UNIVERSIDADE FEDERAL DO CEARÁ

INSTITUTO UNIVERSIDADE VIRTUAL

CURSO DE GRADUAÇÃO EM SISTEMAS E MÍDIAS DIGITAIS

ANDRÉ LUCAS DA SILVA RODRIGUES

UMA BIBLIOTECA PARA FACILITAR O USO DE JPA CRITERIA QUERIES COM MÚLTIPLOS CAMPOS OPCIONAIS E PROJEÇÕES EM SUA RESPOSTA

FORTALEZA

2022

ANDRÉ LUCAS DA SILVA RODRIGUES

UMA BIBLIOTECA PARA FACILITAR O USO DE JPA CRITERIA QUERIES COM MÚLTIPLOS CAMPOS OPCIONAIS E PROJEÇÕES EM SUA RESPOSTA

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Sistemas e Mídias Digitais do Instituto Universidade Virtual da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Sistemas e Mídias Digitais.

Orientador: Prof. Dr. Leonardo Oliveira Moreira

FORTALEZA

2022

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Biblioteca Universitária
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

R611b Rodrigues, André Lucas da Silva.
Uma biblioteca para facilitar o uso de JPA Criteria Queries com múltiplos campos opcionais e projeções em sua resposta / André Lucas da Silva Rodrigues. – 2022.
39 f. : il. color.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Instituto UFC Virtual, Curso de Sistemas e Mídias Digitais, Fortaleza, 2022.
Orientação: Prof. Dr. Leonardo Oliveira Moreira.

1. Java. 2. Consulta. 3. Banco de Dados. 4. Criteria. I. Título.

CDD 302.23

ANDRÉ LUCAS DA SILVA RODRIGUES

UMA BIBLIOTECA PARA FACILITAR O USO DE JPA CRITERIA QUERIES COM MÚLTIPLOS CAMPOS OPCIONAIS E PROJEÇÕES EM SUA RESPOSTA

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Sistemas e Mídias Digitais do Instituto Universidade Virtual da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Sistemas e Mídias Digitais.

Aprovada em:

BANCA EXAMINADORA

Prof. Dr. Leonardo Oliveira Moreira (Orientador)
Universidade Federal do Ceará (UFC)

Prof. Me. Wellington Wagner Ferreira Sarmento
Universidade Federal do Ceará (UFC)

Prof. Me. Carlos Sergio da Silva Marinho
Centro Universitário Christus (UNICHRISTUS)

À minha família, por sua capacidade de acreditar em mim e investir em mim. Mãe, seu cuidado e dedicação foi que deram, em alguns momentos, a esperança para seguir. Pai, sua presença significou segurança e certeza de que não estou sozinho nessa caminhada.

AGRADECIMENTOS

Primeiramente à Deus por minha vida, família e amigos.

À minha mãe, que sempre fez tudo que estava ao seu alcance para me apoiar no meu caminho e ser a base sólida onde construí meu caráter.

Ao meu tio Sérgio, que nos ajudou bastante nessa caminhada e sempre foi o mais importante ponto de apoio nos momentos difíceis.

À minha namorada, Palloma Fernandes, por toda a força que me foi emprestada para enfrentar os desafios da vida acadêmica, da vida profissional e da vida pessoal. Sem seu apoio eu nunca teria chegado tão longe, espero poder construir muito mais ao seu lado.

Ao Prof. Dr. Leonardo Oliveira Moreira, por acreditar no meu projeto e me ajudar a lapidar esse trabalho acadêmico de uma forma única que só ele poderia ter feito.

Aos meus amigos do Felaship que me acompanham há mais de uma década, dos corredores do IFCE até o fim do mundo se precisar. Que ouviram todas as minhas reclamações durante todo esse processo, mas que sempre me apoiaram, me deram força e nunca me deixaram desacreditar do meu potencial. A presença de vocês é parte essencial da formação de quem sou hoje.

Aos meus amigos dos PKM, que estão sempre por perto e sempre disponíveis, para uma conversa descontraída, para conselhos e para horas e horas de streams intermináveis que me ajudaram a manter a sanidade no meio de tanto trabalho realizado. Saber que sempre posso contar com cada um de vocês me ajudou muito a caminhar.

Às grandes amizades que a UFC me trouxe, que facilitaram meu caminho até aqui e que espero muito que me acompanhem daqui pra frente. Poder comemorar minhas conquistas e reclamar meus problemas com vocês, mesmo de longe pelo Telegram, é muito importante pra mim.

Ao meu amigo Emanuel Penas, que me ensinou demais no tempo que trabalhamos juntos. Foi o primeiro a ouvir minha ideia para essa biblioteca aqui apresentada e desde o princípio me ajudou a chegar nas melhores soluções para seu desenvolvimento. Sou hoje um profissional melhor pela sua ajuda.

A todos aqui citados e todos que por ventura esqueci de citar, muito obrigado.

"O sonho é que leva a gente para frente. Se a gente for seguir a razão, fica aquietado, acomodado."

(Ariano Suassuna)

RESUMO

A linguagem de programação Java é uma das mais utilizadas atualmente e tendo mais de 25 anos desde o lançamento de sua primeira versão é muito fácil encontrá-la em sistemas por todo o mundo. Grande parte dessas aplicações são orientadas a dados, logo a relação da camada de aplicação com a base de dados tem bastante importância na estrutura do projeto e sua velocidade. Tendo isso em vista esse trabalho tem como objetivo analisar pontos de melhoria para que o desenvolvedor consiga lidar com grandes consultas e campos opcionais de uma forma simplificada através da criação de uma biblioteca Java e avaliar o uso do produto desenvolvido. A partir de comparações do código escrito com as estratégias utilizadas atualmente com o código desenvolvido utilizando da biblioteca é possível perceber sólido aumento da clareza de código dos métodos de consulta sem que seu desempenho seja afetado de forma considerável.

Palavras-chave: Java. Consulta. Banco de Dados. Criteria.

ABSTRACT

The Java programming language is one of the most widely used today and, with more than 25 years since the release of its first version, it is very easy to find it in systems all over the world. Most of these applications are data-driven, so the relationship of the application layer to the database is very important in the structure and speed of the project. With this in mind, this paper aims to analyze points of improvement for the developer to handle large queries and optional fields in a simplified way by creating a Java library and evaluate the usage of the developed product. From comparing the code written with the most commonly used strategies currently with the code developed using the library, it is possible to notice a solid reduction in the complexity of the query methods without considerably affecting their performance.

Keywords: Java. Query. Database. Criteria.

