DIM0547 - DESENVOLVIMENTO DE SISTEMAS WEB II

Prof: Jerffeson Gomes Dutra

Aula 01

Apresentação e Introdução da Disciplina

Apresentação

Nome: Jerffeson Gomes Dutra

E-mail: jerffeson.gomes@imd.ufrn.br

Quem são vocês:

- Nome;
- Já sabia programação Desenvolvimento Web?;
- Trabalha? Quais tecnologias usa?
- O que você se lembra de Web 1?
- Já viu Banco de dados?

As regras do jogo

- **Não será aceito uso de IA** no desenvolvimento dos Trabalho ou Provas;
- **Trabalhos copiados** em todo ou em parte de outros estudantes ou da Internet receberão automaticamente nota zero;
- **Códigos que não compilam** serão avaliados, porém com penalidade;
- **Pontos extras** serão adicionados em códigos com uso de boas práticas de programação (**Se necessário para aprovação**);

As regras do jogo

- **Trabalhos entregues fora do prazo não serão considerados;**

- **Atrasos resultam em meia falta;**

As regras do jogo

Faltas justificadas apenas mediante atestado

Qualquer problema, fale comigo, logo, não espero para o final

Avaliação

1. Prova escrita;
2. Trabalho Prático*;
3. Trabalho Prático*;
4. Reposição:
    - Prova escrita;
    - Trabalho Prático Correspondente + Algumas coisas;

* Apresentação é obrigatória

* Trabalhos em trio

Objetivos

- Introduzir os principais conceitos relacionados ao desenvolvimento de sistemas Web
- Apresentar arquiteturas utilizadas por sistema web e discutir benefícios e limitações de diferentes abordagens arquiteturais
- Introduzir os elementos dos principais arcabouços de desenvolvimento de sistemas Web
- Projetar e prototipar sistemas Web

COMPETÊNCIAS E HABILIDADES

Desenvolver competências e habilidades relacionadas a comunicação, trabalho em equipe, assertividade, gerenciamento de tempo, pensamento crítico, planejamento e estimativa, gerenciamento de riscos e gestão de projeto

VS Code

O Visual Studio Code (VS Code) é um editor de código-fonte leve, poderoso e altamente extensível, desenvolvido pela Microsoft para Windows, Linux e macOS.

Sprint Boot

O Spring Boot é um framework moderno do ecossistema Java, criado para simplificar e acelerar o desenvolvimento de aplicações web e microsserviços.

H2 Database

O H2 Database é um sistema de gerenciamento de banco de dados relacional (RDBMS) leve, rápido e escrito em Java.

Ele pode ser usado tanto no modo embutido (embedded), integrado diretamente dentro de aplicações Java, como em modo cliente-servidor, permitindo conexões remotas. É muito utilizado para desenvolvimento, testes, aplicações pequenas e também para situações que exigem alta performance com dados temporários, por exemplo, em cache ou processamento rápido.

Cliente API

- Postman
- Insomnia
- REST Client (Extensão VS Code)

Aula 02

Introdução a sistemas Web: conceitos fundamentais, cliente-servidor, HTTP

1980

1º Acesso a internet no Brasil

No Brasil, o acesso à internet começou no final dos anos 1980, com universidades conectando-se à rede americana.

1989

Fundada a Rede Nacional de Ensino e Pesquisa (RNP)

Implantar uma infraestrutura de redes avançada com alta capacidade de transmissão de dados e velocidade para interligar instituições de ensino e pesquisa em todo o país.

1990

Início da World Wide Web (WWW)

Tim Berners-Lee criando o primeiro servidor web, além da definição dos conceitos básicos como HTTP, HTML e URL.

1993

Navegador Mosaic

Popularizou a Web ao permitir combinar textos e gráficos em diferentes plataformas, facilitando o uso da internet para pessoas comuns

1995

Fundação da WWW Consortium para padronizar a Web

Lançamento do navegador Netscape, do Internet Explorer da Microsoft e da Amazon.com, que iniciou o comércio eletrônico de grande escala.

1997

Fundação da Netflix

Como locadora

1998

Fundação do Google

Revolucionando a busca e organização da informação na internet.

2004

Lançamento do Facebook

Transformação das redes sociais

2005

Lançamento do Youtube

O primeiro vídeo postado na plataforma foi "Me at the zoo" (Eu no zoológico), publicado por Jawed Karim em 23 de abril de 2005, com pouco mais de 19 segundos mostrando ele falando sobre elefantes em um zoológico.

2007

iPhone e Netflix como Stream

Impulsionou a internet móvel e o uso de smartphones, mudando profundamente o acesso e consumo de conteúdo digital através dos celulares.

Computadores pessoais

A popularização do computador pessoal desde a década de 1980 possibilitou que o usuário comum tivesse acesso à internet, que se ampliou na década de 1990 com provedores comerciais e expansão da rede para residências.

2010

Lançamento do Instagram e Pinterest

2012

Apple Siri e Google Now

Assistentes pessoais inteligentes ganham popularidade

2013

Lançamento do Telegram

2019

Popularização crescente de inteligência artificial

O lançamento do ChatGPT, que representou uma popularização massiva e mais visível da IA conversacional para o público amplo, só ocorreu em novembro de 2022.

A primeira arquitetura da Web

A internet no seu início, funcionava com uma arquitetura bastante diferente da internet atual, mas já com princípios que são a base até hoje.
A ARPANET tinha uma arquitetura básica. Os “hosts” (computadores de grande porte, como mainframes) funcionavam como servidores e clientes dependendo da tarefa.
Os terminais não eram meramente terminais tolos (simples interfaces), mas sim grandes computadores conectados à rede; usuários acessavam estes hosts muitas vezes por terminais remotos.

A primeira arquitetura da Web

A arquitetura cliente-servidor é um modelo de design onde duas entidades principais interagem para realizar tarefas: o cliente e o servidor.

- C**liente** é o dispositivo ou programa que inicia pedidos de serviço.
    - O terminal tolo, apenas enviando solicitações ao servidor e exibindo suas respostas. Ele apresenta a interface de usuário, mas não processa dados ou aplicações localmente.

- **Servidor** é o sistema que processa esses pedidos e retorna as respostas.

Arquitetura cliente servidor com navegador

A arquitetura passa a ser distribuída com maior participação do cliente. PCs tornam-se clientes com capacidade de processamento local, rotinas de interface do usuário e algumas funções aplicacionais.

Surgem protocolos mais robustos e padronizados de comunicação, preparando terreno para o HTTP (desenvolvido nos anos 90). Redes locais (LANs) e protocolos TCP/IP ganham corpo.

Arquitetura cliente servidor com navegador

Servidor é o sistema que processa esses pedidos e retorna as respostas.

O cliente (navegador) fazia requisições HTTP e recebia documentos para renderização, sem manipular muito a lógica ou o processamento.

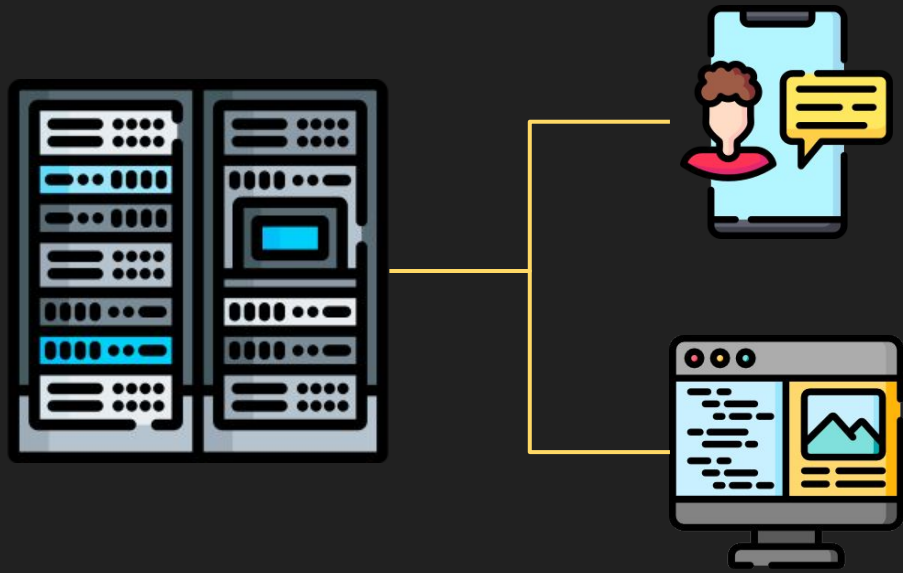
O servidor gerava e processava o conteúdo, seja páginas HTML, scripts para formulários, ou consultas a bancos de dados, enviando o resultado final ao cliente.

Arquitetura cliente servidor com navegador

Segurança: Começam a ser implementados mecanismos de autenticação e controle de acesso via rede, com firewalls e políticas de segurança.

Descentralização: Servidores migraram para múltiplos data centers em nuvem, com balanceamento de carga e escalonamento automático para atender milhões de usuários.

Arquitetura cliente servidor e os SPAs



Com o avanço das tecnologias web, houve uma grande mudança no paradigma cliente-servidor, chamada de Single Page Applications (SPAs);
Aplicações passaram a executar mais lógica e processamento diretamente no cliente (navegador), usando JavaScript para manipular a interface, validar dados, fazer requisições assíncronas (AJAX) e atualizar o conteúdo dinamicamente, sem necessidade de recarregar a página inteira.
Os servidores começaram a atuar mais como APIs, entregando apenas os dados.

Qual a consequência de mudar parte do processamento para o cliente?

Qual a consequência de mudar parte do processamento para o cliente?

Melhora na experiência do usuário (UX): Com processamento no cliente (navegador ou app), a interface torna-se mais responsiva e interativa, pois ações podem ser executadas localmente, sem a necessidade de esperar a resposta do servidor para cada interação. Isso reduz o tempo de carregamento e torna a navegação mais fluida.

Redução da carga no servidor: Parte do processamento e manipulação de dados ocorre no dispositivo do usuário, o que diminui o volume de requisições e o esforço computacional do servidor, podendo melhorar a escalabilidade da aplicação.

Qual a consequência de mudar parte do processamento para o cliente?

Aumento da complexidade no desenvolvimento: Dividir a lógica entre cliente e servidor requer soluções mais elaboradas, como o uso de APIs para comunicação, além de maior atenção à manutenção e sincronização dos estados da aplicação.

Desafios de segurança: Processar no cliente expõe parte da lógica e dados ao usuário final, aumentando o risco de manipulação indevida, fraudes ou vulnerabilidades. Assim, é fundamental implementar validações e controles robustos no servidor, que deve ser a fonte confiável dos dados e regras de negócio.

Qual a consequência de mudar parte do processamento para o cliente?

Dependência do dispositivo do usuário: Como o cliente executa código e processamento, o desempenho da aplicação pode variar conforme a capacidade do hardware, navegador e conexão do usuário, o que pode impactar a experiência em dispositivos mais antigos ou com rede lenta.

Maior uso de recursos locais: O processamento no cliente consome CPU, memória e bateria do dispositivo, aspectos importantes especialmente em celulares.

E se não tivesse havido essa mudança para processamento do cliente?

E se não tivesse havido essa mudança para processamento do cliente?

SPAs: Sem processamento no cliente, cada clique precisaria recarregar a página inteira do servidor, tornando a experiência lenta e ineficiente. Cada clique, que arrastar e soltar seria necessário um carregamento da página.

Rolagem infinita, animações, filtros em tempo real, e interações instantâneas (como digitar e já ver resultados). Isso exigiria processamento imediato no navegador — impossível com um cliente "burro".

E se não tivesse havido essa mudança para processamento do cliente?

Se tudo dependesse do servidor, seria muito mais caro escalar serviços (milhões de usuários ao mesmo tempo). O modelo atual permite que parte da carga vá para o cliente, economizando recursos de backend.

Dark mode, traduções automáticas, acessibilidade, adaptações por dispositivo — tudo isso depende de lógica executada no navegador.

Celulares e tablets, com suas variações de conexão e desempenho, se beneficiam muito do processamento no cliente para otimizar uso da rede, consumo de bateria e dar respostas rápidas ao usuário.

Conceitos Fundamentais da Web e Desenvolvimento Web

Modelo Cliente-Servidor: A Web é baseada em uma arquitetura cliente-servidor, onde o cliente (navegador web, por exemplo) faz requisições para servidores que processam e retornam recursos (páginas, dados, serviços).

Protocolos HTTP/HTTPS: Protocolo de comunicação para troca de informações entre cliente e servidor. HTTPS adiciona camada de segurança via criptografia.

Recurso: Se refere a qualquer elemento ou dado acessível via rede, que pode ser solicitado, manipulado ou entregue por um servidor por meio de um protocolo como o HTTP.

URLs e DNS: URLs são os endereços amigáveis que localizam recursos na web, enquanto DNS traduz esses nomes em endereços IP.

Conceitos Fundamentais da Web e Desenvolvimento Web

HTML: Linguagem de marcação para estruturação de conteúdo nas páginas web.

CSS: Folhas de estilo que definem a aparência dos elementos HTML.

JavaScript: Linguagem de programação que traz interatividade e manipulação dinâmica dos elementos web no cliente.

Frameworks Frontend (React, Angular, Vue): Bibliotecas e frameworks que facilitam o desenvolvimento de interfaces complexas e reativas no lado cliente.

Conceitos Fundamentais da Web e Desenvolvimento Web

MVC (Model-View-Controller): Separação das responsabilidades em camada de dados (Model), camada de apresentação (View) e lógica de aplicação (Controller).

Arquitetura Multicamadas: Separação em camadas distintas, como camada de apresentação, lógica de negócio, persistência e dados.

Middleware: Componentes intermediários que gerenciam comunicação, autenticação, sessões, entre outras funções em aplicações web.

Conceitos Fundamentais da Web e Desenvolvimento Web

Servidores Web (Tomcat, Apache, Nginx): Responsáveis por servir páginas e recursos HTTP aos clientes.

Frameworks Backend (JSF, Spring, EJB): Plataformas para construir a lógica de negócio, controle de fluxo e acesso a dados no servidor.

APIs Web / Serviços Web: Interfaces que permitem comunicação entre sistemas, normalmente via RESTful, SOAP ou GraphQL.

Conceitos Fundamentais da Web e Desenvolvimento Web

Single Page Applications (SPAs): Aplicações web que carregam uma única página e atualizam a interface dinamicamente, melhorando a experiência do usuário.

Comunicação Assíncrona (AJAX, Fetch API): Permite que páginas web solicitem dados em segundo plano sem recarregar a página.

Autenticação e Autorização: Controle de acesso aos recursos da aplicação, uso de sessões, tokens (JWT) e políticas de segurança.

Responsividade: Design adaptativo para diferentes tamanhos de tela e dispositivos.

Progressive Web Apps (PWA): Aplicações web com funcionalidades próximas a apps nativos, como funcionamento offline e notificações push.

Conceitos Fundamentais da Web e Desenvolvimento Web

Maven e Gradle: Maven e Gradle são ferramentas de gerenciamento de dependências e automação de build.

pom.xml: contém todas as informações necessárias para que o Maven possa construir, gerenciar dependências, executar testes, empacotar e implantar o software.

build.gradle: equivalente ao pom.xml, o build.gradle é um script geralmente escrito em Groovy (ou Kotlin) onde o mesmo tipo de configuração é feito, mas com uma sintaxe mais flexível e programável.

Protocolo HTTP

O protocolo HTTP (Hypertext Transfer Protocol) foi criado por Tim Berners-Lee em 1989 no CERN, para viabilizar a comunicação e transferência de documentos na World Wide Web. Sua primeira versão, conhecida como HTTP/0.9 e lançada em 1991, era bastante simples e limitada: permitia apenas o método GET para solicitar páginas HTML e respondia com o conteúdo bruto, sem cabeçalhos ou metadados adicionais.

Protocolo HTTP - Problemas

Simplicidade excessiva: só suportava requisições GET e respostas contendo apenas o corpo da mensagem (conteúdo HTML) sem cabeçalhos.

Sem suporte a outros métodos HTTP: não havia POST, HEAD ou outros que permitissem manipulação mais rica dos recursos.

Sem metadados nas respostas: o servidor não enviava informações como tipo de conteúdo, código de status, ou controle de cache, dificultando o controle e a evolução do protocolo.

Conexão não persistente: uma conexão era aberta para cada requisição, o que gerava overhead e lentidão no carregamento de páginas com múltiplos recursos (imagens, scripts etc.).

Sem mecanismos para negociação de conteúdo ou segurança.

Protocolo HTTP

A evolução do HTTP passou por versões sucessivas que agregaram funcionalidades para resolver essas limitações:

- HTTP/1.0 (1996): Introduziu novos métodos (POST, HEAD), cabeçalhos para metadados, status de resposta e suporte para múltiplos tipos de conteúdo (MIME). Ainda usava conexões não persistentes, o que limitava o desempenho
- HTTP/1.1 (1997): Definiu o padrão da Web até o início dos anos 2010, trazendo conexões persistentes (múltiplas requisições/ respostas na mesma conexão), cache mais eficiente, transfers chunked (para transmissão de dados em fragmentos), e melhorias no controle de fluxo. Essas mudanças reduziram latência e melhoraram a performance da navegação.

Protocolo HTTP - Problemas

- HTTP/2 (2015): Introduziu multiplexação (envio simultâneo de múltiplas requisições sem bloqueio), compactação de cabeçalhos, priorização de recursos e push do servidor, o que aumentou bastante eficiência e rapidez de páginas web modernas. Mantém a mesma semântica do HTTP/1.1, facilitando a adoção.

- HTTPS: Não é uma nova versão do HTTP, mas a aplicação do HTTP sobre TLS/SSL, acrescentando uma camada de segurança por criptografia da comunicação. Isso protege dados contra interceptações, ataques de man-in-the-middle e garante privacidade e integridade, sendo hoje o padrão para quase toda comunicação web segura.

HTTP - Request

Uma request HTTP (requisição HTTP) é o pedido que um cliente (como um navegador ou aplicativo) envia a um servidor para solicitar ou enviar algum recurso ou serviço na web. Uma requisição HTTP é composta basicamente por três partes principais:

- Linha inicial (Request Line)
- Cabeçalhos (Headers)
- Corpo da requisição (Body), que é opcional dependendo do tipo de requisição.

HTTP - Linha inicial (Request Line)

Contém quatro elementos fundamentais:

- **Método HTTP**: verbo que indica a ação desejada pelo cliente, como GET, POST, PUT, DELETE, entre outros.
- **URI (Uniform Resource Identifier)**: o caminho ou recurso solicitado no servidor.
- **Versão do HTTP**: identifica qual versão do protocolo está sendo usada, por exemplo, HTTP/1.1
- **Status Code** (Incluído na resposta)

HTTP - Cabeçalhos (Headers)

São linhas com pares nome: valor que adicionam informações extras sobre a requisição. Esses cabeçalhos permitem controlar cache, cookies, idioma, entre outras configurações.

- Host: identifica o domínio do servidor (obrigatório em HTTP/1.1).
- User-Agent: identifica o cliente (navegador ou app).
- Accept: tipos de conteúdo que o cliente aceita (ex: text/html).
- Content-Type: tipo do corpo da requisição (ex: application/json), quando presente.
- Authorization: dados para autenticação.

HTTP - Corpo da requisição (Body)

Opcional, presente em métodos que enviam dados ao servidor, como POST, PUT ou PATCH.

Contém os dados que o cliente quer enviar, como formulários, JSON, arquivos etc.

O corpo acompanha cabeçalhos como Content-Length (tamanho do corpo) e Content-Type para descrever o formato dos dados.

HTTP - Métodos

Os métodos HTTP são comandos usados na comunicação entre cliente (por exemplo, um navegador) e servidor, indicando a ação que o cliente deseja que o servidor execute sobre um recurso.

- GET
- POST
- PUT
- PATCH
- DELETE
- HEAD
- OPTIONS

HTTP - GET

A requisição GET solicita a representação do recurso (como uma página HTML, JSON, imagem) e não deve alterar dados no servidor (é um método seguro e idempotente).

idempotente: propriedade de uma operação ou solicitação que, quando repetida várias vezes, produz o mesmo resultado que seria obtido com apenas uma execução.

https://www.metropoledigital.ufrn.br/portal/

HTTP - GET

Os dados podem ser enviados via URL na composição da URL ou como query string

- Composição da URL
  - …..ufrn.br/portal/visualizar/**754**
  - ….ufrn.br/portal/noticias/**7707**/nome-noticia

O que acontece se eu não colocar essa informação na URL?

- Conteúdo não encontrado (404)
- Acesso não autorizado (403)
- Erro Interno do Servidor (500)
- Funciona normalmente

HTTP - GET

Os dados podem ser enviados via URL na composição da URL ou como query string

- A query string é a parte após o símbolo de interrogação (?), seguida de pares chave-valor separados por &
  - ……ufrn.br/portal/editais?**tipo-edital=3**&**ano-edital=2024**
  - ….google.com/search?q=imd

O que acontece se eu não colocar essa informação na URL?

- Se bem configurado, não deve dar erro, nem página não autorizada

A URL tem limite que pode variar de 2.000 a 65k caracteres, mas depende do navegador

HTTP - GET

Por que não gera erro não colocar essa informação na URL?

HTTP - GET

Por que não gera erro não colocar essa informação na URL?

Por que continua sendo a mesma URL

HTTP - POST

Usado para enviar dados ao servidor, geralmente para criar um novo recurso.

- Os dados são enviados no corpo (body) da requisição, não visíveis na URL.
- Pode modificar o estado no servidor (não é idempotente).

Exemplo: Enviar um formulário de cadastro.

HTTP - POST

Os dados enviados no corpo da requisição POST não são criptografados por padrão, porque o protocolo HTTP em si é um protocolo de texto simples que transmite as mensagens em texto puro pela rede.

A criptografia dos dados na requisição só ocorre se a comunicação for feita sobre uma camada segura, como no caso do HTTPS, que é o HTTP encapsulado em uma conexão TLS/SSL.

