4.1 Introdução a JSON e XML

APIs REST dependem de formatos de dados estruturados para transmitir informações entre cliente e servidor. Os formatos mais comuns são **JSON** (JavaScript Object Notation) e **XML** (eXtensible Markup Language), ambos legíveis por humanos e máquinas, amplamente suportados e adequados para diferentes cenários. Este capítulo foca em JSON, devido à sua predominância em APIs REST modernas, mas também explora XML para contextos onde é relevante, como sistemas legados ou integrações específicas.

JSON (JavaScript Object Notation)

JSON é um formato leve baseado em texto, inspirado na sintaxe de objetos JavaScript, mas independente de linguagem. Ele é o padrão de fato para APIs REST devido à sua simplicidade, facilidade de parsing e suporte universal em frameworks e linguagens modernas.

- **Características**:
- Estrutura baseada em pares chave-valor (ex.: `{"nome": "Ana"}`).
- Suporta tipos de dados: strings, números, booleanos, arrays, objetos e null.
- Fácil de serializar (converter para string) e desserializar (converter para objetos nativos).
- Compacto, reduzindo o tamanho das mensagens em comparação com XML.
- **Exemplo**: `json { "id": 1, "titulo": "Estudar REST APIs", "concluida": false, "tags": ["estudo", "programação"], "criadoEm": "2025-05-26T00:17:00Z" }`
- **Vantagens**:
- **Legibilidade**: Fácil de ler e escrever para desenvolvedores.
- **Performance**: Menor overhead, ideal para aplicações com alto volume de requisições.
- **Integração**: Suporte nativo em frameworks como Express (Node.js), Flask (Python) e em clientes JavaScript (ex.: fetch, Axios).
- **Casos de Uso**:
- APIs RESTful modernas (ex.: PokéAPI, JSONPlaceholder).
- Integração com front-ends JavaScript (React, Angular, Vue).
- Aplicativos móveis que requerem respostas rápidas.

Parsing JSON

JSON é facilmente processado em diferentes linguagens: - **JavaScript**: `javascript const json = '{"titulo":"Estudar REST","concluida":false}'; const objeto = JSON.parse(json); // Converte string JSON em objeto console.log(objeto.titulo); // "Estudar REST"` - **Python**: `python import json json_str = '{"titulo":"Estudar REST","concluida":false}' objeto = json.loads(json_str) # Converte string JSON em dicionário print(objeto['titulo']) # "Estudar REST"`

XML (eXtensible Markup Language)

XML é um formato baseado em tags, semelhante ao HTML, mas projetado para armazenar e transportar dados. Embora menos comum em APIs REST modernas, ele é usado em sistemas legados, APIs SOAP e cenários que exigem validação rigorosa.

- **Características**:
- Usa tags personalizadas (ex.: `<tarefa><id>1</id></tarefa>`).
- Suporta atributos (ex.: `<tarefa id="1">`).
- Mais verboso que JSON, aumentando o tamanho das mensagens.
- Permite validação com esquemas (XSD - XML Schema Definition).
- **Exemplo**: `xml <tarefa> <id>1</id> <titulo>Estudar REST APIs</titulo> <concluida>false</concluida> <tags> <tag>estudo</tag> <tag>programação</tag> </tags> <criadoEm>2025-05-26T00:17:00Z</criadoEm> </tarefa>`
- **Vantagens**:
- Suporte a esquemas para validação rigorosa.
- Ideal para documentos complexos com metadados (ex.: relatórios corporativos).
- Compatibilidade com sistemas legados.
- **Desvantagens**:
- Maior tamanho de dados, impactando performance.
- Parsing mais complexo, exigindo bibliotecas específicas (ex.: `xml2js` em Node.js, `xml.etree.ElementTree` em Python).
- **Casos de Uso**:
- APIs SOAP (ex.: serviços bancários).

- Integração com sistemas corporativos que usam XML.
- Documentos estruturados, como relatórios ou configurações.

