

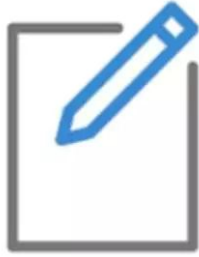
BACK-END

Prof. Bruno Kurzawe



Persistência de Dados (Parte 2)

CRUD



CREATE



READ



UPDATE



DELETE

C

R

U

D

CRUD é um acrônimo que representa as quatro operações básicas usadas em sistemas de gerenciamento de banco de dados relacionais ou em sistemas que envolvem a manipulação de dados.

Salvar (Create)

```
public void salvar(Produto produto) {  
    produtoRepository.save(produto);  
}
```

Buscar todos (Read)

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

Buscar por ID (Read)

```
public Produto buscarPorId(Long id) {  
    return produtoRepository.findById(id).orElse(null);  
}
```


Atualizar (Update)

```
public void atualizar(Produto produto) {  
    if (produtoRepository.existsById(produto.getId())) {  
        produtoRepository.save(produto);  
    } else {  
        // Lógica para lidar com o produto inexistente  
    }  
}
```

Excluir (Delete)

```
public void excluirPorId(Long id) {  
    produtoRepository.deleteById(id);  
}
```

Relacionamentos

One-to-One (Um-para-Um):

Definição: Refere-se a uma relação em que uma instância de uma entidade está associada a no máximo uma instância de outra entidade e vice-versa.

Exemplo: Um exemplo clássico é a relação entre um Usuário e um Perfil. Cada usuário pode ter apenas um perfil e cada perfil está associado a apenas um usuário.

One-to-One (Um-para-Um):

```
@Entity
@Table(name = "usuarios")
public class Usuario {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String nome;

    @OneToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "detalhes_usuario_id")
    private DetalhesUsuario detalhesUsuario;

    // Getters e Setters

}
```

```
@Entity
@Table(name = "detalhes_usuario")
public class DetalhesUsuario {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String endereco;
    private String telefone;

    @OneToOne(mappedBy = "detalhesUsuario")
    private Usuario usuario;

    // Getters e Setters

}
```

One-to-Many (Um-para-Muitos):

Definição: Refere-se a uma relação em que uma instância de uma entidade está associada a zero, uma ou várias instâncias da outra entidade, mas a entidade associada está associada a no máximo uma instância da primeira entidade.

Exemplo: Um exemplo é a relação entre um Autor e seus Livros. Um autor pode ter escrito vários livros, mas cada livro está associado a apenas um autor.

One-to-Many (Um-para-Muitos):

```
@Entity
@Table(name = "posts")
public class Post {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String titulo;
    private String conteudo;

    @OneToMany(mappedBy = "post", cascade = CascadeType.)
    private List<Comentario> comentarios = new ArrayList;

    // Getters e Setters

}
```

```
@Entity
@Table(name = "comentarios")
public class Comentario {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String texto;

    @ManyToOne
    @JoinColumn(name = "post_id")
    private Post post;

    // Getters e Setters

}
```

Many-to-One (Muitos-para-Um):

Definição: Refere-se a uma relação em que várias instâncias de uma entidade estão associadas a no máximo uma instância da outra entidade.

Exemplo: Um exemplo é a relação entre vários Itens de Pedido e um Pedido. Vários itens de pedido podem estar associados a um único pedido.

Many-to-One (Muitos-para-Um):

```
@Entity
@Table(name = "produtos")
public class Produto {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String nome;
    private Double preco;

    @ManyToOne
    @JoinColumn(name = "categoria_id")
    private Categoria categoria;

    // Getters e Setters

}
```

```
@Entity
@Table(name = "categorias")
public class Categoria {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String nome;

    // Getters e Setters

}
```

Many-to-Many (Muitos-para-Muitos):

Definição: Refere-se a uma relação em que várias instâncias de uma entidade podem estar associadas a várias instâncias da outra entidade.

Exemplo: Um exemplo é a relação entre Alunos e Cursos. Um aluno pode estar matriculado em vários cursos, e um curso pode ter vários alunos matriculados.

