UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE SISTEMAS DE INFORMAÇÃO – BACHARELADO

PROTÓTIPO DE UM GERADOR DE APLICAÇÕES WEB COM JHIPSTER

INGMAR SCHMIDT DE AGUIAR

BLUMENAU
2019

INGMAR SCHMIDT DE AGUIAR

PROTÓTIPO DE UM GERADOR DE APLICAÇÕES WEB COM JHIPSTER

Trabalho de Conclusão de Curso apresentado ao curso de graduação em Sistemas de Informação do Centro de Ciências Exatas e Naturais da Universidade Regional de Blumenau como requisito parcial para a obtenção do grau de Bacharel em Sistemas de Informação.

Prof. Mauro Marcelo Mattos, Dr.Eng. - Orientador

BLUMENAU
2019

PROTÓTIPO DE UM GERADOR DE APLICAÇÕES WEB COM JHIPSTER

Por

INGMAR SCHMIDT DE AGUIAR

Trabalho de Conclusão de Curso aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II pela banca examinadora formada por:

Presidente:	Prof(a). Mauro Marcelo Mattos – Orientador(a), FURB
Membro:	Prof(a). Luciana Pereira de Araújo Kohler – FURB
Membro:	Prof(a). Everaldo Artur Grahl – FURB

Blumenau, 11 de dezembro de 2019

Dedico este trabalho a minha esposa Heloísa, pois ela é minha grande inspiração.

AGRADECIMENTOS

Em primeiro lugar agradeço a Deus, que me criou a sua imagem e semelhança, consumador de todas as coisas.

Agradeço imensamente a minha esposa Heloísa por sua paciência e incentivo, sem ela este trabalho não existiria.

Ao meu orientador, professor Mauro Marcelo Mattos, Dr. Eng, não poderia ter um orientador melhor, sem ele este trabalho também não existiria.

Aos meus amigos e familiares pelas palavras de incentivo e muitas orações.

Ao meu colega de trabalho Hendry Steffens, que me emprestou o notebook pra fazer parte deste trabalho.

O que fazemos em vida ecoa pela eternidade.

Gladiador

RESUMO

O presente trabalho descreve a construção de um protótipo de uma ferramenta destinada a facilitar o desenvolvimento de aplicações web utilizando a tecnologia JHipster a partir de banco de dados já existentes. Especificamente foi concebido e implementado um modelo de mapeamento da estrutura de bancos de dados existentes para a metalinguagem JDL do JHipster e desenvolvida uma API de acesso às funcionalidades disponibilizadas pelo JHipster. A ferramenta conecta no banco, lê o modelo de dados, disponibiliza uma área de ajustes dos relacionamentos e produz os artefatos necessários para que o JHipster produza as entidades e relacionamentos necessários e crie uma aplicação web. Para a validação do protótipo foram realizados testes com dois modelos de entidade relacionamento aplicados ao banco de dados, que foram processados e convertidos para a metalinguagem do JHipster. Um projeto base do JHipster foi usado para validar o código gerado e os resultados demonstraram que o trabalho atendeu aos objetivos propostos.

Palavras-chave: JHipster. Protótipo. Ferramenta. Desenvolvimento. Aplicação. API. Código gerado. Banco de dados.

ABSTRACT

This work describes the construction of a prototype tool designed to facilitate the development of web applications using JHipster technology from existing databases. Specifically, an existing database structure mapping model for the JHipster JDL Metalanguage was designed and implemented, and an API for accessing features provided by JHipster was developed. The tool connects to the database, reads the data model, provides a relationship settings area, and produces the artifacts required for JHipster to produce the required entities and relationships and create a web application. To validate the prototype, tests were performed with two relationship entity models applied to the database, which were processed and converted to JHipster metalanguage. A JHipster base project was used to validate the generated code and the results showed that the work met the proposed objectives.

Key-words: JHipster. Prototype. Tool. Development. Application. API. Generated code. Database.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE ABREVIATURAS E SIGLAS

API - Application Programming Interface

CASE - Computer-Aided Software Engineering

CGI - Common Gateway Interface

CRUD - Acrônimo para Create, Read, Update e Delete

DML - Data Manipulation Language ou

DSL - Linguagens de Domínio Específico

IDE - Integrated Development Environment

JDBC - Java Database Connectivity

JDK - Java Development Kit

JDL - JHipster Domain Language

JPA – Java Persistence API

JSP - Java Server Pages

MDD - Model-Driven Development

MVC - Model-View-Controller

NPM – Node Package Manager

SGBD - Sistema Gerenciador de Banco de Dados

SQL - Structured Query Language

SWT - Standard Widget Toolkit

UML - Unified Modeling Language

SUMÁRIO

1 INTRODUÇÃO

O mercado de software tem apresentado grande evolução nos últimos anos impulsionado pela mudança nos paradigmas de programação paralelamente ao surgimento de novas tecnologias (SILVA NETO; VILAR, 2018). O crescente destaque da computação em nuvem desperta nos fornecedores de software o interesse em ganhar posição competitiva, bem como motiva os clientes a mudarem suas estratégias de TI para a nuvem (ROGERS, 2017). Por sua vez, a crescente demanda por software no mercado, traz como consequência uma pressão cada vez maior na equipe de desenvolvimento no sentido de realizar entregas cada vez mais rápidas num ambiente em que a multiplicidade de tecnologias torna o processo de desenvolvimento cada vez mais complexo (SHINDE; SUN, 2016).

> As regras de negócios mudaram. Em todos os setores de atividade, a difusão de novas tecnologias digitais e o surgimento de novas ameaças disruptivas estão transformando modelos e processos de negócios. A revolução digital está virando de cabeça para baixo o velho guia de negócios. (ROGERS, 2017, p. 11).

A transformação digital é o que acontece com as organizações quando elas adotam formas novas e inovadoras de fazer negócios com base em avanços tecnológicos e que "a infraestrutura de TI é a principal responsável pela transformação digital" (RED HAT, 2017). Neste contexto, criar ou redesenhar uma aplicação web, independente da tecnologia, pode não ser uma tarefa tão simples (GONÇALVES, 2018). Como forma de encontrar um caminho de solução para esta questão, ao longo do tempo, diversas tecnologias foram desenvolvidas com o propósito de facilitar o processo de desenvolvimento deste tipo de aplicação.

Conforme Lima Junior (2013), a partir do momento em que se percebeu o potencial de armazenamento de conteúdos para web para além das estratégias de extração, armazenamento e visualização de dados (por exemplo, JavaScript, CSS, etc.) com a introdução da tecnologia Common Gateway Interface (CGI) em 1993, a complexidade dos requisitos tecnológicos necessários para o desenvolvimento de software web aumentou significativamente. Nesta esteira, surgiram tecnologias que pretendem facilitar o processo de desenvolvimento de software para web, tais como: AndroMDA, Celerio, JHipster, ModelJ, Sculptor Framework, Crud-Admin-Generator, CrudKit, JPA-Modeler Plugin e Spring Roo (LIVRAGHI, 2016) (LIMA, 2016).

Assim sendo, este trabalho utiliza a tecnologia JHipster, a qual combina três *frameworks* de desenvolvimento web: Bootstrap, Angular e Spring Boot. Com ela é possível automatizar o processo de criação de uma aplicação web completa a partir de comandos emitidos em um terminal. Se por um lado o uso de ferramentas como o JHipster proporciona um ganho na produtividade, auxiliando na fase inicial de criação do projeto, bem como na prototipação do

sistema, por outro, a utilização do recurso de linha de comando introduz uma complexidade desnecessária.

Diante do exposto, o presente trabalho desenvolveu um protótipo de uma ferramenta que facilita o processo de geração de uma aplicação web baseada no JHipster através da importação dos modelos de tabelas e relações a partir de bases de dados existentes.

1.1 OBJETIVOS

O objetivo deste trabalho é desenvolver um protótipo de uma ferramenta de geração automática de uma aplicação web a partir da importação de modelos de bases de dados já existentes.

Os objetivos específicos são:

a) conceber e implementar um modelo de mapeamento da estrutura de bancos de dados existentes para o modelo de estrutura JHipster;
b) desenvolver uma API de acesso às funcionalidades disponibilizadas pelo JHipster;
c) construir um conjunto de casos de testes para validar o projeto.

1.2 ESTRUTURA

Este trabalho está disposto em quatro capítulos, sendo este o primeiro, onde é apresentada uma introdução ao assunto abordado, os objetivos definidos e a estrutura do trabalho.

No capítulo 2 é descrita a fundamentação teórica sobre geração de código baseada na técnica de *scaffolding*, CRUD, Linguagem de Domínio Específico, Jhispter e trabalhos correlatos.

O capítulo 3 apresenta o desenvolvimento do protótipo, iniciando com o levantamento de requisitos funcionais e não funcionais, especificação, implementação e por fim resultados e discussões.

No capítulo 4 são apresentadas as conclusões deste trabalho, bem como extensões para trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo tem por objetivo apresentar três dos principais assuntos que estão relacionados com o trabalho proposto. A seção 2.1 aborda sobre geração de código baseada na técnica de *scaffolding*, enquanto na seção 2.2 é apresentada uma introdução ao JHipster e finalmente a seção 2.3 apresenta os trabalhos correlatos.

2.1 GERAÇÃO DE CÓDIGO BASEADA NA TÉCNICA DE SCAFFOLDING

Geradores de código são mecanismos auxiliares ao processo de criação de software com objetivo de reduzir tempo e custos de desenvolvimento, bem como ganho de produtividade. A partir de uma entrada predefinida são gerados arquivos fontes de uma aplicação completa ou artefatos específicos. O uso de geradores de código é empregado principalmente na ausência de tempo para o desenvolvimento e/ou quando a equipe para a implementação do projeto é pequena (KLUG, 2007, p.18).

Segundo Herrington (2003, p. 3, tradução nossa), "geração de código trata sobre escrever programas que escrevem programas". Em outras palavras, a geração de código é uma técnica para se construir código utilizando um ou mais programas (KLUG, 2007). As técnicas de geração de código fornecem benefícios substanciais aos engenheiros de software em todos os níveis. Esses benefícios incluem (HERRINGTON, 2003, p. 15, tradução nossa):

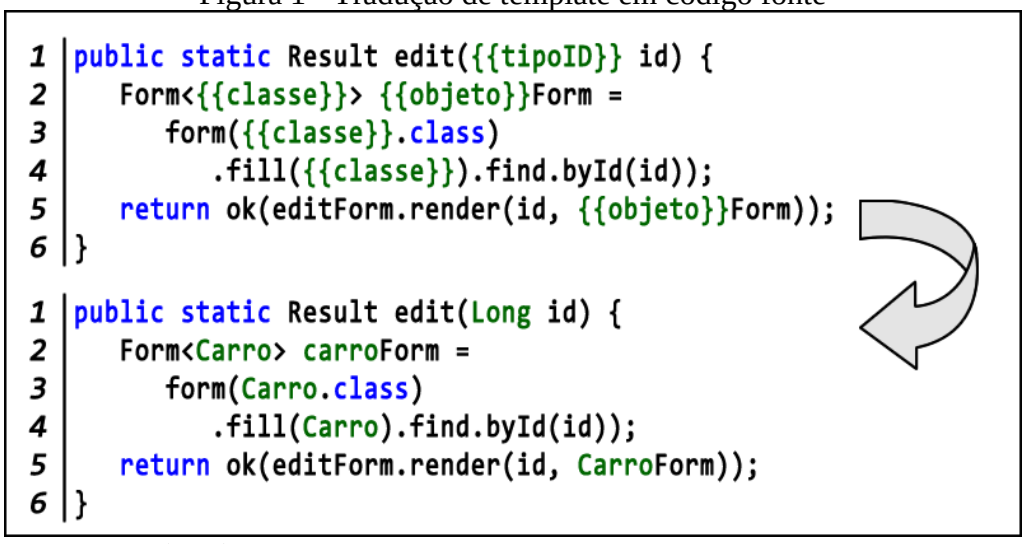
a) qualidade e consistência: a geração de código a partir de modelos cria instantaneamente uma base de código mais consistente, consequentemente com mais qualidade;
b) único ponto de conhecimento: usando um gerador de tabelas de banco de dados como exemplo, um código manual, para alterar o nome de uma tabela, seria necessário alterar todas os códigos que possuam tal tabela manualmente, com um gerador de código basta alterar num ponto e o restante é fica sob sua responsabilidade;
c) mais tempo para o projeto, agilidade: a geração de código reduz significativamente o tempo de desenvolvimento, permitindo reduzir prazos, ou até mesmo ganhar mais tempo para outras etapas do projeto.

> Em engenharia civil, a técnica de cimbramento (scaffolding em inglês) representa uma estrutura de suporte provisório durante o período de construções, normalmente composta por andaimes e escoras. Em programação, a técnica de scaffolding gera todas as estruturas fundamentais ao funcionamento de uma aplicação web. Ela recebeu esse nome pois faz alusão à técnica de cimbramento da engenharia civil: ambas representam estruturas temporárias que beneficiam os seus utilizadores. Em engenharia civil, a estrutura de andaimes é conhecida como scaffold. O análogo em programação é o sistema CRUD gerado (MAGNO, 2015, p. 22).

Diversos *frameworks* utilizam a técnica de *scaffolding*, a qual consiste em gerar o código funcional de um sistema partindo de templates a partir dos quais o código fonte é gerado. Estes templates que constituem um metamodelo essencialmente são arquivos de texto que contém a especificação do código fonte que se deseja gerar os quais possuem parâmetros que são substituídos pelo gerador no momento da geração do código fonte final (SILVA NETO; VILAR, 2018). Neste contexto, programas geradores de código que utilizam a técnica de *scaffolding* para a geração automática de código são chamados *scaffolders* (SHEARD, 2000, p.38).

Conforme Magno (2015) o processo de substituição e sintaxe dos parâmetros no template podem variar de gerador para gerador. No exemplo da Figura 1, o parâmetro `{{classe}}` (linhas 2, 3 e 4 da Figura 1) será substituído pelo valor do atributo classe no metamodelo, sendo realizada a mesma operação para todos os parâmetros restantes até um novo código fonte ser criado permitindo que este template seja reutilizado várias vezes, alterando-se somente o valor do parâmetro (FRANKY; PAVLICH-MARISCAL, 2012).

Figura 1 - Tradução de template em código fonte



Fonte: Magno (2015).