Parsing XML

- **JavaScript** (com `xml2js`): `javascript const xml2js = require('xml2js'); const xml = `<tarefa><id>1</id><titulo>Estudar REST</titulo></tarefa>`; xml2js.parseString(xml, (err, result) => { console.log(result.tarefa.titulo[0]); // "Estudar REST" });`
- **Python**: `python import xml.etree.ElementTree as ET xml = '<tarefa><id>1</id><titulo>Estudar REST</titulo></tarefa>' root = ET.fromstring(xml) print(root.find('titulo').text) # "Estudar REST"`

JSON vs. XML

- **JSON**:
- Predominante em APIs REST devido à sua leveza e facilidade de uso.
- Ideal para aplicações web e móveis.
- Menor overhead (menos bytes em requisições/respostas).
- **XML**:
- Usado em sistemas legados ou cenários com validação complexa.
- Mais verboso, impactando performance em APIs de alto tráfego.
- **Conclusão**: JSON é o padrão para APIs REST modernas, mas XML ainda é relevante em contextos específicos. Este capítulo focará em JSON, com exemplos ocasionais em XML para comparação.

4.2 Estrutura de Requisições e Respostas em APIs REST

APIs REST utilizam o protocolo HTTP para enviar requisições e receber respostas. A estrutura de ambas é composta por métodos, cabeçalhos, corpo e códigos de status, que juntos definem como os dados são transmitidos e processados.

Estrutura de uma Requisição

Uma requisição HTTP em uma API REST inclui: 1. **Método HTTP**: Define a ação: - `GET`: Recupera dados. - `POST`: Cria um recurso. - `PUT`: Atualiza um recurso completamente. - `PATCH`: Atualiza parcialmente um recurso. - `DELETE`: Remove um recurso. 2. **URI**: Identifica o recurso (ex.: `/tarefas/1` para a tarefa com ID 1). 3. **Cabeçalhos**: - `Content-Type`: Formato do corpo (ex.: `application/json` ou `application/xml`). - `Accept`: Formato desejado para a resposta (ex.: `application/json`). - `Authorization`: Credenciais ou tokens (ex.: `Bearer token123`). - `User-Agent`: Informações sobre o cliente. 4. **Corpo**: Dados enviados (usado em POST, PUT, PATCH). Exemplo em JSON: `json { "titulo": "Nova tarefa", "concluida": false, "prioridade": "alta" }` Exemplo em XML: `xml <tarefa> <titulo>Nova tarefa</titulo> <concluida>false</concluida> <prioridade>alta</prioridade> </tarefa>` 5. **Parâmetros**: - **Query Parameters**: Filtros ou opções na URL (ex.: `/tarefas?concluida=true&limit=10`). - **Path Parameters**: Parte da URI (ex.: `/tarefas/1`).

Exemplo de Requisição

- **POST** para criar uma tarefa: ```http POST /tarefas HTTP/1.1 Host: api.exemplo.com Content-Type: application/json Accept: application/json Authorization: Bearer token123

{ “titulo”: “Estudar REST”, “concluida”: false, “prioridade”: “alta” } - `**Teste com cURL**:`bash curl -X POST http://localhost:3000/tarefas \ -H “Content-Type: application/json” \ -H “Authorization: Bearer token123” \ -d ‘{“titulo”:”Estudar REST”,”concluida”:false,”prioridade”:”alta”}’ ```

Estrutura de uma Resposta

Uma resposta HTTP inclui: 1. **Código de Status**: - `200 OK`: Sucesso (GET, PUT). - `201 Created`: Recurso criado (POST). - `204 No Content`: Sucesso sem corpo (DELETE). - `400 Bad Request`: Dados inválidos. - `401 Unauthorized`: Falha na autenticação. - `404 Not Found`: Recurso não encontrado. - `500 Internal Server Error`: Erro no servidor. 2. **Cabeçalhos**: - `Content-Type`: Formato da resposta (ex.: `application/json`). - `Location`: URL do recurso criado (ex.: `/tarefas/123` após POST). - `Cache-Control`: Regras de cache (ex.: `max-age=3600`). 3. **Corpo**: Dados ou mensagens de erro. - Exemplo de sucesso (JSON): `json { "id": 1, "titulo": "Estudar REST", "concluida": false, "prioridade": "alta", "_links": { "self": "/tarefas/1",`