HTTP - POST

Que tipos de dados enviar no corpo para aproveitar essa criptográfica?

HTTP - POST

Que tipos de dados enviar no corpo para aproveitar essa criptográfica?

- Senhas e credenciais de login (exemplo: via POST em um formulário de autenticação)
- Dados pessoais sensíveis (nome, CPF, endereço, telefone)
- Dados financeiros (número de cartão, informações bancárias, transações)
- Formulários complexos (com múltiplos campos e dados estruturados)
- Objetos JSON ou XML para APIs que enviam dados estruturados
- Arquivos enviados via upload
- Tokens de autenticação (JWT, API keys)
- Atualizações e modificações em recursos do servidor (exemplo: via PUT, PATCH)

HTTP - POST

Uma requisição GET e POST podem ter a mesma URL?

HTTP - POST

Uma requisição GET e POST podem ter a mesma URL?

Sim! Uma requisição para a mesmo URL pode ser feita com métodos diferentes e resultados diferentes.

HTTP - PUT

Usado para atualizar ou substituir um recurso existente no servidor.

A requisição carrega uma representação **completa** dos campos do recurso, mesmo os que não sofreram alterações.

É idempotente — várias requisições PUT idênticas produzem o mesmo resultado.

Exemplo: Atualizar as informações de um usuário.

HTTP - PUT

```java
public class Pessoa {
    String nome;
    String cpf;
    LocalDate dataNascimento;
}
```

```json
{
  "nome": "Maria Silva",
  "cpf": "123.456.789-00",
  "dataNascimento": "1990-05-20"
}
```

HTTP - PATCH

Usado para atualizar **parcialmente** um recurso no servidor.

Envia apenas as mudanças específicas, não o recurso completo.

Útil para atualizações parciais que não requerem enviar todos os dados do recurso.

HTTP - PATCH

```java
public class Pessoa {
    String nome;
    String cpf;
    LocalDate dataNascimento;
}
```

```json
{
  "dataNascimento": "1985-12-15"
}
```

HTTP - DELETE

Usado para remover um recurso especificado no servidor.

O servidor processa a exclusão e pode retornar a confirmação.

Exemplo: Excluir um usuário ou um arquivo.

HTTP - HEAD

O método HTTP HEAD funciona solicitando ao servidor apenas os cabeçalhos (headers) da resposta referentes a um recurso específico, sem incluir o corpo da mensagem que conteria os dados do recurso em si.
Isso é semelhante a uma requisição GET, porém o servidor não envia o conteúdo do recurso, apenas os metadados.

HTTP - HEAD

Utilidades práticas:

- Validação de cache: verificando se um recurso foi modificado antes de baixá-lo novamente.
- Cheque de disponibilidade: confirmação rápida se um recurso existe e está acessível.
- Otimização de desempenho: reduz tempo e dados transferidos antes de requisições completas.
- Segurança e monitoramento: pode ajudar a validar links e analisar o comportamento do servidor sem transferir dados completos.

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 256
Last-Modified: Wed, 07 Aug 2025 10:00:00 GMT
```

HTTP - OPTIONS

O método HTTP OPTIONS é usado para que o cliente descubra quais métodos HTTP e opções de comunicação são permitidos para um recurso específico em um servidor.

Ou seja, ele pergunta ao servidor: "O que posso fazer com este recurso?".

HTTP - OPTIONS

O servidor responde com cabeçalhos, como o Allow, que lista os métodos HTTP permitidos para aquele recurso, por exemplo: Allow: GET, POST, OPTIONS.

Também pode retornar cabeçalhos relacionados a CORS (Cross-Origin Resource Sharing), indicando permissões para chamadas de origens diferentes, como Access-Control-Allow-Origin, Access-Control-Allow-Methods, entre outros.

A resposta não contém corpo, apenas os cabeçalhos.