Many-to-Many (Muitos-para-Muitos):

```
@Entity
@Table(name = "estudantes")
public class Estudante {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String nome;

    @ManyToMany(fetch = FetchType.LAZY)
    @JoinTable(
        name = "estudantes_cursos",
        joinColumns = @JoinColumn(name = "estudante_id"),
        inverseJoinColumns = @JoinColumn(name = "curso_id")
    )
    private Set<Curso> cursos = new HashSet<>();

    // Getters e Setters
}
```

```
@Entity
@Table(name = "cursos")
public class Curso {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String nome;

    @ManyToMany(mappedBy = "cursos", fetch = FetchType.LAZY)
    private Set<Estudante> estudantes = new HashSet<>();

    // Getters e Setters
}
```

Considerações Finais:

A escolha entre esses tipos de relacionamento depende da natureza dos dados e dos requisitos do sistema.

É importante lembrar que, na prática, a modelagem de dados pode ser mais complexa do que essas definições básicas e pode envolver nuances adicionais, como bidirecionalidade, tabelas de junção extras, etc.

Além disso, muitos frameworks ORM, como o Hibernate para Java, podem abstrair grande parte da complexidade da implementação desses relacionamentos, tornando o desenvolvimento mais fácil.

Fetch Type

Em JPA (Java Persistence API), o fetch type determina como as associações em uma entidade serão carregadas do banco de dados. Existem dois tipos principais de fetch type Eager (Ansioso) e Fetch Type Lazy (Preguiçoso).


```
@Entity
@Table(name = "estudantes")
public class Estudante {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String nome;

    @ManyToMany(fetch = FetchType.LAZY)
    @JoinTable(
        name = "estudantes_cursos",
        joinColumns = @JoinColumn(name = "estudante_id")
        inverseJoinColumns = @JoinColumn(name = "curso_i
    )
    private Set<Curso> cursos = new HashSet<>();

    // Getters e Setters
}
```



Fetch Type Eager (Ansioso)

Com Eager, as associações são carregadas imediatamente junto com a entidade principal durante a consulta. Isso significa que os dados associados são carregados imediatamente e estão disponíveis para uso sem a necessidade de mais consultas ao banco de dados. Eager é usado quando você sabe que sempre precisará dos dados associados e quer evitar o carregamento adicional. No entanto, ele pode levar ao carregamento desnecessário de dados se as associações nem sempre forem usadas.

Fetch Type Lazy (Preguiçoso)

Com Lazy, as associações são carregadas somente quando são acessadas pela primeira vez. Isso significa que os dados associados só são carregados quando você realmente precisar deles, economizando assim recursos de sistema. Lazy é usado quando você espera que os dados associados não sejam sempre necessários ou quando quer minimizar o carregamento inicial. No entanto, pode levar a `LazyInitializationException` se as associações não forem carregadas adequadamente em um contexto de transação ativo.

```
@ManyToOne(fetch = FetchType.LAZY)  
private Categoria categoria;
```

```
@OneToOne(fetch = FetchType.EAGER)  
private DetalhesUsuario detalhesUsuario;
```

Exemplo

```
@Entity
@Table(name = "posts")
public class Post {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String titulo;
    private String conteudo;

    @OneToMany(mappedBy = "post", fetch = FetchType.LAZY)
    private List<Comentario> comentarios = new ArrayList<>();

    // Getters e Setters
}
```

```
@Entity
@Table(name = "comentarios")
public class Comentario {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String texto;

    @ManyToOne
    @JoinColumn(name = "post_id")
    private Post post;

    // Getters e Setters
}
```

Exemplo

```
@Service
public class PostService {

    @Autowired
    private PostRepository postRepository;

    public Post obterPost(Long postId) {
        Optional<Post> optionalPost = postRepository.findById(postId);

        if (optionalPost.isPresent()) {
            Post post = optionalPost.get();
            // Neste ponto, os comentários NÃO foram carregados
            return post;
        } else {
            // Lidar com o caso em que o post não foi encontrado
            return null;
        }
    }
}
```

Se for Eager, os comentários já serão carregados aqui.

```
Post post = postService.obterPost(1L);
List<Comentario> comentarios = post.getComentarios();
```

Se for Lazy, os comentários já serão carregados aqui.

Gerenciamento de Transações

O gerenciamento de transações em aplicações Java é uma parte crucial do desenvolvimento de sistemas robustos e confiáveis. O Hibernate é uma biblioteca popular de mapeamento objeto-relacional (ORM) que se integra bem com o JPA (Java Persistence API) para facilitar o gerenciamento de entidades e suas relações com o banco de dados.

O Spring Boot configura automaticamente o gerenciamento de transações, mas é bom especificar explicitamente. Adicione a anotação **@Transactional** aos métodos que envolvem transações.