"all": "/tarefas" } } - Exemplo de erro: json { "erro": "Título é obrigatório" }

Exemplo Completo

- **Requisição**: http GET /tarefas/1 HTTP/1.1 Host: api.exemplo.com Accept: application/json
- **Resposta**: ```http HTTP/1.1 200 OK Content-Type: application/json Cache-Control: max-age=3600

{ “id”: 1, “titulo”: “Estudar REST”, “concluida”: false, “prioridade”: “alta”, “_links”: { “self”: “/tarefas/1”, “all”: “/tarefas” } } - **Teste com cURL**:bash curl -v http://localhost:3000/tarefas/1 ```

Suporte a XML

Embora JSON seja predominante, algumas APIs suportam XML. O cliente indica o formato desejado via Accept: - Requisição: http GET /tarefas/1 HTTP/1.1 Host: api.exemplo.com Accept: application/xml - Resposta: ```http HTTP/1.1 200 OK Content-Type: application/xml

1 Estudar REST false alta ```

Content Negotiation

APIs RESTful podem suportar múltiplos formatos (JSON, XML) usando **content negotiation**: - O cliente especifica o formato desejado no cabeçalho Accept. - O servidor responde com o formato indicado no Content-Type. - Exemplo: http GET /tarefas HTTP/1.1 Accept: application/json, application/xml;q=0.9 O servidor prioriza JSON, mas pode retornar XML se JSON não estiver disponível.

4.3 Boas Práticas para Modelagem de Dados

A modelagem de dados é essencial para criar APIs REST claras, eficientes e fáceis de usar. Boas práticas garantem que os payloads sejam consistentes, escaláveis e compatíveis com os princípios REST.

1. Nomes Claros e Consistentes

- Use nomes descritivos para campos (ex.: `titulo` em vez de `txt`).
- Adote uma convenção de nomenclatura (ex.: camelCase para JSON, snake_case para Python).
- Exemplo: `json { "taskId": 1, "taskTitle": "Estudar REST", "isCompleted": false, "priorityLevel": "high" }`

2. Estrutura Hierárquica

- Use objetos aninhados para representar relações (ex.: um usuário com endereço).
- Exemplo: `json { "id": 1, "name": "Ana", "address": { "street": "Rua Principal", "city": "São Paulo", "country": "Brasil" } }`
- **XML Equivalente**: `xml <user> <id>1</id> <name>Ana</name> <address> <street>Rua Principal</street> <city>São Paulo</city> <country>Brasil</country> </address> </user>`

3. Evite Dados Redundantes

- Não inclua informações duplicadas ou desnecessárias.
- Exemplo (evitar): `json { "id": 1, "titulo": "Estudar", "tituloDuplicado": "Estudar", "status": false, "isCompleted": false }`

4. Suporte a Paginação e Filtros

- Para coleções grandes, inclua metadados de paginação (`total`, `page`, `limit`).
- Exemplo: `json { "tarefas": [ { "id": 1, "titulo": "Estudar" }, { "id": 2, "titulo": "Trabalhar" } ], "meta": { "total": 100, "page": 1, "limit": 10, "next": "/tarefas?page=2", "previous": null } }`

5. Implemente HATEOAS

- Inclua links para recursos relacionados, seguindo o princípio REST de interface uniforme.
- Exemplo: `json { "id": 1, "titulo": "Estudar REST", "_links": { "self": "/tarefas/1", "delete": "/tarefas/1", "update": "/tarefas/1", "all": "/tarefas" } }`

6. Validação no Servidor

- Valide os dados recebidos em POST e PUT para garantir consistência.
- Exemplo de validação com Joi (JavaScript): `javascript const Joi = require('joi'); const schema = Joi.object({ titulo: Joi.string().min(3).required(), concluida: Joi.boolean().required(), prioridade: Joi.string().valid('alta', 'média', 'baixa') });`

7. Formatos Padronizados

- **Datas**: Use ISO 8601 (ex.: `"2025-05-26T00:17:00Z"`).
- **Números**: Evite ambiguidades (ex.: `10.0` para decimais).
- **Enums**: Defina valores fixos (ex.: `"prioridade": "alta"`).
- Exemplo: `json { "id": 1, "createdAt": "2025-05-26T00:17:00Z", "price": 29.99, "status": "active" }`

8. Tratamento de Erros

- Forneça mensagens de erro claras, com códigos de status apropriados.
- Exemplo: `json { "erro": "Título deve ter pelo menos 3 caracteres", "campo": "titulo" }`

9. Versionamento

- Inclua a versão da API na URI ou cabeçalhos para suportar mudanças futuras.
- Exemplo: `/v1/tarefas` ou `Accept: application/vnd.exemplo.v1+json`.

