

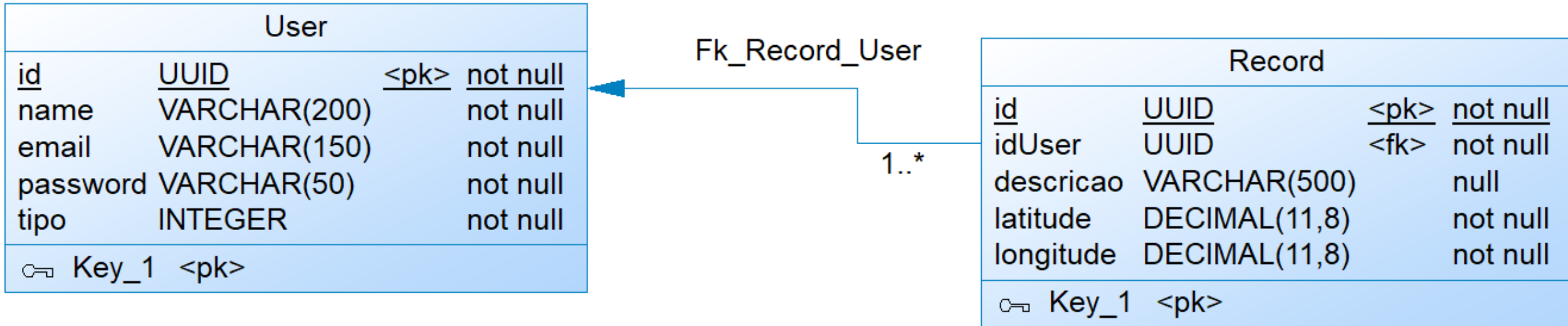


# Ciência da **Computação**

Programação Orientada a Objetos Avançado  
Prof. Luciano Rodrigo Ferretto

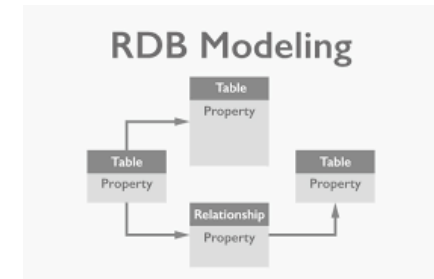
# Desenvolvimento de soluções (Full Stack)

## 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 UUID?
- O que é JSON?
- O que é Spring Framework?

# 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)

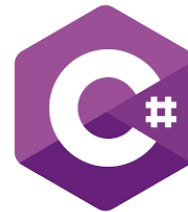
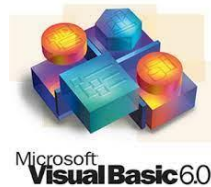
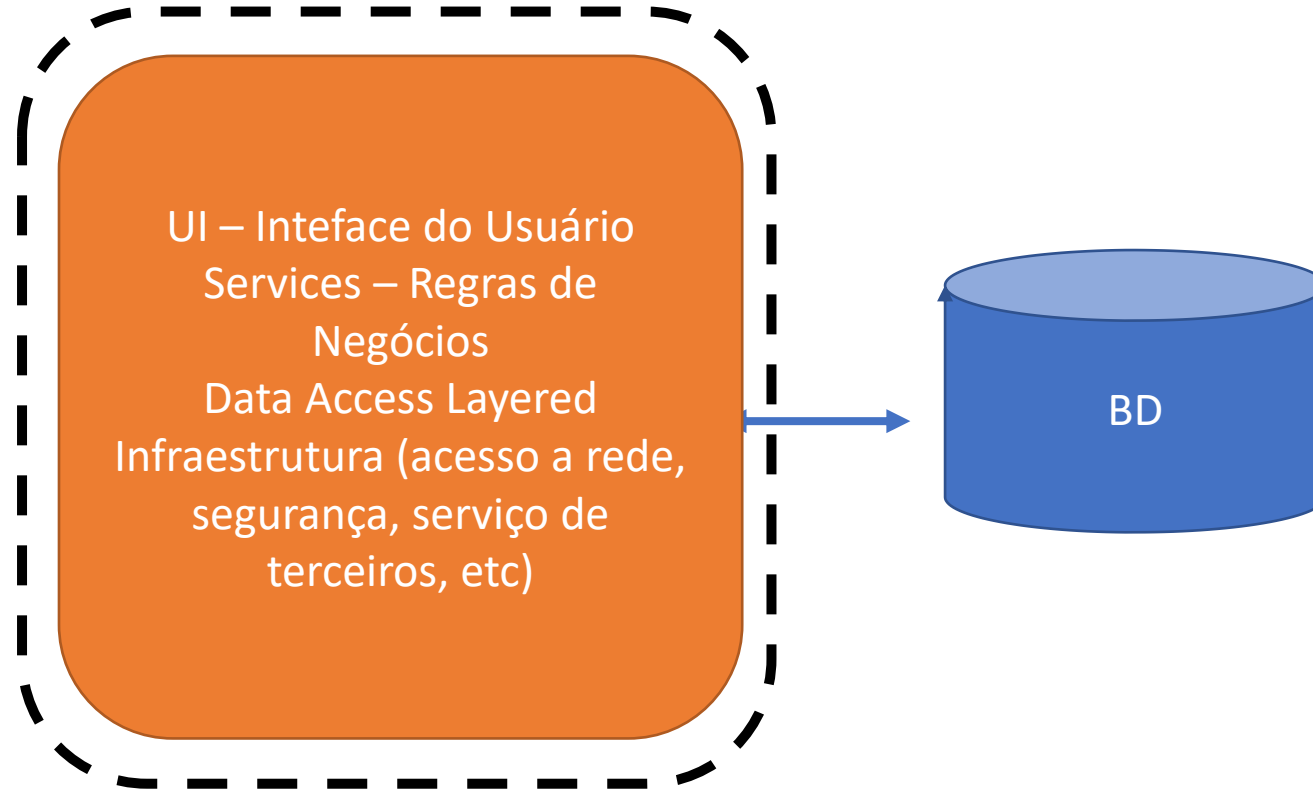
# Arquitetura Monolítica

# 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

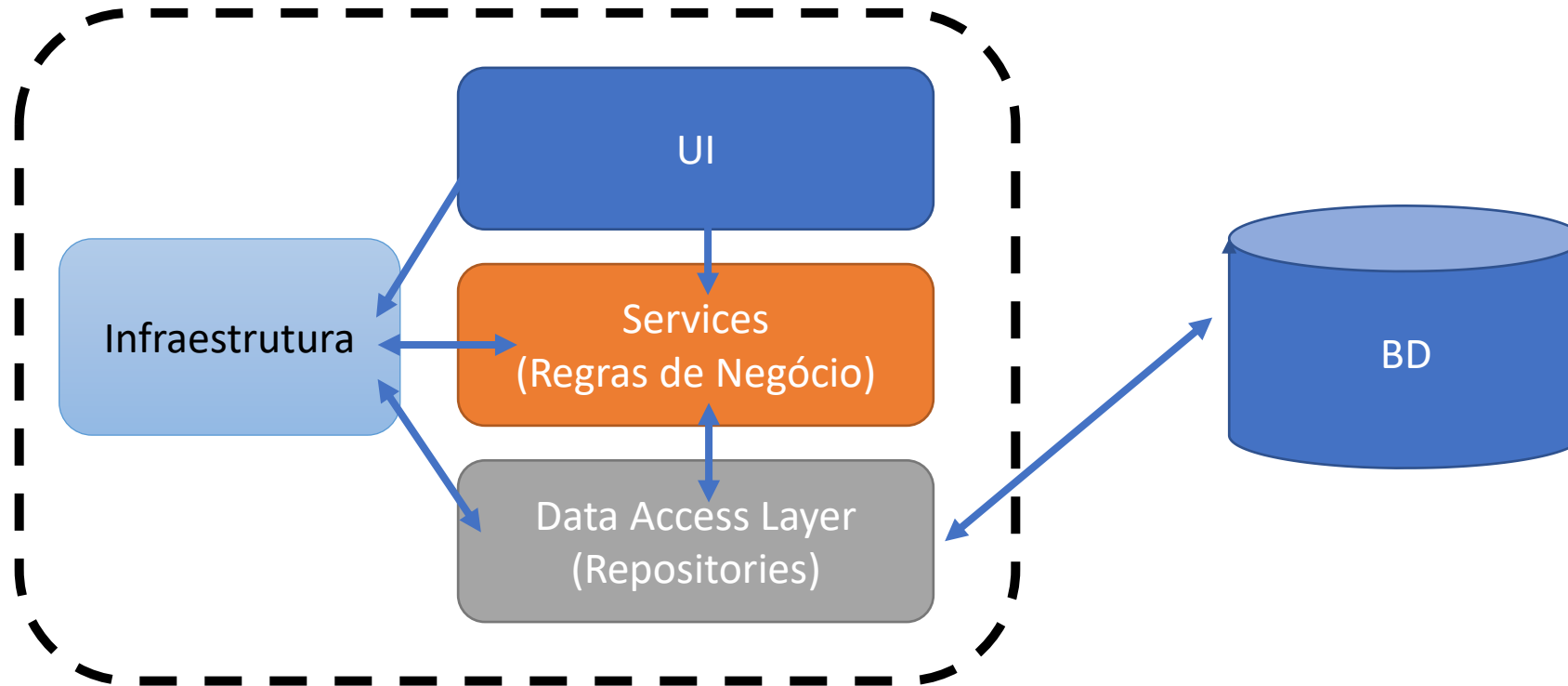
- **Escalabilidade Vertical:** A escalabilidade em uma arquitetura monolítica geralmente é feita aumentando os recursos (como CPU e RAM) da máquina que hospeda o sistema, o que é conhecido como escalabilidade vertical.
- **Limitações de Escalabilidade e Complexidade:** Conforme o sistema cresce e se 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 os as boas práticas e princípios de desenvolvimento, como o SOLID.



