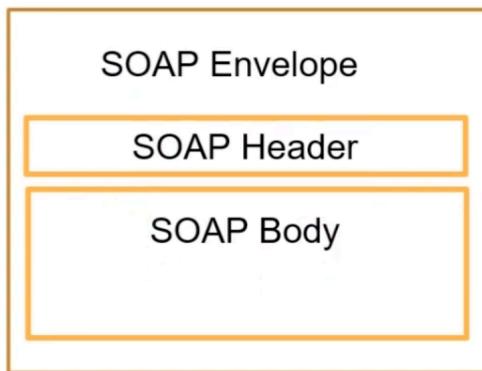


Serviços Web

Serviços Web ou Web Services são soluções para aplicações se comunicarem independente de linguagem, softwares e hardwares utilizados. Inicialmente os Serviços Web foram criados para troca de mensagens utilizando a linguagem XML (Extensible Markup Language) sobre o protocolo HTTP, sendo identificado por URI (Uniform Resource Identifier). Podemos dizer que Serviços Web são API's que se comunicam por meio de redes sobre o protocolo HTTP.

SOAP

SOAP (Simple Object Access Protocol) é um protocolo baseado em XML (ou Extensible Markup Language, que é uma linguagem de marcação criada na década de 90 pela W3C que facilita a separação de conteúdo e que não tem limitação de criação de tags) para acessar serviços web principalmente por HTTP. Pode-se dizer que SOAP é uma definição de como serviços web se comunicam. Foi desenvolvido para facilitar integrações entre aplicações. Permite integrações entre aplicações, independente de linguagem, pois usa como linguagem comum o XML, é independente de plataforma e software e é um meio de transporte genérico, ou seja, pode ser usado por outros protocolos além do HTTP. O "SOAP Message" possui uma estrutura única que deve sempre ser seguida.



- **SOAP Envelope** é o primeiro elemento do documento e é usado para encapsular toda a mensagem SOAP.
- **SOAP Header** é o elemento que possui informações de atributos e metadados da requisição.
- **SOAP Body** é o elemento que contém os detalhes da mensagem.

WSDL

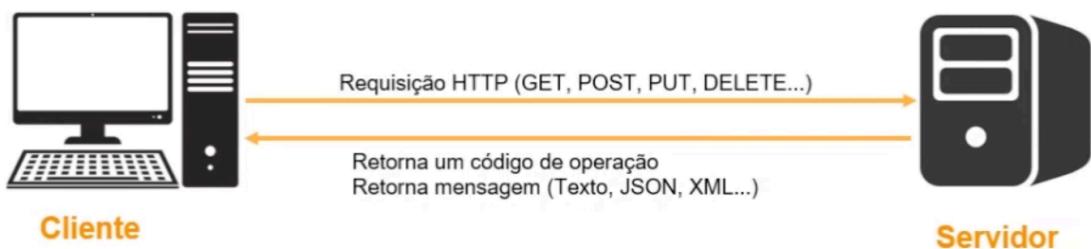
WSDL (Web Services Description Language) é usado para descrever Web Services, funciona como um contrato do serviço. A descrição é feita em um documento XML, onde é descrito o serviço, especificações de acesso, operações e métodos.

XSD

XSD (XML Schema Definition) é um schema no formato XML usado para definir a estrutura de dados que será validada no XML. O XSD funciona como uma documentação de como deve ser montado o SOAP Message (XML) que será enviado através de Web Service.

REST

REST (Representational State Transfer) é um estilo de arquitetura de software que define a implementação de um serviço web. Podem trabalhar com os formatos XML, JSON ou outros. Permite integrações entre aplicações e também entre cliente e servidor em páginas web e aplicações. Utiliza os métodos HTTP para definir a operação que está sendo efetuada e possui arquitetura de fácil compreensão.



API

Um API é um conjunto de rotinas e padrões de programação para acesso a um aplicativo de software ou plataforma baseado na Web. Refere-se ao termo em inglês "Application Programming Interface" que significa em tradução para o português "Interface de Programação de Aplicativos". É criada quando uma empresa de software tem a intenção de que outros criadores de software desenvolvam produtos associados ao seu serviço.

Métodos HTTP

- **GET** - Solicita a representação de um recurso.
- **POST** - Solicita a criação de um recurso.
- **DELETE** - Solicita a exclusão de um recurso.
- **PUT** - Solicita a atualização de um recurso.

JSON

JSON (JavaScript Object Notation) é caracterizado por ser uma formatação leve utilizada para troca de mensagens entre sistemas. Usa-se de uma estrutura de chave e valor e também de

listas ordenadas. É um dos formatos mais populares e mais utilizados para troca de mensagens entre sistemas.

Código de Estado

Usado pelo servidor para avisar o cliente sobre o estado da operação solicitada. Range varia de 100 a 500.