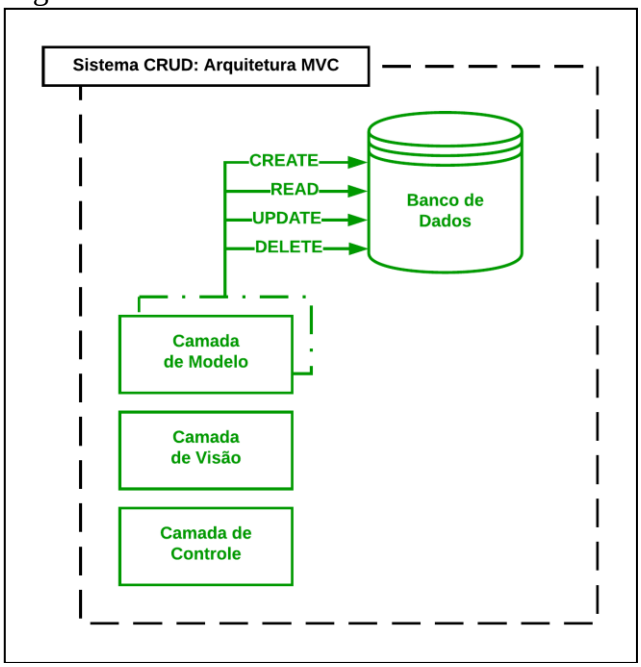
Uma aplicação prática da técnica de *scaffolding* ocorre na camada de *backend* de sistemas web em que as várias entidades que compõem o sistema precisam disponibilizar alguma forma de acesso aos dados. Conforme Cohen-Zardi (2013), “Todo o sistema CRUD possui várias entidades que modelam a aplicação. Para cada entidade existem métodos de acesso ao banco de dados, interfaces com o usuário, lógica da aplicação, etc. Codificar tudo isso manualmente consome tempo e pode introduzir erros”. Magno (2015, p.48) complementa que “personalizações no scaffolder devem ser feitas até certo ponto: o desenvolvedor não deve gastar mais tempo nisso do que resolvendo tarefas da lógica da aplicação” e sugere que o

modelo ideal envolve uma equipe responsável pela manutenção do scaffolder e outras equipes dedicadas à lógica da aplicação.

2.1.1 CRUD

Na medida em que aplicações utilizam sistemas gerenciadores de banco de dados para manipulação e armazenamento persistente dos mesmos, faz-se necessário o domínio de algumas operações básicas de acesso aos dados, quais sejam: criação de um registro, recuperação de registros, atualização e remoção de registros. Estas quatro operações são popularmente conhecidas pelo acrônimo CRUD, originado a partir das palavras: *Create*, *Read*, *Update* e *Delete*. A Figura 2 apresenta um modelo genérico de sistema baseado nas primitivas CRUD no padrão Model-View-Controller (MVC).

Figura 2 - Estrutura básica de um sistema CRUD



Fonte: Magno (2015, p.46).

A camada de acesso aos dados faz uso de uma linguagem de manipulação de dados (do inglês, Data Manipulation Language ou DML) a qual possui estruturas que implementam as quatro funcionalidades do padrão CRUD sendo SQL a DML mais conhecida. Magno (2015, p. 46) destaca que, “apesar de não ser especificamente uma DML, o protocolo HTTP também possui verbos para as operações CRUD”. O Quadro 1 apresenta uma associação entre as ações CRUD e seus correspondentes em SQL e HTTP.

Quadro 1 - Operações CRUD em SQL e HTTP

CRUD	SQL	HTTP
Create	insert	put/post
Read (Retrieve)	select	get
Update	update	put/patch
Delete (Destroy)	delete	delete

Fonte: baseado em Magno (2015, p.46).

2.1.2 Linguagem de Domínio Específico (DSL)

Uma DSL é uma linguagem de programação de computadores limitada, focada em um domínio (problema) específico. Enquanto outras linguagens de programação são de uso mais geral, uma DSL possui sintaxe e semântica delimitadas no mesmo nível de abstração que domínio oferece (GHOSH, 2011, p.10). Programas escritos usando uma DLS devem ter as mesmas qualidades esperadas em um programa escrito em qualquer outra linguagem de programação (GHOSH, 2011). Em outras palavras, uma DSL é puramente uma linguagem otimizada para um determinado tipo de problema, chamado domínio. Baseia-se em abstrações estreitamente alinhadas com o domínio (problema) para qual foi construída (VOELTER, 2013).

As DSLs podem ser classificadas da seguinte forma:

a) interna: uma DSL interna é aquela que usa a estrutura de uma linguagem de programação existente. Uma das DSLs internas mais famosas usada é Rails, que é implementado sobre a linguagem de programação Ruby;
b) externa: uma DSL externa precisa ser desenvolvida e ter uma infraestrutura separada para análise léxica, técnicas de análise, interpretação, compilação e geração de código.

Dentre os motivos para se usar uma DSL pode-se citar (MASCARENHAS, 2017):

a) aprimorar a produtividade de desenvolvimento: quanto mais fácil de ler um trecho de código, mais fácil de entende-lo e consequentemente encontrar erros e modifica-lo;
b) comunicação com especialistas em domínio: mesmo que um especialista não consiga escrever, ele tem a capacidade de ler e entender o que está expresso na DSL;
c) mudança no contexto de execução: de tempo de compilação para tempo de execução, ou do programa principal para um sistema de banco de dados;
d) modelo computacional alternativo: modelos diferentes do tradicional podem dar soluções mais simples para os mais diversos problemas.

Mascarenhas (2017), ainda destaca alguns problemas que podem surgir com o uso de DSLs:

a) cacofonia de linguagens: muitas linguagens no mesmo projeto dificulta a introdução de novas pessoas ao projeto;
b) custo: uma DLS tem um custo para ser implementada e mantida;
c) feature creep: uma DSL que ganha muitos recursos pode acabar se tornando uma linguagem de propósito geral, usada para desenvolver sistemas completos;
d) abstração restrita: a menor expressividade das DSLs pode complicar bastante a solução de problemas que não se encaixam bem no propósito dela.

Fowler e Parsons (2011, p.22) afirmam que a popularidade das DSL decorre do fato de que elas contribuem para melhorar a produtividade dos desenvolvedores e melhorar a comunicação com especialistas em domínio.

> Uma DSL bem escolhida pode facilitar a compreensão de um bloco de código complicado, melhorando assim a produtividade daqueles que trabalham com ela. Também pode facilitar a comunicação com especialistas em domínio, fornecendo um texto comum que funciona como software executável e uma descrição que os especialistas em domínio podem ler para entender como suas idéias são representadas em um sistema (FOWLER; PARSONS, 2011, p. 22)

A tecnologia JHipster (JHIPSTER, 2019) utiliza como um dos seus componentes base, uma linguagem denominada JHipster Domain Language para descrever todas as entidades e seus relacionamentos em arquivos com extensão `.jh` a qual é apresentada na próxima seção.

2.2 JHIPSTER

O JHipster é uma plataforma de desenvolvimento para gerar, desenvolver e implantar aplicativos Web Spring Boot + Angular / React / Vue e microsserviços Spring (JHIPSTER, 2019). O JHipster é um projeto de código aberto, com licença do Apache 2.0 no GitHub, iniciado por Julien Dubois em outubro de 2013, com primeira versão pública lançada em 7 de dezembro de 2013 (RAIBLE, 2019, p. 7).

> Em essência, o JHipster é um gerador Yeoman. Yeoman é um gerador de código que você executa com um comando *yo* para gerar aplicativos completos ou partes úteis de um aplicativo. Os geradores da Yeoman promovem o que a equipe da Yeoman chama de "fluxo de trabalho da Yeoman". Essa é uma pilha de ferramentas do lado do cliente que pode ajudar os desenvolvedores a criar rapidamente aplicativos da Web. Ele se encarrega de fornecer todo o necessário para começar a trabalhar sem as normais dores associadas a uma configuração manual. (RAIBLE, 2019, p.7, tradução nossa).

Conforme JHipster (2019, tradução nossa), o objetivo do JHipster é gerar um aplicativo Web completo e moderno ou uma arquitetura de microsserviços, unificando:

a) uma *stack* Java robusta e de alto desempenho no lado do servidor com o Spring

Boot;

b) um *frontend* elegante, moderno e responsivo usando Angular, React e Bootstrap;

c) uma arquitetura robusta de microsserviços;

d) um poderoso fluxo de trabalho para construir aplicativos com Yeoman, Webpack e Maven/Gradle.

Segundo Lima (2016, tradução nossa), o JHipster constrói um esqueleto de aplicativo a partir da linha de comando (Figura 3), em que o desenvolvedor escolhe dentre as opções suportadas, quais componentes o JHipster deve gerar para o aplicativo. Neste ponto, o modelo de domínio ainda não foi expresso nem criado pela JHipster.

Figura 3 - Configuração de uma aplicação JHipster

Fonte: Raible (2019).

Depois de criar o esqueleto do aplicativo, o JHipster fica pronto para receber as instruções para o modelo de domínio pretendido para o aplicativo. Para cada entidade criada, o JHipster fica responsável por gerar os seguintes componentes:

a) tabela de Banco de Dados com o respectivo registro de mudanças;

b) classe de entidade JPA;

c) interface Spring JPA Repository;

d) classe Spring MVC Rest Controller;

e) view: Angular, rotas, controlador, serviço e a página HTML relacionada.

Conforme descrito na seção 2.1, um gerador de código gera artefatos a partir de uma entrada predefinida, com o JHipster isso não é diferente. O JHipster suporta três métodos diferentes de entrada para a descrição do modelo de domínio pretendido:

a) `jhipster-entity`: é uma ferramenta de linha de comando que solicita aos desenvolvedores um conjunto de perguntas sobre a Entidade a ser criada. Esse processo requer a descrição de todos os campos, bem como os possíveis relacionamentos com outras entidades existentes;
b) é outra alternativa para usar ferramentas visuais. Os editores UML suportados pelo JHipster incluem Modelio, UMLDesigner, GenMyModel e Visual Paradigm;
c) a JHipster Domain Language (JDL) é uma DSL para descrever todas as entidades pretendidas e seus relacionamentos em um único arquivo. jh (ou mais de um) com uma sintaxe direta e fácil de usar (JHIPSTER, 2019).

Quadro 2 - Arquivo de configuração gerado

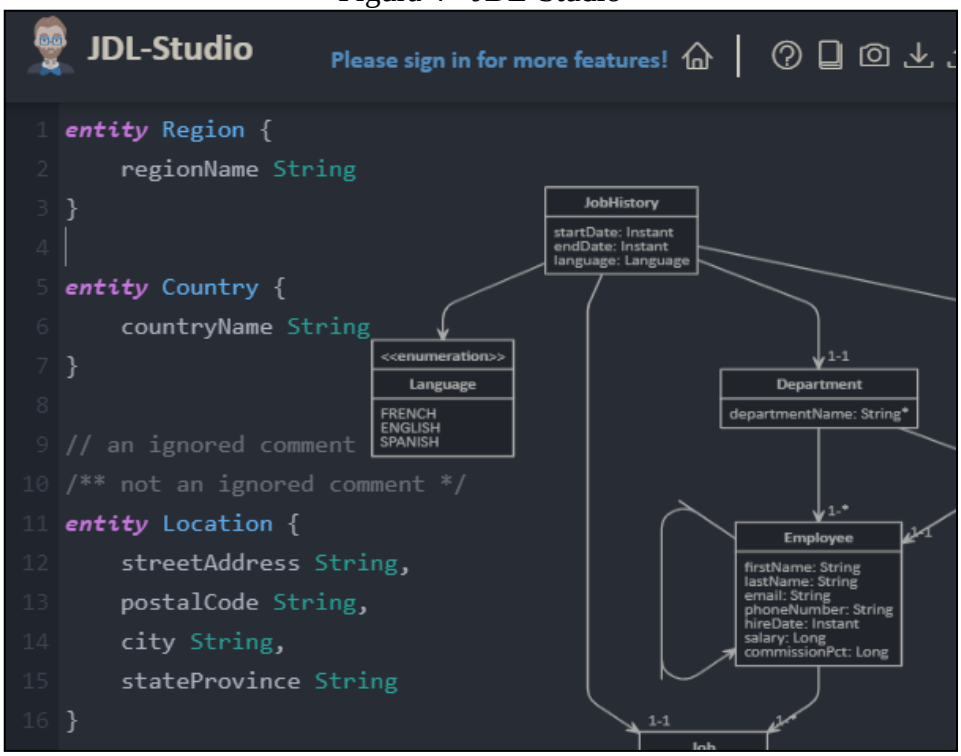
```
{
  "generator-jhipster": {
    "promptValues": {
      "packageName": "org.jhipster.health",
      "nativeLanguage": "en"
    },
    "jhipsterVersion": "5.2.1",
    "applicationType": "monolith",
    "baseName": "TwentyOnePoints",
    "packageName": "org.jhipster.health",
    "packageFolder": "org/jhipster/health",
    "serverPort": "8080",
    "authenticationType": "jwt",
    "clientPackageManager": "yarn",
    "testFrameworks": [
      "gatling",
      "protractor"
    ],
    "jhiPrefix": "jhi",
    "enableTranslation": true,
    "nativeLanguage": "en",
    "languages": [
      "en",
      "fr"
    ]
  }
}
```

Fonte: Raible (2019, p.22).

A partir destas configurações é gerado um arquivo `.yo-rc.json` (Quadro 2), que captura todas as opções que foram selecionadas. Este arquivo pode ser utilizado em uma pasta vazia para criar um projeto com as mesmas configurações (RAIBLE, 2019, p.12).

O JHipster fornece o JDL-Studio[1], uma ferramenta on-line com o intuito de ajudar os desenvolvedores a descrever o modelo de domínio para seus aplicativos, e durante a descrição o JDL-Studio oferece uma visualização em tempo real de um diagrama representando o modelo descrito. Conforme a Figura 4, percebe-se a linguagem descrita no lado esquerdo do JDL-Studio e sua representação em forma de diagrama no lado direito.

Figura 4 - JDL-Studio



Fonte: JHipster (2019).

Segundo Lima (2016, p. 48, tradução nossa) qualquer aplicativo criado pelo JHipster está pronto para produção, pois fornece métricas de monitoramento, armazenamento em cache mecanismos como Ehcache (para cache local) e Hazelcast (para sistema de cache distributivo); recursos estáticos otimizados usando cabeçalhos de cache GZip e HTTP; gerenciamento de log de tempo de execução usando Logback; e usa pool de conexão HikariCP.

A linguagem JDL possui elementos para a especificação de entidades e relacionamentos. Segundo JHipster (2019) a sintaxe de declaração de uma entidade se dá conforme apresentado no Quadro 3. Um exemplo de declaração de entidades pode ser visto na Figura 4.

[1] https://start.jhipster.tech/jdl-studio/

Quadro 3 - Sintaxe de declaração de uma entidade

```
[<entity javadoc>]
entity <entity name> [(<table name>)] {
  [<field javadoc>]
  <field name> <field type> [<validation>*]
}
```

Fonte: JHipster (2019).

Em que:

a) `<entity name>`: é o nome da entidade;

b) `<field name>`: é o nome de um campo da entidade;

c) `<field type>`: o tipo Jhipster suportado do campo da entidade;

d) `<validation>`: validações para o campo (não será abordado nesse trabalho)

e) `<entity javadoc>` e `<field javadoc>`: são opções de documentação para a entidade e os campos.

Conforme LIMA (2016, p.71), quando o JHipster está criando uma entidade, ele gera um arquivo em formato `JSON` e outro em formato `YALM` que descrevem esta entidade. Isto permite que os desenvolvedores alterem estes arquivos e gerem novamente a entidade utilizando recursos de linha de comando.

A sintaxe de declaração de um relacionamento se dá conforme apresentado no Quadro 4 (JHIPSTER, 2019). Um exemplo de declaração de entidades pode ser visto na Figura 5.

Quadro 4 - Sintaxe de declaração de um relacionamento

```
relationship (OneToMany | ManyToOne | OneToOne | ManyToMany) {
  <from entity>[{<relationship name>[(<display field>)]}] to <to
entity>[{<relationship name>[(<display field>)]}]+
}
```

Fonte: JHipster (2019).