# Arquitetura de Camadas - Desvantagens

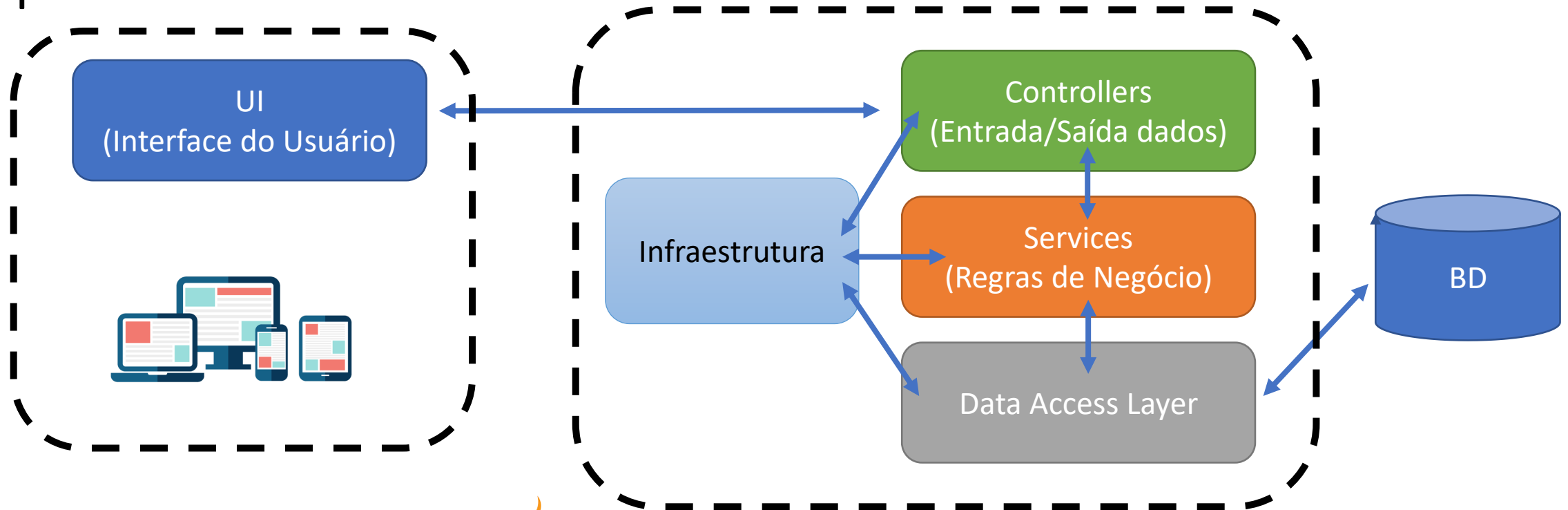
- **Overhead de Comunicação:** A comunicação entre camadas pode adicionar overhead, especialmente em sistemas distribuídos.
- **Complexidade de Design:** O design de sistemas em camadas pode se tornar complexo à medida que o sistema cresce, exigindo uma atenção cuidadosa à definição de interfaces e à 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