A anotação **@Transactional** é uma anotação em Java que indica que um método deve ser executado dentro de uma transação. Ela pode ser aplicada em nível de classe (para todos os métodos) ou em nível de método (apenas para o método específico).


```
@Service
public class ProdutoService {

    @Autowired
    private ProdutoRepository produtoRepository;

    @Transactional
    public void salvarProduto(Produto produto) {
        produtoRepository.save(produto);
    }

    // ...
}
```

Parâmetros do @Transactional

propagation (Propagação)

Define como a transação se comporta quando chamadas de método aninhadas são feitas dentro do mesmo contexto de transação. Alguns valores possíveis incluem **REQUIRED**, **REQUIRES_NEW**, **SUPPORTS**, etc.

Ex.:

```
@Transactional(propagation = Propagation.REQUIRED)
```

propagation (Propagação)

REQUIRED	Se já existe uma transação, a nova transação se junta a ela. Se não existe uma transação, uma nova é iniciada.
SUPPORTS	A nova transação se junta à transação existente se já existe uma. Se não há transação, a nova transação é executada sem contexto de transação.
MANDATORY	Exige que já exista uma transação. Se não houver uma transação existente, uma exceção será lançada.
REQUIRES_NEW	Sempre inicia uma nova transação. Se já existe uma transação, ela será suspensa até que a nova transação seja concluída.
NOT_SUPPORTED	Executa a transação sem contexto de transação. Se já existe uma transação, ela será suspensa até que a nova transação seja concluída.
NEVER	Garante que nenhuma transação está presente. Se uma transação existir, uma exceção será lançada.

isolation (Isolamento)

Define o nível de isolamento da transação. Isso determina o grau em que as operações de uma transação são isoladas de operações simultâneas realizadas por outras transações.

Ex.:

```
@Transactional(isolation = Isolation.READ_COMMITTED)
```

isolation (Isolamento)

DEFAULT	O nível de isolamento padrão do banco de dados será utilizado. Isso depende da configuração do banco de dados específico.
READ_UNCOMMITTED	A transação pode ler dados não confirmados por outras transações. Isso pode resultar em leituras de dados sujos, ou seja, dados que podem ser revertidos por outras transações.
READ_COMMITTED	A transação só pode ler dados que foram confirmados por outras transações. Isso evita leituras de dados sujos, mas ainda permite que outros usuários modifiquem os dados entre as leituras.
REPEATABLE_READ	Garante que, durante a execução da transação, os dados lidos não mudarão. Isso significa que, se uma linha é lida em uma transação, ela será a mesma em qualquer leitura subsequente dentro da mesma transação.
SERIALIZABLE	Oferece o nível mais alto de isolamento. Garante que a execução de múltiplas transações simultâneas produzirá o mesmo resultado que se as transações fossem executadas sequencialmente.

readOnly (Somente Leitura)

Indica se a transação deve ser tratada como somente leitura. Isso pode otimizar o desempenho, pois o banco de dados pode aplicar certas otimizações quando sabe que os dados não serão alterados.

Ex.:

```
@Transactional(readOnly = true)
```

timeout (Tempo Limite)

Define o tempo limite para a transação. Se a transação não for concluída dentro do tempo especificado, será automaticamente revertida.

Ex.:

```
@Transactional(readOnly = true)
```


rollbackFor e noRollbackFor

Específica exceções que devem ou não causar um rollback da transação.

Ex.:

```
@Transactional(rollbackFor = CustomException.class)
```

```
@Transactional(noRollbackFor = SpecificException.class)
```

QueryDSL

O **QueryDSL** é uma biblioteca que oferece uma maneira mais fluente e tipada de construir consultas SQL (ou consultas JPA) em Java. Com o QueryDSL, você pode escrever consultas usando uma sintaxe Java parecida com SQL, o que torna o código mais legível e menos propenso a erros de digitação.

O QueryDSL é frequentemente usado em conjunto com JPA e Hibernate para simplificar a construção de consultas complexas em bancos de dados relacionais.