Em que:

a) `OneToMany`, `ManyToOne`, `OneToOne` ou `ManyToMany`: tipo do relacionamento;

b) `<from entity>`: é o nome da entidade dona do relacionamento, a origem;

c) `<to entity>`: é o nme da entidade para onde o relacionamento se direciona, o destino;

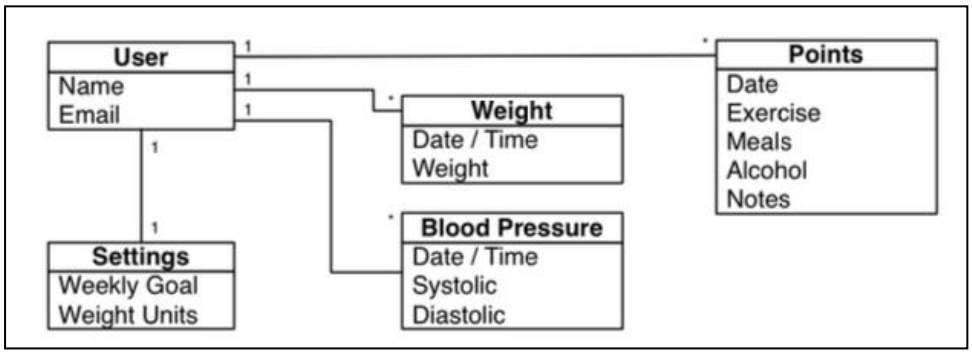
d) `<relationship name>`: é o nome do campo que será referenciado pelo relacionamento;

e) `<display field>`: o nome do campo que será exibido pelo *frontend* da aplicação gerada, o padrão é *id*.

Figura 5 - Exemplo de relacionamentos

Fonte: JHipster (2019).

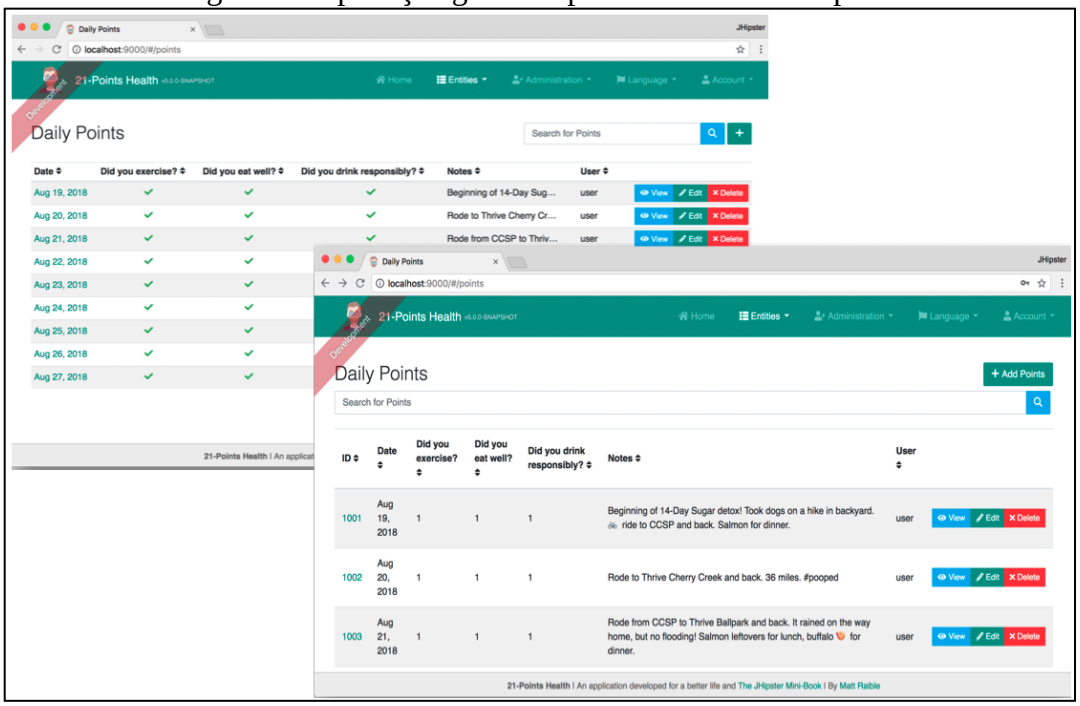
Figura 6 - Exemplo de modelo de dados



Fonte: Raible (2019, p.23).

Raible (2019, p.23) apresenta um modelo de dados de uma aplicação exemplo (Figura 6) e a correspondente aplicação produzida pelo JHipster (Figura 7).

Figura 7 - Aplicação gerada a partir do modelo JHipster



Fonte: Raible (2019, p.27).

2.3 TRABALHOS CORRELATOS

Nesta seção são apresentados três trabalhos correlatos com objetivos e/ou características similares ao proposto. Na seção 2.3.1 é detalhado o trabalho de Klug (2007) que consiste em uma ferramenta para geração automática de páginas Java Server Pages (JSP) a partir de uma base de dados pré-existente. Na seção 2.3.2 será apresentado o OpenXava, desenvolvido pela OpenXava (2019), que é um *framework* Java que gera aplicações Web a partir de uma simples classe Java. Por fim, na seção 2.3.3, é apresentado o AutoCRUD (RODRIGUEZ-ECHEVERRIA et al., 2016).

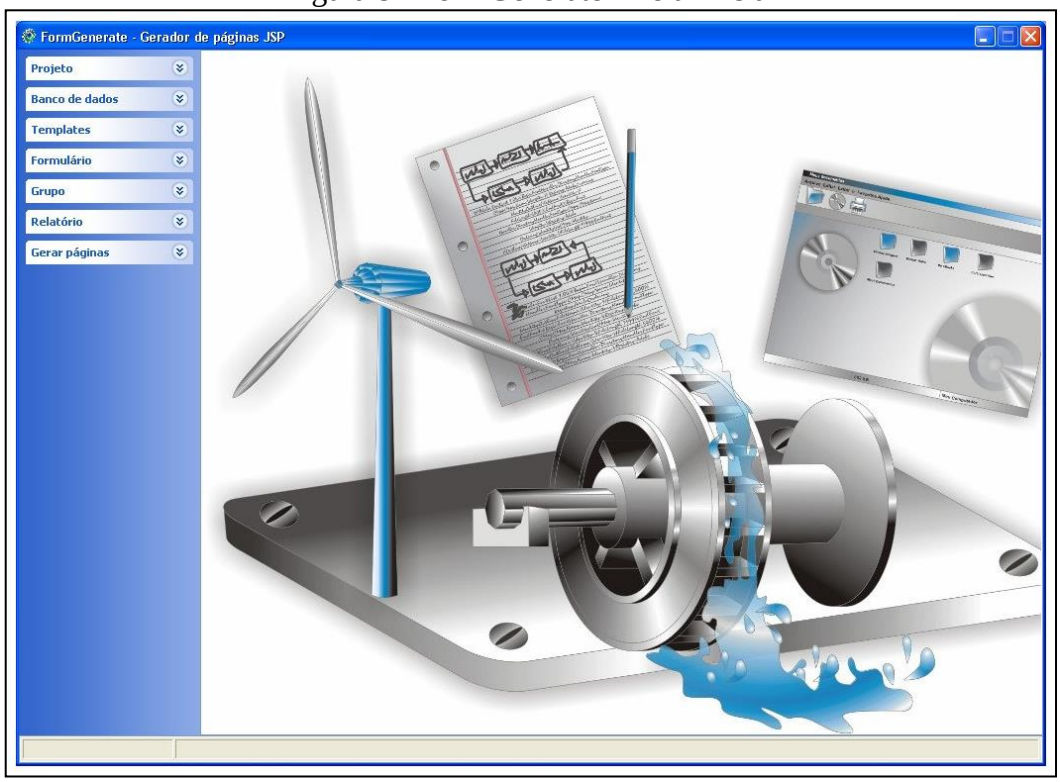
2.3.1 FormGenerate

Segundo Klug (2007, p. 92), “o FormGenerate foi desenvolvido com o objetivo de auxiliar o desenvolvedor na criação de rotinas de acesso ao banco de dados e de telas com funcionalidades de inclusão, alteração, exclusão e consulta de informações”. Com ele, o desenvolvedor pode configurar telas, grupos de telas e relatórios a partir da leitura dos metadados de um SGBD relacional.

A Figura 8 apresenta a tela inicial do FormGenerate que é o ponto de partida para a geração dos artefatos necessários para atender o objetivo da ferramenta. A ferramenta não gera uma aplicação completa, mas apenas classes Java para as camadas de modelo e controle (padrão MVC) e páginas JSP para a camada de visão, sendo necessário criar todo o projeto da aplicação

externamente, com o auxílio de ferramentas como o Eclipse ou o Netbeans. O FormGenerate não inclui nenhum suporte a controle de segurança (autenticação e autorização), sendo que essa funcionalidade já deve estar previamente construída no projeto externo.

Figura 8 - FormGenerate - Tela Inicial



Fonte: Klug (2007).

Para o desenvolvimento do FormGenerate, Klug (2007) utilizou os seguintes recursos:

a) linguagem Java;
b) banco de dados: Oracle, Microsoft Sql Server e Mysql;
c) API Java Database Connectivity (JDBC), responsável pela interação com o banco de dados;
d) biblioteca Standard Widget Toolkit (SWT), responsável pela exibição dos componentes gráficos;
e) padrão de arquitetura MVC;
f) Apache Velocity Project: motor de templates baseado em Java que auxilia na geração dos artefatos esperados.

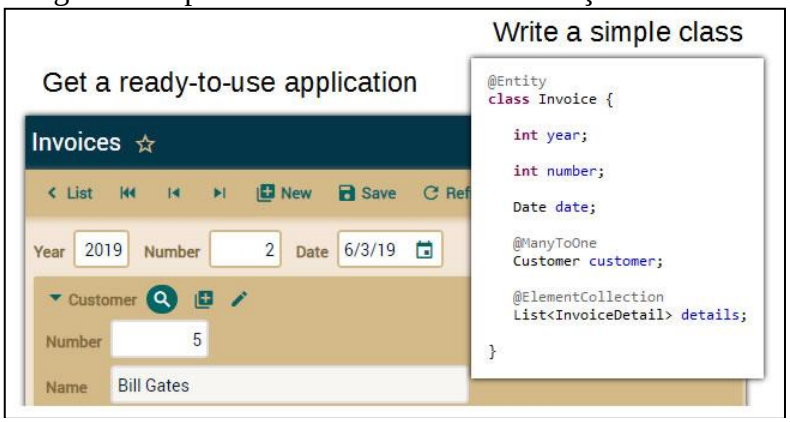
2.3.2 OpenXava

O OpenXava é um *framework* para o desenvolvimento de aplicativos a partir de especificações em Java. Conforme OpenXava (2019) o desenvolvimento usa uma abordagem

orientada por modelo, onde o núcleo do sistema são classes Java que modelam o problema, garantindo produtividade enquanto mantém um alto nível de encapsulamento.

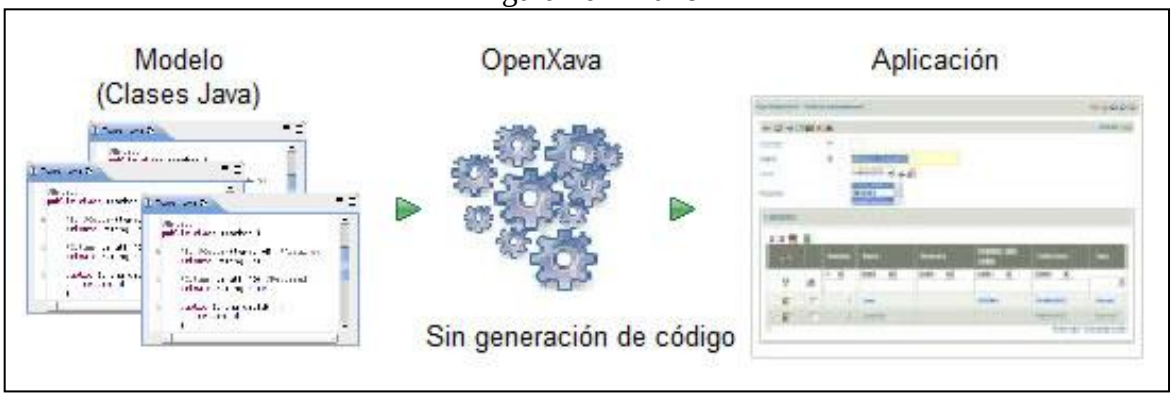
O OpenXava usa o conceito de Desenvolvimento Dirigido por Modelo (Model-Driven Development, MDD) e também o conceito de Componente de Negócio. Ambos os conceitos são aplicados de maneira simples, através de uma classe Java com anotações. A Figura 9 mostra um exemplo em que as anotações são identificadas por palavras iniciadas pelo símbolo @. Segundo OpenXava (2019), ao contrário do modelo de arquitetura MVC, em que todos os artefatos para a interface de usuário, modelo e controlador estão no mesmo lugar, no OpenXava toda essa estrutura está encapsulada em componentes de negócio separados por assuntos (OPENXAVA, 2019).

Figura 9 - OpenXava classe modelo e exibição dos dados



Fonte: OpenXava (2019).

Figura 10 - Fluxo



Fonte: OpenXava (2019).

O OpenXava não gera nenhum tipo de código, pois ele espera que todas as definições do modelo e negócio estejam definidas na classe Java. O uso dessa classe se dá dinamicamente em tempo de execução. Na Figura 10 tem-se esse fluxo representado graficamente.

O OpenXava utiliza as seguintes tecnologias e metodologias:

a) linguagem Java;

b) IDE Eclipse;

c) biblioteca Apache Ant: responsável pela geração dos projetos;
d) JPA e Hibernate: para acesso ao banco de dados;
e) JSP para renderização e exibição do lado do cliente.

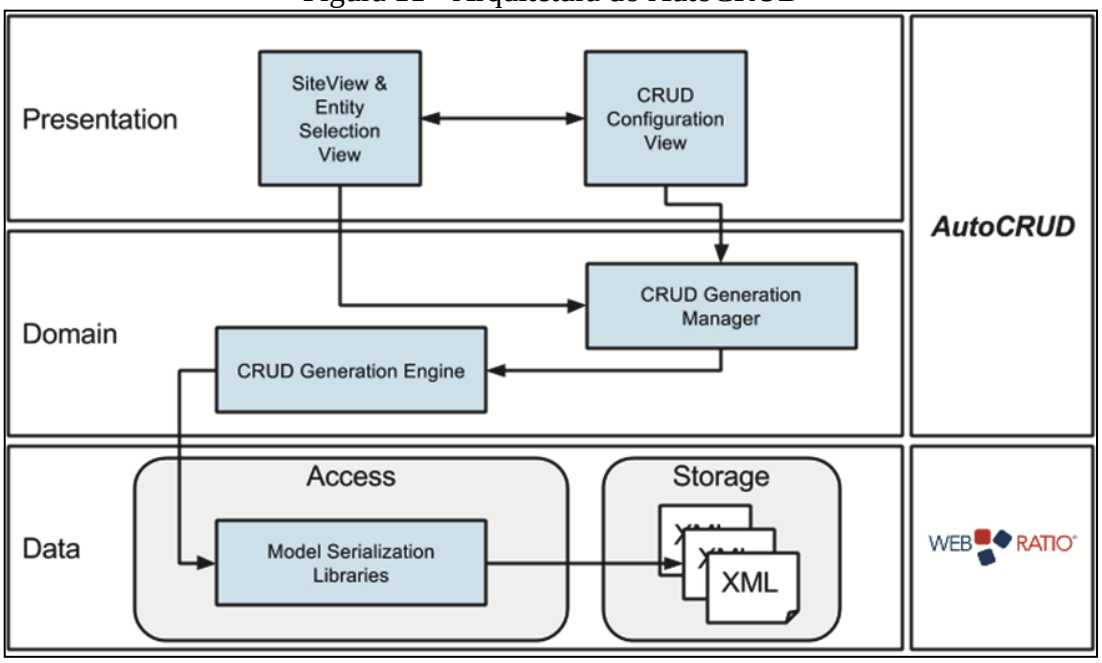
O OpenXava não possui nativamente uma forma de criar classes de modelo ou projeto a partir dos metadados de um SGBD relacional. Entretanto foi escolhido como trabalho correlato por ser uma ferramenta de geração rápida de aplicações Web. A versão gratuita do OpenXava não dá suporte a controle de segurança (autenticação e autorização) sendo que para usar essa funcionalidade é necessário adquirir uma licença.

