



Ciência da **Computação**

Programação Orientada a Objetos Avançado
Prof. Luciano Rodrigo Ferretto

Packages – Pacotes em Java

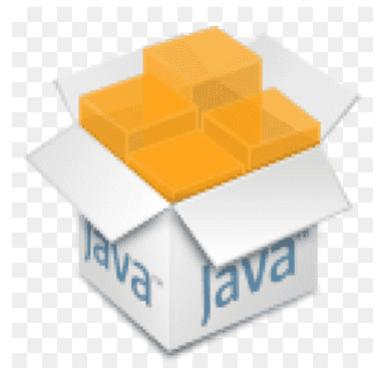
Organização é a palavra certa



Pacotes em Java - Package

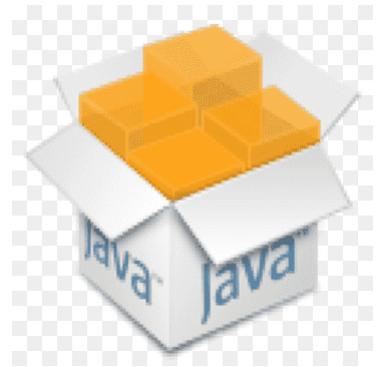
- Em Java, um pacote (package) é um mecanismo de organização de classes e interfaces em grupos lógicos.
- **Benefícios dos Pacotes:**
- **Organização:** Mantenha seu código organizado e estruturado, evitando um emaranhado de classes dispersas.
- **Evita Conflitos de Nomes:** Classes com o mesmo nome podem coexistir em pacotes diferentes, sem gerar conflitos.
- **Reutilização de Código:** Facilite a reutilização de classes em diferentes projetos, promovendo modularidade e eficiência.
- **Importação Simplificada:** Importe classes de pacotes específicos com facilidade, utilizando a palavra-chave “**import**”
- **Visibilidade Controlada:** Defina o nível de acesso das classes dentro do pacote, protegendo-as de acessos indesejados.

Pacotes em Java - Package



- **Hierarquia de Pacotes:** Os pacotes são organizados em uma hierarquia, semelhante a pastas em um sistema de arquivos.
- Por exemplo, você pode ter um pacote chamado:
 - **br.edu.atitus.meuprojeto** que contém subpacotes como **br.edu.atitus.meuprojeto.modelo** e **br.edu.atitus.meuprojeto.util**
 - ✓ **br.edu.atitus.meuprojeto**
 - > **HelloWorld.java**
 - br.edu.atitus.meuprojeto.modelo**
 - ✓ **br.edu.atitus.meuprojeto.util**
 - > **ValidadorCPF.java**

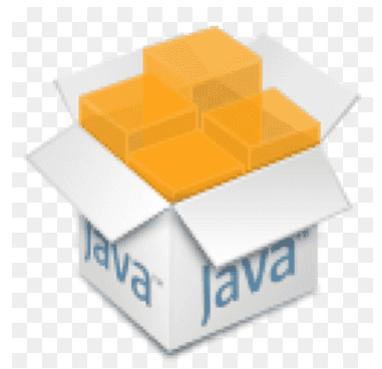
Pacotes em Java - Package



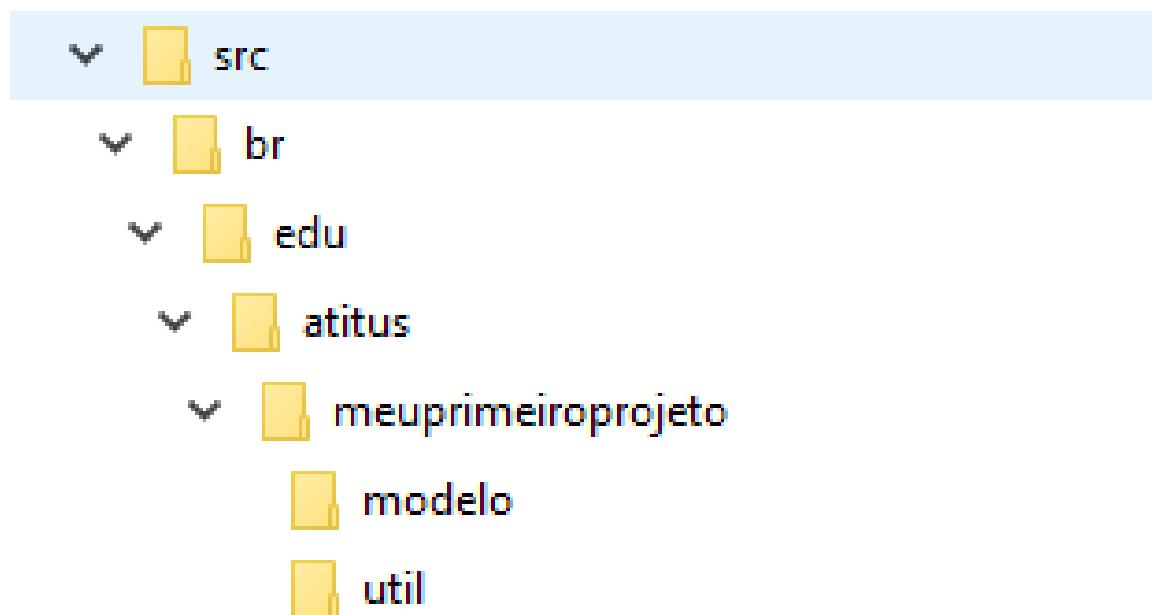
- **Declaração de Pacote:** No início de um arquivo de classe ou interface Java, você deve declarar o pacote ao qual ele pertence usando a palavra-chave ***package***.
 - No exemplo abaixo temos a classe ***HelloWorld*** que pertence (está) no pacote ***br.edu.atitus.meuprimeiroprojeto***

```
1 package br.edu.atitus.meuprimeiroprojeto;  
2  
3 public class HelloWorld {  
4     public static void main(String[] args) {  
5         System.out.println("Hello World!!");  
6     }  
7 }
```

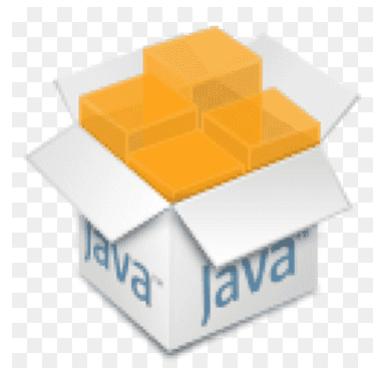
Pacotes em Java - Package



- Ao criarmos um pacote, a nossa IDE cria uma estrutura de pastas em nosso sistema de arquivos.



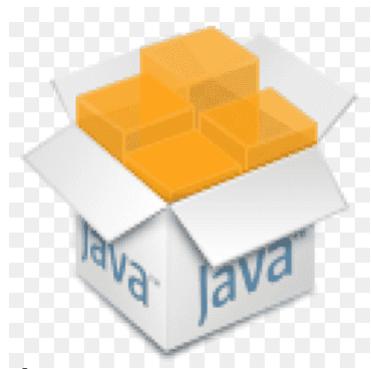
Pacotes em Java - Package



- **Importação de Pacote:** Para usar classes de outros pacotes em seu código, você pode importá-las usando a palavra-chave ***import***.
 - No exemplo abaixo a linha com o ***import*** permite que você use a classe **ValidadorCPF** sem precisar digitar o nome completo com o pacote toda vez que a referenciar.

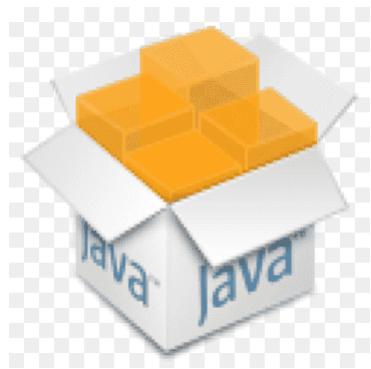
```
1 package br.edu.atitus.meuprimeiroprojeto;
2
3 import br.edu.atitus.meuprimeiroprojeto.util.ValidadorCPF;
4
5 public class HelloWorld {
6     public static void main(String[] args) {
7         System.out.println("Hello World!!!");
8         String cpf = "999.999.999-99";
9         boolean cpfValido = ValidadorCPF.validarCPF(cpf);
10        System.out.println(cpfValido);
11    }
12 }
```

Pacotes em Java - Package



- **Java Standard Library:** A plataforma Java possui uma série de pacotes padrão, como **java.util**, **java.io**, **java.lang**, entre outros.
- Você não precisa importar explicitamente o pacote **java.lang**, pois ele é automaticamente disponibilizado para você em seus programas Java.
- No entanto, pacotes como **java.util** e **java.io** precisam ser importados explicitamente.
- **Convenção de Nomenclatura:** Os nomes de pacotes geralmente seguem a convenção de nomenclatura reversa do domínio da empresa para evitar conflitos de nome.
 - Por exemplo, se o site da sua empresa é *minhaempresa.com*, você pode usar **com.minhaempresa** como o início do nome do pacote.

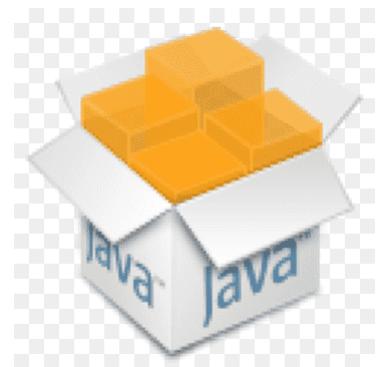
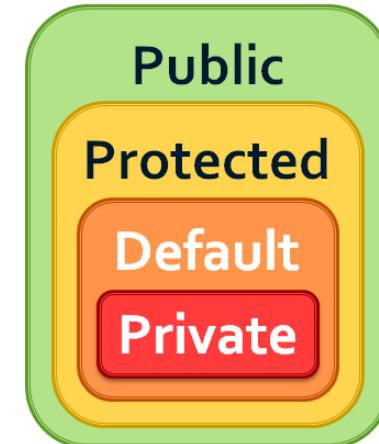
Pacotes em Java - Package



- **Acesso Protegido:** Classes com o modificador de acesso "padrão" (sem public, private ou protected) são acessíveis apenas dentro do mesmo pacote. Isso ajuda a controlar o acesso às classes e manter a encapsulação.

Pacotes em Java - Package

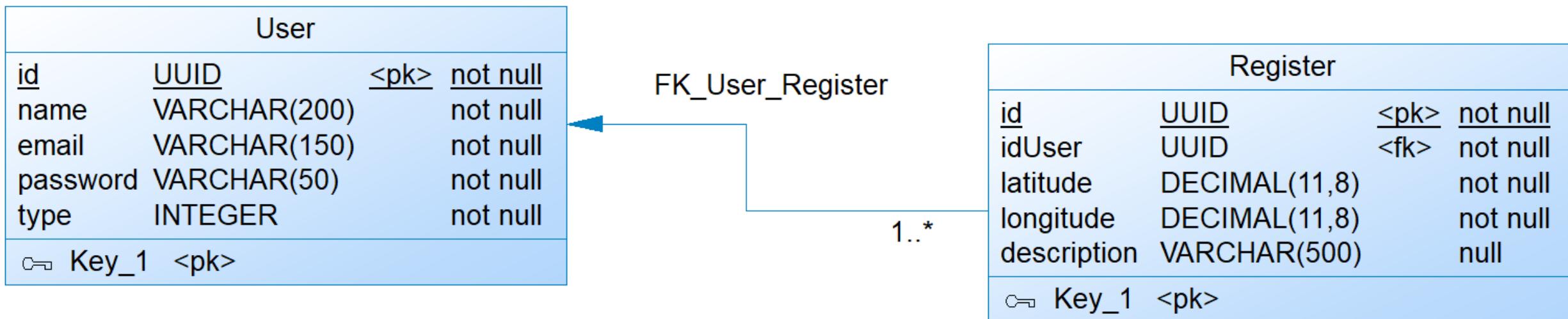
- Modificadores de Acesso



Visibilidade	public	protected	default	private
A partir da mesma classe	✓	✓	✓	✓
Qualquer classe no mesmo pacote	✓	✓	✓	✗
Qualquer classe filha no mesmo pacote	✓	✓	✓	✗
Qualquer classe filha em pacote diferente	✓	✓	✗	✗
Qualquer classe em pacote diferente	✓	✗	✗	✗

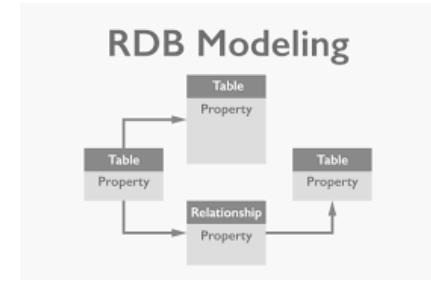
Desenvolvimento de soluções (Full Stack)

Objetivo do Semestre → Projeto APISample



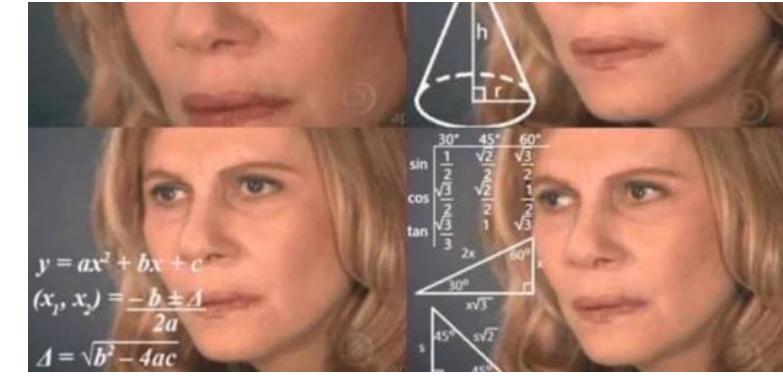
Mas qual o objetivo específico da disciplina Organização e Abstração na Programação

- Modelar e estruturar o banco de dados.
- Desenvolver o Backend:
 - Orientação a Objetos com Java
 - No estilo arquitetural API Rest
 - Utilizando o ecossistema Spring Framework
- Hospedar na Nuvem.



Mas para isso precisamos entender algumas coisas...

- ✓ • Linguagem Java (sintaxe e funcionamento)
- ✓ • Paradigma Orientado a Objetos (conceitos desenvolvidos em Java)
- O que é um Backend? (Arquitetura de Sistemas)
- O que é uma API?
- O que é API Rest? (Padrão arquitetural RestFull)
 - O que é e como funciona o HTTP?
- O que é Spring Framework?
- O que é UUID?
- O que é JSON?



O que é um backend???

Para entender isso precisamos estudar um pouco de
Arquitetura de Sistemas.

Arquitetura de Sistemas

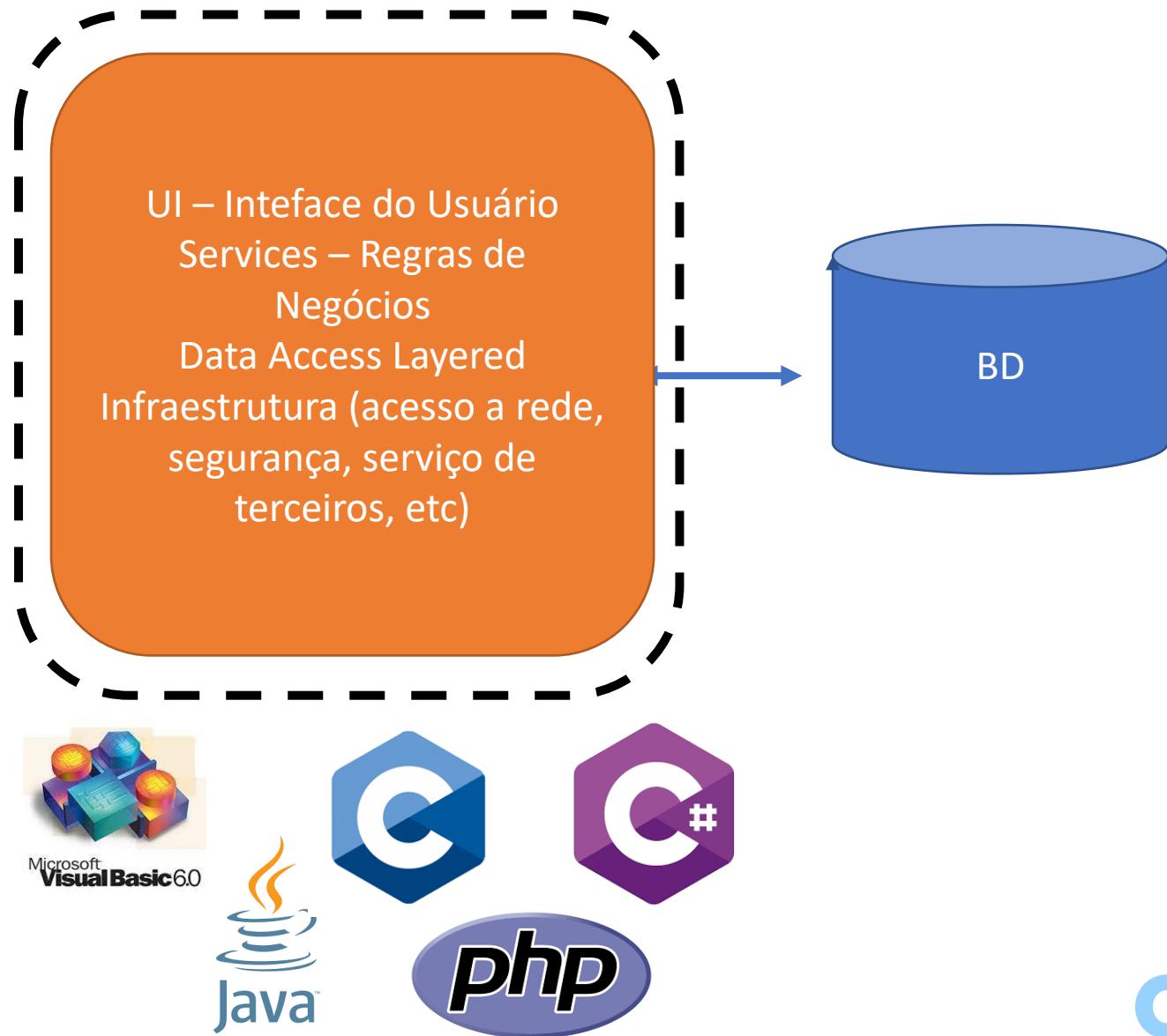
- A Arquitetura de Sistemas é o projeto estrutural de sistemas complexos, que envolvem componentes de hardware, software, redes e outros elementos.
 - Visa criar uma estrutura organizada que facilite a compreensão, o desenvolvimento, a manutenção e a evolução do sistema ao longo do tempo.
 - A escolha da arquitetura adequada pode influenciar diretamente na qualidade, desempenho, segurança e escalabilidade de um sistema.
-
- Arquitetura Monolítica
 - Arquitetura em Camadas (Layered)
 - Arquitetura Cliente-Servidor
 - Microservices (Microserviços)

Arquitetura Monolítica

Arquitetura Monolítica

- Nesse tipo de arquitetura, todo o sistema é desenvolvido como um único componente ou aplicação.
- Ele geralmente é mais simples de implementar, mas pode se tornar difícil de escalar e manter à medida que o sistema cresce.
- Exemplos incluem aplicativos de desktop tradicionais.

Arquitetura Monolítica



Arquitetura Monolítica - Vantagens

- **Simplicidade:** É relativamente simples de desenvolver e implantar, já que todo o sistema está contido em um único pacote. Isso pode ser vantajoso para projetos menores e equipes com recursos limitados.
- **Comunicação Interna Eficiente:** Como todas as partes do sistema estão dentro do mesmo processo, a comunicação interna entre os componentes é eficiente e de baixa latência.
- **Facilidade de Depuração e Testes:** A depuração e os testes podem ser mais fáceis em uma arquitetura monolítica, pois não há complexidade de comunicação entre diferentes serviços.
- **Manutenção INICIAL Simples:** Em estágios iniciais de desenvolvimento, a arquitetura monolítica pode ser fácil de manter. Mudanças em uma parte do sistema podem ser feitas diretamente no código-base.

Arquitetura Monolítica - Desvantagens

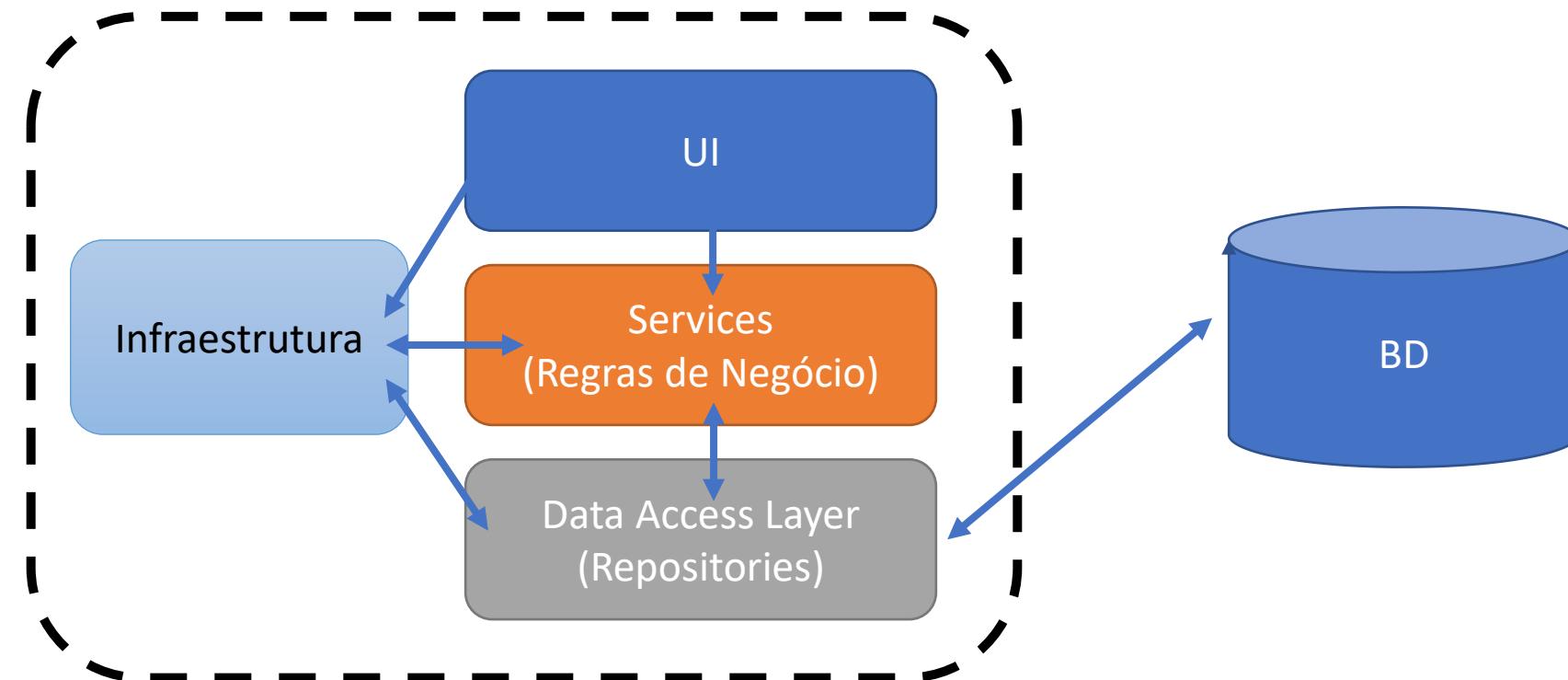
- **Escalabilidade Vertical:** A escalabilidade em uma arquitetura monolítica geralmente é feita aumentando os recursos (como CPU e RAM) da máquina que hospeda o sistema, o que é conhecido como escalabilidade vertical.
- **Limitações de Escalabilidade e Complexidade:** Conforme o sistema cresce e se torna mais complexo, pode se tornar difícil de escalar e manter. A adição de novas funcionalidades ou a realização de mudanças em uma parte do sistema pode afetar outras partes, tornando o desenvolvimento e a evolução mais complicados.
- **Dependências e Acoplamento:** A arquitetura monolítica pode levar ao acoplamento entre diferentes partes do sistema, o que pode dificultar a modificação ou substituição de componentes individuais sem afetar todo o sistema.
- **Risco de Falhas Globais:** Um erro em uma parte do sistema pode impactar todo o monólito, levando a possíveis falhas globais.

