Apostila Aula 3: Introdução ao REST

Informações Gerais

- **Curso**: APIs REST e SOAP
- **Aula**: 3 de 20
- **Carga Horária**: 4 horas
- **Objetivo**: Compreender os princípios do REST e sua aplicação em APIs.
- **Conteúdo**:
- Definição de REST e os 6 princípios de Roy Fielding.
- Diferença entre REST e RESTful.
- Estrutura de uma API REST (recursos, métodos HTTP, endpoints).
- Atividade prática: Análise de exemplos de APIs REST públicas.

---

1. Definição de REST e os 6 Princípios de Roy Fielding

O que é REST?

REST (*Representational State Transfer*, ou Transferência de Estado Representacional) é um estilo arquitetural proposto por Roy Fielding em sua dissertação de doutorado em 2000. Ele descreve um conjunto de diretrizes para projetar sistemas distribuídos que sejam escaláveis, simples e interoperáveis. O REST é amplamente utilizado em APIs web, pois aproveita os recursos do protocolo HTTP para facilitar a comunicação entre clientes e servidores.

No contexto de APIs, o REST define como os recursos (como usuários, produtos ou pedidos) são identificados, manipulados e transferidos entre sistemas. Em vez de ser um protocolo rígido como o SOAP, o REST é um conjunto de princípios que orientam o design de APIs, permitindo flexibilidade e adaptação a diferentes necessidades.

Os 6 Princípios de Roy Fielding

Roy Fielding definiu seis princípios arquiteturais que caracterizam o REST. Esses princípios garantem que as APIs REST sejam eficientes, escaláveis e fáceis de usar. Vamos explorar cada um deles em detalhes:

1. Cliente-Servidor

O princípio cliente-servidor estabelece uma separação clara entre o cliente (que faz requisições) e o servidor (que processa requisições e retorna respostas). Essa separação permite que: - O cliente foque na interface do usuário e na experiência do cliente. - O servidor gerencie a lógica de negócios, armazenamento de dados e escalabilidade. - Cada componente evolua independentemente, desde que a interface (API) permaneça consistente.

Exemplo: Um aplicativo móvel (cliente) faz uma requisição GET para `/api/usuarios` e o servidor retorna uma lista de usuários em JSON. O aplicativo não precisa saber como o servidor armazena os dados (em um banco SQL, NoSQL, etc.), desde que a API siga o contrato estabelecido.

2. Sem Estado (*Stateless*)

Cada requisição do cliente para o servidor deve conter todas as informações necessárias para processá-la. O servidor não mantém informações sobre requisições anteriores, ou seja, não armazena o “estado” da sessão no servidor. Isso simplifica o design do servidor e melhora a escalabilidade, pois cada requisição pode ser processada por qualquer servidor disponível.

Exemplo: Para autenticar um usuário, a requisição deve incluir um token JWT no cabeçalho `Authorization`, como:

```
GET /api/usuarios/123
Authorization: Bearer abc123
```

O servidor valida o token em cada requisição, sem depender de sessões anteriores.

3. Cacheável

As respostas do servidor devem indicar se podem ser armazenadas em cache pelo cliente, reduzindo a carga no servidor e melhorando o desempenho. O HTTP fornece mecanismos como cabeçalhos `Cache-Control`, `ETag` e `Last-Modified` para controlar o cache.

Exemplo: Uma requisição GET para `/api/produtos` pode incluir o cabeçalho:

```
Cache-Control: max-age=3600
```

Isso indica que o cliente pode armazenar a resposta por 1 hora, evitando novas requisições desnecessárias.

4. Interface Uniforme

O REST define uma interface padronizada para interagir com recursos, garantindo consistência e simplicidade. A interface uniforme inclui quatro componentes principais: - **Identificação de recursos**: Cada recurso é identificado por uma URL única (ex.: `/api/usuarios/123`). - **Manipulação de recursos por representações**: Os recursos são manipulados por meio de representações, como JSON ou XML, enviadas nas requisições e respostas. - **Mensagens autoexplicativas**: As requisições e respostas usam métodos HTTP (GET, POST, etc.) e cabeçalhos para descrever a intenção. - **HATEOAS** (*Hypermedia as the Engine of Application State*): O cliente pode navegar pela API usando links fornecidos nas respostas.

Exemplo de HATEOAS: Uma resposta para `GET /api/usuarios/123` pode incluir links:

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

O cliente pode usar esses links para acessar recursos relacionados, como os pedidos do usuário.

