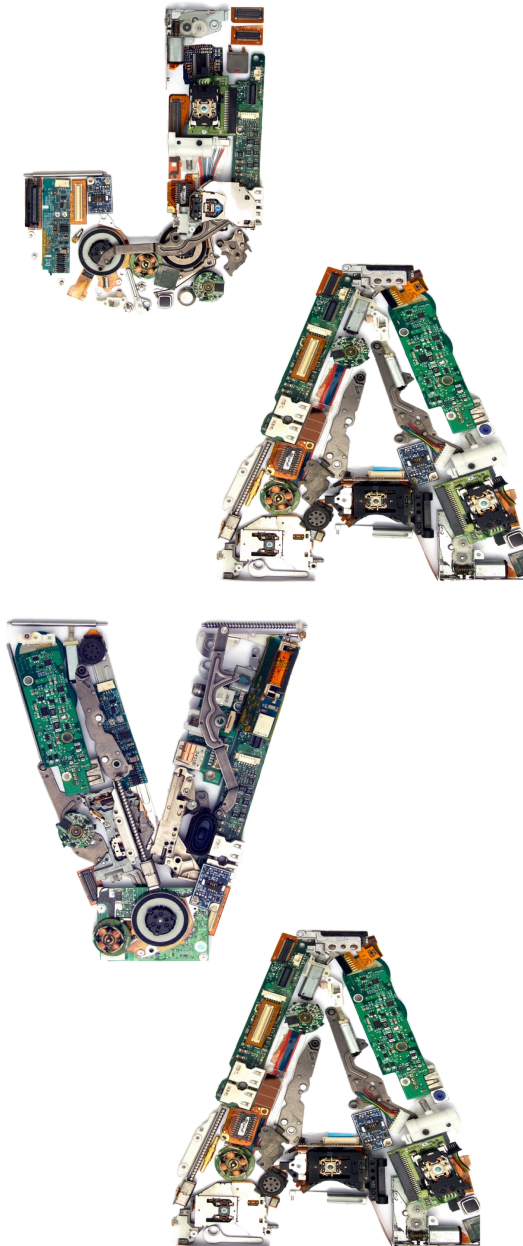


Programação Orientada a Objetos com Java e WEB

Introdução ao JPA



Exercício – Projeto Assistência Técnica

Uma assistência técnica recebe diariamente equipamentos (notebooks, celulares, tablets) para diagnóstico e reparo.

O atendimento precisa registrar o **cliente**, o **equipamento** trazido e abrir uma **Ordem de Serviço (OS)** com status que evoluem ao longo do processo (ABERTA → EM_ANDAMENTO → CONCLUIDA/CANCELADA).

O objetivo do projeto é construir, em **Spring Boot**, uma **API REST** conectada ao **Oracle** para gerenciar esse fluxo.

Exercício – Projeto Assistência Técnica

Objetivos

- ✓ Modelar um domínio com **3 tabelas** relacionadas.
- ✓ Integrar **Spring Boot + JPA/Hibernate** com **Oracle**.
- ✓ Implementar **CRUD** e **regras de negócio** com **transações**.
- ✓ Expor **endpoints REST** testáveis por Postman/Insomnia.

JPA – Java Persistence API

JPA (Java Persistence API) é uma **especificação** (um “contrato”) da plataforma Java para mapeamento objeto-relacional (ORM). Ela define **como** classes Java viram tabelas/colunas e **como** você consulta e persiste dados.

Linha do tempo (resumida):

- ✓ Anos 2000 (pré-JPA): Java EE usava **Entity Beans (EJB 2.x)**, pesados/verbosos.
- ✓ 2006 — JPA 1.0 (JSR 220): primeira versão (JSR do Java Community Process). Traz entidades POJO, JPQL e mapeamentos simples — uma resposta mais leve aos Entity Beans.
- ✓ 2009 — JPA 2.0 (JSR 317): Criteria API, mapeamentos avançados, cache 2º nível mais claro.
- ✓ 2013 — JPA 2.1 (JSR 338): Stored procedures, *@Converter*, *Entity Graphs*.
- ✓ 2017 — JPA 2.2: suporte a `java.time` (Java 8) e pequenos aprimoramentos.
- ✓ 2020+ — **Jakarta Persistence 3.x**: migração do **javax** para **jakarta** (Jakarta EE 9) e evolução contínua sob a Eclipse Foundation. Hoje o nome “oficial” é **Jakarta Persistence** (mas “JPA” segue sendo o termo de uso).
- ✓ **Ponto-chave:** JPA ~~não é implementação~~; é a interface/contrato.

Hibernate

Hibernate é a **implementação** de ORM Java mais usada — e a implementação **JPA** mais popular. Ele executa o que JPA define: gera SQL para cada banco (Oracle, Postgres...), gerencia o *persistence context*, faz *dirty checking*, *fetching*, cache etc.