- **Ixx** - Informativo
- **2xx** - Sucesso
- **3xx** - Redirecionamento
- **4xx** - Erro do Cliente (quem fez a requisição colocou alguma mensagem errada. Ex: 404 not found)
- **5xx** - Erro do Servidor (erro de interno de processamento)

Tipos de arquitetura

Monolito

Monolito significa “obra construída em uma só pedra” por isso é utilizado para definir a arquitetura de alguns sistemas, refere-se a forma de desenvolver um sistema, programa ou aplicação onde todas as funcionalidades e códigos estejam em um único processo. Essas diversas funcionalidades estão em um mesmo código fonte e em sua execução compartilham recursos da mesma máquina, seja processamento, memória, bancos de dados e arquivos. Como o sistema está inteiro em um único bloco, seu desenvolvimento é mais ágil, se comparado com outras arquiteturas, sendo possível desenvolver uma aplicação em menos tempo e com menor complexidade inicial. A medida em que uma aplicação é descrita como monolítica depende muito de sua perspectiva. Em termos menos formais, a palavra é usada para se referir a algum sistema grande, que tenha apenas um código fonte.

Vantagens e Desvantagens de um Sistema Monolítico

As desvantagens e vantagens variam muito da proposta e do problema que seu sistema precisa resolver. Generalizando os problemas encontrados, podemos listar:

Manutenibilidade

Conforme uma aplicação monolítica cresce, diversas funções são adicionadas a um mesmo código e processo, o que pode acarretar em quedas em cascata da aplicação como um

todo. O código se torna complexo e difícil de dar manutenção, as entregas por sua vez acabam se tornam mais críticas, menos frequentes e até estáveis.

Escalabilidade

Como estamos na era da nuvem e custos por demanda, uma aplicação monolítica pode se tornar cara para ser escalada, por se tratar de um único código, todas as funcionalidades precisam ser escaladas como um todo, normalmente escalada verticalmente, adicionando mais máquinas (processador, memória, ...) para a aplicação, ou horizontalmente por modelos de平衡ador de carga.

Complexidade

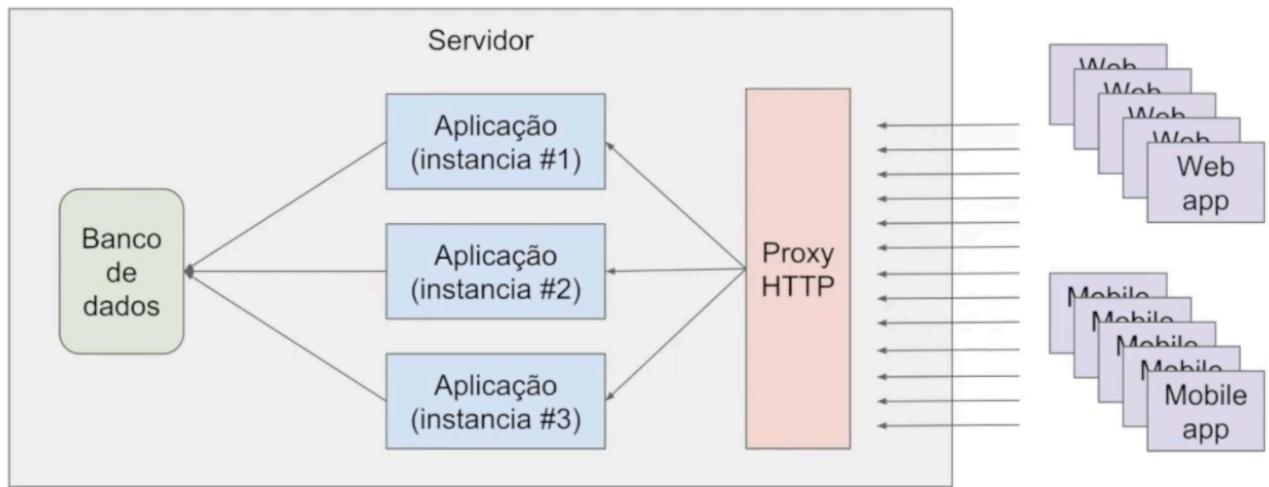
Para aplicações novas ou provas de conceito, onde a ideia ainda precisar ser validada, o sistema monolítico apresenta menor complexidade para desenvolvimento inicial, se comparado à outras arquiteturas. Muitas empresas optam por um caminho “híbrido”, onde desenvolvem e validam suas ideias construindo um monólito, mas já pensando e preparando o terreno para uma possível migração para sistemas distribuídos, por exemplo.

Custos

O custo de uma aplicação pode ser uma vantagem ou uma desvantagem, dependendo do cenário em que o sistema se encontra. Para projetos iniciais o seu custo tende a ser uma vantagem, visto que você precisa apenas de uma máquina para rodar todo o sistema. Com o passar do tempo e com a sua aplicação escalando, ela pode se tornar uma desvantagem.

Podemos também destacar os seguintes pontos sobre uma aplicação monolítica:

- Compartilha os recursos da máquina com todos os módulos do sistema.
- Possui apenas uma stack.
- Seu monitoramento é simplificado.
- Sua escalabilidade é complexa e demorada.



Microsserviços

Microsserviços são uma abordagem de arquitetura para a criação de aplicações. O que diferencia a arquitetura de microsserviços das abordagens monolíticas tradicionais é como ela decompõe a aplicação por funções básicas. Cada função é denominada um serviço e pode ser criada e implantada de maneira independente. Isso significa que cada serviço individual pode funcionar ou falhar sem comprometer os demais. Um microsserviço é uma função essencial de uma aplicação e é executado independentemente dos outros serviços. No entanto, a arquitetura de microsserviços é mais complexa do que o mero acoplamento flexível das funções essenciais de uma aplicação. Trata-se da restruturação das equipes de desenvolvimento e da comunicação entre serviços de modo a preparar a aplicação para falhas inevitáveis, escalabilidade futura e integração de funcionalidades novas. Pode-se dizer que microsserviços desenvolvem sistemas mais flexíveis, escaláveis, com manutenção mais fácil em comparação com outros sistemas tradicionais.

