

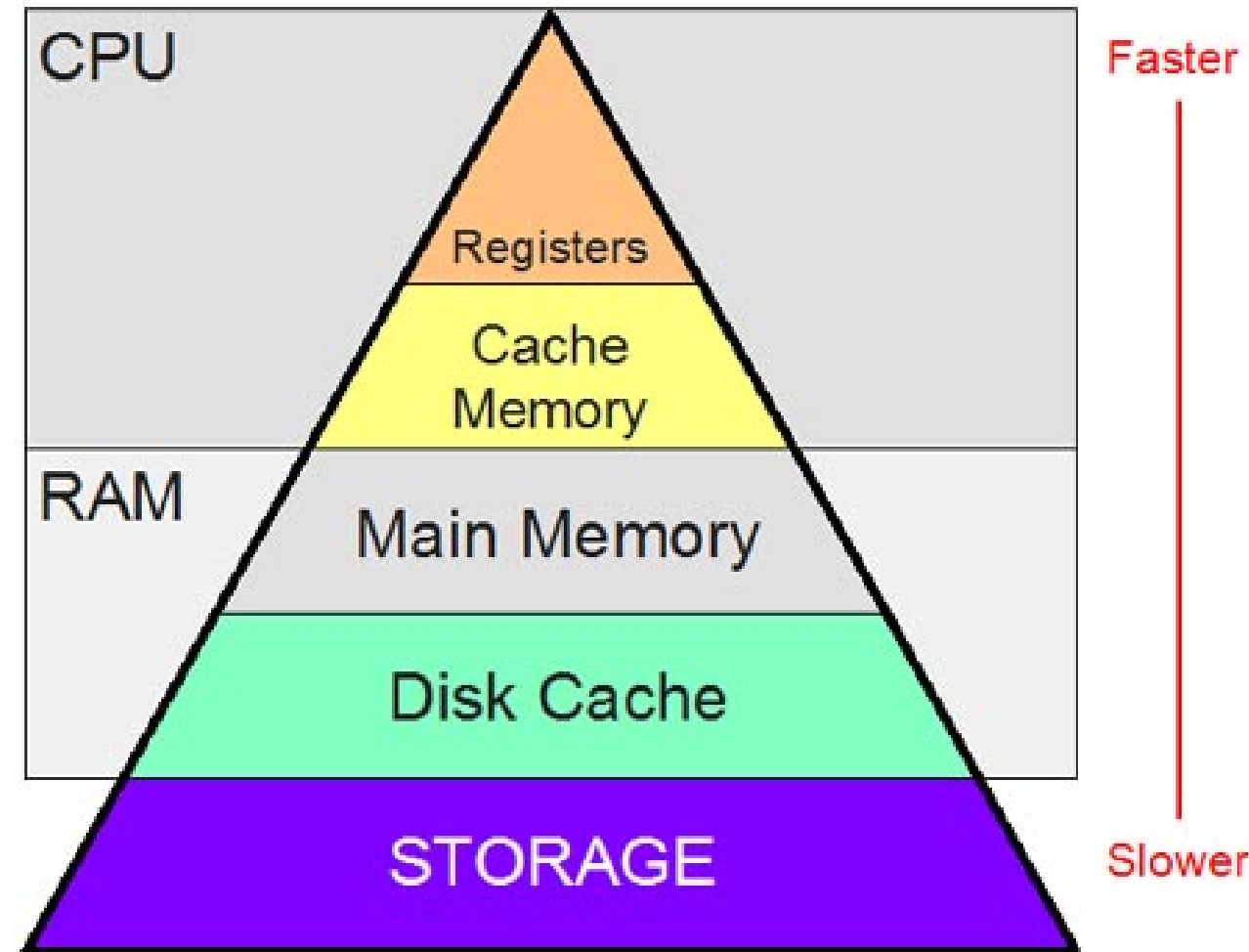
TÓPICO 10 - PERSISTÊNCIA DE DADOS

Disciplina de Backend - Professor Ramon Venson - SATC 2026.2

Persistência de Dados

- Dado preservados após o tempo de vida da aplicação;
- Oposto de **Dados Efêmeros**;
- Dados armazenados diferentes estruturas de hardware/software;