Histórico

- ✓ **2001:** criado por **Gavin King** (substituir EJB Entity Beans).
- ✓ **Hibernate 2/3 (2003–2005):** consolidação, HQL, mapeamentos poderosos.
- ✓ **2006+:** o Hibernate adota/implementa **JPA 1.0** e passa a expor a API padrão JPA além da sua API nativa.
- ✓ **Hibernate 4/5 (2011/2015):** melhorias de performance, integração com JPA 2.1/2.2.
- ✓ **Hibernate 6 (2021+):** reescrita do mecanismo de SQL, suporte moderno a bancos/dialetos e alinhamento com **Jakarta Persistence** (pacote jakarta.persistence.*).

JPA + Hibernate

JPA diz **o quê** (interfaces, anotações, regras). **Hibernate** faz **o como** (código que roda e conversa com o banco). Quando você usa **Spring Data JPA**, por baixo o **provider** costuma ser o **Hibernate**.

Analogia: JPA é a **tomada padrão**, Hibernate é um **aparelho** compatível com essa tomada. O Spring Data JPA é o “**adaptador inteligente**” que deixa tudo plug-and-play (ex.: JpaRepository).

- ✓ **Produtividade:** menos JDBC “na mão”; foco em entidades e regras.
- ✓ **Portabilidade:** trocar banco é menos doloroso (JPQL/Criteria + dialetos).
- ✓ **Transações e cache** coerentes (via `@Transactional`, *persistence context*).
- ✓ **Ecosistema Spring:** com **Spring Data JPA**, CRUD, paginação, ordenação e *query derivation* (findByEmail) vêm quase de graça.

Spring Data

Spring Data é um conjunto de bibliotecas do Spring que **facilita ler e salvar dados**. Ele elimina boa parte do código repetitivo para acessar bancos (relacionais e não-relacionais). Usamos **Spring Data JPA**, que trabalha por cima do **JPA/Hibernate**.

Ideia central

Você **descreve o que quer** (interfaces e nomes de métodos) e o Spring Data **gera a implementação** para você. Assim, em vez de criar DAOs e escrever SQL de rotina, você cria **repositórios** com métodos como `findAll`, `save`, `findByEmail`.

Spring Data

Relação com JPA e Hibernate

- ✓ **JPA:** a “regra do jogo” (anotações e APIs).
- ✓ **Hibernate:** quem executa de fato as operações (provider JPA).
- ✓ **Spring Data JPA:** uma camada que **aproveita** JPA/Hibernate para te dar **repositórios prontos e convenções de consulta**.

Pense no Spring Data como **o jeito mais rápido e organizado** de conectar seu **modelo** (@Entity) ao **banco**, mantendo o foco nas regras do seu sistema, não na infraestrutura de acesso a dados.

Banco de dados – Tabela java_cliente

No seu banco de dados Oracle crie a seguinte tabela:

```
CREATE TABLE JAVA_CLIENTE (  
  ID_CLIENTE NUMBER GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,  
  NOME    VARCHAR2(120) NOT NULL,  
  EMAIL   VARCHAR2(120) UNIQUE,  
  TELEFONE VARCHAR2(20),  
  senha   varchar2(512)  
);
```

Para verificar a versão do oracle, execute a seguinte query no banco de dados:

```
SELECT product, version FROM product_component_version;
```

Estrutura do projeto Java Spring



Acesse o **Spring Initializr** (<https://start.spring.io/>) e crie o seu projeto Spring. Adicione as dependências padrão e mais as dependências listadas a seguir:

Essenciais (obrigatórios)

- ✓ **Spring Web** – para criar a API REST.
- ✓ **Spring Data JPA** – para usar JPA/Hibernate com repositórios.
- ✓ **Oracle Driver** – JDBC do Oracle (ojdbc).
- ✓ **Validation** (*spring-boot-starter-validation*) – para futuras validações com @NotBlank, @Email etc.

Recomendados (facilitam o dev)

- ✓ **Spring Boot DevTools** – auto-reload em dev.
- ✓ **Actuator** – health check (/actuator/health), útil pra verificar conexão.
- ✓ **Lombok** (*opcional*) – reduz getters/setters/constructors.
- ✓ **Configuration Processor** (*opcional*) – melhora autocomplete no application.properties.