2.3.3 AutoCRUD

Conforme Rodriguez-Echeverria et al. (2016, p. 309) a Interaction Flow Modelling Language (IFML) é "uma linguagem de modelagem padronizada pela Object Management Group (OMG) para representar um *frontend* de uma aplicação independentemente da tecnologia utilizada na implementação ou dos dispositivos alvo". Basicamente a IFML especifica um conjunto de elementos visuais que representam as interações que o usuário pode realizar e o comportamento do *frontend*. A linguagem IFML define os seguintes elementos: *View Container*, *View Component*, *Binding*, *Parameter*, *Event*, *Action*, *Navigation Flow*, e *Data Flow* (BRAMBILLA; FRATERNALI, 2015). O padrão é suportado por uma ferramenta denominada WebRatio a qual permite a edição e validação de modelos baseados em IFML mas também permite a geração do código final da aplicação para uma plataforma de *deployment* específica, reduzindo o tempo de disponibilização ao mercado e o esforço de desenvolvimento destas aplicações (RODRIGUEZ-ECHEVERRIA et al., 2016, p.307).

O projeto AutoCRUD foi desenvolvido a partir de ferramentas de *scaffolding* aplicadas no desenvolvimento de aplicativos da web na forma de um *plugin* para WebRatio de modo a reduzir o esforço de especificação de operações CRUD sobre as entidades expressas em IFML (RODRIGUEZ-ECHEVERRIA et al., 2016, p.309). O projeto é baseado na arquitetura tradicional de software de três camadas (Figura 11), estando o AutoCRUD posicionado dentro das camadas de domínio e apresentação. Em relação à apresentação, a interface do usuário é basicamente organizada em algumas caixas de diálogo, permitindo a seleção da entidade de dados e da visualização do site, por um lado, e o gerenciamento da operação CRUD, por outro.

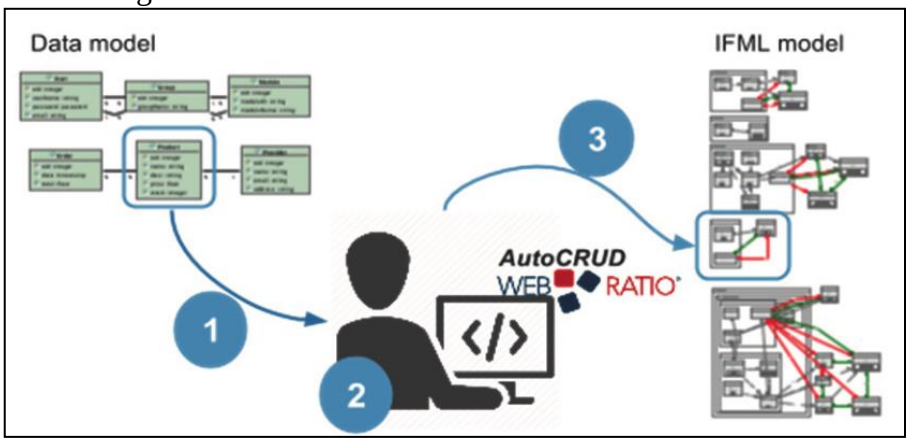
Figura 11 - Arquitetura do AutoCRUD



Fonte: Rodriguez-Echeverria et al. (2016, p. 309).

Em relação à camada de domínio, uma camada de fachada (*facade*) foi definida na API WebRatio para orquestrar sua funcionalidade, a fim de gerar a especificação IFML das operações CRUD. Esta fachada é formada por dois componentes principais: o gerente e o motor. O gerente é responsável por armazenar a configuração da operação CRUD a ser gerada. O mecanismo, chamado pelo gerente, é responsável por converter essa configuração em uma sequência concreta de chamadas de funcionalidade do WebRatio que eventualmente geram a especificação IFML.

Figura 12 - Modelo de funcionamento do AutoCRUD



Fonte: Rodriguez-Echeverria et al. (2016, p. 309).

Como mostrado na Figura 12, basicamente, o desenvolvedor deve realizar três ações:

a) selecionar uma entidade concreta no modelo de dados;
b) escolher as operações CRUD desejadas (criar, ler, atualizar, excluir ou multifuncional);

c) fornecer a ligação adequada para os parâmetros da operação CRUD (por exemplo, qual é o elemento a ser excluído).

Conforme os autores, este *plugin* fornece ao desenvolvedor a funcionalidade adequada para gerar automaticamente qualquer operação CRUD no IFML a partir de entidades de dados concretas.

> Na verdade, o AutoCRUD foi definido para se comportar como um orquestrador das diferentes funcionalidades do WebRatio, chamando os componentes certos no momento certo para essa geração. Os benefícios dessa abordagem são duplos: (1) permite uma integração mais durável com as funcionalidades WebRatio / IFML; e (2) o engenheiro pode usar o AutoCRUD em qualquer momento do ciclo de vida do desenvolvimento sem problemas. (RODRIGUEZ-ECHEVERRIA et al., 2016, p.309)

Os autores relatam que realizaram um estudo de caso tomando por base projetos reais desenvolvidos por uma empresa que desenvolveu mais de cem projetos utilizando WebRatio e IFML num período de nove anos. Os resultados observados apontaram evidências dos ganhos de otimização obtidos pelo uso do AutoCRUD e também a possibilidade de escalabilidade demonstrada em projetos de maior porte. Além disso, “o número de erros introduzidos na fase de especificação destas operações foi reduzido dramaticamente” (RODRIGUEZ-ECHEVERRIA et al., 2016, p. 313).

3 DESENVOLVIMENTO DO PROTÓTIPO

Este capítulo apresenta o desenvolvimento do protótipo de ferramenta chamado DB2JHipster, descrevendo as tecnologias e ferramentas utilizadas na sua implementação. Na elaboração e construção do DB2JHipster, foram seguidas as seguintes etapas:

a) especificação dos requisitos funcionais e não funcionais;

b) elaboração do diagrama de casos de uso;

c) elaboração do diagrama de pacotes;

d) elaboração do diagrama de dados;

e) elaboração da modelagem de dados;

f) descrição da implementação do protótipo;

O Quadro 5 apresenta os Requisitos Funcionais (RF) previstos para o protótipo e suas vinculações com os Casos de Usos associados.

Quadro 5 - Requisitos Funcionais

Requisitos Funcionais	**Caso de Uso**
RF01: O protótipo deverá permitir configurar o acesso a uma plataforma de banco de dados.	UC01
RF02: O protótipo deverá permitir a importação dos dados das tabelas e relacionamentos do banco de dados.	UC01
RF03: O protótipo deverá permitir o ajuste nos relacionamentos entre as tabelas (entidades).	UC01
RF04: O protótipo deverá permitir realizar a conversão das tabelas e relacionamentos importados para a linguagem JDL.	UC01
RF05: O protótipo deverá permitir a exportação e alteração da conversão gerada.	UC01

Fonte: elaborado pelo autor.

O Quadro 6 apresenta os Requisitos Não Funcionais (RNF) previstos para o protótipo.

Quadro 6 - Requisitos Não Funcionais

Requisitos Não Funcionais
RNF01: O protótipo deverá ser desenvolvido em Java.
RNF02: O protótipo deve suportar os bancos de dados: Postgresql, MySql, Sql Server e Oracle.
RNF03: O protótipo deve rodar em sistemas operacionais Windows e Linux e opcionalmente no Mac OS.
RNF04: Orientar o usuário no *deploy* do projeto e na utilização do modelo gerado.
RNF05: Chamar (opcional) o JHipster via interface ou linha de comando.
RNF06: Exibir o código gerado no JDL Studio via Selenium WebDriver.

Fonte: elaborado pelo autor.

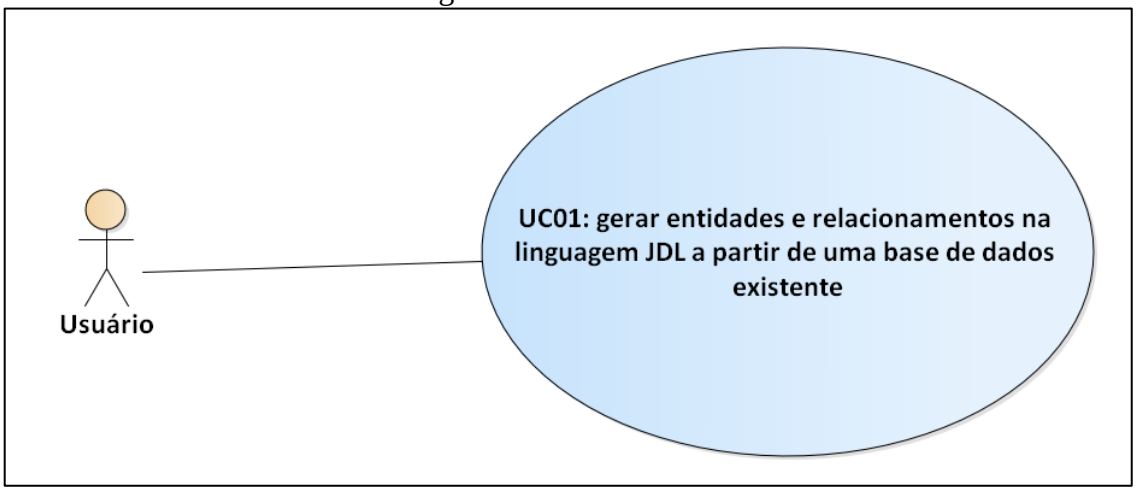
3.1 ESPECIFICAÇÃO

A especificação do sistema foi realizada a partir da ferramenta Enterprise Architect (EA), utilizando a linguagem de modelagem Unified Modeling Language (UML). Nesta seção são apresentados os diagramas de casos de uso, diagrama de classes, diagrama de sequência e a modelagem de dados.

3.1.1 Diagrama de Casos de Uso

Esta seção apresenta o diagrama de casos de uso do protótipo (Figura 13) contendo um caso de uso que contempla os requisitos funcionais definidos no Quadro 5. O detalhamento do caso de uso encontra-se no Apêndice A.

Figura 13 - Caso de Uso

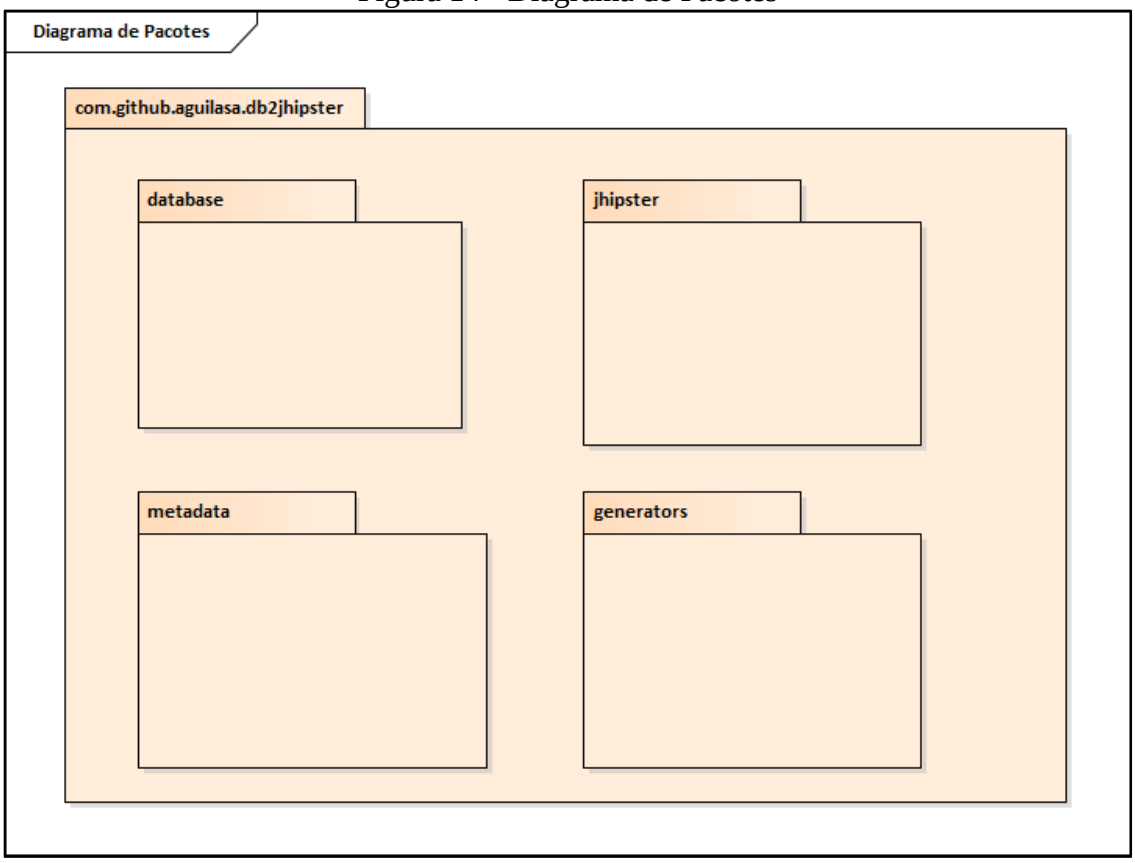


Fonte: elaborado pelo autor.

3.1.2 Diagrama de Pacotes

O diagrama de pacotes mostrado na Figura 14, exibe quatro principais pacotes: `database`, `metadata`, `jhipster` e `generators`. O pacote `database` contém as classes responsáveis por fazer a conexão com o banco de dados. O pacote `metadata` contempla as classes que representam os objetos do banco de dados, como tabela, campos e relacionamentos, e também é responsável for fazer a extração dos metadados da base. O pacote `jhispter` agrega as classes que identificam os tipos de objetos do JHipster e por fim o pacote `generators` contém as classes responsáveis de fazer a conversão dos objetos da base de dados no formato da linguagem JDL.

Figura 14 - Diagrama de Pacotes



Fonte: elaborado pelo autor.

3.1.3 Diagrama de Classes

Esta seção apresenta os diagramas de classes dos pacotes descritos na seção 3.1.2.