Vamos configurar esse cara no nosso projeto

```
</dependency>  
<dependency>  
  <groupId>org.springframework.boot</groupId>  
  <artifactId>spring-boot-starter-test</artifactId>  
  <scope>test</scope>  
</dependency>
```

```
<dependency>  
  <groupId>com.querydsl</groupId>  
  <artifactId>querydsl-apt</artifactId>
```

```
</dependency>
```


```
<dependency>  
  <groupId>com.querydsl</groupId>  
  <artifactId>querydsl-jpa</artifactId>
```

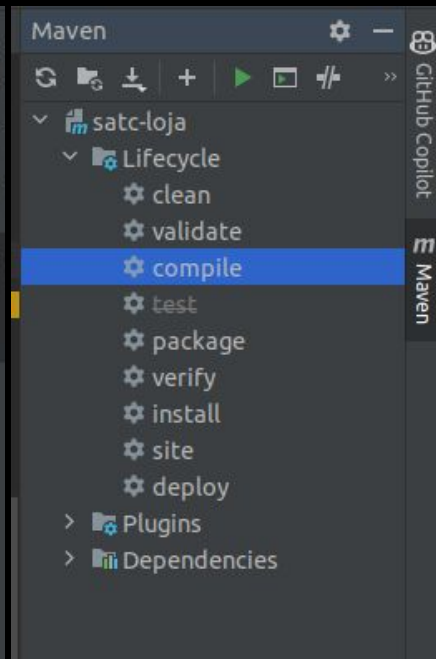
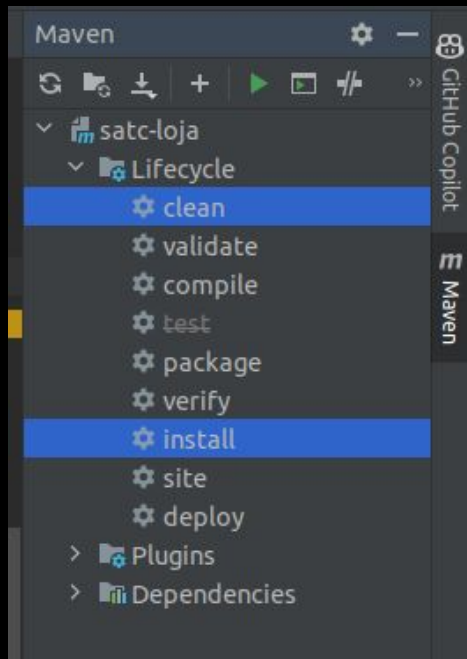
```
</dependency>
```

```
</dependencies>
```

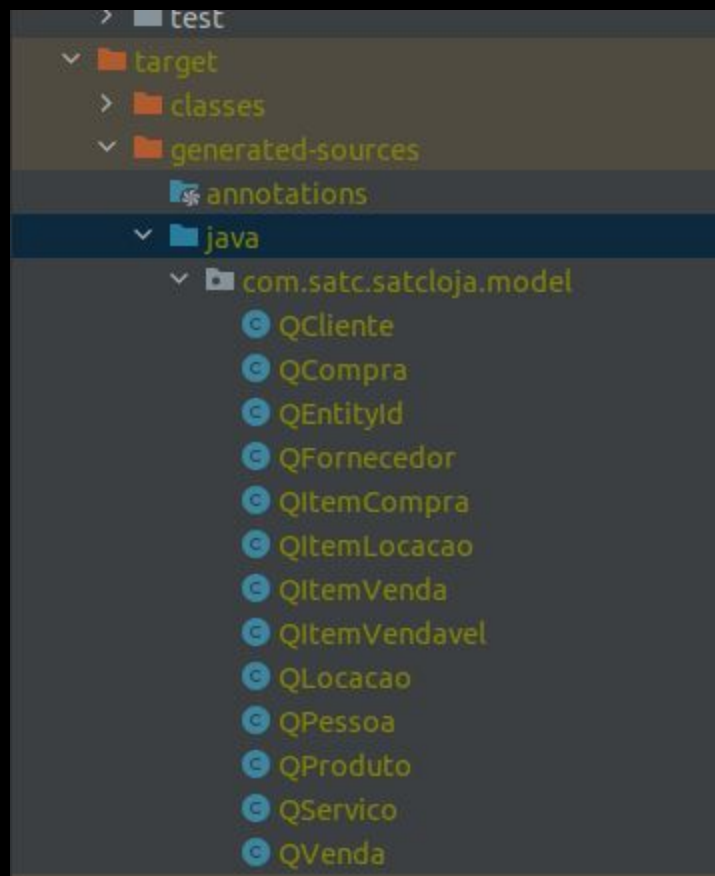


```
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
    <plugin>
      <groupId>com.mysema.maven</groupId>
      <artifactId>apt-maven-plugin</artifactId>
      <version>1.1.3</version>
      <executions>
        <execution>
          <phase>generate-sources</phase>
          <goals>
            <goal>process</goal>
          </goals>
          <configuration>
            <outputDirectory>target/generated-sources/java</outputDirectory>
            <processor>com.querydsl.apt.jpa.JPAAnnotationProcessor</processor>
          </configuration>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```