Quais são os benefícios da arquitetura de microsserviços?

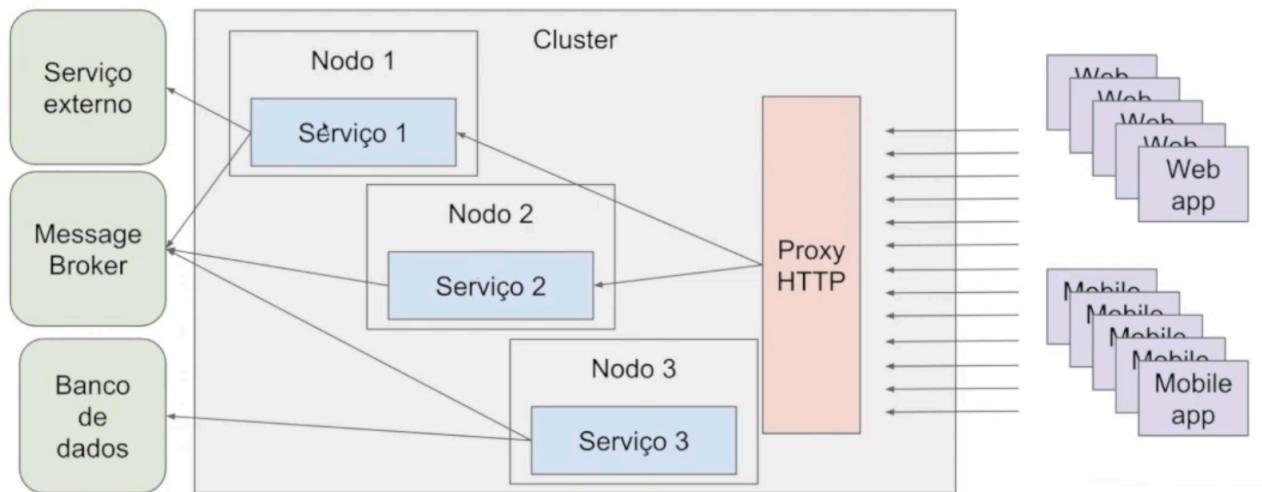
Com os microsserviços, suas equipes e tarefas rotineiras podem se tornar mais eficientes por meio do desenvolvimento distribuído. Além disso, é possível desenvolver vários microsserviços ao mesmo tempo. Isso significa que você pode ter mais desenvolvedores trabalhando simultaneamente na mesma aplicação, o que resulta em menos tempo gasto com desenvolvimento. Dentre os benefícios, temos:

- **Lançamento no mercado com mais rapidez:** Como os ciclos de desenvolvimento são reduzidos, a arquitetura de microsserviços é compatível com implantações e atualizações mais ágeis.
- **Altamente escalável:** À medida que a demanda por determinados serviços aumenta, você pode fazer implantações em vários servidores e infraestruturas para atender às suas necessidades.

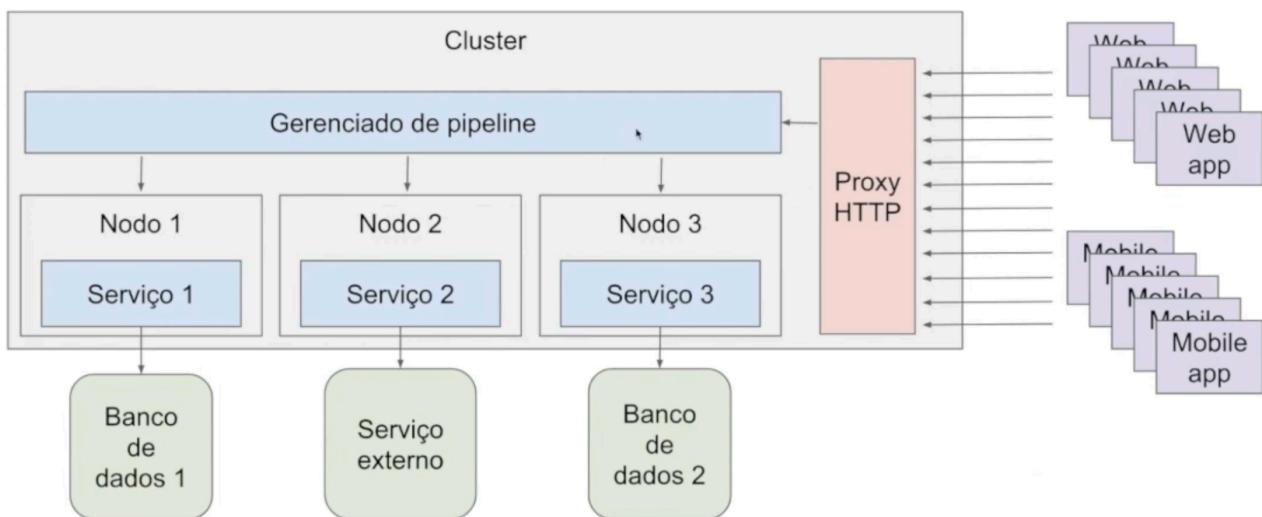
- **Resiliente:** Os serviços independentes, se construídos corretamente, não afetam uns aos outros. Isso significa que, se um elemento falhar, o restante da aplicação permanece em funcionamento, diferentemente do modelo monolítico.
- **Fácil de implementar:** Como as aplicações baseadas em microsserviços são mais modulares e menores do que as aplicações monolíticas tradicionais, as preocupações resultantes dessas implantações são invalidadas. Isso requer uma coordenação maior, mas as recompensas podem ser extraordinárias.
- **Acessível:** Como a aplicação maior é decomposta em partes menores, os desenvolvedores têm mais facilidade para entender, atualizar e aprimorar essas partes. Isso resulta em ciclos de desenvolvimento mais rápidos, principalmente quando também são empregadas as tecnologias de desenvolvimento ágil. Vale ressaltar que, apesar de estarem separados do sistema principal, as funcionalidades do microsserviço possuem a mesma responsabilidade.
- **Mais open source:** Devido ao uso de APIs poliglotas, os desenvolvedores têm liberdade para escolher a melhor linguagem e tecnologia para a função necessária (cada serviço pode ser composto por tecnologias distintas e complementares).