```
HTTP/1.1 200 OK
Allow: GET, POST, DELETE, OPTIONS
Access-Control-Allow-Origin: https://app.exemplo.com
Access-Control-Allow-Methods: GET, POST, DELETE, OPTIONS
Access-Control-Allow-Headers: Content-Type, Authorization
Access-Control-Max-Age: 86400
```

HTTP - Cabeçalho

Host: Indica o domínio do servidor que está sendo acessado. Exemplo: Host: www.exemplo.com

User-Agent: Identifica o cliente que faz a requisição, geralmente um navegador ou aplicação, informando o nome, versão e sistema operacional. Exemplo: User-Agent: Mozilla/5.0

Accept: Especifica os formatos de conteúdo que o cliente aceita, como text/html ou application/json.

Accept-Language: Indica a preferência de idiomas do cliente, para negociação de conteúdo multilíngue. Exemplo: Accept-Language: en-US,en;q=0.9

Authorization: Carrega credenciais para autenticação no servidor.

Content-Type: Indica o tipo dos dados no corpo da requisição (ex: application/json), importante em métodos como POST e PUT.

Content-Length: Especifica o tamanho do corpo da requisição em bytes.

Cookie: Envia dados de sessão ao servidor para manter estado entre requisições.

HTTP Status Code

Os códigos de status de resposta HTTP indicam se uma solicitação HTTP específica foi concluída com êxito ou erro.

As respostas são agrupadas em cinco classes:

Respostas Informativas (100 – 199)

Respostas bem-sucedidas (200 – 299)

Mensagens de redirecionamento (300 – 399)

Respostas de erro do cliente (400 – 499)

Respostas de erro do servidor (500 – 599)

HTTP Status Code

Código	Significado	Descrição
200	OK	A solicitação foi bem-sucedida.
201	Created	A requisição foi bem sucedida e um novo recurso foi criado como resultado. Esta é normalmente a resposta enviada após as solicitações POST ou algumas solicitações PUT.
400	Bad Request	O servidor não pode ou não irá processar a solicitação devido a algo que é percebido como um erro do cliente (por exemplo, sintaxe de solicitação malformada, enquadramento de mensagem de solicitação inválida ou roteamento de solicitação enganosa).
401	Unauthorized	Embora o padrão HTTP especifique "unauthorized", semanticamente, essa resposta significa "unauthenticated". Ou seja, o cliente deve se autenticar para obter a resposta solicitada.
403	Forbidden	O cliente não tem direitos de acesso ao conteúdo; ou seja, não é autorizado, portanto o servidor está se recusando a fornecer o recurso solicitado.

HTTP Status Code

Código	Significado	Descrição
404	Not Found	O servidor não pode encontrar o recurso solicitado. No navegador, isso significa que o URL não é reconhecido. Em uma API, isso também pode significar que o endpoint é válido, mas o próprio recurso não existe. Os servidores também podem enviar esta resposta em vez de 403 Forbidden para ocultar a existência de um recurso de um cliente não autorizado.
405	Method Not Allowed	O método de solicitação é conhecido pelo servidor, mas não é suportado pelo recurso de destino. Por exemplo, uma API pode não permitir chamar DELETE para remover um recurso.
408	Request Timeout	Esta resposta é enviada por alguns servidores em uma conexão ociosa, mesmo sem qualquer requisição prévia pelo cliente. Isso significa que o servidor gostaria de desligar esta conexão não utilizada.
415	Unsupported Media Type	O formato de mídia dos dados requisitados não é suportado pelo servidor, portanto, o servidor está rejeitando a requisição.

HTTP Status Code

Código	Significado	Descrição
500	Internal Server Error	O servidor encontrou uma situação com a qual não sabe lidar.
501	Not Implemented	O método de solicitação é conhecido pelo servidor, mas não é suportado pelo recurso de destino. Por exemplo, uma API pode não permitir chamar DELETE para remover um recurso.

Para mais acesse: https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Reference/Status

MIME (Media Types)

São usados para indicar o tipo de conteúdo que é trabalhado na requisição.

MIME (Media Types)

São usados para indicar o tipo de conteúdo que é trabalhado na requisição.

Categoria	Exemplos de tipos MIME comuns	Extensões típicas
texto simples	`text/plain`	`.txt`
HTML para páginas web	`text/html`	`.html`, `.htm`
folhas de estilo CSS	`text/css`	`.css`
scripts JS	`text/javascript` ou `application/javascript`	`.js`
Valores separados por vírgula	`text/csv`	`.csv`

MIME (Media Types)

São usados para indicar o tipo de conteúdo que é trabalhado na requisição.

Categoria	Exemplos de tipos MIME comuns	Extensões típica
imagem JPEG	`image/jpeg`	`.jpeg`, `.jpg`
imagem PNG	`image/png`	`.png`
imagem GIF animada	`image/gif`	`.gif`
imagem SVG vetorial	`image/svg+xml`	`.svg`

MIME (Media Types)

São usados para indicar o tipo de conteúdo que é trabalhado na requisição.

Categoria	Exemplos de tipos MIME comuns	Extensões típica
MIDI	`audio/midi`	`.mid`, `.midi`
MP3	`audio/mpeg`	`.mp3`
OGG áudio	`audio/ogg`	`.ogg`
WAV	`audio/wav`	`.wav`

MIME (Media Types)

São usados para indicar o tipo de conteúdo que é trabalhado na requisição.

Categoria	Exemplos de tipos MIME comuns	Extensões típica
MP4	`video/mp4`	`.mp4`
WEBM	`video/webm`	`.webm`
OGG vídeo	`video/ogg`	`.ogv`

MIME (Media Types)

São usados para indicar o tipo de conteúdo que é trabalhado na requisição.

Categoria	Exemplos de tipos MIME comuns	Extensões típica
dados JSON para APIs	`application/json`	`.json`
documentos PDF	`application/pdf`	`.pdf`
arquivos ZIP compactados	`application/zip` ()	`.zip`
binários genéricos	`application/octet-stream`	(qualquer)
dados XML	`application/xml`	`.xml`
dados de formulários padrão	`application/x-www-form-urlencoded` ()	(usado em POST)
scripts JavaScript	`application/javascript`	`.js`

MIME (Media Types)

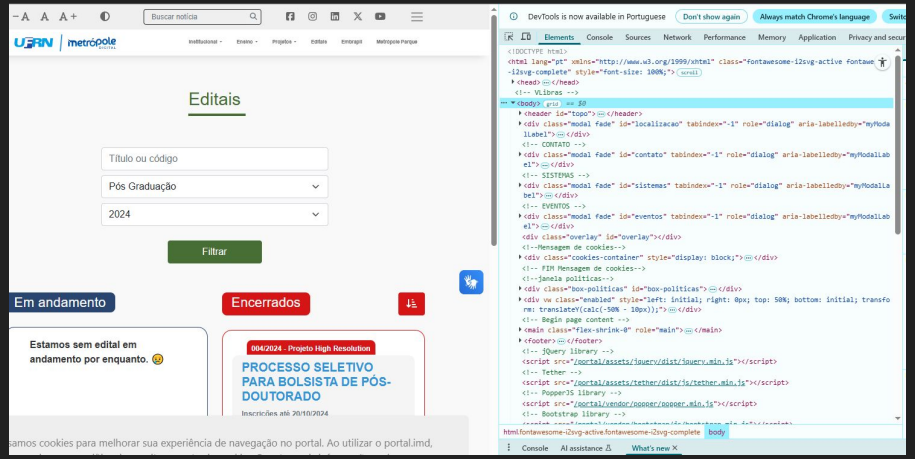
Se você enviar um tipo (Content-Type) diferente do arquivo que está enviando ou recebendo, podem ocorrer alguns problemas:

- O navegador pode interpretar ou tratar o arquivo incorretamente. Por exemplo, se você enviar um arquivo de áudio mas indicar um tipo MIME de texto (text/plain), o navegador pode tentar exibi-lo como texto, gerando erro ou conteúdo ilegível.
- Ações automáticas de software podem falhar ou agir de forma inesperada. Aplicações que dependem do tipo MIME para decidir como abrir, visualizar ou processar o arquivo podem rejeitar o conteúdo ou apresentar erro.
- Mecanismos de segurança podem bloquear ou restringir o acesso. Navegadores modernos, por segurança, impõem restrições se um arquivo for servido com tipo MIME incompatível, para evitar ataques como injeção de código ou execução indevida.
- O padrão recomendado na ausência do tipo MIME correto é usar application/octet-stream. Esse tipo genérico indica que o conteúdo é um arquivo binário e força o navegador a tratar o arquivo como download, sem tentar exibi-lo. No entanto, isso impede ações específicas, como reprodução em players integrados.

O servidor e o cliente podem usar "MIME sniffing" para tentar adivinhar o formato real do arquivo, o que pode mitigiar parcialmente problemas, mas considerado inseguro e com comportamento variável entre navegadores.

Como ver esses dados?

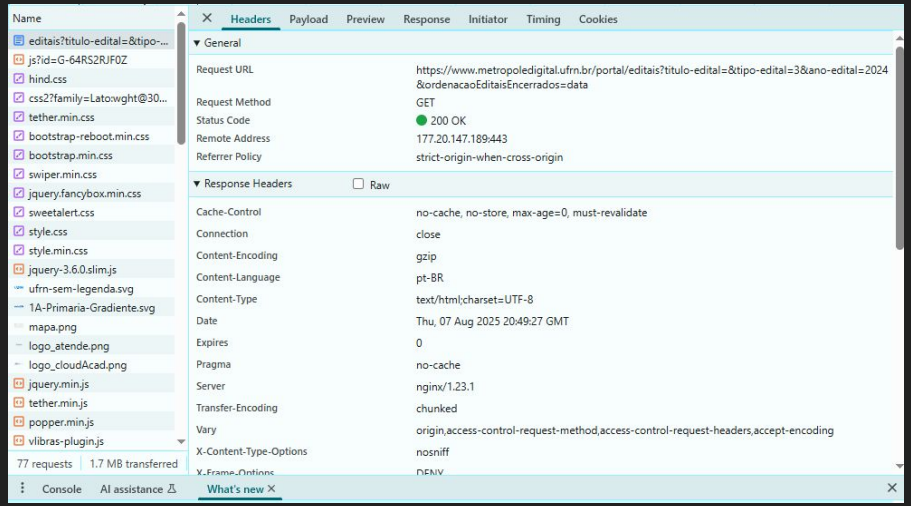
Basta inspecionar a página do navegador ou abrir o console do navegador (F12 no Chrome)
Selecionar a opção Rede ou Network e escolher a Request que você quer analisar



Como ver esses dados?

Basta inspecionar a página do navegador ou abrir o console do navegador (F12 no Chrome) Selecionar a opção Rede ou Network e escolher a Request que você quer analisar

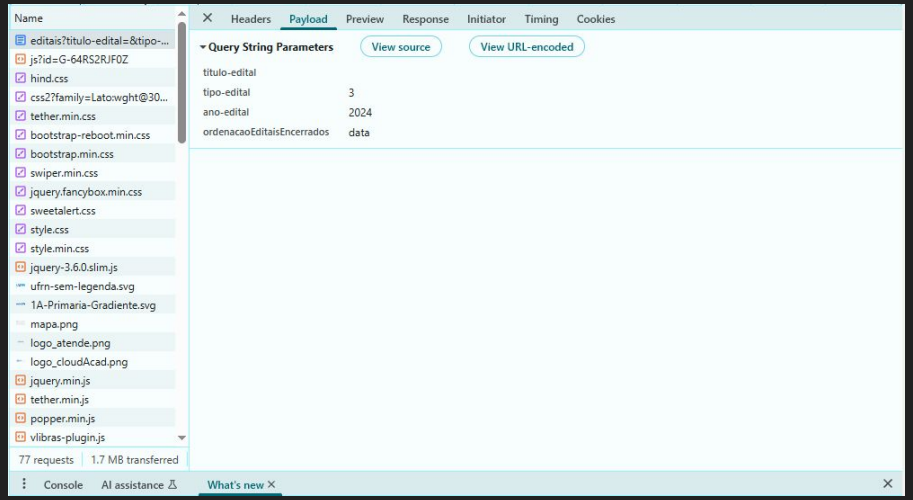
Na aba Headers exibe o cabeçalho da requisição e da resposta



Como ver esses dados?

Basta inspecionar a página do navegador ou abrir o console do navegador (F12 no Chrome) Selecionar a opção Rede ou Network e escolher a Request que você quer analisar

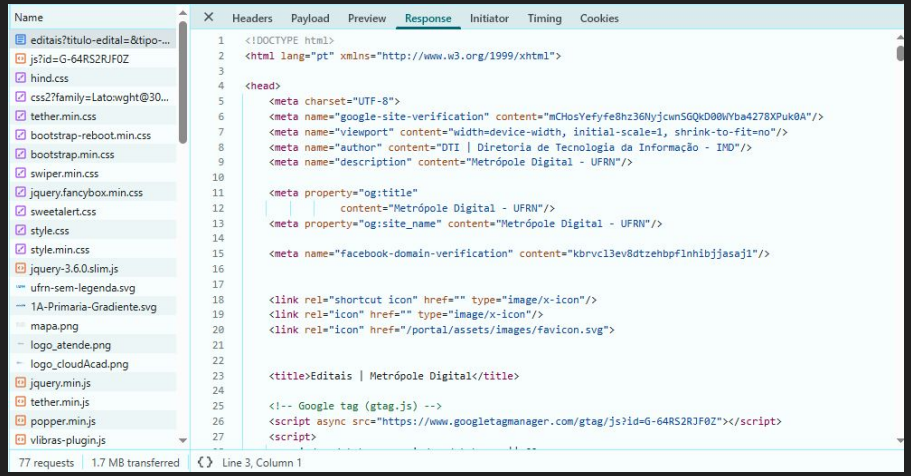
Na aba payload exibe os dados enviados na URL como query string ou os dados enviados no corpo



Como ver esses dados?

Basta inspecionar a página do navegador ou abrir o console do navegador (F12 no Chrome) Selecionar a opção Rede ou Network e escolher a Request que você quer analisar

Na aba Response ou Resposta exibe a resposta da requisição



Atividade

1- Brinquem com a API https://dummyjson.com/docs e testem requisições dos mais diversos tipos

2 - Acesse algum site que você tem curiosidade

Aula 03

Arquiteturas para sistemas Web: monolítica, multicamadas, cliente-servidor, arquitetura REST.

Arquiteturas para Sistemas Web

A arquitetura de sistemas é a estrutura organizacional e o design geral de um sistema de software, que define seus componentes principais, as formas como eles se interconectam e interagem, além das diretrizes para seu funcionamento eficaz, manutenção e escalabilidade.

Arquiteturas para Sistemas Web

Sistemas mal arquitetados têm vida útil reduzida, exigem reparos constantes, atualizações emergenciais e muitas vezes acabam sendo substituídos mais cedo, aumentando o custo total de propriedade.

Arquiteturas para Sistemas Web

Quem já trabalhou com sistemas legados com arquitetura duvidosa?

Qual as dificuldades que você encontrou?

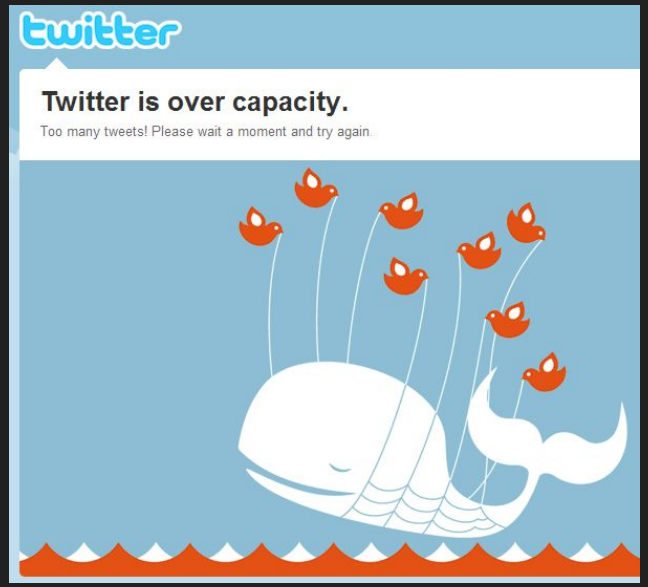
Arquiteturas para Sistemas Web

Spotify (primeiros anos)

- Inicialmente um monólito Python.
- À medida que o número de usuários cresceu, o monólito ficou impossível de escalar.
- Migraram para microserviços para dividir responsabilidades (streaming, playlists, recomendações).

Twitter (2006–2010)

- Originalmente escrito em Ruby on Rails, tudo rodava em um monólito.
- Com aumento de usuários, enfrentaram o “Fail Whale”: travamentos constantes.
- Reescreveram partes críticas em Scala e microserviços.

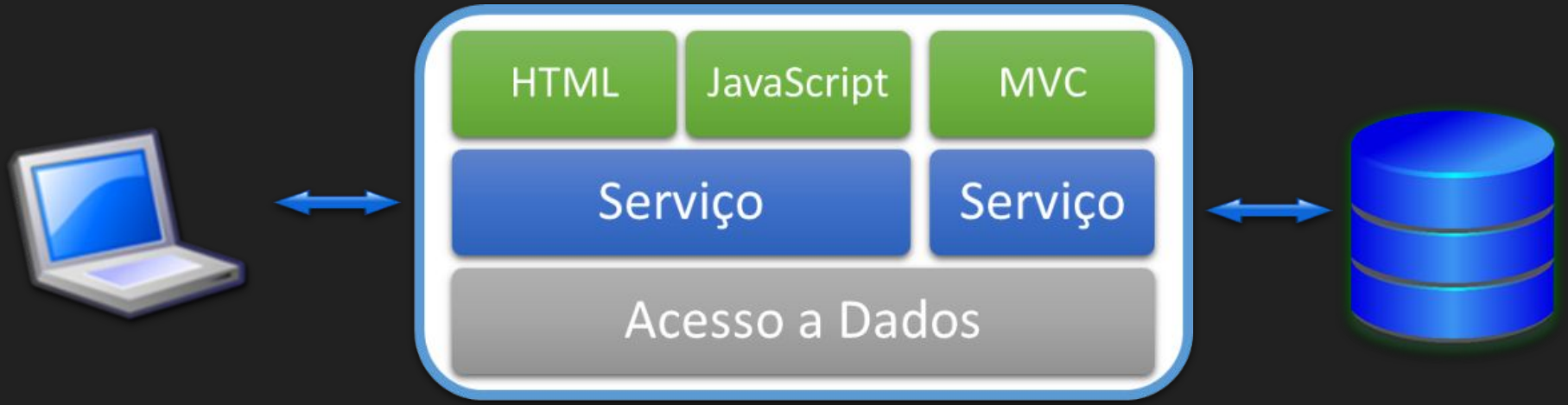


Arquitetura Monolítica

A arquitetura monolítica consiste em um sistema em que todos os componentes — interface, lógica de negócio e acesso a dados — são desenvolvidos e implantados como uma única aplicação indivisível.

Isso significa que o código fonte, o processo de execução e os recursos são agrupados em um único bloco, facilitando o desenvolvimento inicial, mas criando vínculos fortes entre as partes.