3.1.3.1 Diagrama de classes do pacote `database`

Conforme descrito na seção 3.1.2, o pacote `database` contém as classes responsáveis pela conexão com o banco de dados. A Figura 15 ilustra o diagrama de classes desse pacote. Segue o detalhamento das classes relacionadas, descrevendo o papel de cada uma delas na estrutura:

a) `DatabaseConfiguration`: classe responsável por representar as configurações da base do banco de dados, como nome da base, tipo do banco, servidor, porta, *schema*, usuário e senha;

b) `DatabaseType`: enumeração que representa o tipo de banco de dados, como Oracle, Sql Server, PostgreSQL ou MySql;

c) `BaseConnection`: classe abstrata que serve de base para realizar a conexão com os diferentes tipos bancos de dados;

d) `MySqlConnection`: implementação da classe que fará a conexão com o banco de dados MySql;

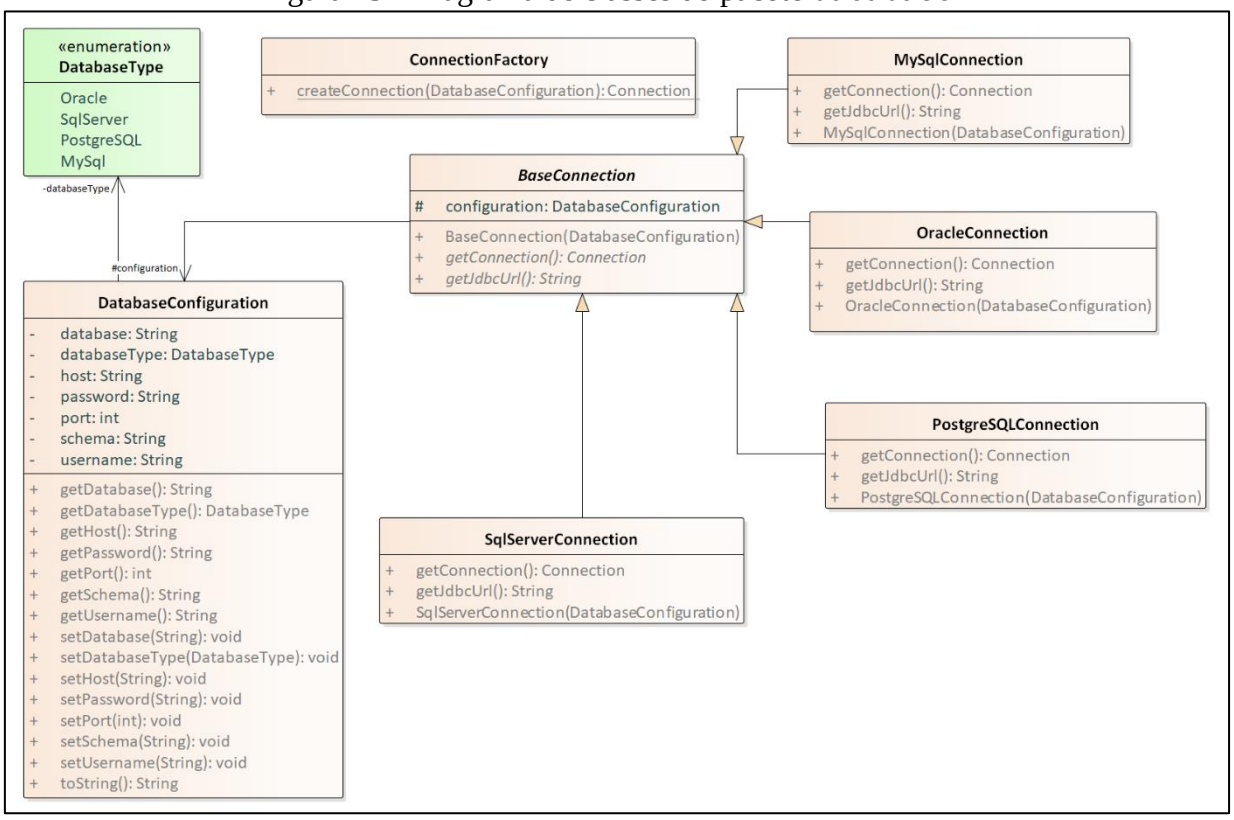
e) `OracleConnection`: implementação da classe que fará a conexão com o banco de dados Oracle;

f) `PostgreSQLConnection`: implementação da classe que fará a conexão com o banco de dados PostgreSQL;

g) `SqlServerConnection`: implementação da classe que fará a conexão com o banco de dados Sql Server;

h) `ConnectionFactory`: classe estática que a partir de uma instância da classe `DatabaseConfiguration` fará a conexão com o banco de dados de acordo com o valor do atributo `databaseType`.

Figura 15 - Diagrama de classes do pacote `database`



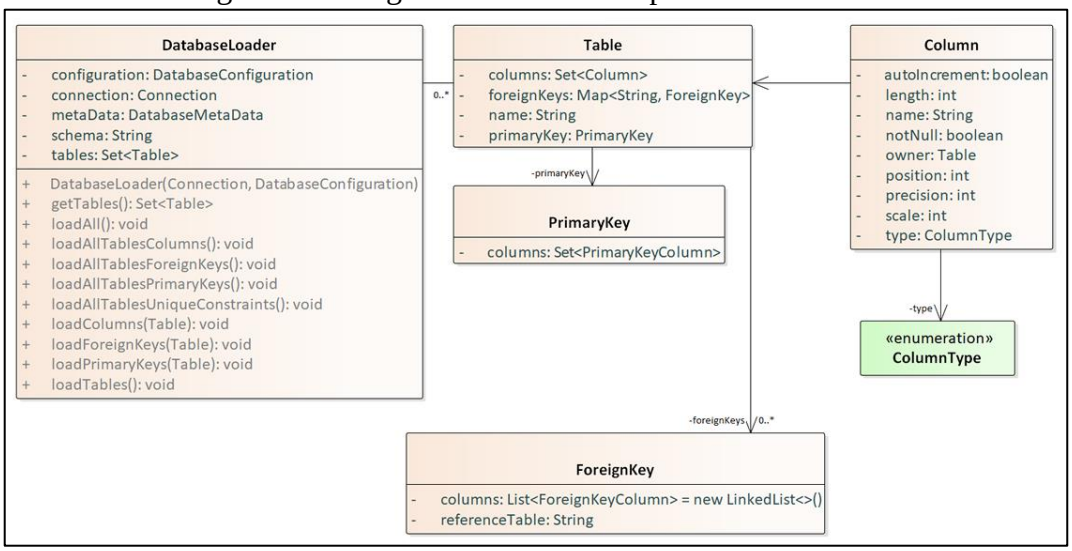
Fonte: elaborado pelo autor.

3.1.3.2 Diagrama de classes do pacote `metadata`

Conforme descrito na seção 3.1.2, o pacote `metadata` contempla as classes que representam os objetos do banco de dados, como tabela, campos e relacionamentos, a Figura 16 representa o diagrama de classes desse pacote. Segue o detalhamento das classes relacionadas, descrevendo o papel de cada uma delas na estrutura:

a) `DatabaseLoader`: classe responsável por carregar os metadados do banco de dados, como tabelas, campos, chaves primárias e estrangeiras e *constraints*;
b) `Table`: clase que representa uma tabela do banco de dados;
c) `Column`: classe que representa um campo ou coluna de uma tabela do banco de dados;
d) `ColumnType`: enumeração contendo os tipos dos campos;
e) `Constraint`: classe base para as chaves primárias e estrangeiras;
f) `ForeignKey`: classe que representa uma chave estrangeira de uma tabela;
g) `ForeignKeyColumn`: classe que representa um campo de uma chave estrangeira;
h) `PrimaryKey`: classe que representa a chave primária de uma tabela;
i) `PrimaryKeyColumn`: classe que representa um campo de uma chave primária;
j) `UniqueConstraint`: classe que representa uma *constraint* única de uma tabela.

Figura 16 - Diagrama de classes do pacote `metadata`



Fonte: elaborado pelo autor.

3.1.3.3 Diagrama de classes do pacote `jhipster`

Conforme descrito na seção 3.1.2, o pacote `jhipster` agrega as classes que identificam os tipos de objetos do JHipster, a Figura 17 representa o diagrama de classes desse pacote. Segue o detalhamento das classes relacionadas, descrevendo o papel de cada uma delas na estrutura:

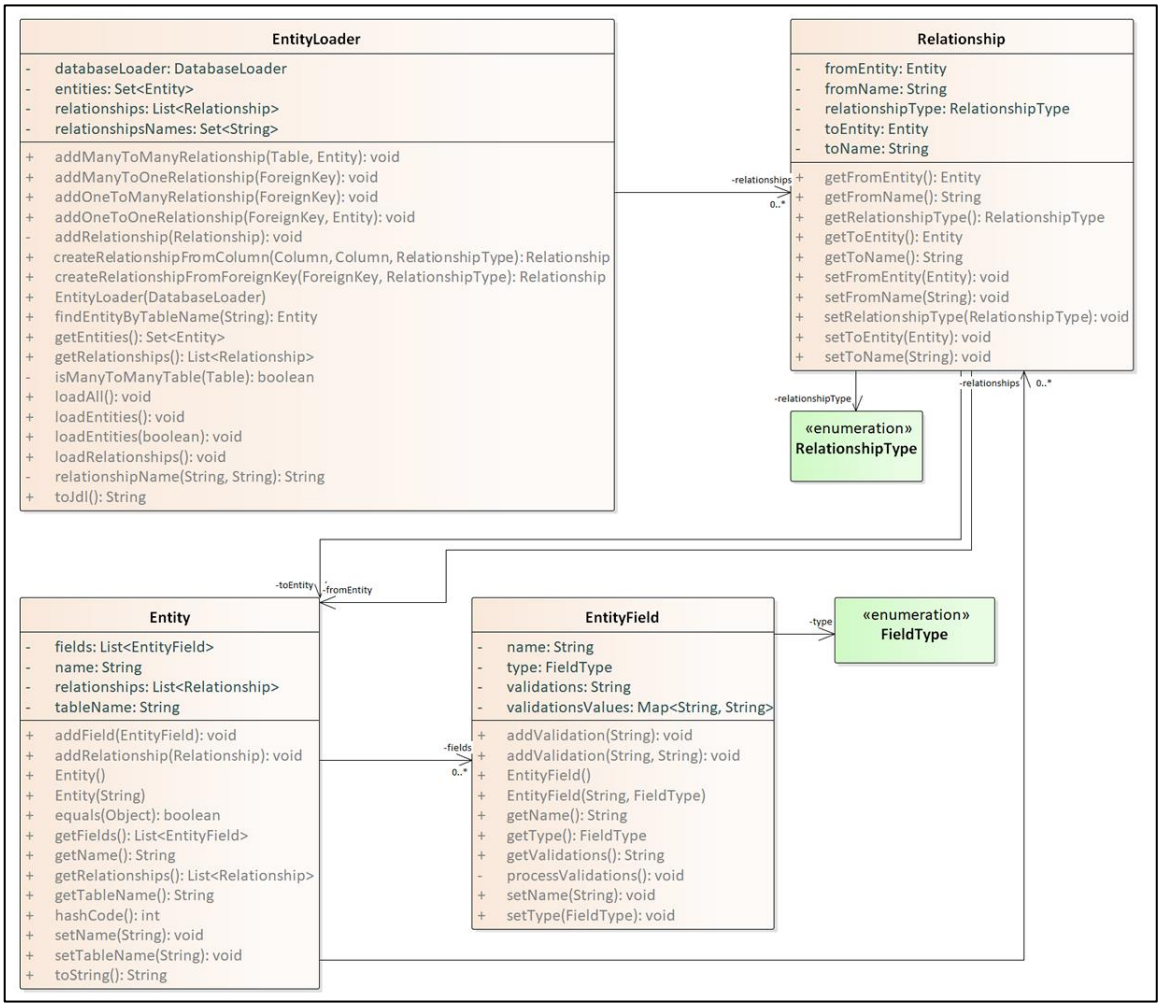
a) `EntityLoader`: classe responsável por gerar as entidades, campos e relacionamentos do JHipster, a partir das tabelas, campos e relacionamentos carregados pela classe `DatabaseLoader` do pacote `metadata`;
b) `Entity`: classe que representa uma entidade no formato do JHipster;
c) `EntityField`: classe que representa um campo de uma entidade;

d) `FieldType`: enumeração que representa o tipo do campo de uma entidade;

e) `Relationship`: classe que representa um relacionamento no formato do JHipster;

f) `RelationshipType`: enumeração que representa o tipo de relacionamento.

Figura 17 - Diagrama de classes do pacote `jhipster`



Fonte: elaborado pelo autor.

3.1.3.4 Diagrama de classes do pacote `generators`

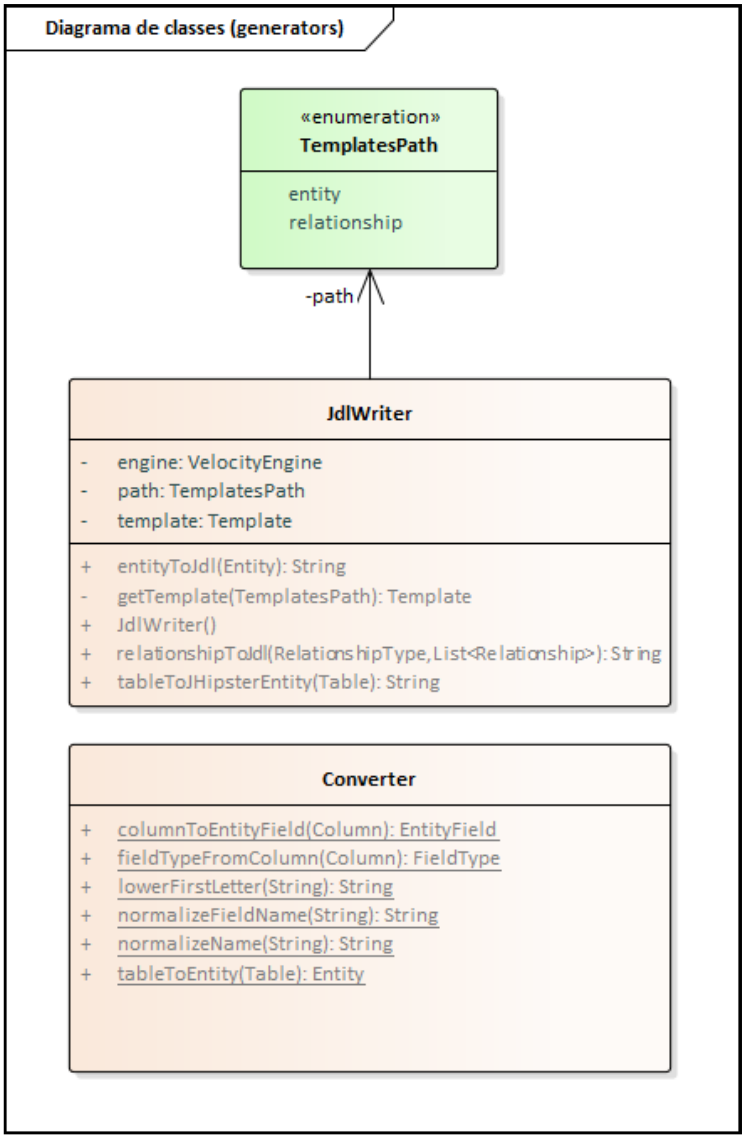
Conforme descrito na seção 3.1.2, o pacote `generators` contém as classes responsáveis de fazer a conversão dos objetos da base de dados no formato da linguagem JDL, a Figura 18 representa o diagrama de classes desse pacote. Segue o detalhamento das classes relacionadas, descrevendo o papel de cada uma delas na estrutura:

a) `JdlWriter`: classe responsável por gerar o código na linguagem JDL, a geração é feita utilizando o motor de templates Velocity;

b) `TemplatesPath`: enumeração para auxiliar o carregamento do template de acordo com o que será gerado, entidade ou relacionamento;

c) `Converter`: classe estática com métodos auxiliares para realizar a conversão para os tipos do JHipster.