5. Sistema em Camadas

O REST permite que o sistema seja dividido em camadas (ex.: camada de apresentação, lógica de negócios, banco de dados), onde cada camada só interage com a camada adjacente. Isso melhora a escalabilidade e a segurança, pois o cliente não precisa saber como o servidor está estruturado internamente.

Exemplo: Um cliente faz uma requisição para uma API REST que passa por um balanceador de carga, uma camada de autenticação e, finalmente, o servidor de

aplicação. O cliente só interage com a API, sem conhecer as camadas intermediárias.

6. Código Sob Demanda (Opcional)

Este princípio, menos comum, permite que o servidor envie código executável (ex.: JavaScript) para o cliente, estendendo suas funcionalidades. Em APIs REST, isso raramente é usado, mas pode ser aplicado em casos como widgets dinâmicos.

Exemplo: Um servidor pode enviar um script JavaScript para renderizar um gráfico interativo no cliente.

Por que esses Princípios são Importantes?

Os princípios de Fielding garantem que as APIs REST sejam: - **Escaláveis**: O modelo sem estado e cacheável suporta grandes volumes de requisições. - **Flexíveis**: A interface uniforme permite que diferentes clientes (web, móvel, IoT) usem a mesma API. - **Manuteníveis**: A separação cliente-servidor e o sistema em camadas facilitam atualizações. - **Interoperáveis**: O uso de padrões HTTP e formatos como JSON garante compatibilidade.

---

2. Diferença entre REST e RESTful

REST vs. RESTful

Os termos **REST** e **RESTful** são frequentemente usados de forma intercambiável, mas há uma distinção técnica: - **REST**: Refere-se ao estilo arquitetural definido por Roy Fielding, com os seis princípios descritos acima. É um conjunto de diretrizes teóricas para projetar sistemas distribuídos. - **RESTful**: Refere-se a uma implementação ou sistema que segue os princípios do REST. Uma API é considerada RESTful se adere estritamente aos princípios de Fielding, como interface uniforme, ausência de estado e uso de hiperlinks (HATEOAS).

Diferenças Práticas

Na prática, muitas APIs são chamadas de “RESTful” mesmo sem seguir todos os princípios do REST, especialmente o HATEOAS, que é menos comum. Por exemplo: - Uma API que usa HTTP, métodos GET/POST e JSON, mas não inclui links HATEOAS, é frequentemente chamada de RESTful, embora tecnicamente não

siga todos os princípios do REST. - Uma API verdadeiramente RESTful implementaria todos os princípios, incluindo respostas com hiperlinks para navegação dinâmica.

Exemplo de API RESTful Completa: Uma requisição para `/api/pedidos/456` retorna:

```
{
 "id": 456,
 "produto": "Smartphone",
 "status": "pendente",
 "links": [
   { "rel": "self", "href": "/api/pedidos/456" },
   { "rel": "cancelar", "href": "/api/pedidos/456/cancelar", "method": "P
 ]
}
```

Aqui, a API é RESTful porque inclui HATEOAS, permitindo que o cliente descubra ações disponíveis (como cancelar o pedido) diretamente pela resposta.

Exemplo de API "Parcialmente RESTful":

```
{
 "id": 456,
 "produto": "Smartphone",
 "status": "pendente"
}
```

Essa API usa HTTP e JSON, mas não inclui HATEOAS, sendo apenas parcialmente RESTful.

Quando uma API é Considerada RESTful?

Para ser considerada RESTful, uma API deve: - Usar métodos HTTP de forma semântica (GET para leitura, POST para criação, etc.). - Identificar recursos por URLs claras e consistentes. - Ser sem estado, com cada requisição contendo todas as informações necessárias. - Preferencialmente, implementar HATEOAS para navegação dinâmica.

Na prática, muitas APIs modernas são chamadas de RESTful mesmo sem HATEOAS, desde que sigam os outros princípios e usem HTTP e JSON de forma consistente.

---

3. Estrutura de uma API REST

Uma API REST é estruturada em torno de **recursos**, **métodos HTTP** e **endpoints**, que juntos definem como os clientes interagem com o servidor. Vamos explorar cada componente:

Recursos

Um **recurso** é qualquer entidade ou objeto que a API manipula, como um usuário, um pedido ou um produto. Cada recurso é identificado por uma URL única e representado em um formato como JSON ou XML.