Configuração do servidor

Após a criação do projeto e a sua abertura no IntelliJ, faça as configurações do servidor no arquivo `src/main/resources/application.properties`.

```
server.port=8080
```

```
spring.datasource.url=jdbc:oracle:thin:@//servidor:porta/SID
```

```
spring.datasource.username=USER
```

```
spring.datasource.password=SENHA
```

```
spring.datasource.driver-class-name=oracle.jdbc.OracleDriver
```

```
# use com cuidado (zera tudo a cada start)
```

```
# spring.jpa.hibernate.ddl-auto=create
```

```
# ou: cria se não existe / altera conforme as entidades (arriscado p/ dados)
```

```
# spring.jpa.hibernate.ddl-auto=update
```

```
spring.jpa.hibernate.ddl-auto=none
```

```
spring.jpa.show-sql=true
```

```
spring.jpa.properties.hibernate.format_sql=true
```

Estrutura das pastas do projeto

assistencia-tecnica/	← raiz do projeto Maven
├ src/	← código-fonte da aplicação
│ └ main/	
│ │ └ java/	
│ │ │ └ br/fiap/assistenciatecnica/	← pacote base (component-scan)
│ │ │ │ └ domain/	← entidades JPA (modelo de domínio)
│ │ │ │ └ repository/	← repositórios Spring Data JPA
│ │ │ │ └ service/	← regras de negócio + @Transactional
│ │ │ │ └ web/	← camada HTTP
│ │ │ │ │ └ controller/	← controllers REST (@RestController)
│ │ │ │ │ └ dto/	← DTOs de entrada/saída da API
│ │ │ │ │ └ error/	← tratamento global de erros da API
│ │ │ └ resources/	← recursos não Java
│ │ │ │ └ static/	← arquivos estáticos (css/js/img) opcional
│ │ │ │ └ templates/	← views Thymeleaf (se usar) opcional
│ │ │ │ └ sql/	← scripts SQL (schema/seed) opcional
│ │ │ │ └ db/migration/	← migrations (Flyway), opcional
│ └ test/	← testes automatizados
│ │ └ java/	← testes de unidade/integração
│ │ │ └ br/fiap/assistenciatecnica/	← pacote de testes
│ │ │ │ └ (subpastas por módulo)	← organização dos testes por domínio
└ postman/	← coleções/ambientes Postman (opcional)
└ src/test/http/	← requests HTTP do IntelliJ (opcional)

Testando a conexão com o banco de dados

Para testar a conexão com o banco de dados, coloque o código a seguir na sua classe **Application**, logo abaixo do método **main()**.

```
@org.springframework.context.annotation.Bean
org.springframework.boot.CommandLineRunner pingOracle(javax.sql.DataSource ds) {
    return args -> {
        try (var conn = ds.getConnection()) {
            if (conn.isValid(2)) { // timeout 2s
                System.out.println("Oracle OK");
            } else {
                System.err.println("Conexão inválida");
            }
        } catch (Exception e) {
            System.err.println("Falha ao conectar: " + e.getMessage());
        }
    };
}
```

Classe de domínio JPA

Uma classe de domínio em JPA, também conhecida como **classe de entidade**, é uma classe Java que representa uma tabela em um banco de dados.

Ela é marcada com a anotação **@Entity** e cada instância dessa classe representa uma linha (registro) nessa tabela, tornando a classe uma representação persistente de um objeto de negócio.

Para identificar de forma única cada registro, você deve definir um campo que será a chave primária usando a anotação **@Id**.

Principais anotações

Anotação	Significado	Propriedades
@Entity	Marca a classe como entidade JPA (gerenciada pelo provedor).	—
@Table	Configura a tabela no banco.	name, schema, uniqueConstraints={@UniqueConstraint(name, columnNames)}, indexes={@Index(name, columnList)}
@Id	Indica a chave primária .	—
@GeneratedValue	Define como o ID é gerado .	`strategy = IDENTITY
@SequenceGenerator	Declara um gerador de sequência (se usar SEQUENCE).	name, sequenceName, allocationSize
@Column	Mapeia campo ↔ coluna e restrições.	name, nullable, length, precision, scale, unique, updatable, insertable

Principais anotações

Anotação

Significado

Propriedades úteis

Default importante

@ManyToOne

Muitos → um (FK).

fetch, optional,
cascade

EAGER por padrão
(considere LAZY)

@OneToMany

Um → muitos (coleção).

mappedBy, fetch,
cascade

LAZY por padrão

@OneToOne

Um ↔ um.

mappedBy, fetch,
cascade, optional

EAGER por padrão

@ManyToMany

Muitos ↔ muitos.

mappedBy, fetch,
cascade

LAZY por padrão

@JoinColumn

Define a **FK**.

name, nullable,
foreignKey=@ForeignKey
(name="...")