Figura 18 - Diagrama de classes do pacote `generators`

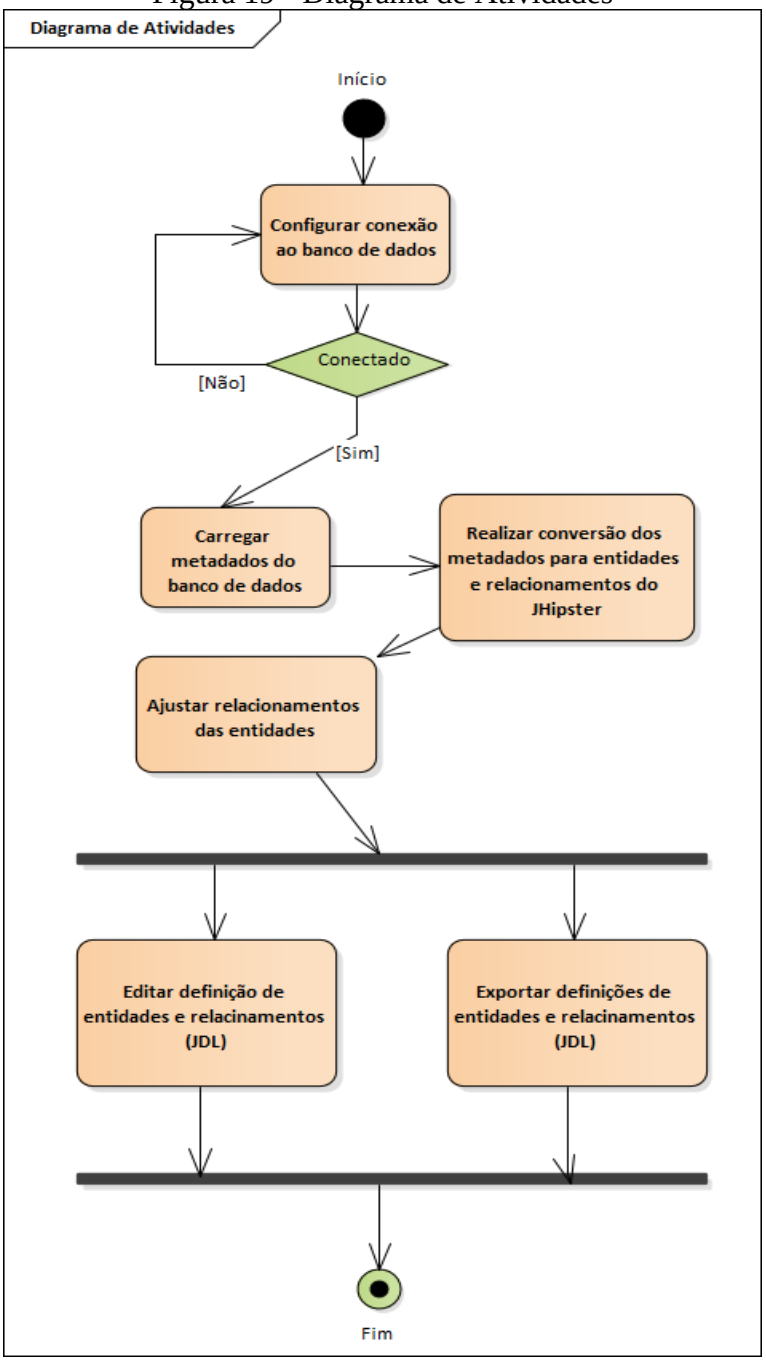


Fonte: elaborado pelo autor.

3.1.4 Diagrama de Atividades

A Figura 19 apresenta o diagrama de atividades do protótipo, através do diagrama de atividades, desde a conexão com o banco de dados à exportação da base no formato da linguagem JDL.

Figura 19 - Diagrama de Atividades



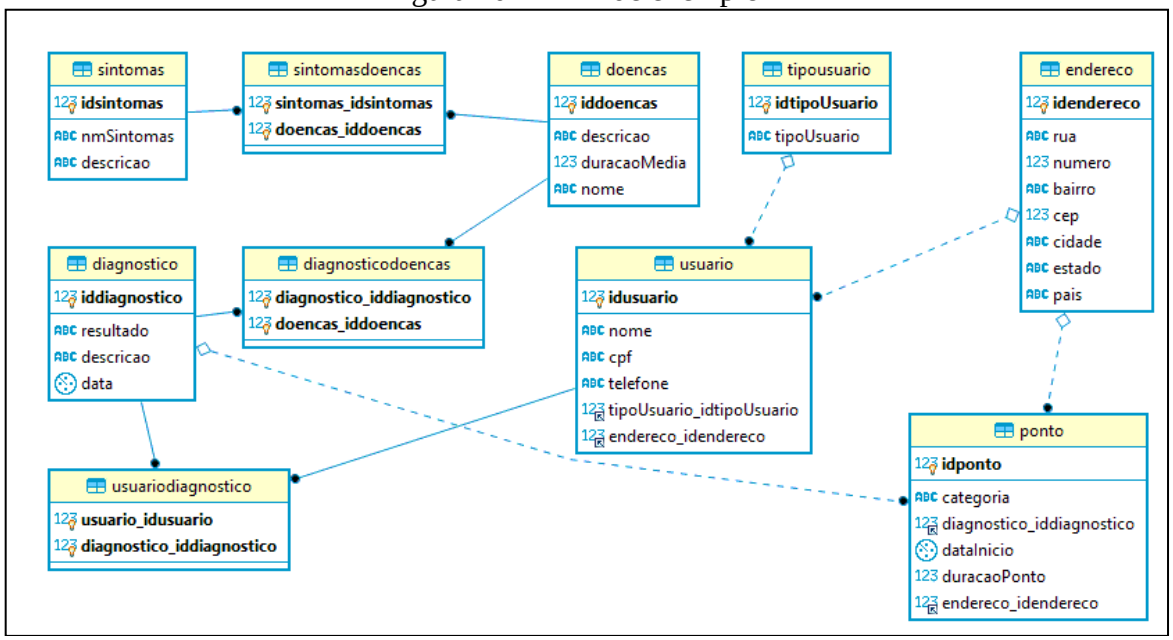
Fonte: elaborado pelo autor.

Para que o processo de importação dos metadados da base seja iniciado, é necessário que se faça a conexão com o banco de dados, assim que é realizada a conexão o protótipo passa a carregar e processar os metadados. Terminado o processo de extração, é feita a conversão dos metadados em entidades e relacionamentos de acordo com a especificação do JHipster, a próxima etapa do processo permite que sejam feitos ajustes nos relacionamentos, podendo ser alterado seu tipo. Finalmente existem duas etapas que permitem a edição e exportação das entidades e relacionamentos gerados.

3.1.5 Modelagem de Dados

Nesta seção são apresentados dois diagramas de modelo entidade relacionamento de exemplo, conforme Figura 20 e Figura 21. Esta modelagem foi feita apenas com o intuito de auxiliar o estudo de caso, sendo base para os testes do protótipo. A ferramenta usada para elaborar os diagramas foi o DBeaver[2].

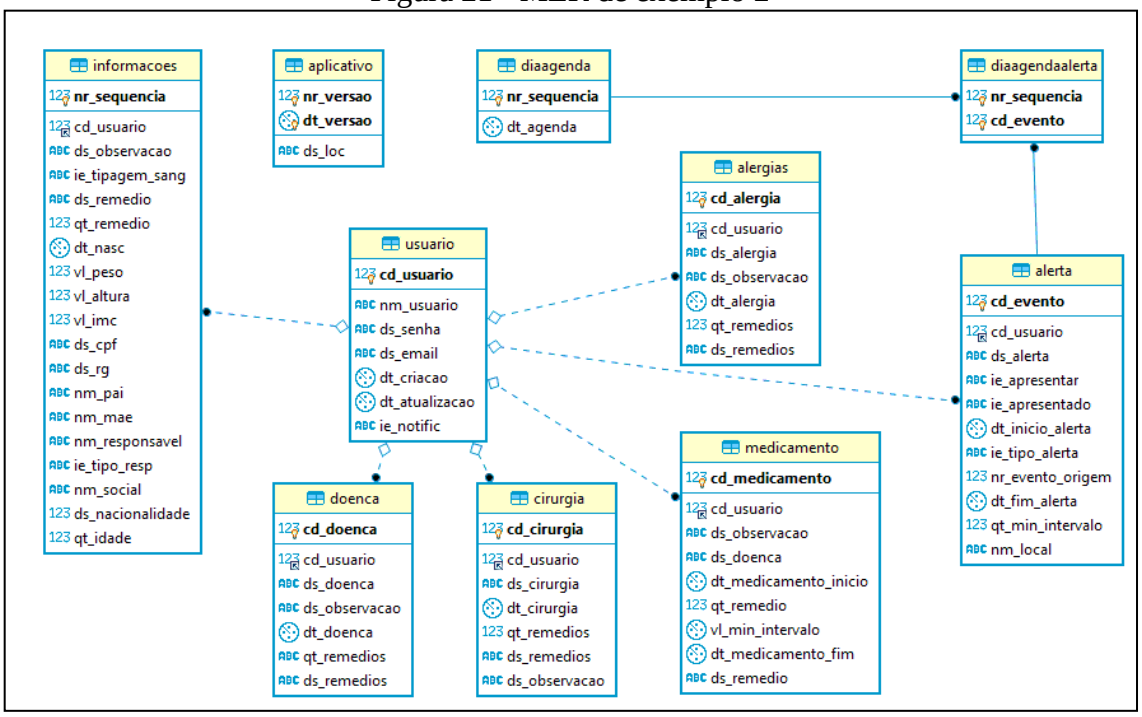
Figura 20 - MER de exemplo 1



Fonte: elaborado pelo autor.

[2] https://dbeaver.io/

Figura 21 - MER de exemplo 2



Fonte: Elaborado pelo autor.

3.2 IMPLEMENTAÇÃO

Nesta seção são mostradas as técnicas e ferramentas utilizadas para o desenvolvimento do protótipo, bem como a operacionalidade da implementação.

3.2.1 Técnicas e ferramentas utilizadas

O protótipo foi desenvolvido a linguagem Java, na versão JDK 8, como ambiente de desenvolvimento foi usado a IDE Eclipse; para auxiliar o gerenciamento de dependências (bibliotecas) e *deploy* do projeto, foi utilizado o Maven. Segue um detalhamento das principais bibliotecas, gerenciadas pelo Maven, utilizadas na implementação do protótipo:

a) `ojdbc8`: Oracle Database 12.2.0.1 JDBC Driver utilizado para conectar e gerenciar os metadados do banco de dados Oracle;

b) `sqljdbc42`: Microsoft JDBC Driver 4.2 utilizado para trabalhar com o banco Sql Server;

c) `postgresql`: PostgreSQL JDBC 4.2 Driver, verão 42.2.8, para gerenciar o banco de PostgreSQL;

d) `mysql-connector-java`: MySQL Connector/J, versão 5.1.46, driver JDBC do MySQL;

e) `velocity`: Velocity Template Engine, versão 1.7, biblioteca que auxilia a geração do código referente à linguagem JDL;

f) `selenium-java`: biblioteca usada para testes de aplicações web, onde é possível invocar um navegador e simular seu uso.

3.2.2 Implementação do protótipo

Nessa seção é apresentado o desenvolvimento do protótipo, exibindo e descrevendo trechos de código fonte. É detalhada a implementação dos pacotes definidos na seção 3.1.2 e introduz um novo pacote `view`, que contém as telas para a interação do usuário com as funcionalidades desenvolvidas.

3.2.2.1 Implementação do pacote `database`

As classes desse pacote já foram listadas e introduzidas na seção 3.1.3.1. A classe `DatabaseConfiguration` e a enumeração `DatabaseType` servem para representar as configurações da base de dados. As demais classes `BaseConnection`, `MySqlConnection`, `OracleConnection`, `PostgreSQLConnection`, `SqlServerConnection` e `ConnectionFactory` são responsáveis por realizar a conexão com o banco de dados configurado.

Para realizar a conexão foi criada a classe abstrata `BaseConnection`, conforme Quadro 7, onde em seu construtor é esperada uma instância da classe `DatabaseConfiguration` e são assinados dois métodos abstratos: `getConnection()` e `getJdbcUrl()`. Estes métodos são implementados pelas classes `MySqlConnection`, `OracleConnection`, `PostgreSQLConnection`, e `SqlServerConnection` que herdam de `BaseConnection`, a implementação desses métodos é apresentada no Quadro 8.

Quadro 7 - Classe `BaseConnection`

```
public abstract class BaseConnection {

	protected DatabaseConfiguration configuration;

	public BaseConnection(DatabaseConfiguration configuration) {
		this.configuration = configuration;
	}

	public abstract Connection getConnection() throws SQLException,
ClassNotFoundException;

	public abstract String getJdbcUrl();

}
```

Fonte: elaborado pelo autor.

Quadro 8 - Classe PostgreSQLConnection

```
public class PostgreSQLConnection extends BaseConnection {
	private static final String JDBC_URL_PG =
"jdbc:postgresql://%s:%d/%s";

	public PostgreSQLConnection(DatabaseConfiguration configuration) {
		super(configuration);
	}

	@Override
	public Connection getConnection() throws SQLException {
		Connection connection = DriverManager.getConnection(//
				getJdbcUrl(), //
				configuration.getUsername(), //
				configuration.getPassword()//
		);
		if (!StringUtils.isEmpty(configuration.getSchema())) {
			connection.setSchema(configuration.getSchema());
		}
		return connection;
	}

	@Override
	public String getJdbcUrl() {
		return String.format(JDBC_URL_PG, //
				configuration.getHost(), //
				configuration.getPort(), //
				configuration.getDatabase());
	}
}
```

Fonte: elaborado pelo autor.

Finalmente tem-se a classe estática `ConnectionFactory` que é responsável por criar uma instância da interface `java.sql.Connection` de acordo com o tipo de banco de dados configurado, o Quadro 9 mostra a implementação da classe ConnectionFactory.

Quadro 9 - Classe `ConnectionFactory`

```
public class ConnectionFactory {
	public static Connection createConnection(DatabaseConfiguration
configuration)
			throws SQLException, ClassNotFoundException {
		switch (configuration.getDatabaseType()) {
		case Oracle:
			return new OracleConnection(configuration).getConnection();
		case PostgreSQL:
			return new
PostgreSQLConnection(configuration).getConnection();
		case SqlServer:
			return new
SqlServerConnection(configuration).getConnection();
		case MySql:
			return new MySqlConnection(configuration).getConnection();
		default:
			break;
		}
		throw new RuntimeException("Tipo de banco de dados não suportado.");
	}
}
```

Fonte: elaborado pelo autor.