Exemplo: - Recurso: Usuário - URL: `/api/usuarios/123` - Representação (JSON):

```
{
 "id": 123,
 "nome": "Maria Silva",
 "email": "maria@exemplo.com"
}
```

Boas Práticas para Recursos: - Use substantivos no plural para coleções (ex.: `/usuarios`). - Use IDs para recursos específicos (ex.: `/usuarios/123`). - Estruture hierarquicamente para recursos relacionados (ex.: `/usuarios/123/pedidos`).

Métodos HTTP

Os métodos HTTP definem as ações que podem ser realizadas em um recurso. Os mais comuns em APIs REST são: - **GET**: Recupera um recurso ou uma lista de recursos. - Exemplo: `GET /api/usuarios` retorna uma lista de usuários. - **POST**: Cria um novo recurso. - Exemplo: `POST /api/usuarios` com um payload JSON cria um novo usuário. - **PUT**: Atualiza um recurso existente. - Exemplo: `PUT /api/usuarios/123` atualiza os dados do usuário com ID 123. - **DELETE**: Remove um recurso. - Exemplo: `DELETE /api/usuarios/123` exclui o usuário

com ID 123. - **PATCH**: Atualiza parcialmente um recurso. - Exemplo: `PATCH /api/usuarios/123` altera apenas o email do usuário.

Exemplo de Fluxo: 1. Criar um usuário: ``` POST /api/usuarios Content-Type: application/json

{ “nome”: “João Silva”, “email”: “joao@exemplo.com” } `Resposta:`json { “id”: 124, “nome”: “João Silva”, “email”: “joao@exemplo.com” } `2. Recuperar o usuário:` GET /api/usuarios/124 `Resposta:`json { “id”: 124, “nome”: “João Silva”, “email”: “joao@exemplo.com” } ```

Endpoints

Um **endpoint** é uma URL específica que representa um recurso ou uma ação. Em APIs REST, os endpoints são projetados para serem intuitivos e consistentes.

Exemplo de Endpoints para uma API de Loja Virtual: - `GET /api/produtos`: Lista todos os produtos. - `GET /api/produtos/456`: Detalhes do produto com ID 456. - `POST /api/produtos`: Cria um novo produto. - `PUT /api/produtos/456`: Atualiza o produto com ID 456. - `DELETE /api/produtos/456`: Exclui o produto com ID 456. - `GET /api/usuarios/123/pedidos`: Lista os pedidos do usuário com ID 123.

Boas Práticas para Endpoints: - **Consistência**: Use convenções consistentes (ex.: sempre plural para coleções). - **Clareza**: Evite verbos (ex.: `/usuarios` em vez de `/getUsuarios`). - **Filtros e Paginação**: Use query parameters para filtros (ex.: `/produtos?categoria=eletronicos`). - **Versionamento**: Inclua a versão da API na URL (ex.: `/v1/usuarios`) para facilitar atualizações.

Estrutura de uma Requisição e Resposta

- **Requisição**:
- Método HTTP (ex.: GET, POST).
- URL do endpoint.
- Cabeçalhos (ex.: `Content-Type: application/json`, `Authorization`).
- Corpo (para POST, PUT, PATCH).
- **Resposta**:
- Código de status (ex.: 200 OK, 404 Not Found).
- Cabeçalhos (ex.: `Content-Type`, `Cache-Control`).

- Corpo (ex.: JSON com os dados do recurso).

Exemplo Completo: Requisição:

```
POST /api/usuarios
Content-Type: application/json
Accept: application/json

{
 "nome": "Ana Costa",
 "email": "ana@exemplo.com"
}
```

Resposta:

```
HTTP/1.1 201 Created
Content-Type: application/json

{
 "id": 125,
 "nome": "Ana Costa",
 "email": "ana@exemplo.com",
 "links": [
   { "rel": "self", "href": "/api/usuarios/125" }
 ]
}
```

---

4. Atividade Prática: Análise de Exemplos de APIs REST Públicas

Objetivo da Atividade

Analisar APIs REST públicas para entender como os princípios do REST são aplicados na prática, identificando recursos, métodos HTTP, endpoints e padrões de design.

