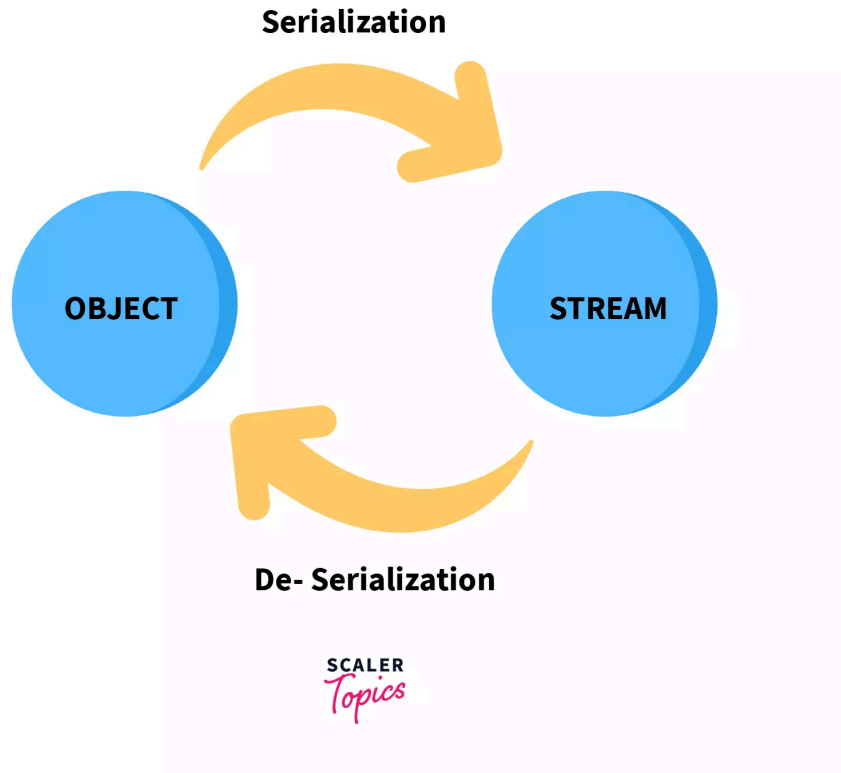


TÓPICO 12 - COMUNICAÇÃO DE DADOS

Disciplina de Backend - Professor Ramon Venson - SATC 2026.2

Por que formatos de troca de dados?



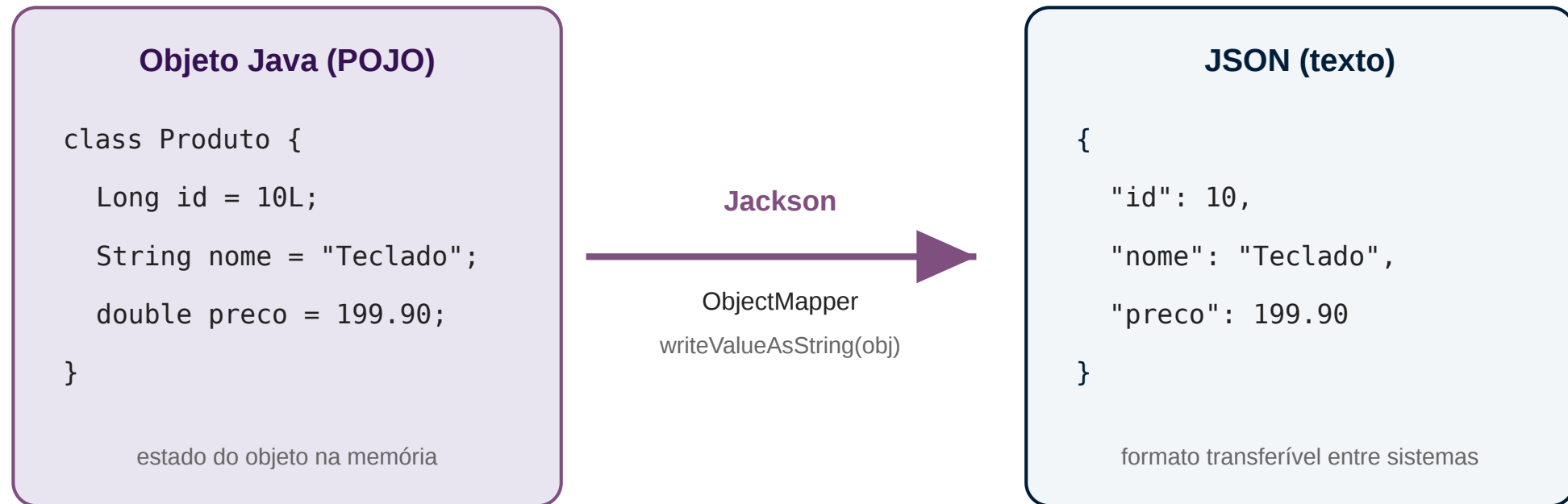
- Objetos na memória são incompatíveis entre aplicações e linguagens;
- É preciso um formato que possa ser reconhecido por diferentes plataformas;
- O processo de conversão deve ter custo computacional aceitável.