3.2.2.2 Implementação do pacote `metadata`

As classes desse pacote já foram enumeradas e detalhadas seção 3.1.3.2, a principal classe desse pacote é a classe `DatabaseLoader`, sendo que seus principais métodos são:

a) `getMetaData`: retorna uma instância da interface `java.sql.DatabaseMetaData`, responsável por buscar as informações do banco de dados;
b) `loadTables`: carrega as tabelas do banco de dados e cria instâncias da classe `Table`, armazenando-as em uma lista, sua implementação é exibida no Quadro 10;
c) `loadColumns`: recebe uma instância da classe `Table` como parâmetro e carrega os campos da tabela do banco de dados;
d) `loadPrimaryKeys`: recebe uma instância da classe `Table` como parâmetro e carrega as chaves primárias da tabela do banco de dados;
e) `loadForeignKeys`: recebe uma instância da classe `Table` como parâmetro e carrega as chaves estrangeiras do banco de dados;
f) `getTables`: retorna uma lista contendo todas as tabelas carregadas do banco de dados.

Quadro 10 - Método `loadTables` da classe `DatabaseLoader`

```
public void loadTables() throws SQLException {
      tables.clear();
      try (ResultSet rs = getMetaData().getTables(null, null, null,
TABLE_TYPE);) {
            while (rs.next()) {
                  String tableSchema = rs.getString("TABLE_SCHEM");
                  String tableType = rs.getString("TABLE_TYPE");
                  if (isTable(tableType) && validateSchema(tableSchema))
{
                        Table table = new
Table(rs.getString("TABLE_NAME"));
                        tables.add(table);
                  }
            }
      }
}
```

Fonte: elaborado pelo autor.

A interface `java.sql.DatabaseMetaData` não provê um método para fazer o carregamento total de uma tabela, cada tipo deve ser carregado individualmente. Para carregar as tabelas é disponibilizado o método `getTables`, para os campos o método `getColumns` e para as chaves primárias e estrangeiras os métodos `getPrimaryKeys` e `getImportedKeys`. Visando otimizar essas chamadas, foram criados os seguintes métodos, na classe `DatabaseLoader`:

a) `loadAllTablesColumns`: percorre a lista de tabelas carregadas pelo método `loadTables` e carrega os campos de cada uma delas;

b) `loadAllTablesPrimaryKeys`: percorre a lista de tabelas carregadas pelo método `loadTables` e carrega as chaves primárias de cada uma delas;

c) `loadAllTablesForeignKeys`: percorre a lista de tabelas carregadas pelo método `loadTables` e carrega as chaves estrangerias de cada uma delas;

d) `loadAll`: carrega tabelas, campos e chaves em um método único.

3.2.2.3 Implementação do pacote `jhipster`

Os principais tipos que a linguagem JDL apresenta são: entidade, campos e relacionamentos. Conforme as classes listadas na seção 3.1.3.3, estes tipos são representados pelas respectivas classes: `Entity`, `EntityField` e `Relationship`.

Os relacionamentos no JHipster seguem o padrão do JPA, são eles: `OneToOne`, `OneToMany`, `ManyToOne` e `ManyToMany`. Todos os relacionamentos são chaves estrangeiras na tabela do banco de dados e no carregamento e criação dos mesmos a partir do banco de dados. Estes são classificados de duas formas: `OneToOne` ou `ManyToMany`, para alteração do `OneToOne` para os outros tipos é disponibilizada uma tela para ajustes.

Os tipos dos campos da entidade seguem um padrão de tipos similar ao Java, eles são representados pela enumeração `FieldType`, sendo eles: `String`, `Integer`, `Long`, `Float`, `Double`, `BigDecimal`, `LocalDate`, `Instant`, `ZonedDateTime`, `Duration`, `UUID`, `Boolean`, `Enumeration`, `Blob`, `TextBlob`, `ImageBlob`.

Quadro 11 - Método `loadEntities` da classe `EntityLoader`

```
public void loadEntities(boolean loadMetadata) throws SQLException {
      try {
            entities.clear();
            if (loadMetadata) {
                  databaseLoader.loadAll();
            }
            Set<Table> tables = databaseLoader.getTables();
            for (Table table : tables) {
                  entities.add(Converter.tableToEntity(table));
            }
      } catch (Exception e) {
            e.printStackTrace();
      }
}
```

Fonte: elaborado pelo autor.

A principal classe desse pacote é a classe `EntityLoader`, que recebe uma instância da classe `DatabaseLoader`, e partir dela pode fazer criação das entidades no formato do JHipster. Os principais métodos da classe `EntityLoader` são:

a) `loadEntities`: percorre a lista de tabelas carregadas pela instância de `DatabaseLoader` e converte cada tabela e seus campos para um objeto da classe `Entity` e armazena numa lista de entidades, conforme Quadro 11;

b) `loadRelationships`: percorre a lista de tabelas carregadas pela instância de `DatabaseLoader` e converte cada chave estrangeira encontrada para um objeto da classe `Relationship` e armazena numa lista de relacionamentos, conforme Quadro 12;

c) `getEntities`: retorna a lista com todas as entidades criadas;

d) `getRelationships`: retorna a lista com todos os relacionamentos criados.

Quadro 12 - Método `loadRelationships` da classe `EntityLoader`

```
public void loadRelationships() {
      relationships.clear();
      Set<Entity> remove = new LinkedHashSet<>();
      for (Entity entity : entities) {
            Table table = databaseLoader.findTableByName(//
                        entity.getTableName() //
            );
            if (isManyToManyTable(table)) {
                  addManyToManyRelationship(table, entity);
                  remove.add(entity);
            } else {
                  table.getForeignKeys().entrySet().stream().map(//
                              Map.Entry::getValue).forEach(f -> {
                                    addOneToOneRelationship(f, entity);
                              });
            }
      }
      for (Entity entity : remove) {
            entities.remove(entity);
      }
}
```

Fonte: elaborado pelo autor.

3.2.2.4 Implementação do pacote `generators`

As classes do pacote `generators` funcionam como rotinas auxiliares às classes do pacote `jhipster`, e estas foram listadas e introduzidas na seção 3.1.3.4. A classe `Converter` funciona basicamente como um conversor de tabelas (classe `Table`) para entidades (classe `Entity`) e de campos de tabela (classe `Column`) para campos de entidade (classe `EntityField`), seus principais métodos são:

a) `tableToEntity`: recebe uma instância de `Table` e converte para `Entity`, conforme Quadro 13;

b) `columnToEntityField`: recebe uma instância de `Column` e converte para `EntityField`, conforme Quadro 14.

Quadro 13 - Método `tableToEntity` da classe `Converter`

```
public static Entity tableToEntity(Table table) {
	Entity entity = new Entity(normalizeName(table.getName()));
	entity.setTableName(table.getName());
	Set<Column> columns = table.getColumns();
	for (Column column : columns) {
		if (checkCreateField(table, column)) {
			EntityField entityField = columnToEntityField(column);
			entity.addField(entityField);
		}
	}
	return entity;
}
```

Fonte: elaborado pelo autor.

Quadro 14 - Método `columnToEntityField` da classe `Converter`

```
public static EntityField columnToEntityField(Column column) {
	EntityField field = new EntityField();
	field.setName(normalizeFieldName(column.getName()));
	field.setType(fieldTypeFromColumn(column));
	return field;
}
```

Fonte: elaborado pelo autor.

Quadro 15 - Método `fieldTypeFromColumn` da classe `Converter`

```
public static FieldType fieldTypeFromColumn(Column column) //
	throws RuntimeException {
	ColumnType columnType = column.getType();
	if (COLUMN_TYPE_FIELD_TYPES_MAP.containsKey(columnType)) {
		return COLUMN_TYPE_FIELD_TYPES_MAP.get(columnType);
	} else if (columnType.equals(ColumnType.DECIMAL) || //
			columnType.equals(ColumnType.NUMERIC)) {
		int columnSize = column.getLength();
		if (column.getScale() >= 0) {
			if (columnSize <= 19) {
				return FieldType.DOUBLE;
			}
		} else {
			if (columnSize <= 10) {
				return FieldType.INTEGER;
			} else if (columnSize <= 19) {
				return FieldType.LONG;
			}
		}
		return FieldType.BIGDECIMAL;
	}
	throw new RuntimeException(//
			String.format("Não foi possível converver o tipo: %s",
columnType)//
	);
}
```

Fonte: elaborado pelo autor.

Para realizar a conversão do tipo do campo da tabela, foi criado o método `fieldTypeFromColumn`, que verifica se o tipo existe num mapa de tipos, ou se é um tipo numérico ou decimal, e então retorna o tipo equivalente do JHipster. Sua implementação pode ser vista no Quadro 15.

Para geração do código da linguagem JDL foi usada a biblioteca Apache Velocity que é um motor de templates que roda em Java. Baseado nas sintaxes vistas na seção 2.2, foram criados dois templates, conforme Quadro 16 e Quadro 17.

Quadro 16 - Template para geração de uma entidade

```
entity ${name}Entity {
#foreach( $field in $fields )
      $field.name $field.type#if( $field.validations ) $field.validations
#end

#end
}
```

Fonte: elaborado pelo autor.

Quadro 17 - Template para geração de relacionamentos

```
relationship ${name} {
#foreach( $relationship in $relationships )
      ${relationship.fromEntity.name}Entity{$relationship.fromName} to
${relationship.toEntity.name}Entity{$relationship.toName}
#end
}
```

Fonte: elaborado pelo autor.

Quadro 18 - Trecho de código da classe JdlWriter

```
public class JdlWriter {
      ...código suprimido
      public String entityToJdl(Entity entity) {
            VelocityContext context = new VelocityContext();
            context.put("name", entity.getName());
            List<EntityField> fields = entity.
                        getFields().
                        stream().
                        collect(Collectors.toList());
            context.put("fields", fields);
            StringWriter writer = new StringWriter();
            getTemplate(TemplatesPath.entity).merge(context, writer);
            return writer.toString();
      }

      public String relationshipToJdl(RelationshipType relationshipType,
                  List<Relationship> relationships) {
            VelocityContext context = new VelocityContext();
            context.put("name", relationshipType.getName());
            context.put("relationships", relationships);
            StringWriter writer = new StringWriter();
            getTemplate(TemplatesPath.relationship).merge(context,
writer);
            return writer.toString();
      }
      ...código suprimido
}
```

Fonte: elaborado pelo autor.

A classe responsável por fazer a geração do código é classe `JdlWriter`, que a partir dos métodos `entityToJdl` e `relationshipToJdl`, carrega os templates em memória e gera a saída no formato suportado pela linguagem JDL, conforme mostra o trecho de código do Quadro 18.

3.2.2.5 Implementação do pacote `view`

O fluxo de uso de uso do protótipo foi criado para funcionar como um *wizard*, ou seja, por etapas, com botões de avanço e retorno e para contemplar esse funcionamento foi implementada uma tela principal e quatro painéis (visões) para cada etapa do passo a passo.

As principais classes do pacote view são:

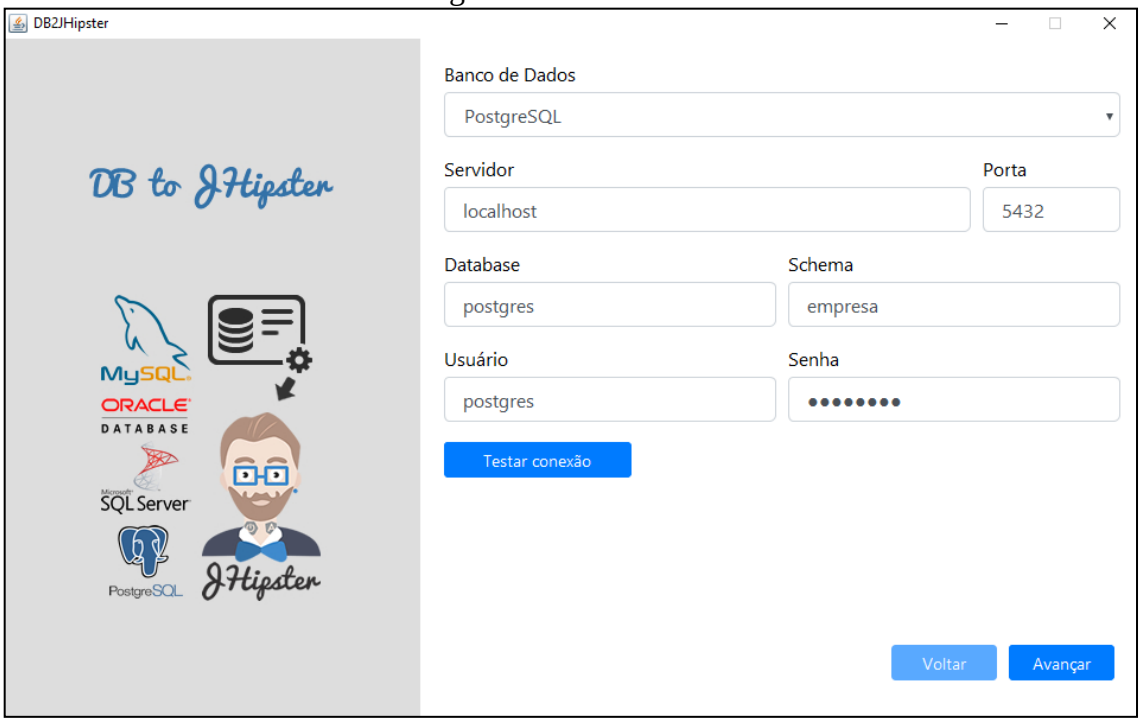
a) `MainView`: esta classe contempla a tela principal do protótipo, com uma área onde as demais visões serão exibidas, bem como os botões de avanço e retorno;
b) `DBConfigView`: esta classe implementa a visão da configuração das informações do banco de dados;
c) `LoadingView`: esta classe apenas exibe uma barra de progresso informando o usuário sobre o que está sendo carregado do banco de dados;
d) `RelationshipView`: esta classe implementa a visão para o usuário poder alterar o tipo dos relacionamentos após o carregamento;
e) `JdlView`: e finalmente, esta classe exibe o código JDL gerado, possibilitando o usuário fazer alterações de acordo com o que achar necessário.

3.2.3 Operacionalidade da implementação

Nesta seção são apresentadas as telas do protótipo com a descrição de suas funcionalidades.

Na tela inicial (Figura 22) o usuário deverá optar pelo banco de dados desejado e informar os dados de conexão suas configurações.