LISTA DE TABELAS

LISTA DE CÓDIGOS-FONTE

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Programming Interface*
HQL	*Hibernate Query Language*
IDE	*Integrated Development Environment*
JDBC	*Java DataBase Connectivity*
JPA	*Java Persistence API*
JPQL	*Java Persistence Query Language*
OO	Orientada a Objetos
ORM	*Object–Relational Mapping*
POJO	*Plain Old Java Object*
SGBD	Sistema Gerenciador de Banco de Dados
SQL	*Structured Query Language*

SUMÁRIO

1 INTRODUÇÃO

No desenvolvimento de sistemas, atualmente, a construção de aplicações *backend* para *web* é realizada pela grande maioria dos desenvolvedores, que tem como a segunda linguagem primária mais popular o Java, estando atrás apenas do JavaScript, segundo a pesquisa The State of Developer Ecosystem 2021, realizada pela empresa Jetbrains (JETBRAINS, 2021). Uma pesquisa realizada, em janeiro de 2022, pela TIOBE Index aponta a linguagem Java como a terceira linguagem de programação no *ranking* (TIOBE, 2022). A TIOBE Index atualiza, mensalmente, a classificação das linguagens de programação levando em consideração frequência de pesquisa em *websites*, como o Google, MSN, Yahoo!, Wikipedia e YouTube.

Como muitas aplicações são orientadas a dados e o Sistema Gerenciador de Banco de Dados (SGBD), normalmente, é o componente que gasta mais tempo de resposta em uma requisição de processamento (MARINHO *et al.*, 2018), surge a necessidade de utilizar tecnologias que possam mitigar tal situação. Diante disso e levando em consideração o mundo Java, várias soluções para o âmbito de gestão de dados. No contexto particular de consultas a dados, é muito comum se deparar com o caso de uso de consulta com filtros e campos opcionais. Uma tarefa simples pode ser realizada de várias formas dentro do mundo Java, que possui várias opções para a interação com o banco de dados, onde será feita a consulta.

Como descrito por Caelum (2021b), tecnologias para auxiliar na tarefa de interação de Java com um banco de dados se tornaram populares entre os desenvolvedores Java e são conhecidas como ferramentas de Mapeamento Objeto-Relacional (do inglês Object Relational-Mapping (*Object–Relational Mapping* (ORM))). A *Application Programming Interface* (API) *Java Persistence API* (JPA) fornece um modelo de Mapeamento Objeto-Relacional de persistência de dados baseado em objetos do tipo *Plain Old Java Object* (POJO). É uma especificação para acesso à banco de dados, que funciona com outros *frameworks* ORM, conforme descrito por Neto *et al.* (2016).

Utilizando uma implementação de um *framework* JPA, o desenvolvedor pode se utilizar de consultas nativas e consultas com *Java Persistence Query Language* (JPQL), sendo a última uma linguagem utilizada para gerar consultas a partir dos objetos e seus relacionamentos ao invés das tabelas ou utilizar a chamada API JPA Criteria Queries que gera consultas programaticamente. Embora a API JPA Criteria Queries consiga resolver o problema da consulta com campos opcionais melhor do que as outras tecnologias fornecidas pela API JPA, ela acaba trazendo outro: uma grande quantidade de código repetitivo gerada por tal biblioteca. O progra-

mador trabalhando em uma consulta mais complexa vai, facilmente, lidar com várias linhas de código que serão repetidas entre si, alterando muitas vezes apenas o nome do atributo consultado em determinado trecho (boilerplate).

Para tentar simplificar a formulação de consultas complexas com campos opcionais, diminuindo a dificuldade do uso da API JPA Criteria Queries, foi identificada uma oportunidade de contribuir com a comunidade Java por meio de uma nova biblioteca que seria um *wrapper* para seu uso, permitindo que o desenvolvedor não tenha contato com o *boilerplate* de tal tecnologia. Assim, fornecendo controle para alterar facilmente e rapidamente a consulta realizada e o retorno desejado, evitando também problemas como trazer mais informações do banco de dados do que realmente necessário, o que pode aumentar o tempo de resposta da consulta e quantidade de dados trafegados (*overfetch*).

Sendo assim, o problema de pesquisa foi formulado da seguinte forma: "Como se deve desenvolver uma biblioteca Java que facilite o uso de JPA Criteria Queries com vários campos opcionais e projeções em sua resposta?". Uma hipótese é uma suposta, provável e provisória resposta a um problema de pesquisa, cuja comprovação será verificada através da pesquisa (MARCONI; LAKATOS, 2003). Neste sentido, as hipóteses que permeiam este ensaio são:

- a biblioteca desenvolvida pode reduzir o número de linhas de códigos Java e a complexidade no uso de JPA Criteria Queries com vários campos opcionais e projeções em sua resposta; e
- a forma que a biblioteca foi implementada pode acarretar em um aumento significativo no tempo de resposta das consultas com vários campos opcionais e projeções em sua resposta.

O objetivo geral deste trabalho é desenvolver uma biblioteca no intuito de facilitar o uso de JPA Criteria Queries com vários campos opcionais e projeções em sua resposta. Segundo Marconi e Lakatos (2003) os objetivos específicos visam, de um lado, atingir o objetivo geral e, de outro, aplicá-los a situações particulares. Assim, para alcançar o objetivo geral, os seguintes objetivos específicos foram elencados:

- estudar a API JPA no intuito de entender sua coleção de classes e métodos, além das características dos principais *frameworks* que implementam API JPA;
- projetar a biblioteca para suportar a implementação de API JPA Criteria Queries com vários campos opcionais e projeções em sua resposta; e
- validar a biblioteca por meio de sua utilização, demonstração do uso do número de códigos

e análise do tempo de resposta das consultas.

O restante deste documento está organizado em quatro capítulos. O Capítulo 2 apresenta a teoria necessária para uma compreensão do trabalho. Assim, aborda a linguagem Java e suas características principais, destacando seu paradigma de orientação a objetos. Além disso, no Capítulo 2 comenta sobre as características de um Banco de Dados Relacional. Ainda no Capítulo 2 apresenta as tecnologias e padrões para gestão de dados relacionais por meio da linguagem Java. Já o Capítulo 3 apresenta e detalha a metodologia utilizada para alcançar os objetivos do trabalho. O Capítulo 4, por sua vez, apresenta os resultados obtidos e discute seus detalhes. Por fim, o Capítulo 5 concluí o trabalho e destaca os trabalhos futuros almejados.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo situa os principais conceitos que atravessam este trabalho, compreendendo a linguagem de programação Java, interação da linguagem Java com Banco de Dados Relacional e APIs que realizem o Mapeamento Objeto-Relacional (ORM) para possibilitar aplicações escritas em Java e que se servem de dados relacionais possam ser desenvolvidas em um mesmo paradigma.