Arquitetura de Camadas - Layered

Arquitetura de Camadas - Layered

- Arquitetura em camadas, também conhecida como arquitetura em níveis, é um modelo de design de software que organiza um sistema em camadas distintas, cada uma com responsabilidades bem definidas.
- Cada camada é responsável por um conjunto específico de funcionalidades e interage com camadas adjacentes através de interfaces bem definidas.
- Isso promove a modularidade, a reutilização de código e a manutenção facilitada.

Arquitetura de Camadas - Layered



Arquitetura de Camadas - Layered

- **Camada de Apresentação (Interface do Usuário)**: Responsável pela interação com o usuário.
 - Pode incluir interfaces gráficas, páginas da web ou outros mecanismos de interação.
- **Camada de Lógica de Aplicação (Negócio)**: Contém a lógica de negócios do sistema. Ela processa as regras de negócios, toma decisões e coordena as operações entre as diferentes partes do sistema.
 - A camada de lógica de aplicação muitas vezes não sabe nada sobre a apresentação ou os detalhes de armazenamento.
- **Camada de Acesso a Dados**: Também conhecida como camada de persistência, esta camada lida com o acesso e a manipulação dos dados. Ela interage com bancos de dados ou outros sistemas de armazenamento para recuperar, inserir ou modificar informações.
 - A camada de acesso a dados abstrai os detalhes específicos do armazenamento de dados da camada de negócios.
- **Camada de Infraestrutura**: Lida com detalhes técnicos, como comunicação em rede, segurança, serviços de terceiros e outros aspectos de infraestrutura.
 - Ela oferece suporte às camadas superiores, fornecendo serviços que são necessários para o funcionamento do sistema, mas não estão diretamente relacionados à lógica de negócios.

Arquitetura de Camadas - Vantagens

- **Modularidade:** A divisão clara em camadas facilita a compreensão e a organização do sistema. Cada camada pode ser desenvolvida e mantida separadamente, promovendo a reutilização de código.
- **Manutenção Facilitada:** As mudanças em uma camada podem ser feitas sem afetar outras partes do sistema, desde que as interfaces sejam mantidas. Isso reduz o risco de introduzir erros em outras partes do sistema durante a manutenção.
- **Escalabilidade:** Cada camada pode ser escalada independentemente, conforme necessário. Por exemplo, a camada de acesso a dados pode ser otimizada para lidar com um grande volume de solicitações de leitura/gravação.
- **Padrões de Projeto:** A arquitetura em camadas se alinha bem com muitos padrões de projeto comuns, como MVC (Model-View-Controller), e também atendendo às boas práticas e princípios de desenvolvimento, como o SOLID.

Arquitetura de Camadas - Desvantagens

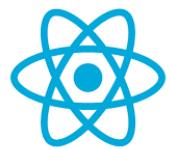
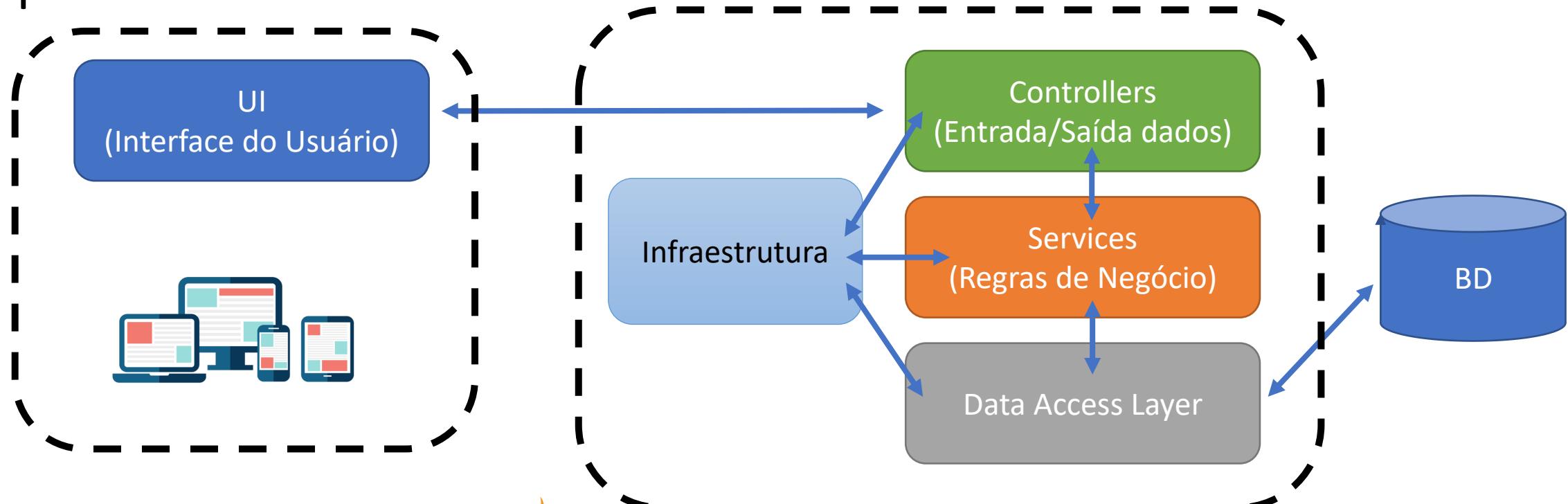
- **Overhead de Comunicação:** A comunicação entre camadas pode adicionar overhead, especialmente em sistemas distribuídos.
- **Complexidade de Design:** O design de sistemas em camadas pode se tornar complexo à medida que o sistema cresce, exigindo uma atenção cuidadosa à definição de interfaces e à coordenação entre as camadas.
- **Latência:** A comunicação entre camadas pode introduzir latência no sistema, especialmente quando os dados precisam passar por várias camadas.

Arquitetura Cliente Servidor Backend-Frontend

Arquitetura Cliente Servidor

- A arquitetura cliente-servidor é um modelo de design de software onde as funcionalidades do sistema são divididas em duas partes principais
- Essas partes se comunicam entre si por meio de solicitações e respostas, formando um sistema distribuído.
- É, atualmente, a arquitetura mais utilizadas em Aplicações Web e Mobile.
- Suas parte são comumente chamadas de **backend** e **frontend**

Arquitetura Cliente-Servidor



React Native



Flutter



Swift



ncia da
Computação

Arquitetura Cliente Servidor

- **Cliente - Frontend:** Parte da aplicação que interage diretamente com o usuário.
 - Solicita serviços ou recursos ao servidor e exibe as informações processadas ao usuário.
 - Os clientes podem variar em complexidade, desde aplicativos de desktop até aplicativos web e móveis e até mesmo embarcados em dispositivos IoT.
- **Servidor - Backend:** Responsável por atender às solicitações dos clientes, processar as operações necessárias e fornecer as respostas apropriadas.
 - Ele pode hospedar dados, lógica de negócios e outros recursos importantes do sistema.
- **Comunicação:** A comunicação entre o cliente e o servidor ocorre por meio de protocolos de comunicação, como o protocolo HTTP e outros.
 - O cliente envia solicitações ao servidor, que processa as solicitações e retorna as respostas.

Arquitetura Cliente Servidor - Vantagens

- **Separação de Responsabilidades:** Permite uma clara separação de responsabilidades.
 - A lógica de apresentação e interação com o usuário fica a cargo do cliente.
 - Enquanto a lógica de processamento e manipulação de dados fica no servidor.
- **Escalabilidade:** AFacilita a escalabilidade, pois é possível adicionar mais servidores para atender a um número crescente de clientes.
- **Segurança:** Como a lógica de negócios e os dados críticos estão centralizados no servidor, é possível aplicar medidas de segurança mais rigorosas para proteger esses recursos.
- **Atualizações e Manutenção:** A separação entre cliente e servidor facilita a atualização e manutenção do sistema. Mudanças na lógica de negócios podem ser feitas no servidor sem afetar diretamente os clientes.

Arquitetura Cliente Servidor - Desvantagens

- **Desvantagens de Desempenho:** Dependendo da arquitetura, pode haver um aumento na latência devido à comunicação entre o cliente e o servidor, especialmente em sistemas distribuídos geograficamente.
- **Dependência de Rede:** Como a comunicação entre cliente e servidor ocorre através da rede, a disponibilidade e a qualidade da rede podem afetar o desempenho do sistema.

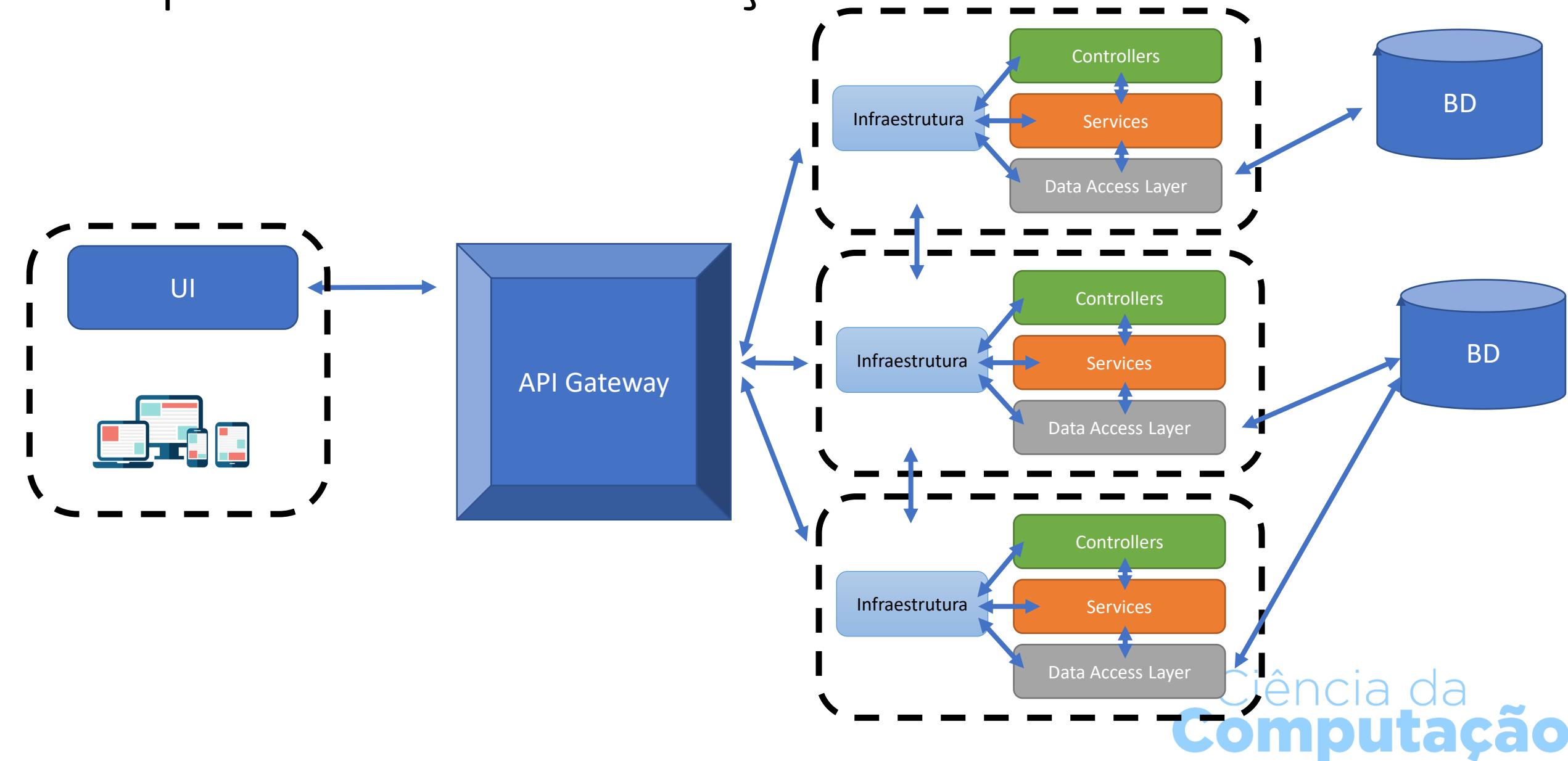
Arquitetura de MicroServiços

MicroServices

Arquitetura de MicroServiços

- A arquitetura de microserviços é um modelo de design de software que envolve a criação de um sistema como um conjunto de serviços independentes, autônomos e altamente especializados.
- Cada serviço, chamado de microserviço, é responsável por uma única funcionalidade de negócios e se comunica com outros microserviços através de APIs.

Arquitetura MicroServiços



Arquitetura de MicroServiços

- A arquitetura de microserviços é um modelo de design de software que envolve a criação de um sistema como um conjunto de serviços independentes, autônomos e altamente especializados.
- Cada serviço, chamado de microserviço, é responsável por uma única funcionalidade de negócios e se comunica com outros microserviços através de APIs.
- **Comunicação via API:** Os microserviços se comunicam entre si através de APIs (Interfaces de Programação de Aplicativos). Essas APIs definem como os microserviços interagem e trocam informações, permitindo uma separação clara de responsabilidades.

Arquitetura de MicroServiços - Vantagens

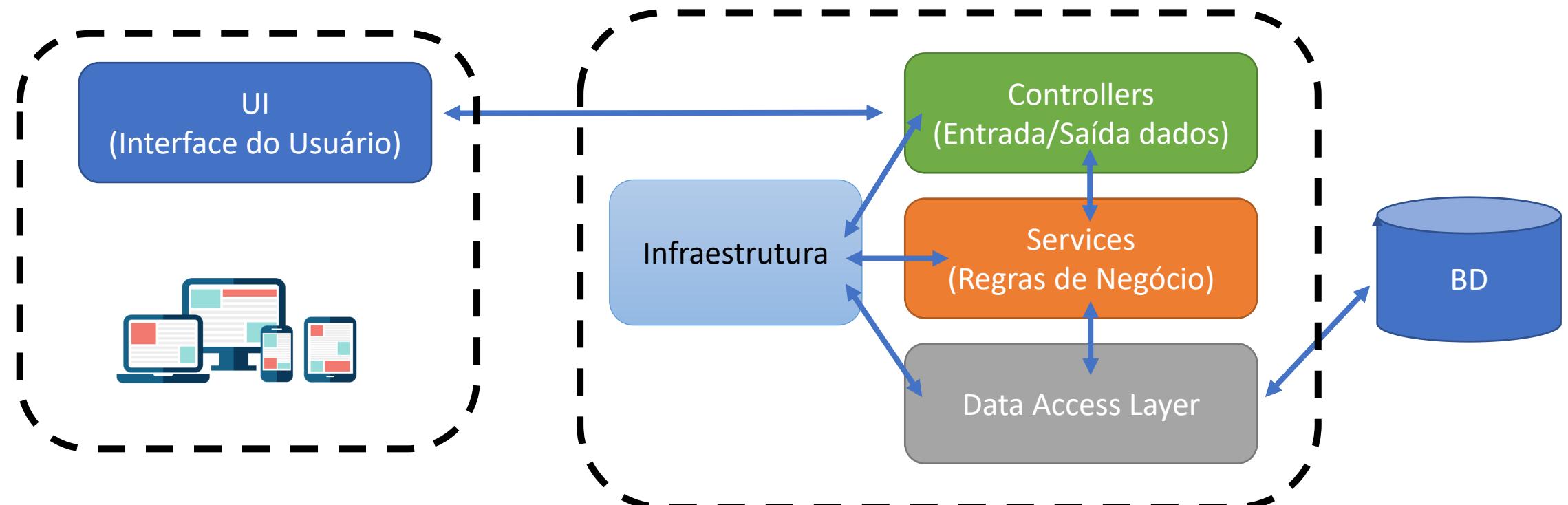
- **Escalabilidade Individual:** Cada microserviço pode ser escalado individualmente, o que significa que recursos adicionais podem ser alocados apenas para os serviços que precisam lidar com mais demanda.
- **Desenvolvimento Independente:** Equipes de desenvolvimento podem trabalhar em diferentes microserviços ao mesmo tempo, utilizando tecnologias e linguagens de programação mais adequadas para cada serviço.
- **Manutenção Facilitada:** Como os microserviços são independentes, as atualizações e correções podem ser feitas em serviços específicos sem afetar o sistema como um todo.
- **Desacoplamento:** A arquitetura de microserviços promove um alto grau de desacoplamento entre os componentes, permitindo que mudanças em um serviço não afetem outros serviços.
- **Resiliência:** Se um microserviço falhar, isso não afeta necessariamente o sistema inteiro. A resiliência pode ser obtida por meio de estratégias como circuit breakers e tratamento de falhas.

Arquitetura de MicroServiços - Desvantagens

- **Complexidade de Gerenciamento:** A arquitetura de microserviços introduz uma complexidade adicional em termos de gerenciamento e coordenação de diferentes serviços.
- **Monitoramento e Observabilidade:** Devido à natureza distribuída dos microserviços, é importante implementar ferramentas de monitoramento para rastrear o desempenho e o estado de cada serviço.

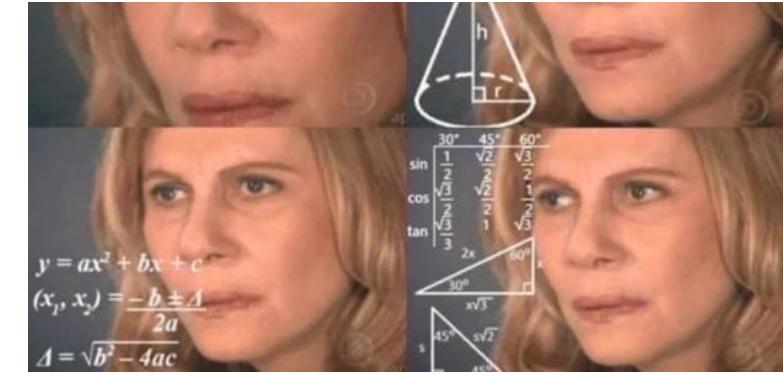
E o nosso projeto??? Como vai ser???

- Arquitetura Cliente-Servidor em conjunto com Arquitetura de Camadas
 - Somente o lado do Servidor (Backend)
 - Resumindo será um API

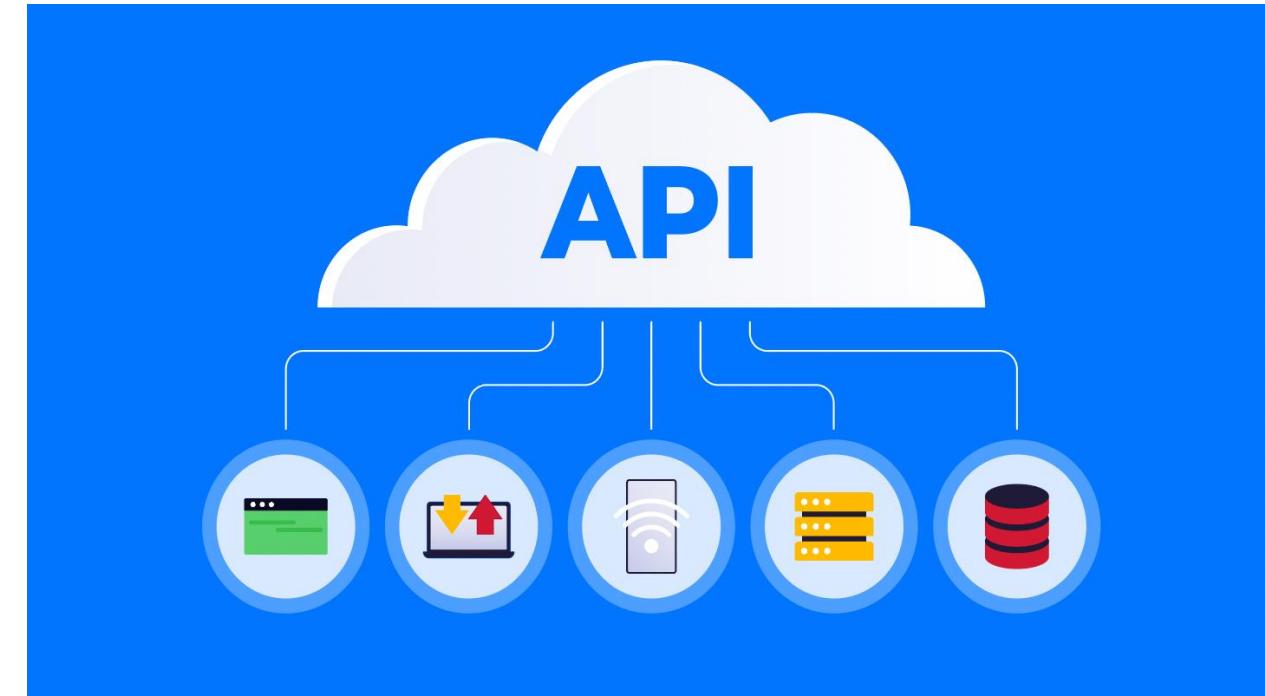


Mas para isso precisamos entender algumas coisas...

- ✓ • Linguagem Java (sintaxe e funcionamento)
- ✓ • Paradigma Orientado a Objetos (conceitos desenvolvidos em Java)
- ✓ • O que é um Backend? (Arquitetura de Sistemas)
- O que é uma API?
- O que é API Rest? (Padrão arquitetural RestFull)
 - O que é e como funciona o HTTP?
- O que é Spring Framework?
- O que é UUID?
- O que é JSON?



O que é uma API???



- API – Application Programming Interface
 - Interface de Programação de Aplicativos
- Permite que dois softwares (de qualquer nível) se comuniquem entre si através de um conjunto de regras e protocolos.
 - Podem ser Sistemas operacionais, softwares de streaming, ERPs, etc.

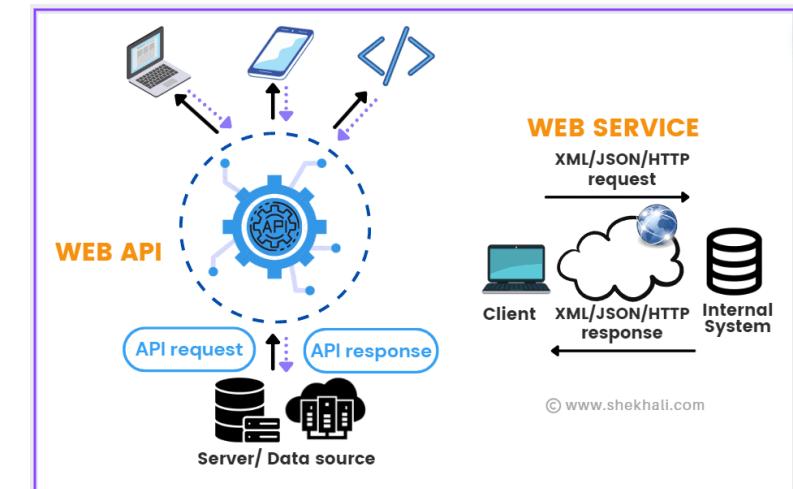
API

- Em outras palavras, é uma interface que define como diferentes componentes de software devem se comunicar e como os desenvolvedores podem usar os recursos e funcionalidades disponibilizados por um sistema.
- As APIs podem ser usadas em uma ampla variedade de cenários, desde integração de serviços e sistemas até desenvolvimento de aplicativos e acesso a dados. Elas são amplamente utilizadas na construção de aplicativos web, aplicativos móveis, integração de sistemas e na criação de mashups.

APIs podem ser classificadas em diferentes categorias

APIs de serviços da web: São APIs disponibilizadas por serviços online, permitindo que os desenvolvedores acessem e utilizem suas funcionalidades e dados. Essas APIs são acessadas geralmente por meio de solicitações HTTP e podem retornar os dados em diferentes formatos, como JSON ou XML.

- **Google Maps API:** Permite que os desenvolvedores acessem e incorporem mapas, informações de localização e serviços relacionados em seus aplicativos ou sites.
- **Twitter API:** Fornece acesso a dados e funcionalidades do Twitter, permitindo que os desenvolvedores criem aplicativos que interajam com a plataforma, como ler e postar tweets.



