**ZAIRO LINS RIBEIRO CUNHA**

**TÓPICOS EM PROGRAMAÇÃO I**

Estudo sobre as ferramentas e tecnologias usadas durante a matéria de Tópicos em Programação I

**MAVEN**

O **Maven** é uma ferramenta de automação de build e gerenciamento de dependências para projetos Java. Ele simplifica a compilação, empacotamento, teste e implantação de aplicações, garantindo um fluxo de desenvolvimento mais organizado e eficiente.

* **Principais funções do Maven**

1. **Gerenciamento de Dependências**
   * O Maven permite adicionar bibliotecas externas ao projeto sem precisar baixá-las manualmente. Basta declarar as dependências no arquivo pom.xml, e ele baixa e gerencia automaticamente as versões corretas.
2. **Automação de Build**
   * Ele compila o código-fonte, executa testes, empacota a aplicação (JAR/WAR), gera documentação e até realiza o deploy automaticamente.
3. **Padronização da Estrutura do Projeto**
   * Utiliza uma estrutura de diretórios padronizada, facilitando a organização e entendimento do código.
4. **Gerenciamento de Plugins**
   * O Maven usa plugins para estender funcionalidades, como geração de código, execução de testes, análise estática e muito mais.

* **Como ele funciona?**

1. O Maven utiliza um arquivo chamado pom.xml (**Project Object Model**), onde são definidas as configurações do projeto, dependências e plugins.
2. Quando você executa um comando (mvn clean install, por exemplo), ele segue um **Ciclo de Vida do Build**, que inclui fases como validação, compilação, teste e empacotamento.

* **Exemplo de dependência no pom.xml**

Se você quiser adicionar o **Hibernate** ao seu projeto, basta incluir isso no pom.xml:

<dependencies>

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-core</artifactId>

<version>6.3.1.Final</version>

</dependency>

</dependencies>

Depois, ao rodar mvn install, o Maven baixa essa biblioteca automaticamente.

**Comandos básicos do Maven**

* mvn clean → Limpa os arquivos gerados pelo build anterior
* mvn compile → Compila o código-fonte
* mvn test → Executa os testes automatizados
* mvn package → Gera um JAR ou WAR do projeto
* mvn install → Instala o pacote gerado no repositório local
* mvn dependency:tree → Lista as dependências do projeto

**QUARKUS**

O **Quarkus** é um framework Java otimizado para rodar em ambientes nativos, como **containers e serverless**, trazendo alta performance e baixo consumo de memória. Ele é focado principalmente em **microservices** e **cloud-native applications**.

**Por que usar o Quarkus?**

1. **Baixo Consumo de Memória**
   * Usa menos RAM em comparação com frameworks tradicionais como Spring Boot.
2. **Startup Rápido**
   * O tempo de inicialização é muito menor, tornando-o ideal para aplicações **serverless** e **containers**.
3. **Integração com GraalVM e Kubernetes**
   * Suporta **compilação nativa** com **GraalVM**, gerando binários executáveis que dispensam o uso da JVM em tempo de execução.
   * Possui suporte nativo para **Kubernetes** e **OpenShift**, facilitando a implantação em nuvem.
4. **Suporte a JPA/Hibernate e REST fácil de configurar**
   * Quarkus tem integração com **JPA/Hibernate** para banco de dados e usa **Jakarta REST (JAX-RS)** para criar APIs RESTful rapidamente.

**Como funciona?**

* Ele usa **extensões** para adicionar funcionalidades ao projeto (banco de dados, REST, segurança, etc.).
* Seu código pode rodar tanto na **JVM** quanto em **modo nativo** (compilado com GraalVM).