2.1 Linguagem de Programação Java

A linguagem Java, segundo Caelum (2021b), foi apresentada em 1995 pela Sun Mycrosystems com o objetivo inicial de ser utilizada em pequenos dispositivos, como televisores, videocassetes, aspiradores, liquidificadores e outros. Atualmente, a linguagem Java pertence a Oracle e tem seu destaque no desenvolvimento de aplicações corporativas, sendo atualmente uma das linguagens mais utilizadas no mundo, como demonstra a pesquisa State of Developer 2021 (JETBRAINS, 2021). Ademais, em uma pesquisa realizada, em janeiro de 2022 pela TIOBE Index, aponta a linguagem Java como a terceira linguagem de programação no *ranking* (TIOBE, 2022).

Neste sentido, Java é uma plataforma de programação que ainda provoca entusiasmo em programadores, analistas e projetistas de software, pois é o resultado de um enorme trabalho de pesquisa científica e tecnológica (JUNIOR, 2021). Essa plataforma é um ambiente completo de desenvolvimento e execução de programas que reúne um conjunto ímpar de facilidades: uma linguagem completamente Orientada a Objetos (OO), robusta, muito portátil, que permite operação em rede (com destaque à Internet), a distribuição de aplicações e que incorpora diversas características voltadas à segurança (JUNIOR, 2021).

Segundo Junior (2021), a plataforma Java como ambiente de programação é plenamente justificada, porque permite o emprego integral das técnicas de programação OO aliado a uma extensa e versátil biblioteca de classes. Além disso, deve-se considerar que o Java pode ser utilizado nas principais plataformas computacionais existentes, o que não é apenas conveniente, pois simplifica sobremaneira o projeto de sistemas de software que operam em ambientes de rede, que são potencialmente heterogêneos, como a web. Além disso, Java é considerada uma linguagem de programação de propósito geral, concorrente, baseada em classes e orientada a objetos. Projetada para ser simples o bastante para que a maioria dos programadores se torne

fluente na linguagem. Java tem relação com C e C++, porém é organizada de forma diferente, com vários aspectos de C e C++ omitidos e algumas ideias de outras linguagens incluídas.

Santos (2018) confronta a linguagem Java com o paradigma de orientação a objetos, que se baseia no conceito de classes e objetos. Dentro do paradigma existem quatro grandes pilares. A abstração, capacidade de criar representações de objetos reais e teóricos em uma classe que com atributos e métodos. O encapsulamento, que define a visibilidade desses atributos e métodos, garantindo maior segurança e coesão das informações representadas. Herança é a possibilidade de criar uma nova classe que reaproveita comportamentos e propriedades de uma classe base, aumentando a velocidade de desenvolvimento e diminuindo a repetição de código. O último pilar é o polimorfismo, que permite alterar o funcionamento interno de um comportamento obtido por herança, permitindo adaptar para situações específicas sem que haja custo em outras partes do sistema (DEVMEDIA, 2014).

Dentro da orientação de objetos e no mundo Java, as classes que representam as entidades são comumente criadas na forma de JavaBeans, definido por Caelum (2021a), como classes que possuem o construtor sem argumentos e métodos de acesso do tipo *get* (obtenção do estado de um atributo privado) e *set* (configuração de um novo estado de um atributo privado), responsáveis pelos acessos aos atributos de instância (dados ou estado) do objeto seguindo o pilar do encapsulamento.

2.2 Banco de Dados Relacional

Um banco de dados que se serve do modelo relacional de dados pode ser conceituado como um conjunto de relações (tabelas), colunas (atributos), tuplas (linhas) e as restrições de integridade oriundos do modelo relacional (ELMASRI; NAVATHE, 2011). Apesar de serem tecnologias de persistência antiga, os bancos de dados relacionais são predominantes no mercado atual (YUAN *et al.*, 2021). O sucesso dos bancos de dados relacionais se deve, em parte, ao modelo relacional, que apresentou uma forma padrão de representar e consultar dados que poderiam ser usados por qualquer aplicativo escrito em qualquer linguagem de programação.

A *Structured Query Language* (SQL) também é outra razão do sucesso dos bancos de dados relacionais que implementam o modelo relacional. A SQL é considerada uma linguagem poderosa, predominante em muitos domínios de problemas (MIGLER; DEKHTYAR, 2020). Outro fator importante da linguagem SQL é que seu formalismo é baseado na álgebra relacional, uma linguagem matemática utilizada internamente nos SGBDs no intuito de realizar processos

de otimizações de consultas para melhorar o desempenho na recuperação de dados e informações (ELMASRI; NAVATHE, 2011).

Mesmo que o banco de dados relacional ainda seja um dos mais utilizados, ele possui algumas restrições que podem ter impacto negativamente em certas aplicações que necessitem uma estrutura de dados flexível e processamento em larga escala. O estudo conduzido por Farias (2014) aponta algumas restrições do modelo relacional de dados: a) não permite uma estrutura de dados flexível que forneça uma obtenção de um nível mais alto de escalabilidade; e b) não fornece a estruturação dos seus dados de acordo com os anseios individuais.

2.3 Java Database Connectivity (JDBC)

Como muitas aplicações são orientadas a dados (MARINHO *et al.*, 2018), surge a necessidade de uma forma para implementação a gestão de dados nas aplicações, neste caso em Java. Sendo assim, muitas aplicações e sistemas precisam manter as informações com as quais eles trabalham para permitir consultas futuras, geração de relatórios ou possíveis alterações nas informações. Para que esses dados ou informações sejam mantidos de forma mais confiável, geralmente, tais dados e informações são armazenadas em um banco de dados, que as mantém de forma organizada para futuras manipulações na forma de consultas (CAELUM, 2021a). A principal forma de interação entre aplicações Java e banco de dados é através da API *Java DataBase Connectivity* (JDBC).

Java Database Connectivity (JDBC) é uma API de programação para desenvolvedores Java que necessitam de aplicações que acessem informações armazenadas em bancos de dados, planilhas e arquivos simples (HENRY, 2001). Segundo Henry (2001), JDBC é comumente usado para conectar uma aplicação Java a um banco de dados, independentemente de qual *software* SGBD é usado para gerenciar o banco de dados. Desta forma, JDBC é considerado multiplataforma, pois a ideia é que da mesma forma que se interage um banco de dados A é o mesmo com o banco de dados B. A nível de programação, a API JDBC padroniza a forma de programar, independente do fabricante da fonte de dados.