APIs podem ser classificadas em diferentes categorias

APIs de bibliotecas: São APIs que fornecem um conjunto de funções e métodos para que os desenvolvedores possam criar aplicativos mais facilmente. Essas APIs são geralmente incorporadas a um software e fornecem funcionalidades específicas que podem ser reutilizadas em diferentes contextos.

- **jQuery:** É uma biblioteca JavaScript popular que simplifica a interação com elementos HTML, manipulação do DOM, animações e outras tarefas comuns do desenvolvimento web.
- **NumPy:** É uma biblioteca para Python que fornece suporte a matrizes multidimensionais e funções matemáticas de alto desempenho, sendo amplamente utilizada em computação científica e análise de dados.
- **JDBC (Java Database Connectivity):** É uma API padrão do Java para acessar bancos de dados relacionais. Ela fornece métodos e classes para conectar-se a um banco de dados, executar consultas SQL e gerenciar transações.
- **Spring Framework:** É um framework de desenvolvimento de aplicações Java que fornece uma ampla gama de funcionalidades e bibliotecas para criar aplicativos empresariais. O Spring oferece várias APIs, incluindo o Spring Core, Spring MVC, Spring Data, Spring Security e muitos outros, que abrangem desde injeção de dependência até desenvolvimento web e acesso a bancos de dados.

APIs podem ser classificadas em diferentes categorias

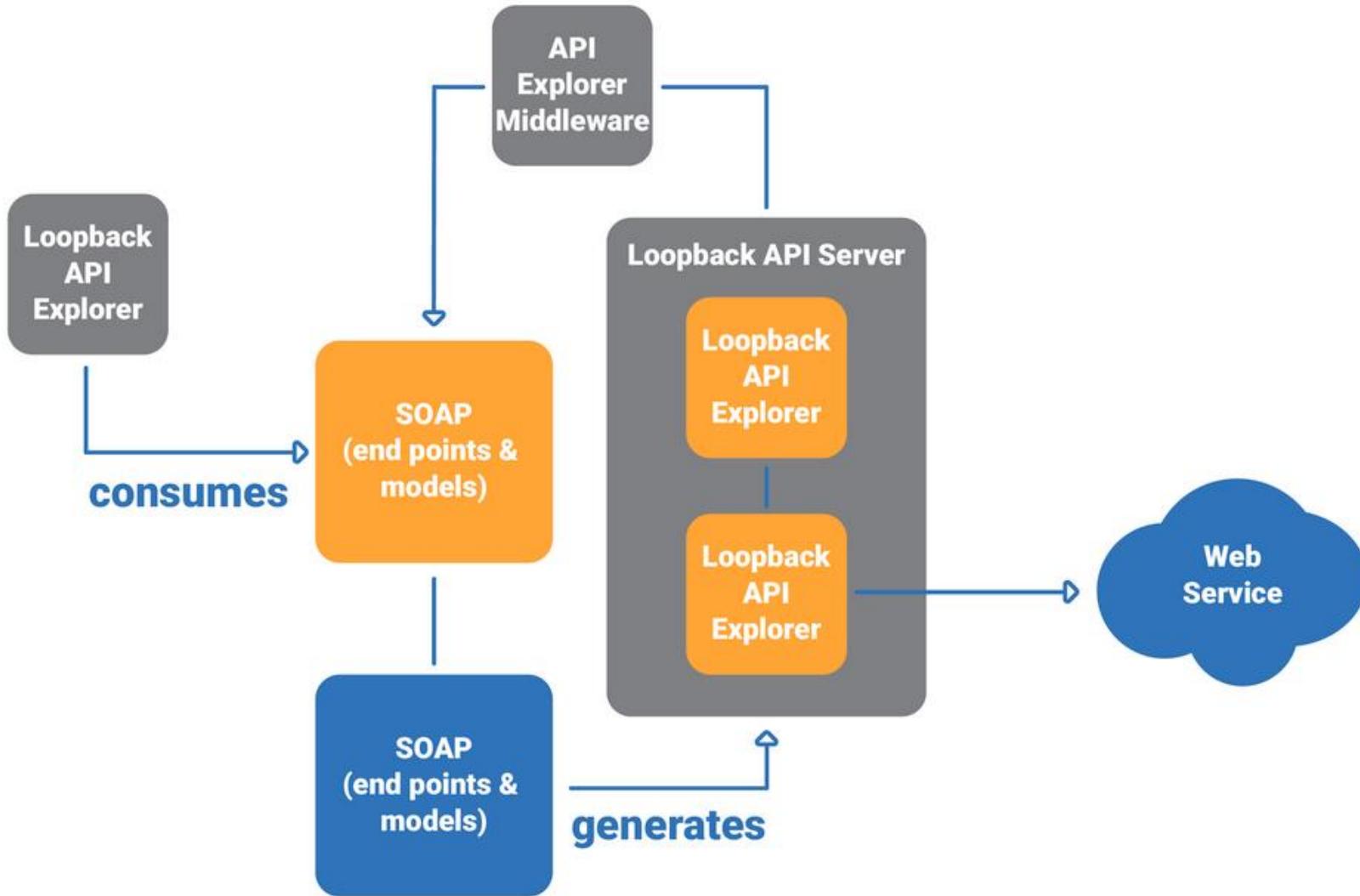
APIs de sistema operacional: São APIs fornecidas pelos sistemas operacionais, permitindo que os desenvolvedores accessem recursos do sistema, como acesso à câmera, geolocalização, armazenamento, entre outros. Essas APIs permitem que os aplicativos interajam com o sistema operacional e tirem proveito de suas funcionalidades.

- **Android API:** É a API fornecida pelo sistema operacional Android, permitindo que os desenvolvedores accessem recursos do dispositivo, como câmera, sensores, contatos, entre outros, para criar aplicativos para dispositivos Android.
- **Windows API:** É a API fornecida pelo sistema operacional Windows, permitindo que os desenvolvedores accessem recursos do sistema, como gerenciamento de arquivos, interface de usuário, serviços de rede, entre outros, para criar aplicativos para o sistema Windows.

Conclusão

- As APIs são essenciais para a construção de sistemas complexos, pois permitem a interconexão entre diferentes componentes de software. Elas facilitam a integração de sistemas, a reutilização de código, o compartilhamento de dados e a criação de novas aplicações combinando recursos de várias fontes.
- Em resumo, as APIs são interfaces que permitem a comunicação entre diferentes softwares e sistemas, permitindo que os desenvolvedores accessem e utilizem recursos e funcionalidades específicas. Elas desempenham um papel fundamental na construção de aplicativos, integração de sistemas e criação de mashups, proporcionando uma maneira padronizada e eficiente de interconectar componentes de software.

SOAP API



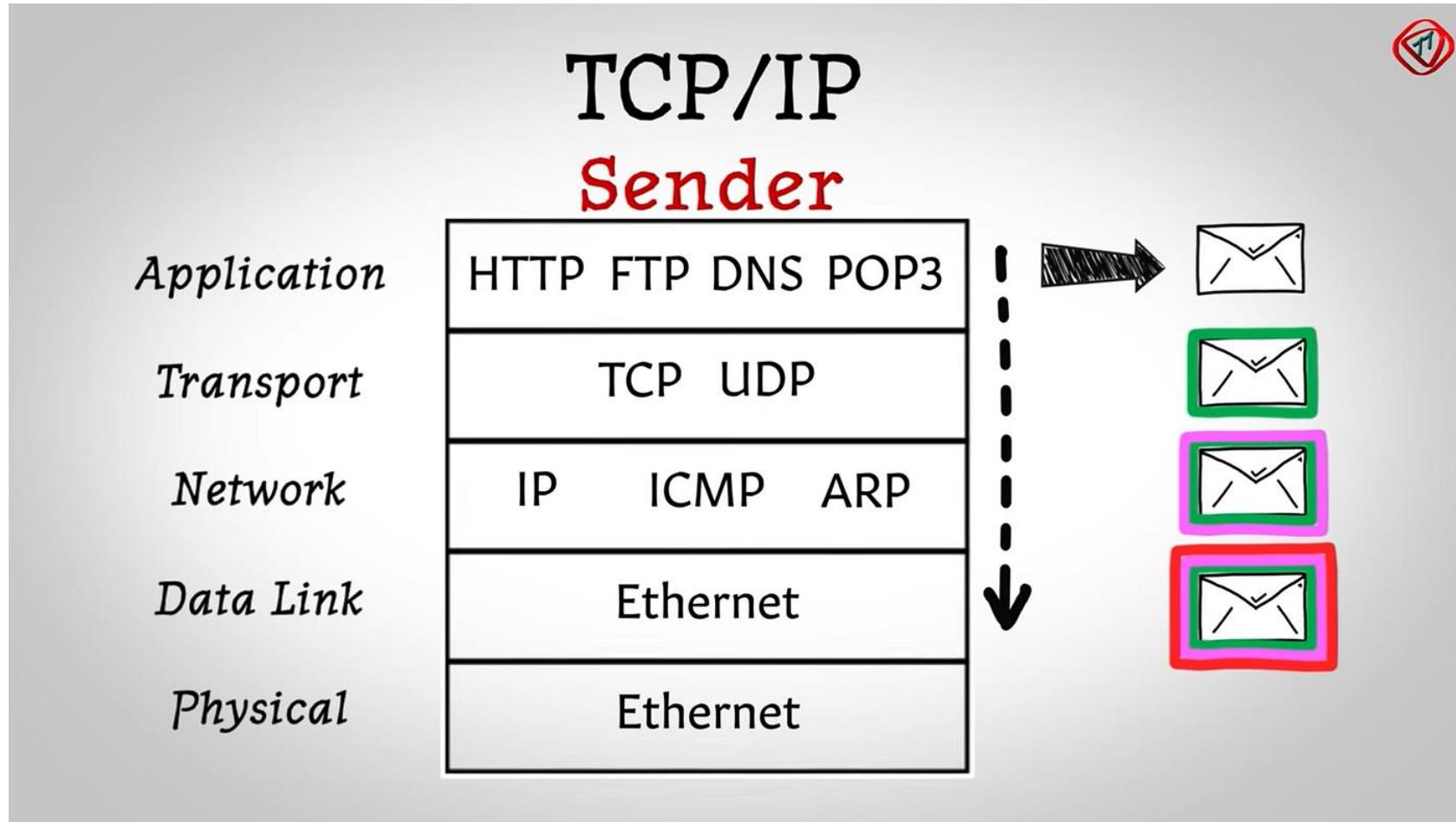
SOAP (Simple Object Access Protocol)

- **Protocolo de comunicação** usado para trocar informações estruturadas entre aplicativos na web. Ele é baseado em **XML** (Extensible Markup Language) e foi amplamente utilizado para implementar serviços web antes do advento do REST.
- É possível criar serviços SOAP em várias linguagens de programação, como Java, C#, PHP e muitas outras. Existem frameworks e bibliotecas disponíveis que facilitam a criação e o consumo de serviços SOAP, como Apache CXF, Metro, JAX-WS, entre outros.
- É possível, além do HTTP, utilizar outros protocolos, como SMPT (Simple Mail Transfer Protocol) e JMS (Java Message Service)

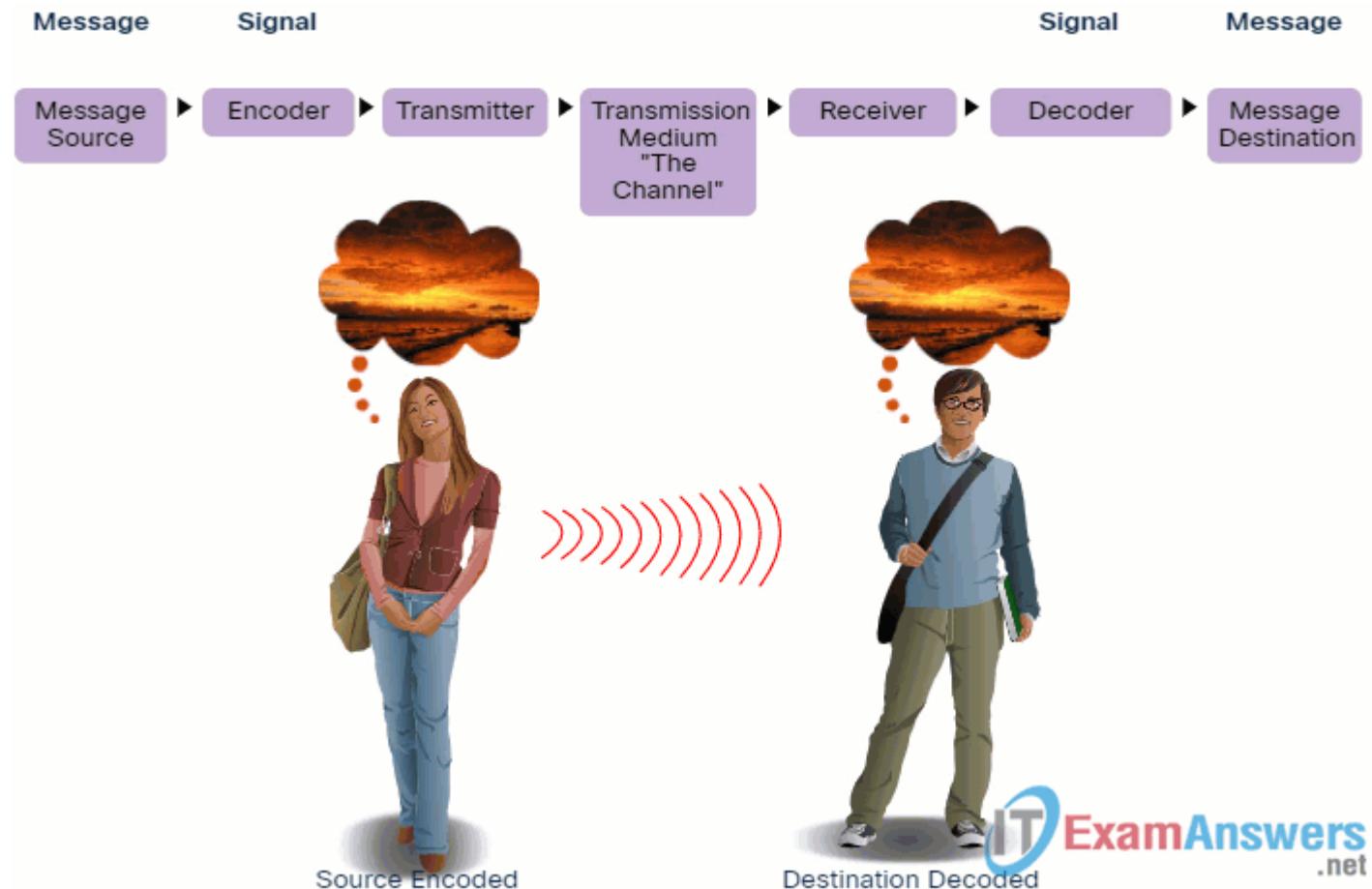
Um breve parênteses... O que é Protocolo de comunicação???

- Os protocolos de comunicação em TI (Tecnologia da Informação) são conjuntos de regras e padrões que permitem a troca de dados entre dispositivos em uma rede.
- Eles garantem que a comunicação seja eficiente, segura e compreensível para todas as partes envolvidas.
- Resumindo seria equivalente a duas pessoas conversando em inglês. Mesmo que uma pessoa seja brasileiro e outra japonês, se ambas entendem inglês, irão poder se comunicar. A língua inglês seria o protocolo adotado para conversarem.

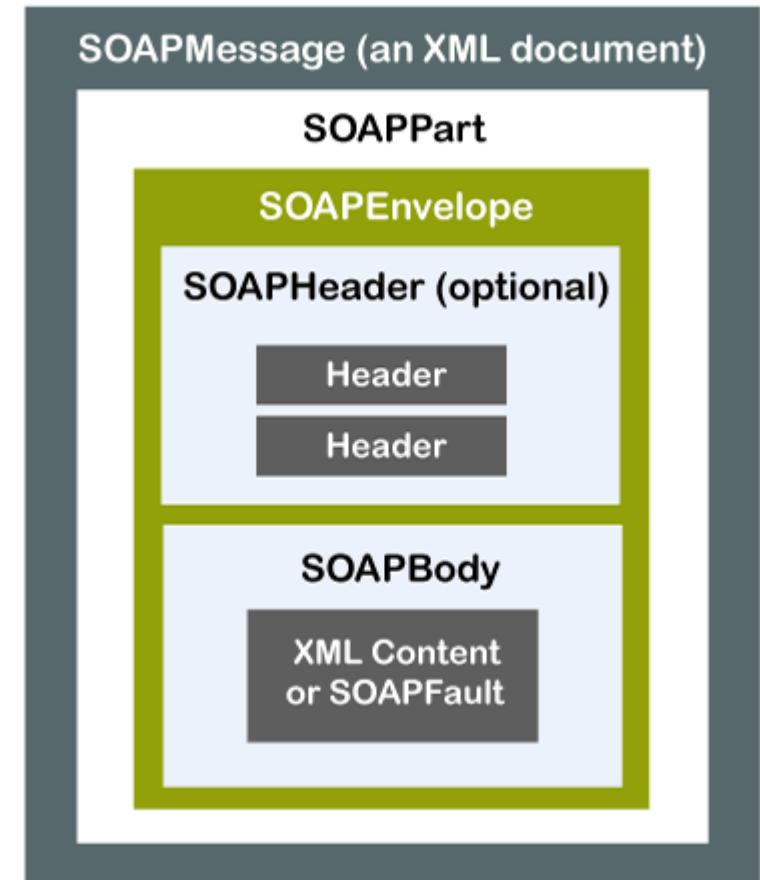
Um breve parênteses... O que é Protocolo de comunicação???



Um breve parênteses... O que é Protocolo de comunicação???



- O SOAP define uma estrutura para formatar mensagens de solicitação e resposta, permitindo que os aplicativos se comuniquem de forma padronizada e interoperável. O protocolo é baseado em troca de mensagens, onde um aplicativo cliente envia uma solicitação SOAP para um aplicativo servidor, que processa a solicitação e retorna uma resposta SOAP correspondente.
- Uma mensagem SOAP consiste em um envelope SOAP, que envolve os dados a serem transmitidos, e possui uma estrutura hierárquica definida pelo esquema XML. O envelope SOAP contém elementos como o cabeçalho (header), que pode conter informações adicionais sobre a mensagem, e o corpo (body), que carrega os dados propriamente ditos.



Principais pontos negativos do SOAP

- SOAP possui algumas características que podem torná-lo mais complexo em comparação com outras alternativas, como REST.
 - Uso extensivo de XML pode aumentar o tamanho das mensagens, resultando em uma sobrecarga na comunicação.
 - Abordagem baseada em operações e contratos do SOAP pode adicionar uma camada de complexidade à implementação e ao consumo de serviços.
 - WSDL (Web Services Description Language)

```
xml
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="ExemploServiço"
    targetNamespace="http://www.exemplo.com/servico"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://www.exemplo.com/servico"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

    <types>
        <xsd:schema targetNamespace="http://www.exemplo.com/servico">
            <xsd:element name="NomeRequest">
                <xsd:complexType>
                    <xsd:sequence>
                        <xsd:element name="nome" type="xsd:string"/>
                    </xsd:sequence>
                </xsd:complexType>
            </xsd:element>

            <xsd:element name="NomeResponse">
                <xsd:complexType>
                    <xsd:sequence>
                        <xsd:element name="mensagem" type="xsd:string"/>
                    </xsd:sequence>
                </xsd:complexType>
            </xsd:element>
        </xsd:schema>
    </types>
    <message name="SaudarRequest">
        <part name="parameters" element="tns:NomeRequest"/>
    </message>

    <message name="SaudarResponse">
        <part name="parameters" element="tns:NomeResponse"/>
    </message>

    <portType name="ExemploPortType">
        <operation name="Saudar">
            <input message="tns:SaudarRequest" />
            <output message="tns:SaudarResponse" />
        </operation>
    </portType>

    <binding name="ExemploBinding" type="tns:ExemploPortType">
        <soap:binding style="document" transport="http://schemas.xmlsoap.org/soa
        <operation name="Saudar">
            <soap:operation soapAction="http://www.exemplo.com/servico/Saudar" />
            <input>
                <soap:body use="literal" />
            </input>
            <output>
                <soap:body use="literal" />
            </output>
        </operation>
    </binding>

    <service name="ExemploService">
        <port name="ExemploPort" binding="tns:ExemploBinding">
            <soap:address location="http://www.exemplo.com/servico" />
        </port>
    </service>
</definitions>
```

Regenerate response

Exemplo simples de WSDL gerado pelo chat GPT

Neste exemplo, o WSDL descreve um serviço chamado "ExemploServiço".

Ele possui uma operação chamada "Saudar", que recebe um nome como entrada e retorna uma mensagem como saída.

O WSDL define as mensagens "SaudarRequest" e "SaudarResponse" com seus respectivos elementos de dados.

O WSDL também especifica o tipo de transporte usado (HTTP) e o estilo de ligação (document) para o serviço. Além disso, ele define o local do serviço, que neste caso é ["http://www.exemplo.com/servico"](http://www.exemplo.com/servico).