Quais são os desafios da arquitetura de microsserviços?

- **Compilação:** é necessário dedicar um tempo à identificação das dependências entre os serviços. É preciso estar ciente de que concluir uma compilação pode gerar muitas outras devido a essas dependências. Também é necessário levar em consideração como os microsserviços afetam os dados.
- **Testes:** os testes de integração, assim como os testes end-to-end, podem ser mais difíceis e importantes como jamais foram. Uma falha em uma parte da arquitetura pode provocar outra mais adiante, dependendo de como os serviços foram projetados para suportar uns aos outros.
- **Controle de versão:** ao atualizar para versões novas, a compatibilidade com versões anteriores pode ser rompida. É possível resolver esse problema usando a lógica condicional, mas isso pode se tornar outra complicaçāo rapidamente. Como alternativa, você pode colocar no ar várias versões ativas para clientes diferentes, mas essa solução é mais complexa em termos de manutenção e gerenciamento.
- **Implantação:** sim, isso também é um desafio, pelo menos na configuração inicial. Para facilitar a implantação, primeiro é necessário investir bastante na automação, pois a complexidade dos microsserviços é demais para a implantação manual. Pense sobre como e em que ordem os serviços serão implementados.
- **Geração de logs:** com os sistemas distribuídos, é necessário ter logs centralizados para unificar tudo. Caso contrário, é impossível gerenciar a escala.
- **Monitoramento:** é crítico ter uma visualização centralizada do sistema para identificar as fontes de problemas.
- **Depuração:** a depuração remota não é uma opção e não funciona com centenas de serviços. Infelizmente, no momento não há uma única resposta para como realizar depurações.
- **Conectividade:** considere a detecção de serviços, seja de maneira centralizada ou integrada.



O message broker é um intermediário entre as conexões realizadas entre os serviços que permite maior segurança e retorno de condições indevidas do sistema.



O gerenciador de pipeline recebe a requisição do proxy HTTP e conforme a etapa da requisição, a envia para um serviço específico. Caso fique fora do ar, o sistema inteiro também ficará.

IOT



A flexibilidade dos tópicos



mqqt // broker.io / a6g3l9 / gps / position
PROTÓCOLO **BROKER** **USER IDENTIFIER** **SENSOR** **INFORMATION TYPE**