Dois lados da mesma moeda: **serialização** (objeto → texto/bytes) e **desserialização** (texto/bytes → objeto).

Serialização e Desserialização

- **Serialização:** converte um objeto em memória para um formato armazenável ou transferível;
- **Desserialização:** reconstrói um objeto a partir de um dado serializado;
- O formato escolhido define como os tipos são representados.

Serialização: objeto em memória → JSON



Formatos Estruturados

Existem diversos formatos para troca de dados, independentes de linguagem e plataforma:

- **JSON** — leve e dominante em APIs modernas;
- **XML** — flexível, comum em sistemas corporativos/SOAP;
- **YAML** — muito usado em configurações;
- **CSV** — dados tabulares separados por vírgula;
- **HTML** — dados estruturados para a Web;
- **GeoJSON** — dados geoespaciais;
- **MessagePack** — formato binário compacto.

Comparação entre formatos

Formato	Tipo	Verbosidade	Uso típico
JSON	Texto	Baixa	APIs REST, mobile
XML	Texto	Alta	SOAP, integrações corporativas
YAML	Texto	Baixa	Configuração
CSV	Texto	Muito baixa	Planilhas, exportações
MessagePack	Binário	Muito baixa	Alta performance

JSON

JSON (*JavaScript Object Notation*) é um formato de troca de dados em modo texto, derivado da notação de objetos do JavaScript.

É amplamente usado em APIs Web por ser simples, leve e de fácil parsing.

```
// example of data.json file
[
  {
    "id": 1,
    "firstName": "Thomas",
    "lastName": "Bækker",
    "address": {
      "streetName": "Tornelisevej",
      "streetNumber": "10",
      "city": "Solrød Strand"
    },
    "emergencyPhoneNumbers": ["+4530303030", "+4510203040"]
  },
  {
    "id": 2,
    "firstName": "Niels",
    "lastName": "Bohr",
    "address": {
      "streetName": "Flemsevej",
      "streetNumber": "31A",
      "city": "København"
    },
    "emergencyPhoneNumbers": []
  },
  ...
]
```

Características do JSON

- Formato para troca de dados;
- Independente de linguagem de programação;
- Sintaxe simples e textual;
- Fácil de ler por humanos e por máquinas.

Exemplo JSON

```
{
  "usuarios": {
    "john": {
      "email": "john@matrix.com",
      "senha": "12345"
    },
    "mary": {
      "email": "m.mary@bol.com.br",
      "senha": "abc123",
      "admin": true
    }
  }
}
```

Sintaxe JSON

O formato JSON segue as seguintes regras:

- Os dados são encapsulados em pares (`chave: valor`);
- Os pares são separados por `,` (vírgula);
- Um objeto é encapsulado por `{ }` (chaves);
- Vetores são encapsulados por `[ ]` (colchetes);
- As chaves são sempre circundadas por `" "` (aspas duplas).

Um objeto vazio é representado por:

```
{ }
```

Um par chave/valor pode ser descrito como:

```
{ "usuarios" : "john e mary" }
```

Tipos de Dados JSON

O JSON suporta os seguintes tipos:

- Strings (ex.: `"cem"`);
- Números (ex.: `100` , `100.0` , `1.0E+2`);
- Objeto (ex.: `{ "cem" : 100 }`);
- Vetor (ex.: `[100, 100.1, "cem"]`);
- Booleano (ex.: `true` ou `false`);
- Nulo (ex.: `null`).

Valores que **não** existem como tipo no JSON: funções, datas e `void` .

XML

XML (*eXtensible Markup Language*) representa dados usando **tags** hierárquicas.

- Suporta **atributos** (`id="10"`) e **namespaces** (`xmlns`);
- Muito usado em integrações corporativas e no protocolo SOAP.



Exemplo XML

```
<?xml version="1.0" encoding="UTF-8"?>
<produto id="10">
  <nome>Teclado</nome>
  <preco>199.90</preco>
  <categorias>
    <categoria>Periféricos</categoria>
    <categoria>Acessórios</categoria>
  </categorias>
</produto>
```

Bem-formado x Válido

- **Bem-formado**: segue a sintaxe XML (tags fechadas, aninhamento correto);
- **Válido**: além de bem-formado, respeita um esquema (**XSD/Schema**);
- A validação por esquema garante contrato e tipos esperados.