# 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

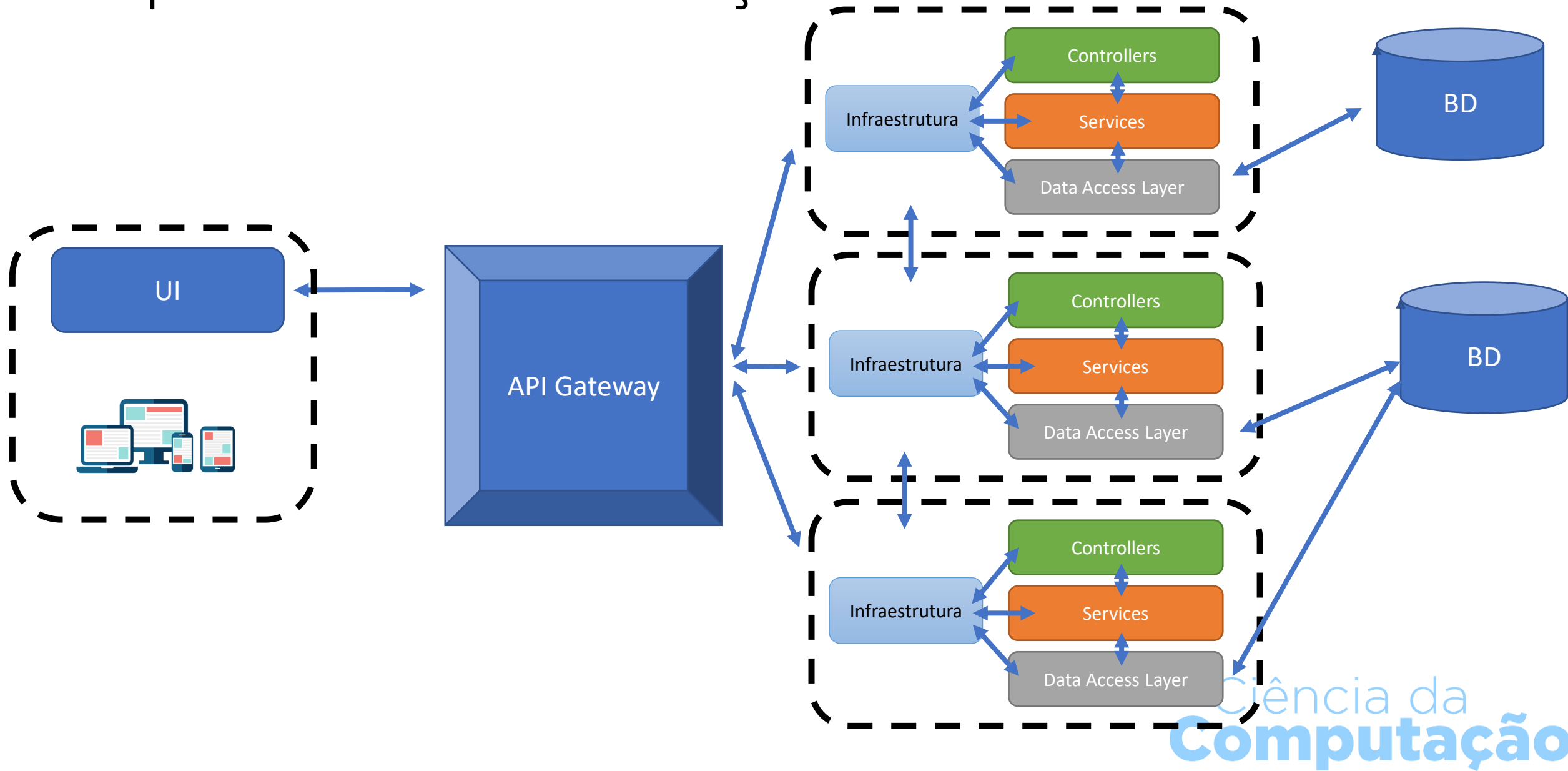
## MicroServices



# 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

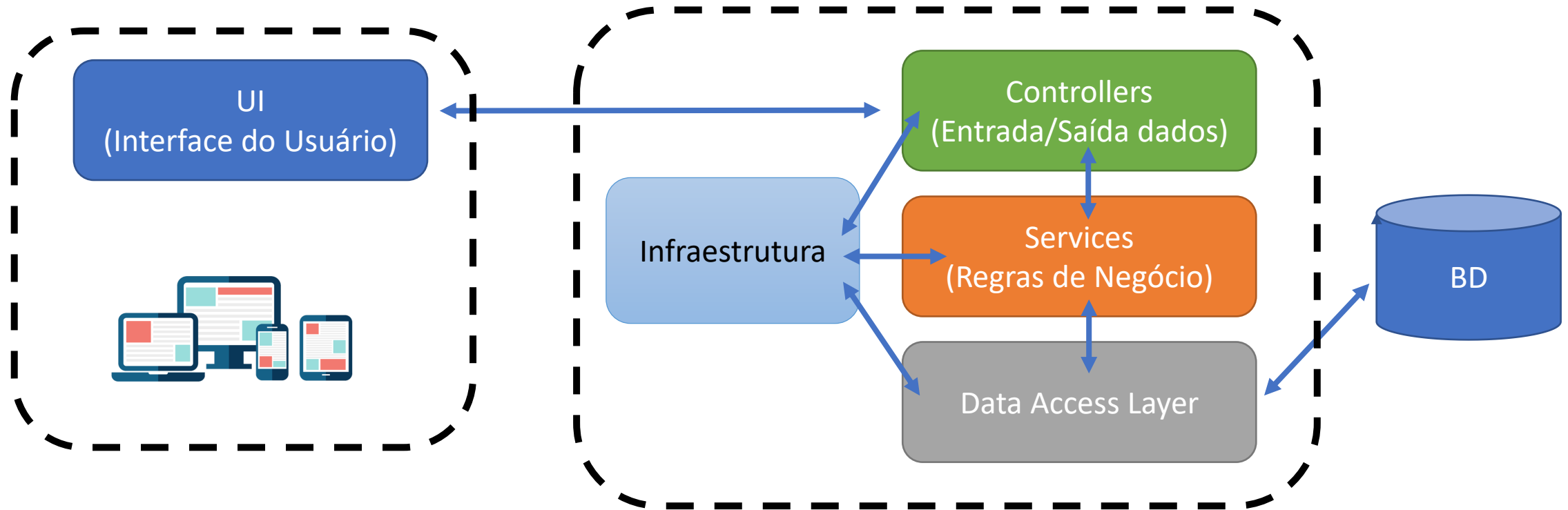
- **Escalabilidade Individual:** Cada microserviço pode ser escalado individualmente, o que significa que recursos adicionais podem ser alocados apenas para os serviços que precisam lidar com mais demanda.
- **Desenvolvimento Independente:** Equipes de desenvolvimento podem trabalhar em diferentes microserviços 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**



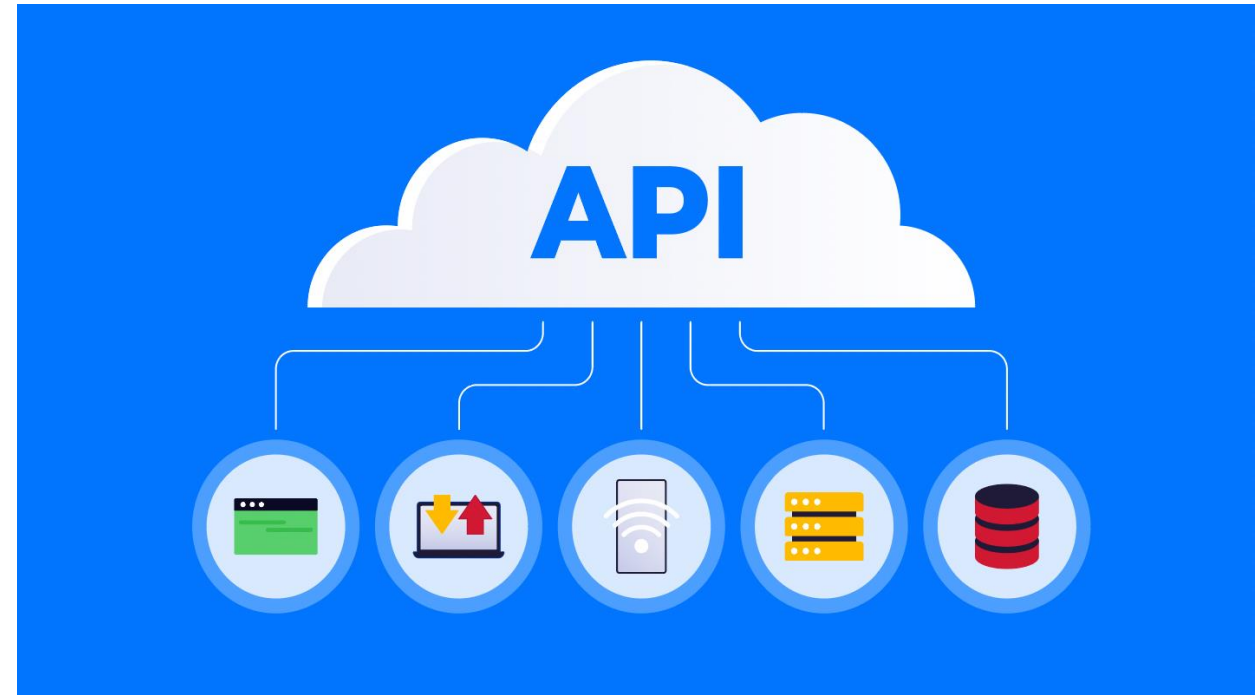
# Mas para isso precisamos entender algumas coisas...



- ✓ • 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 UUID?
- O que é JSON?
- O que é Spring Framework?

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.