QoS 0



MQTT Client

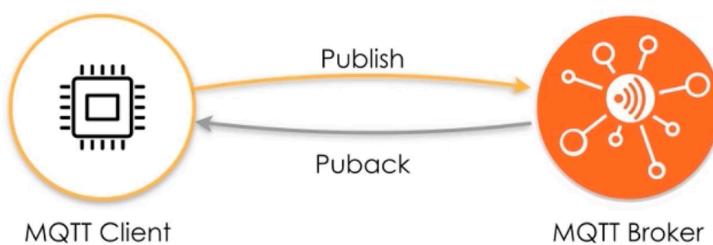
MQTT Broker

Nível mínimo de menor esforço

Sem garantia de entrega

Mensagem não é retransmitida

QoS 1



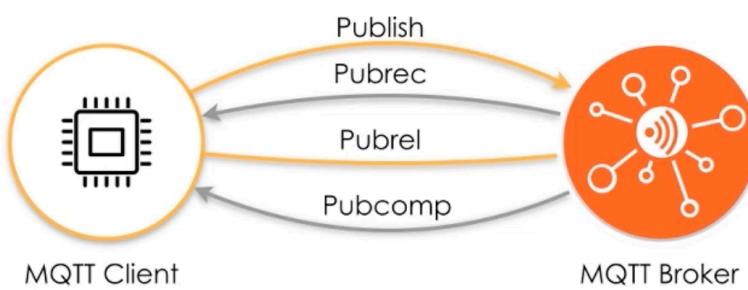
MQTT Client

MQTT Broker

Garante que a mensagem foi entregue no mínimo uma vez ao recebedor

Mensagem pode ser retransmitida se não houver confirmação de entrega

QoS 2



MQTT Client

MQTT Broker

Garante que a mensagem foi entregue no mínimo uma vez ao recebedor

Mensagem pode ser retransmitida se não houver confirmação de entrega

Cloud

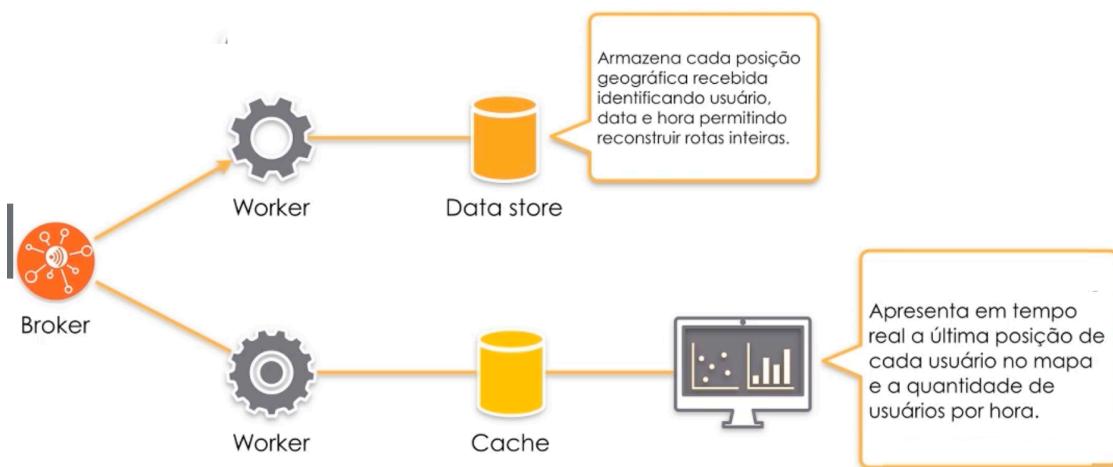


Grande e cada vez maior número de devices conectados

TBs ou PBs de informações

Potencial de escala global

Cloud



O que usar em cada etapa

Prova de Conceito



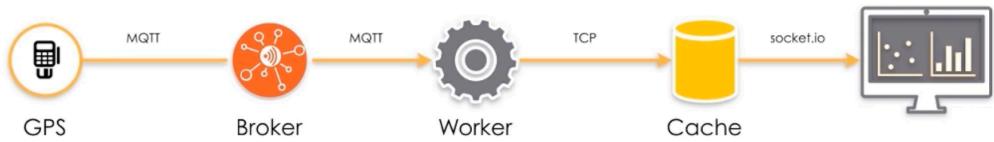
Mínimo Produto Viável



Solução



IoT na prática

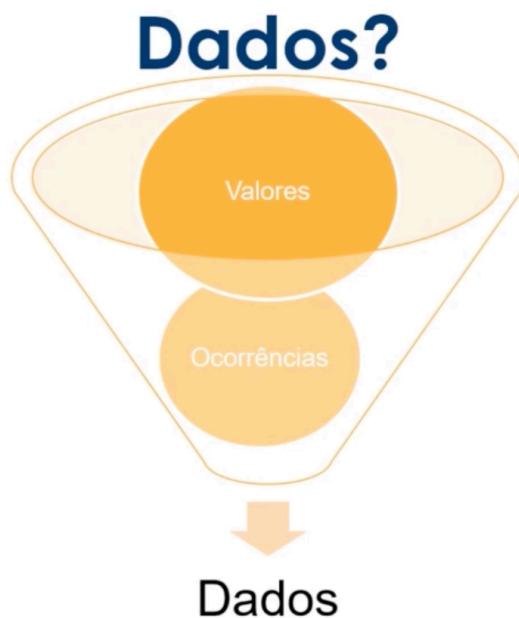


IoT na prática

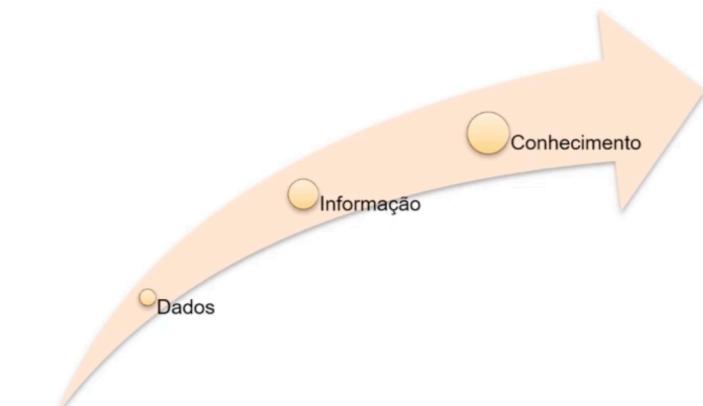


*Lambda é uma função.