Essa é apenas uma representação simplificada de um arquivo WSDL. Na prática, arquivos WSDL podem ser mais complexos, incluindo a descrição de mais operações, tipos de dados personalizados e outros detalhes relevantes para o serviço web.

xml

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="ExemploServico"
    targetNamespace="http://www.exemplo.com/servico"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://www.exemplo.com/servico"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

    <types>
        <xsd:schema targetNamespace="http://www.exemplo.com/servico">
            <xsd:element name="NomeRequest">
                <xsd:complexType>
                    <xsd:sequence>
                        <xsd:element name="nome" type="xsd:string"/>
                    </xsd:sequence>
                </xsd:complexType>
            </xsd:element>

            <xsd:element name="NomeResponse">
                <xsd:complexType>
                    <xsd:sequence>
                        <xsd:element name="mensagem" type="xsd:string"/>
                    </xsd:sequence>
                </xsd:complexType>
            </xsd:element>
        </xsd:schema>
    </types>
```

```
<message name="SaudarRequest">
    <part name="parameters" element="tns:NomeRequest"/>
</message>

<message name="SaudarResponse">
    <part name="parameters" element="tns:NomeResponse"/>
</message>

<portType name="ExemploPortType">
    <operation name="Saudar">
        <input message="tns:SaudarRequest"/>
        <output message="tns:SaudarResponse"/>
    </operation>
</portType>

<binding name="ExemploBinding" type="tns:ExemploPortType">
    <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>

    <operation name="Saudar">
        <soap:operation soapAction="http://www.exemplo.com/servico/Saudar"/>
        <input>
            <soap:body use="literal"/>
        </input>
        <output>
            <soap:body use="literal"/>
        </output>
    </operation>
</binding>

<service name="ExemploService">
    <port name="ExemploPort" binding="tns:ExemploBinding">
        <soap:address location="http://www.exemplo.com/servico"/>
    </port>
</service>
</definitions>
```

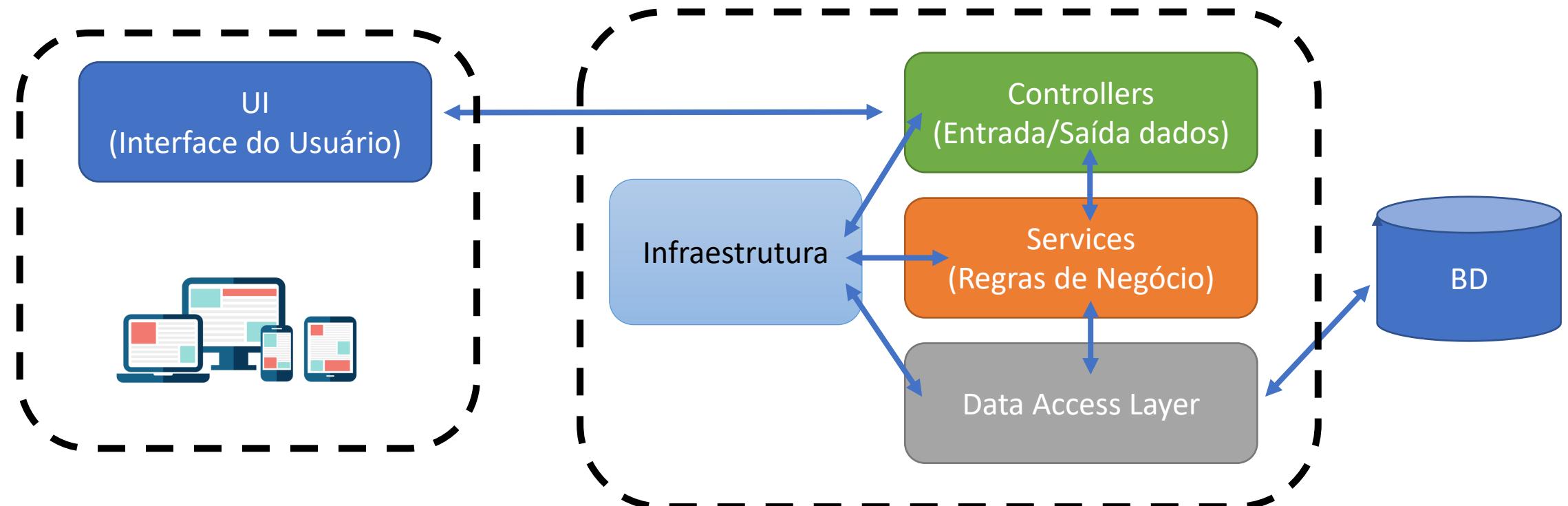
 Regenerate response

Conclusão - SOAP

- SOAP ainda é utilizado em alguns cenários específicos, como em serviços legados ou quando requisitos específicos de segurança e confiabilidade são necessários.
- Porém, a arquitetura REST (Representational State Transfer) se tornou mais popular e amplamente adotado na construção de serviços web devido à sua **simplicidade** e ao uso de padrões **HTTP**.

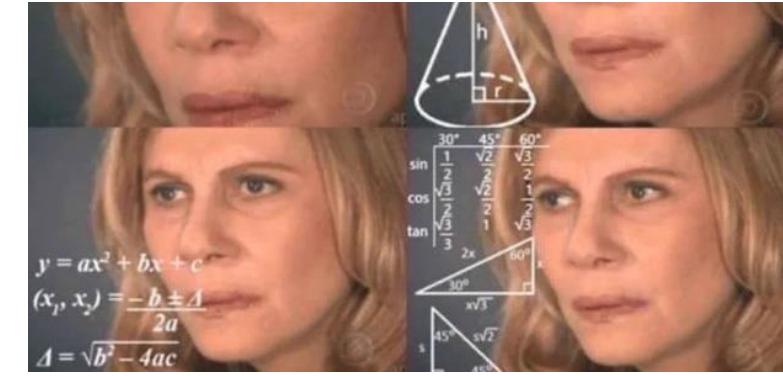
E o nosso projeto??? Como vai ser???

- API, porém **não** SOAP, e sim um **API Rest**
 - Arquitetura Cliente-Servidor em conjunto com Arquitetura de Camadas
 - Somente o lado do Servidor (Backend)



Mas para isso precisamos entender algumas coisas...

- ✓ • Linguagem Java (sintaxe e funcionamento)
- ✓ • Paradigma Orientado a Objetos (conceitos desenvolvidos em Java)
- ✓ • O que é um Backend? (Arquitetura de Sistemas)
- ✓ • O que é uma API?
- O que é API Rest? (Padrão arquitetural RestFull)
 - O que é e como funciona o HTTP?
- O que é Spring Framework?
- O que é UUID?
- O que é JSON?





REST (Representational State Transfer)

- Estilo arquitetural utilizado para projetar serviços web que sejam escaláveis, flexíveis e interoperáveis. Ele se baseia em princípios **simples** e utiliza os recursos da web de forma nativa.
 - NÃO É UM PROTOCOLO
- Em vez de usar protocolos complexos e especializados, como o SOAP (Simple Object Access Protocol), REST utiliza os métodos e recursos básicos do protocolo HTTP para criar APIs e serviços web.

HTTP

Antes de continuarmos estudando o padrão arquitetural Rest, é preciso entendermos um pouco sobre HTTP

Protocolo de Transferência de Hipertexto (HTTP)



Uma visão geral do HTTP

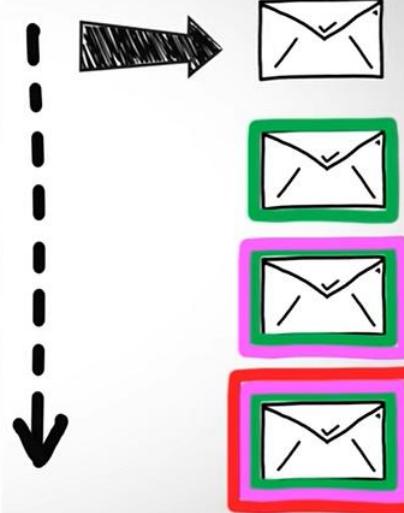
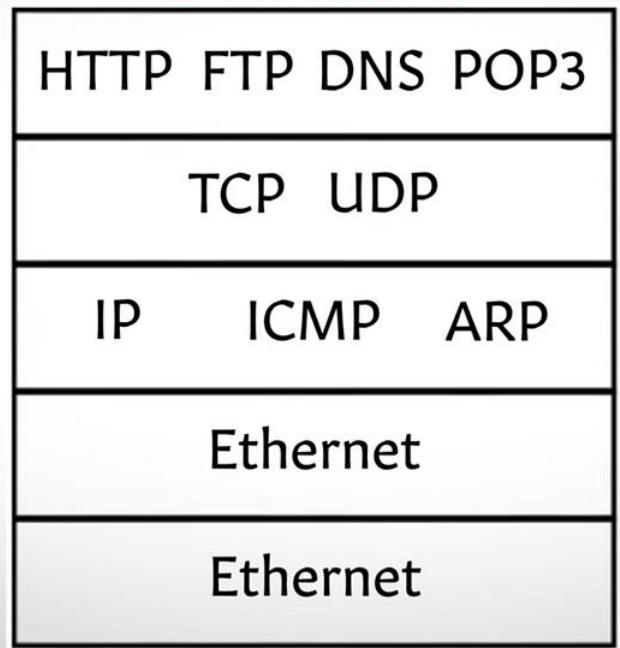
- Protocolo de Transferência de Hipertexto, conhecido como HTTP (Hypertext Transfer Protocol), é um protocolo de comunicação utilizado para transferir dados na World Wide Web (WWW).
- Base da comunicação entre um cliente (geralmente um navegador da web ou um aplicação Mobile) e um servidor web.
- Protocolo de camada de aplicação, o que significa que opera no topo do conjunto de protocolos TCP/IP, que é a base da internet.
- A principal finalidade do HTTP é permitir que os clientes solicitem recursos, como páginas da web, imagens, vídeos e outros tipos de conteúdo, a partir de servidores web.

TCP/IP

Sender



Application
Transport
Network
Data Link
Physical



Tim Berners-lee

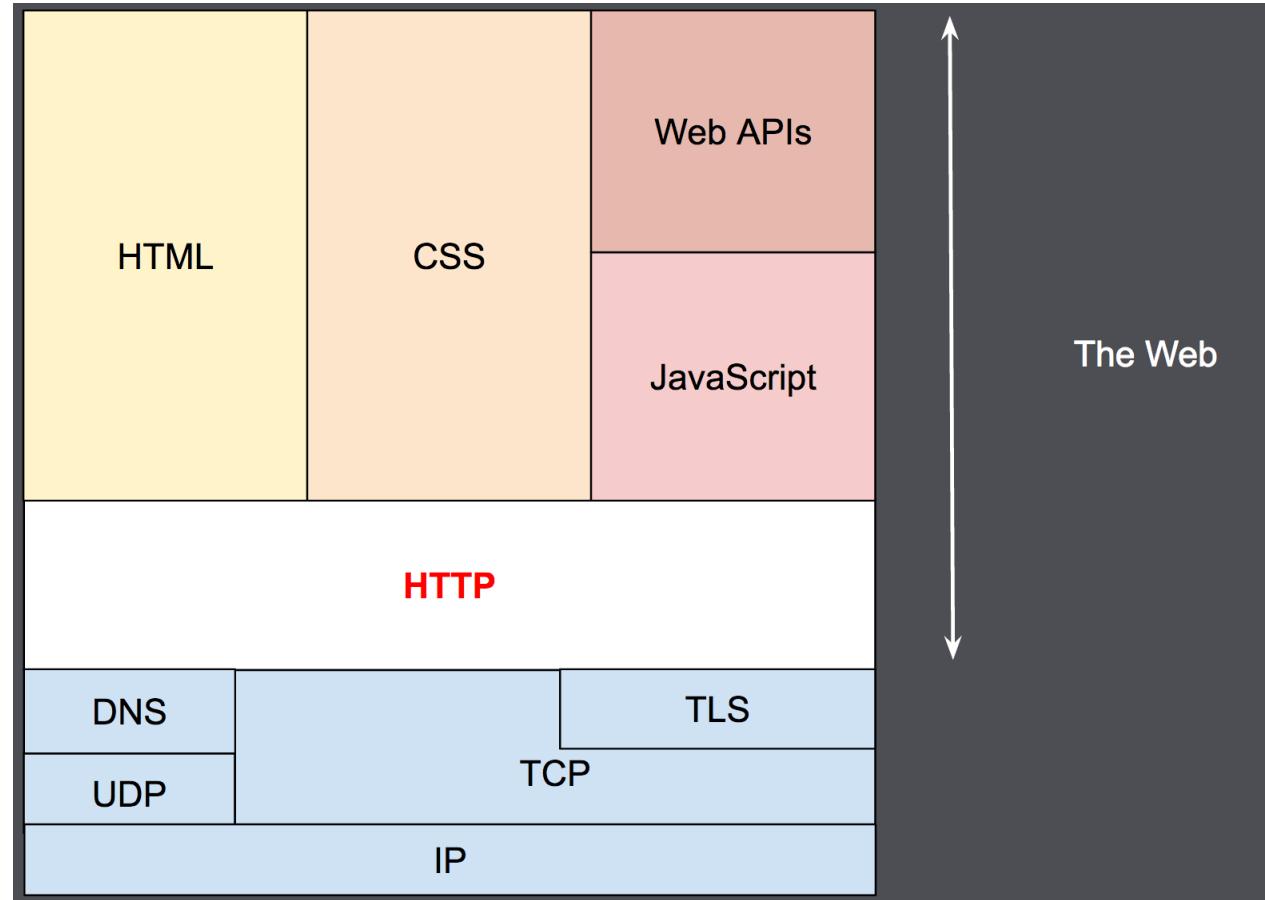


- Cientista da computação britânico que inventou a World Wide Web e desenvolveu as tecnologias fundamentais que a tornaram possível.
- Criou o HTTP como parte de seu trabalho no CERN (Organização Europeia para a Pesquisa Nuclear) nos anos 1980.
- Em 1989, Berners-Lee escreveu uma proposta chamada "Information Management: A Proposal", que descrevia um sistema para gerenciar e compartilhar informações usando hipertexto. Esse sistema evoluiu para a World Wide Web, e em 1990, Berners-Lee e seu colega Robert Cailliau publicaram formalmente uma proposta detalhada para o que se tornou a World Wide Web.
- O HTTP, junto com o HTML (Hypertext Markup Language) e o URI (Uniform Resource Identifier), foram componentes fundamentais da invenção de Berners-Lee. Sua visão de uma web interconectada permitiu que informações fossem compartilhadas e acessadas de maneira ampla, transformando a maneira como as pessoas interagem com a informação e revolucionando a sociedade e a tecnologia.
- Portanto, Tim Berners-Lee é amplamente reconhecido como o pioneiro e "pai" do HTTP e da World Wide Web.

Fonte: <https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Overview>

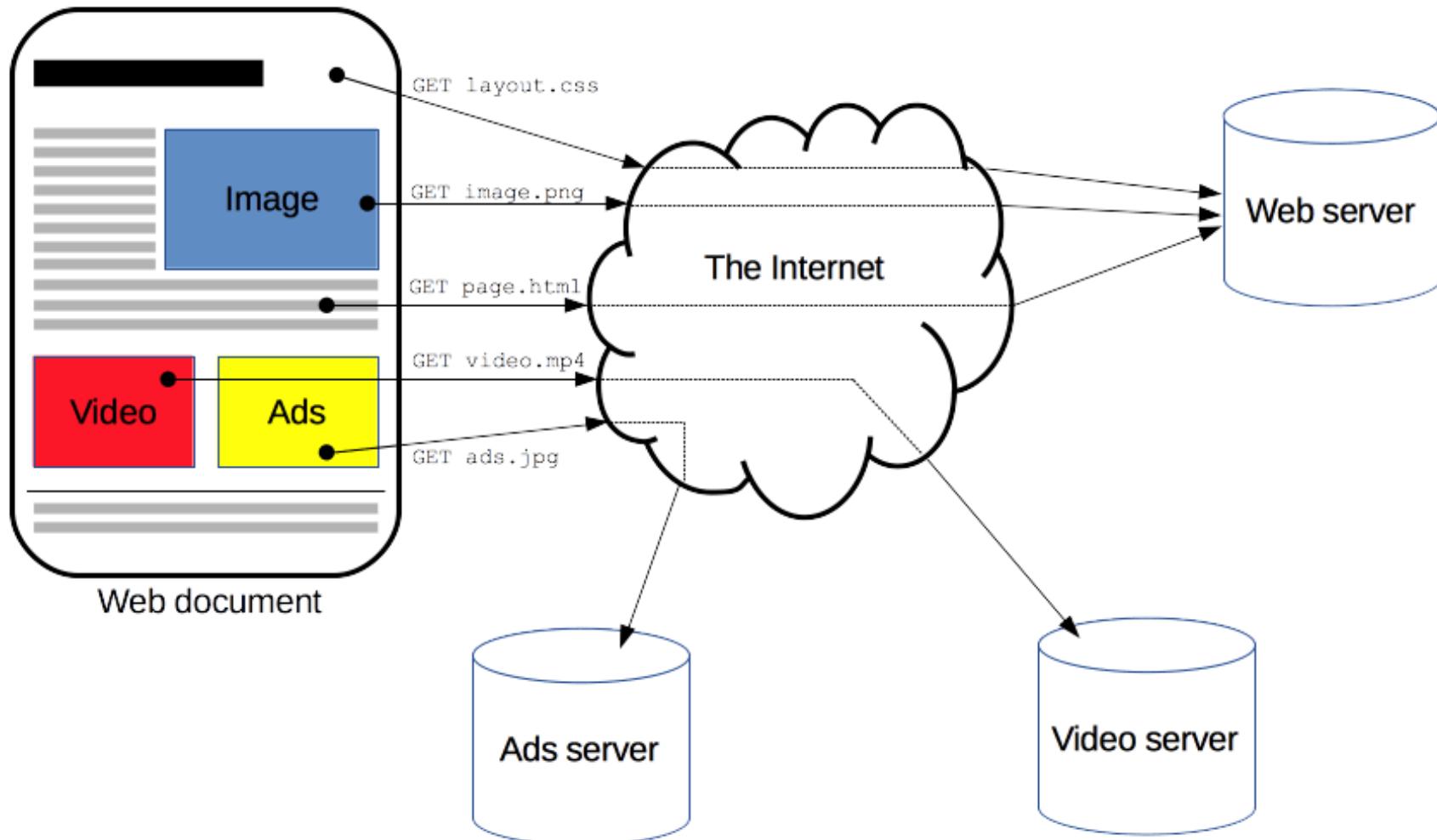


Uma visão geral do HTTP



Fonte: <https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Overview>

Uma visão geral do HTTP



Fonte: <https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Overview>

Para carregar uma página, várias requisições são realizadas.

The screenshot shows a browser window displaying the ATITUS Educação website (<https://www.atitus.edu.br>). The page features a large banner with the text "ASSUMA O CONTROLE DO SEU FUTURO E COMECE AGORA". The Network tab of the developer tools is active, showing a list of 96 requests made during the page load. The table includes columns for Status, Método, Arquivo, URL, Iniciador, Tipo, Transferido, Tamanho, and Duration. Most requests are successful (200 status code) and are of type GET. The requests include various files such as CSS, JS, and images, along with chunks and scripts for the Next.js application. The total transfer size is 12,57 MB, and the total load time is 5,20 s.

Status	Método	Arquivo	URL	Iniciador	Tipo	Transferido	Tamanho	0 ms	5,12 s
200	GET	/	https://www.atitus.edu.br/	document	html	54,09 kB	516,14 kB	507 ms	
200	GET	9c7b0252171d42a9.css	https://www.atitus.edu.br/_next/static/css/9c7b0252171d42a9.css	stylesheet	css	912 B	365 B	26 ms	
200	GET	738e8f61204d6bba.css	https://www.atitus.edu.br/_next/static/css/738e8f61204d6bba.css	stylesheet	css	1,71 kB	3,57 kB	410 ms	
200	GET	webpack-8c5c0c3f68c0e85c.js	https://www.atitus.edu.br/_next/static/chunks/webpack-8c5c0c3f68c0e85c.js	script	js	2,93 kB	4,93 kB	45 ms	
200	GET	framework-0ba0ddd33199226d.js	https://www.atitus.edu.br/_next/static/chunks/framework-0ba0ddd33199226d.js	script	js	47,66 kB	140,95 kB	24 ms	
200	GET	main-4c7077cc9079bf11.js	https://www.atitus.edu.br/_next/static/chunks/main-4c7077cc9079bf11.js	script	js	31,15 kB	101,80 kB	24 ms	
200	GET	_app-1e99834d85520fc7.js	https://www.atitus.edu.br/_next/static/chunks/pages/_app-1e99834d85520fc7.js	script	js	195,93 kB	634,31 kB	24 ms	
200	GET	b637e9a5-23b066982ed14374.js	https://www.atitus.edu.br/_next/static/chunks/b637e9a5-23b066982ed14374.js	script	js	32,92 kB	87,59 kB	126 ms	
200	GET	174-cc415934023b003c.js	https://www.atitus.edu.br/_next/static/chunks/174-cc415934023b003c.js	script	js	50,74 kB	175,02 kB	188 ms	
200	GET	916-79365548d3dd02d5.js	https://www.atitus.edu.br/_next/static/chunks/916-79365548d3dd02d5.js	script	js	90,85 kB	317,65 kB	227 ms	
200	GET	183-a08a9c4d1247b802.js	https://www.atitus.edu.br/_next/static/chunks/183-a08a9c4d1247b802.js	script	js	6,08 kB	15,05 kB	329 ms	
200	GET	762-51bc6c278552aa1c.js	https://www.atitus.edu.br/_next/static/chunks/762-51bc6c278552aa1c.js	script	js	25,65 kB	73,36 kB	349 ms	
200	GET	553-8b378216325c63db.js	https://www.atitus.edu.br/_next/static/chunks/553-8b378216325c63db.js	script	js	7,63 kB	23,16 kB	368 ms	
200	GET	638-733014086cefef94f.js	https://www.atitus.edu.br/_next/static/chunks/638-733014086cefef94f.js	script	js	5,43 kB	13,53 kB	367 ms	
200	GET	483-2e89cf1d41d17147.js	https://www.atitus.edu.br/_next/static/chunks/483-2e89cf1d41d17147.js	script	js	7,37 kB	24,48 kB	365 ms	

96 requisições | 12,57 MB / 9,01 MB transferidos | Tempo: 5,98 s | DOMContentLoaded: 484 ms | load: 5,20 s

Uma visão geral do HTTP

- **Requisições/Solicitações (Requests):** Quando um usuário digita um endereço URL em um navegador e pressiona Enter, o navegador envia uma solicitação HTTP para o servidor web associado ao URL.
 - Essa solicitação inclui informações como o método de requisição (por exemplo, GET, POST, PUT, DELETE), o caminho do recurso e a versão do protocolo.
- **Respostas (Responses):** O servidor processa a solicitação e envia uma resposta HTTP de volta ao cliente.
 - Essa resposta inclui um código de status, que indica o resultado da solicitação (por exemplo, 200 OK para sucesso, 404 Not Found para recurso não encontrado) e o conteúdo solicitado, se aplicável.

Uma visão geral do HTTP

- **URI (Uniform Resource Identifier):** Cada recurso acessível via HTTP é identificado por um URI, que é uma sequência de caracteres que fornece um endereço exclusivo para o recurso.
- **Cabeçalhos HTTP:** Tanto as solicitações quanto as respostas HTTP podem conter cabeçalhos, que fornecem informações adicionais sobre a solicitação ou a resposta. Isso inclui detalhes sobre o tipo de conteúdo, cookies, autenticação, idioma preferido, entre outros.
- **HTTPs:** O HTTP tradicional não oferece segurança na transmissão dos dados, o que levou ao desenvolvimento do HTTP seguro (HTTPS). O HTTPS utiliza criptografia SSL/TLS para proteger os dados durante a transmissão, tornando mais difícil para terceiros mal-intencionados interceptarem ou modificarem as informações.

HTTP - Evolução

- **HTTP/0.9:** Lançada em 1991.
 - Suportava apenas o método de solicitação GET.
 - Não havia cabeçalhos ou códigos de status.
- **HTTP/1.0:** Lançada em 1996.
 - Diferentes métodos de solicitação (GET, POST, HEAD)
 - Cabeçalhos para controlar o tipo de conteúdo
 - Códigos de status (como 404 para recurso não encontrado)
 - Suporte para envio de dados do cliente para o servidor usando o método POST.
- **HTTP/1.1:** Lançada em 1997;
 - Persistência de conexão (keep-alive), que permitia que várias solicitações e respostas fossem transmitidas pela mesma conexão TCP, reduzindo a latência e o tempo necessário para estabelecer conexões.
 - Cabeçalhos de cache e suporte para compressão de dados, o que melhorou ainda mais o desempenho.

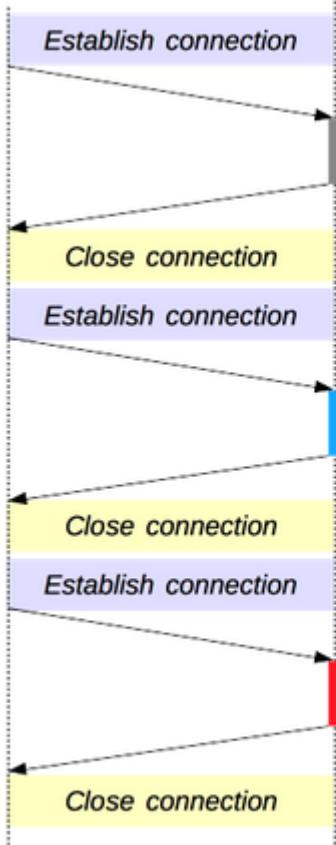
HTTP - Evolução

- **HTTP/2:** Lançada em 2015
 - Conceito de "multiplexação", permitindo que múltiplas solicitações e respostas fossem transmitidas simultaneamente pela mesma conexão, eliminando o problema de bloqueio de solicitações mais lentas.
 - Compressão de cabeçalhos, reduzindo ainda mais o tamanho das requisições e respostas
 - Carregamento incremental de recursos, melhorando o desempenho global da página.
- **HTTP/3:** Lançada em 2020
 - Utiliza um novo protocolo de transporte chamado QUIC, que é construído em cima do protocolo UDP em vez do TCP. Isso ajuda a reduzir a latência e melhorar a segurança.
 - Mantém muitos dos aprimoramentos do HTTP/2, como multiplexação e compressão de cabeçalhos, mas oferece uma experiência mais eficiente e rápida em redes com alta perda de pacotes, como redes móveis.

HTTP e conexões

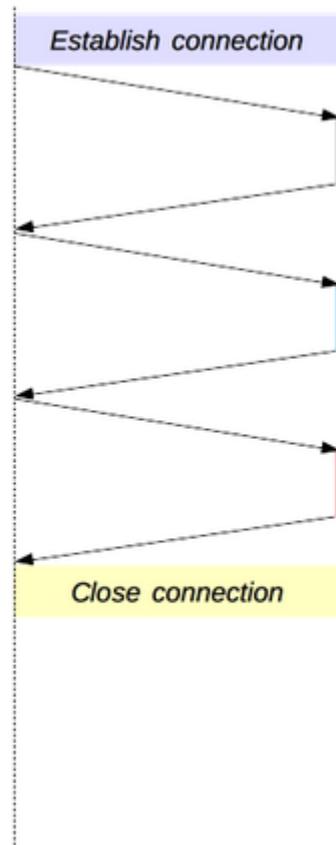
HTTP 1.0

Client Server



HTTP 1.1

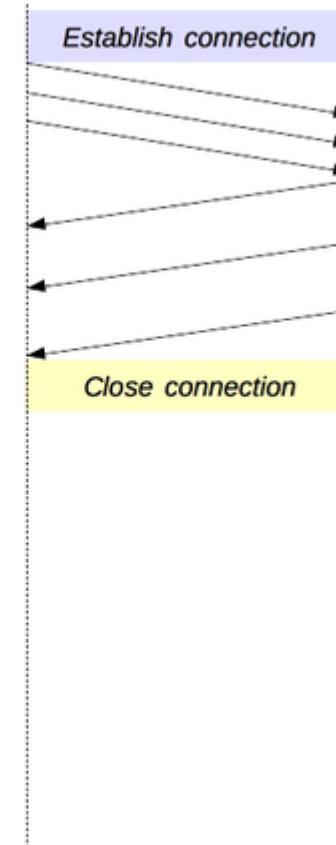
Client Server



HTTP 1.1 com pipeline

Desativado na maioria dos navegadores modernos

Client Server



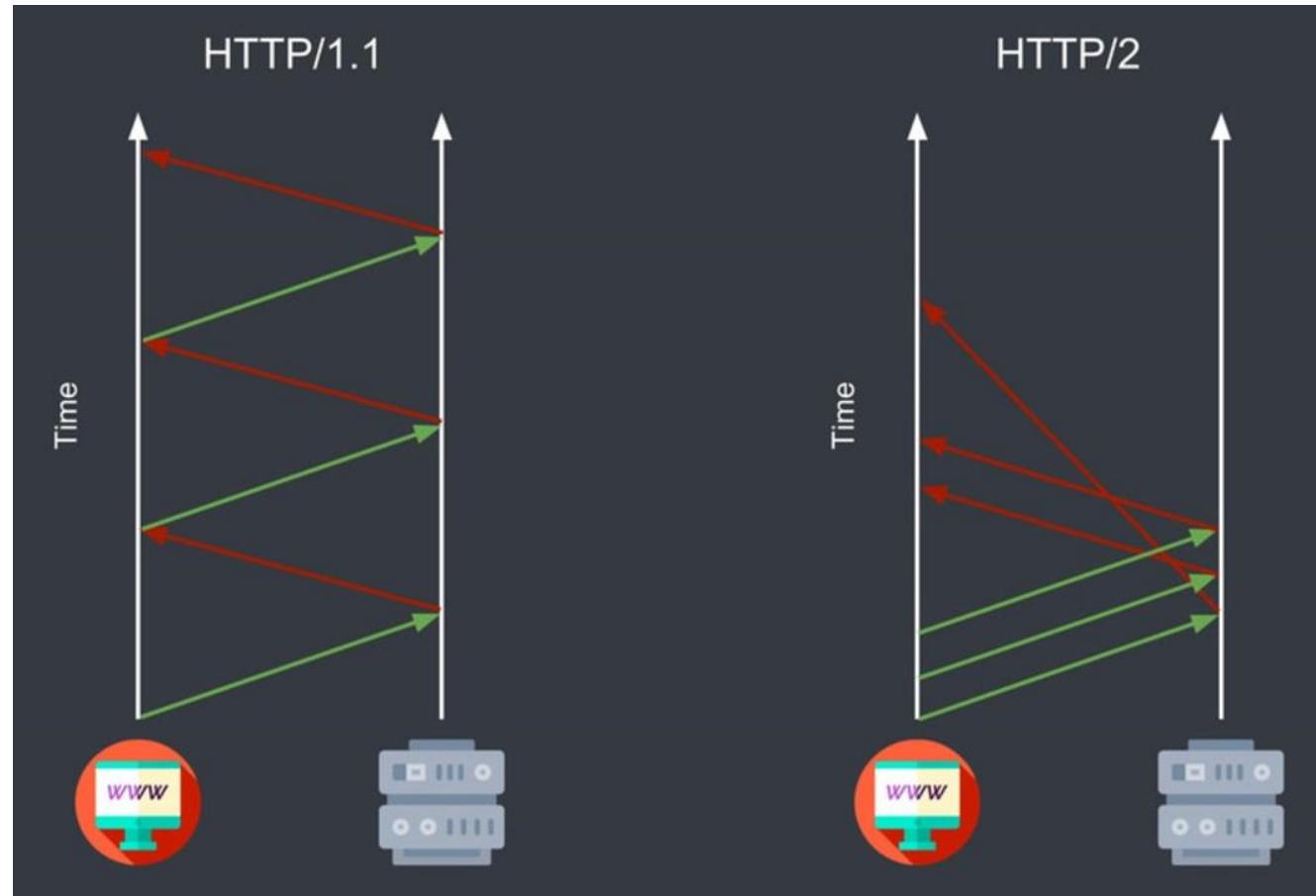
Short-lived connections

Persistent connection

HTTP Pipelining

HTTP e conexões

Multiplexação



Fonte: <https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Overview>

Aspectos básicos do HTTP

- HTTP é simples
- HTTP é extensível
- HTTP **não tem estado, mas tem sessões**

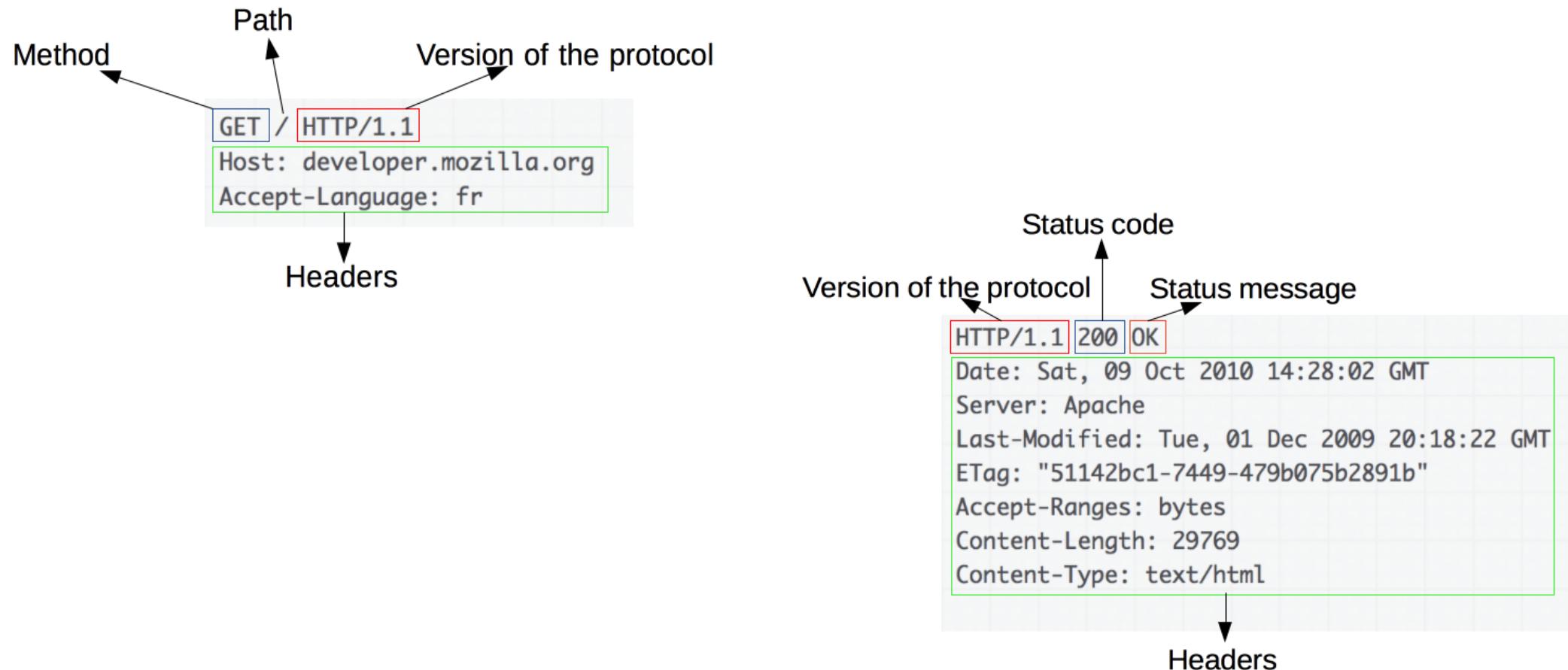
Nota: * O problema do carrinho de compras de e-commerce e o protocolo HTTP: como o protocolo HTTP não guarda o estado das requisições e respostas, é **impossível** fazer com que um site guarde as informações de um carrinho de compras **somente através do HTTP**. Por exemplo, imagine que você irá comprar um computador novo e um jogo de xícaras de chá. Para que esses dados possam ser mantidos enquanto você navega no site do e-commerce olhando mais produtos (cada página visitada gera um novo par de requisição/resposta), duas estratégias podem ser usadas, já que o HTTP por si só, não permitiria isso:

1. Você possui um cadastro no e-commerce e um programa escrito no servidor é responsável por armazenar suas informações do carrinho; ou
2. Um programa escrito em linguagem cliente (como JavaScript), gerencia essas informações através dos *cookies* e de bancos de dados que os próprios navegadores disponibilizam para as aplicações, para armazenamento **temporário** dessas informações de carrinho.

Controles via HTTP

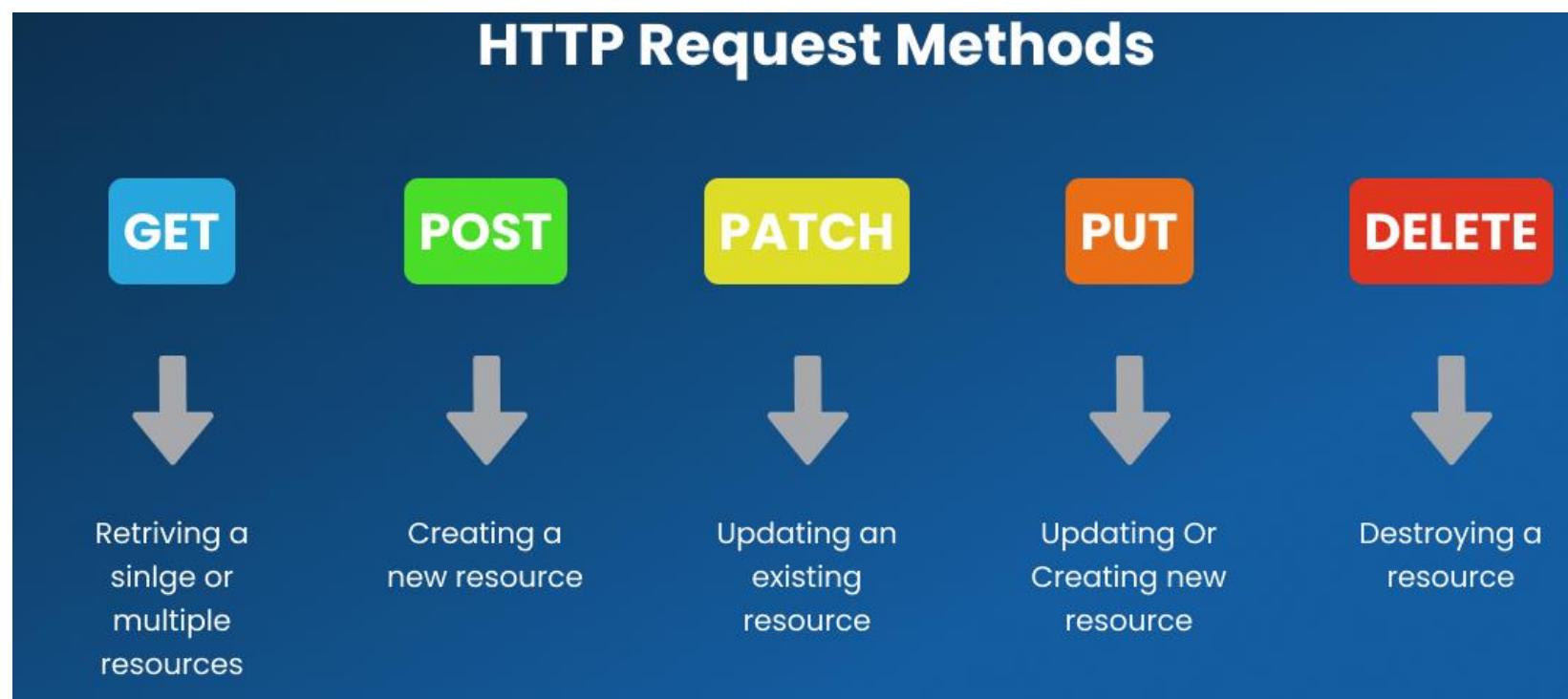
- Cache
- Relaxamento das restrições de origem
- Autenticação
- Proxy e tunelamento
- Sessões usando os cookies HTTP

Fluxo HTTP



Métodos HTTP

- Métodos HTTP, também conhecidos como verbos HTTP, são utilizados para indicar a ação que o cliente deseja realizar em um recurso específico no servidor.
- Cada método tem um propósito diferente e desencadeia uma ação específica do servidor.



Métodos HTTP

- **GET:** O método GET é usado para solicitar um recurso específico do servidor. Quando um cliente faz uma solicitação GET, ele está pedindo ao servidor para enviar o conteúdo do recurso solicitado de volta ao cliente. É importante observar que solicitações GET não devem ter efeitos colaterais no servidor, ou seja, não devem modificar o estado do servidor ou dos recursos.
- **POST:** O método POST é usado para enviar dados do cliente para o servidor, geralmente em um formulário HTML. O servidor processa os dados enviados e pode realizar uma ação com base nesses dados, como salvar informações em um banco de dados. Diferentemente do método GET, as solicitações POST podem ter efeitos colaterais no servidor, alterando seu estado.
- **DELETE:** O método DELETE é usado para solicitar a remoção de um recurso específico no servidor. O servidor processa a solicitação e remove o recurso, se existir. Assim como o método GET, o DELETE também não deve ter efeitos colaterais no servidor.

Métodos HTTP

- **PUT:** O método PUT é usado para enviar dados ao servidor para criar ou atualizar um recurso específico no servidor. Se o recurso já existir, o servidor atualiza os dados com os novos dados fornecidos na solicitação. Se o recurso não existir, o servidor pode criar um novo recurso com os dados fornecidos.
- **PATCH:** O método PATCH é usado para aplicar modificações parciais a um recurso. Ele é geralmente usado quando um cliente deseja atualizar apenas uma parte específica do recurso, sem enviar todas as informações novamente.
- **HEAD:** O método HEAD é semelhante ao GET, mas solicita apenas os cabeçalhos da resposta, sem o conteúdo real. Isso é útil quando o cliente deseja obter informações sobre um recurso, como seu tamanho ou tipo de conteúdo, sem baixar todo o conteúdo.

Códigos de Resposta HTTP

- Claro, os códigos de resposta HTTP são códigos numéricos que um servidor web retorna em resposta a uma solicitação feita por um cliente.
- Esses códigos indicam o resultado da solicitação e permitem que o cliente entenda o que aconteceu com sua requisição.
- Os códigos de resposta são divididos em cinco classes, cada uma com um intervalo numérico específico

HTTP Status Codes



Códigos de Status nas Respostas HTTP

1. Respostas de informação (100 - 199),
2. Respostas de sucesso (200 - 299),
3. Redirecionamentos (300 - 399)
4. Erros do cliente (400 - 499)
5. Erros do servidor (500 - 599).

Os status abaixo são definidos pela [seção 10 da RFC 2616](#). Você pode encontrar uma versão atualizada da especificação na [RFC 7231](#).

i **Nota:** Se você receber uma resposta que não está nesta lista, é uma resposta não padrão, provavelmente personalizada pelo software do servidor.

Códigos de Status nas Respostas HTTP

Code	Reason-Phrase	Defined in...				
100	Continue	Section 6.2.1	400	Bad Request	Section 6.5.1	
101	Switching Protocols	Section 6.2.2	401	Unauthorized	Section 3.1 of [RFC7235]	
200	OK	Section 6.3.1	402	Payment Required	Section 6.5.2	
201	Created	Section 6.3.2	403	Forbidden	Section 6.5.3	
202	Accepted	Section 6.3.3	404	Not Found	Section 6.5.4	
203	Non-Authoritative Information	Section 6.3.4	405	Method Not Allowed	Section 6.5.5	
204	No Content	Section 6.3.5	406	Not Acceptable	Section 6.5.6	
205	Reset Content	Section 6.3.6	407	Proxy Authentication Required	Section 3.2 of [RFC7235]	
206	Partial Content	Section 4.1 of [RFC7233]	408	Request Timeout	Section 6.5.7	
300	Multiple Choices	Section 6.4.1	409	Conflict	Section 6.5.8	
301	Moved Permanently	Section 6.4.2	410	Gone	Section 6.5.9	
302	Found	Section 6.4.3	411	Length Required	Section 6.5.10	
303	See Other	Section 6.4.4	412	Precondition Failed	Section 4.2 of [RFC7232]	
304	Not Modified	Section 4.1 of [RFC7232]	413	Payload Too Large	Section 6.5.11	
305	Use Proxy	Section 6.4.5	414	URI Too Long	Section 6.5.12	
307	Temporary Redirect	Section 6.4.7	415	Unsupported Media Type	Section 6.5.13	
			416	Range Not Satisfiable	Section 4.4 of [RFC7233]	
			417	Expectation Failed	Section 6.5.14	
			426	Upgrade Required	Section 6.5.15	
			500	Internal Server Error	Section 6.6.1	
			501	Not Implemented	Section 6.6.2	
			502	Bad Gateway	Section 6.6.3	
			503	Service Unavailable	Section 6.6.4	
			504	Gateway Timeout	Section 6.6.5	
			505	HTTP Version Not Supported	Section 6.6.6	

Response Status Code mais utilizados

200 OK

Estas requisição foi bem sucedida. O significado do sucesso varia de acordo com o método HTTP:

201 Created

A requisição foi bem sucedida e um novo recurso foi criado como resultado. Esta é uma tipica resposta enviada após uma requisição POST.

400 Bad Request

Essa resposta significa que o servidor não entendeu a requisição pois está com uma sintaxe inválida.

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 portanto o servidor está rejeitando dar a resposta. Diferente do código 401, aqui a identidade do cliente é conhecida.

500 Internal Server Error

O servidor encontrou uma situação com a qual não sabe lidar.

Server HTTP com Java através de Socket