Mesmos dados: JSON x XML

JSON

```
{  
  "id": 10,  
  "nome": "Teclado",  
  "preco": 199.90,  
  "categorias": [  
    "Periféricos",  
    "Acessórios"  
  ]  
}
```

- Mais leve e menos verboso
- Mapeia direto para objetos em JS
- Tipos nativos (número, booleano, null)
- Padrão dominante em APIs REST

XML

```
<produto id="10">  
  <nome>Teclado</nome>  
  <preco>199.90</preco>  
  <categorias>  
    <categoria>Periféricos</categoria>  
    <categoria>Acessórios</categoria>  
  </categorias>  
</produto>
```

- Mais verboso (tags de abertura/fechamento)
- Suporta atributos, namespaces e XSD
- Contratos formais e validação de esquema
- Base histórica do SOAP

Object Mapping

Object Mapping é a conversão entre **representações textuais** (JSON/XML) e **objetos** da aplicação.

- O objeto mapeado costuma ser um **POJO** (*Plain Old Java Object*);
- Cada campo do objeto corresponde a uma chave no JSON;
- Anotações ajustam nomes, formatos e campos ignorados.

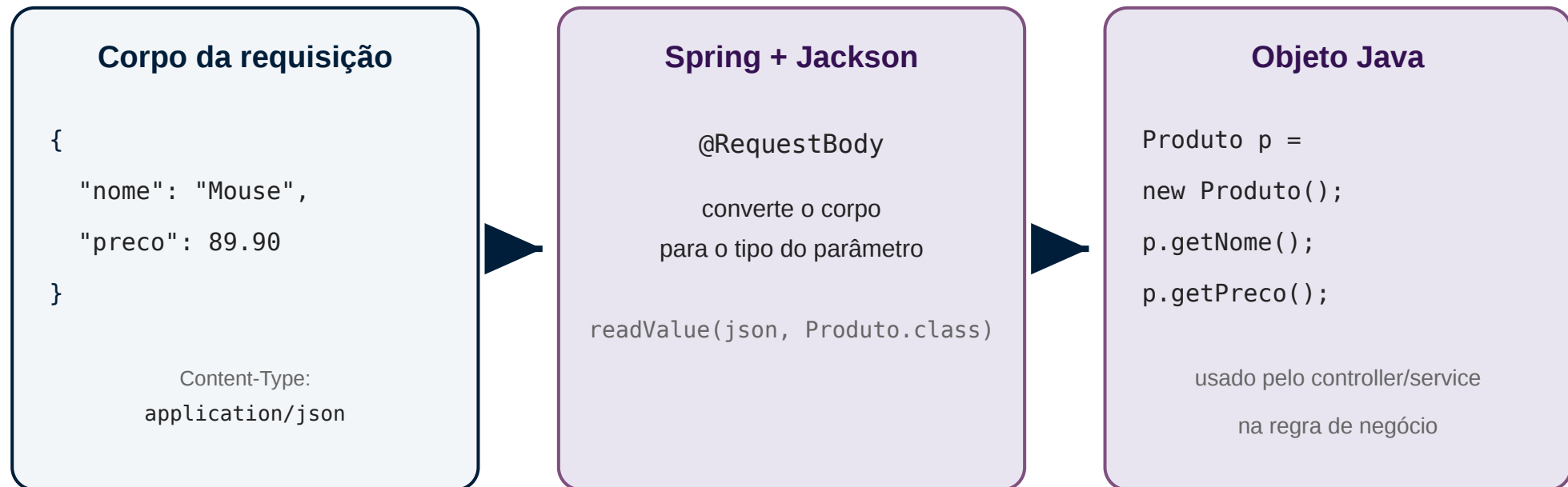
```
public class Produto {  
    private UUID id;  
    private String nome;  
    private double preco;  
    // getters e setters...  
}
```

Mapeando nomes com anotações

```
public class Produto {  
    @JsonProperty("product_name")  
    private String nome;  
  
    @JsonIgnore  
    private String senhaInterna;  
  
    @JsonFormat(pattern = "yyyy-MM-dd")  
    private LocalDate dataCriacao;  
  
    @JsonInclude(JsonInclude.Include.NON_NULL)  
    private Boolean promocao;  
}
```

- `@JsonProperty` : define o nome usado no JSON;
- `@JsonIgnore` : omite o campo na serialização;
- `@JsonFormat` : controla formatos (datas, números);

Desserialização: JSON da requisição → objeto Java



Jackson

O **Jackson** é a biblioteca de serialização/desserialização usada por padrão no Spring Boot.