Arquitetura Monolítica



Arquitetura Monolítica - Tipos

Sistemas Monolíticos Totalmente Acoplados

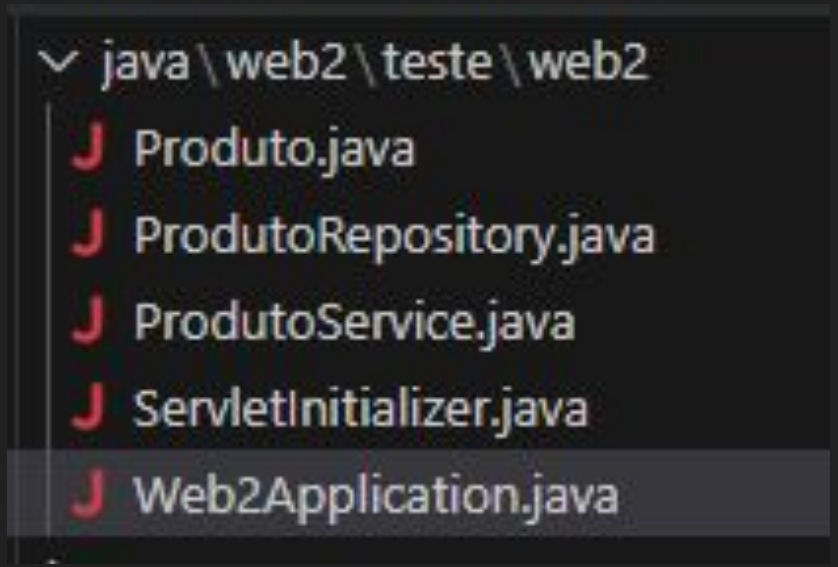
Caracterizam-se por forte acoplamento entre seus módulos, onde todos os componentes estão interligados e dependem diretamente uns dos outros.

A comunicação entre partes ocorre internamente sem separação clara de responsabilidades. Embora simples, esse tipo dificulta a manutenção e evolução, pois mudanças em uma parte podem impactar outras facilmente.

Arquitetura Monolítica - Tipos

Sistemas Monolíticos Totalmente Acoplados

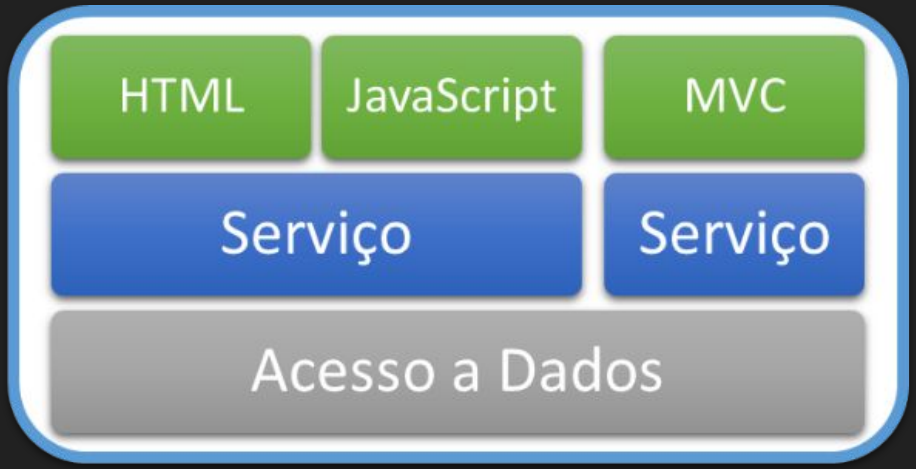
Neste tipo, há pouca ou nenhuma separação lógica entre camadas e responsabilidades.
Tudo fica misturado, dificultando manutenção.



Arquitetura Monolítica - Tipos

Sistemas Monolíticos Separados ou Monólitos Distribuídos

Nesse tipo, embora seja um sistema único, os módulos podem estar mais separados ("distribuídos" quase que logicamente), mas ainda dependem de um banco de dados compartilhado e precisam subir juntos no deployment.

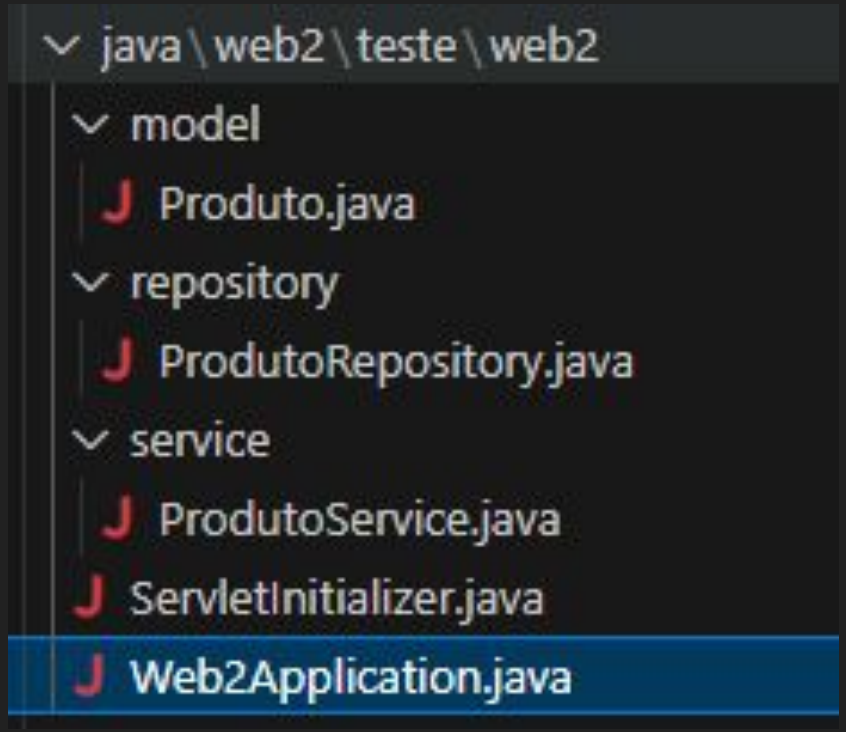


Aplicação Monolítica

Arquitetura Monolítica - Tipos

Sistemas Monolíticos Separados ou Monólitos Distribuídos

Aqui o sistema é estruturado internamente em pacotes que representam as camadas lógicas: apresentação, negócio e persistência, mas tudo está num único projeto e deploy.



Arquitetura Monolítica - Tipos

Sistemas Monolíticos Modulares

O monolito é um projeto único no deploy, mas dividido em módulos internos (subprojetos dentro do projeto), cada um responsável por um contexto ou funcionalidade do sistema. Cada módulo (produto, usuario) tem seu próprio código, pacotes, e pom.xml com dependências.

Arquitetura Monolítica - Tipos

Sistemas Monolíticos Modulares

Traz algumas dificuldades extras

- Dependências duplicadas
- Entender a comunicação entre módulos
- Complexidade de build

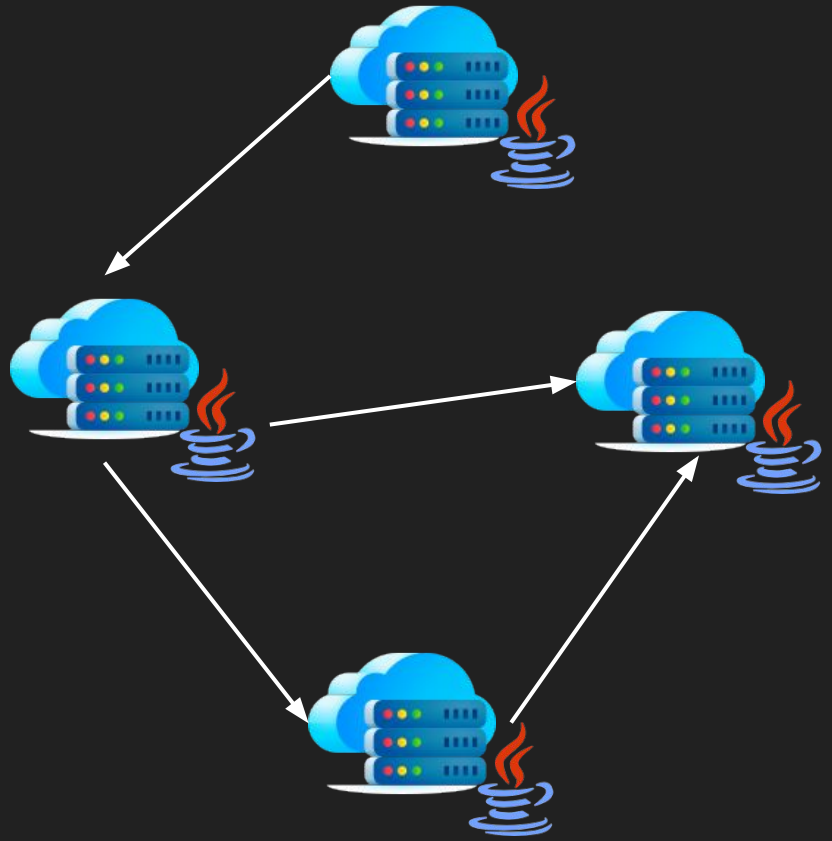
```
meu-sistema-monolito/
├── produto
│   └── src/main/java/com/exemplo/produto/...
├── usuario
│   └── src/main/java/com/exemplo/usuario/...
└── aplicacao
    └── src/main/java/com/exemplo/aplicacao/MainApp.java
```

Arquitetura Monolítica - Desvantagens

- Escalabilidade limitada: Escalar um componente específico exige escalar toda a aplicação.
- Manutenção difícil: Mudanças pequenas em um módulo podem exigir a recompilação e o redeploy de toda a aplicação.
- Risco de falhas abrangentes: Uma falha em uma parte pode derrubar todo o sistema.
- Dificuldade para evolução tecnológica: Inovar em partes específicas pode ser restrito, pois as tecnologias ficam atreladas ao monolito.
- Longo tempo para builds e deploys à medida que o sistema cresce.

Micro Serviços

A arquitetura de microserviços é um estilo de desenvolvimento onde um sistema é dividido em pequenos serviços independentes, cada um responsável por uma funcionalidade específica (ex.: autenticação, pagamentos, estoque, catálogo de produtos).

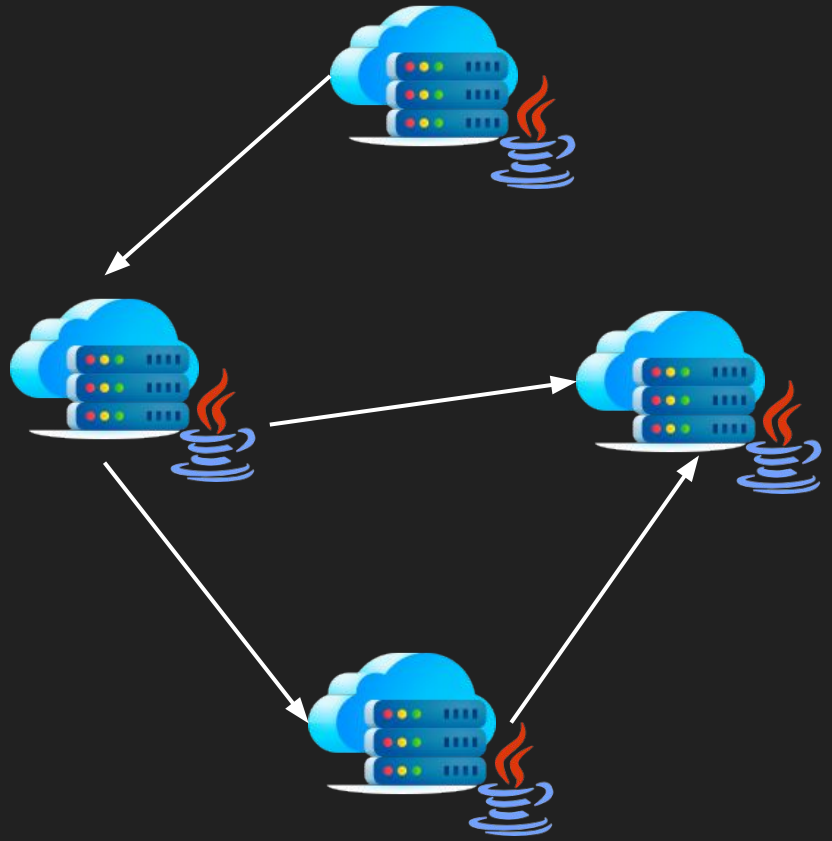


Micro Serviços

Diferença para Monólito

Monólito: todo o sistema está num único projeto/deploy (se uma parte quebra ou precisa ser escalada, o sistema todo é afetado).

Microserviços: o sistema é composto de vários serviços menores e independentes, que se comunicam entre si via rede (HTTP, mensageria, etc.).

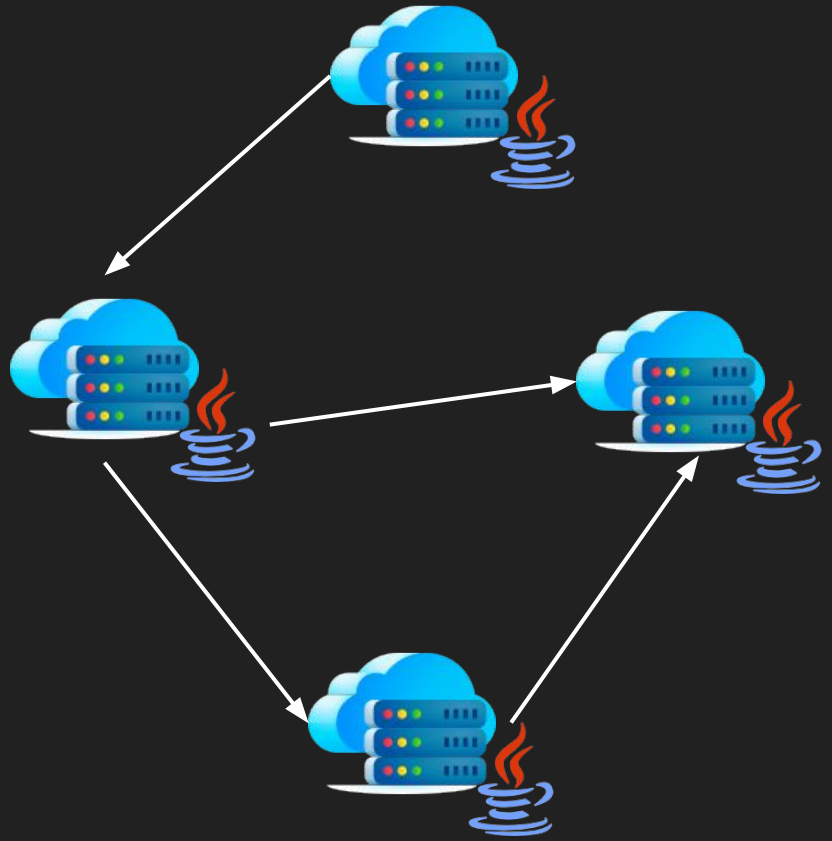


Micro Serviços

Diferença para Monólito

Monólito: todo o sistema está num único projeto/deploy (se uma parte quebra ou precisa ser escalada, o sistema todo é afetado).

Microserviços: o sistema é composto de vários serviços menores e independentes, que se comunicam entre si via rede (HTTP, mensageria, etc.).



Arquitetura Monolítica - Desvantagens

- Por que se usa?
- Por que não ir direto para uma arquitetura de microserviços?

Arquitetura Monolítica - Vantagens

- **Simplicidade no desenvolvimento inicial**: Menos complexidade na integração entre módulos, pois tudo está no mesmo ambiente.
- **Facilidade de teste**: Como é um único sistema, realizar testes iniciais pode ser mais direto.
- **Implantação única**: Um único artefato é implantado, simplificando os processos de deployment.
- **Menor necessidade de coordenação entre equipes**: Equipes pequenas podem trabalhar sem a necessidade complexa de integração entre serviços.
- **Menor custo de setup**: Não precisa de infraestrutura complexa: não precisa de container, orquestração (Kubernetes), API Gateway, filas, deploys independentes.

Arquitetura Monolítica - Vantagens

Ir direto para um micro serviços gera:

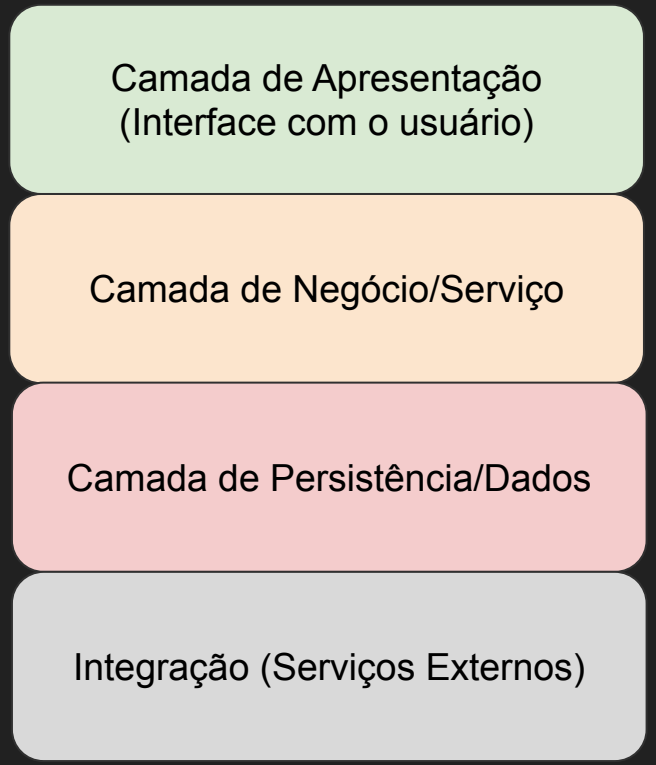
- Complexidade alta desde o início;
- Overhead de infraestrutura;
- Dificuldade de design inicial;
- Equipe com capacidade técnica mais elevada.

Arquitetura Multicamadas (Multitier)

Separa a aplicação em camadas lógicas distintas. Cada camada foca em uma função específica, facilitando o desenvolvimento, manutenção e testes. Isso promove maior organização e clareza no projeto. As 3 camadas são:

1. Camada de Apresentação (Interface com o usuário): Responsável por interagir com o usuário.
2. Camada de Negócio/Serviço: Contém regras de negócio e processamento.
3. Camada de Persistência/Dados: Acesso e mapeamento de dados.
4. Integração (Serviços Externos).

O padrão MVC é o mais famoso dessa arquitetura



Arquitetura Multicamadas (Multitier)

Sua ideia é que cada camada seja:

- Independência e reutilização:

  As camadas podem ser desenvolvidas, alteradas, testadas e até substituídas independentemente, desde que respeitadas as interfaces entre elas.

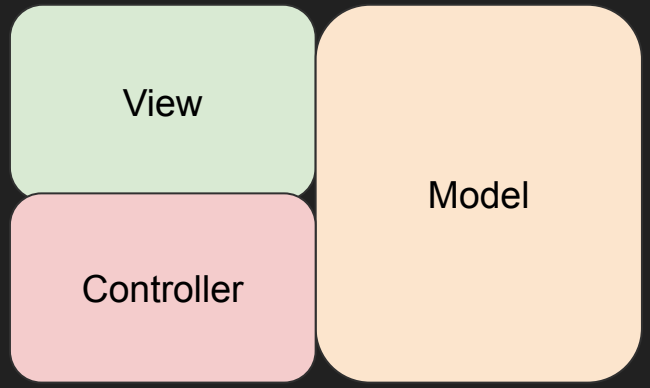
- Escalável:

  É possível distribuir as camadas em diferentes servidores ou máquinas, otimizando recursos e performance. Por exemplo, a camada de apresentação pode rodar no cliente, a de negócio em servidores de aplicação, e a de dados em servidores específicos de banco.

Padrão MVC

O padrão Model-View-Controller geralmente se encaixa na arquitetura multicamadas, onde:

1. View corresponde à camada de apresentação,
2. Controller faz parte da lógica intermediária (negócio e controle),
3. Model representa os dados e regras associadas.



Arquitetura Multicamadas (Multitier) - Vantagens

- **Facilita manutenção e evolução**: mudanças em uma camada raramente impactam outras diretamente.
- **Melhora a organização do código**: claro isolamento de responsabilidades.
- **Reutilização de componentes**: camadas podem servir para múltiplos canais ou sistemas.
- **Escalabilidade**: camadas podem ser escaladas independentemente.
- **Facilidade para aplicar segurança**: restrições podem ser aplicadas em determinadas camadas, como autenticação na camada de negócio.
- **Independência tecnológica**: as camadas podem usar diferentes tecnologias (ex: JSF na apresentação, Spring/EJB no negócio, diferentes bancos na camada de dados).

Arquitetura Multicamadas (Multitier) - Desvantagens

- **Overhead de comunicação**: comunicação entre camadas, especialmente se estiverem distribuídas, pode gerar latência e custos computacionais.
- **Complexidade inicial**: planejamento e design exigem mais cuidado, exigindo definição clara de interfaces e contratos.
- **Possível impacto no desempenho**: devido à passagem de mensagens entre camadas.
- **Dificuldade em definir granularidade correta**: muitas camadas ou camadas muito finas podem aumentar a complexidade sem ganhos proporcionais.
- **Tratamento de erros e transações mais complexos**: erros precisam ser propagados e tratados entre camadas de forma adequada.

O que não MVC

- Aplicações puramente de backend sem interface
- Aplicações CLI (linha de comando) simples
- Sistemas event-driven ou reativos
    - Ex.: sistemas de mensageria, IoT, processamento de eventos.

Por que não são MVC?

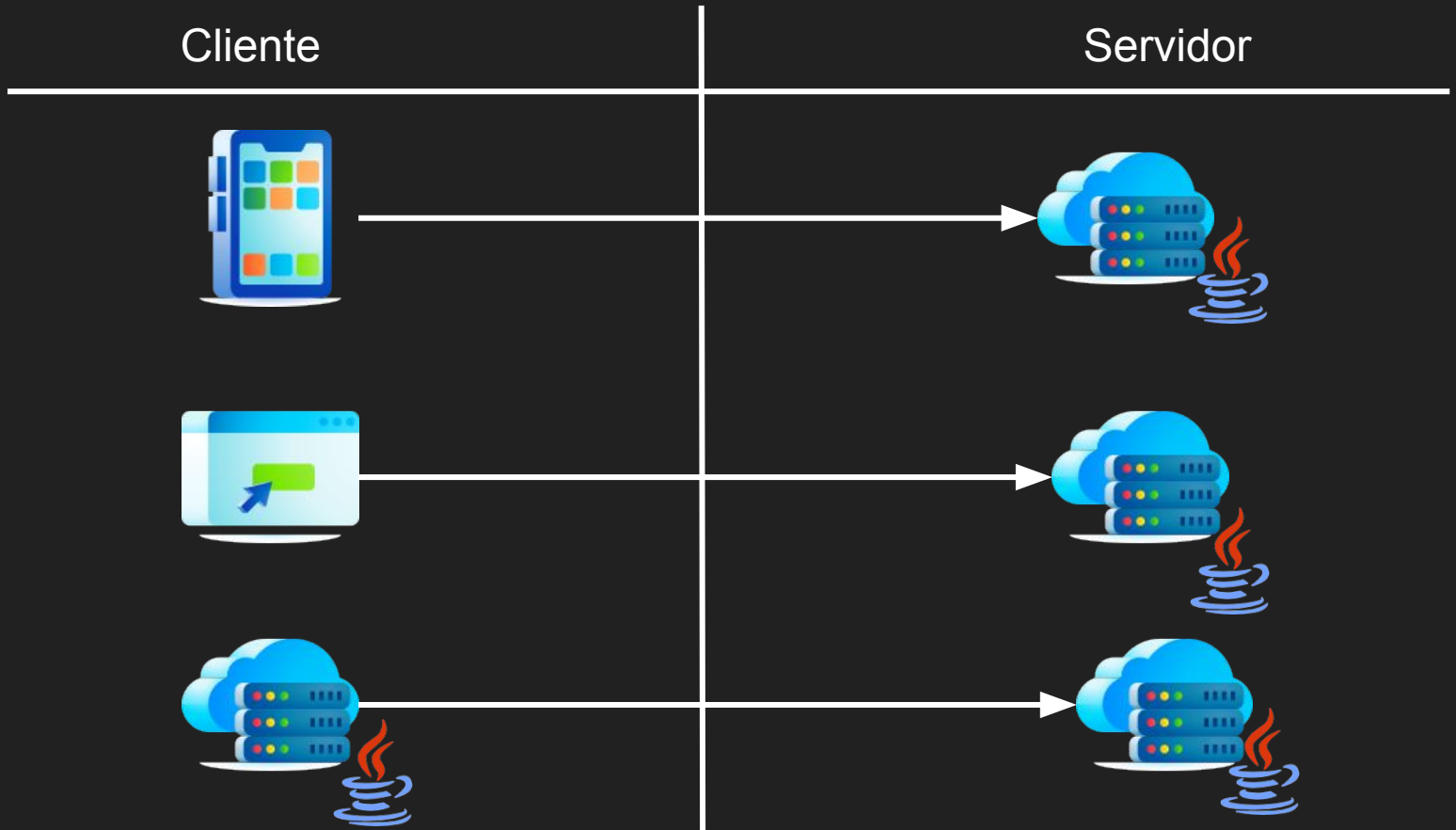
Podem ser multicamadas?

Arquitetura Cliente-Servidor

Divisão do sistema entre dois tipos principais de dispositivos:

- **Cliente**: Interface que interage com o usuário, geralmente um navegador ou aplicativo.
- **Servidor**: Processa as solicitações do cliente, realiza operações de negócio e acesso a dados.

Arquitetura Cliente-Servidor



Arquitetura Cliente-Servidor

Cliente

Funções principais:

1. Gerenciar interface;
2. Capturar comandos;
3. Enviar requisições;
4. Processar respostas e
5. Validar dados.

Servidor

Funções principais:

1. Processar comandos;
2. Gerenciar recursos e segurança;
3. Fornecer dados;
4. Serviços ao cliente e
5. Regras de negócio.

Arquitetura Cliente-Servidor

Cliente

Servidor

Validação de Dados

Garantir que os dados fornecidos para o sistema estejam corretos no formato, tipo e em um estado básico aceitável antes de serem processados.

Regras de Negócio

Garantir que o comportamento do sistema siga as políticas, decisões e processos específicos do domínio do problema, ou seja, como o sistema deve funcionar para atender aos objetivos do negócio.

Arquitetura Cliente-Servidor - Cliente

Validação de Dados:
Geralmente são simples, voltadas para a usabilidade e experiência do usuário, como campos obrigatórios, formato correto, tamanhos, consistência básica, etc.

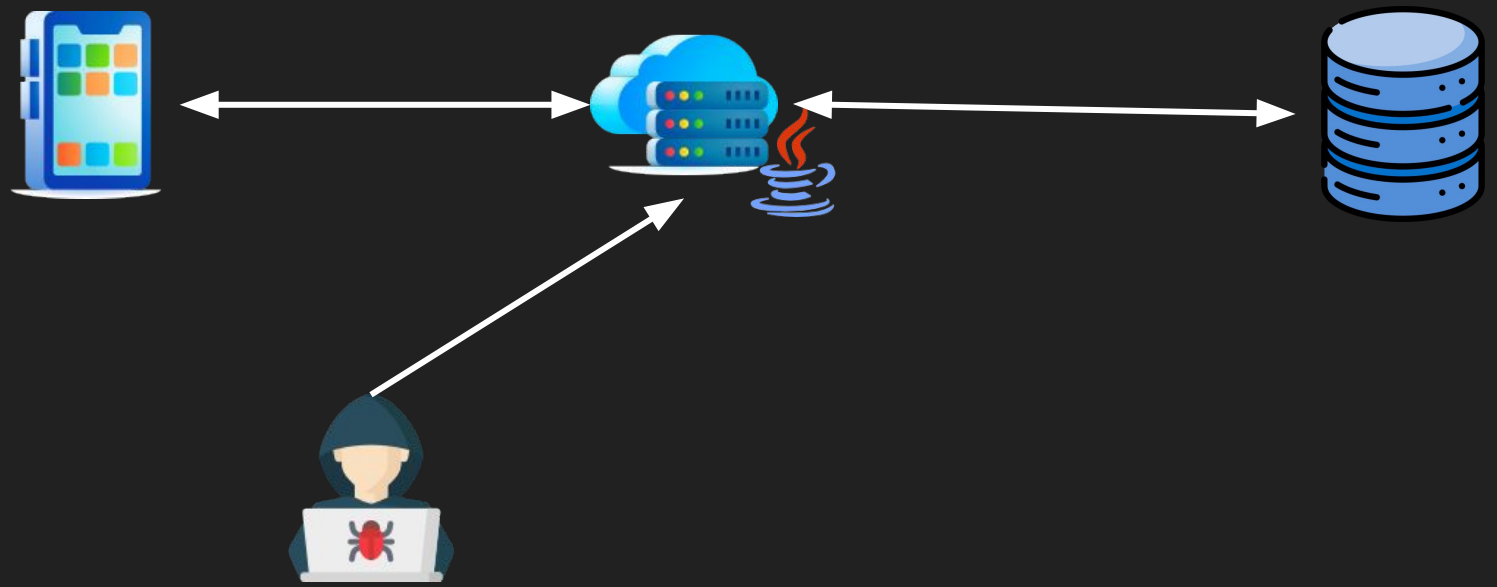
Por que usar:

- Feedback imediato e amigável para o usuário.
- Evita requisições desnecessárias ao servidor.
- Melhora performance e responsividade da aplicação.

Limitações:

- Pode ser burlada facilmente (por exemplo, via manipulação direta da requisição HTTP).
- Não deve ser usada sozinha para garantir integridade dos dados.

Arquitetura Cliente-Servidor - Servidor



Arquitetura Cliente-Servidor - Servidor

Regra de negócio:

Garantir que os dados enviados efetivamente respeitem as regras da aplicação, que podem ser mais complexas do que uma simples verificação de campo vazio.

São validações que asseguram a integridade, coerência e conformidade do negócio, abrangendo políticas comerciais, cálculos, restrições específicas, etapas sequenciais, autorização, etc.

Arquitetura Cliente-Servidor - Servidor

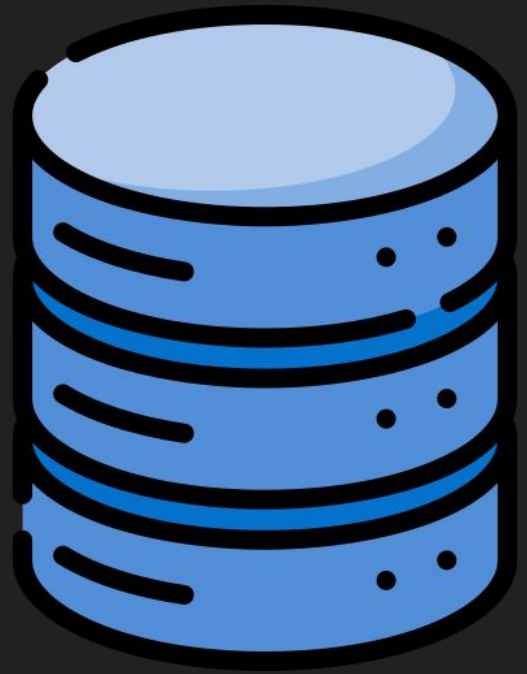
Arquitetura Cliente-Servidor - Servidor

Faz sentido ter validação no Banco de dados?