2.4 Java APIs para ORM

Além de padronizar a interação com o banco de dados, o ecossistema Java fornece a especificação JPA para simplificar o processo de representação de tabelas pelas classes e facilitar

a criação de consultas sem que o desenvolvedor tenha que se preocupar com especificidades dos diferentes bancos de dados possíveis de serem utilizados. Essa ligação é chamada de Mapeamento Objeto Relacional (ORM). Um exemplo muito importante de ferramenta ORM no mundo Java é o Hibernate, um framework que nasceu anteriormente a própria JPA, mas que hoje é comum ser utilizado através da especificação JPA. Entre as implementações mais comuns, podemos citar: Hibernate da Red Hat, EclipseLink da Eclipse Foundation e o OpenJPA da Apache. Apesar do Hibernate ter originado a JPA, o EclipseLink é a implementação referencial (CAELUM, 2021b).

Esta especificação nos fornece algumas formas de realizar a consulta com base nos objetos mapeados para o banco de dados. A consulta pode ser realizada através de código nativo SQL, porém não é a opção mais recomendada por perder as simplificações advindas do uso da biblioteca que implementa a JPA. Pode também ser realizada com uma linguagem própria da JPA chamada JPQL, muito parecida com o SQL original, porém faz referencia a objetos Java ao invés das tabelas diretamente, fornecendo também vários recursos para lidar com parametrização, junção (*join*) e acessos a propriedades e atributos da classe desejada, fazendo na hora da consulta automaticamente o mapeamento para a coluna configurada na declaração da entidade. Igualmente, é possível construir as consultas de forma mais complexa utilizando de uma API fornecida pela especificação JPA chamada de Criteria Queries, onde as consultas são montadas programaticamente, permitindo que todos os recursos da linguagem Java sejam utilizados na construção, parametrização e organização dos requisitos necessários na consulta, não requisitando assim do desenvolvedor conhecimentos sobre a escrita da consulta realizada no banco de dados.

3 METODOLOGIA

Este capítulo apresenta a metodologia utilizada na elaboração do trabalho, destacando as etapas importantes, onde são destacados o levantamento de requisitos, a análise e projeto, tecnologias utilizadas e a forma de validação do trabalho.

3.1 Aspectos Metodológicos

Segundo Oliveira e Seabra (2015) as metodologias de desenvolvimento de software foram concebidas para utilização orientada de métodos, ferramentas e procedimentos, com foco na elaboração de um produto de software. Tais metodologias visam o aprimoramento, reduzir custos, tempo de execução e a melhoria da qualidade do produto final (OLIVEIRA; SEABRA, 2015).

O modelo incremental permite que o usuário ou cliente do produto priorize quais funcionalidades serão entregues primeiro (SOMMERVILLE, 2011). Assim, a cada entrega de um conjunto de funcionalidades que engloba uma iteração, esse conjunto de funcionalidades pode ser agregados ao produto em produção, evoluindo sempre em direção ao produto almejado pelo usuário ou cliente (OLIVEIRA; SEABRA, 2015).

3.2 Etapas Importantes do Processo Metodológico

Nas próximas subseções são ressaltadas as etapas mais importantes inseridas no processo metodológico utilizado para a concepção da biblioteca para facilitar o uso de JPA Criteria Queries com vários campos opcionais e projeções em sua resposta.

3.2.1 Levantamento dos Requisitos

A primeira parte da pesquisa foi concebida por meio do caráter exploratório (WAZLAWICK, 2009), com o objetivo de elucidar as necessidades, reunir os conceitos necessários para a implementação do projeto e firmar o escopo do desenvolvimento funcional da biblioteca. Esse procedimento ocorreu em duas etapas; na primeira foi um estudo sobre a API JPA com uma maior exploração na implementação das Criteria Queries. Sendo assim, após o levantamento as ações esperadas da biblioteca incluem:

- estudo do funcionamento das soluções de persistência suportadas pela linguagem Java por

meio da API JPA;

- identificação das oportunidades de pesquisa por meio da criação de extensões no contexto da API JPA; e
- estudo do funcionamento das Criteria Queries;

3.2.2 Análise e Projeto

A segunda parte da pesquisa continua o caráter exploratório por meio projeto das extensões das Criteria Queries seguindo o padrão da API JPA. Além disso, também foram especificados os algoritmos de implementação das extensões. A segunda fase engloba a especificação das classes concretas que implementaram Criteria Queries na API JPA, técnicas de implementações dos métodos abstratos etc.

3.2.3 Tecnologias

A terceira parte da pesquisa teve caráter descritivo, onde foram estabelecidos o Ambiente Integrado de Desenvolvimento (do inglês, *Integrated Development Environment* (IDE)) de desenvolvimento Java e também o SGBD PostgreSQL[1] que irá persistir os dados em modelo relacional.

3.2.4 Implementação

A quarta parte do trabalho caracteriza a implementação da pesquisa descrita na produção de uma biblioteca Java que possa ser utilizada em qualquer projeto Java que necessite fazer persistência ORM. As etapas de execução descrevem o processo de realização das interfaces da API JPA e técnicas de programação dos algoritmos nos métodos abstratos, onde classes concretas e métodos abstratos foram identificados na etapa de Análise e Projeto.

3.2.5 Validação

A última parte do trabalho ficou dedicada a validar a biblioteca desenvolvida no intuito de verificar possíveis limitações do projeto e também suas vantagens. A validação se divide em duas ações específicas:

[1] PostgreSQL. PostgreSQL: The World's Most Advanced Open Source Relational Database. Disponível em: <https://www.postgresql.org/>. Acesso em: 13 de janeiro de 2022.

- a apresentação, por meio de um exemplo, do desenvolvimento de uma aplicação, em nível técnico, para que desenvolvedores possam se familiarizar com os aspectos de implementação e uso da biblioteca; e
- a realização de uma análise de desempenho, baseada na métrica tempo de resposta, entre alguns métodos de consulta similares com o proposto neste trabalho.

4 RESULTADOS

Este capítulo apresenta os resultados alcançados no trabalho, destacando a biblioteca desenvolvida e a validação da biblioteca em duas perspectiva, complexidade de desenvolvimento por meio do número de linhas e desempenho por meio do tempo de resposta das consultas.