```
1 import java.io.IOException;
2 import java.net.ServerSocket;
3 import java.net.Socket;
4
5 public class SimpleHTTPServer {
6     public static void main(String[] args) {
7         int port = 80;
8         try {
9             // Instancia um novo objeto ServerSocket o qual já abre a porta TCP definida
10            ServerSocket serverSocket = new ServerSocket(port);
11            System.out.println("Server rodando na porta " + port);
12            while (true) { //loop para receber várias conexões
13                /*
14                 * Aguarda uma requisição (request),
15                 * ao receber é criado um novo thread para lidar com a solicitação
16                 */
17                Socket clientSocket = serverSocket.accept();
18                System.out.println("Recebeu conexão");
19            }
20        } catch (IOException e) {
21            e.printStackTrace();
22        }
23    }
24 }
```

Server HTTP com Java – Hello World

```
private static void handleRequest(Socket clientSocket) throws IOException {
    /*InputStreamReader = Converte os dados brutos em caracteres
     * BufferedReader = Fornece buffering para melhorar o desempenho na leitura
     */
    BufferedReader in = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
    /*Declara a variável apontando para o response da solicitação */
    OutputStream out = clientSocket.getOutputStream();
    String requestLine = in.readLine();
    if (requestLine != null) {
        System.out.println("Received request: " + requestLine);
        // Send a basic HTTP response
        String response = "HTTP/1.1 200 OK\r\nContent-Length: 12\r\n\r\nHello, World!";
        out.write(response.getBytes());
    }
    out.close();
    in.close();
    clientSocket.close();
}
```

Server HTTP com Java – Vários Métodos

```
private static void handleRequest(Socket clientSocket) throws IOException {
    /*
     * InputStreamReader = Converte os dados brutos em caracteres
     * BufferedReader = Fornece buffering para melhorar o desempenho na leitura
     */
    BufferedReader in = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
    /* Declara a variável apontando para o response da solicitação */
    OutputStream out = clientSocket.getOutputStream();
    String requestLine = in.readLine();
    String response = "";
    if (requestLine != null) {
        System.out.println("Received request: " + requestLine);
        if (requestLine.startsWith("GET")) {
            // Send a basic HTTP response
            response = "HTTP/1.1 200 OK\r\nContent-Length: 12\r\n\r\nHello, GET!";
        } else {
            response = "HTTP/1.1 405 Method Not Allowed\r\n\r\n";
        }
        out.write(response.getBytes());
    }
    out.close();
    in.close();
    clientSocket.close();
}
```

Server HTTP com Java – Recebendo Dados

```
} else if (requestLine.startsWith("POST")) {
    // Read the content length from headers
    String contentLengthHeader = findHeader(in, "Content-Length");
    int contentLength = Integer.parseInt(contentLengthHeader);

    // Read the POST data from the body
    char[] postData = new char[contentLength];
    in.read(postData);

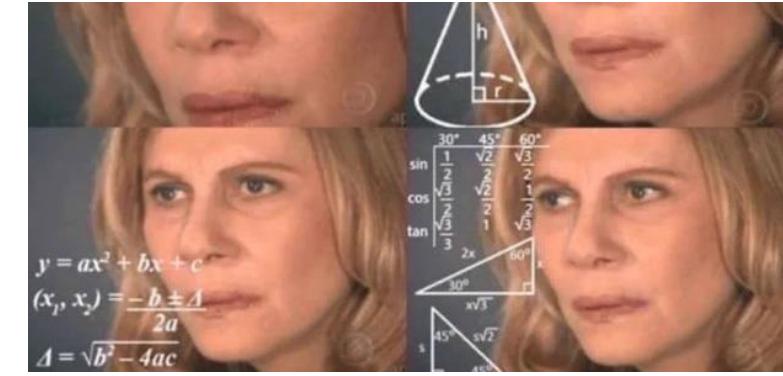
    String postDataStr = new String(postData);
    System.out.println("Received POST data: " + postDataStr);

    // Send a response for POST request
    response = "HTTP/1.1 200 OK\r\nContent-Length: 12\r\n\r\nRecebi seus dados!";
}
```