Não confundir com Estruturas de Dados Persistentes
(*Persistent Data Structure*);



Exemplos de dados persistentes

- Clipe de música salvo no sistema de arquivos;
 - O sistema de arquivos pode estar contido em um HDD ou Pendrive.
- Registro de compra, salvo em um banco de dados Postgres
 - O banco de dados é uma abstração para os dados que serão armazenados em uma mídia persistente (como um HDD);
- Logs de erros da aplicação armazenados em um arquivo de texto;

Não são exemplos de dados persistentes

- Objetos da aplicação mantidos na memória;
- Mensagens HTTP ou de qualquer outro protocolo de rede;
- Qualquer tipo de dado armazenado em memória não persistente (DRAM, Cache, Registradores).

Presentation Layer
UserController



Business Layer
User, UserService



Persistence Layer
UserRepository



Database Layer
h2 database

Camada de Persistência

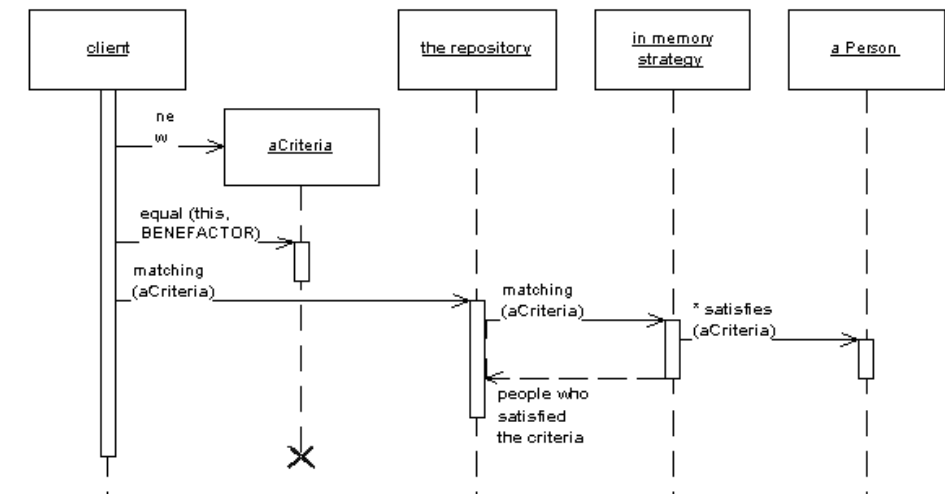
Vamos definir uma nova camada de persistência para abstrair todas as interações relacionadas à persistência dos dados.

Para isso vamos usar um padrão (*pattern*) chamado **Repositórios**.