O código fonte da biblioteca desenvolvida pode ser encontrado no github através do link: https://github.com/andreldsr/criteriaresolver

4.1 Biblioteca Criteria Resolver e sua Utilização

A biblioteca utiliza de *reflection* (LI *et al.*, 2019), recurso do Java que permite executar métodos e acessar atributos de uma classe em tempo de execução, para determinar tanto os termos da consulta quanto o retorno desejado para a mesma. Para controlar todas as operações geradas foi criado um repositório genérico com todos os métodos necessários para a criação das consultas, a classe abstrata `CriteriaResolverBaseRepository`, que recebe em sua implementação um tipo genérico que definirá a entidade a ser consultada.

Código-fonte 1 – Assinatura da classe abstrata do repositório base

```
1 public abstract class CriteriaResolverBaseRepository<T>
```

Em seu construtor, esse repositório deve receber um `EntityManager`, responsável pela conexão com o Banco de Dados e geração de todas as consultas. O seu construtor também é responsável por capturar a classe da entidade genérica e guardá-la para criação das *queries* posteriormente.

Código-fonte 2 – Construtor da classe abstrata do repositório base

```
1 public CriteriaResolverBaseRepository(EntityManager
     entityManager){
2     ParameterizedType genericSuperclass = (
         ParameterizedType) getClass().getGenericSuperclass()
         ;
3     this.c = (Class<T>) genericSuperclass.
         getActualTypeArguments()[0];
```

```
4       this.em = entityManager;
5       joinMap = new HashMap<>();
6   }
```

As consultas são geradas programaticamente a partir de um objeto de consulta construído pelo desenvolvedor que estenda da classe abstrata `SearchObject`, decisão tomada para que as classes usadas para a geração das consultas sejam específicas para tal função, evitando assim o reaproveitamento das próprias entidades.

Código-fonte 3 – Classe abstrata do objeto de consulta

```
1  public abstract class SearchObject {
2      private Map<String, JoinType> joins = new HashMap<>();
3      public Map<String, JoinType> getJoins() {
4          return joins;
5      }
6      public void createJoins() { }
7      public SearchObject(){
8          this.createJoins();
9      }
10 }
```

A propriedade ou atributo de instância *joins* e o método `createJoins()` são relacionados aos tipos diferentes de *joins* feitos nas consultas e serão mencionados novamente quando descrita essa parte da geração da consulta. Uma classe criada para a consulta estendendo a classe abstrata `SearchObject` deve ter suas propriedades anotadas com a anotação criada para esta biblioteca `@CriteriaField`.

Código-fonte 4 – Anotação para os campos da classe que estenda SearchObject

```
1  @Target(ElementType.FIELD)
2  @Retention(RetentionPolicy.RUNTIME)
3  @Inherited
4  public @interface CriteriaField {
```

```
5       String fieldName() default "";
6       ComparationType comparationType() default
           ComparationType.EQUALS;
7       enum ComparationType {
8           EQUALS, LIKE, GREATER_THAN, LESS_THAN,
               GREATER_EQUALS, LESS_EQUALS, IN, NOT_IN,
               DIFFERENT, STARTS_WITH, ENDS_WITH
9       }
10  }
```

Esta anotação possui dois campos opcionais com valor padrão, que serão usados na construção da consulta pelo repositório. O campo `fieldName` define o nome do campo da entidade a ser consultada que deve ser utilizado nesta comparação. O campo `comparationType` define o tipo de consulta que deverá ser realizado com essa propriedade da entidade.

A consulta será realizada comparando o campo da entidade com o nome definido, utilizando o tipo de comparação definido pela propriedade, que deverá ser preenchida com um valor presente na enumeração `ComparationType` e será adicionada na consulta apenas se a propriedade anotada do objeto de consulta tiver um valor preenchido diferente de nulo.

O valor padrão de campo `fieldName` é vazio e será substituído pelo nome da propriedade anotada. O valor padrão do campo `comparationType` é `ComparationType.EQUALS` que define uma comparação de igualdade a ser realizada, caso o campo anotado esteja preenchido. Considerando as entidades de exemplo:

Código-fonte 5 – Entidade de exemplo para a consulta representando categoria

```
1   @Entity
2   public class Category {
3       @Id
4       @GeneratedValue(strategy = GenerationType.IDENTITY)
5       private Long id;
6       private String name;
7       public Category(String name){
8           this.name = name;
```

```
9       }
10  }
```

Código-fonte 6 – Entidade de exemplo para consulta representando produto

```
1   @Entity
2   public class Product {
3       @Id
4       @GeneratedValue(strategy = GenerationType.IDENTITY)
5       private Long id;
6       private String name;
7       private String description;
8       private Double price;
9       @ManyToOne
10      private Category category;
11  }
```

A classe de consulta a seguir permite gerar consultas da entidade `Product` utilizando critério `LIKE` para a propriedade `name`, `LIKE` para a propriedade `description`, maior igual e menor igual ou exatamente igual para a propriedade `price`, além de permitir consultar com o critério `LIKE` para a propriedade `name` do objeto relacionado `Category`, fazendo automaticamente a cláusula `JOIN` necessária para a consulta. Será usado na geração da consulta apenas os campos preenchidos com valor diferente de nulo.

Código-fonte 7 – Classe que representa objeto de consulta para a entidade produto

```
1   public class ProductSearchObject extends SearchObject {
2       @CriteriaField(comparationType = CriteriaField.
            ComparationType.LIKE)
3       private String name;
4       @CriteriaField(comparationType = CriteriaField.
            ComparationType.LIKE)
5       private String description;
```

```
6       @CriteriaField(fieldName = "price", comparationType =
            CriteriaField.ComparationType.GREATER_EQUALS)
7       private Double minPrice;
8       @CriteriaField(fieldName = "price", comparationType =
            CriteriaField.ComparationType.LESS_EQUALS)
9       private Double maxPrice;
10      @CriteriaField(fieldName = "price")
11      private Double exactPrice;
12      @CriteriaField(fieldName = "category.name",
            comparationType = CriteriaField.ComparationType.LIKE
            )
13      private String category;
14  }
```

Assim, para gerar a consulta basta chamar um método de consulta do repositório genérico passando o objeto da consulta com os campos desejados preenchidos. O repositório base irá varrer os campos do objeto de consulta utilizando de *reflection* para verificar quais campos estão preenchidos e utilizá-los para gerar os argumentos da consulta com as informações trazidas pela anotação `@CriteriaField` presente em cada campo para escolher o nome do campo e o tipo da comparação a ser gerada.