10. Suporte a Múltiplos Idiomas

- Para APIs internacionais, inclua campos para tradução ou use cabeçalhos como `Accept-Language`.
- Exemplo: `json { "id": 1, "titulo": { "pt": "Estudar REST", "en": "Study REST" } }`

4.4 Atividade Prática: Criação de Payloads JSON para Requisições

Nesta atividade, você criará payloads JSON para uma API REST fictícia de gerenciamento de tarefas, aplicando as boas práticas de modelagem. Usaremos ferramentas open-source (cURL, REST Client para VS Code, Postman gratuito) para simular requisições e analisar respostas.

Objetivo

Criar payloads JSON para os seguintes cenários: 1. Criar uma nova tarefa (POST). 2. Atualizar uma tarefa existente (PUT). 3. Atualizar parcialmente uma tarefa (PATCH). 4. Filtrar tarefas por prioridade (GET com query parameter). 5. Analisar respostas de erro.

Cenários e Payloads

1. **POST: Criar uma Nova Tarefa**:
2. **Endpoint**: `POST /v1/tarefas`
3. **Payload JSON**: `json { "titulo": "Preparar apresentação", "concluida": false, "prioridade": "alta", "descricao": "Preparar slides para reunião de equipe", "criadoEm": "2025-05-26T00:17:00Z", "tags": ["trabalho", "apresentação"] }`

   Boas Práticas:
   - Nomes claros (camelCase).
   - Data em ISO 8601.
   - Campo `descricao` limitado a 200 caracteres (validação no servidor).
   - Array `tags` para metadados adicionais.
5. **Teste com cURL**: `bash curl -X POST http://localhost:3000/v1/tarefas \ -H "Content-Type: application/json" \ -d '{"titulo":"Preparar apresentação","concluida":false,"prioridade":"alta","descricao":"Prepa slides para reunião de equipe","criadoEm":"2025-05-26T00:17:00Z","tags":["trabalho","apresenta`

   Resposta Esperada: `json { "id": 1, "titulo": "Preparar apresentação", "concluida": false, "prioridade": "alta", "descricao": "Preparar slides para reunião de equipe", "criadoEm": "2025-05-26T00:17:00Z", "tags": ["trabalho", "apresentação"], "_links": { "self": "/v1/tarefas/1", "all":`

`"/v1/tarefas" } }`

PUT: Atualizar uma Tarefa:

8. **Endpoint**: `PUT /v1/tarefas/1`
9. **Payload JSON**: `json { "titulo": "Preparar apresentação final", "concluida": true, "prioridade": "média", "descricao": "Slides atualizados com feedback da equipe", "atualizadoEm": "2025-05-26T01:00:00Z", "tags": ["trabalho", "apresentação", "final"] }`

   Boas Práticas:
   - Substituição completa do recurso (PUT).
   - Consistência nos campos.
   - Data de atualização em ISO 8601.

   Teste com cURL: `bash curl -X PUT http://localhost:3000/v1/tarefas/1 \ -H "Content-Type: application/json" \ -d '{"titulo":"Preparar apresentação final","concluida":true,"prioridade":"média","descricao":"Slides atualizados com feedback da equipe","atualizadoEm":"2025-05-26T01:00:00Z","tags":["trabalho","apres`

   PATCH: Atualizar Parcialmente uma Tarefa:

13. **Endpoint**: `PATCH /v1/tarefas/1`
14. **Payload JSON**: `json { "concluida": true }`

    Boas Práticas:
    - Atualização parcial (apenas o campo `concluida`).
    - Payload mínimo, evitando dados desnecessários.

    Teste com cURL: `bash curl -X PATCH http://localhost:3000/v1/tarefas/1 \ -H "Content-Type: application/json" \ -d '{"concluida":true}'`

    GET: Filtrar Tarefas por Prioridade:

18. **Endpoint**: `GET /v1/tarefas?prioridade=alta`
19. **Payload**: Não aplicável (GET usa query parameters).

