com determinados desafios, mas todos têm soluções. Ainda assim, dependendo das necessidades do seu contexto, sua escolha pode não ser a melhor.

- **Gerenciamento de estados**: as functions Serverless são stateless.
- **Durable process**: tempos de execução curtos facilitam para o provedor serverless liberar recursos. Geralmente, o tempo de execução é limitado a 10 minutos, dependendo do provider de nuvem. Se o processo levar mais tempo, você deve 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”.
- **Atualizações e migrações de dados**: uma vantagem de um código serverless é que você pode liberar novas funções sem precisar reimplantar todo o aplicativo.
- **Dimensionamento**: é um equívoco comum achar que serverless significa a “ausência de servidor”. É, na verdade, “menos servidor”. A maioria das plataformas fornecem controles para manipular a infraestrutura e seu dimensionamento quando há necessidade.

- **Monitoramento**: um aspecto frequentemente ignorado do DevOps é o monitoramento. É importante ter uma estratégia para monitorar serverless functions.

- **Dependências**: uma arquitetura serverless pode incluir funções que dependem de outras funções. Não é incomum. Mas evite esse acoplamento direto referenciando functions umas nas outras.

- **Gerenciamento de falhas**: também é importante considerar o padrão de Circuit Breaker para evitar uma carga excessiva no consumo de sua function quando não há possibilidades.

- **Implantações blue/green**: é uma boa prática controlar as versões das functions e também de suas publicações. Para isso, a utilização de estratégias de publicação como "blue-green deployment" são ótimas.

Capítulo 6. Arquitetura Cloud Native

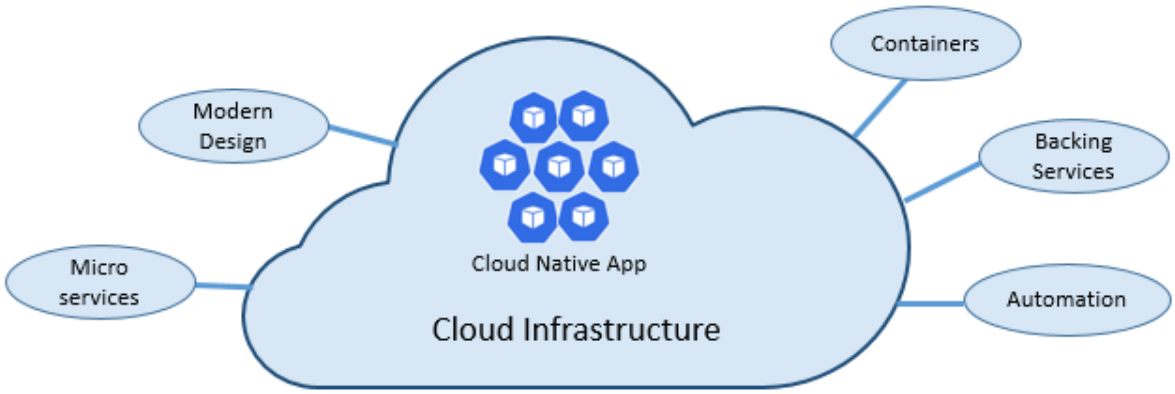
Arquiteturas Cloud Native capacitam as organizações para o desenvolvimento e o uso de soluções escaláveis em ambientes modernos, seja para nuvens públicas, privadas ou híbridas.

Falar sobre Cloud Native é falar sobre *velocidade* e *agilidade*. Os sistemas de negócios estão evoluindo para permitir que os recursos sejam parte de um processo de transformação estratégica, acelerando o crescimento dos negócios.

Esse estilo de arquitetura permite que as empresas respondam rapidamente às condições do mercado.

A velocidade e a agilidade da nuvem nativa vem de vários fatores. A maioria é a infraestrutura de nuvem. Cinco pilares básicos adicionais 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, outros pontos importantes no design devem ser consierados:
  - Comunicação: Como serão feitas as comunicações entre suas aplicações e serviços? Quais protocolos serão utilizados para tráfego

das informações? Serão utilizados eventos? Todas essas perguntas devem ser respondidas na definição do seu modelo arquitetural.

  - Resiliência: Como sua arquitetura responde a falhas?

  - Dados distribuídos: Arquiteturas baseadas em microsserviços possuem sua contextualização negocial. Por isso, em um cenário distribuído, como seria a estratégia para manter a integridade das informações ou quais seriam as estratégias de transação distribuída?

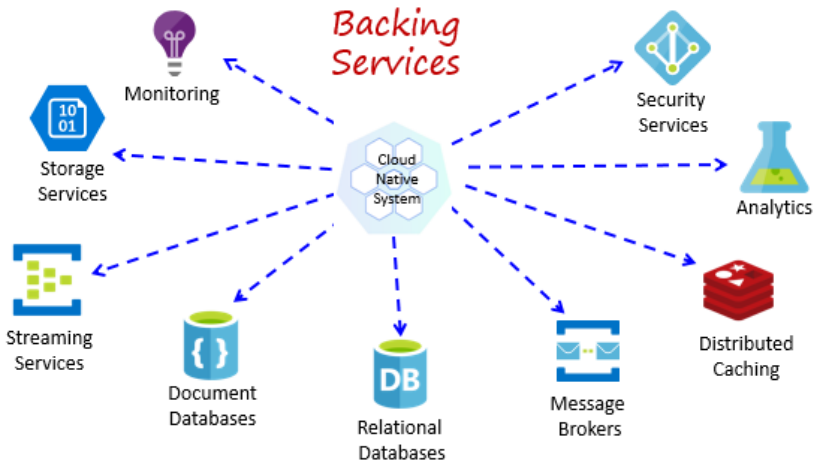
  - Identidade – Quem está consumindo os serviços? Qual o limite que deve ser imposta para o consumo do recurso?

- **Microservices**: os sistemas nativos de nuvem adotam os microsserviços para construir aplicativos modernos.

- **Containers**: estratégia Containers é um assunto intimamente relacionado à ***Cloud native***.

- **Serviços de backup**: os sistemas Cloud Native atuam com vários serviços diferentes, que auxiliam na composição de suas soluções.

Figura 33 – Backing Services.



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

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

Automação – O uso de recursos como IaC (Infraestruture as code) são imprescindíveis nesse modelo arquietural.

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 vezes 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 a dependências sobre plataformas ou tecnologias. Esse 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.

Esse 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 itens da arquitetura com mais detalhes, utilizando, por exmeplo, UML.

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

Referências

GRAÇA, Herberto. *DDD, Hexagonal, Onion, Clean, CQRS, … How I put it all together.* 16 nov. 2017. Disponível em: <https://herbertograca.com/2017/11/16/explicit-architecture-01-ddd-hexagonal-onion-clean-cqrs-how-i-put-it-all-together>. Acesso em: 3 mar. 2021.

MICROSOFT. *Microsoft Docs*. 2020. Disponível em: <https://docs.microsoft.com>. Acesso em: 3 mar. 2021.

GRAPHQL. *Home*. 2020. Disponível em: <https://graphql.org/>. Acesso em: 3 mar. 2021.

C4 MODEL. *Home.* 2020. Disponível em: <https://graphql.org/>. Acesso em: 3 mar. 2021.

MICROSOFT. *Architecture Guid*. 2020. Disponível em <https://dotnet.microsoft.com/learn/dotnet/architecture-guides>. Acesso em: 3 mar. 2021.