Código-fonte 8 – Método que utiliza o repositório do CriteriaResolver para gerar a consulta

```
1   public List<Product> getProductByCriteria(
        ProductSearchObject productSearchObject){
2       return productCriteriaRepository.getResultList(
            productSearchObject);
3   }
```

Utilizando apenas este método todas as consultas possíveis das combinações de campos presentes no objeto de consulta já estão disponíveis, sendo necessário preencher apenas os campos desejados, que serão utilizados na criação da consulta.

No caso desta consulta, se o objeto tiver seu campo `category` preenchido, este

irá ser utilizado na consulta com o `fieldName` “category.name” o qual representa uma união entre `Product` e `Category`, visto que a classe `Product` possui uma propriedade `category` que aponta para esta outra classe, buscando assim por uma propriedade desta segunda entidade. Esta união será gerada também automaticamente sem nenhum problema como um `INNER JOIN`. A forma que essas uniões são feitas pode ser alterada isoladamente através da propriedade *joins* presente na classe `SearchObject`, tanto antes de invocar o método caso seja algo único daquela consulta, como sobrescrevendo o método `createJoins()` para que ocorra em todas as consultas realizadas com aquele `SearchObject`.

Código-fonte 9 – Exemplo de sobrescrita do método createJoins

```
1 @Override
2 public void createJoins() {
3     this.getJoins().put("entidade", JoinType.LEFT);
4 }
```

Ao adicionar ao mapa o tipo de *join* desejado, dentre as opções presentes na enumeração `JoinType`, sendo a chave o nome da entidade que se deseja alterar o tipo da união, o repositório ao gerar a consulta levará esse tipo de *join* como prioridade sobre a forma padrão.

Comparando a uma consulta normal, esse método economiza tempo na hora de alterações na consulta desejada e diminui a quantidade de código envolvida. Tendo como exemplo uma consulta de produto por nome, faixa de preço e nome da categoria, poderíamos escrevê-la com a própria linguagem da JPA, a JPQL, da seguinte forma.

Código-fonte 10 – Consulta realizada com JPQL

```
1 SELECT p FROM Product p INNER JOIN p.category c WHERE p.
    name = :name AND p.price BETWEEN (:minPrice AND :
    maxPrice) AND c.name = :categoryName
```

Porém esta consulta está limitada a apenas esta possibilidade, sendo assim não podendo ser utilizada para as outras possibilidades com campos opcionais, além de necessitar de grandes conhecimentos de JPQL para sua alteração e de seu crescimento exponencial com a adição de novos campos na consulta.

A criação de consulta com campos opcionais utilizando JPQL se utiliza de comparações extras do parâmetro que pode ser pesquisado com o valor nulo, para permitir que a consulta não tenha seu resultado alterado pelo não preenchimento de algum valor.

Código-fonte 11 – Consulta realizada com JPQL e campos opcionais

```
1  SELECT p FROM Product p WHERE (:productName = null or
2  p.name = :productName) AND (:productMinPrice = null or
3  p.price >= :productMinPrice) AND (:productMaxPrice = null
       or p.price <= :productMaxPrice)
```

O aumento da complexidade no código é visível e a legibilidade do mesmo é prejudicada pelo número de verificações de valor nulo.

Utilizando `Criteria` para gerar a mesma consulta podemos ver maior flexibilidade podendo deixar os campos opcionais utilizando-se de estruturas condicionais do tipo `if` para verificar a presença do campo antes de adicionar a cláusula a consulta.

Código-fonte 12 – Consulta realizada com Criteria

```
1  Criteria crit = session.createCriteria(Product.class);
2  if(name != null) {
3      crit.add(Restrictions.eq("name", name));
4  }
5  if(minPrice != null) {
6      crit.add(Restrictions.gt("price", minPrice));
7  }
8  if(maxPrice != null) {
9      crit.add(Restrictions.lt("price", maxPrice));
10 }
11 if(categoryName != null) {
12     crit.addAlias("category", "c");
13     crit.add(Restrictions.eq("c.name", categoryName));
14 }
15 List results = crit.list();
```

Porém é fácil perceber o quanto o código vai crescer com a adição de novos campos para esta consulta, além da repetição de código para cada propriedade consultada, que possui quase nenhuma alteração para os outros.

Além de controlar os critérios da consulta, a biblioteca também permite escolher facilmente os campos retornados a partir de um segundo objeto, que será aqui chamado de objeto de retorno. O objeto de retorno não precisa estender uma classe específica, seus campos serão lidos através de *reflection* também e através do nome das propriedades a projeção será criada. Caso o nome do campo do objeto de resposta seja diferente da propriedade da entidade consultada, ou caso exista a necessidade de realizar um *join* para obter tal informação este campos deve ser anotado com a anotação criada para a biblioteca `@ProjectionField`.

Código-fonte 13 – Anotação da biblioteca @ProjectionField

```
1 @Target(ElementType.FIELD)
2 @Retention(RetentionPolicy.RUNTIME)
3 @Inherited
4 public @interface ProjectionField {
5     String projectionPath();
6 }
```

Código-fonte 14 – Exemplo de objeto de retorno

```
1 public class ProductDTO {
2     @ProjectionField(projectionPath = "name")
3     private String productName;
4     private Double price;
5     @ProjectionField(projectionPath = "category.name")
6     private String categoryName;
7 }
```

A classe presente no exemplo, `ProductDTO`, possui 3 campos que serão usados para construir o retorno da consulta, diminuindo assim a quantidade de dados trafegados desde o banco de dados, evitando a sobrecarga com dados desnecessários e trazendo apenas o necessário para o desenvolvedor. Através de *reflection* sobre esses campos o retorno gerado trará o campo

`name` e `price` da entidade `Product` e trará também o campo `name` da entidade `Category`, já realizando a união necessária para trazer esta informação, caso já não seja feita anteriormente pelo objeto de consulta.

Código-fonte 15 – Exemplo de consulta gerada utilizando um objeto de consulta e um objeto de retorno.

```
SELECT
        product0_.name as col_0_0_,
        product0_.price as col_1_0_,
        category1_.name as col_2_0_
FROM
        product product0_
CROSS JOIN
        category category1_
WHERE
        product0_.category_id = category1_.id
        and (product0_.description like ?)
        and product0_.price = 10;
```

Para utilizar o objeto de retorno com o repositório genérico basta utilizar o método `getGenericQuery` passando como parâmetros o objeto de consulta e a classe desejada como resposta. O método retorna um objeto `Query` que pode ser manipulado antes de buscar a resposta.