**Exemplo prático (API REST com Quarkus + JPA)**

**1️. Criando uma entidade e um endpoint REST**

import jakarta.persistence.Entity;

import jakarta.persistence.GeneratedValue;

import jakarta.persistence.Id;

import jakarta.ws.rs.\*;

import jakarta.ws.rs.core.MediaType;

import java.util.List;

@Entity

class Produto {

@Id

@GeneratedValue

public Long id;

public String nome;

public double preco;

}

@Path("/produtos")

@Produces(MediaType.APPLICATION\_JSON)

@Consumes(MediaType.APPLICATION\_JSON)

public class ProdutoResource {

@GET

public List<Produto> listar() {

return Produto.listAll();

}

@POST

public void adicionar(Produto produto) {

produto.persist();

}

}

**Comandos básicos do Quarkus**

* mvn quarkus:dev → Executa a aplicação em **modo dev** com hot reload
* mvn package → Empacota a aplicação
* mvn package -Pnative → Gera um binário nativo com **GraalVM**
* mvn quarkus:add-extension -Dextensions="hibernate-orm" → Adiciona extensões ao projeto

**Quarkus vs Spring Boot**

| **Recurso** | **Quarkus** | **Spring Boot ☕** |
| --- | --- | --- |
| Consumo de RAM | Baixo | Médio/Alto |
| Tempo de inicialização | Rápido | Lento |
| Suporte para GraalVM | Sim | Parcial |
| Facilidade para microsserviços | Alta | Média |
| Ecossistema | Menor | Grande |

**Conclusão**

O **Quarkus** é uma excelente opção para quem quer **performance, baixo consumo de memória e rapidez**, especialmente em **microservices e aplicações cloud-native**. Se você já conhece **Java e JPA/Hibernate**, a curva de aprendizado é bem tranquila.

**API (Application Programming Interface)**

Uma **API** é um conjunto de regras que permite que **sistemas diferentes se comuniquem entre si**. No desenvolvimento web, usamos APIs para expor funcionalidades de um sistema de forma padronizada, permitindo que outros sistemas ou aplicações consumam esses serviços.

**Tipos de API mais comuns:**

1. **REST API (Representational State Transfer)** → Baseada em HTTP e geralmente retorna dados em **JSON**.
2. **SOAP API (Simple Object Access Protocol)** → Mais antiga, usa XML e é mais estruturada.
3. **GraphQL** → Permite que o cliente defina quais dados deseja receber.
4. **gRPC** → Usa **Protocol Buffers (Protobuf)** e é mais eficiente para comunicação entre microsserviços.

**Exemplo de API REST em Quarkus**

@Path("/mensagem")

@Produces(MediaType.APPLICATION\_JSON)

public class MensagemResource {

@GET

public String dizerOla() {

return "Olá, mundo!";

}

}

Se rodarmos essa API e acessarmos http://localhost:8080/mensagem, ela retornará:

"Olá, mundo!"

**DOCKER**

O **Docker** é uma plataforma que permite criar, executar e gerenciar **containers**, que são ambientes isolados para rodar aplicações.

**Vantagens do Docker:**

**Portabilidade** → Um container roda da mesma forma em qualquer sistema (Windows, Linux, Mac).  
**Leveza** → Containers compartilham o mesmo kernel do sistema operacional, consumindo menos recursos que máquinas virtuais.  
**Facilidade na implantação** → Um ambiente pronto pode ser criado e replicado rapidamente.

**Exemplo de Dockerfile para rodar um app Java**

FROM openjdk:17

COPY target/minha-api.jar app.jar

ENTRYPOINT ["java", "-jar", "app.jar"]

**Comandos básicos do Docker**

* docker build -t minha-api . → Cria a imagem do container
* docker run -p 8080:8080 minha-api → Roda o container na porta 8080
* docker ps → Lista os containers em execução
* docker stop <container\_id> → Para um container

**POSTGRESQL**

O **PostgreSQL** é um **banco de dados relacional (SQL)** **open-source** e robusto, muito usado por empresas devido à sua confiabilidade e suporte a operações complexas.

**Vantagens do PostgreSQL:**

Suporte a **transações ACID** (Atomicidade, Consistência, Isolamento e Durabilidade)  
**Consultas SQL avançadas** (JOINs, subconsultas, CTEs, etc.)  
**Extensibilidade** → Permite criar funções, extensões e até armazenar JSON

**Exemplo de tabela PostgreSQL**

CREATE TABLE clientes (

id SERIAL PRIMARY KEY,

nome VARCHAR(100),

email VARCHAR(100) UNIQUE,

saldo DECIMAL(10,2)

);

**Rodando PostgreSQL com Docker**

docker run --name meu-postgres -e POSTGRES\_PASSWORD=1234 -p 5432:5432 -d postgres

Agora podemos conectar no banco com:

psql -h localhost -U postgres