20. **Resposta Esperada**: `json { "tarefas": [ { "id": 1, "titulo": "Preparar apresentação", "concluida": false, "prioridade": "alta", "descricao": "Preparar slides para reunião de equipe", "_links": { "self": "/v1/tarefas/1" } } ], "meta": { "total": 1, "page": 1, "limit": 10, "next": null, "previous": null } }`

    Boas Práticas:

    - Metadados de paginação.
    - HATEOAS com links.

    Teste com cURL: `bash curl http://localhost:3000/v1/tarefas?prioridade=alta`

    Análise de Resposta de Erro:

24. **Cenário**: POST com título inválido (vazio).
25. **Payload JSON**: `json { "titulo": "", "concluida": false }`
26. **Resposta Esperada**: `json { "erro": "Título deve ter pelo menos 3 caracteres", "campo": "titulo" }`

    Boas Práticas:

    - Mensagem de erro clara.
    - Indicação do campo problemático.

28. **Teste com cURL**: `bash curl -X POST http://localhost:3000/v1/tarefas \ -H "Content-Type: application/json" \ -d '{"titulo":"","concluida":false}'`

Tarefa Prática

1. Crie payloads JSON para os seguintes cenários:
2. Criar uma tarefa com um campo `prazo` (data ISO 8601).
3. Atualizar parcialmente uma tarefa (PATCH) para mudar `prioridade` e `tags`.
4. Listar tarefas criadas após uma data específica (`GET /v1/tarefas?criadoApos=2025-05-01T00:00:00Z`).
5. Simule as requisições usando cURL ou REST Client.
6. Documente cada cenário:
7. Payload JSON.
8. Resposta esperada.

9. Boas práticas aplicadas.
10. Exemplo de documentação:
11. **Cenário**: Criar tarefa com prazo.
12. **Payload JSON**: `json { "titulo": "Enviar relatório", "concluida": false, "prioridade": "baixa", "prazo": "2025-06-01T17:00:00Z", "tags": ["relatório", "trabalho"] }`
13. **Resposta Esperada**: `json { "id": 2, "titulo": "Enviar relatório", "concluida": false, "prioridade": "baixa", "prazo": "2025-06-01T17:00:00Z", "tags": ["relatório", "trabalho"], "_links": { "self": "/v1/tarefas/2", "all": "/v1/tarefas" } }`

    Boas Práticas:

    - Nomes claros (camelCase).
    - Data `prazo` em ISO 8601.
    - HATEOAS com links.

Dicas para a Atividade

- Use a extensão REST Client no VS Code para criar arquivos `.http` com requisições: ``` POST http://localhost:3000/v1/tarefas HTTP/1.1 Content-Type: application/json

{ “titulo”: “Enviar relatório”, “concluida”: false, “prioridade”: “baixa”, “prazo”: “2025-06-01T17:00:00Z”, “tags”: [“relatório”, “trabalho”] } `` - Teste erros intencionalmente (ex.: título com menos de 3 caracteres) para analisar respostas. - Explore a JSONPlaceholder (/todos`) para praticar payloads semelhantes.

Conclusão

Este capítulo aprofundou a manipulação de dados em APIs REST, cobrindo JSON e XML, a estrutura de requisições e respostas, e boas práticas de modelagem. A atividade prática consolidou esses conceitos, criando payloads JSON que seguem padrões RESTful. Este conhecimento é essencial para o Capítulo 5, onde começaremos a implementar uma API REST com Node.js e Express, aplicando os formatos e práticas aprendidos aqui.

Dicas para Continuar

- Converta um payload JSON para XML e compare o tamanho e a legibilidade.
- Experimente bibliotecas como **Joi** (JavaScript) ou **Pydantic** (Python) para validação de payloads.
- Teste APIs públicas (ex.: JSONPlaceholder, PokéAPI) para analisar a estrutura de seus payloads.

Conexão com o Próximo Capítulo

O Capítulo 5 iniciará o desenvolvimento prático de uma API REST, usando Node.js com Express para criar endpoints GET. Os conceitos de JSON, requisições/respostas e modelagem de dados deste capítulo serão diretamente aplicados na implementação.