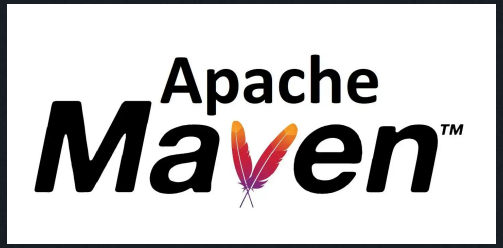
Gerenciamento de dependências e builds

Programação Web II

Gerenciamento de Dependências com Maven/Gradle

Maven/**Gradle** é uma ferramenta de gerenciamento de projetos e automação de builds. Ela facilita a inclusão de bibliotecas externas e o controle de versões.



Manipulação de dados com Jackson

Jackson é uma biblioteca popular para converter objetos Java em JSON e vice-versa.

A classe *ObjectMapper* é a responsável por fazer essa conversão de maneira simples, como no exemplo a seguir.

```java
import com.fasterxml.jackson.databind.ObjectMapper;

public class ExemploJackson {
    public static void main(String[] args) {
        try {
            ObjectMapper mapper = new ObjectMapper();
            Produto produto = new Produto("Notebook", 2500.0);
            String json = mapper.writeValueAsString(produto);
            System.out.println(json);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Links para consulta: baeldung, jackson-annotations, jackson

Manipulação de dados com Jackson

Programação Web II

Conversão de JSON para objeto:

```
String json = "{\"nome\":\"Notebook\",\"preco\":2500.0}";
Produto produto = mapper.readValue(json, Produto.class);
```

Anotações Jackson para manipulação de JSON

Jackson fornece diversas anotações para personalizar a maneira como os campos e classes são serializados e desserializados. Aqui estão algumas das principais:

@JsonProperty

define o nome que será usado no JSON para um campo específico. É útil quando o nome do campo na classe Java difere daquele usado no JSON.

```
import com.fasterxml.jackson.annotation.JsonProperty;

public class Produto {

    @JsonProperty("nome_produto")
    private String nome;

    @JsonProperty("preco_produto")
    private double preco;

    // Construtor, getters e setters
}
```

Anotações Jackson para manipulação de JSON

Programação Web II

@JsonIgnore

Ignora um campo durante a serialização e desserialização.

Isso é útil para excluir informações sensíveis ou desnecessárias no JSON.

```java
import com.fasterxml.jackson.annotation.JsonIgnore;

public class Produto {

    private String nome;
    private double preco;

    @JsonIgnore
    private String codigoInterno;

    // Construtor, getters e setters
}
```

Anotações Jackson para manipulação de JSON

Programação Web II

@JsonFormat

Usado para especificar o formato de datas ou outros tipos de valores durante a serialização e desserialização.

Por exemplo, formatar uma data para o padrão dd/MM/yyyy.

```java
import com.fasterxml.jackson.annotation.JsonFormat;
import java.util.Date;

public class Produto {

    private String nome;
    private double preco;

    @JsonFormat(shape = JsonFormat.Shape.STRING, pattern = "dd/MM/yyyy")
    private Date dataLancamento;

    // Construtor, getters e setters
}
```

Anotações Jackson para manipulação de JSON

Programação Web II

@JsonCreator

Permite configurar um construtor ou método estático para ser usado na desserialização de um objeto.

É especialmente útil quando a classe não possui um construtor padrão.

```java
import com.fasterxml.jackson.annotation.JsonCreator;
import com.fasterxml.jackson.annotation.JsonProperty;

public class Produto {

    private String nome;
    private double preco;

    @JsonCreator
    public Produto(@JsonProperty("nome") String nome, @JsonProperty("preco") double preco) {
        this.nome = nome;
        this.preco = preco;
    }

    // Getters e setters
}
```

Lombok

Redução de Código com Lombok

Programação Web II

Lombok é uma biblioteca que reduz a necessidade de escrever métodos repetitivos como *getters*, *setters* e *toString*.

Configuração do Lombok:

- Maven **pom.xml**:

```xml
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.18.20</version>
    <scope>provided</scope>
</dependency>
```

- Gradle **build.gradle**:

```
dependencies {
   compileOnly("org.projectlombok:lombok:1.18.42" )
   annotationProcessor ("org.projectlombok:lombok:1.18.42" )
}
```

Redução de Código com Lombok

Programação Web II

Exemplo com Lombok

```java
import lombok.Data;

@Data
public class Produto {
    private String nome;
    private double preco;

    public Produto(String nome, double preco) {
        this.nome = nome;
        this.preco = preco;
    }
}
```

Consultar documentação para mais detalhes sobre anotação @Data e a documentação completa.

Exercícios

Exercício 1: Desenvolvimento de uma API simples

- Crie uma aplicação que consome uma API pública, usando **Maven ou Gradle** como gerenciador de dependências.
- Converta a resposta JSON em objetos Java usando Jackson.
- Use **Lombok** para definir classes de modelo com menos código.

Exercícios

Gerenciamento de dependências e builds

Exercício 1.1: Desenvolvimento de uma API simples (Desafio)

Crie uma aplicação que:

1. Consome uma API pública de produtos.
2. Converte a resposta em objetos Java.
3. Organiza os objetos em uma lista e exibe as informações formatadas.

Exemplo:

```
import java.net.HttpURLConnection;
import java.net.URL;
import java.util.Scanner;
import com.fasterxml.jackson.databind.ObjectMapper;

public class ApiProdutos {
    public static void main(String[] args) {
        try {
            URL url = new URL("https://dummyjson.com/products");
            HttpURLConnection conn = (HttpURLConnection) url.openConnection();
            conn.setRequestMethod("GET");

            Scanner scanner = new Scanner(url.openStream());
            StringBuilder json = new StringBuilder();
            while (scanner.hasNext()) {
                json.append(scanner.nextLine());
            }
            scanner.close();

            ObjectMapper mapper = new ObjectMapper();
            Produto[] produtos = mapper.readValue(json.toString(), Produto[].class);
            for (Produto p : produtos) {
                System.out.println(p);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Exercício 2: Console App “Catálogo & Tarefas” (DummyJSON)

Contexto
Você foi contratada para criar um utilitário de linha de comando que ajude um time a consultar um catálogo de produtos e gerenciar uma lista de tarefas (todos). A aplicação deve consumir a API pública **DummyJSON** e exibir as informações no console.

Objetivo Geral

Construir uma aplicação Java (sem frameworks) que:

1. modele classes simples do domínio,
2. consuma endpoints REST da DummyJSON para consultas e ações (CRUD),
3. forneça um menu interativo no console.

Tecnologias e Restrições

- Java 17+.
- Sem Spring / frameworks web.
- Jackson para (de)serialização JSON.
- Lombok para reduzir boilerplate (getters/setters, construtores, toString, etc.).
- Cliente HTTP com HttpURLConnection (ou java.net.http.HttpClient se preferir).
- Entrada/saída via console (Scanner).

Exercício 2: Console App “Catálogo & Tarefas” (DummyJSON)

Endpoints da DummyJSON a utilizar

1. **Produtos**
   a. Listagem com paginação: GET /products?limit={n}&skip={k}
   b. Busca por texto: GET /products/search?q={termo}

2. **Todos**
   a. Listar: GET /todos?limit={n}&skip={k}
   b. Adicionar: POST /todos/add
   c. Atualizar (marcar/desmarcar): PUT /todos/{id}
   d. Remover: DELETE /todos/{id}

Funcionalidades Obrigatórias

1) Menu de Console

Exibir um menu em loop com as opções:

1. Listar produtos (solicitar limit e skip)
2. Buscar produtos por texto (q)
3. Listar todos (solicitar limit e skip)
4. Adicionar todo (texto + userId)
5. Marcar/Desmarcar todo (id + completed=true/false)
6. Remover todo (id)
7. Sair

Exercício 2: Console App “Catálogo & Tarefas” (DummyJSON)

Funcionalidades Obrigatórias

2) Modelagem com Lombok

Exibir um menu em loop com as opções:

1. Criar as classes (mínimo):
2. Product { id, title, price }
3. Todo { id, todo, completed, userId }
4. (Opcional) User { id, firstName, lastName }

Use **@Data**, **@NoArgsConstructor** e **@AllArgsConstructor** (ou equivalentes).

3) DTOs de Resposta (Wrappers)

DTOs para mapear respostas:

1. ProductListResponse { List<Product> products, int total, int skip, int limit }
2. TodoListResponse { List<Todo> todos, int total, int skip, int limit }

4) Cliente HTTP

Implementar utilitário para GET, POST, PUT, DELETE:

1. Enviar/receber JSON.
2. Tratar status HTTP: se não for 2xx, exibir mensagem de erro amigável.

Exercício 2: Console App “Catálogo & Tarefas” (DummyJSON)

Funcionalidades Obrigatórias

5) Serviço de Domínio

1. **ProductService**
   a. **list(limit, skip)** → retorna List<Product>
   b. **search(q)** → retorna List<Product>

2. **TodoService**
   a. **list(limit, skip)** → retorna List<Todo>
   b. **add(text, userId)** → retorna Todo criado
   c. **toggle(id, completed)** → retorna Todo atualizado
   d. **delete(id)** → retorna boolean (sucesso)

6) Apresentação no Console

- **Produtos:** imprimir linhas no formato id | title | price.
- **Todos:** id | todo | (OK/PEND) | user={userId}.
- Mensagens de sucesso/erro claras para cada ação.

Exercício 2: Console App “Catálogo & Tarefas” (DummyJSON)

Requisitos de Qualidade

- **Tratamento de erros:** IO, JSON inválido, status HTTP ≠ 2xx, entradas vazias.
- **Código legível:** nomes claros, toString() útil nas entidades, mensagens ao usuário.
- **Build:** fornecer build.gradle ou pom.xml com Jackson e Lombok.

Obrigada