Arquitetura de dados



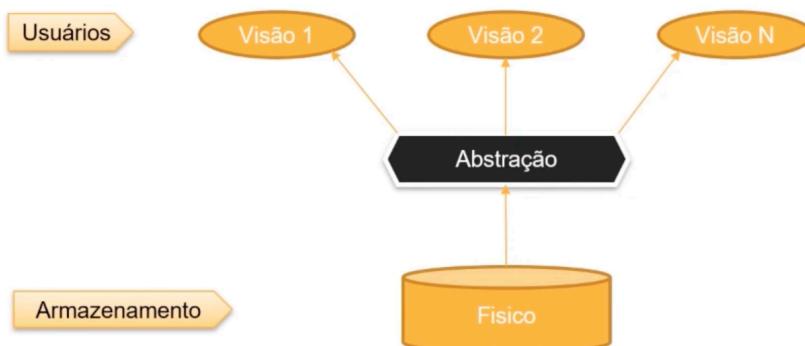
Importância



Modelo Sustentável



Abstração



SGDBs

Um Sistema de Gerenciamento de Banco de Dados (SGBD) — do inglês Data Base Management System (DBMS) — é o conjunto de softwares responsáveis pelo gerenciamento de um banco de dados. Seu principal objetivo é retirar da aplicação cliente a responsabilidade de gerenciar o acesso, a persistência, a manipulação e a organização dos dados. O SGBD disponibiliza uma interface para que seus clientes possam incluir, alterar ou consultar dados previamente armazenados. Seus três pilares são:

- **Linguagem de definição de dados:** especifica conteúdos, estrutura a base de dados e define os elementos de dados.
- **Linguagem de manipulação de dados:** para poder alterar os dados na base.
- **Dicionário de dados:** guarda definições de elementos de dados e respetivas características — descreve os dados, quem os acessam, etc.

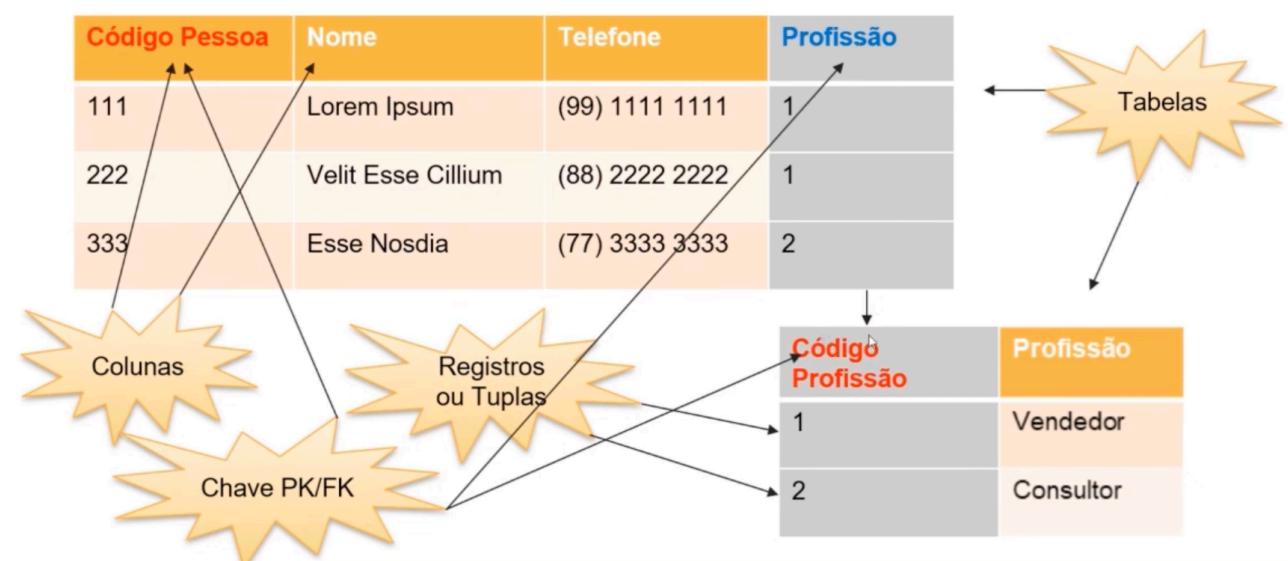
Quanto à estrutura, há vários modelos:

- **Modelo flat:** registros alotados em uma linha, como uma estruturação de colunas e linhas. Uma planilha de Excel é um exemplo deste modelo.
- **Modelo hierárquico:** a informação é dividida em grupos que formam uma árvore hierárquica. É um modelo pouco usado atualmente e oferece rapidez no acesso da informação. Por outro lado, pode apresentar redundâncias.
- **Modelo relacional:** modelo mais usado atualmente. É um conjunto de modelos flat que comunicam entre si, observando algumas regras.

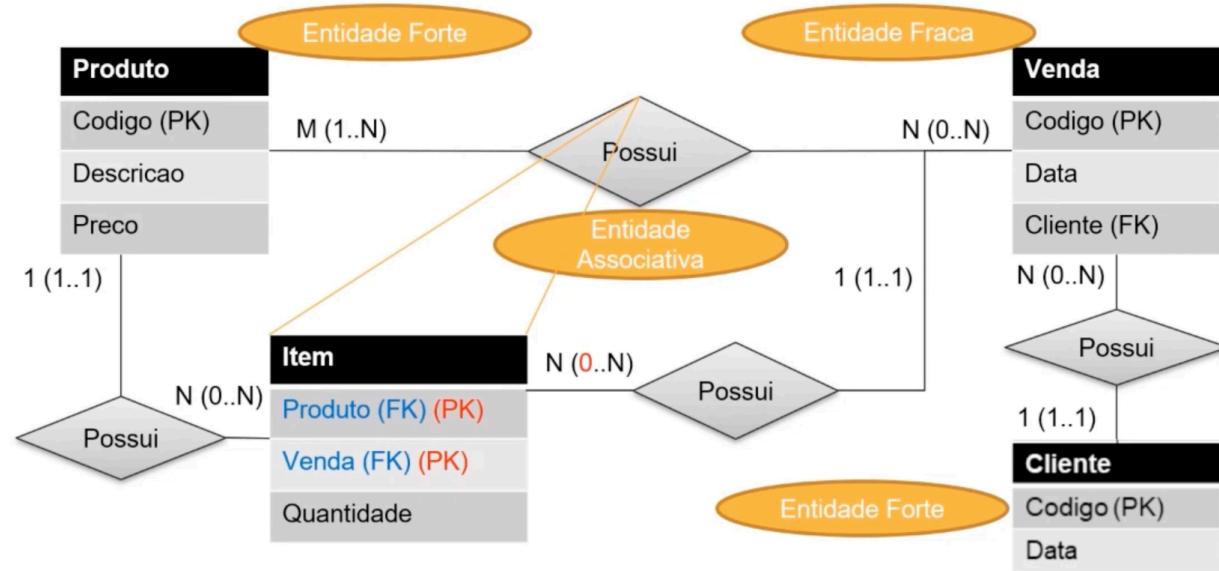
Outros modelos incluem: grapho, orientado a objetos, objeto-relacional e Big Data.