Código-fonte 16 – Exemplo de chamada com objeto de consulta e retorno.

```
public List<ProductDTO> getProductByCriteria(
   ProductSearchObject productSearchObject){
    return productCriteriaRepository.getGenericQuery(
       productSearchObject, ProductDTO.class).getResultList
       ();
}
```

4.2 Análise do Tempo de Resposta

Para analisar as diferenças no desempenho entre os métodos de consulta foi criado um projeto Spring contendo uma entidade `Product` que será consultada em uma base de dados com mil registros e os seus tempos de execução serão comparados para verificar se a utilização dessa biblioteca afeta de forma significativa as pesquisas do projeto.

Uma classe de testes possuindo os repositórios para consultar a classe `Product` através do *Hibernate Query Language* (HQL), Spring JPA, API JPA Criteria e utilizando a biblioteca possui métodos de teste isolados que serão chamados separadamente e terão seus tempos de execução registrados e comparados através da média de 5 execuções de consultas de cada método.

A consulta realizada utiliza os parâmetros nome do produto, preço mínimo e preço máximo de forma opcional, podendo ser preenchidos ou não para alterar o resultado retornado. Dessa forma todas recebem os mesmos parâmetros, nulos ou preenchidos, para retornar a mesma lista resultante.

No teste realizado a consulta construída com HQL foi a mais rápida com média de tempo de 225,2ms de execução no método de teste.

Código-fonte 17 – Consulta construída com HQL

```
1  public List<Product> findByNameAndPriceBetweenOptional(
       String name, Double minPrice, Double maxPrice) {
2      String hql = "SELECT p FROM Product p WHERE
3    (:name = null or p.name = :name) " +
4    "AND (:minPrice = null or p.price >= :minPrice) " +
5    "AND (:maxPrice = null or p.price <= :maxPrice)";
6
7    TypedQuery<Product> query = em.createQuery(hql, Product.
       class);
8    query.setParameter("name", name);
9    query.setParameter("minPrice", minPrice);
10   query.setParameter("maxPrice", maxPrice);
11   return query.getResultList();
12 }
```

Para a realização da mesma consulta com Spring JPA é necessário utilizar da anotação `@Query` e construir a consulta de maneira próxima ao método com HQL, porém ganhando no quesito organização e facilidade de leitura. A velocidade da consulta no entanto foi um pouco inferior ao encontrado anteriormente sendo de 240ms a média de execução.

Código-fonte 18 – Consulta construída com Spring JPA

```
@Query("SELECT p FROM Product p WHERE
    (:productName = null or p.name = :productName) " +
    "AND (:productMinPrice = null or p.price >= :
       productMinPrice) " +
    "AND (:productMaxPrice = null or p.price <= :
       productMaxPrice)")
List<Product> findByNameAndPriceBetweenOptional(String
   productName, Double productMinPrice, Double
   productMaxPrice);
```

A construção dessa consulta com a API JPA Criteria começa a ficar um pouco mais verbosa pela adição de condicionais para cada campo opcional da consulta. O tempo de execução é um pouco superior comparado ao encontrado com HQL, porém um pouco mais rápido que com Spring JPA, sendo de 231,2ms.

Código-fonte 19 – Consulta construída com Criteria API

```
public List<Product> findByNameAndPriceBetweenOptional(
   String name, Double minPrice, Double maxPrice) {
CriteriaBuilder criteriaBuilder = em.getCriteriaBuilder();
CriteriaQuery<Product> query = criteriaBuilder.createQuery(
   Product.class);
Root<Product> root = query.from(Product.class);
List<Predicate> predicates = new ArrayList<>();
if(name != null)
    predicates.add(criteriaBuilder.equal(root.get("name"),
```

```
        name));
if(minPrice != null)
    predicates.add(criteriaBuilder.ge(root.get("price"),
        minPrice));
if(maxPrice != null)
    predicates.add(criteriaBuilder.le(root.get("price"),
        maxPrice));
query.where(predicates.toArray(new Predicate[0]));
return em.createQuery(query).getResultList();
}
```

Para realizar a mesma consulta utilizando a biblioteca desenvolvida é necessário apenas a criação de uma classe que represente os campos opcionais desejados e invocar o método já criado dentro do repositório.

Código-fonte 20 – Objeto de consulta de Product

```
public class ProductSearchObject extends SearchObject {
    @CriteriaField
    private String name;
    @CriteriaField(fieldName = "price", comparationType =
        CriteriaField.ComparationType.GREATER_EQUALS)
    private Double minPrice;
    @CriteriaField(fieldName = "price", comparationType =
        CriteriaField.ComparationType.LESS_EQUALS)
    private Double maxPrice;
}
```

Código-fonte 21 – Invocação da consulta de Product

```
List<Product> all = productCriteriaResolverRepository.
   getResultList(productSearchObject);
```

E com esse método de consulta a média no tempo de execução foi bem próxima a encontrada com a API JPA Criteria sendo de 231,8ms. A Tabela 1 sumariza os resultados obtidos após a um número de execuções de consulta para cada estratégia utilizada na comparação.

Tabela 1 – Sumário dos Tempos de Resposta do Número de Execução de Consultas por Estratégia

Estratégia	1 Execução	2 Execuções	3 Execuções	4 Execuções	5 Execuções	Média de Execuções
HQL	223ms	231ms	215ms	215ms	242ms	225,2ms
Spring JPA	293ms	210ms	246ms	231ms	220ms	240ms
API JPA Criteria	235ms	234ms	234ms	234ms	219ms	231,2ms
Criteria Resolver	245ms	219ms	219ms	217ms	259ms	231,8ms

Por meio da Tabela 1 e levando em consideração a média de execução, pode-se perceber que as médias possuem valores similares. O HQL foi a estratégia que teve um menor tempo de resposta em média e Spring JPA obteve o maior tempo de resposta em média.

Todas as consultas foram realizadas em um único banco PostgreSQL rodando via imagem docker com uma tabela com 1000 registros. As medições foram feitas com a ferramenta de testes da IDE Intellij IDEA Community Version 2021.3.3.

5 CONCLUSÃO E TRABALHOS FUTUROS

Este trabalho apresentou o desenvolvimento de uma biblioteca no intuito de facilitar o uso de JPA Criteria Queries com vários campos opcionais e projeções em sua resposta. Para isso, foi feito um estudo sobre a linguagem e plataforma Java, tecnologias de persistência de dados com Java com foco na estratégia ORM em bancos de dados relacionais. Após, foi feito uma análise e projeto para implementação da biblioteca. Em seguida, a biblioteca foi implementada e, por fim, avaliada na perspectiva do número de código utilizado para o uso e o tempo de respostas das consultas.