Repositórios

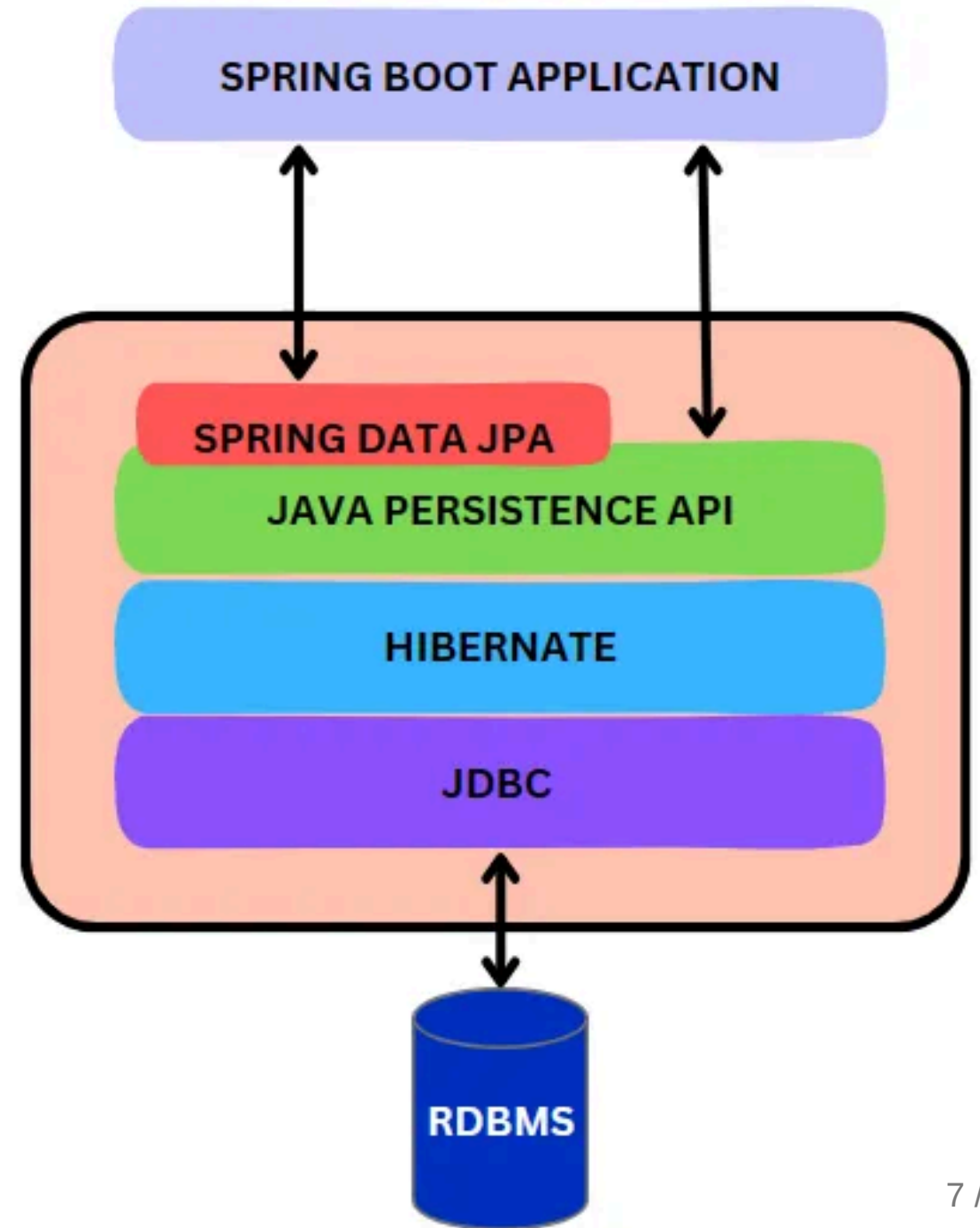
- Isolamento da persistência dos dados;
- Mediação entre domínio (models) e mapeamento (relacional);
- Define operações da aplicação sobre a persistência.