Banco de dados relacionais

SGDBR - RDBMS



MER - DER



- **Mer:** O Modelo Entidade Relacionamento (também chamado Modelo ER, ou simplesmente MER), como o nome sugere, é um modelo conceitual utilizado na Engenharia de Software para descrever os objetos (entidades) envolvidos em um domínio de negócios, com suas características (atributos) e como elas se relacionam entre si (relacionamentos). Em geral, este modelo representa de forma abstrata a estrutura que possuirá o banco de dados da aplicação. Obviamente, o banco de dados poderá conter várias outras entidades, tais como chaves e tabelas intermediárias, que podem só fazer sentido no contexto de bases de dados relacionais.
- **Der:** O Diagrama Entidade Relacionamento (Diagrama ER ou ainda DER) é a representação gráfica de um banco de dados e sua principal ferramenta. Em situações práticas, o diagrama é tido muitas vezes como sinônimo de modelo, uma vez que sem uma forma de visualizar as informações, o modelo pode ficar abstrato demais para auxiliar no desenvolvimento do sistema. Dessa forma, quando se está modelando um domínio, o mais comum é já criar sua representação gráfica, seguindo algumas regras. O diagrama facilita ainda a comunicação entre os integrantes da equipe, pois oferece uma linguagem comum utilizada tanto pelo analista, responsável por levantar os requisitos, e os desenvolvedores, responsáveis por implementar aquilo que foi modelado.
- **PK:** Este tipo de chave refere-se aos conjuntos de um ou mais campos, cujos valores, considerando a combinação de valores de todos os campos da tupla (registro), nunca se repetem e que podem ser usadas como um índice para os demais campos da tabela do banco de dados. Em chaves primárias, não pode haver valores nulos nem repetição de tuplas.
- **FK :** Este outro tipo de chave é utilizado para criar os relacionamentos entre as tabelas. Imagine que você queira cadastrar vários produtos que sejam de uma determinada categoria. Toda vez que você preencher os dados do produto, precisaremos indicar a chave primária da tabela categoria que seja da categoria que o nosso produto pertencerá. Ou seja, quando inserirmos um registro na tabela de produtos com o “id_categoria”, essa chave primária da tabela “categorias”

representará uma chave estrangeira (FK) dentro da tabela de produtos. É uma chave que vem de fora, de outra tabela.

- **Entidade forte:** pode existir por si só (um produto pode existir sem a venda).
- **Entidade fraca:** depende de outra entidade para existir (uma venda não pode existir sem um produto).
- **Relacionamento M para N:** no exemplo, temos um relacionamento M(1..N) para N(0..N), ou seja, uma venda pode possuir pelo menos 1 e N produtos, enquanto um produto pode ser parte de 0 ou N vendas. Este tipo de relacionamento existe a criação de uma nova entidade, a entidade associativa.
- **Entidade associativa:** entidade presente quando temos um relacionamento de M para N. A entidade Item é um exemplo.
- **Relacionamento 1 para N :** no exemplo, temos um relacionamento 1(1..1) para N(0..N). Isto significa que a venda só pode ser feita para um cliente, e que é necessário no máximo um cliente para que uma venda ocorra. Por outro lado, um cliente pode estar relacionado a 0 ou N vendas.
- **Relacionamento de n para n ou *..* (muitos para muitos):** neste tipo de relacionamento cada entidade, de ambos os lados, podem referenciar múltiplas unidades da outra. Por exemplo, em um sistema de biblioteca, um título pode ser escrito por vários autores, ao mesmo tempo em que um autor pode escrever vários títulos. Assim, um objeto do tipo autor pode referenciar múltiplos objetos do tipo título, e vice versa.

Normalização

Normalização é um processo a partir do qual se aplicam regras a todas as tabelas do banco de dados com o objetivo de evitar falhas no projeto, como redundância de dados e mistura de diferentes assuntos numa mesma tabela.

Ao projetar um banco de dados, se temos um modelo de entidades e relacionamentos e a partir dele construirmos o modelo relacional seguindo as regras de transformação corretamente, o modelo relacional resultante estará, provavelmente, normalizado. Mas, nem sempre os modelos que nos deparamos são implementados dessa forma e, quando isso acontece, o suporte ao banco de dados é dificultado.