- `ObjectMapper` é a classe central;
- `writeValueAsString(obj) : objeto → JSON`;
- `readValue(json, Classe.class) : JSON → objeto`.

```
ObjectMapper mapper = new ObjectMapper();  
  
String json = mapper.writeValueAsString(produto);  
Produto p = mapper.readValue(json, Produto.class);
```

Configurações comuns do Jackson

- **Formato de datas:** define como as datas são (de)serializadas;
- **Inclusão/omissão de `null`:** evita enviar campos nulos no JSON;
- **Estratégia de nomes** (`snake_case` , `camelCase`): padroniza nomes de campos;
- **Falha (ou não) em campos desconhecidos:** controla o comportamento quando campos não esperados são encontrados.

```
spring.jackson.serialization.indent-output=true  
spring.jackson.default-property-inclusion=non_null
```

Recebendo JSON no Spring (`@RequestBody`)

Imagine receber usuário e senha para login, enviados pelo cliente em JSON:

```
{ "usuario": "ronaldinho", "senha": "bruxo123" }
```

Para receber e processar esses dados em um *controller*, usamos `@RequestBody` com um parâmetro cujo tipo tem os mesmos atributos:

```
public class LoginController {
    @PostMapping("/login")
    public ResponseEntity<Object> login(@RequestBody Login login) {
        Boolean ok = login.getUsuario().equals("ronaldinho")
            && login.getSenha().equals("bruxo123");

        if (ok) {
            return ResponseEntity.ok("Login realizado com sucesso");
        }
        return ResponseEntity.status(401).body("Senha ou usuário incorreto");
    }
}
```

A classe usada no mapeamento é um POJO:

```
public class Login {  
    private String usuario;  
    private String senha;  
  
    // construtores, getters e setters...  
}
```

O Spring então faz o mapeamento automático do corpo da requisição para o objeto (via Jackson).

Respondendo JSON no Spring

Com `@RestController` o Spring serializa o objeto retornado para JSON:

```
@RestController
public class ProdutoController {
    @GetMapping("/produto")
    public Produto gerar() {
        return new Produto(UUID.randomUUID(), "Teclado", 199.90);
    }
}
```

Resposta:

```
HTTP/1.1 200 OK
```

```
Content-Type: application/json
```

```
{ "id": "3fa85f64-5717-4562-b3fc-2c963f66afa6", "nome": "Teclado", "preco": 199.9 }
```

Múltiplas representações

O mesmo recurso pode ser devolvido em formatos diferentes, conforme o `Accept` do cliente.

```
@GetMapping(value = "/produto",  
            produces = { MediaType.APPLICATION_JSON_VALUE,  
                        MediaType.APPLICATION_XML_VALUE })  
public Produto detalhar() {  
    return new Produto(UUID.randomUUID(), "Teclado", 199.90);  
}
```

- Para responder XML no Spring, adicione a dependência `jackson-dataformat-xml`;
- `produces` define os formatos que o endpoint pode gerar.

Dependência para XML

```
<dependency>  
  <groupId>com.fasterxml.jackson.dataformat</groupId>  
  <artifactId>jackson-dataformat-xml</artifactId>  
</dependency>
```

Com ela, o mesmo objeto é serializado para JSON ou XML conforme o cabeçalho

Accept .

Consumindo XML (POST)

Com a mesma dependência, o `@RequestBody` também desserializa XML quando o `Content-Type` da requisição é `application/xml`:

```
@PostMapping(consumes = { MediaType.APPLICATION_JSON_VALUE,  
                        MediaType.APPLICATION_XML_VALUE })  
public Produto criar(@RequestBody Produto produto) {  
    return produtoRepository.save(produto);  
}
```

```
POST /produtos HTTP/1.1  
Content-Type: application/xml
```

```
<Produto>  
  <nome>Mouse</nome>  
  <preco>89.90</preco>  
</Produto>
```

O que aprendemos hoje

- O que é serialização e desserialização;
- Os principais formatos de troca de dados (JSON, XML, YAML, CSV);
- Como representar e interpretar dados em JSON e XML;
- O que é Object Mapping e o papel das anotações;
- Como o Jackson integra Java e JSON no Spring Boot;
- Como devolver diferentes representações de um recurso;
- A diferença entre **cache HTTP** (cabeçalhos) e **cache de aplicação** (ex.: Spring Cache).

Materiais de apoio

- JSON — <https://www.json.org/json-pt.html>
- Jackson Project — <https://github.com/FasterXML/jackson>
- Spring Boot + Jackson — <https://docs.spring.io/spring-boot/how-to/webserver.html>
- XML (W3C) — <https://www.w3.org/XML/>