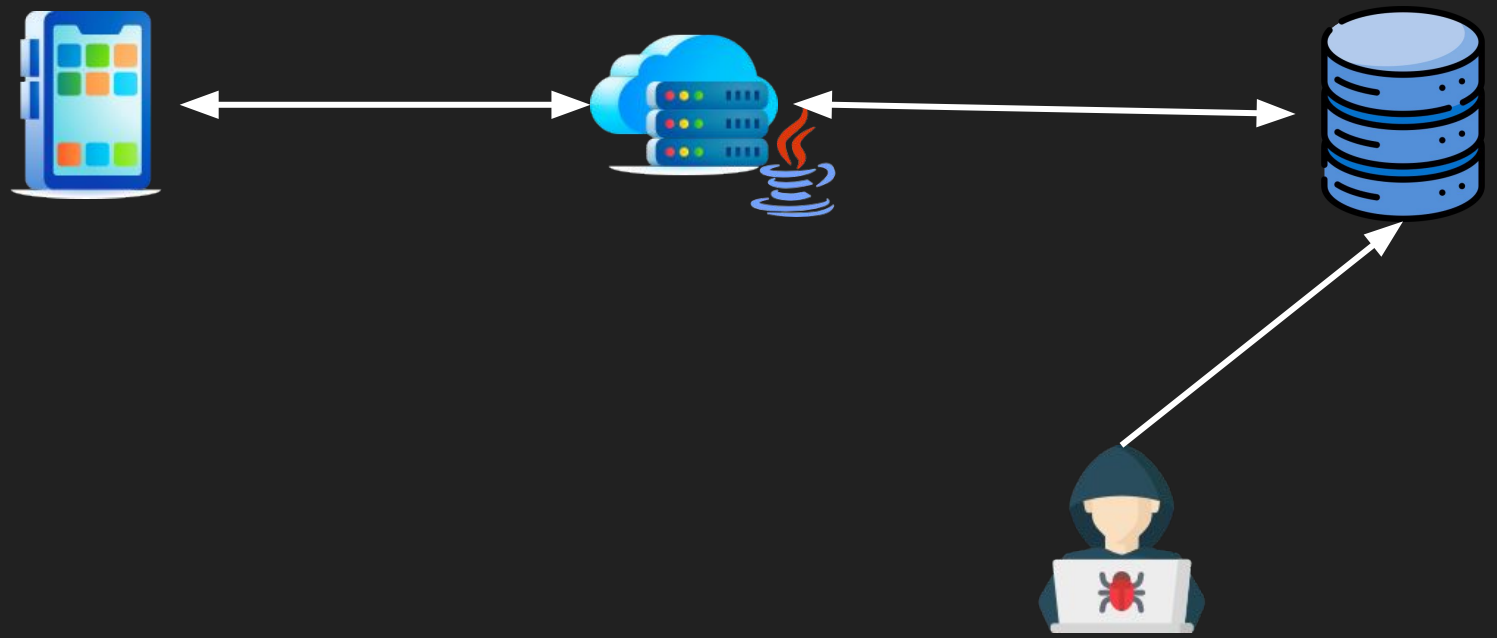
Serve como último escudo para garantir a integridade e consistência dos dados.

Restrições como chaves primárias, estrangeiras, unicidade, tipos de dados, triggers e constraints garantem que os dados inválidos não sejam armazenados.

Protege contra inconsistências causadas por erros ou falhas no servidor.



Arquitetura Cliente-Servidor - Servidor



Arquitetura Cliente-Servidor - Servidor

Faz sentido ter validação no Banco de dados?

Se o banco não tem validação de integridade (ex.: CHECK, FOREIGN KEY, NOT NULL, ENUM, etc.), um atacante ou até mesmo um usuário malicioso pode inserir dados inválidos.

- Data Integrity Attack (Ataque à integridade dos dados)
  - O atacante insere dados que não seguem as regras esperadas, corrompendo a base.
- Data Poisoning (Envenenamento de dados)
  - Mais usado em Machine Learning, mas também em bancos, quando o atacante insere dados inválidos ou maliciosos para atrapalhar relatórios, análises ou funcionamento.
- Business Logic Abuse (Abuso da lógica de negócio)
  - A falha não está no SQL em si, mas na lógica da aplicação que não valida corretamente os inputs antes de salvar.

Arquitetura Cliente-Servidor - Servidor

Desvantagens de Validação em várias camadas

Duplicação de regras: As mesmas regras de validação precisam ser implementadas e mantidas em diferentes camadas, o que pode gerar inconsistências se não forem perfeitamente sincronizadas.

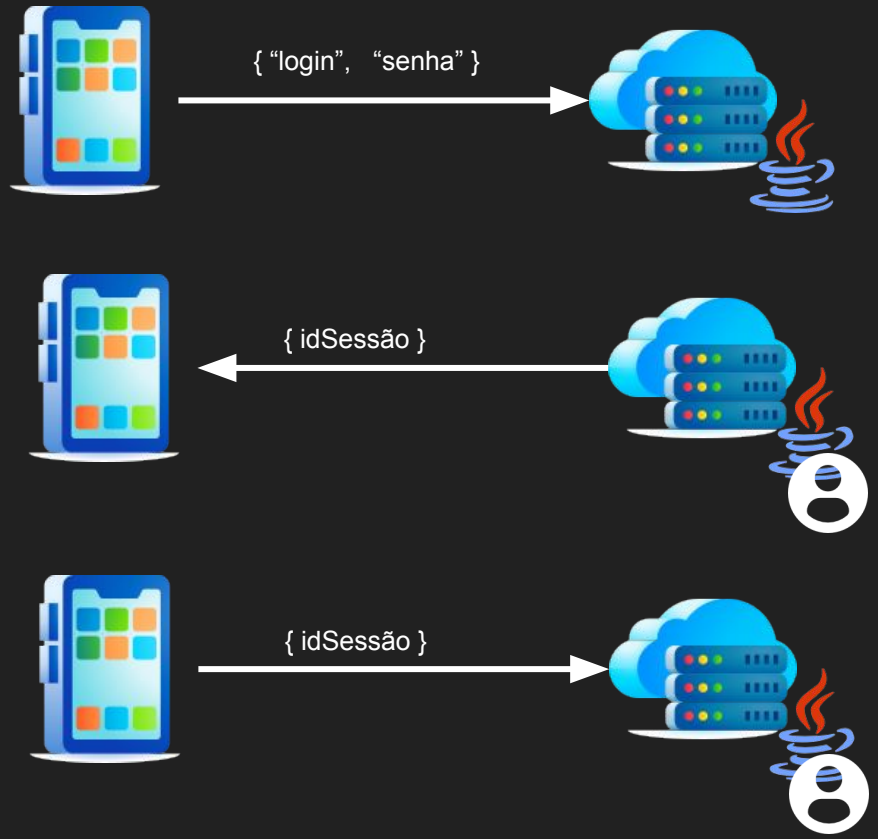
Maior trabalho de desenvolvimento: Cada camada precisa ter código para validar dados, aumentando o tempo e esforço para escrever e testar toda a lógica.

Risco de divergência: Se regras mudarem, existe o risco de serem atualizadas em apenas algumas camadas, causando comportamentos inesperados ou falhas na validação.

Complexidade na gestão de erros: Tratamento de erros pode se tornar mais complexo, com múltiplos pontos onde a validação pode falhar, dificultando rastreamento e diagnóstico de problemas.

Arquitetura Stateful

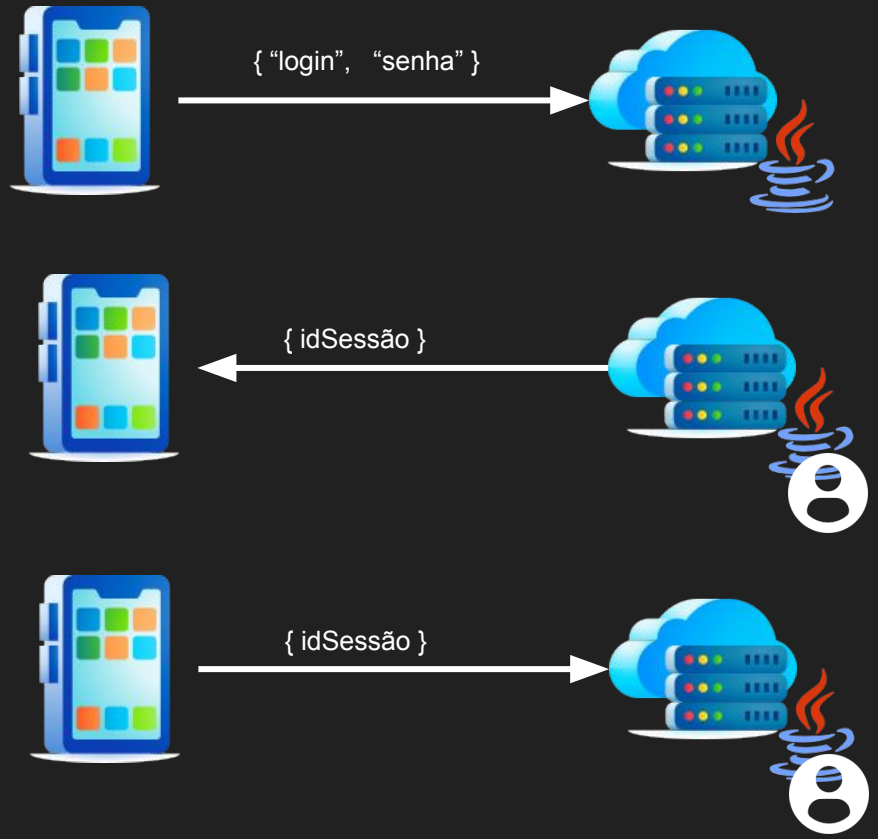
Em um servidor stateful (com estado), o funcionamento envolve o armazenamento do contexto da interação com o usuário entre as requisições.
Isso significa que o servidor mantém informações temporárias sobre as sessões dos usuários para gerenciar o estado da aplicação e da interação.



Arquitetura Stateful

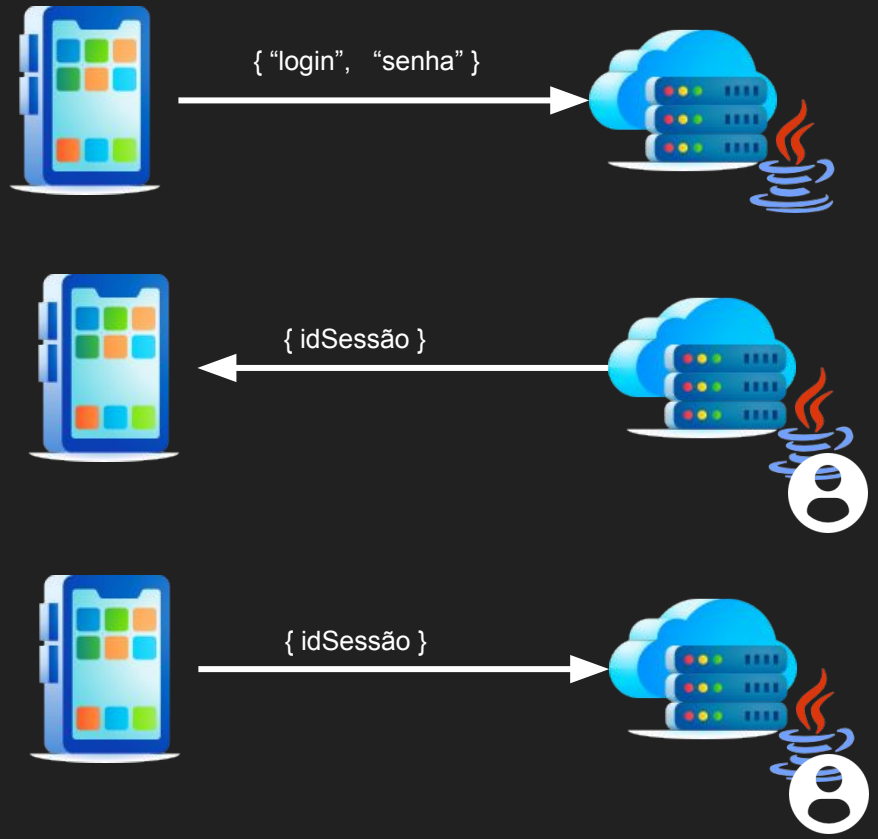
No servidor, esse ID aponta para uma estrutura de dados que guarda coisas como:

- ID do usuário autenticado
- Nome / dados básicos do perfil
- Permissões / papéis (ex.: admin, user)
- Tempo de expiração da sessão
- Dados temporários da aplicação (ex.: carrinho de compras, progresso de formulário, preferências)
- Possivelmente dados de cache específicos do usuário



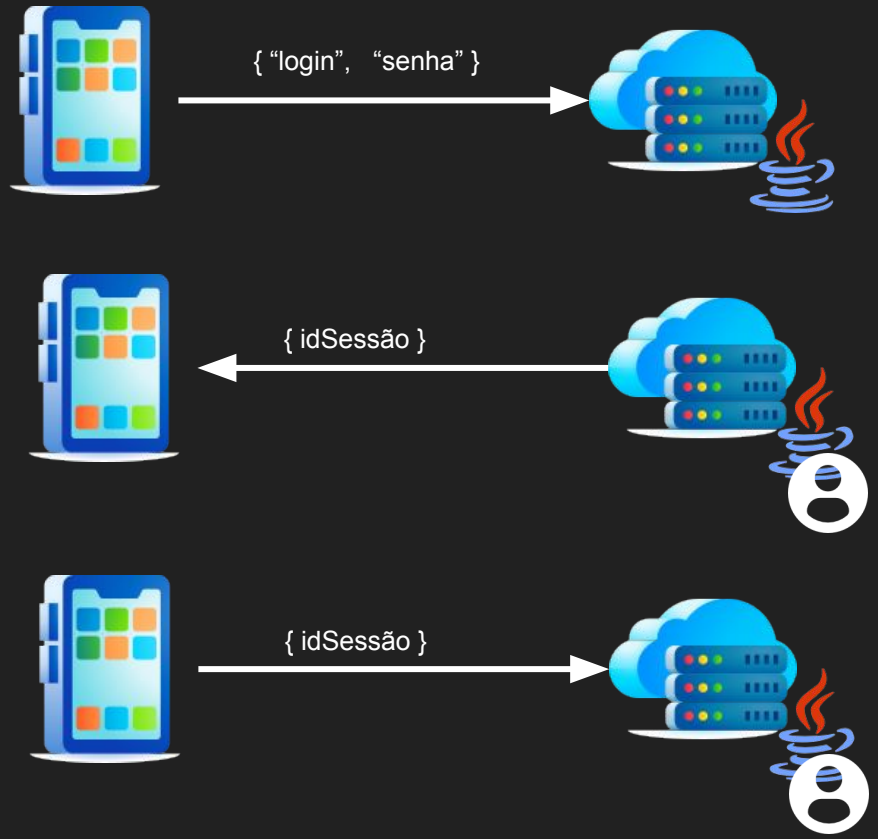
Arquitetura Stateful - Vantagens

- Gerenciamento de sessão simplificado
  - O servidor mantém diretamente o contexto do usuário → fácil acessar dados da sessão sem precisar passá-los em cada requisição.
- Menor sobrecarga no cliente
  - O cliente não precisa enviar todo o estado a cada requisição (como tokens ou dados de sessão complexos).
  - Reduz a complexidade de implementação do frontend.
- Controle centralizado de segurança e permissões
  - O servidor pode verificar roles, permissões e validade da sessão diretamente.
  - Facilita logout, expiração de sessão e invalidação de tokens.
- Possibilidade de cache de dados temporários
  - Informações frequentemente acessadas pelo usuário podem ser mantidas em memória, melhorando performance em operações repetidas.



Arquitetura Stateful - Desvantagens

- Escalabilidade mais difícil
  - Cada servidor precisa manter o estado de cada sessão.
  - Ao aumentar o número de usuários, você precisa replicar ou compartilhar estado entre servidores → aumenta complexidade.
- Recuperação em caso de falha
  - Se o servidor que mantém a sessão cair, o usuário perde o estado.
- Consumo de memória
  - Sessões ativas ficam em memória → limita quantidade de usuários simultâneos por servidor.
  - Dados grandes ou longas sessões podem consumir muita RAM.
- Menor flexibilidade para balanceamento de carga
  - Load balancers precisam garantir que o mesmo usuário vá sempre para o mesmo servidor (sticky sessions), ou replicar estado entre servidores → complexidade extra.

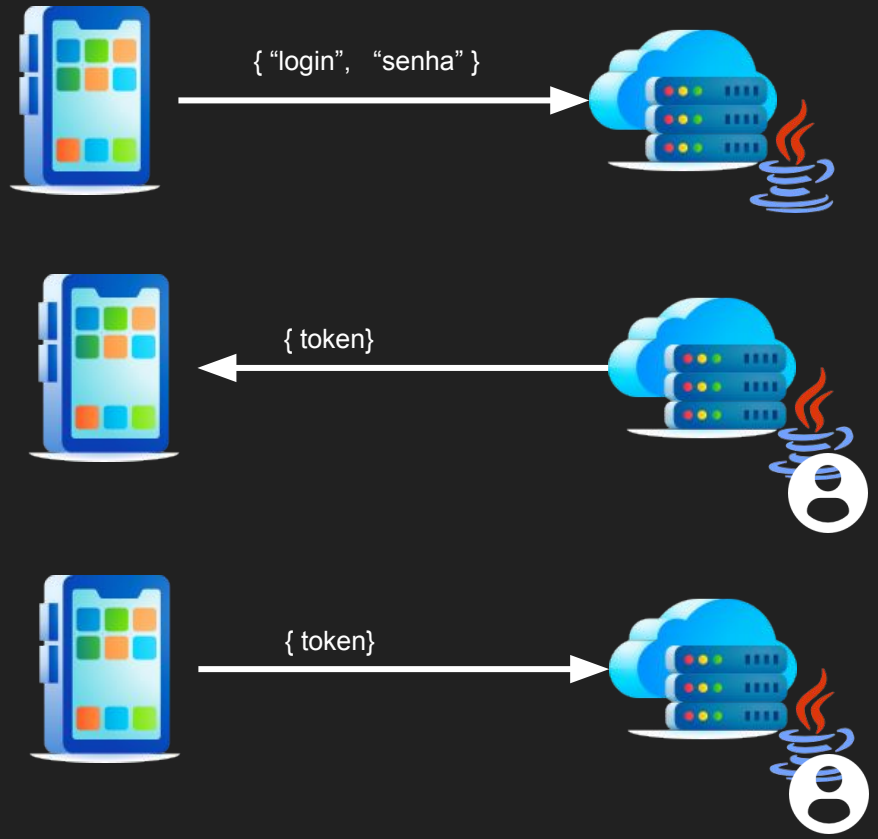


Arquitetura Stateless

Um servidor stateless é aquele que não guarda informações de contexto do cliente entre requisições.

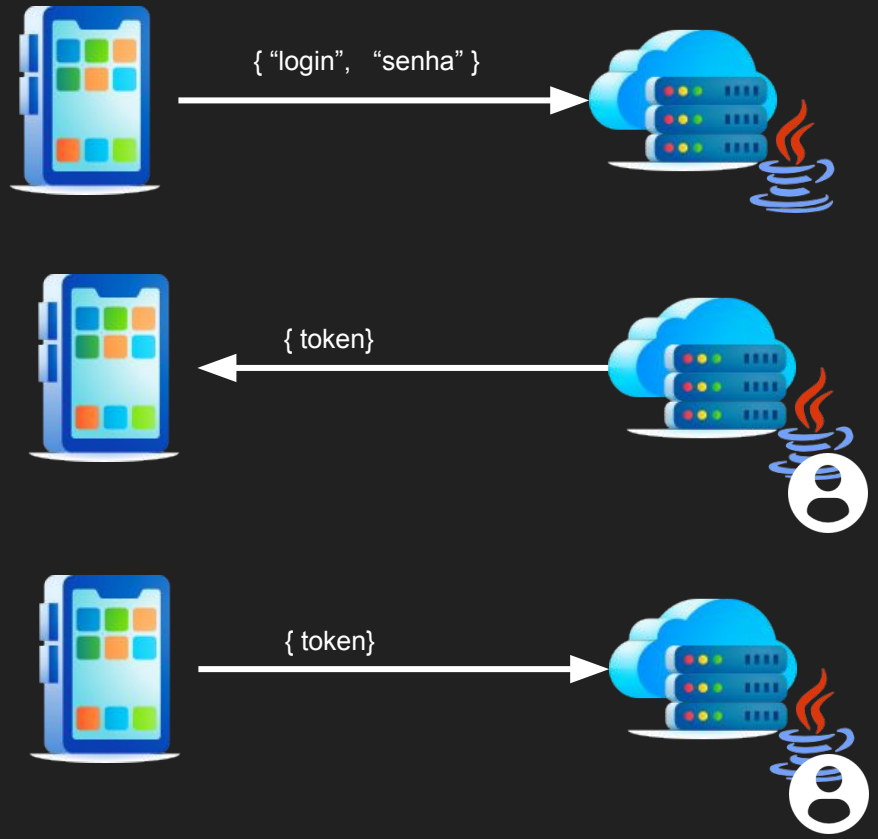
Ou seja: cada requisição é independente, como se fosse a primeira vez que o cliente está falando com o servidor.

Esse token contém informações do usuário (ex.: id, nome, permissões).



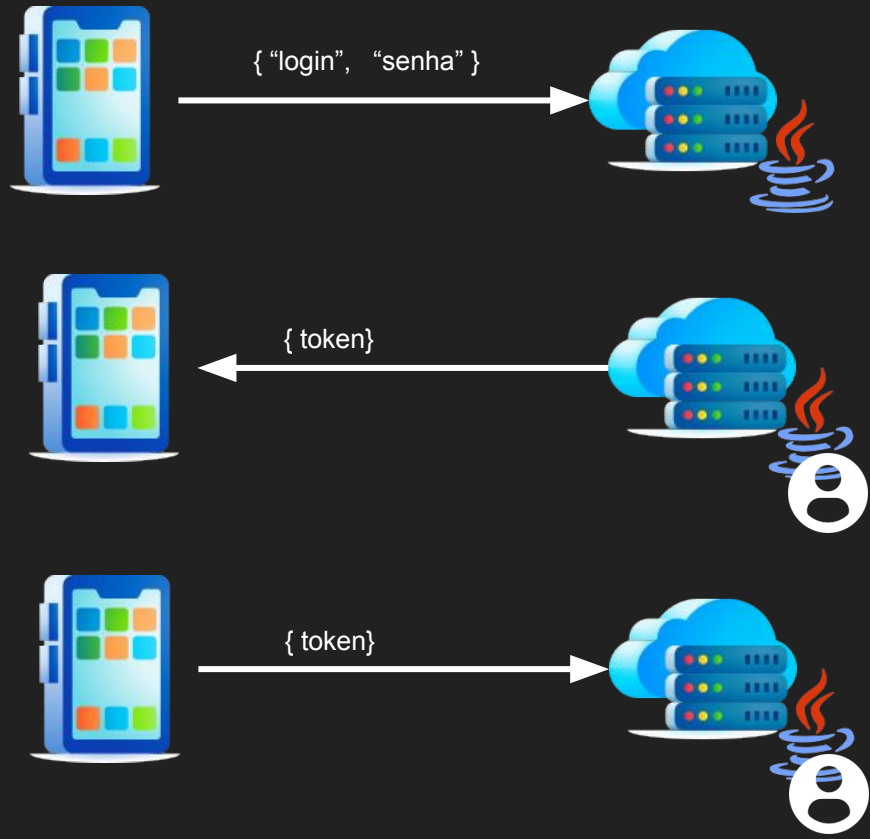
Arquitetura Stateless - Vantagens

- Escalabilidade fácil
  - Qualquer servidor pode atender qualquer requisição, porque não há estado local de sessão.
  - Deploy horizontal simples: adiciona servidores sem se preocupar com replicação de sessão.
- Tolerância a falhas
  - Se um servidor cair, o cliente pode ser atendido por outro sem perder sessão.
- Menor consumo de memória
  - Servidor não armazena dados temporários do usuário → RAM livre para processamento.
- Simplicidade de balanceamento de carga
  - Load balancer não precisa garantir “sticky session”.



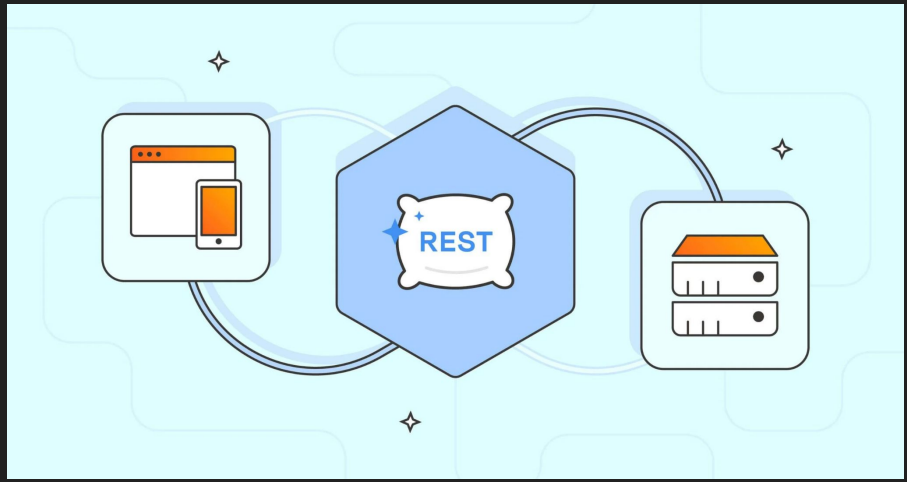
Arquitetura Stateless - Desvantagens

- Cada requisição precisa enviar todas as informações necessárias
  - Tokens grandes ou informações complexas podem aumentar overhead da rede.
- Segurança depende do token
  - Se o token for interceptado, pode ser usado até expirar.
  - Revogação de tokens exige estratégias adicionais, como listas de revogação ou expiração curta.
- Operações que dependem de estado
  - Ex.: carrinho de compras, progresso de formulários, sessão de jogo.
  - Precisam ser armazenadas no cliente ou em banco/cache compartilhado, aumentando complexidade.



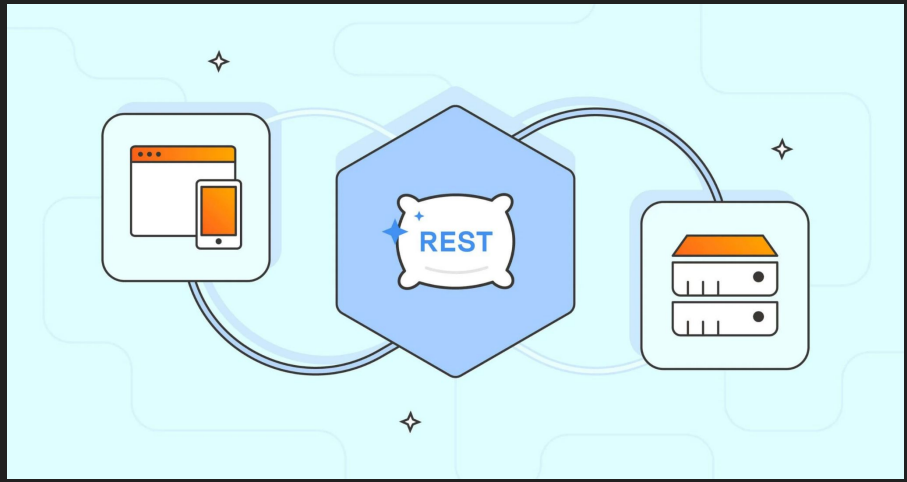
Arquitetura REST (Representational State Transfer)

A arquitetura REST (Representational State Transfer) é um estilo arquitetural bastante utilizado para o desenvolvimento de serviços Web, especialmente APIs que suportam comunicação entre sistemas distribuídos, como aplicações web, móveis e sistemas integrados.



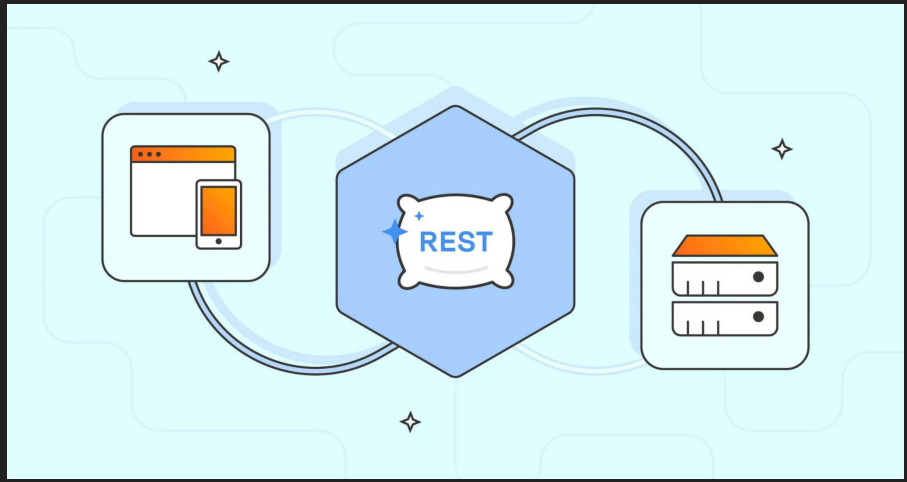
Arquitetura REST (Representational State Transfer)

REST (Representational State Transfer) é um conjunto de princípios e restrições para criar serviços Web que sejam leves, escaláveis e fáceis de manter. Ela se baseia no protocolo HTTP para definir como os recursos (dados ou serviços) são identificados, manipulados e representados.



Arquitetura REST (Representational State Transfer)

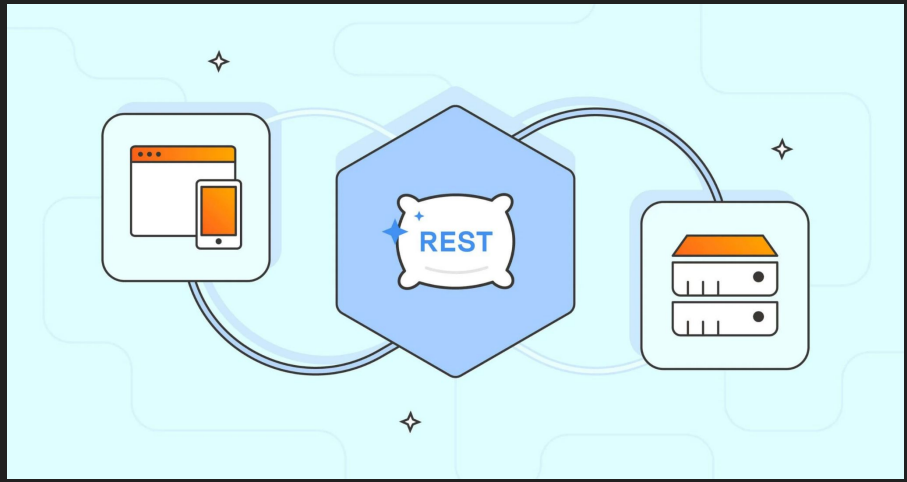
Foi criada por Roy Fielding em sua tese de doutorado, publicada em 2000 na Universidade da Califórnia, Irvine. Fielding foi um dos principais autores do protocolo HTTP e seu objetivo era definir um conjunto de diretrizes para a construção de sistemas distribuídos, especialmente para a Web, baseados em padrões abertos, simples e interoperáveis.



Arquitetura REST (Representational State Transfer)

Evolução da arquitetura REST

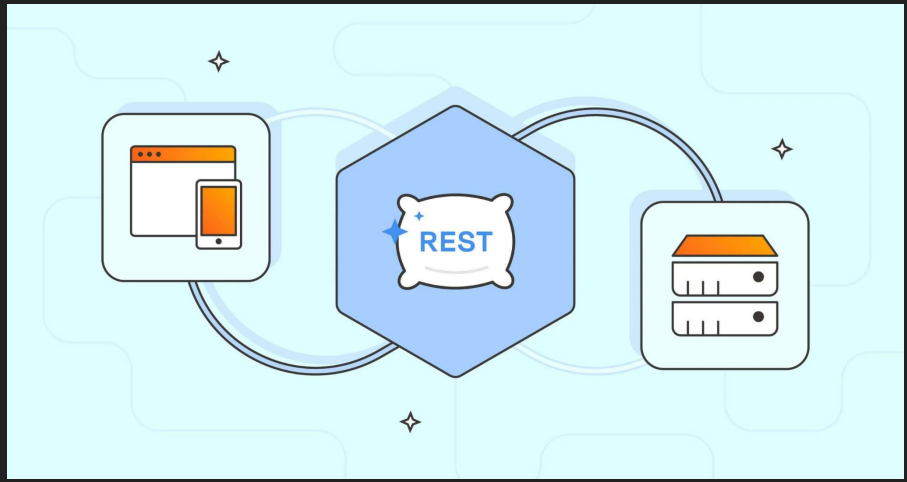
- **2000**: REST definido por Roy Fielding, influenciando diretamente versões modernas do HTTP e URI.
- **2000-2010**: APIs REST começam a substituir SOAP e outras alternativas, acelerando o desenvolvimento de aplicações distribuídas e Web 2.0.
- **2010 em diante**: REST se torna o padrão de mercado para APIs, usando principalmente JSON como formato principal de dados, impulsionando microsserviços, mobile apps e sistemas cloud.



Arquitetura REST (Representational State Transfer)

Antes do REST, grande parte das integrações entre sistemas eram complexas, baseadas em protocolos pesados como SOAP, ou modelos proprietários que dificultavam a interoperabilidade. Essas abordagens apresentavam problemas como:

- **Dificuldade de escalabilidade**: Servidores precisavam manter estados de conexão ou sessões complexas.
- **Alto acoplamento**: APIs baseadas em RPC ou SOAP tinham contratos rígidos e difícil adaptação.
- **Baixa interoperabilidade**: Sistemas de diferentes tecnologias tinham dificuldade de comunicação.
- **Overhead**: Uso intenso de XML, envelopes e padrões complexos aumentava o custo computacional e a dificuldade para desenvolvimento.



Arquitetura REST (Representational State Transfer)

Princípios fundamentais da arquitetura REST

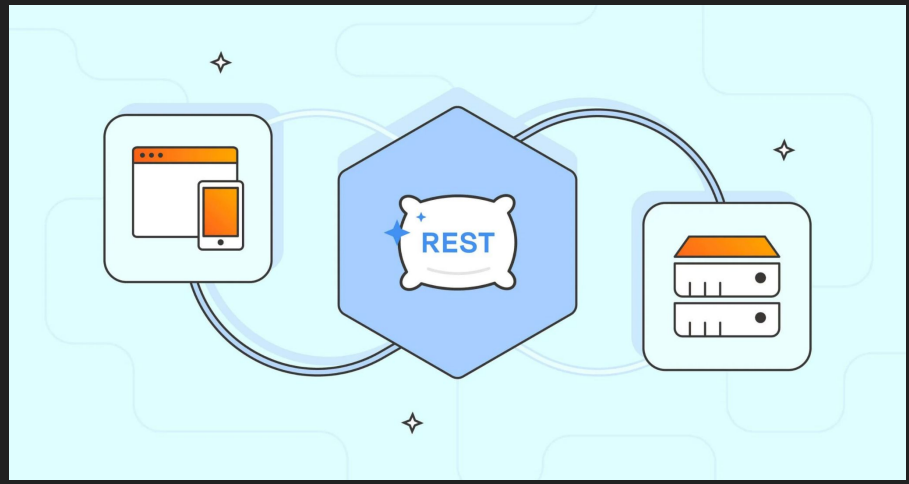
Recursos identificados por URIs: Cada recurso (exemplo: usuário, pedido, produto) é identificado de forma única por uma URL.

Operações via métodos HTTP:

- GET: para recuperar recursos.
- POST: para criar novos recursos.
- PUT: para atualizar recursos.
- DELETE: para remover recursos.

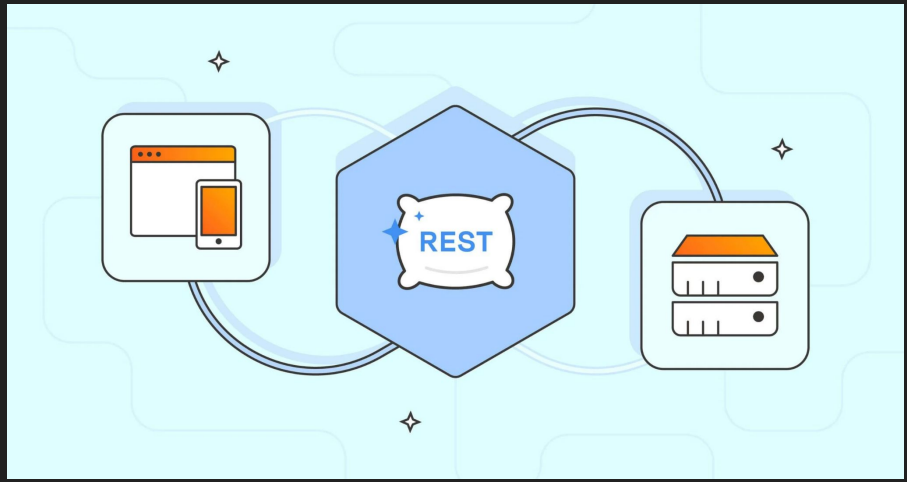
Stateless (Sem estado): Cada requisição enviada ao servidor deve conter todas as informações necessárias para ser entendida, ou seja, o servidor não deve manter informações da sessão entre requisições.

Representações dos recursos: Os recursos podem ser representados em diferentes formatos, como JSON, XML, HTML etc., sendo o JSON o mais comum atualmente.



Arquitetura REST (Representational State Transfer)

Princípios fundamentais da arquitetura REST



Operações via métodos HTTP:

- GET: para recuperar recursos.
  - /api/produtos → Todos os produtos
  - /api/produtos/10 → Produto específico
- POST: para criar novos recursos.
  - /api/produtos
- PUT: para atualizar recursos.
  - /api/produtos/10 → atualiza
- DELETE: para remover recursos.
  - /api/produtos/10 → remove

Arquitetura REST (Representational State Transfer)

Princípios fundamentais da arquitetura REST

Operações via métodos HTTP:

- GET: para recuperar recursos.
  - /api/produtos → Todos os produtos
  - /api/produtos/10 → Produto específico
- POST: para criar novos recursos.
  - /api/produtos
- PUT: para atualizar recursos.
  - /api/produtos/10 → atualiza
- DELETE: para remover recursos.
  - /api/produtos/10 → remove