Por meio dos resultados obtidos, foi possível validar as duas hipóteses. A primeira hipótese consistia em verificar se a biblioteca conseguiria diminuir o volume de código produzido. No Capítulo 4 foi possível observar que o projeto da biblioteca, conseguiu simplificar a codificação e, como consequência, diminuição do código produzido. Já a segunda hipótese consistia em verificar o tempo de respostas das consultas. Também no Capítulo 4 foi possível notar que os tempos de resposta ficam dentro dos limites de outras estratégias relacionadas.

Como trabalhos futuros, almeja-se:

- fazer uma análise minuciosa do código da biblioteca por meio de testes funcionais, análise de desempenho e outras técnicas de inspeção de código;
- realizar uma avaliação da biblioteca com um conjunto maior de consultas e bancos de dados mais volumosos no intuito de verificar o impacto do desempenho da biblioteca em tais cenários;
- comparar a biblioteca com outras estratégias JPA e de persistência no âmbito Java; e
- melhorar a documentação do código da biblioteca para que seja possível uma maior facilidade de interpretação e entendimento por parte dos desenvolvedores.
- analisar o código gerado junto a um grupo focal de desenvolvedores e avaliar as reações geradas.

REFERÊNCIAS

CAELUM. **Java para Desenvolvimento Web**. 2021. Disponível em: https://www.caelum.com.br/apostila/apostila-java-web.pdf. Acessado em 22 de novembro de 2021.

CAELUM. **Uma Introdução Prática ao JPA com Hibernate**. 2021. Disponível em: https://www.caelum.com.br/apostila-java-web/uma-introducao-pratica-ao-jpa-com-hibernatejava-persistence-api-e-frameworks-orm. Acessado em 16 de agosto de 2021.

DEVMEDIA. **Os 4 pilares da Programação Orientada a Objetos**. 2014. Disponível em: https://www.devmedia.com.br/os-4-pilares-da-programacao-orientada-a-objetos/9264. Acessado em 22 de novembro de 2021.

ELMASRI, R.; NAVATHE, S. B. **Sistemas de Banco de Dados**. 6a.. ed. [S.l.]: Pearson Education Brasil, 2011. ISBN 9788579360855.

FARIAS, F. A. d. M. **Avaliação de Desempenho entre Bancos de Dados Relacionais e NoSQL**. 2014. Monografia de Graduação. Bacharel em Sistemas de Informação, Centro de Ciências Aplicadas à Educação, Universidade Federal da Paraíba, Rio Tinto, Brasil.

HENRY, K. Objective viewpoint: Jdbc- java database connectivity. **XRDS**, Association for Computing Machinery, New York, NY, USA, v. 7, n. 3, p. 3–ff., mar 2001. ISSN 1528-4972. Disponível em: <https://doi.org/10.1145/367884.367889>.

JETBRAINS. **The State of Developer Ecosystem 2021**. 2021. Disponível em: https://www.jetbrains.com/lp/devecosystem-2021/. Acessado em 16 de agosto de 2021.

JUNIOR, P. J. **Java Guia do Programador - 4a Edição: Atualizado para Java 16**. 4a.. ed. São Paulo: Novatec Editora, 2021. ISBN 9786586057584. Disponível em: <https://books.google.com.br/books?id=6hcuEAAAQBAJ>.

LI, Y.; TAN, T.; XUE, J. Understanding and analyzing java reflection. **ACM Trans. Softw. Eng. Methodol.**, Association for Computing Machinery, New York, NY, USA, v. 28, n. 2, feb 2019. ISSN 1049-331X. Disponível em: <https://doi.org/10.1145/3295739>.

MARCONI, M. d. A.; LAKATOS, E. M. **Fundamentos de Metodologia Científica**. 5a.. ed. São Paulo: Atlas, 2003. ISBN 85-224-3397-6.

MARINHO, C. S. S.; MOREIRA, L. O.; COUTINHO, E. F.; FILHO, J. S. C.; SOUSA, F. R. C.; MACHADO, J. C. Labareda: A predictive and elastic load balancing service for cloud-replicated databases. **Journal of Information and Data Management**, v. 9, n. 1, p. 94–106, Jun. 2018. Disponível em: <https://sol.sbc.org.br/journals/index.php/jidm/article/view/1639>.

MIGLER, A.; DEKHTYAR, A. Mapping the sql learning process in introductory database courses. In: ____. **Proceedings of the 51st ACM Technical Symposium on Computer Science Education**. New York, NY, USA: Association for Computing Machinery, 2020. p. 619–625. ISBN 9781450367936. Disponível em: <https://doi.org/10.1145/3328778.3366869>.

NETO, A. A. C.; OLIVEIRA, L. Chagas de; OLIVEIRA, A. C. M.; LEMOS, D. Barreiro de. HIBERNATE E JPA: CONCEITOS PARA UTILIZAÇÃO. In: **XIV CEEL Conferência de**

Estudos em Engenharia Elétrica. Uberlândia, Minas Gerais, Brasil: Universidade Federal de Uberlândia (UFU), 2016. (XIV CEEL), p. 1–5.

OLIVEIRA, F. G.; SEABRA, J. M. P. Metodologias de desenvolvimento de software: Uma análise no desenvolvimento de sistemas na web. **Periódico Científico Tecnologias em Projeção**, v. 6, n. 1, p. 20–34, 2015. ISSN 2178-6267.

SANTOS, M. R. **Sistema para Informatização de Comércio em Java**. 2018. Disponível em: https://semanaacademica.org.br/system/files/artigos/tcc_artigo.pdf. Acessado em 22 de novembro de 2021.

SOMMERVILLE, I. **Engenharia de Software**. 9a.. ed. [S.l.]: Pearson Education do Brasil, 2011. ISBN 9788579361081.

TIOBE. **TIOBE - The Software Quality Company**. 2022. Disponível em: https://www.tiobe.com/tiobe-index/. Acessado em 07 de janeiro de 2022.

WAZLAWICK, R. S. **Metodologia de Pesquisa para Ciência da Computação**. 1a.. ed. Rio de Janeiro: Elsevier Editora, 2009. ISBN 978-85-352-3522-7.

YUAN, G.; LU, J.; ZHANG, S.; YAN, Z. Storing multi-model data in rdbmss based on reinforcement learning. In: ____. **Proceedings of the 30th ACM International Conference on Information amp; Knowledge Management**. New York, NY, USA: Association for Computing Machinery, 2021. p. 3608–3611. ISBN 9781450384469. Disponível em: <https://doi.org/10.1145/3459637.3482191>.