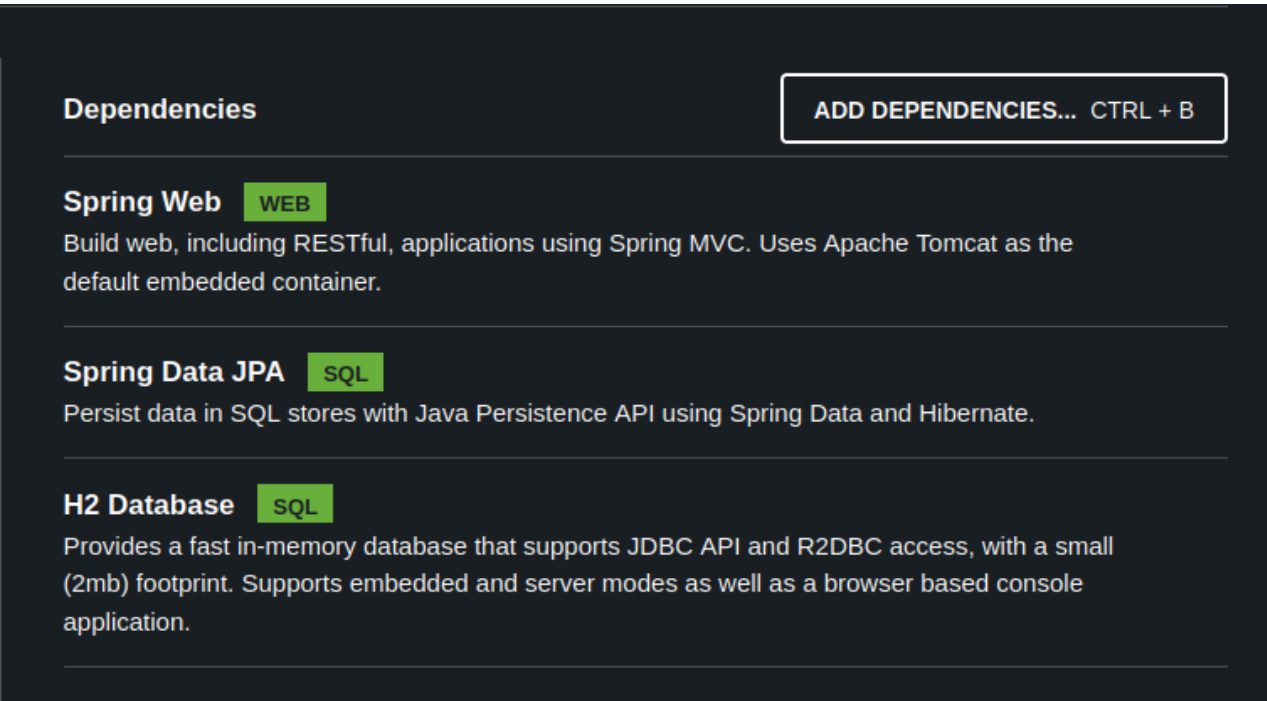
Repository Pattern por Martin Fowler



Repositórios no Spring

- Utiliza o Spring Data JPA como base;
- Define uma mesma interface para todos os tipos de persistência;
- Recebe um driver e, em conjunto com o spring boot, configura automaticamente.





Dependências

- Spring Web
- Spring Data JPA
- Conexão com Banco (H2, Postgres, Maria, Oracle...)

Classes de Repositório

Das classes abaixo, pela hierarquia de funcionalidades e herança:

- `Repository`
- `CrudRepository`
- `PagingAndSortingRepository`
- `JpaRepository`

`Repository` possui a menor quantidade de métodos, enquanto `JpaRepository` estende as funcionalidades de todas as outras classes.

Model

No modelo, vamos adicionar as anotações `@Entity` e `@Table` (com o nome da tabela a ser gerada), além de um `@Id` `@GeneratedValue`, que define a chave primária e automatiza a geração do ID.

```
@Entity
@Table(name = "produtos")
public class Produto {
    @Id // do pacote Jakarta
    @GeneratedValue(strategy = GenerationType.UUID)
    private UUID id;
    private String nome;
    private Double preco;

    public Produto() {} // construtor vazio exigido pelo JPA

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

JpaRepository

Criando uma classe de repositório para a entidade `Produto`. Usamos o `<Modelo, TipoPrimaryKey>`:

```
@Repository
public interface ProdutoRepository extends JpaRepository<Produto, UUID> {
}
```

Service

Injetando repositório no service

Vamos injetar uma instância de repositório para habilitar a manipulação no *service*:

```
@Service
public class ProdutoService {
    @Autowired
    ProdutoRepository produtoRepository;
}
```

Retornando todas as instâncias

Podemos utilizar o método `findAll()` para retornar todas as instâncias do modelo:

```
public List<Produto> getAll() {  
    return produtoRepository.findAll();  
}
```

Retornando instância pelo ID

Podemos utilizar o método `findById()` para retornar uma instância específica do modelo:

```
public Produto getOne(UUID id) {  
    return produtoRepository.findById(id).get();  
}
```

O método `findById` retorna um `Optional`, por isso usamos o método `get()` para retornar a instância diretamente.

Salvando nova instância

Para salvar uma nova instancia, precisamos de um objeto da instância em conjunto com o método `save(entidade)` :