```
@GetMapping("")  no usages
Produto getProdutos(){
    // Produtos
}

@GetMapping("/{id}")  no usages
Produto getProduto(@PathParam("id") int id){
    // Produtos
}

@PostMapping("")  no usages
Produto getProdutos(@RequestBody Produto produto){
    // Produtos
}

@PutMapping("")  no usages
Produto getProdutos(@RequestBody Produto produto){
    // Produtos
}

@DeleteMapping("/{id}")  no usages
Produto getProdutos(@PathParam("id") int id){
    // Produtos
}
```

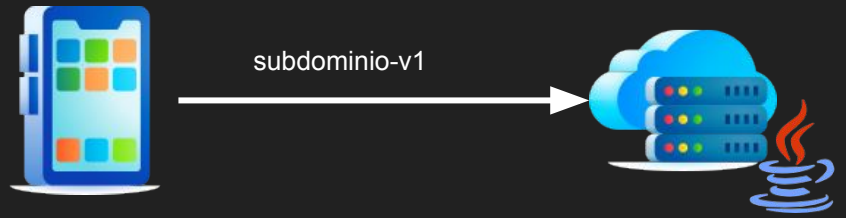
Arquitetura REST (Representational State Transfer)

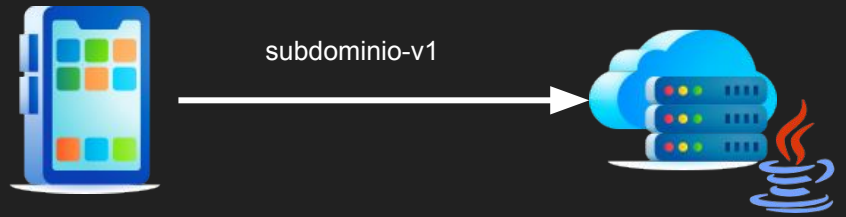
Versionamento da API

Sempre pense em versionamento, para não quebrar clientes.

subdominio-v1.com/api/produtos → Todos os produtos

subdominio-v2.com/api/produtos → Todos os produtos





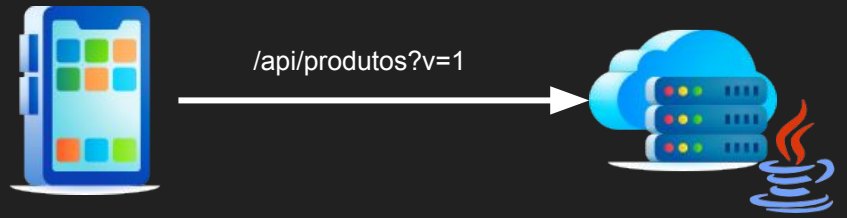
Arquitetura REST (Representational State Transfer)

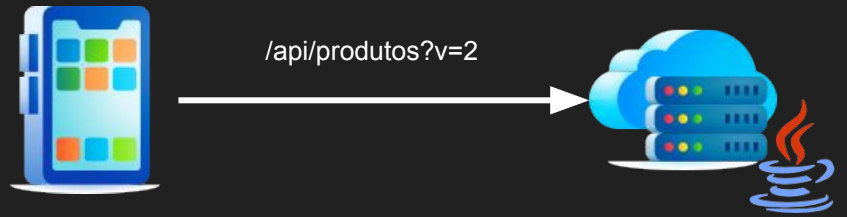
Versionamento da API

Sempre pense em versionamento, para não quebrar clientes.

/api/produtos?v=1 → Todos os produtos

/api/produtos?v=2 → Todos os produtos





Arquitetura REST (Representational State Transfer)

Versionamento da API

Sempre pense em versionamento, para não quebrar clientes.


```
GET /produtos HTTP/1.1
Host: api.meusite.com
Accept: application/vnd.meusite.v2+json
```

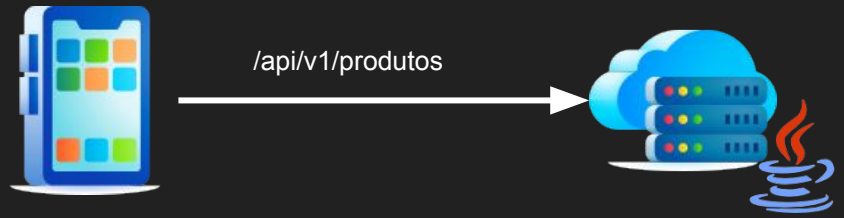
Arquitetura REST (Representational State Transfer)

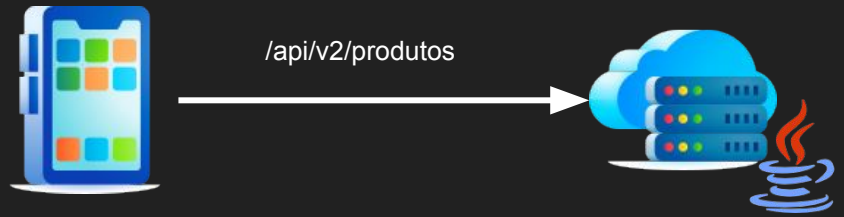
Versionamento da API

Sempre pense em versionamento, para não quebrar clientes.

/api/v1/produtos → Todos os produtos

/api/v2/produtos → Todos os produtos

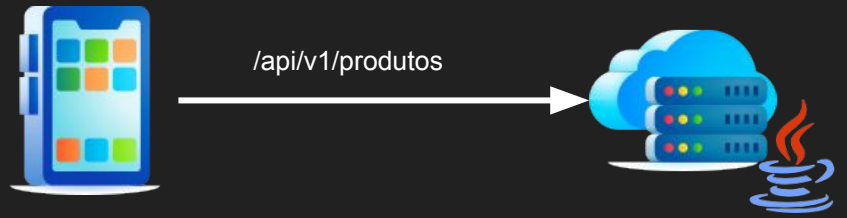


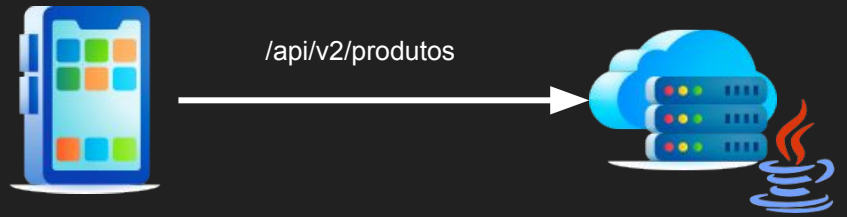


Arquitetura REST (Representational State Transfer)

Padrões de Resposta

- Formato
  - Json
  - XML
- Status HTTP corretos:
  - Criação de Recurso
    - 200 OK → sucesso
    - 201 Created → recurso criado
  - Erro do cliente
    - 404 Not Found
  - recurso não existe
    - 404 Not Found
    - 400 Bad Request
  - Erro do servidor
    - 500 Internal Server Error

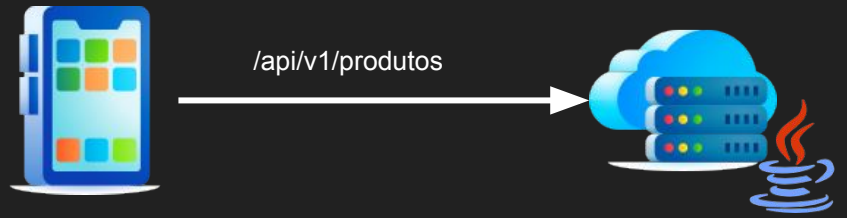


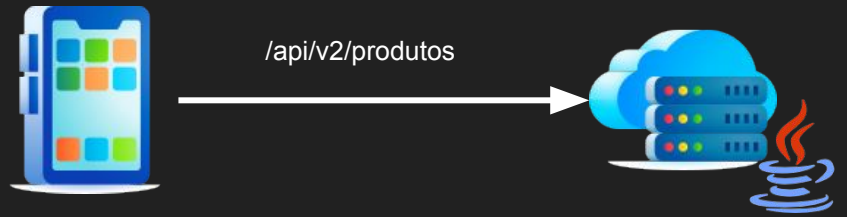


Arquitetura REST (Representational State Transfer)

Padrões de Resposta

- Filtros, buscas e pagianação (GET /produtos?categoria=eletronicos&precoMax=2000
- Ou manda um Post com os dados no corpo





Níveis do Rest

Ele descreve níveis de evolução de uma API até chegar ao que podemos chamar de REST "de verdade".

- Nível 0 – The Swamp of POX (pântano do XML/JSON simples)
- Nível 1 – Recursos
  - Começa a separar os recursos em URLs diferentes e Cada entidade ganha seu próprio endpoint (/clientes, /produtos, etc.).
- Nível 2 – Verbos HTTP
  - Além de separar os recursos e usar os status codes do HTTP, começa a usar os verbos HTTP da forma correta:
  - GET → consultar
  - POST → criar
  - PUT/PATCH → atualizar
  - DELETE → excluir

Níveis do Rest

- Nível 3 – HATEOAS (Hypermedia As The Engine Of Application State)
  - A API passa a retornar links e ações possíveis no recurso, orientando o cliente sobre o que pode fazer.
  - A resposta da API contém hipermídia que guia o consumo, como na navegação da web.

```
GET /clientes/123
Response:
{
   "id": 123,
   "nome": "Maria",
   "links": [
      { "rel": "self", "href": "/clientes/123" },
      { "rel": "pedidos", "href": "/clientes/123/pedidos" },
      { "rel": "atualizar", "href": "/clientes/123", "method": "PUT" }
   ]
}
```

Arquitetura REST (Representational State Transfer)

Vantagens da arquitetura REST

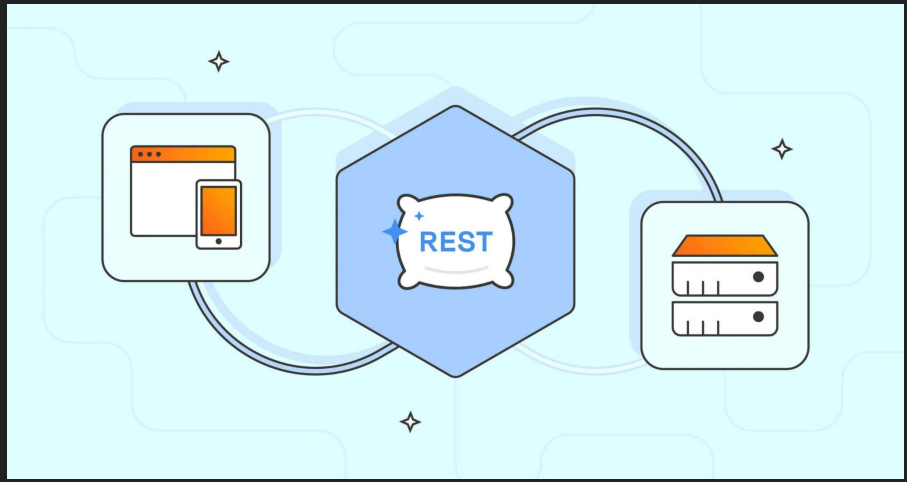
Simplicidade e padronização: Utiliza padrões amplamente conhecidos e protocolos HTTP padrão.

Escalabilidade: Por ser stateless, facilita o balanceamento de carga e a escalabilidade horizontal.

Flexibilidade: Pode servir diferentes clientes (Web, mobile, IoT, etc.) usando o mesmo serviço API.

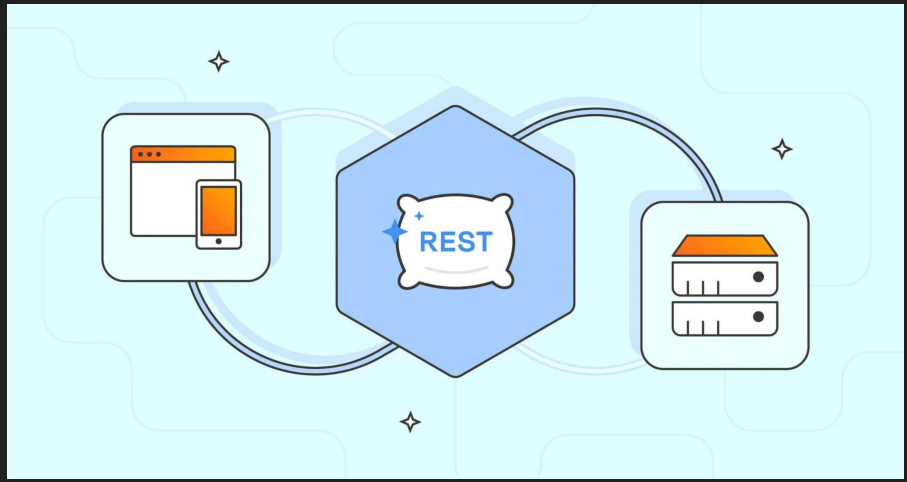
Facilidade de integração: Permite integração fácil entre sistemas heterogêneos via web.

Cacheabilidade: Respostas podem ser explicitamente marcadas como cacheáveis para melhorar desempenho.



E os problemas?

E quais as desvantagens de se utilizar Rest?



Arquitetura REST (Representational State Transfer)

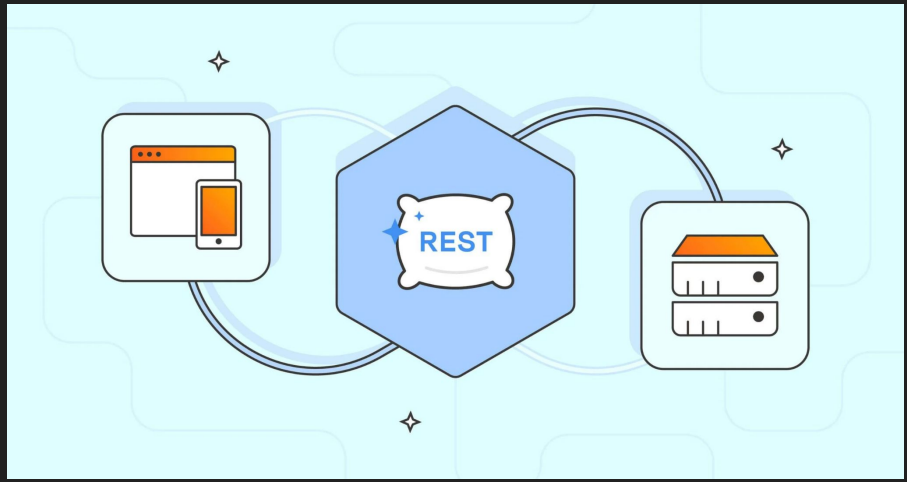
Desvantagens e pontos de atenção

Sem estado pode ser limitante: Para algumas aplicações complexas, manter contexto entre requisições pode ser um desafio.

Latência: Cada operação é uma requisição HTTP independente, o que pode gerar maior latência em alguns casos.

Projetar APIs RESTful boas exige cuidado: Escolher bem os recursos, endpoints e o uso correto dos métodos é essencial para manter coerência e usabilidade.

Segurança: Deve ser implementada adequadamente, já que REST por si só não define mecanismos.

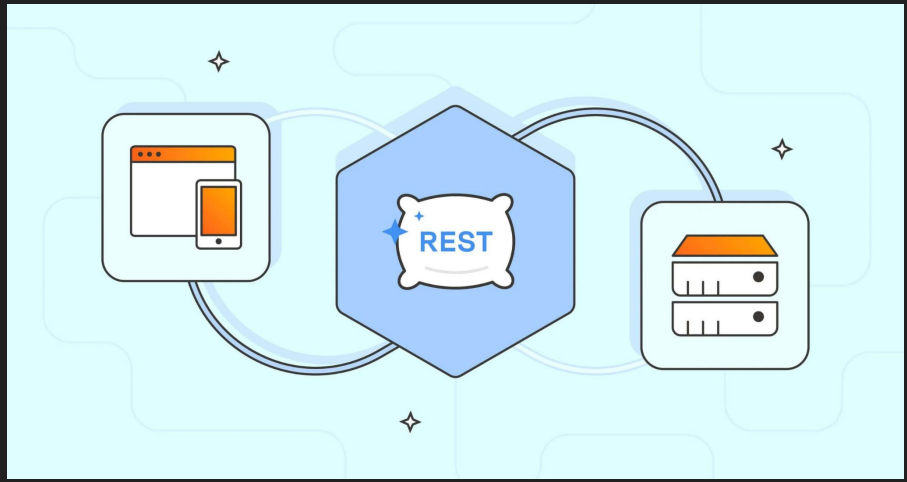


Arquitetura REST (Representational State Transfer)

Desvantagens e pontos de atenção

Sem estado pode ser limitante: Para algumas aplicações complexas, manter contexto entre requisições pode ser um desafio.

Exemplo: Um sistema ERP que precisa garantir que várias operações relacionadas à um pedido sejam concluídas com sucesso ou revertidas todas juntas.

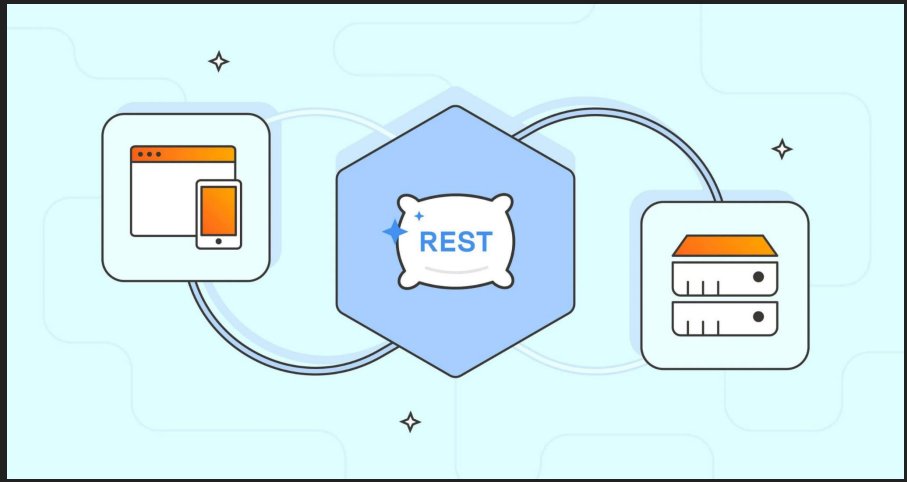


Arquitetura REST (Representational State Transfer)

Desvantagens e pontos de atenção

Sem estado pode ser limitante: Para algumas aplicações complexas, manter contexto entre requisições pode ser um desafio.

Exemplo: Um sistema ERP que precisa garantir que várias operações relacionadas à um pedido sejam concluídas com sucesso ou revertidas todas juntas.



Arquitetura REST (Representational State Transfer)

Desvantagens e pontos de atenção

Exemplo: No REST você recebe todos os campos do recurso, mesmo que precise só de alguns.
Se o cliente só precisa do `nome` e `preco`, REST ainda manda tudo.