—

@JoinTable

Configura **tabela de junção**.

name, joinColumns,
inverseJoinColumns

—

Principais anotações

Anotação	Significado	Onde usar	Observações
@NotBlank	String não nula, nem vazia, nem só espaços.	Preferência: DTOs de entrada (pode ser na entidade).	Use com <code>message="..."</code> . Diferente de <code>@NotNull/@NotEmpty</code> .
@Email	Formato básico de e-mail válido.	DTOs ; pode estar na entidade.	Combine com <code>@NotBlank</code> e, se preciso, <code>@Size(max=...)</code> .
@NotNull	Valor obrigatório (qualquer tipo).	DTOs/entidade.	—
@Size	Tamanho de String/coleção.	DTOs/entidade.	<code>min, max</code> .
@Pattern	Regex para String.	DTOs/entidade.	Valida formato (ex.: senha).

Principais anotações

`@Entity`

`@Table(name = "TB_PRODUTO")`

`public class Produto {`

`@Id`

`@GeneratedValue(strategy = GenerationType.IDENTITY)`

`@Column(name = "ID_PRODUTO")`

`private Long id;`

`@Column(name = "NOME", nullable = false, length = 120)`

`private String nome;`

`@Column(name = "DESCRICAO", length = 500)`

`private String descricao;`

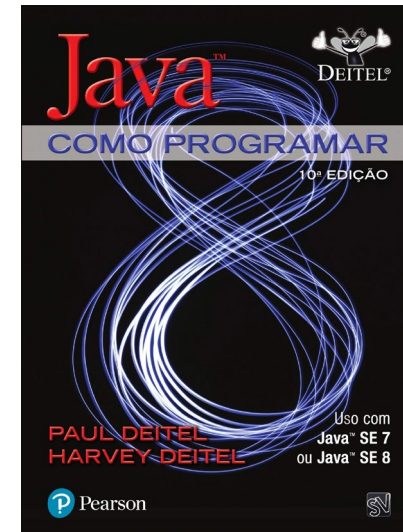
`GenerationType.IDENTITY` diz ao JPA/Hibernate que **o próprio banco gera o valor da PK** no INSERT (auto-incremento).

No Oracle 12c/19c+, isso normalmente corresponde a uma coluna **IDENTITY** (ex.:
GENERATED BY DEFAULT AS
IDENTITY).

Referências



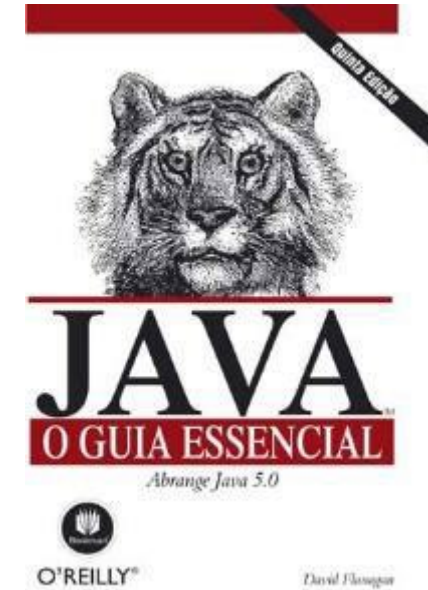
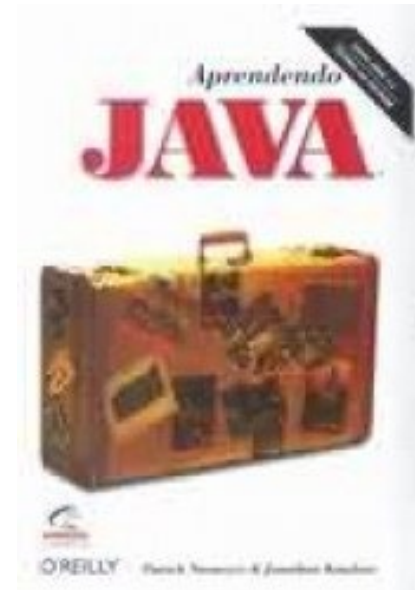
- ❑ DEITEL, H. M., DEITEL, P. J. **JAVA como programar**. 10ª edição. São Paulo: Prentice-Hall, 2010.
- ❑ SCHILDT, H. **Java para Iniciantes – Crie, Compile e Execute Programas Java Rapidamente**. 6ª Edição, Editora Bookman, Porto Alegre, RS, 2015.



Referências



- ❑ KNUDSEN, J., NIEMEYER, P. **Aprendendo Java**. Rio de Janeiro: Editora Elsevier Campus, 2000.
- ❑ FLANAGAN, D. **Java – o guia essencial**. Porto Alegre: Editora Bookman, 2006.



Referências



- ❑ ARNOLD, K., GOSLING, J., HOLMES, D., **Java programming language**. 4th Edition, Editora Addison-Wesley, 2005.
- ❑ JANDL JUNIOR, P. **Introdução ao Java**. São Paulo: Editora Berkeley, 2002.