```
public Produto save(Produto novoProduto) {  
    return produtoRepository.save(novoProduto);  
}
```

Também podemos instanciar um novo objeto do tipo `Produto` se for necessário e salvá-lo usando o método `save` .

Atualizando uma instância pelo ID

Para atualizar uma instancia, precisamos recuperar objeto da instância através do ID. Antes de salvar, é necessário alterar as propriedades pretendidas e então usar o método `save`:

```
public Produto update(UUID id, Produto produtoAtualizado) {  
    Produto produto = produtoRepository.findById(id).get();  
    produto.setNome(produtoAtualizado.getNome());  
    produto.setPreco(produtoAtualizado.getPreco());  
    produtoRepository.save(produto);  
    return produto;  
}
```

Deletando uma instância pelo ID

Para deletar uma instancia, precisamos recuperar objeto da instância através do ID e então deletá-lo usando o método `delete(entidade)` :

```
public Produto delete(UUID id) {  
    Produto produto = produtoRepository.findById(id).get();  
    produtoRepository.delete(produto);  
    return produto;  
}
```

Também é possível usar o método `deleteById` caso não haja a necessidade de retornar o objeto deletado

Métodos Comuns

Método	Descrição
<code>findAll()</code>	Retorna todas as instâncias
<code>findAll(pageable)</code>	Retorna todas as instâncias com configuração de paginação
<code>findById(id)</code>	Retorna uma instância de acordo com o ID
<code>getReferenceById(id)</code>	Retorna uma referência da instância pelo ID
<code>save(entidade)</code>	Salva a entidade passada como argumento
<code>delete(entidade)</code>	Deleta a entidade passada como parâmetro

Queries Derivadas

O Spring Data implementa uma linguagem padrão para criação de queries personalizadas baseadas em uma linguagem padrão de métodos.

Para criar um novo método que pesquisa por nome ao invés de ID, podemos incluir o seguinte método no repositório:

```
@Repository
public interface ProdutoRepository extends JpaRepository<Produto, UUID> {
    List<Produto> findByName(String nome)
}
```

Outras Queries Derivadas

```
@Repository
public interface ProdutoRepository extends JpaRepository<Produto, UUID> {
    List<Produto> findByName(String nome) // retorna todos com o nome
    List<Produto> findTop3ByOrderByPrecoDesc() // retorna os 3 produtos mais caros
    List<Produto> findByNameIsNot(String nome); // retorna todos que nao possuem o nome
    List<Produto> findByPrecoLessThan(Double preco); // retorna com condicional
    List<Produto> findByNameContaining(String trecho); // retorna os que contém o nome
    List<Produto> findByNameOrderByPreco(String nome); // ordena por preço
}
```

Documentação completa [aqui](#)

Query Customizada

Para os casos onde não é possível realizar consultas ou a otimização do SQL é fundamental, também podemos mapear consultas SQL personalizadas

```
@Repository
public interface ProdutoRepository extends JpaRepository<Produto, UUID> {
    @Query("SELECT * FROM produtos p WHERE p.preco = 0")
    Collection<Produto> findFreeProducts();
}
```

Configuração do H2

```
spring.datasource.url=jdbc:h2:file:./produtos.db  
spring.jpa.generate-ddl=true  
spring.jpa.hibernate.ddl-auto=update  
spring.h2.console.enabled=true  
spring.jpa.show-sql=true
```

- `datasource.url` : define o caminho para o arquivo da base de dados
- `jpa.generate-ddl` : Ativa a geração automática de tabelas
- `jpa.hibernate.ddl-auto` : Gera as tabelas automaticamente (DDL)
- `h2.console.enabled` : Habilita a UI em `http://{hostname:port}/h2-console`
- `jpa.show-sql` : Faz o log das consultas SQL no console

Exemplo em YAML:

```
spring:
  datasource:
    url: jdbc:h2:file:./produtos.db
  jpa:
    generate-ddl: true
    hibernate:
      ddl-auto: update
  h2:
    console:
      enabled: true
  jpa:
    show-sql: true
```