```
private static String findHeader(BufferedReader reader, String headerName) throws IOException {
    String line;
    while ((line = reader.readLine()) != null && !line.isEmpty()) {
        if (line.startsWith(headerName)) {
            String[] parts = line.split(": ");
            if (parts.length > 1) {
                return parts[1];
            }
        }
    }
    return null;
}
```

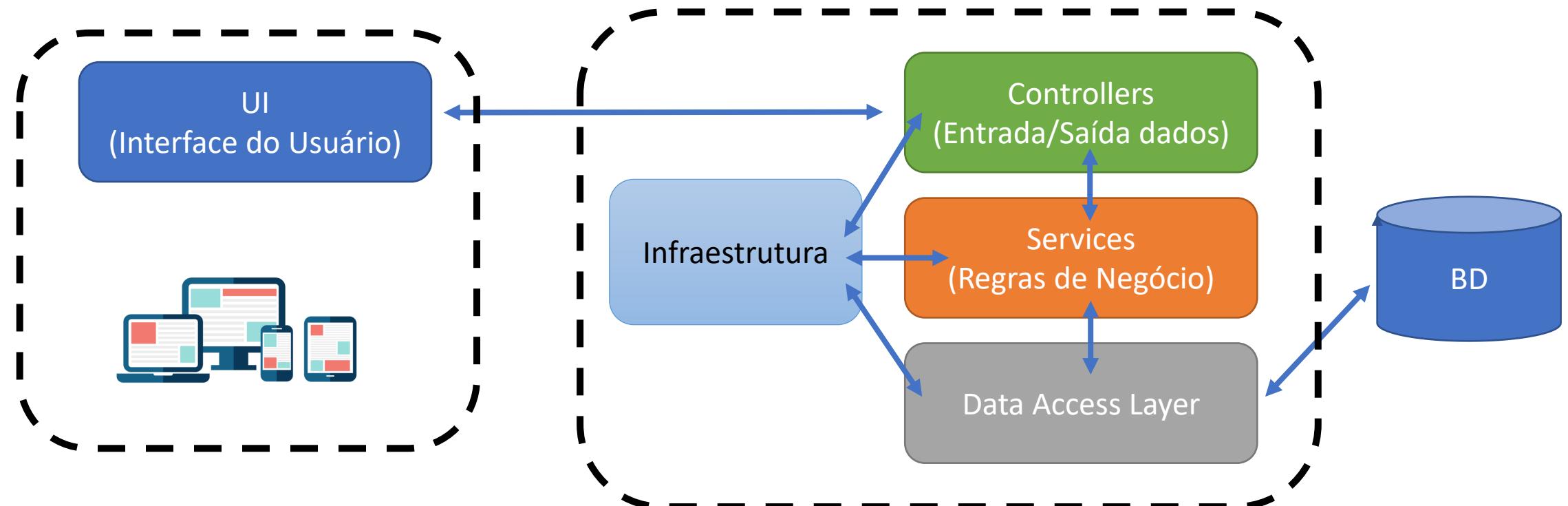
Mas para isso precisamos entender algumas coisas...

- ✓ • Linguagem Java (sintaxe e funcionamento)
- ✓ • Paradigma Orientado a Objetos (conceitos desenvolvidos em Java)
- ✓ • O que é um Backend? (Arquitetura de Sistemas)
- ✓ • O que é uma API?
- O que é API Rest? (Padrão arquitetural RestFull)
 - O que é e como funciona o HTTP?
- O que é Spring Framework?
- O que é UUID?
- O que é JSON?



E o nosso projeto??? Como vai ser???

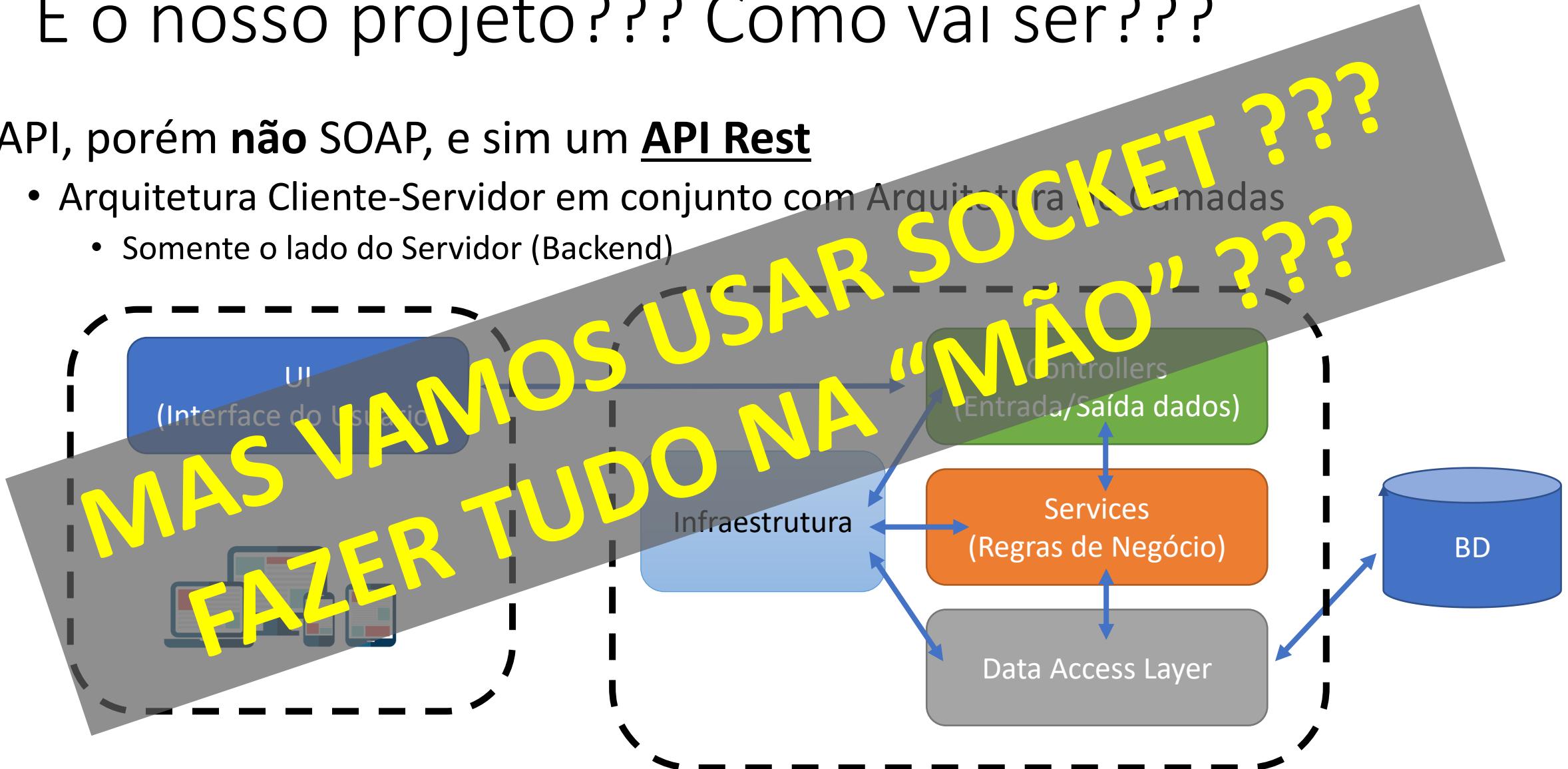
- API, porém **não** SOAP, e sim um **API Rest**
 - Arquitetura Cliente-Servidor em conjunto com Arquitetura de Camadas
 - Somente o lado do Servidor (Backend)



E o nosso projeto??? Como vai ser???

- API, porém **não** SOAP, e sim um API Rest

- Arquitetura Cliente-Servidor em conjunto com Arquitetura de Camadas
 - Somente o lado do Servidor (Backend)



Vamos utilizar Sockets direto no Java???

```
package br.edu.atitus.denguealertaexp;
import java.io.*;
import java.net.*;

public class SimpleHTTPServer {
    public static void main(String[] args) {
        try {
            // Cria um servidor socket na porta 8080
            ServerSocket serverSocket = new ServerSocket(8080);
            System.out.println("Servidor HTTP iniciado na porta 8080...");

            while (true) {
                // Aceita uma conexão
                Socket clientSocket = serverSocket.accept();

                // Processa a conexão em uma nova thread
                new Thread(new ClientHandler(clientSocket)).start();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```



Apache Tomcat

- É um servidor de aplicação web de código aberto que implementa as tecnologias Java Servlet e JavaServer Pages (JSP). Ele é desenvolvido pela Apache Software Foundation e é amplamente utilizado para hospedar e executar aplicativos web Java.



Apache Tomcat x Sockets

- **Usando Sockets para Abrir uma Porta HTTP:**
- **Vantagens:**
 - Controle total: Ao usar sockets diretamente, você tem controle total sobre o processamento das solicitações e respostas HTTP. Isso permite uma personalização completa.
 - Baixa sobrecarga: Em situações específicas, pode haver menos sobrecarga de recursos em comparação com um servidor de aplicação completo como o Tomcat, pois você pode otimizar o código exatamente conforme necessário.
- **Desvantagens:**
 - Complexidade: Implementar um servidor HTTP a partir do zero usando sockets é complexo e requer um conhecimento profundo de protocolos HTTP e gerenciamento de conexões.
 - Falta de recursos: Você terá que implementar muitos recursos comuns, como suporte a servlets, JSPs, segurança, balanceamento de carga, escalabilidade e gerenciamento de sessões por conta própria.
 - Manutenção difícil: Manter um servidor personalizado é desafiador e pode levar a problemas de segurança e compatibilidade com o tempo.



Apache Tomcat x Sockets

- **Usando o Tomcat:**

- **Vantagens:**

- Facilidade de uso: O Tomcat é uma solução pronta para uso que lida com a maior parte da complexidade de um servidor HTTP. É fácil de configurar e implantar aplicativos Java.
- Ampla gama de recursos: O Tomcat oferece suporte completo a servlets, JSPs, segurança, escalabilidade, balanceamento de carga e gerenciamento de sessões, economizando tempo e esforço no desenvolvimento.
- Comunidade e suporte: O Tomcat tem uma grande comunidade de usuários e uma vasta documentação, tornando mais fácil encontrar ajuda e soluções para problemas.

- **Desvantagens:**

- **Maior sobrecarga:** O Tomcat é uma solução mais pesada em termos de recursos em comparação com implementar um servidor HTTP personalizado usando sockets. Isso pode ser um problema em ambientes com recursos limitados.
- **Menos controle:** Embora o Tomcat seja altamente configurável, pode haver casos em que você precise de um controle mais granular, que só pode ser alcançado com um servidor personalizado.



Apache Tomcat x Sockets

- Usar o Tomcat é a escolha mais comum e recomendada para executar aplicativos Java na web devido à sua facilidade de uso, recursos prontos para uso e suporte da comunidade.
- No entanto, se você tiver necessidades muito específicas ou desejar um controle completo sobre o servidor HTTP, implementar seu próprio servidor usando sockets pode ser uma opção, embora seja uma abordagem muito mais complexa e exigente em termos de desenvolvimento e manutenção.

And now???

- Certo, vamos desenvolver um backend em Java (é claro, kkk), utilizando arquitetura de camadas, e o Tomcat para rodar a nossa aplicação!!!
- Mas como vamos fazer isso????





What Spring can do



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 Framework e Spring Boot

- O **Spring Framework** é um framework de código aberto que fornece um ambiente de desenvolvimento abrangente para aplicativos Java. Ele foi lançado pela primeira vez em 2002 e desde então se tornou uma escolha popular para o desenvolvimento de aplicativos corporativos. Algumas características-chave do Spring Framework incluem:
 - Injeção de Dependência (Dependency Injection - DI)
 - Programação Orientada a Aspectos (Aspect-Oriented Programming – AOP)
 - MVC (Model-View-Controller)
 - Integração com Banco de Dados
 - Segurança

Spring Framework e Spring Boot

- O **Spring Boot** é uma extensão do Spring Framework que simplifica muito o desenvolvimento de aplicativos Java. Ele foi projetado para acelerar o processo de configuração e desenvolvimento, permitindo que os desenvolvedores se concentrem mais na lógica de negócios e menos na configuração. Algumas características do Spring Boot incluem:

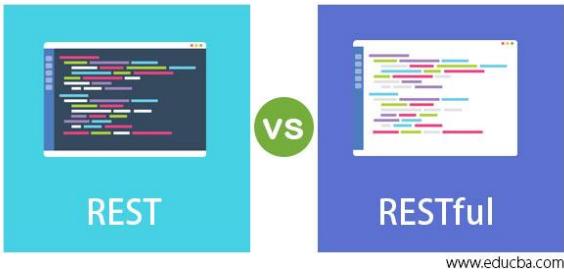
- **Configuração Automática**
- **Embedded Web Server**
- **Pronto para Produção**
- **Spring Boot Starter**
- **Facilidade de Teste**
- **Ampla Comunidade**

Um breve histórico do Spring

- Em **2002**, Rod Johnson publicou um livro que tratava das complexidades do J2EE.
- Em **2003**, Rod Johnson, Juergen Hoeller e Yann Caroff iniciaram o projeto open source Spring Framework. No ano seguinte, lançaram a versão **1.0** do framework.
- Em **2008**, o **Spring Security 2.0** foi lançado, renomeando-o de Acegi.
- Em **2011**, o Spring Data expandiu para cargas de trabalho **NoSQL** com o Spring Data MongoDB, Spring Data **Redis**, Spring Data **Neo4j** e Spring Data **GemFire**.
- Em **2014**, foi lançado o **Spring Boot 1.0**, que introduziu o desenvolvimento rápido de aplicações. A comunidade técnica destacou que o Spring passou a focar mais na tarefa em mãos do que na infraestrutura.
- Em **2015**, o **Spring Cloud 1.0** foi lançado, fornecendo ferramentas para a construção rápida de software para sistemas distribuídos.

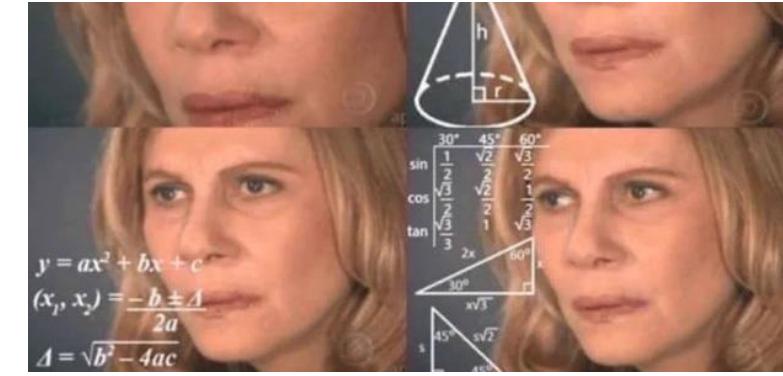


- Spring Boot**
Takes an opinionated view of building Spring applications and gets you up and running as quickly as possible.
- Spring Framework**
Provides core support for dependency injection, transaction management, web apps, data access, messaging, and more.
- Spring HATEOAS**
Simplifies creating REST representations that follow the HATEOAS principle.
- Spring REST Docs**
Lets you document RESTful services by combining hand-written documentation with auto-generated snippets produced with Spring MVC Test or REST Assured.
- Spring Security**
Protects your application with comprehensive and extensible authentication and authorization support.
- Spring Data**
Provides a consistent approach to data access – relational, non-relational, map-reduce, and beyond.



Mas para isso precisamos entender algumas coisas...

- ✓ • Linguagem Java (sintaxe e funcionamento)
- ✓ • Paradigma Orientado a Objetos (conceitos desenvolvidos em Java)
- ✓ • O que é um Backend? (Arquitetura de Sistemas)
- ✓ • O que é uma API?
- O que é API Rest? (Padrão arquitetural RestFull)
 - O que é e como funciona o HTTP?
- ✓ • O que é Spring Framework?
- O que é UUID?
- O que é JSON?



Voltando a API Rest ...

- Então... o padrão arquitetural REST é um estilo de arquitetura de software amplamente utilizado na concepção de sistemas distribuídos na web.
- Enfatiza princípios simples, como recursos identificados por URLs, interações baseadas em métodos HTTP
- A arquitetura REST promove a escalabilidade, a simplicidade, a independência de estado e a capacidade de cache, tornando-a uma escolha popular para o desenvolvimento de APIs e serviços web que sejam eficientes, escaláveis e facilmente comprehensíveis.

REST (Representational State Transfer)

- A arquitetura REST (Representational State Transfer) foi proposta por Roy Fielding em sua tese de doutorado de 2000.
- Fielding desenvolveu a arquitetura REST enquanto trabalhava na definição do protocolo HTTP 1.1 e em outros padrões da World Wide Web.
- A ideia por trás do REST é fornecer uma arquitetura de comunicação simples, escalável e orientada a recursos para sistemas distribuídos na web.
- Em vez de usar protocolos complexos e especializados, como o SOAP (Simple Object Access Protocol), REST utiliza os métodos e recursos básicos do protocolo HTTP para criar APIs e serviços web.

REST - Características chaves

- **Recursos (Resources):** Os recursos são as entidades que podem ser acessadas através de uma API REST. Eles representam informações específicas e são identificados por meio de URIs (Uniform Resource Identifier).
 - Por exemplo, um serviço de produtos pode ter recursos como "https://exemplo.com/produtos" e " https://exemplo.com /produtos/1" para representar todos os produtos e um produto específico, respectivamente.
- **Verbos HTTP (HTTP Verbs):** REST utiliza os verbos HTTP, como GET, POST, PUT e DELETE, para indicar as operações a serem realizadas nos recursos.
 - Por exemplo, o verbo GET é usado para recuperar um recurso, POST para criar um novo recurso, PUT para atualizar um recurso existente e DELETE para excluir um recurso.

REST - Características chaves

- **Representação dos Recursos (Resource Representation)**: Os recursos são representados em um formato específico, como JSON (JavaScript Object Notation), XML (eXtensible Markup Language) ou YAML (YAML Ain't Markup Language).
- **Sem Estado (Stateless)**: Cada solicitação para um recurso em uma API REST contém todas as informações necessárias para entender e processar a solicitação. O servidor não mantém informações de estado entre as solicitações. Isso permite que os serviços REST sejam altamente escaláveis e independentes de sessão.
- **Links (HATEOAS)**: O princípio HATEOAS (Hypermedia as the Engine of Application State) é uma característica do REST que fornece links ou referências navegáveis junto com as respostas.

NÃO é um protocolo, mas existem “regras”

- Quando falamos de comunicação via HTTP, as únicas regras que devemos seguir são as do próprio protocolo HTTP.
- No entanto, para evitar desorganização, existem várias convenções e boas práticas recomendadas, assim como acontece em programação de forma geral.
- Para APIs REST, Leonard Richardson propôs em 2008 um modelo de maturidade que ajuda a estruturar melhor o desenvolvimento dessas APIs.
- Esse modelo é conhecido como os Níveis de Maturidade de Richardson.

"The past is dust, and the future but smart nanodust."

Modelo de Maturidade de Richardson

- Richardson Maturity Model
- Leonard Richardson
 - <https://www.crummy.com/self/>
- Qcon 2008
 - <https://qconsf.com/sf2008/>
 - <https://www.crummy.com/writing/speaking/2008-QCon/act3.html>



Nível Zero (0) – POX (Plain Old XML)

- One URI, one HTTP method
- XML-RPC and most SOAP services

- Nesse nível, a API utiliza apenas o protocolo HTTP como transporte, mas não segue os princípios REST. A comunicação ocorre usando XML (ou outro formato) em mensagens POST, geralmente com uma única URL para todas as operações.

Requisição POST /appointmentService HTTP/1.1
[various other headers]

```
<openSlotRequest date = "2010-01-04" doctor = "mjones"/>
```

Resposta HTTP/1.1 200 OK
[various headers]

```
<openSlotList>
  <slot start = "1400" end = "1450">
    <doctor id = "mjones"/>
  </slot>
  <slot start = "1600" end = "1650">
    <doctor id = "mjones"/>
  </slot>
</openSlotList>
```

Requisição POST /appointmentService HTTP/1.1
[various other headers]

```
<appointmentRequest>
  <slot doctor = "mjones" start = "1400" end = "1450"/>
  <patient id = "jsmith"/>
</appointmentRequest>
```

Resposta HTTP/1.1 200 OK
[various headers]

```
<appointment>
  <slot doctor = "mjones" start = "1400" end = "1450"/>
  <patient id = "jsmith"/>
</appointment>
```

Requisição

POST /appointmentService HTTP/1.1

[various other headers]

```
<openSlotRequest date = "2010-01-04" doctor = "mjones"/>
```

Resposta

HTTP/1.1 200 OK

[various headers]

```
<openSlotList>
  <slot start = "1400" end = "1450">
    <doctor id = "mjones"/>
  </slot>
  <slot start = "1600" end = "1650">
    <doctor id = "mjones"/>
  </slot>
</openSlotList>
```

Requisição

POST /appointmentService HTTP/1.1

[various other headers]

```
<appointmentRequest>
  <slot doctor = "mjones" start = "1400" end = "1450"/>
  <patient id = "jsmith"/>
</appointmentRequest>
```

Resposta

HTTP/1.1 200 OK

[various headers]

```
<appointment>
  <slot doctor = "mjones" start = "1400" end = "1450"/>
  <patient id = "jsmith"/>
</appointment>
```

Nível Um (1) – Recursos Individuais



- Many URLs, one HTTP method
- Most "RESTful" services that aren't

- No Nível 1, a API começa a utilizar recursos individuais identificados por URLs únicas. Cada recurso é acessado por meio de uma URL específica, mas ainda há uma **dependência** em torno das operações POST, GET, PUT e DELETE para manipular esses recursos.

Requisição POST /doctors/mjones HTTP/1.1
[various other headers]

```
<openSlotRequest date = "2010-01-04"/>
```

Resposta HTTP/1.1 200 OK
[various headers]

```
<openSlotList>
  <slot id = "1234" doctor = "mjones" start = "1400" end = "1450"/>
  <slot id = "5678" doctor = "mjones" start = "1600" end = "1650"/>
</openSlotList>
```

Requisição POST /slots/1234 HTTP/1.1
[various other headers]

```
<appointmentRequest>
  <patient id = "jsmith"/>
</appointmentRequest>
```

Resposta HTTP/1.1 200 OK
[various headers]

```
<appointment>
  <slot id = "1234" doctor = "mjones" start = "1400" end = "1450"/>
  <patient id = "jsmith"/>
</appointment>
```

Requisição

POST /doctors/mjones HTTP/1.1
[various other headers]

<openSlotRequest date = "2010-01-04"/>

Resposta

HTTP/1.1 200 OK
[various headers]

```
<openSlotList>
    <slot id = "1234" doctor = "mjones" start = "1400" end = "1450"/>
    <slot id = "5678" doctor = "mjones" start = "1600" end = "1650"/>
</openSlotList>
```

Requisição

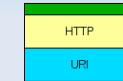
POST /slots/1234 HTTP/1.1
[various other headers]

```
<appointmentRequest>
    <patient id = "jsmith"/>
</appointmentRequest>
```

Resposta

HTTP/1.1 200 OK
[various headers]

```
<appointment>
    <slot id = "1234" doctor = "mjones" start = "1400" end = "1450"/>
    <patient id = "jsmith"/>
</appointment>
```



- Many URIs, each supporting multiple HTTP methods
- The best example is Amazon S3

Nível Dois (2) – Verbos HTTP

- No Nível 2, a API adere aos verbos HTTP corretos para realizar operações nos recursos. Em vez de depender de uma única URL e de operações genéricas, os verbos HTTP (GET, POST, PUT, DELETE) são utilizados de forma adequada para realizar as operações apropriadas nos recursos.

Requisição

```
GET /doctors/mjones/slots?date=20100104&status=open HTTP/1.1
Host: royalhope.nhs.uk
```

Resposta

```
HTTP/1.1 200 OK
[various headers]
```

```
<openSlotList>
  <slot id = "1234" doctor = "mjones" start = "1400" end = "1450"/>
  <slot id = "5678" doctor = "mjones" start = "1600" end = "1650"/>
</openSlotList>
```

Requisição

```
POST /slots/1234 HTTP/1.1
[various other headers]
```

```
<appointmentRequest>
  <patient id = "jsmith"/>
</appointmentRequest>
```

Resposta

```
HTTP/1.1 201 Created
Location: slots/1234/appointment
[various headers]
```

```
<appointment>
  <slot id = "1234" doctor = "mjones" start = "1400" end = "1450"/>
  <patient id = "jsmith"/>
</appointment>
```

Requisição

GET /doctors/mjones/slots?date=20100104&status=open HTTP/1.1

Host: royalhope.nhs.uk

Resposta

HTTP/1.1 200 OK

[various headers]

```
<openSlotList>
    <slot id = "1234" doctor = "mjones" start = "1400" end = "1450"/>
    <slot id = "5678" doctor = "mjones" start = "1600" end = "1650"/>
</openSlotList>
```

Requisição

POST /slots/1234 HTTP/1.1

[various other headers]

```
<appointmentRequest>
    <patient id = "jsmith"/>
</appointmentRequest>
```

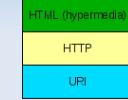
Resposta

HTTP/1.1 201 Created

Location: slots/1234/appointment

[various headers]

```
<appointment>
    <slot id = "1234" doctor = "mjones" start = "1400" end = "1450"/>
    <patient id = "jsmith"/>
</appointment>
```



- Resources describe their own capabilities and interconnections
- WWW, AtomPub, Netflix, Launchpad...

Nível Três (3) – Hipermídia (HATEOAS)

A web que usamos usa apenas GET e POST, porque esses são os únicos métodos suportados pelo HTML 4. Mas a maioria dos RESTful vai além e separa PUT e DELETE de POST. Acho que você provavelmente está familiarizado com essa controvérsia.

Existem debates sobre o valor desses métodos, mas este é um debate sobre o nível dois da heurística da maturidade, não um debate sobre quem é mais puro ou mais prático. O argumento para esses métodos, ou para quaisquer métodos, é que, se os separarmos do POST, eles começarão a significar algo além de "tanto faz!" e podemos otimizar em torno deles.