Instruções

1. **Seleção de APIs Públicas**:
2. Escolha uma das seguintes APIs REST públicas (ou outra de sua preferência, com aprovação do professor):
   - **JSONPlaceholder**: API fictícia para testes (https://jsonplaceholder.typicode.com).
   - **ReqRes**: API de simulação (https://reqres.in).
   - **OpenWeatherMap**: API de previsão do tempo (https://openweathermap.org/api).
   - **GitHub API**: API para acesso a repositórios e usuários (https://api.github.com).
3. Acesse a documentação oficial da API para entender os endpoints disponíveis, métodos suportados e formatos de resposta.
4. **Análise de Endpoints**:
5. Selecione pelo menos **três endpoints** da API escolhida, cada um usando um método HTTP diferente (ex.: GET, POST, DELETE).
6. Para cada endpoint, analise:
   - **URL**: Qual é o recurso representado?
   - **Método HTTP**: Qual ação o método realiza?
   - **Cabeçalhos**: Quais cabeçalhos são obrigatórios ou recomendados?
   - **Corpo da Requisição**: Se aplicável, qual é o formato (JSON, XML)?
   - **Resposta**: Qual é o código de status esperado e o formato da resposta?
   - **Princípios REST**: Como o endpoint reflete os princípios de Fielding (ex.: interface uniforme, sem estado)?
7. **Teste com Postman**:
8. Use o Postman para fazer requisições aos endpoints selecionados.
9. Exemplo com JSONPlaceholder:
   - **GET /users**: Lista todos os usuários.
   - **POST /posts**: Cria um novo post.
   - **DELETE /posts/1**: Exclui o post com ID 1.
10. Registre as respostas, incluindo códigos de status, cabeçalhos e corpo.
11. **Relatório**:
12. Escreva um relatório (mínimo de 300 palavras) descrevendo:
    - A API escolhida e seu propósito.
    - Os três endpoints analisados, com detalhes sobre URL, método, cabeçalhos, corpo e resposta.

- Como a API segue (ou não) os princípios do REST (ex.: uso de HATEOAS, sem estado).
- Desafios encontrados (ex.: autenticação, limites de requisições).

13. Inclua capturas de tela das requisições feitas no Postman.

Exemplo de Análise (JSONPlaceholder)

Endpoint 1: - **URL**: `GET https://jsonplaceholder.typicode.com/users` - **Método**: GET - **Cabeçalhos**: `Accept: application/json` - **Corpo**: Nenhum. - **Resposta Esperada**: `json [ { "id": 1, "name": "Leanne Graham", "email": "Sincere@april.biz" }, ... ]` - **Código de Status**: 200 OK - **Princípios REST**: Interface uniforme (recurso `/users` identificado por URL), sem estado (requisição independente), cacheável (resposta pode ser armazenada).

Endpoint 2: - **URL**: `POST https://jsonplaceholder.typicode.com/posts` - **Método**: POST - **Cabeçalhos**: `Content-Type: application/json`, `Accept: application/json` - **Corpo**: `json { "title": "Novo Post", "body": "Conteúdo do post", "userId": 1 }` - **Resposta Esperada**: `json { "id": 101, "title": "Novo Post", "body": "Conteúdo do post", "userId": 1 }` - **Código de Status**: 201 Created - **Princípios REST**: Interface uniforme (cria um novo recurso), sem estado.

Endpoint 3: - **URL**: `DELETE https://jsonplaceholder.typicode.com/posts/1` - **Método**: DELETE - **Cabeçalhos**: Nenhum. - **Corpo**: Nenhum. - **Resposta Esperada**: Corpo vazio. - **Código de Status**: 200 OK - **Princípios REST**: Interface uniforme (exclui um recurso específico), idempotente.

Entrega

Cada aluno deve entregar um relatório individual com: - Descrição da API escolhida e seu propósito. - Análise detalhada dos três endpoints (URL, método, cabeçalhos, corpo, resposta, princípios REST). - Capturas de tela das requisições no Postman. - Reflexão (300 palavras) sobre como a API reflete os princípios do REST e os desafios enfrentados.

Conclusão

Esta aula introduziu os fundamentos do REST, incluindo os seis princípios de Roy Fielding, a diferença entre REST e RESTful, e a estrutura de uma API REST baseada em recursos, métodos HTTP e endpoints. Esses conceitos são a base para o desenvolvimento e consumo de APIs REST nas próximas aulas. A atividade prática permitiu aos alunos explorar APIs REST públicas, aplicando os conceitos aprendidos e analisando como os princípios do REST são implementados na prática.

Próximos passos: Na Aula 4, abordaremos a estrutura de dados em APIs REST, focando em JSON e XML, e exploraremos como modelar dados de forma eficiente.