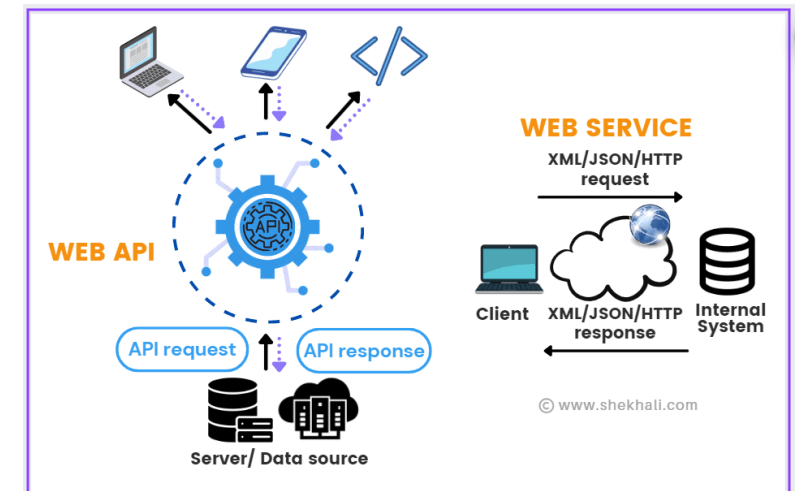
# API

- 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

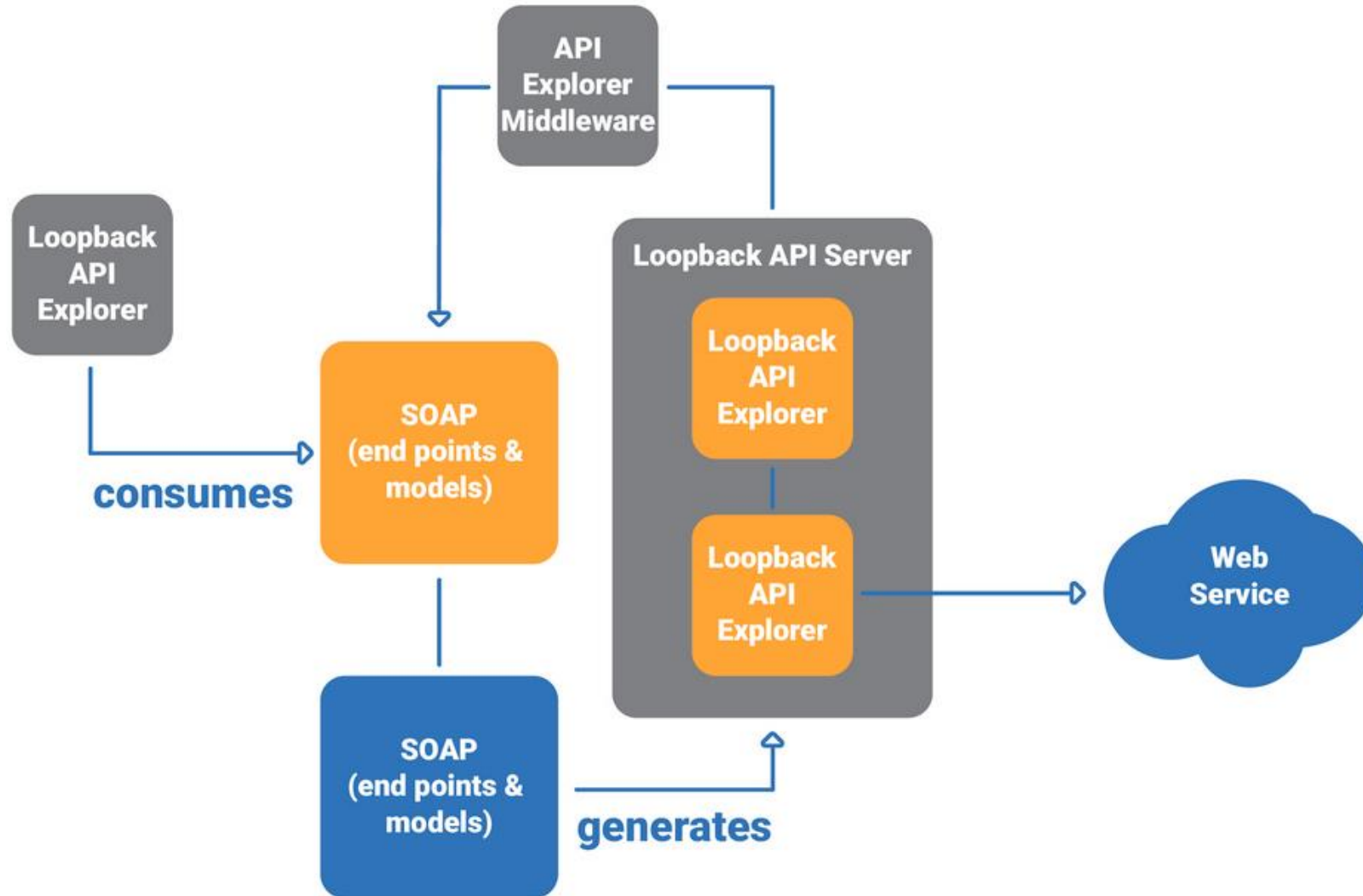
**APIs de sistema operacional:** São APIs fornecidas pelos sistemas operacionais, permitindo que os desenvolvedores acessem recursos do sistema, como acesso à câmera, geolocalização, armazenamento, entre outros. Essas APIs permitem que os aplicativos interajam com o sistema operacional e tirem proveito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

- **Android API:** É a API fornecida pelo sistema operacional Android, permitindo que os desenvolvedores acessem 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 acessem 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 acessem 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)

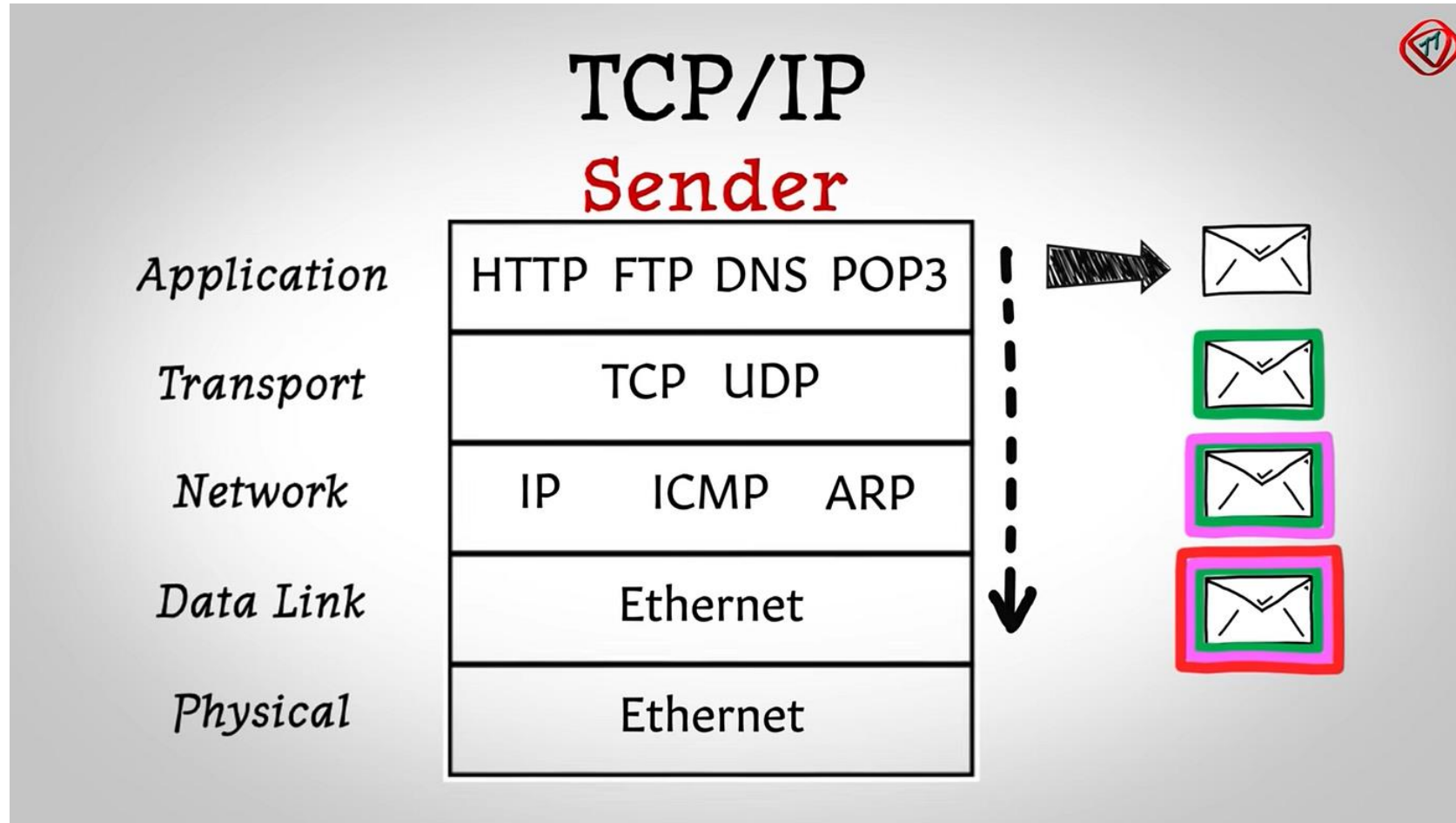
- **Protocolo de comunicação** usado para trocar informações estruturadas entre aplicativos na web. Ele é baseado em **XML** (Extensible Markup Language) e foi amplamente utilizado para implementar serviços web antes do advento do REST.
- É possível criar serviços SOAP em várias linguagens de programação, 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 SMTP (Simple Mail Transfer Protocol) e JMS (Java Message Service)



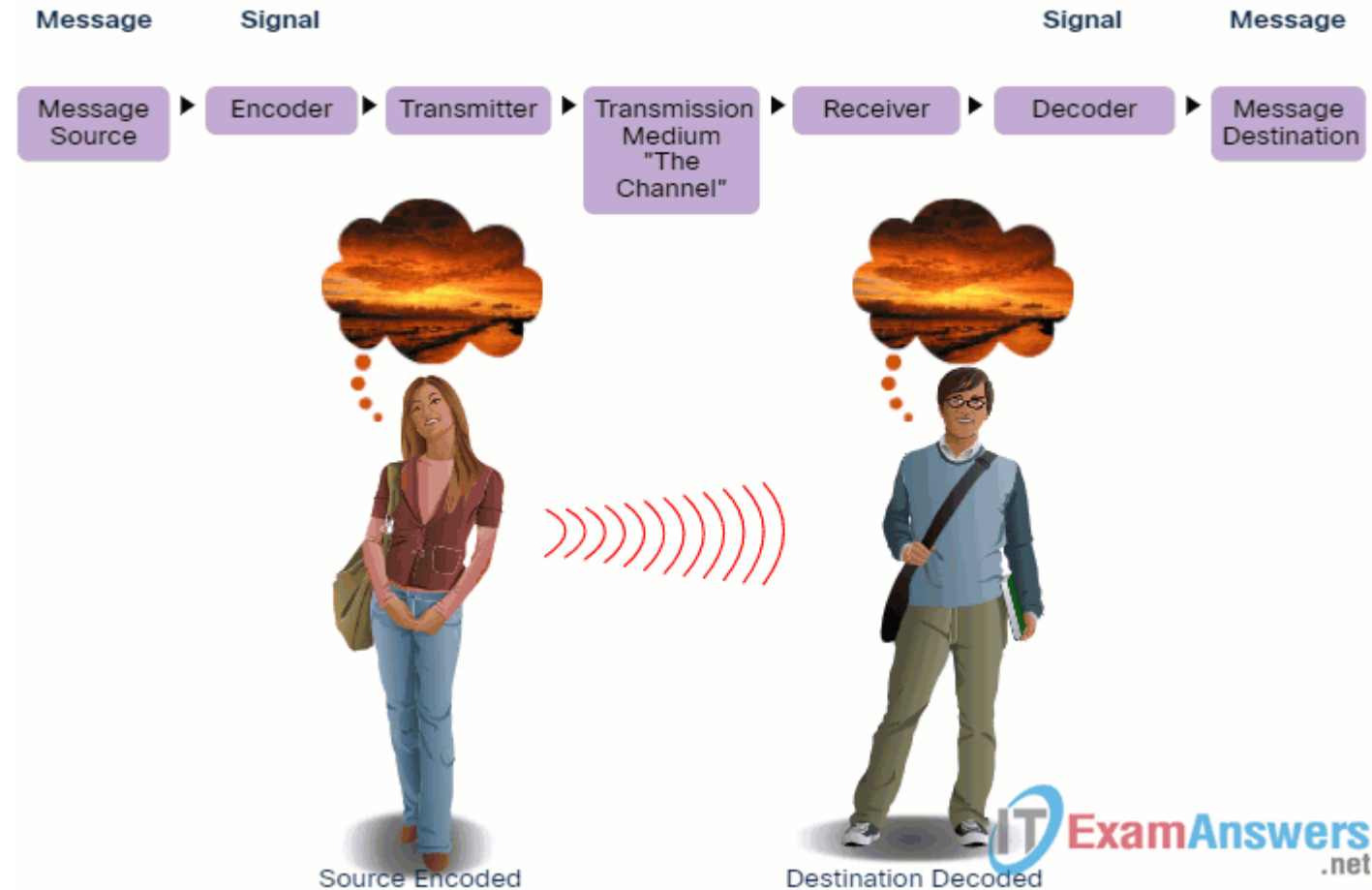
# Um breve parênteses... O que é Protocolo de comunicação???

- 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.

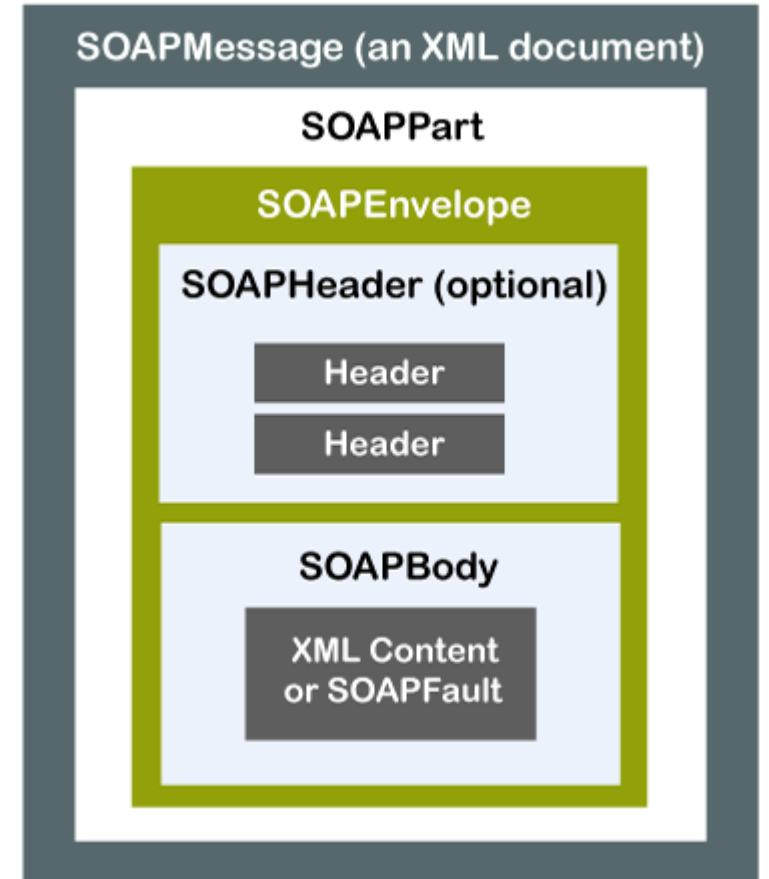
# Um breve parênteses... O que é Protocolo de comunicação???



# 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)

# Exemplo simples de WSDL gerado pelo chat GPT

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

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">
    </soap:binding>

    <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

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">
  </soap:binding>

  <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

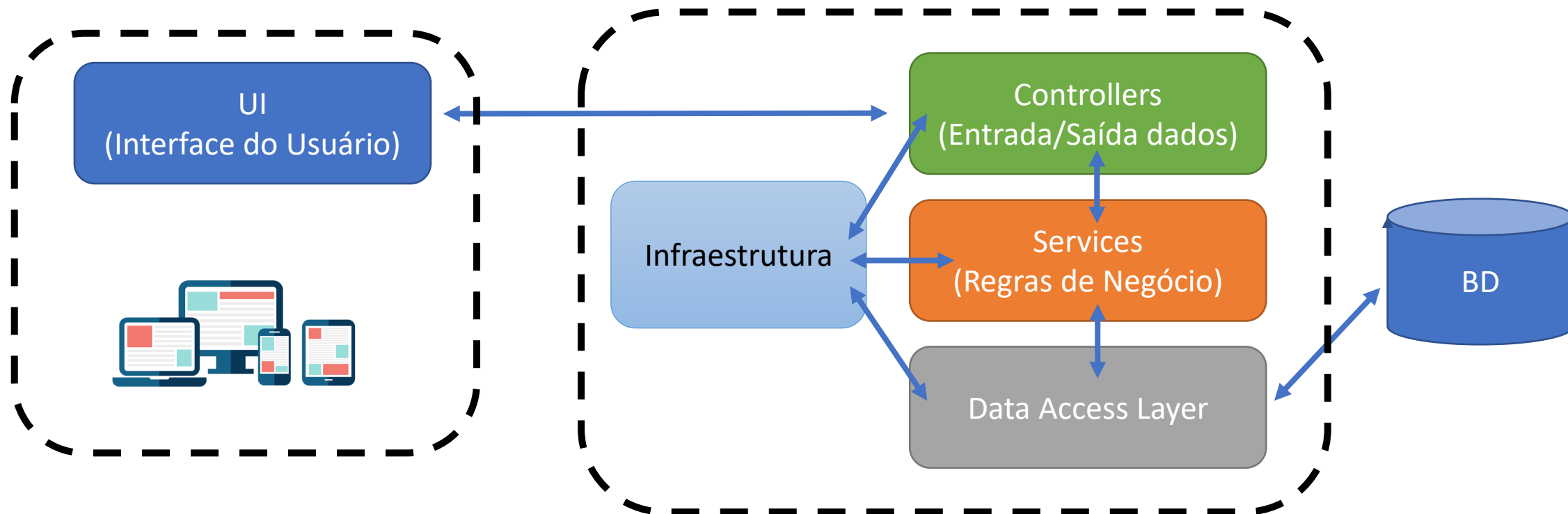
# Conclusão - SOAP

- 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 UUID?
- O que é JSON?
- O que é Spring Framework?



# 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*

HTTP FTP DNS POP3

*Transport*

TCP UDP

*Network*

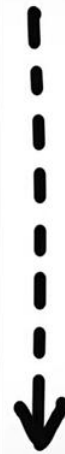
IP ICMP ARP

*Data Link*

Ethernet

*Physical*

Ethernet





# Tim Berners-lee

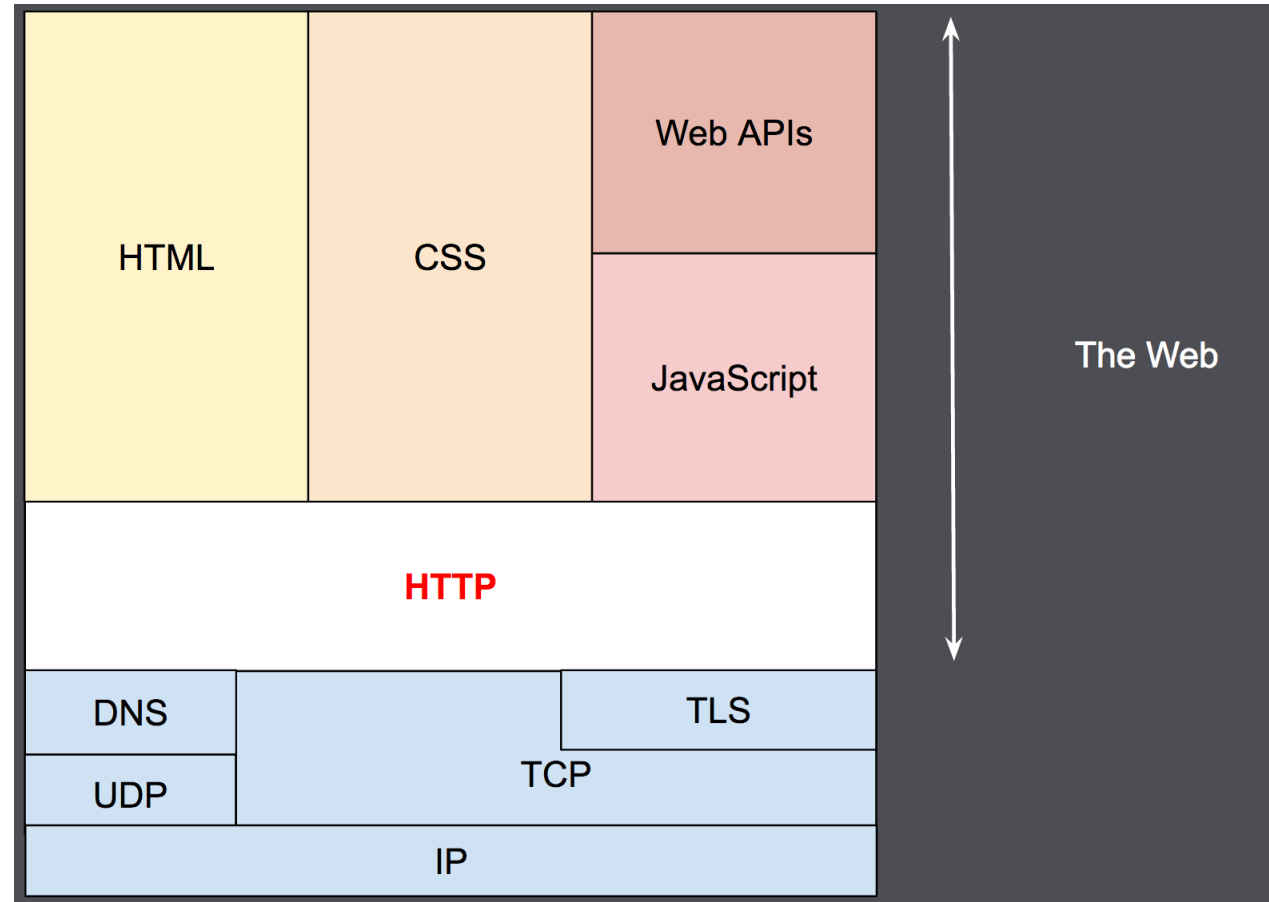


- Cientista da computação britânico que inventou a World Wide Web e desenvolveu as tecnologias fundamentais que a tornaram possível.
- Criou o HTTP como parte de seu trabalho no CERN (Organização Europeia para a Pesquisa Nuclear) nos anos 1980.
- Em 1989, Berners-Lee escreveu uma proposta chamada "Information 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>

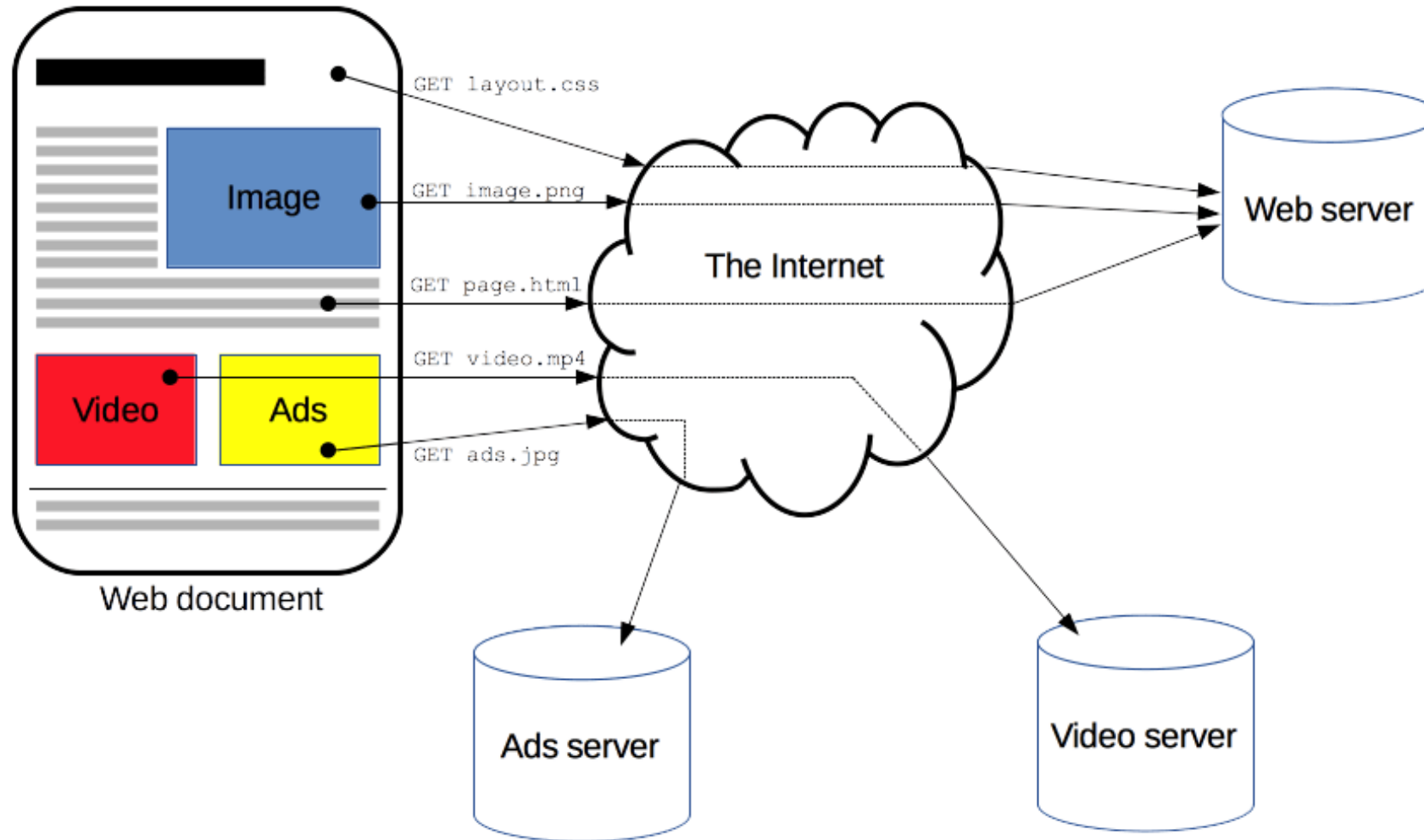


# Uma visão geral do HTTP



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

# Uma visão geral do HTTP



# Para carregar uma página, várias requisições são realizadas.

The screenshot displays the ATITUS website in a browser. The page features a large hero image of a smiling woman with the text "ASSUMA O CONTROLE DO SEU FUTURO E COMECE AGORA". Navigation links include "Atitus", "Cursos", "Diferenciais", "Inovação", and "Minha conta". Promotional tiles for "Graduação" and "Acelere sua Carreira" are visible on the right.

Below the browser window, the Chrome DevTools Network tab is open, showing a list of 17 network requests. The first request is the main HTML document, followed by 16 requests for various CSS and JavaScript files. All requests have a status of 200.

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	638-733014086cfe94f.js	https://www.atitus.edu.br/_next/static/chunks/638-733014086cfe94f.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	

Summary: 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

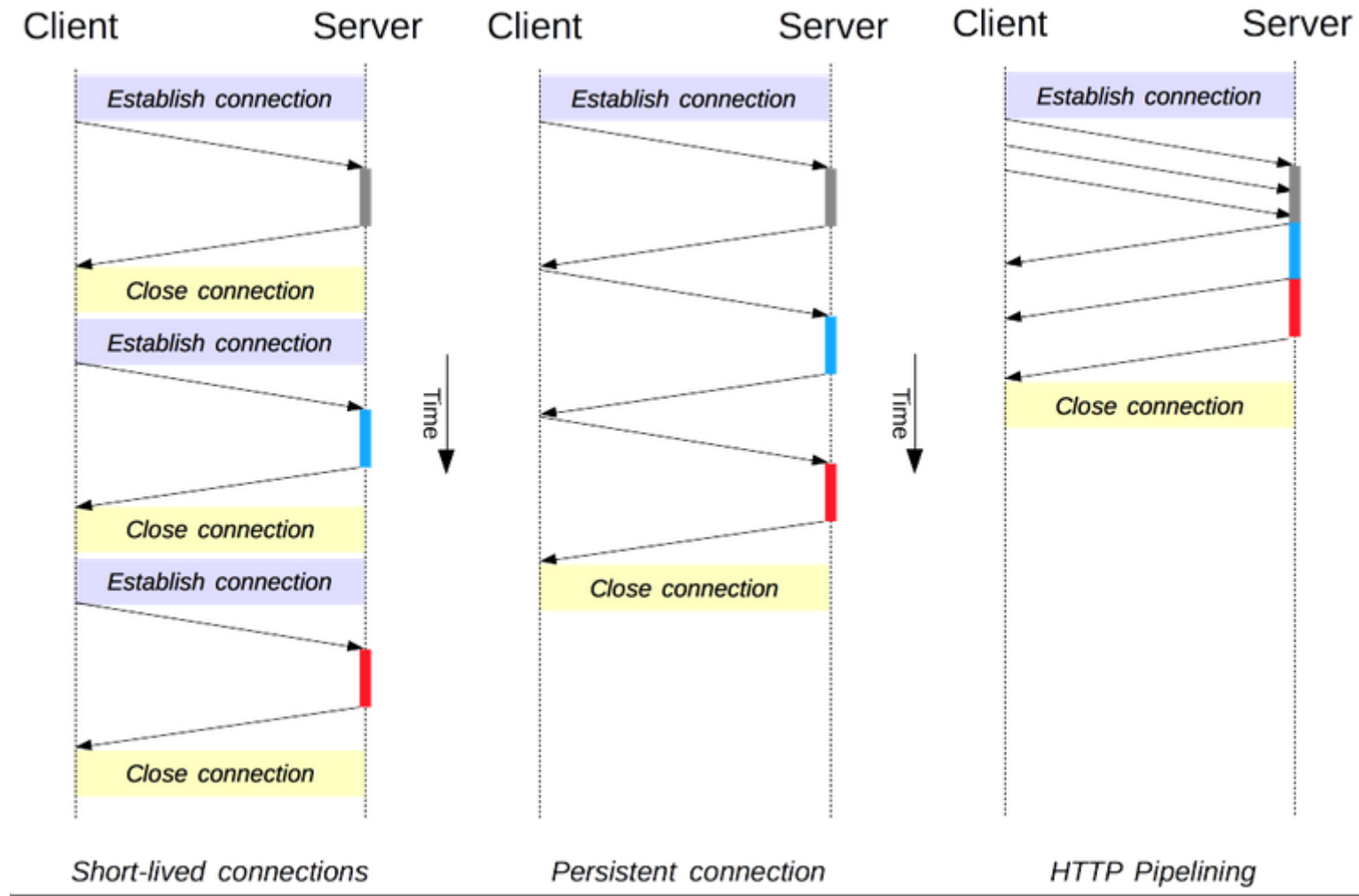
- Conceito de "multiplexação", permitindo que múltiplas solicitações e respostas fossem transmitidas simultaneamente pela mesma conexão, eliminando o problema de bloqueio de solicitações mais lentas.
- Compressão de cabeçalhos, reduzindo ainda mais o tamanho das requisições e respostas
- Carregamento 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



# Aspectos básicos do HTTP

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

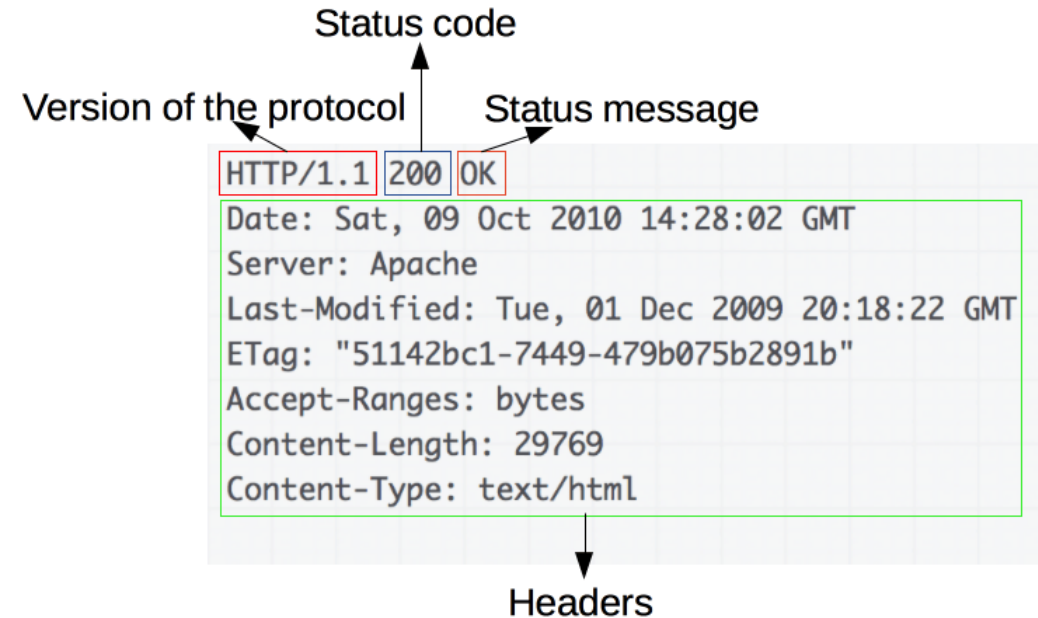
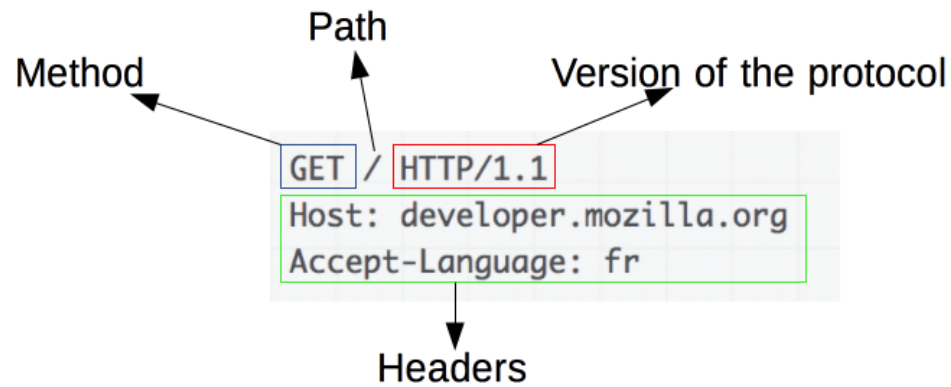
**i Nota:** \* O problema do carrinho de compras de *e-commerces* e o protocolo HTTP: como o protocolo HTTP não guarda o estado das requisições e respostas, é **impossível** fazer com que um site guarde as informações de um carrinho de compras **somente através do HTTP**. Por exemplo, imagine que você 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

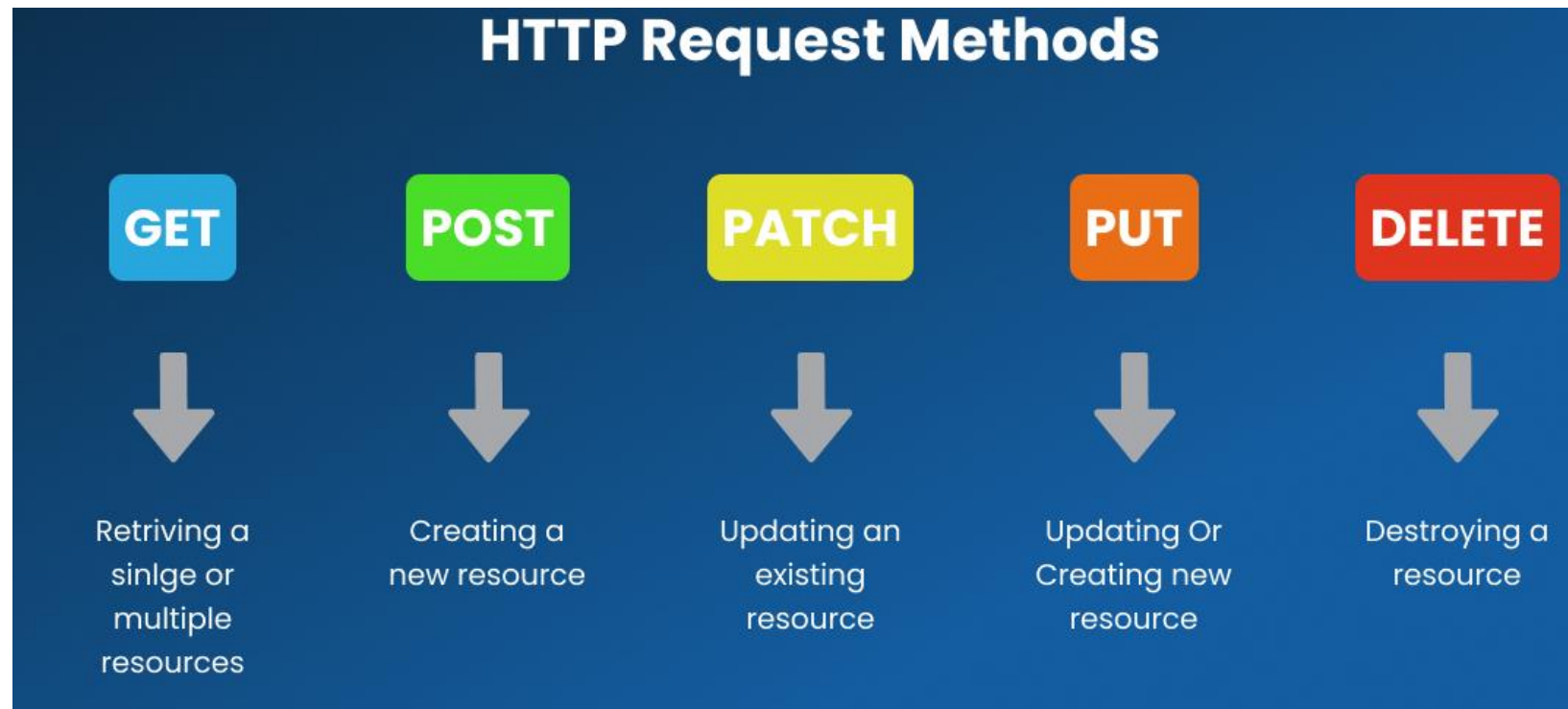
- Cache
- Relaxamento das restrições de origem
- Autenticação
- Proxy e tunelamento
- Sessões usando os cookies HTTP

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

# Response Status Code mais utilizados

## 200 OK

Esta 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 típica 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 clienSocket = 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;  
}
```



# 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 URIs, 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 compreensíveis.



# 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.

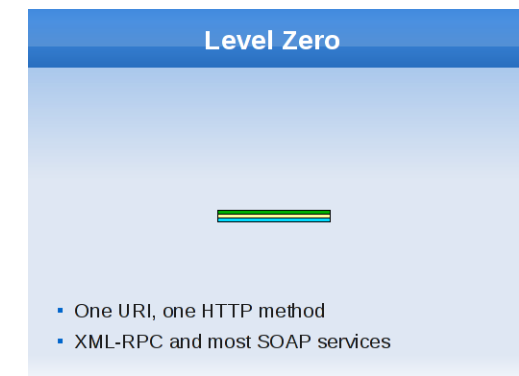
"The past is dust, and the future but smart nanodust."

# 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)



- 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 URI 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]

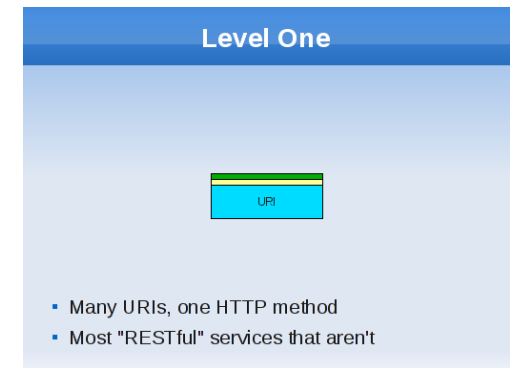
```
<appointmentRequest>  
  <slot doctor = "mjones" start = "1400" end = "1450"/>  
  <patient id = "jsmith"/>  
</appointmentRequest>
```

## Resposta

HTTP/1.1 200 OK  
[various headers]

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

# Nível Um (1) – Recursos Individuais



- 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

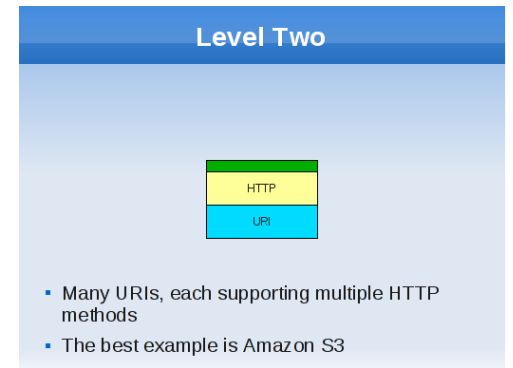
```
POST /slots/1234 HTTP/1.1  
[various other headers]
```

```
<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>
```



# 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]
```

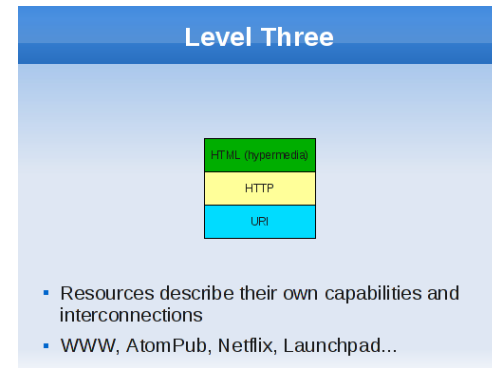
```
<appointmentRequest>
  <patient id = "jsmith"/>
</appointmentRequest>
```

## 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>
```

# 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**



# Nível Três (3) – Hiperarmídia (HATEOAS)

## 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

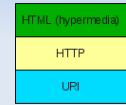
```
POST /slots/1234 HTTP/1.1
[various other headers]
```

```
<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>
```

### Level Three



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

# 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..

# 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 UUID?
- O que é JSON?
- O que é Spring Framework?

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 hífen, 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.

version

random

random

random

6e378977-d7ed-4214-bd49-45fb17b238a3

random

random

variant

df6fdea1-10c3-474c-ae62-e63def80de0b

time\_low

time\_mid

time\_hi\_and\_version

clock\_seq\_hi\_and\_res clock\_seq\_low

node



# 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 UUID?
- O que é JSON?
- O que é Spring Framework?