```
{
  "id": 1,
  "nome": "Notebook Gamer",
  "preco": 5500.0,
  "descricao": "Notebook com RTX 4060",
  "estoque": 10,
  "fabricante": {
    "id": 5,
    "nome": "Dell",
    "pais": "EUA"
  },
  "avaliacoes": [
    { "usuario": "João", "nota": 5, "comentario": "Ótimo!" },
    ...
  ]
}
```

Arquitetura REST (Representational State Transfer)

Desvantagens e pontos de atenção

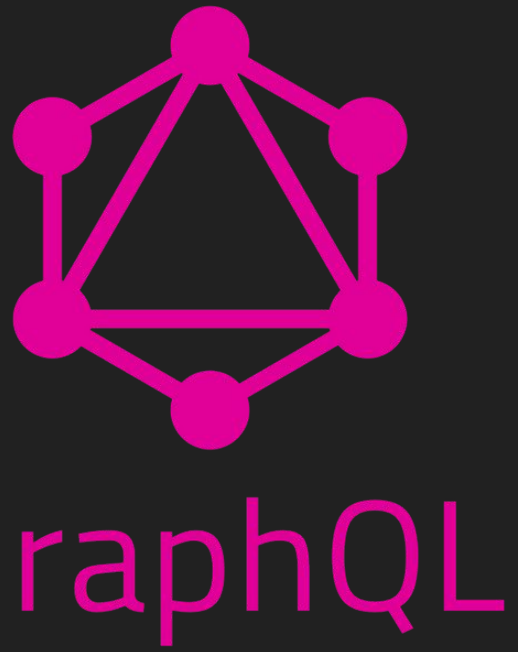
Exemplo: REST muitas vezes exige várias requisições para compor todos os dados necessários.

```
GET /api/produtos/1
GET /api/fabricantes/5
GET /api/produtos/1/avaliacoes
```

Alternativas ao REST

Novas abordagens para desafios: Surgimento de GraphQL, gRPC para contextos específicos, mas REST permanece dominante graças à sua simplicidade e compatibilidade.

GraphQL é uma linguagem de consulta para APIs e um runtime para executar essas consultas, que permite aos clientes solicitar exatamente os dados que precisam, nem mais nem menos. Foi criada pelo Facebook em 2015 para superar limitações das APIs REST tradicionais.



Alternativas ao REST

GraphQL retorna exatamente o que você pediu.

graphql

```graphql
{
  produtoPorId(id: 1) {
    nome
    preco
  }
}
```

Resposta:

json

```json
{
  "data": {
    "produtoPorId": {
      "nome": "Notebook Gamer",
      "preco": 5500.0
    }
  }
}
```

Arquitetura REST (Representational State Transfer)

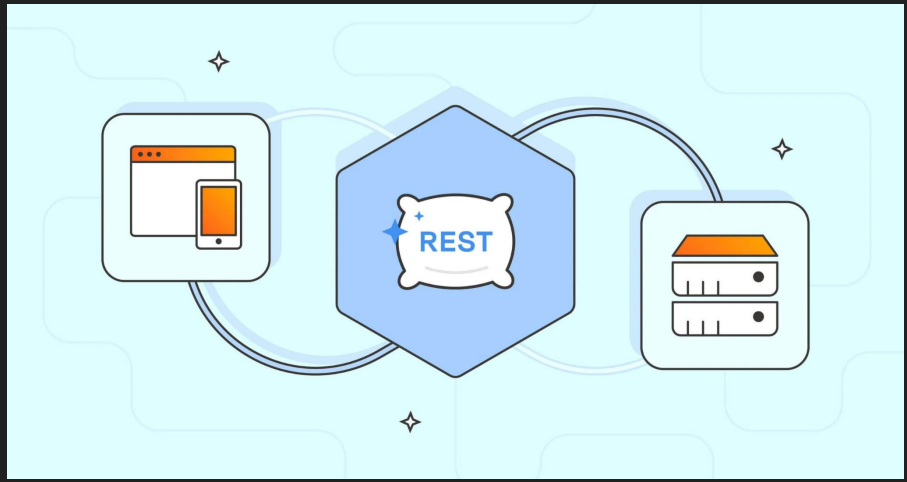
Desvantagens e pontos de atenção

Exemplo: Uma única query traz tudo de uma vez.

```
{
  produtoPorId(id: 1) {
    nome
    preco
    descricao
    fabricante {
      nome
      pais
    }
    avaliacoes {
      usuario
      nota
    }
  }
}
```

Quando usar REST

Simplicidade importa mais que flexibilidade (CRUD direto: criar, listar, editar, deletar).

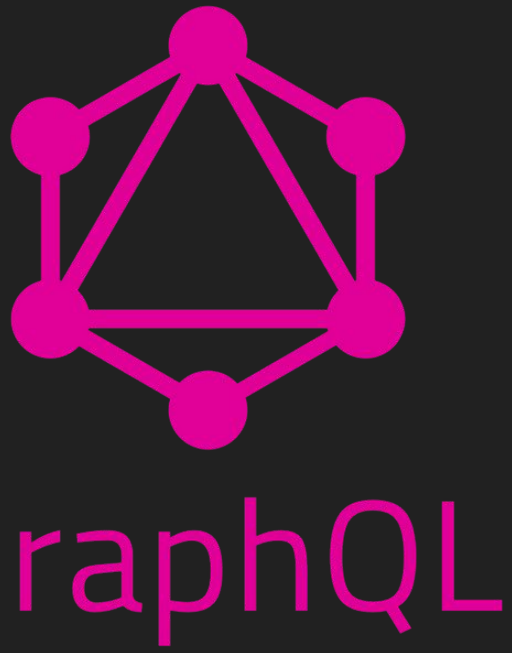


- Você tem poucos relacionamentos entre entidades (ex: sistema de autenticação, cadastro simples).
- A API será consumida por clientes controlados (você desenvolve o front e o back, então sabe exatamente o que precisa).
- O tráfego e payload não são um problema (mandar alguns dados extras não impacta muito).
- Você quer cache fácil: REST funciona muito bem com cache HTTP (ETag, Cache-Control, CDN).

Quando usar GraphQL

Precisa evitar overfetching e underfetching (o cliente quer dados bem específicos).

- Tem múltiplos clientes diferentes (web, mobile, BI, etc.) consumindo a mesma API, cada um precisando de coisas distintas.
- Vai lidar com relacionamentos complexos (usuário → pedidos → produtos → avaliações).
- Quer evoluir a API sem quebrar clientes antigos (evita versionamento REST).
- Precisa reduzir a quantidade de requisições (muito útil em mobile, onde cada request pesa).
- Trabalha em cenários de front-end dinâmico (o front decide em runtime o que precisa).



Quando usar REST e GraphQL

Alguém usou as duas? Qual a sua opinião?

Indo Além

- Limite de requisições
- Documentação
- Ambiente de Homologação
- Paginação
- Filtros
- Cache
- Monitoramento
- Autenticação
- Timeouts
- Teste
- Opção de exportação

Aula 04

Introdução ao Java e Spring Framework

Java

Java é uma linguagem de programação e plataforma de computação liberada pela primeira vez pela Sun Microsystems em 1995.
De um início humilde, ela evoluiu para uma grande participação no mundo digital dos dias atuais, oferecendo a plataforma confiável na qual muitos serviços e aplicativos são desenvolvidos.
Produtos e serviços novos e inovadores projetados para o futuro continuam a confiar no Java também.

O que eu preciso para criar uma aplicação?

JDK (https://jdk.java.net) - O **JDK** (Java Development Kit) é um conjunto de ferramentas essenciais para o desenvolvimento de aplicações em Java. Ele é o "kit de ferramentas" que você usa para **escrever, compilar, depurar e executar programas Java**.

Linux: Dependendo da versão, será necessário buscar algum repositório

- sudo apt install openjdk-17-jdk

Para testar:

java -version

javac -version

Oracle, Open e default JDK

Default - Versão padrão para sua distribuição linux (normalmente Open JDK)

Oracle - A Oracle mudou sua política de licenciamento a partir de 2019, sendo distribuído sob uma licença comercial. Isso significa que o Oracle JDK não é mais gratuito para uso comercial (a menos que você use a versão de long-term support ou LTS sem atualizações de segurança), e você pode precisar pagar pelo suporte e atualizações se for usá-lo em ambientes corporativos.

Open - mantido pela comunidade de código aberto e é distribuído sob a Licença Pública Geral (GPL), o que significa que qualquer pessoa pode modificar, redistribuir ou usar o OpenJDK sem custos.

IDE

1. **Injelij IDE**
2. **Netbenas**
3. **Eclipse**
4. **Vs Code**
    - **Extension Pack for Java**

Tudo em Java é classe

1. Todos os arquivos java (.java) tem uma classe de mesmo nome do arquivo.
2. O método **`main`** é o ponto de entrada de qualquer programa Java que será executado

```java
public class ExemploMain {
    public static void main(String[] args) {
        System.out.println("Olá, mundo!");
    }
}
```

Criando variáveis

Uma variável é um espaço na memória que armazena um valor e possui:

1. Modificador de acesso → Define quem pode acessar.
2. Tipo → define que tipo de dado ela pode guardar (int, double, String, etc.).
3. Nome → usado para acessar o valor.
4. Valor → conteúdo armazenado.

[modificador] tipo nome [= valor];

Criando variáveis

```
tipo nomeDaVariavel;
```

```
tipo nomeDaVariavel = valor;
```

Java é uma linguagem fortemente tipada, ou seja, é obrigatório definir um tipo para as variáveis.

```java
// Variável inteira
int idade = 25;

// Variável decimal
double altura = 1.75;

// Variável texto
String nome = "Maria";

// Variável booleana
boolean ativo = true;
```

Tipos em java

int	Representa números inteiros de -128 a 127.	byte b = 100;
long	Representa números inteiros de -2^{63} a $2^{63}-1$.	long l = 100000000L;
float*	Representa números de ponto flutuante com precisão simples (aproximadamente 6-7 dígitos).	float f = 3.14f;
double	Representa números de ponto flutuante com precisão dupla (aproximadamente 15-16 dígitos).	double d = 3.14159;
boolean	Representa um valor lógico, pode ser true ou false.	boolean b = true;
String	Um tipo especial em Java para representar sequências de caracteres	String s = “Qualquer texto”;

Criando Array

Um array é uma estrutura de dados que armazena vários valores do mesmo tipo em posições (índices) numeradas, começando do 0.

```
tipo[] nomeDaVariavel;

tipo nomeDaVariavel =  new tipo[tamanho];

tipo[] nomeDoArray = {valor1, valor2, valor3};
```

```
int[] meuArray;

int[] meuArray = new int[10];

int[] meuArray = {50, 80, 30};
```

Criando Matriz

Uma matriz é um Array com 2 dimensões que armazenam vários valores do mesmo tipo em posições (índices) numeradas, começando do 0.

```java
int[][] matriz;

int[][] matriz = new int[3][5]; // 3 linhas e 5 colunas

int[][] matriz = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};

int[][] matriz = new int[3][]; // só declara as linhas
matriz[0] = new int[2];        // primeira linha com 2 colunas
matriz[1] = new int[3];        // segunda linha com 3 colunas
matriz[2] = new int[1];        // terceira linha com 1 coluna
```

Nomear variáveis - Restrições

1. Não pode começar com número
2. Não pode conter espaços ou caracteres especiais como @, #, -, %, etc.
3. Não pode ser uma palavra reservada da linguagem (ex.: class, if, while, public, etc.).

Java é case-sensitive: `nome` e `Nome` são variáveis diferentes

Boas Práticas para nomear variáveis

1. **Usar camelCase para variáveis**: nomeCompleto, precoProduto.
2. **Nomes claros e descritivos**: contador, quantidadeEstoque.
3. **Evitar uso de _ e $ no início**: a menos que tenha motivo técnico.
4. **Seguir padrão de idioma do projeto:** tudo em português ou tudo em inglês, não misturar.
    - Dê preferência por utilizar o nome em linguagem nativa para Entidade do core do sistema

Operadores

Operador	Significado	Exemplo (`int a=5, b=2`)	Resultado
`+`	Soma	`a + b`	`7`
`-`	Subtração	`a - b`	`3`
`*`	Multiplicação	`a * b`	`10`
`/`	Divisão inteira	`a / b`	`2`
`%`	Módulo (resto da divisão)	`a % b`	`1`

Operador	Significado	Exemplo	Resultado
`++`	Incremento	`a++` ou `++a`	+1
`--`	Decremento	`a--` ou `--a`	-1

Diferença:

- **Pré** (++a): incrementa antes de usar o valor.
- **Pós** (a++): usa o valor e só depois incrementa.

```
int a = 5;
System.out.println(++a); // 6 (pré-incremento)
System.out.println(a++); // 6 (mostra, depois vira 7)
```

Operador	Significado	Exemplo (a=5 , b=3)	Resultado
`==`	Igual a	`a == b`	false
`!=`	Diferente de	`a != b`	true
`>`	Maior que	`a > b`	true
`<`	Menor que	`a < b`	false
`>=`	Maior ou igual a	`a >= b`	true
`<=`	Menor ou igual a	`a <= b`	false

Operadores

Operador	Significado	Exemplo	Resultado
`&&`	E (AND)	`(a > 0 && b > 0)`	true se ambos forem verdadeiros
`||`	Ou (OR)	`(a > 0 ||b > 0)`	true se um deles for verdadeiro
`!`	Negação	`!(a > 0)`	Inverte o valor lógico true -> false false -> true

Estrutura de Decisão - IF

Usado para executar um bloco de código apenas se uma condição for verdadeira.

```
if (idade >= 18) {
    System.out.println("Você é maior de idade.");
}
```

Estrutura de Decisão - IF

Diferente de outras linguagens, Java não permite usar apenas a variavel para comparação, diferente de outras linguagens

```java
int numero;

if (numero) {
    System.out.println("Tem número");
}
```

Estrutura de Decisão - IF..ELSE

Permite executar um bloco caso a condição seja verdadeira, e outro caso contrário.

```
if (numero % 2 == 0) {
    System.out.println("O número é par.");
} else {
    System.out.println("O número é ímpar.");
}
```

Estrutura de Decisão - if...else if...else

Usado para verificar múltiplas condições.

```
if (nota >= 90) {
    System.out.println("Conceito A");
} else if (nota >= 70) {
    System.out.println("Conceito B");
} else {
    System.out.println("Conceito C");
}
```

Estrutura de Decisão - switch

Mais eficiente que múltiplos if...else quando a verificação é sobre o mesmo valor.

```
switch (dia) {
    case 1: nomeDia = "Domingo"; break;
    case 2: nomeDia = "Segunda"; break;
    case 3: nomeDia = "Terça"; break;
    case 4: nomeDia = "Quarta"; break;
    case 5: nomeDia = "Quinta"; break;
    case 6: nomeDia = "Sexta"; break;
    case 7: nomeDia = "Sábado"; break;
    default: nomeDia = "Dia inválido"; break;
}
```

1. Sem o `break`, os `cases` seguintes iram ser executados, mesmo se a condição for falso.

Estruturas de Repetição - while

Repete o bloco enquanto a condição for verdadeira.

```
while (contador <= 5) {
    System.out.println("Contagem: " + contador);
    contador++;
}
```

Estruturas de Repetição - do...while

Executa o bloco ao menos uma vez e depois repete enquanto a condição for verdadeira.

```
do {
    System.out.println("Número: " + numero);
    numero++;
} while (numero <= 3);
```

continue

Ele interrompe a execução da iteração atual do loop e faz o programa saltar diretamente para a próxima iteração, sem executar o restante do código dentro do loop para aquela iteração.

```
do {
        /* Código */
    continue;
    /* Código */
} while (condição);
```

Estruturas de Repetição - for

Usado quando se sabe o número de repetições.

```java
for (int i = 0; i < 5; i++) {
    System.out.println("Valor de i: " + i);
}
```

Estruturas de Repetição - for

Usado para percorrer elementos de arrays ou coleções.

```
String[] frutas = {"Maçã", "Banana", "Laranja"};

for (String fruta : frutas) {
    System.out.println(fruta);
}
```

Valor `null`

Em Java, o valor `null` representa a ausência de valor para variáveis de tipos de referência (objetos, arrays, Strings etc.).

Uso comum do `null`

1. Representar ausência de dados em objetos.
2. Indicar que algo ainda não foi inicializado.
3. Valor padrão para objetos.

Evite usar `null` como algo diferente de `0`

Métodos

Um método é um bloco de código que executa uma tarefa específica e pode receber parâmetros de entrada e retornar um resultado.

Ele serve para organizar o código em partes reutilizáveis, facilitar a manutenção e melhorar a legibilidade do programa.

```
[modificador] tipoRetorno nomeDoMetodo([parâmetros]) {
    // corpo do método
    // comandos que o método executa
    [return valor];  // opcional, depende do tipoRetorno
}
```

Métodos

Um método é um bloco de código que executa uma tarefa específica e pode receber parâmetros de entrada e retornar um resultado.

Ele serve para organizar o código em partes reutilizáveis, facilitar a manutenção e melhorar a legibilidade do programa.

```java
// método que imprime uma mensagem
public static void mostrarMensagem() {
    System.out.println("Olá, mundo!");
}
```

Métodos

Métodos `void` não retornam valor.

Para métodos que retornam valor, é obrigatório usar o comando `return` com o valor correto.

```java
// método que imprime uma mensagem
public static void mostrarMensagem() {
    System.out.println("Olá, mundo!");
}
```

Métodos

Um método é um bloco de código que executa uma tarefa específica e pode receber parâmetros de entrada e retornar um resultado.

Ele serve para organizar o código em partes reutilizáveis, facilitar a manutenção e melhorar a legibilidade do programa.

```java
// método que recebe um nome e imprime uma saudação
public static void cumprimentar(String nome) {
    System.out.println("Olá, " + nome + "!");
}
```

Métodos

Um método é um bloco de código que executa uma tarefa específica e pode receber parâmetros de entrada e retornar um resultado.

Ele serve para organizar o código em partes reutilizáveis, facilitar a manutenção e melhorar a legibilidade do programa.

```java
// método que recebe um nome e imprime uma saudação
public static void cumprimentar(String nome) {
    System.out.println("Olá, " + nome + "!");
}
```

Métodos

Um método é um bloco de código que executa uma tarefa específica e pode receber parâmetros de entrada e retornar um resultado.

Ele serve para organizar o código em partes reutilizáveis, facilitar a manutenção e melhorar a legibilidade do programa.

```java
// método que soma dois números e retorna o resultado
public static int somar(int a, int b) {
    return a + b;
}
```

Convertendo Valores

Para evitar problemas relacionados a \n sobrando no buffer, a principal estratégia é ler o dado como String e depois converter para algum outro tipo correspondente.

Tipo primitivo	Método para converter de `String`
`int`	`Integer.parseInt(String s)`
`long`	`Long.parseLong(String s)`
`float`	`Float.parseFloat(String s)`
`double`	`Double.parseDouble(String s)`
`short`	`Short.parseShort(String s)`
`boolean`	`Boolean.parseBoolean(String s)`

Convertendo Valores

Para evitar problemas relacionados a \n sobrando no buffer, a principal estratégia é ler o dado como String e depois converter para algum outro tipo correspondente.

Só retorna true se a string for "true" (ignora maiúsculas/minúsculas).

1. "true", "TRUE", "TrUe" → true

Qualquer outro valor (inclusive "yes", "1", "verdadeiro") retorna `false`

Convertendo Valores

```
System.out.println(Boolean.parseBoolean("true"));   // true
System.out.println(Boolean.parseBoolean("TRUE"));   // true
System.out.println(Boolean.parseBoolean("yes"));    // false
System.out.println(Boolean.parseBoolean("false"));  // false
System.out.println(Boolean.parseBoolean("banana")); // false
```

Método Main

O método **main** em Java é o ponto de entrada de qualquer aplicação Java.

Ele é:

- Público;
- Estático;
- void;

```java
public class Main {

    public static void main(String[] args) {
        System.out.println("Hello World");
    }

}
```

Instanciando uma classe

```
Bolo meuBolo = new Bolo();
```

O operador `new` em Java é usado para criar novos objetos a partir de classes. Ele aloca memória para um objeto na heap e retorna uma referência a esse objeto.

Ao contrário de outras linguagens de programação, o Java gerencia automaticamente a memória através do garbage collector, o que significa que o programador não precisa se preocupar em liberar memória manualmente.

O this

O `this` em Java é uma palavra-chave especial que serve para referenciar o objeto atual — ou seja, o objeto cujo método ou construtor está sendo executado.

Ela também serve para diferenciar atributos da classe de parâmetros locais com mesmo nome.

O this

Por exemplo, dentro de um método, se você tem um parâmetro cor e um atributo cor, usar this.cor indica que está acessando o atributo da instância, e não o parâmetro.