Em ambos os casos, é necessário aplicar as técnicas de normalização, ou para normalizar (segundo caso citado), ou apenas para validar o esquema criado (primeiro caso citado). Aplicando as regras descritas a seguir, é possível garantir um banco de dados mais íntegro, sem redundâncias e inconsistências.

Existem 3 formas normais mais conhecidas:

- **1FN - 1^a Forma Normal:** todos os atributos de uma tabela devem ser atômicos, ou seja, a tabela não deve conter grupos repetidos e nem atributos com mais de um valor. Para deixar nesta forma normal, é preciso identificar a chave primária da tabela, identificar a(s) coluna(s) que tem(em) dados repetidos e removê-la(s), criar uma nova tabela com a chave primária para armazenar o dado repetido e, por fim, criar uma relação entre a tabela

principal e a tabela secundária. Por exemplo, considere a tabela Pessoas a seguir. $\text{PESSOAS} = \{\underline{\text{ID}} + \text{NOME} + \text{ENDERECO} + \text{TELEFONES}\}$

Ela contém a chave primária ID e o atributo TELEFONES é um atributo multivalorado e, portanto, a tabela não está na 1FN. Para deixá-la na 1FN, vamos criar uma nova tabela chamada TELEFONES que conterá PESSOA_ID como chave estrangeira de PESSOAS e TELEFONE como o valor multivalorado que será armazenado.

$\text{PESSOAS} = \{\underline{\text{ID}} + \text{NOME} + \text{ENDERECO}\}$

$\text{TELEFONES} = \{\underline{\text{PESSOA_ID}} + \text{TELEFONE}\}$

- **2FN - 2ª Forma Normal:** antes de mais nada, para estar na 2FN é preciso estar na 1FN. Além disso, todos os atributos não chaves da tabela devem depender unicamente da chave primária (não podendo depender apenas de parte dela). Para deixar na segunda forma normal, é preciso identificar as colunas que não são funcionalmente dependentes da chave primária da tabela e, em seguida, remover essa coluna da tabela principal e criar uma nova tabela com esses dados. Por exemplo, considere a tabela ALUNOS_CURSOS a seguir.

$\text{ALUNOS_CURSOS} = \{\underline{\text{ID_ALUNO}} + \underline{\text{ID_CURSO}} + \text{NOTA} + \text{DESCRICAO_CURSO}\}$

Nessa tabela, o atributo DESCRICAO_CURSO depende apenas da chave primária ID_CURSO. Dessa forma, a tabela não está na 2FN. Para tanto, cria-se uma nova tabela chamada CURSOS que tem como chave primária ID_CURSO e atributo DESCRICAO retirando, assim, o atributo DESCRICAO_CURSO da tabela ALUNOS_CURSOS.

$\text{ALUNOS_CURSOS} = \{\underline{\text{ID_ALUNO}} + \underline{\text{ID_CURSO}} + \text{NOTA}\}$

$\text{CURSOS} = \{\underline{\text{ID_CURSO}} + \text{DESCRICAO}\}$

- **3FN - 3ª Forma Normal:** para estar na 3FN, é preciso estar na 2FN. Além disso, os atributos não chave de uma tabela devem ser mutuamente independentes e dependentes unicamente e exclusivamente da chave primária (um atributo B é funcionalmente dependente de A se, e somente se, para cada valor de A só existe um valor de B). Para atingir essa forma normal, é preciso identificar as colunas que são funcionalmente dependentes das outras colunas não chave e extraí-las para outra tabela. Considere, como exemplo, a tabela FUNCIONARIOS a seguir.

$\text{FUNCIONARIOS} = \{\underline{\text{ID}} + \text{NOME} + \underline{\text{ID_CARGO}} + \text{DESCRICAO_CARGO}\}$

O atributo DESCRICAO_CARGO depende exclusivamente de ID_CARGO (atributo não chave) e, portanto, deve-se criar uma nova tabela com esses atributos. Dessa forma, ficamos com as seguintes tabelas:

$\text{FUNCIONARIOS} = \{\underline{\text{ID}} + \text{NOME} + \underline{\text{ID_CARGO}}\}$

$\text{CARGOS} = \{\underline{\text{ID_CARGO}} + \text{DESCRICAO}\}$

DDL – Data Definition Language

DML – Data Manipulation Language

DQL – Data Query Language

```
Create Table Cliente
(
    Código number(10) Not Null Primary Key,
    Nome varchar(50) Not Null,
    Telefone varchar(15)
)
```

```
Insert into Cliente (Código,Nome,Telefone)
values (1,"Lorem ipsum","(88) 999 9999")
```

↳

```
Delete from Cliente
Where Código = 1
```

```
Update Cliente
set Nome = "Lorem Ipsum"
Where Código = 1
```

```
Select Código,
      Nome
  from Cliente
<Where> Código = 1
<Group by> Profissão
<Having> Count(1) > 0
<Order by> Nome, Código
```

Transactions



Sistema ACID

Atomicidade: todas as operações são executadas com sucesso (o sistema sempre toma uma ação diante de um comando, seja commit ou rollback).

Consistência: sistema precisa garantir a unicidade de chaves, restrições, integridade lógica, etc.

Isolamento: várias transações podem acontecer simultaneamente.

Durabilidade - após aplicação do commit as alterações são realizadas, não importa o que ocorra.



SGDBs-R

ORACLE®