```
[INFO] -----  
[INFO] BUILD SUCCESS  
[INFO] -----  
[INFO] Total time: 2.833 s  
[INFO] Finished at: 2023-09-10T11:49:29-03:00  
[INFO] -----
```





Aqui faremos um novo **refactoring!**

```
package com.satc.satcloja.repository;

import com.satc.satcloja.model.Cliente;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface ClienteRepository extends JpaRepository<Cliente, Long> {
}


```

```
package com.satc.satcloja.repository;

import com.satc.satcloja.model.Cliente;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.querydsl.QuerydslPredicateExecutor;
import org.springframework.stereotype.Repository;

@Repository
public interface ClienteRepository extends JpaRepository<Cliente, Long>, QuerydslPredicateExecutor<Cliente> {
}
```



Vamos fazer isso para todas as classes **repository**!

Mas para que serve isso?

Vamos criar um service, **ProdutoService**

Digamos que eu precise de um serviço que busque todos os produtos alugados.

Poderia fazer desta forma?

```
@Service
public class ProdutoService {

    @Autowired
    private ProdutoRepository repository;

    public List<Produto> findProdutosAlugados() {
        List<Produto> alugados = new ArrayList<>();
        List<Produto> todos = repository.findAll();

        for (Produto produto : todos) {
            if (produto.getStatus().equals(Status.ALUGADO)) {
                alugados.add(produto);
            }
        }

        return alugados;
    }
}
```

Tem uma forma melhor, porém antes, vamos fazer um **refactoring!**

Criem uma interface **CustomQuerydslPredicateExecutor**, vamos mudar um comportamento padrão do JPA com queryDSL

```
public interface CustomQuerydslPredicateExecutor<T> extends QuerydslPredicateExecutor<T> {  
  
    @Override  
    List<T> findAll(Predicate predicate);  
}
```

Vamos trocar por essa classe

```
package com.satc.satcloja.repository;

import com.satc.satcloja.enterprise.CustomQuerydslPredicateExecutor;
import com.satc.satcloja.model.Produto;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

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



Vamos fazer isso para todas as classes **repository**!

```
@Service
public class ProdutoService {

    @Autowired
    private ProdutoRepository repository;

    public List<Produto> findProdutosAlugados() {
        List<Produto> alugados = repository.findAll(QProduto.produto.status.eq(Status.ALUGADO));
        return alugados;
    }
}
```

Podemos usar EQ, NE, IN, GT, LT, GOE, LOE, BETWEEN, NOTIN, NOTBETWEEN e mais alguns métodos.


```
@Autowired  
private ProdutoRepository repository;
```

```
public List<Produto> findProdutosAlugados() {
```

```
    List<Produto> alugados = repository.findAll(QProduto.produto.status.eq(Status.ALUGADO));
```

```
    return alugados;
```

```
}
```

- 
- m eq(Status right)
 - m eq(Expression<? super Status> right)
 - m asc() Ord
 - m eqAll(CollectionExpression<?, ? super...
 - m eqAll(SubQueryExpression<? extends St...
 - m as(Path<Status> alias) Equ


```
@Service
public class ProdutoService {

    @Autowired
    private ProdutoRepository repository;

    public List<Produto> findProdutosAlugados() {
        List<Produto> alugados = repository.findAll(QProduto.produto.status.in(Status.ALUGADO, Status.DISPONIVEL));
        return alugados;
    }
}
```



```
@Service
public class ProdutoService {

    @Autowired
    private ProdutoRepository repository;

    public List<Produto> findProdutosAlugados() {
        List<Produto> alugados = repository.findAll(QProduto.produto.status.eq(Status.ALUGADO)
            .or(QProduto.produto.status.eq(Status.DISPONIVEL)));
        return alugados;
    }
}
```

Podemos fazer AND e OR.

Usaremos muito esse cara quando estivermos criando as lógicas dentro das nossas APIs

Mapeamento Objeto-Documento

```
@Entity
public class Cliente extends Pessoa {

    @Column(name = "cpf", nullable = false)
    private String cpf;

    @Column(name = "rg", nullable = false)
    private String rg;

    public String getCpf() { return cpf; }
```

```
@Document(collection = "clientes")
public class Cliente extends Pessoa {
```

```
    @Id
    private String id;
```

```
    @Field("numero_cpf") // Define o nome d
    private String cpf;
```

```
    @Field("numero_rg") // Define o nome do
    private String rg;
```

Fim da aula 07...