A desvantagem é que, ao adicionar métodos HTTP, você limita o universo de clientes que podem entender a semântica do seu serviço. A partir de certo ponto é melhor descrever as especificidades de uma operação com hipermídia. O que nos leva a...

Nível três: hipermídia

Requisição

GET /doctors/mjones/slots?date=20100104&status=open HTTP/1.1
Host: royalhope.nhs.uk

Resposta

HTTP/1.1 200 OK

[various headers]

```
<openSlotList>
  <slot id = "1234" doctor = "mjones" start = "1400" end = "1450">
    <link rel = "/linkrels/slot/book"
      uri = "/slots/1234"/>
  </slot>
  <slot id = "5678" doctor = "mjones" start = "1600" end = "1650">
    <link rel = "/linkrels/slot/book"
      uri = "/slots/5678"/>
  </slot>
</openSlotList>
```

Requisição

POST /slots/1234 HTTP/1.1

[various other headers]

```
<appointmentRequest>
  <patient id = "jsmith"/>
</appointmentRequest>
```

Resposta

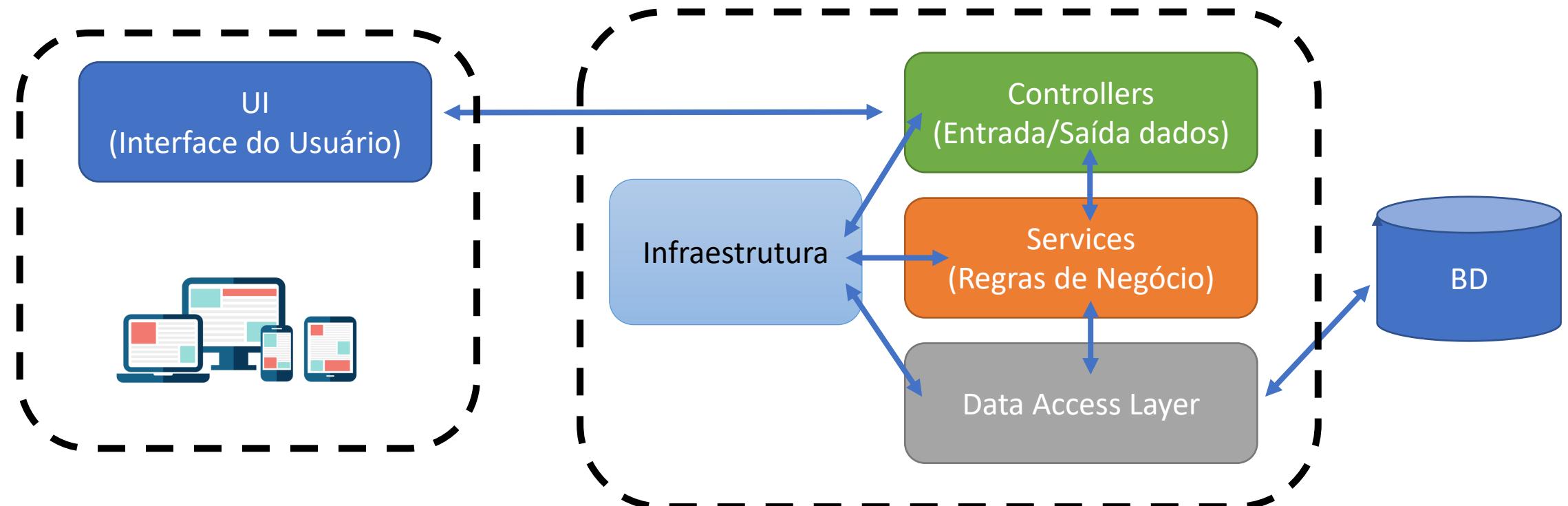
```
<appointment>
  <slot id = "1234" doctor = "mjones" start = "1400" end = "1450"/>
  <patient id = "jsmith"/>
  <link rel = "/linkrels/appointment/cancel"
        uri = "/slots/1234/appointment"/>
  <link rel = "/linkrels/appointment/addTest"
        uri = "/slots/1234/appointment/tests"/>
  <link rel = "self"
        uri = "/slots/1234/appointment"/>
  <link rel = "/linkrels/appointment/changeTime"
        uri = "/doctors/mjones/slots?date=20100104&status=open"/>
  <link rel = "/linkrels/appointment/updateContactInfo"
        uri = "/patients/jsmith/contactInfo"/>
  <link rel = "/linkrels/help"
        uri = "/help/appointment"/>
</appointment>
```

Conclusão - REST

- A abordagem REST é amplamente utilizada no desenvolvimento de APIs para serviços web, pois é simples, escalável e amplamente suportada por diferentes tecnologias. Ela enfatiza a arquitetura orientada a recursos e a utilização dos verbos e status HTTP para realizar operações nos recursos. APIs REST são conhecidas por serem mais leves e menos complexas do que as APIs baseadas em SOAP
- É importante ressaltar que REST é um estilo arquitetural e **não** uma especificação ou protocolo específico..

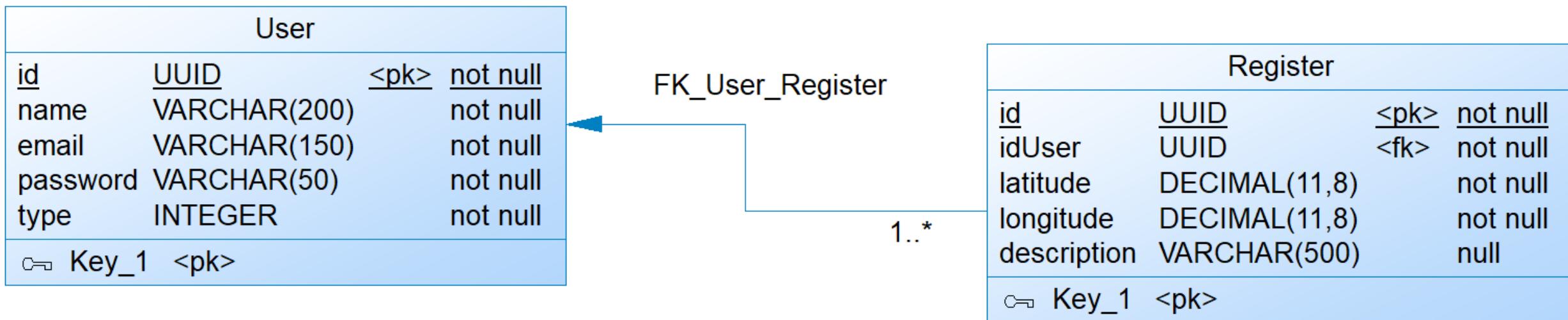
E o nosso projeto??? Como vai ser???

- Arquitetura Cliente-Servidor em conjunto com Arquitetura de Camadas
 - Somente o lado do Servidor (Backend)
 - Resumindo será um API



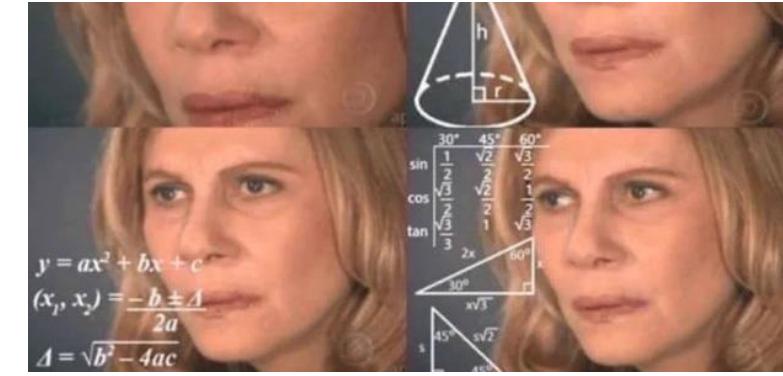
Desenvolvimento de soluções (Full Stack)

Objetivo do Semestre → Projeto APISample



Mas para isso precisamos entender algumas coisas...

- ✓ • Linguagem Java (sintaxe e funcionamento)
- ✓ • Paradigma Orientado a Objetos (conceitos desenvolvidos em Java)
- ✓ • O que é um Backend? (Arquitetura de Sistemas)
- ✓ • O que é uma API?
- ✓ • O que é API Rest? (Padrão arquitetural RestFull)
 - O que é e como funciona o HTTP?
- ✓ • O que é Spring Framework?
- O que é UUID?
- O que é JSON?



UUID - Universally Unique
Identifier

UUID - Universally Unique Identifier

- **UUID: "Identificador Único Universal"**
- Tipo de identificador único que é gerado de forma aleatória e única, de modo que a probabilidade de dois UUIDs serem iguais é extremamente baixa.
- UUIDs são comumente usados em sistemas de software para identificar de forma única recursos, objetos ou entidades.

UUID - Universally Unique Identifier

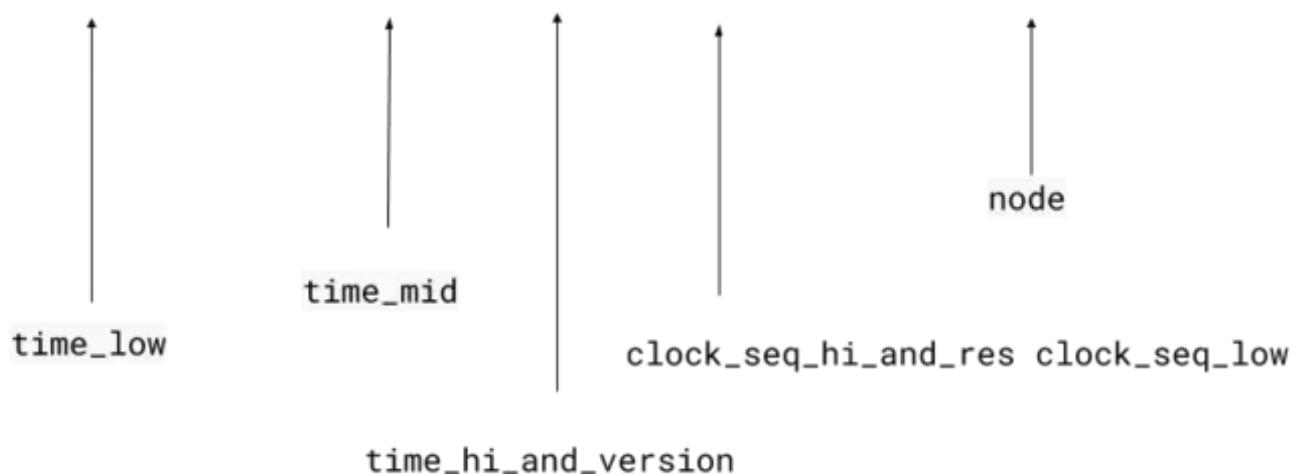
- **Unicidade Universal:** Devem ser únicos **não** apenas em um sistema específico, mas em **todo o mundo**. Isso é alcançado usando um espaço de identificadores tão grande que a probabilidade de colisão é considerada insignificante.
- **Aleatoriedade:** Geralmente gerados de forma aleatória, o que significa que não há um padrão previsível para sua criação.
 - Existem diferentes versões de UUIDs que usam diferentes métodos de geração, incluindo timestamps e informações de hardware.
- **Formato:** Sequência de 128 bits (16 bytes) geralmente exibida em formato hexadecimal com hífens, como "550e8400-e29b-41d4-a716-446655440000".
 - A estrutura de um UUID inclui diferentes campos, como um timestamp, um identificador de versão e informações de clock.

UUID - Universally Unique Identifier

- **Versões de UUID:** Existem várias versões de UUIDs, cada uma com um propósito específico. As versões mais comuns são UUIDv1 e UUIDv4.
 - UUIDv1 é gerado com base no tempo e no endereço MAC do computador
 - UUIDv4 é gerado de forma completamente aleatória.
- **Aplicações:** Amplamente usados em sistemas distribuídos, bancos de dados, identificação de dispositivos, gerenciamento de sessões em aplicativos da web e em muitos outros cenários onde a garantia de unicidade é fundamental.
- **Colisões:** Embora seja altamente improvável que dois UUIDs gerados aleatoriamente colidam, a probabilidade não é zero.
- **Representação:** UUIDs são frequentemente representados como strings para serem facilmente armazenados e transmitidos entre sistemas.

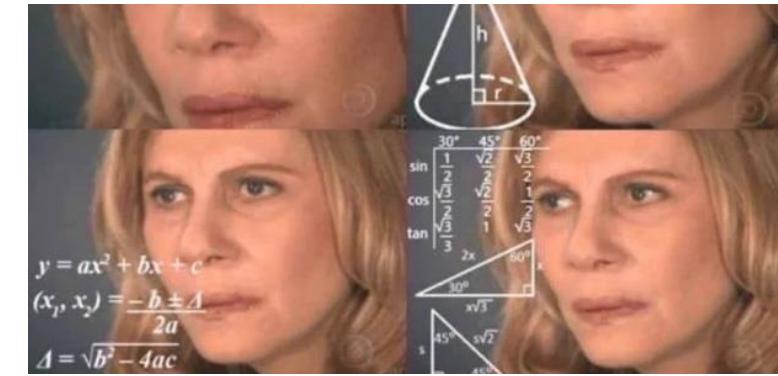


df6fdea1-10c3-474c-ae62-e63def80de0b



Mas para isso precisamos entender algumas coisas...

- ✓ • Linguagem Java (sintaxe e funcionamento)
- ✓ • Paradigma Orientado a Objetos (conceitos desenvolvidos em Java)
- ✓ • O que é um Backend? (Arquitetura de Sistemas)
- ✓ • O que é uma API?
- ✓ • O que é API Rest? (Padrão arquitetural RestFull)
 - O que é e como funciona o HTTP?
- ✓ • O que é Spring Framework?
- ✓ • O que é UUID?
- ✓ • O que é JSON?

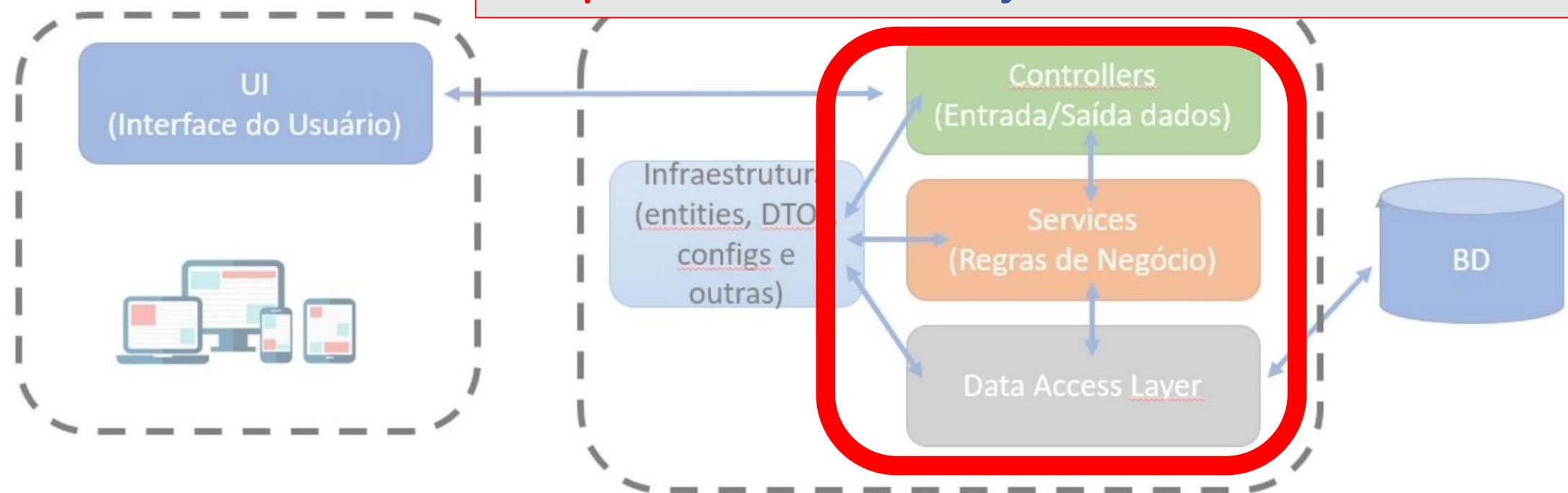


E o nosso projeto??? Como vai ser???

- Arquitetura Cliente-Serviço
 - Somente o lado do Serviço
 - Seguindo os princípios KISS

Como será essa comunicação interna?

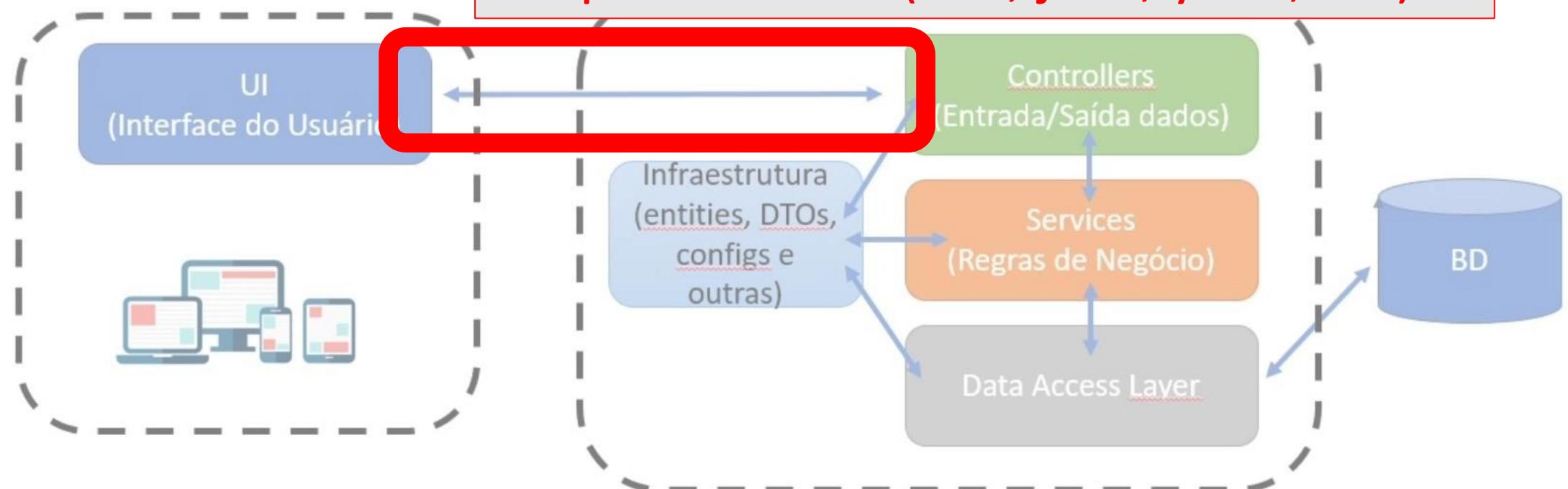
- **Como?** Através da invocação de métodos
- **Tipo de dados?** Objetos Java



E o nosso projeto??? Como vai ser???

- Arquitetura Cliente-Serviço (VC)
 - Somente o lado do Serviço
 - Seguindo os princípios KISS

Como será essa comunicação?
- Protocolo (soap, http, smtp, etc)?
- Tipo de dados (xml, json, yaml, etc)?



```
{  
  Routers: [
```

```
    {  
      hostname: Backbone-1,  
      model: CRS-X,  
      role: Core  
    }  
  ]
```

JSON

Routers:

```
  hostname: Backbone-1  
  model: CRS-X  
  role: Core
```

YAML

XML

```
<Routers>
```

```
  <Router>
```

```
    <hostname>Backbone-1</hostname>  
    <model>CRS-X</model>  
    <role>Core</role>
```

```
  </Router>
```

```
</Routers>
```

Por que JSON???

Quando usar o YAML versus JSON

Graças ao suporte abrangente e à integração com JavaScript, o JSON é um formato de serialização de dados mais popular para a maioria dos casos de uso do que o YAML. O JSON é usado extensivamente em comunicações de software distribuídas, aplicações Web, arquivos de configuração e APIs.

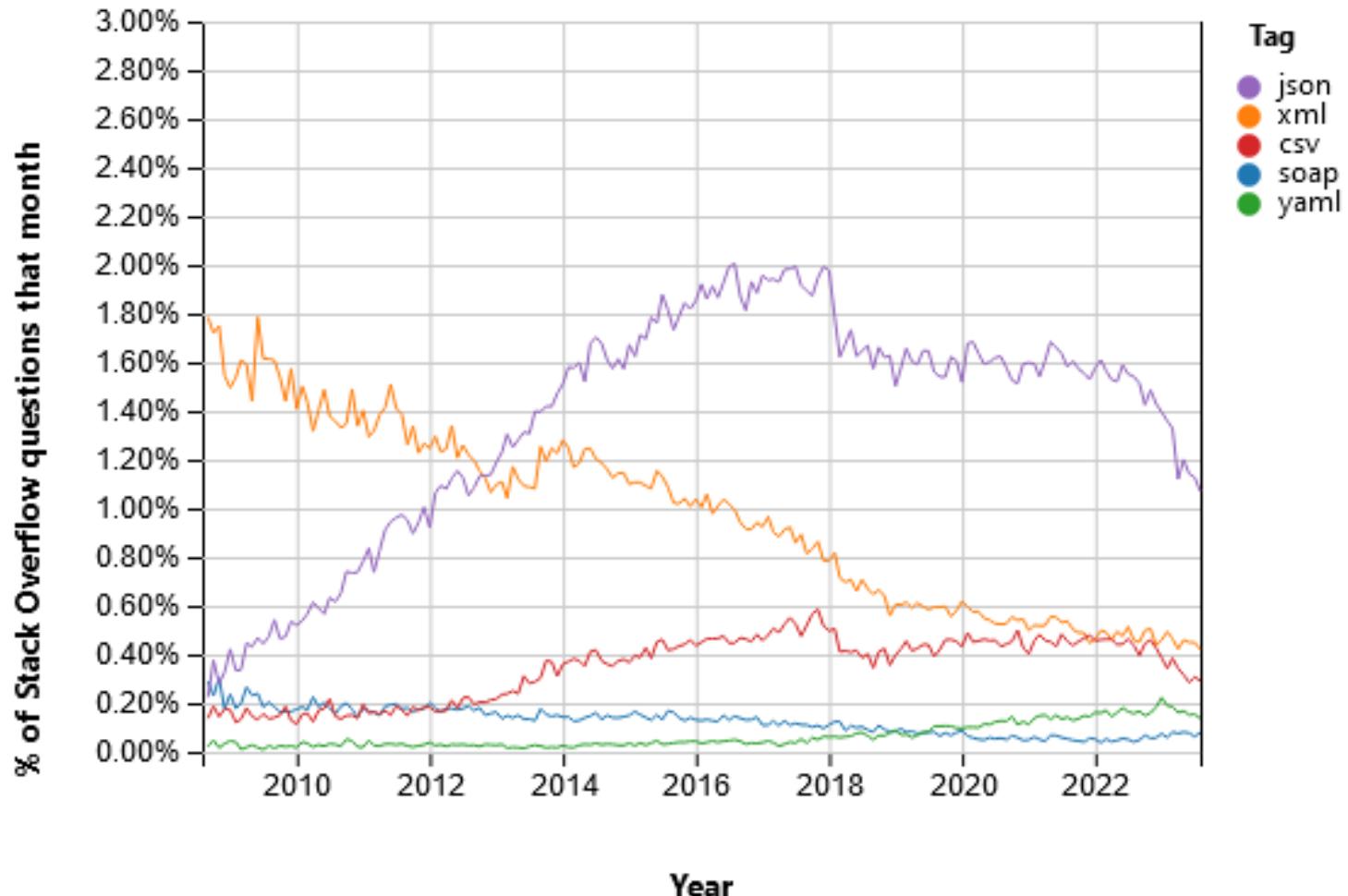


Resumo das diferenças: YAML versus JSON

	JSON	YAML
O que é isso?	Um formato de serialização de dados para trocar dados estruturados entre aplicações e serviços de software. Prioriza o uso de aplicações em relação ao uso humano.	Um formato de serialização de dados para trocar dados estruturados entre aplicações e serviços de software. Prioriza o uso humano em relação ao uso de aplicações.
Principais casos de uso	Difundido em plataformas, linguagens, comunicações de software distribuídas, aplicações Web, arquivos de configuração e APIs.	Arquivos de configuração em várias ferramentas e serviços de automação, DevOps e infraestrutura como código (IaC).
Legibilidade	Fácil.	A mais fácil.
Tipos de dados	Número, booleano, nulo, string, matriz e objeto.	Oferece suporte a todos os tipos de dados por meio da coleta de dados aninhados que incluem sequências, escalares e mapeamentos.
Suporta comentários	Não.	Sim.
Suporta objetos de dados como valores	Sim.	Não.