```
class Pessoa {

    String nome;

    void atulizarNome(String nome) {
        nome = nome;
    }
```

Construtores

Um construtor é um método especial usado para inicializar objetos e é chamado automaticamente no momento da instanciação.

Tem o mesmo nome da classe e não possui tipo de retorno (nem void).

```
Pessoa p = new Pessoa();
```

```
class Pessoa {

    Pessoa () {
        // Código
    }

}
```

Construtores

Ele é importante para garantir que:

1. Um objeto comece em um estado válido.
2. Reduzir o risco de atributos ficarem sem valor ou inconsistentes.
3. Facilitar a criação de objetos, permitindo definir valores logo na instanciação.
4. Encapsular a lógica de inicialização (ex.: validar parâmetros, gerar IDs, abrir conexões, etc.).

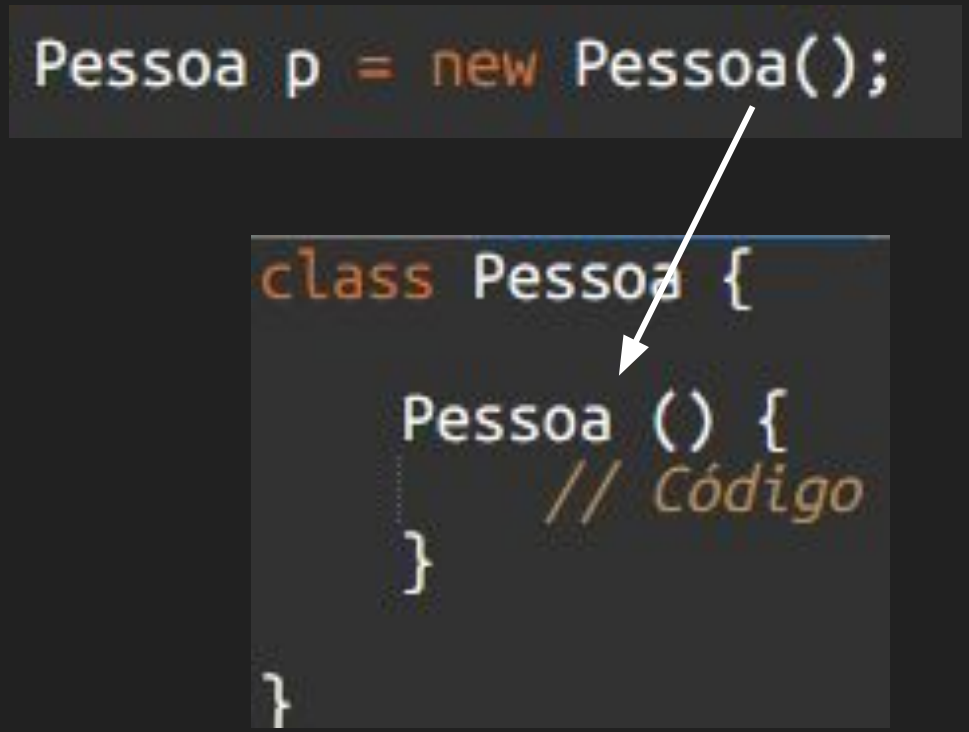
```
Pessoa p = new Pessoa();
```

```
class Pessoa {

    Pessoa () {
        // Código
    }

}
```

Construtores

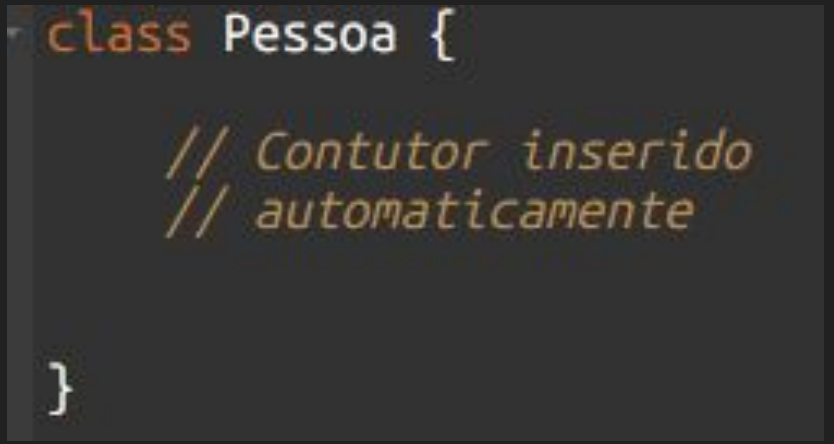
Um construtor é um método especial usado para inicializar objetos e é chamado automaticamente no momento da instanciação.

Tem o mesmo nome da classe e não possui tipo de retorno (nem void).



Construtores

Por padrão, quando nenhum construtor é definido na classe, o Java cria automaticamente o construtor padrão.



Construtores

Quando você adiciona um construtor com parâmetros você pode inicializar os atributos da sua classe

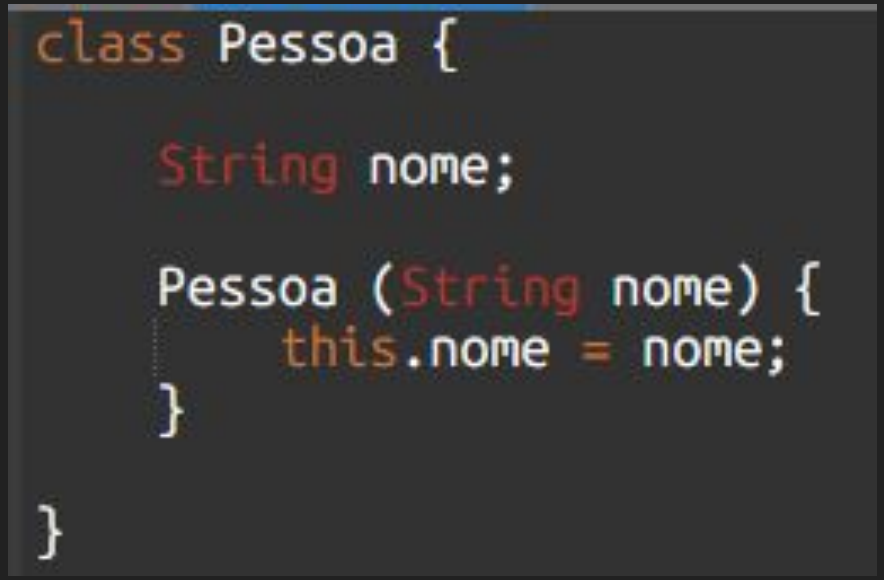
```
Pessoa p = new Pessoa("Jerff");
```

```
class Pessoa {

    String nome;

    Pessoa (String nome) {
        this.nome = nome;
    }

}
```

Construtores

Porém, se um construtor com é definido (Com ou sem Paramêtros), o Java não insere o construtor padrão.

Se o construtor padrão não for definido e uma uma instância de classe que chame o construtor padrão não poderá ser criada.



Construtores

Porém, se um construtor com é definido (Com ou sem Parâmetros), o Java não insere o construtor padrão.

Se o construtor padrão não for definido e uma uma instância de classe que chame o construtor padrão não poderá ser criada.

```
Pessoa p = new Pessoa();
```

```
Main.java:13: error: constructor Pessoa in class Pessoa cannot be applied to given types;
                Pessoa p = new Pessoa();
                           ^
```

Construtores

Uma classe pode ter mais de um construtor, com assinaturas diferentes. Isso permite criar objetos de várias formas.

```
class Pessoa {
    String nome;
    Pessoa() {
    }
    Pessoa (String nome) {
    }
}
```

Modificadores de Acesso

Controlam **quem pode acessar** uma classe, método ou atributo.

Modificador	**Visível para…**
public	Todas as classes em qualquer pacote.
protected	Classes do mesmo pacote ou subclasses, mesmo que estejam em outros pacotes.
default	Apenas classes do mesmo pacote.
private	Somente dentro da própria classe.

Tipos Objeto (Wrapper)

São classes que "encapsulam" os tipos primitivos como objetos.

Tipo Primitivo	Tipo Wrapper (Classe)	Exemplo Primitivo	Exemplo Wrapper
`int`	`Integer`	`int i = 3;`	`Integer i = 3;`
`long`	`Long`	`long l = 4L;`	`Long l = 4L;`
`float`	`Float`	`float f = 5.5f;`	`Float f = 5.5f;`
`double`	`Double`	`double d = 6.6;`	`Double d = 6.6;`
`char`	`Character`	`char c = 'A';`	`Character c = 'A';`
`boolean`	`Boolean`	`boolean b = true;`	`Boolean b = true;`

Tipos Objeto (Wrapper)

Eles são úteis quando você precisa de:

1. Armazenar null (representar ausência de valor)
2. Métodos utilitários (ex: conversão, parsing)
3. Comparações com `.equals()`
4. Trabalhar com coleções (List<Integer> e não List<int>)

Tipos Objeto (Wrapper) - Métodos utilitários (ex: conversão, parsing)

São métodos que ajudam a trabalhar com números, conversões, parsing de texto, comparações, etc.

```
Integer.parseInt("123")      // converte String em int
Integer.toString(42)         // converte int em String
Integer.valueOf("123")       // converte String em Integer
Integer.compare(10, 20)      // compara dois ints: retorna -1, 0 ou 1
Integer.max(5, 9)            // retorna o maior entre dois valores
Integer.min(5, 9)            // retorna o menor
Integer.sum(4, 6)            // retorna a soma
```

Tipos Objeto (Wrapper) - Comparações com .equals()

```java
Integer a = 128;
Integer b = 128;

System.out.println(a == b); // false (objetos diferentes)
System.out.println(a.equals(b)); // true (valores iguais)
```

Arrays e Listas em Java

Listas

Usada para armazenar elementos de forma dinâmica, como um array que pode crescer ou diminuir.

```java
import java.util.ArrayList;

ArrayList<String> nomes = new ArrayList<>();
nomes.add("Ana");
nomes.add("João");
nomes.remove("Ana");
System.out.println(nomes.get(0)); // João
```

Formas de percorrer uma ArrayList em Java

```java
ArrayList<String> lista = new ArrayList<>();
lista.add("Maçã");
lista.add("Banana");
lista.add("Laranja");

for (int i = 0; i < lista.size(); i++) {
    System.out.println(lista.get(i));
}
```

Formas de percorrer uma ArrayList em Java

```java
ArrayList<String> lista = new ArrayList<>();
lista.add("Maçã");
lista.add("Banana");
lista.add("Laranja");

for (String fruta : lista) {
    System.out.println(fruta);
}
```

Formas de percorrer uma ArrayList em Java

```java
ArrayList<String> lista = new ArrayList<>();
lista.add("Maçã");
lista.add("Banana");
lista.add("Laranja");

lista.forEach(fruta -> System.out.println(fruta));
```

Formas de percorrer uma ArrayList em Java

```java
ArrayList<String> lista = new ArrayList<>();
lista.add("Maçã");
lista.add("Banana");
lista.add("Laranja");

lista.forEach(fruta -> {

    // código
    System.out.println(fruta)

});
```

Spring

O Spring Framework é um dos frameworks opensource Java mais importantes e amplamente utilizados para desenvolvimento de aplicações, especialmente aplicações web e coorporativas.
Ele simplifica a criação de sistemas Java ao facilitar o gerenciamento de objetos (beans) e suas dependências, reduzindo a complexidade e tornando o código mais modular, testável e escalável.

Spring

Na década de 2000, o desenvolvimento de aplicações Java EE era complexo e pesado, especialmente com tecnologias como EJB (Enterprise Java Beans).

Em 2002, Rod Johnson publicou um livro criticando essa complexidade e propondo alternativas mais simples para o desenvolvimento Java corporativo.

Em 2003, Rod Johnson, junto com colaboradores, iniciou o desenvolvimento do Spring Framework como projeto open source.

Em 2004, lançou a versão 1.0 do Spring Framework, já focado em solucionar problemas comuns, como excesso de configuração XML e dependências rígidas.

Spring

Configuração pesada e fragmentada

Era comum precisar criar muitos arquivos XML de configuração para diversas partes da aplicação: web.xml, ejb-jar.xml, applicationContext.xml, e arquivos específicos para segurança, mensageria, etc.

Por exemplo, para configurar um simples EJB (Enterprise Java Bean), era necessário definir interfaces remotas, locais, beans, e descrever tudo no XML. Isso tornava o desenvolvimento prolixo e sujeito a erros.

Spring

Aplicações Java EE geralmente rodavam em servidores de aplicação robustos como WebSphere, WebLogic ou JBoss, que exigiam configuração complexa e não favoreciam a agilidade.

Cada servidor tinha suas particularidades, o que gerava baixa portabilidade e dificuldade para migrar aplicações.

Spring

Generative AI

Integrate AI into your Spring applications without reinventing the wheel.

Microservices

Quickly deliver production-grade features with independently evolvable microservices.

Reactive

Spring's asynchronous, nonblocking architecture means you can get more from your computing resources.

Cloud

Your code, any cloud—we've got you covered. Connect and scale your services, whatever your platform.

Web apps

Frameworks for fast, secure, and responsive web applications connected to any data store.

Serverless

The ultimate flexibility. Scale up on demand and scale to zero when there's no demand.

Event Driven

Integrate with your enterprise. React to business events. Act on your streaming data in realtime.

Batch

Automated tasks. Offline processing of data at a time to suit you.

Spring

Os módulos principais do Spring Framework são a base que o tornam um ecossistema flexível e poderoso para o desenvolvimento Java. Eles podem ser usados isoladamente ou em combinação conforme as necessidades da aplicação.

Spring

Core Container: É o núcleo do Spring e engloba funcionalidades essenciais como:

Core: Define o container de Inversão de Controle (IoC) que gerencia o ciclo de vida e dependências dos objetos (beans).

Beans: Contém as definições e o gerenciamento dos beans no container.

Spring

Context: Implementa o ApplicationContext que oferece recursos avançados como internacionalização, eventos e suporte para integração com Java EE.

Spring Expression Language (SpEL): Linguagem para configurar valores e comportamentos dos beans.

Spring

Data Access/Integration: Voltado para integração e acesso a dados:

Facilita o acesso a bancos através de JDBC, ORM (Hibernate, JPA), e integrações com mensageria (JMS).

Suporta mapeamento objeto-relacional e facilita transações e persistência.

Spring

Web: Suporta desenvolvimento de aplicações web e APIs REST.

Inclui o Spring MVC para implementar o padrão Model-View-Controller.

Facilita a criação de controladores, tratamento de requisições HTTP, manipulação de formulários, sessões, e integração com tecnologias web.

Spring

AOP (Programação Orientada a Aspectos) e Instrumentation

Permite separar preocupações transversais, como logging, segurança e transações, do código principal.

Modula funcionalidades transversais para facilitar manutenção e reutilização.

Suporte à instrumentação de classes em tempo real para monitoramento.

Spring

Módulos adicionais e ecossistema Spring mais amplo:

Spring Boot: Facilita a configuração e inicialização rápida de projetos Spring com configuração automática.

Spring Data: Simplifica a integração e manipulação de dados em múltiplos bancos relacionais e NoSQL.

Spring Security: Gerência autenticação e autorização.

Spring Cloud: Ferramentas para desenvolvimento de sistemas distribuídos e microserviços.

Spring Batch: Para processamento em lote.

Spring Web Services: Facilita a criação de serviços SOAP.

Spring - Criando o projeto

O Spring Boot CLI (Command Line Interface) é uma forma de rodar aplicações Spring direto do terminal, sem precisar configurar um projeto completo no início. Ele é útil para prototipagem rápida, scripts ou até para ensinar conceitos de Spring sem ficar preso à estrutura pesada de projeto.

https://docs.spring.io/spring-boot/installing.html#getting-started.installing.cli

Spring - Criando o projeto

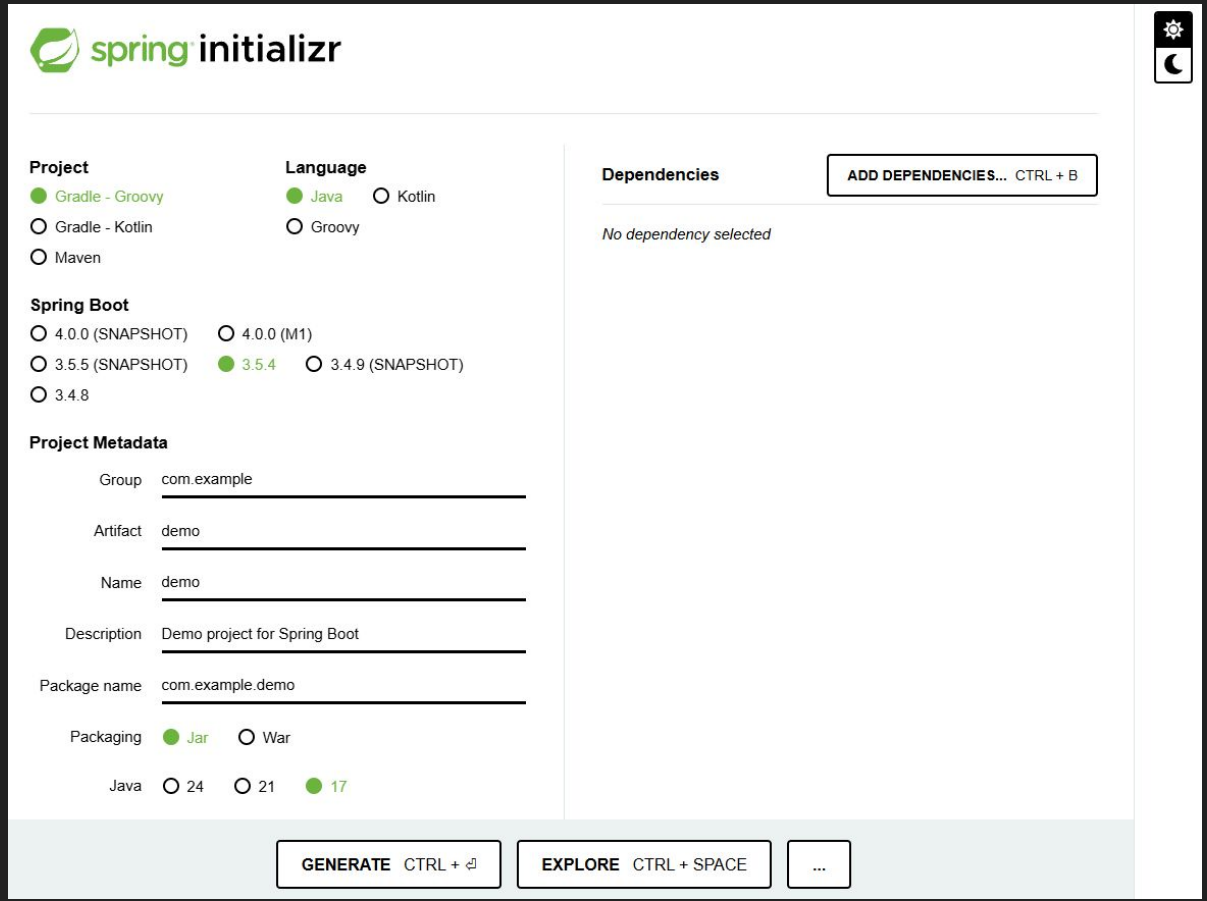
Spring Initializr

É uma ferramenta oficial do Spring (https://start.spring.io) que ajuda a gerar a estrutura inicial de um projeto Spring Boot.

Funciona como um “template generator”, onde você escolhe algumas configurações e ele entrega um projeto pronto para compilar e rodar.

Evita que você precise configurar Maven/Gradle e dependências do zero.

Spring - Criando o projeto



Spring - Criando o projeto

No site do Spring Initializr, você seleciona:

1. Project: Maven ou Gradle.
2. Language: Java, Kotlin ou Groovy.
3. Spring Boot Version: versão que deseja usar (ex: 3.5.x).
4. Project Metadata:
    - Group: como se fosse o “pacote base” da sua aplicação (com.seuprojeto).
    - Artifact: É o identificador técnico do projeto. Vai ser usado como: Nome do arquivo gerado (artifactId-version.jar).
    - Name: nome amigável da aplicação.
    - Description e Package name opcionais.
5. Packaging: jar (mais comum) ou war.
6. Java Version: versão do Java instalada (ex: 17, 21).
7. Dependencies: as dependências que a aplicação precisa. Exemplo:
    - spring-boot-starter-web → para aplicações web REST.
    - spring-boot-starter-data-jpa → para persistência com bancos relacionais.
    - spring-boot-starter-security → para autenticação/autorização.

Depois de escolher, você clica em Generate → baixa um .zip com o projeto já configurado.

Spring - Criando o projeto

Há padrões de boas práticas para definir o nome do pacote base (o campo Group e o Package name no Spring Initializr).

Isso é importante porque o Spring Boot usa o pacote base para o component scan, ou seja, para encontrar automaticamente classes com @Component, @Service, @Repository, @Controller etc.

Spring - Criando o projeto

1. Começar com o domínio invertido da organização:
    a. Exemplo: se sua empresa é empresa.com.br, o início do pacote deve ser: br.com.empresa
2. Depois vem o nome do projeto/aplicação
    a. Se o projeto chama sistemadepagamento: br.com.empresa.sistemadepagamento
3. Estrutura interna por camada/módulo
    a. Dentro do pacote base, normalmente organiza-se em camadas: