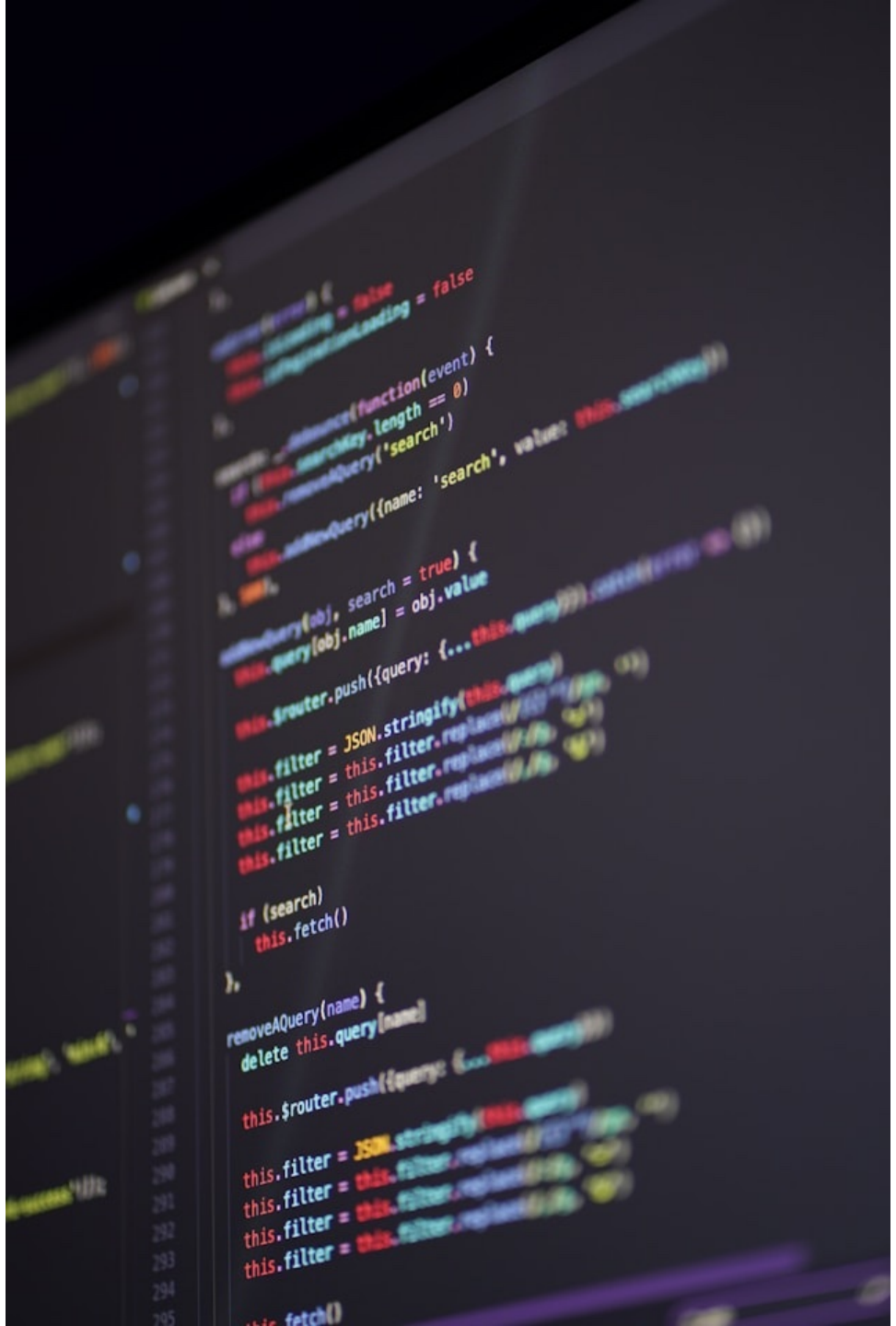


5954025 - Sistemas Distribuídos

Aula 09a - Web Services: REST

Prof. Dr. Denis M. L. Martins
DCM | FFCLRP | USP

Objetivos de Aprendizagem

- Compreender o conceito de **REST** (Representational State Transfer) como estilo arquitetural.
- Identificar os princípios fundamentais que regem uma API RESTful.
- Mapear as **Métodos HTTP** (GET, POST, PUT, DELETE) às suas operações lógicas.
- Interpretar **Códigos de Status HTTP** e códigos de erro comuns.
- Criar e consumir uma API REST utilizando Python (FastAPI) e JavaScript (Fetch API).
- Entender conceitos básicos de segurança (CORS, Autenticação via Tokens).

Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

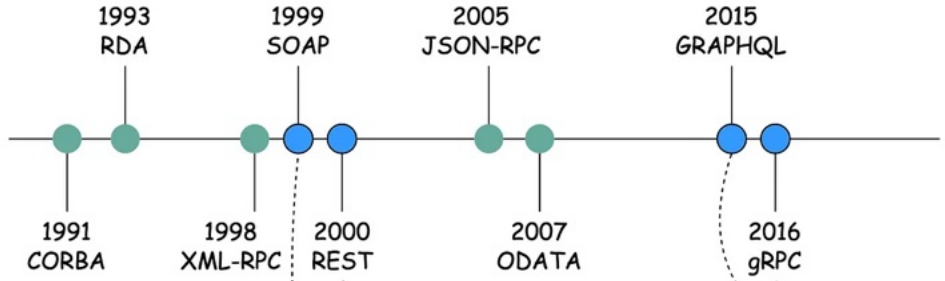
Relembrando

- SOAP é um protocolo baseado em XML que define uma maneira formal e rigorosa para os serviços se comunicarem.
- Em vez de criar protocolos complexos para cada interação, o REST propõe usar os recursos já existentes e bem definidos da Web: HTTP.

> Fonte da imagem: ByteByteGo.

API Architectural Styles Comparison

Source: altexsoft



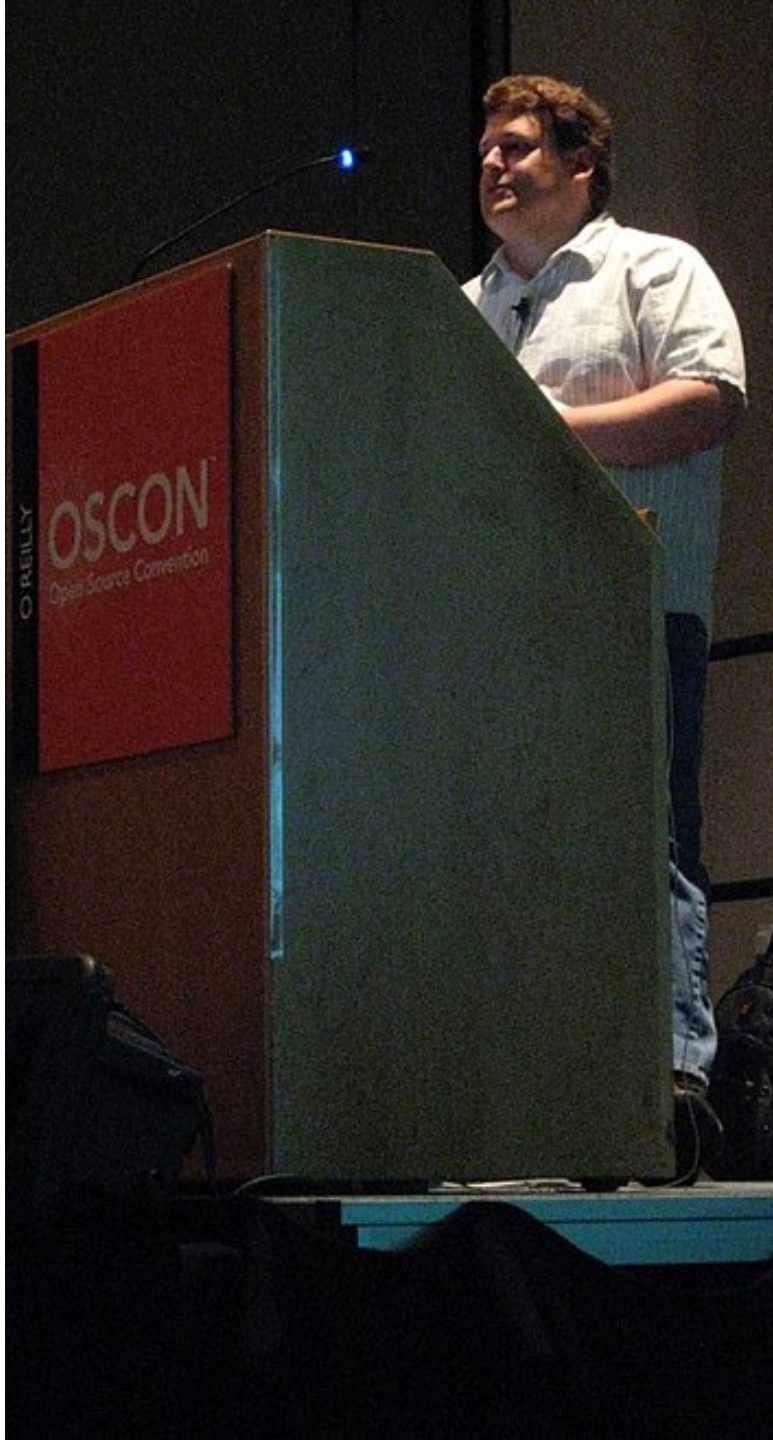
	SOAP (Simple Object Access Protocol)	REST (REpresentational State Transfer)	GraphQL	RPC (Remote Procedure Call)
Organized in terms of	enveloped message structure	compliance with six architectural constraints	schema & type system	local procedure call
Format	XML only	XML, JSON, HTML, plain text	JSON	JSON, XML, Protobuf, Thrift, FlatBuffers
Learning curve	Difficult	Easy	Medium	Easy
Community	Small	Large	Growing	Large
Use cases	- payment gateways - identity management - CRM solutions - financial and telecommunication services - legacy system support	- public APIs - simple resource-driven apps	- mobile APIs - complex systems - micro-services	- command and action-oriented APIs - high performance communication in massive micro-services systems

Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

Representational State Transfer (REST)

REST significa *Representational State Transfer* (Transferência de Estado Representacional).

- É um **estilo arquitetural**, não um protocolo ou padrão rígido.
- Foi definido por Roy Fielding em sua dissertação de doutorado no ano 2000.
- **Baseado em HTTP:** Utiliza o protocolo HTTP como mecanismo de transporte.



> Na imagem: Roy Fielding palestrando na OSCON 2008. Fonte: Wikipedia.

Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

Recurso

A abstração fundamental no REST é o **Recurso**.

- Um recurso é qualquer informação que pode ser nomeada (documento, imagem, pessoa, serviço).
- Cada recurso deve ser identificado exclusivamente por um **URI** (*Uniform Resource Identifier*).
- O estado do recurso em um determinado momento é transferido via **Representações** (ex: JSON).

Exemplo de URI: **https://api.exemplo.com/usuarios/123**

- **/usuarios**: Recurso genérico.
- **/123**: Identificador específico (ID) desse usuário.

Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

Projetando URIs (Nomes, não Verbos)

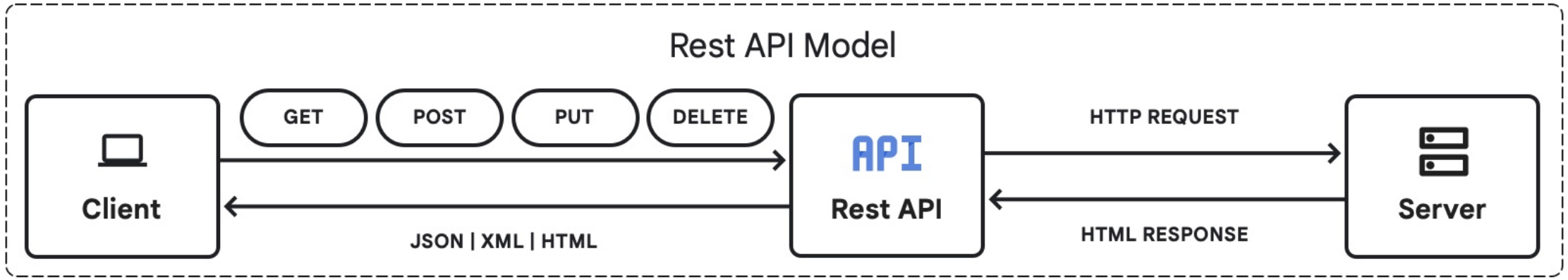
- URIs devem representar **recursos (substantivos)** e não ações (verbos).
- Utilize nomes no plural para coleções.
- A hierarquia é indicada por barras `/`.

Incorreto (RPC style)	Correto (RESTful)	Ação
`/getUsuarios`	`GET /usuarios`	Listar todos
`/criarUsuario`	`POST /usuarios`	Criar novo
`/deleteUsuario?id=1`	`DELETE /usuarios/1`	Remover um usuário

> Dica: O nome da URI deve refletir o recurso, não a ação (ex: `GET /users`, não `GET /getUsers`).

Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

REST API



> Fonte: https://cloud.google.com/discover/what-is-rest-api

Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

Métodos HTTP (Verbs)

REST usa os verbos do HTTP para definir ações sobre recursos:

Método	Ação Lógica	Idempotente?	Seguro?	Exemplo
GET	Ler/Consultar	Sim	Sim	**GET /users** (Lista todos)
POST	Criar Novo Recurso	Não	Não	**POST /users** (Criar usuário)
PUT	Substituir Totalmente	Sim	Não	**PUT /users/123** (Atualiza tudo)
PATCH	Atualizar Parcialmente	Não	Não	**PATCH /users/123** (Muda apenas email)
DELETE	Remover Recurso	Sim	Não	**DELETE /users/123**

> "Idempotente" significa que executar a operação várias vezes com o mesmo dado resulta no mesmo estado final.
>
> Ex: Criar um usuário duas vezes (POST) cria erro ou duplicidade, mas deletar (DELETE) uma vez já removeu.
>
> Mais informações em: https://www.restapitutorial.com/introduction/idempotence

Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

Servidor sempre responde com código HTTP

2xx (Sucesso):

- **200 OK**: Leitura ou atualização bem-sucedida.
- **201 Created**: Recurso criado com sucesso (geralmente após POST).

3xx (Redirecionamento): **301 Moved Permanently**. O recurso mudou de endereço.

4xx (Erro do Cliente):

- **400 Bad Request**: Dados inválidos enviados.
- **401 Unauthorized**: Falta de autenticação (não logado).
- **403 Forbidden**: Autenticado, mas sem permissão.
- **404 Not Found**: Recurso não existe na URI.

5xx (Erro do Servidor): **500 Internal Server Error**. Algo quebrou no backend.

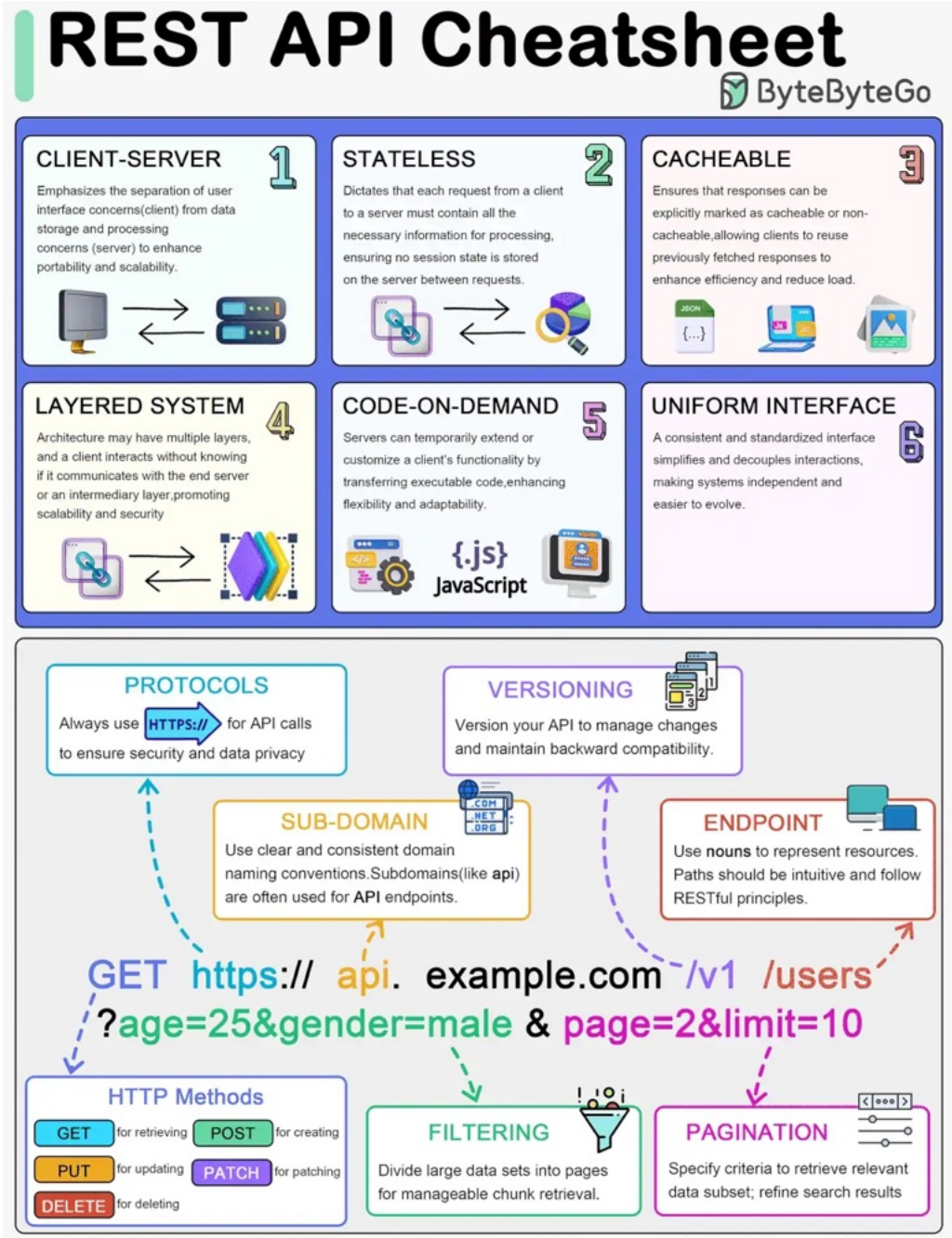
Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

Princípios Arquiteturais do REST

Para ser considerado "RESTful", um serviço deve seguir princípios:

- **Cliente-Servidor:** Separação de responsabilidades (UI vs Lógica de Dados).
- **Estado Sem Estado (Stateless):** Cada requisição contém toda a informação necessária. O servidor não guarda sessão do usuário entre requisições (ex: usa JWT no header).
- **Cacheabilidade:** Respostas devem indicar se podem ser armazenadas para melhorar performance.
- **Interface Uniforme:** Uso de URIs, Representações padronizadas e manipulação de recursos.

> Fonte da imagem: ByteByteGo.



Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

Statelessness em Detalhes

- Cada requisição do cliente deve conter **toda a informação necessária** para ser processada.
- O servidor não guarda "sessão" ou "estado da aplicação".
- Isso permite que a API escale horizontalmente com facilidade (balanceamento de carga).

> **Exemplo:** Em vez de pedir "próxima página", o cliente deve enviar `GET /produtos?pagina=2`.

Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

Formatos de Dados: JSON

REST precisa enviar e receber dados. O padrão moderno é **JSON**.

Exemplo de Resposta JSON:

```
{
  "id": 123,
  "nome": "Ana Silva",
  "email": "ana@exemplo.com",
  "ativo": true
}
```

- **Vantagens do JSON:** Leve, nativo em JavaScript, fácil de ler.
- **Headers Comuns:**
  - **Content-Type: application/json** (Diz que o corpo é JSON).
  - **Accept: application/json** (O cliente quer receber JSON).

Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

Consumindo uma API REST com JavaScript

Exemplo:

```
// 1. Requisição para listar usuários
fetch('https://api.exemplo.com/users')
  .then(response => {
    if (!response.ok) throw new Error('Erro na requisição');
    return response.json(); // Converte o corpo para JSON
  })
  .then(data => console.log(data))
  .catch(error => console.error(error));
```


Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

Consumindo uma API REST com JavaScript (cont.)

Outro exemplo:

```
// 2. Requisição para criar usuário (POST)
fetch('https://api.exemplo.com/users', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json'
  },
  body: JSON.stringify({ nome: 'Carlos', email: 'carlos@exemplo.com' })
})
.then(response => response.json())
.then(data => console.log('Usuário criado:', data));
```


Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

Exercício 1

1. Abra o console do navegador (F12).
2. Acesse uma API pública (ex: **https://jsonplaceholder.typicode.com/users**).
3. Execute um **fetch** para listar os usuários.
4. Tente acessar um usuário específico (**/users/1**) e verifique a resposta JSON.
5. Observe o cabeçalho **Content-Type**.

Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

Exercício 2

Vamos realizar o tutorial de criação de API simples com FastAPI disponível em:
https://denmartins.github.io/labs/rest-api-tutorial

Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

Desafios Práticos: CORS e Autenticação

- **CORS (Cross-Origin Resource Sharing):** O navegador bloqueia requisições de um domínio para outro por segurança.
  - *Solução:* O servidor deve enviar o header **Access-Control-Allow-Origin**.
- **Autenticação:** Como saber quem é o usuário?
  - **Tokens (JWT):** Enviados no Header **Authorization: Bearer <token>**.
  - **Cookies:** Usados em sessões tradicionais.
- **Exemplo de Header com Token:**

```
GET /users/123
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...
```


Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

CORS

Same-Origin Policy (SOP): É uma política de segurança implementada pelos navegadores web.

- O navegador bloqueia requisições de um domínio para outro por segurança.
- Por padrão, um script rodando em site-a.com só pode interagir com recursos do mesmo domínio (site-a.com).
- **Por quê?** Para evitar ataques maliciosos (Ex: Um site falso tentando roubar dados de sessão de outro site).

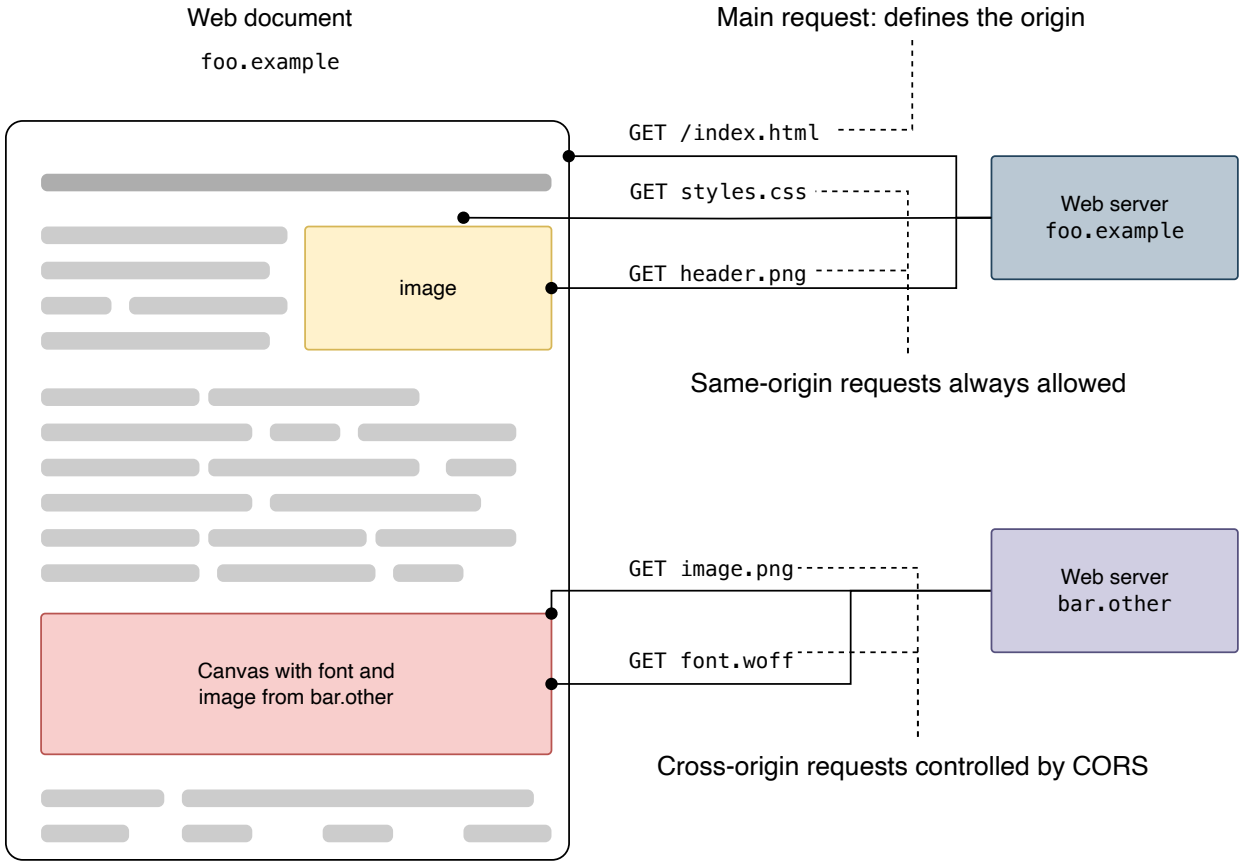
Cross-Origin Resource Sharing (CORS): É o mecanismo que **permite exceções controladas** à SOP. Ele define quais domínios externos têm permissão para acessar recursos de um domínio específico.

- **Onde configurar?** CORS é configurado **no lado do Servidor (Backend)**, através de *HTTP Headers*.
- **Header:** O header mais importante é **Access-Control-Allow-Origin**

Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

CORS

- Metadados HTTP que o servidor precisa enviar de volta ao navegador.
- Se o backend não incluir esses headers corretamente, mesmo que seu código esteja perfeito, o navegador vai bloquear a requisição e você receberá um erro de **CORS Policy Error**.



> Fonte da imagem: MDN.

Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

HATEOAS

- HATEOAS:*Hypermedia As The Engine Of Application State*.
- A resposta do servidor deve conter **links** que guiam o cliente sobre o que ele pode fazer a seguir.
- Isso torna a API auto-descritiva e permite que o cliente navegue sem conhecimento prévio da estrutura total.

Exemplo de JSON com links:

```
{
    "id": 1,
    "status": "pago",
    "links": [ { "rel": "entrega", "href": "/pedidos/1/rastreio" } ]
}
```


Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

HATEOAS: Exemplo

Imagine que um cliente solicita os detalhes de um usuário específico.

Requisição: **GET /usuarios/123**

Resposta sem HATEOAS (Apenas dados):

```
{
    "id": 123,
    "nome": "João Silva",
    "email": "joao@email.com"
}
```

Problema: O cliente precisa saber "por fora" (documentação externa) qual é a URL para ver os pedidos desse usuário ou para deletá-lo.

Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

HATEOAS: Exemplo (cont.)

Resposta com HATEOAS (Dados + Controles):

```
{
    "id": 123,
    "nome": "João Silva",
    "links": [
        { "rel": "self", "href": "/usuarios/123" },
        { "rel": "pedidos", "href": "/usuarios/123/pedidos" },
        { "rel": "deletar", "href": "/usuarios/123", "method": "DELETE" }
    ]
}
```

Vantagem: O servidor informa ao cliente **o que ele pode fazer a seguir** e onde deve ir para realizar essas ações.

Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

Cache e Performance

A restrição de **Cache** exige que as respostas se declarem cacheáveis ou não.

- **Vantagens:** Melhora a performance ao reduzir requisições duplicadas e diminui a latência percebida pelo usuário.
- **Cabeçalhos principais:**
  - **Cache-Control**: Define por quanto tempo a resposta é válida.
  - **ETag**: Um identificador único para uma versão específica de uma representação.
  - **Last-Modified**: Indica a data da última alteração no recurso.
- Se o recurso não mudou, o servidor retorna o status **304 Not Modified**.

Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

Segurança em APIs RESTful

- APIs precisam verificar a identidade (**Autenticação**) e as permissões (**Autorização**).
- **HTTP Basic Auth:** O cliente envia usuário e senha codificados em Base64 no cabeçalho.
- **API Keys:** Um valor único gerado pelo servidor e atribuído ao cliente para identificação.
- **OAuth:** Combina senhas e tokens para acesso altamente seguro, permitindo delegação de autoridade.
- **HTTPS:** É obrigatório usar criptografia SSL/TLS para proteger os dados em trânsito.

Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

Boas Práticas de Engenharia em REST

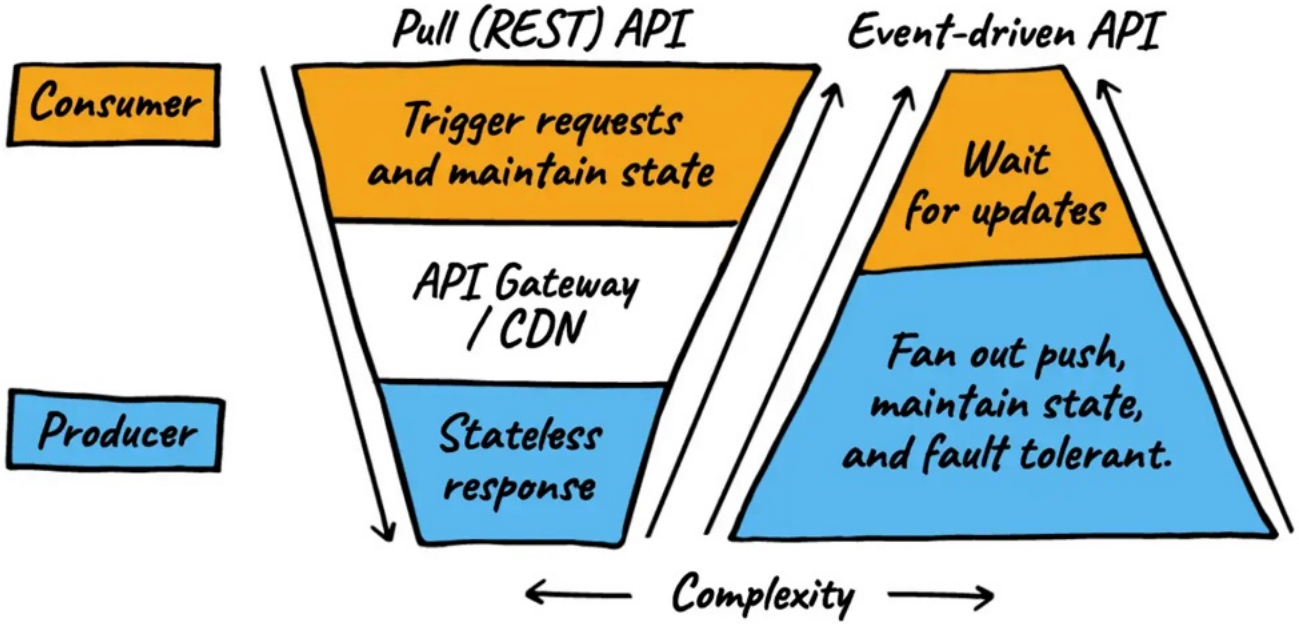
Para evitar "APIs ruins" no futuro:

1. **Versionamento:** Use `v1` na URL (`/api/v1/users`) para permitir mudanças sem quebrar apps antigos.
2. **Paginação:** Nunca retorne todos os dados se houver muitos. Use parâmetros `?page=1&limit=10`.
3. **Documentação:** Sempre use ferramentas como Swagger/OpenAPI para documentar a API.
4. **Manutenção de Estado:** Evite guardar estado no servidor (Stateless). Se precisar, use banco de dados ou cache externo (Redis).

Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

Em perspectiva

- **REST API**: O cliente é proativo. Ele deve sempre solicitar o estado atual dos dados.
- **Event-Driven API**: O cliente é reativo. A informação é enviada ao cliente assim que ocorre uma mudança relevante.



> Fonte da imagem: Ably.com.

Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

Conclusão

Resumo:

- REST é sobre **Recursos** identificados por **URIs**.
- Use os verbos HTTP corretamente (**GET**, **POST**, **PUT**, **DELETE**).
- Entenda os códigos de status para depurar erros.
- Consuma APIs usando **Fetch** ou bibliotecas como **Axios**.

Material Adicional:

- JWT Authentication in Django
- REST Quick Tips

Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

Dúvidas e Discussão

Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

Exercício e Questões

Exercício 1

1. Crie um arquivo **index.html** simples com um botão "Carregar Usuários".
2. No JavaScript, use o **fetch** para buscar dados da API pública **https://jsonplaceholder.typicode.com/users**.
3. Renderize os nomes e emails em uma lista na tela.
4. Adicione tratamento de erro (ex: mostrar mensagem se a rede estiver offline).

Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

Exercício 2

Faça o lab prático disponível em:
https://denmartins.github.io/labs/web-services-rest

Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

Questão 1

Explique a diferença entre **PUT** e **PATCH** no contexto de atualização de recursos. Dê um exemplo prático de quando usar cada um.

Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

Questão 2

Dado o cenário "O usuário deseja atualizar apenas o email de um perfil existente", qual é a melhor prática REST?

A) **GET /users/123**
B) **POST /users/123**
C) **PUT /users/123**
D) **PATCH /users/123**

Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

Questão 3

Uma requisição retorna o código **404 Not Found**. O que isso indica sobre a aplicação?

A) O servidor está com erro interno.
B) O cliente enviou dados incorretos.
C) O recurso solicitado não existe na URI.
D) A autenticação falhou.

Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

Questão 4

Analise o código abaixo e identifique o erro lógico:

```
fetch('/users', { method: 'GET' })
  .then(res => res.text())
  .then(data => console.log(data));
```


Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

Questão 5

O estilo arquitetural REST (Representational State Transfer) é amplamente adotado para o desenvolvimento de APIs web modernas. Um dos pilares fundamentais do REST é a natureza stateless (sem estado). Explique detalhadamente o que significa um sistema ser stateless no contexto de uma API RESTful e por que essa característica é crucial para garantir escalabilidade, resiliência e desempenho em ambientes distribuídos.

Além disso, utilize um exemplo prático (ex: gerenciamento de pedidos) para ilustrar como a abordagem orientada a recursos (Resource-based) do REST permite modelar o estado da aplicação usando URIs (Uniform Resource Identifiers), em vez de depender de sessões complexas no servidor.

Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

Questão 6

Em uma API RESTful que gerencia um catálogo de produtos, você precisa implementar as operações CRUD (Create, Read, Update, Delete). Os desenvolvedores frequentemente confundem o uso dos métodos HTTP PUT e POST.

- Explique a diferença semântica entre os métodos POST e PUT.
- Defina o conceito de Idempotência no contexto de APIs REST.
- Considerando este conceito, qual dos dois métodos (POST ou PUT) é considerado idempotente? Justifique sua resposta explicando por que a repetição da requisição não altera o estado final do sistema após a primeira execução bem-sucedida.

Prof. Dr. Denis M. L. Martins | martins.denis@usp.br