<https://aws.amazon.com/pt/compare/the-difference-between-yaml-and-json/>

Por que JSON???



<https://insights.stackoverflow.com/trends?tags=soap%2Cxml%2Cyaml%2Ccsv%2Cjson>

JSON - JavaScript Object Notation

JSON - JavaScript Object Notation

- **JSON** - Notação de Objeto JavaScript
- Formato de dados leve e fácil de ler que é amplamente utilizado para trocar informações entre sistemas, especialmente na web.
- Muito popular devido à sua simplicidade e facilidade de uso.
- O JSON foi “criado” como um formato de intercâmbio de dados por Douglas Crockford.

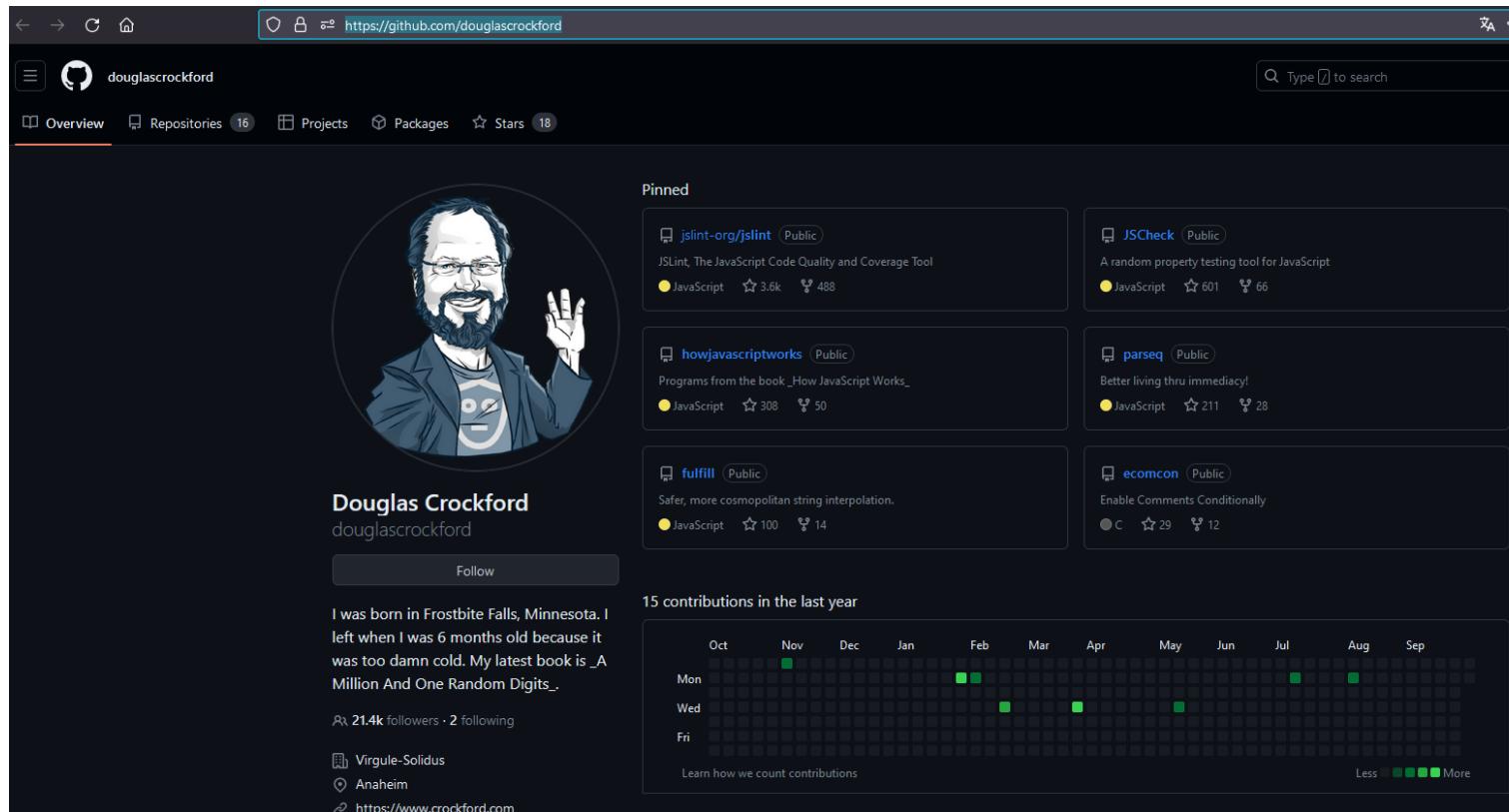
JSON – Resumo da história e evolução

- **Década de 1990:** Influenciado pelo padrão de notação de objeto literário JavaScript.
- **2001:** Douglas Crockford formalizou o formato JSON → "The JSON Saga".
- **RFC 4627 em 2006:** Padronizado pela IETF (Internet Engineering Task Force) no RFC 4627.
- **Amplamente Adotado:** A simplicidade do JSON, juntamente com a crescente popularidade da web e das tecnologias da web, o tornou uma escolha popular para trocar dados entre aplicativos, especialmente em serviços da web, APIs REST e AJAX.
- **Suporte em Linguagens de Programação:** A maioria das linguagens de programação modernas oferece suporte nativo ou bibliotecas para trabalhar com JSON.
- **JSON-LD e Outros Formatos:** Outras variantes do JSON surgiram para atender a necessidades específicas. Por exemplo, o JSON-LD (JSON Linked Data) é uma extensão que permite a representação de dados semânticos para a Web Semântica.
- **Aplicações Além da Web:** Além de ser amplamente utilizado na web, o JSON também é utilizado em muitos outros contextos, como armazenamento de configurações, troca de dados em sistemas embutidos, armazenamento de registros de log, entre outros.

Douglas Crockford



- <https://www.crockford.com/>
- <https://github.com/douglascrockford>
- <https://www.json.org/>



A screenshot of Douglas Crockford's GitHub profile page. The profile features a caricature of him with a beard and glasses. The GitHub interface shows pinned repositories: `jshint-org/jshint`, `JSCheck`, `howjavascriptworks`, `parseq`, `fulfill`, and `ecomcon`. Below the pinned repos, there's a timeline showing 15 contributions in the last year, with activity concentrated in November, February, and August. The bio on the page includes a personal anecdote about his birthplace and a mention of his book `_A Million And One Random Digits_`.

Follow

I was born in Frostbite Falls, Minnesota. I left when I was 6 months old because it was too damn cold. My latest book is `_A Million And One Random Digits_`.

21.4k followers • 2 following

Virgule-Solidus
Anaheim
<https://www.crockford.com>

Oct Nov Dec Jan Feb Mar Apr May Jun Jul Aug Sep

Mon Wed Fri

Less More

I Discovered JSON

- I do not claim to have invented JSON.
It already existed in nature.
- I do not claim to have been the first to discover it.
- I gave it a specification and a little website.
- The rest happened by itself.

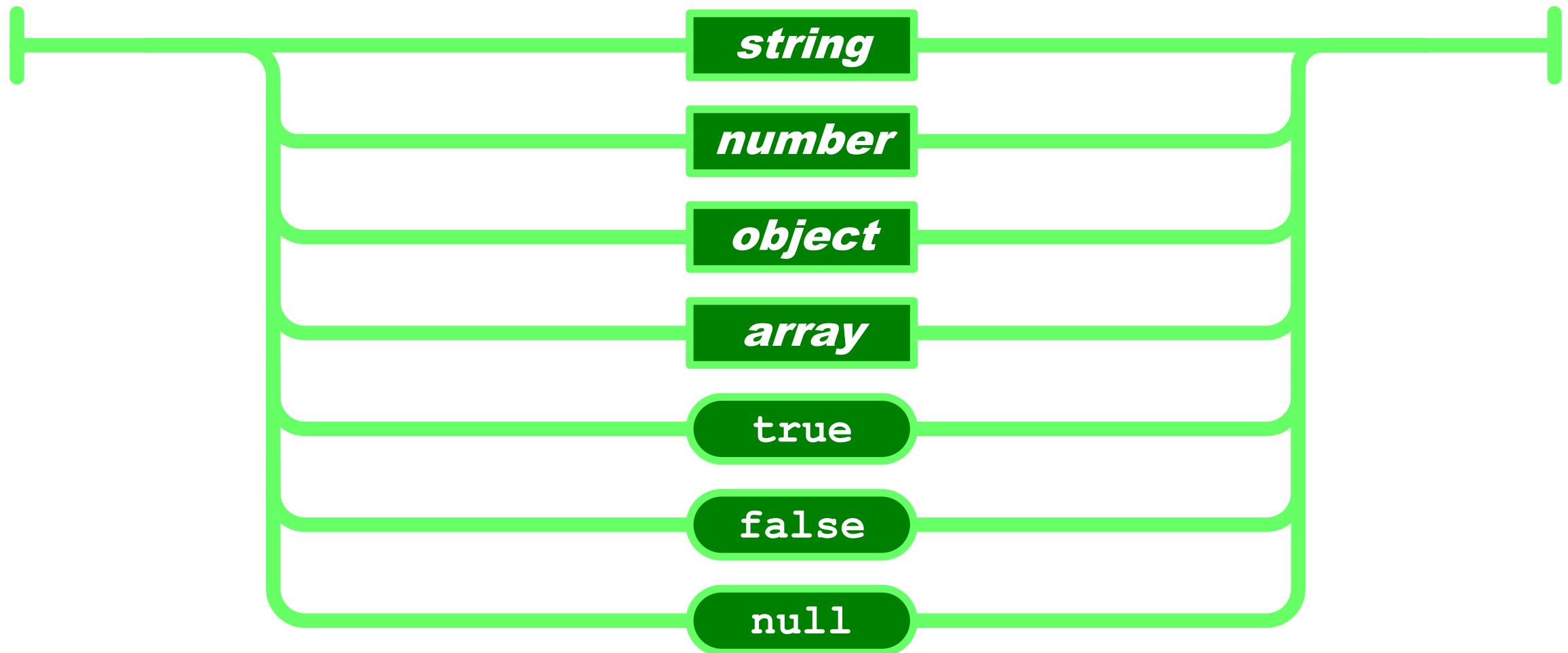
Sintaxe JSON

- Coleção de pares chave-valor.
- Os dados são armazenados em objetos JSON, que são delimitados por chaves {}.
- Cada par chave-valor é separado por vírgula e as **chaves** são strings que devem estar entre aspas duplas.

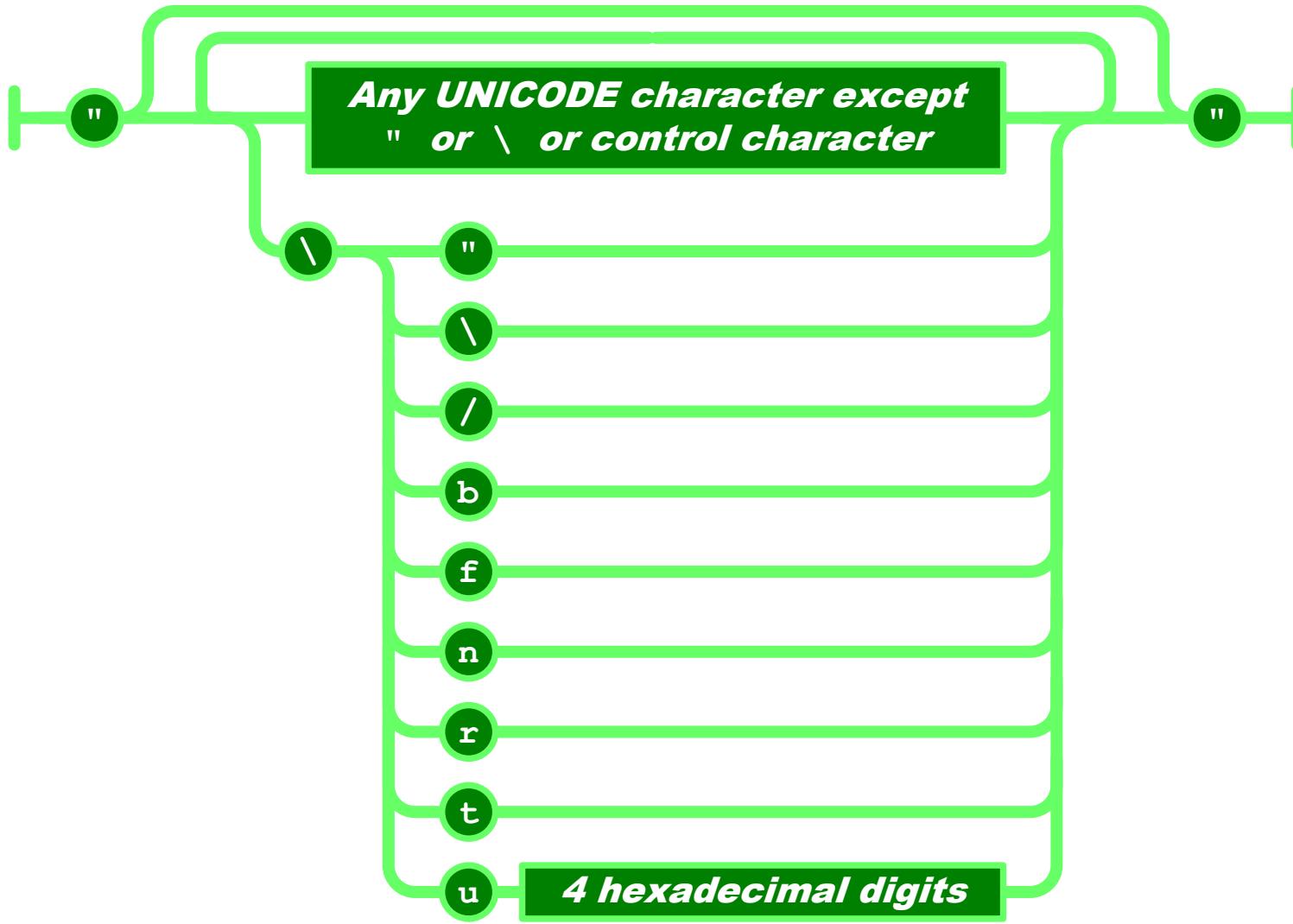
Exemplo Objeto JSON

```
1 v {
2   "name": "Luke Skywalker",
3   "height": "172",
4   "mass": "77",
5   "hair_color": "blond",
6   "skin_color": "fair",
7   "eye_color": "blue",
8   "birth_year": "19BBY",
9   "gender": "male",
10  "homeworld": "https://swapi.dev/api/planets/1/",
11  "films": [
12    "https://swapi.dev/api/films/2/",
13    "https://swapi.dev/api/films/6/",
14    "https://swapi.dev/api/films/3/",
15    "https://swapi.dev/api/films/1/",
16    "https://swapi.dev/api/films/7/"
17  ],
18  "species": [
19    "https://swapi.dev/api/species/1/"
20  ],
21  "vehicles": [
22    "https://swapi.dev/api/vehicles/14/",
23    "https://swapi.dev/api/vehicles/30/"
24  ],
25  "starships": [
26    "https://swapi.dev/api/starships/12/",
27    "https://swapi.dev/api/starships/22/"
28  ],
29  "created": "2014-12-09T13:50:51.644000Z",
30  "edited": "2014-12-20T21:17:56.891000Z",
31  "url": "https://swapi.dev/api/people/1/"
32 }
```

Value



String

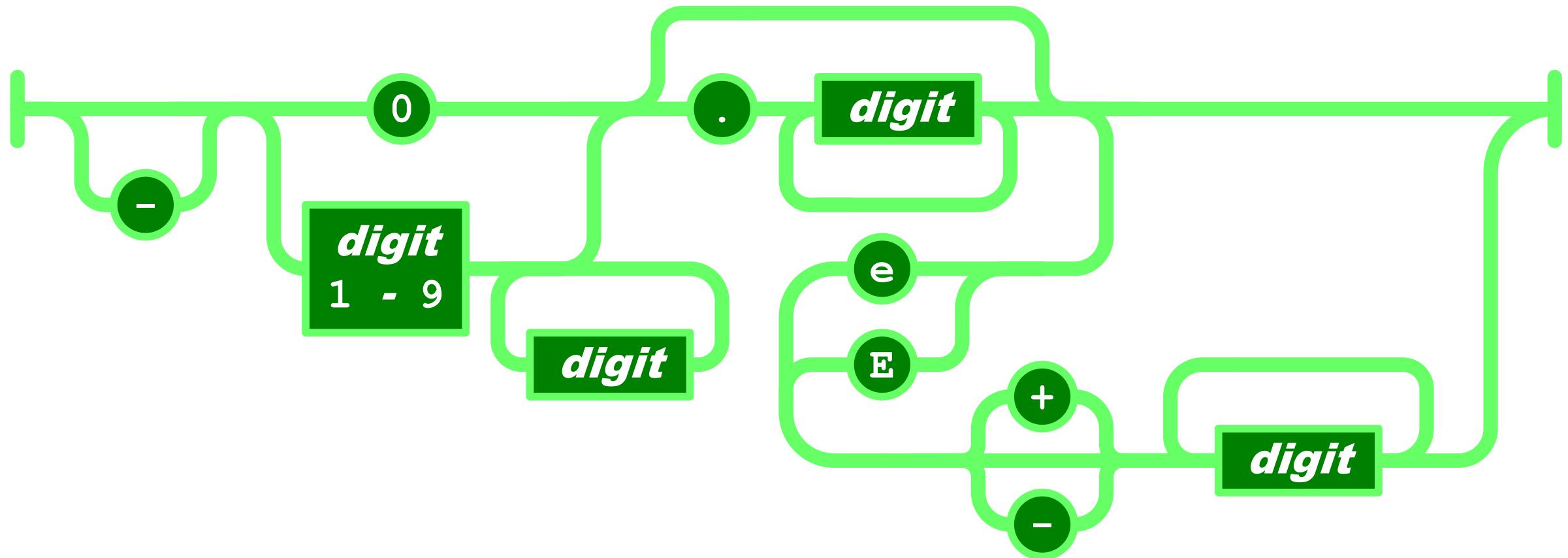


Tipos de Dados: String

- Uma sequência de caracteres entre aspas duplas.

```
"name": "Luke Skywalker",
"height": "172",
"mass": "77",
"hair_color": "blond",
"skin_color": "fair",
"eye_color": "blue",
"birth_year": "19BBY",
```

Number

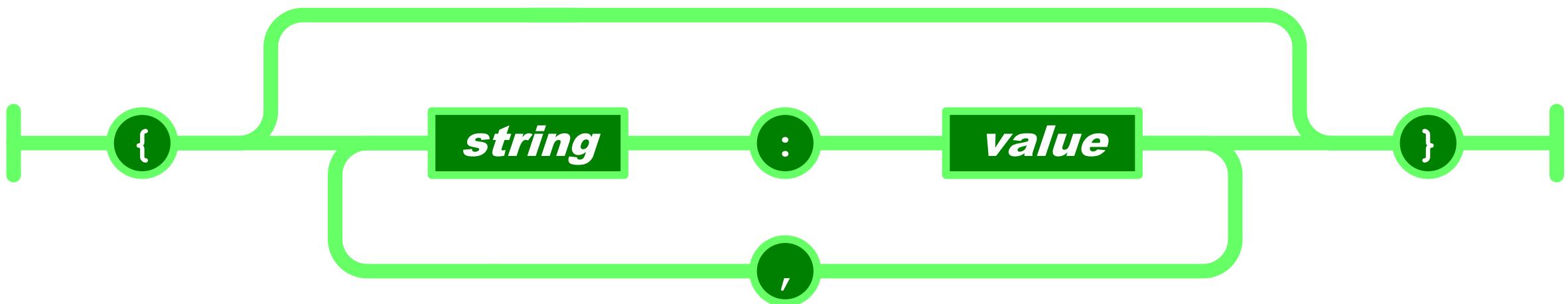


Tipos de Dados: Número

- Um número inteiro ou de ponto flutuante.

```
"height": 1.72,  
"mass": 77,
```

Object

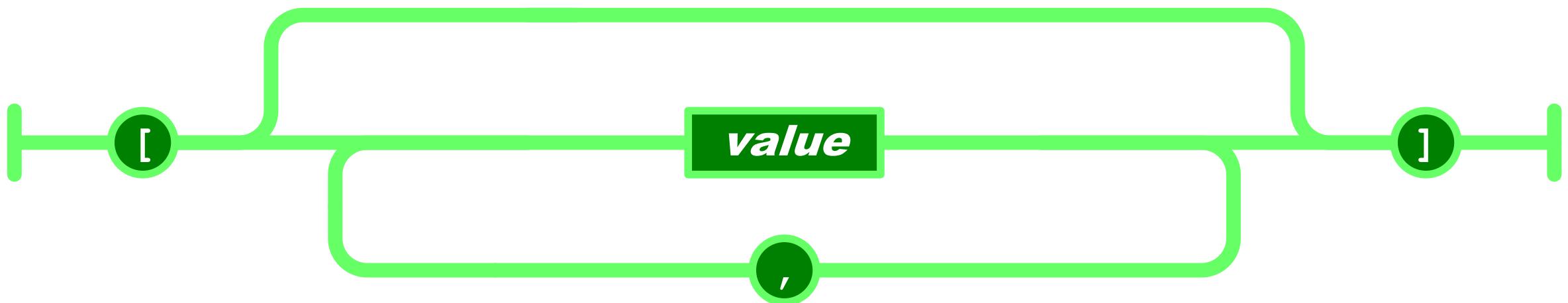


Tipos de Dados: Objeto

- Um conjunto não ordenado de pares chave-valor, separados por vírgulas e delimitados por chaves.

```
1 v {  
2   "name": "Han Solo",  
3   "height": "180",  
4   "mass": "80",  
5   "hair_color": "brown",  
6   "skin_color": "fair",  
7   "eye_color": "brown",  
8   "birth_year": "29BBY",  
9   "gender": "male",  
10  "homeworld": "https://swapi.dev/api/planets/22/",  
11  "starship_main": {  
12    "name": "Millennium Falcon",  
13    "model": "YT-1300 light freighter",  
14    "manufacturer": "Corellian Engineering Corporation",  
15    "cost_in_credits": "100000",  
16    "length": "34.37",  
17    "max_atmosphering_speed": "1050"  
18  }  
19 }
```

Array



Tipos de Dados: Vetores

- Uma coleção ordenada de valores, separados por vírgulas e delimitados por colchetes

```
1 v {
2   "name": "Han Solo",
3   "height": "180",
4   "mass": "80",
5   "starships": [
6     {
7       "name": "Millennium Falcon",
8       "model": "YT-1300 light freighter",
9       "manufacturer": "Corellian Engineering Corporation",
10      "cost_in_credits": "100000",
11      "length": "34.37",
12      "max_atmosphering_speed": "1050"
13    },
14    {
15      "name": "Imperial shuttle",
16      "model": "Lambda-class T-4a shuttle",
17      "manufacturer": "Sienar Fleet Systems",
18      "cost_in_credits": "240000",
19      "length": "20",
20      "max_atmosphering_speed": "850"
21    }
22  ]
23 }
```

Tipos de Dados: Boolean

- Pode ser **true** ou **false**
- Sem aspas ("")

```
1 v {  
2   "name": "Luke Skywalker",  
3   "height": "172",  
4   "mass": "77",  
5   "single": true  
6 }
```

Tipos de Dados: Nulo

- Representado pela palavra-chave **null**
- Sem aspas (“”)
- Usado para indicar que um valor está ausente ou indefinido.

```
1 v {  
2   "name": "C-3PO",  
3   "height": "167",  
4   "mass": "75",  
5   "hair_color": null,  
6   "skin_color": "gold",  
7   "eye_color": "yellow",  
8   "birth_year": "112BBY",  
9   "gender": null  
10 }
```