Figura 22 - Tela inicial

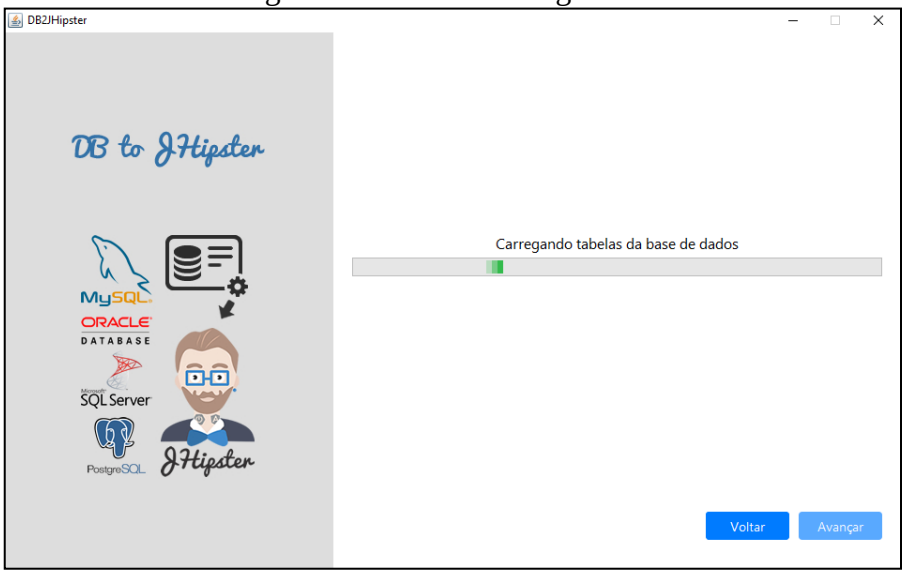


Fonte: elaborado pelo autor.

Conforme Figura 22, a primeira opção é a seleção do tipo de banco de dados: Postgresql, MySql, Sql Server ou Oracle. A segunda configuração é o servidor onde o banco está instalado, e em seguida sua porta. Devem ser informadas ainda o nome da base de dados, bem como o *schema* da base. E finalmente o usuário e senha para o acesso ao banco.

A segunda tela do protótipo exibe ao usuário uma barra de progresso, indicando o que está sendo carregado do banco de dados. Seu funcionamento é ilustrado na Figura 23.

Figura 23 - Tela de carregamento

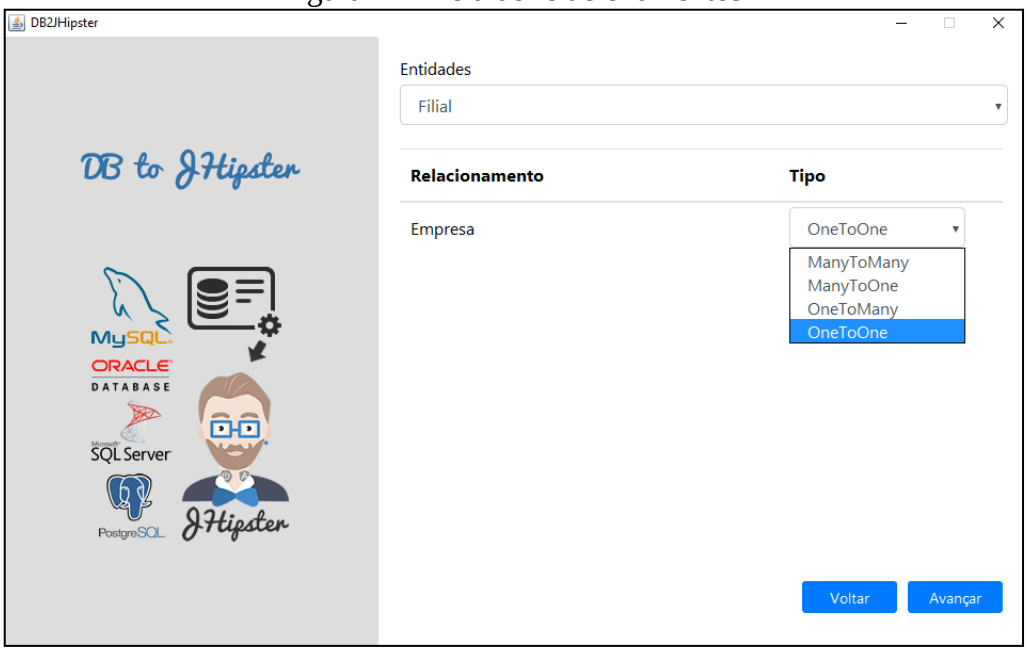


Fonte: elaborado pelo autor.

Após o carregamento de todos os objetos, o botão Avançar fica habilitado, possibilitando a continuação do processo.

Na próxima tela do protótipo o usuário poderá alterar o tipo de relacionamento, para isso basta selecionar uma entidade e na lista de relacionamentos escolher uma opção, conforme Figura 24.

Figura 24 - Tela de relacionamentos



Fonte: elaborado pelo autor.

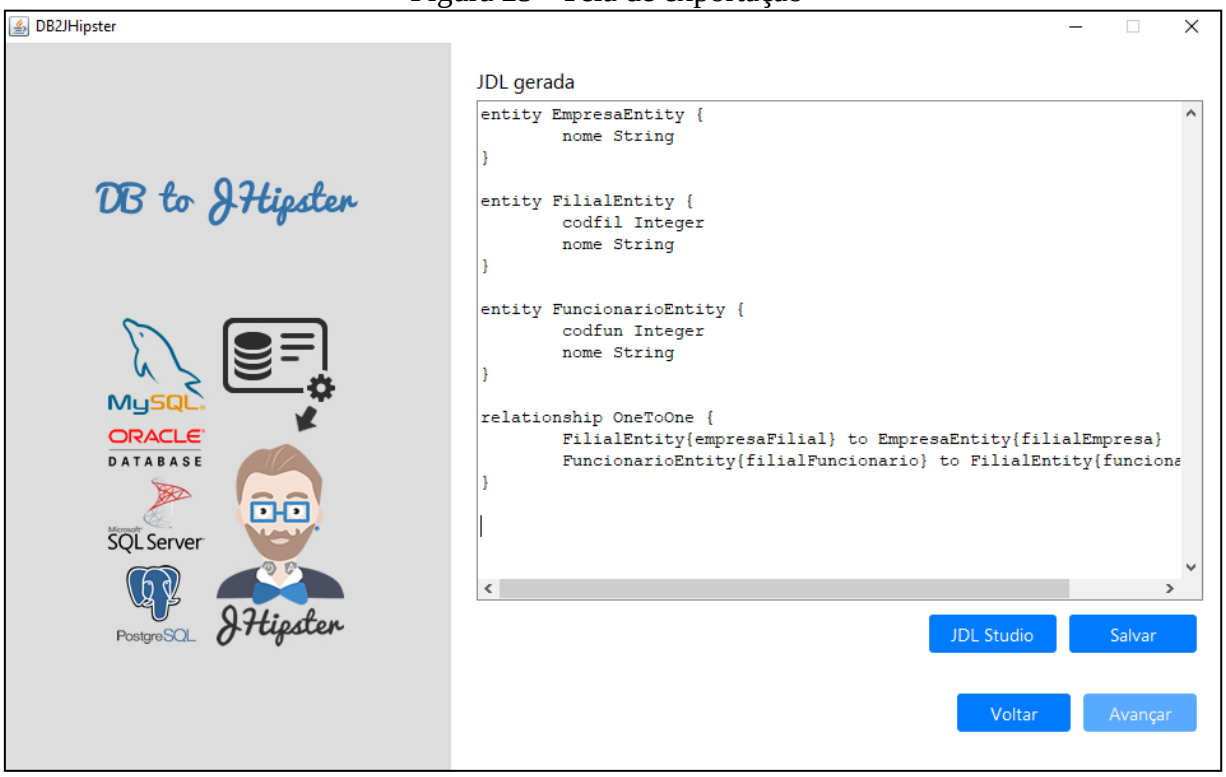
Por fim a tela que exibe o código JDL gerado a partir das tabelas, campos e relacionamentos encontrado na base de dados. Conforme observado na Figura 25, o código gerado é exibido em uma caixa de texto editável, onde o usuário pode fazer algum ajuste, porém não existe nenhum tipo de validação no código alterado.

Existem dois botões extras nessa tela, são eles:

a) *JDL Studio*: conforme visto na seção 2.2, o JHipster fornece uma ferramenta de edição on-line chamada JDL Studio, este botão executa uma instância do navegador Google Chrome diretamente no endereço do JDL Studio, passando o código gerado pelo protótipo;
b) Salvar: através deste botão o usuário pode salvar o código gerado diretamente em um arquivo de extensão `.jh`, suportado pelo JHipster.

Para que fosse possível realizar a integração com a ferramenta on-line JDL Studio foi utilizada a biblioteca Selenium WebDriver, que permite que seja possível realizar testes automatizados em *frontends*, simulando a interação do usuário com a página web.

Figura 25 - Tela de exportação



Fonte: elaborado pelo autor.

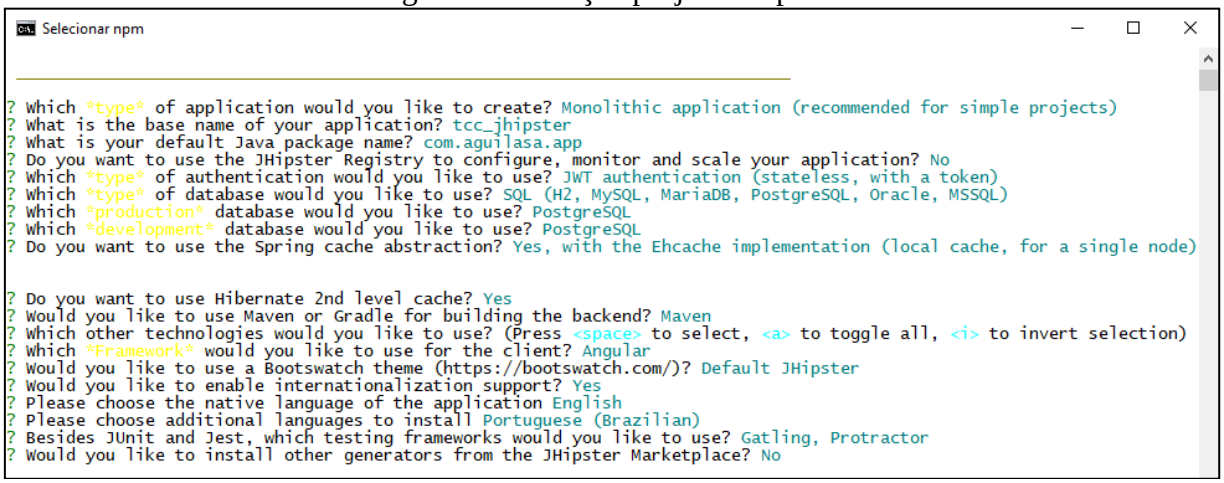
3.3 RESULTADOS E DISCUSSÕES

Esta seção apresenta os resultados e discussões deste trabalho. A seção 3.3.1 relata a etapa de validação do protótipo e a seção 3.3.2 descreve as discussões a respeito desse trabalho, bem como comparações com os trabalhos correlatos.

3.3.1 Validação do protótipo

Para validar o código JDL gerado pelo protótipo foi necessário criar um novo projeto JHipster, e foi escolhido como banco de dados de desenvolvimento o PostgreSQL, conforme opções exibidas na Figura 26. O detalhamentoi de como iniciar uma aplicação e usar o código gerado pelo protótipo encontra-se no Apêndice C.

Figura 26 - Criação projeto JHipster



Fonte: elaborado pelo autor.

Após a conclusão da geração do projeto, foi configurado o acesso ao banco de dados PostgreSQL citado. Para gerenciar os dados de acesso foi necessário alterar o arquivo *application-dev.yml* que se encontra dentro da pasta *src\main\resources\config*. Essa configuração é demonstrada no Quadro 19.

Quadro 19 - Configuração do banco de dados da aplicação

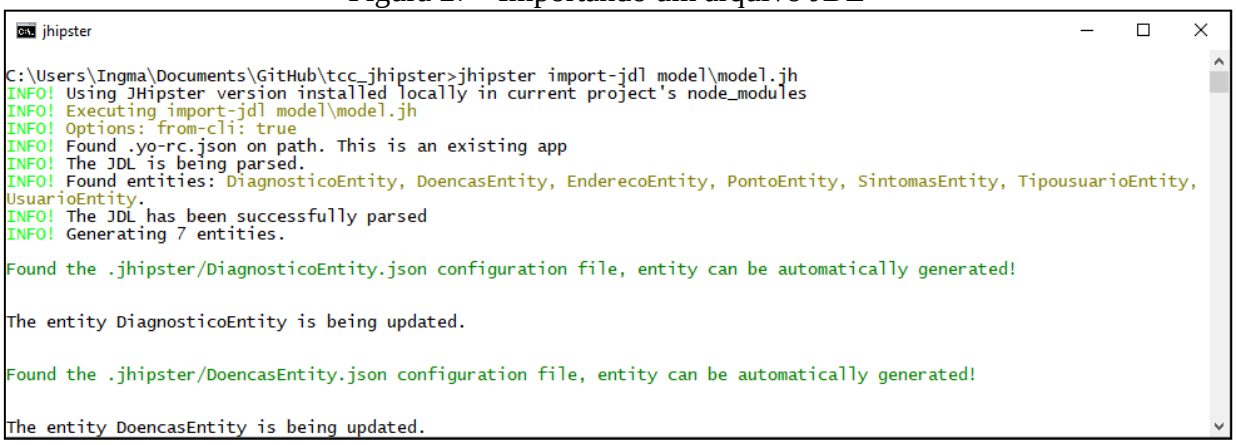
```
datasource:
  type: com.zaxxer.hikari.HikariDataSource
  url: jdbc:postgresql://localhost:5432/tcc_jhipster
  username: tcc_jhipster
  password: tcc_jhipster
  hikari:
    poolName: Hikari
    auto-commit: false
jpa:
  database-platform: io.github.jhipster.domain.util.FixedPostgreSQL95Dialect
  database: POSTGRESQL
  show-sql: true
  properties:
    hibernate.id.new_generator_mappings: true
    hibernate.connection.provider_disables_autocommit: true
    hibernate.cache.use_second_level_cache: true
    hibernate.cache.use_query_cache: false
    hibernate.generate_statistics: false
    hibernate.default_schema: tcc_jhipster
liquibase:
```

Fonte: elaborado pelo autor.

Com o acesso ao banco configurado, foi criada uma pasta dentro da estrutura chamada `model`, dentro dessa pasta foi salvo o arquivo de extensão .jh gerado pelo protótipo. Este nome pode ser qualquer nome de arquivo válido pelo sistema operacional. Para fins de validação este arquivo foi nomeado de `model.jh`.

Para que o JHipster gere os artefatos necessários para levantar a aplicação, o modelo gerado precisa ser importado e para isso o JHipster disponibiliza o comando `import-jdl`. Conforme Figura 27 é possível ver o comando em ação. Nessa importação são gerados todos os artefatos para as novas entidades, tanto para o *frontend*, quanto para o *backend*.

Figura 27 - Importando um arquivo JDL



Fonte: elaborado pelo autor.

3.3.1.1 Validação a partir do MER de exemplo 1

Conforme visto na seção 3.1.5, foram definidos dois MERs para auxiliar na validação do código JDL gerado pelo protótipo nesta seção, o MER de exemplo representa uma base de dados rodando num banco de dados MySQL e o código gerado encontra-se no Quadro 20.

Quadro 20 - JDL gerada a partir do MER de exemplo 1

```
entity DiagnosticoEntity {
	resultado String
	descricao String
	data LocalDate
}

... demais entidades

relationship OneToOne {
	PontoEntity{diagnosticoPonto} to
DiagnosticoEntity{pontoDiagnostico}
	... demais relacionamentos
}

relationship ManyToMany {
	DiagnosticoEntity{doencasDiagnostico} to
DoencasEntity{diagnosticoDoencas}
	... demais relacionamentos
}
```

Fonte: elaborado pelo autor.

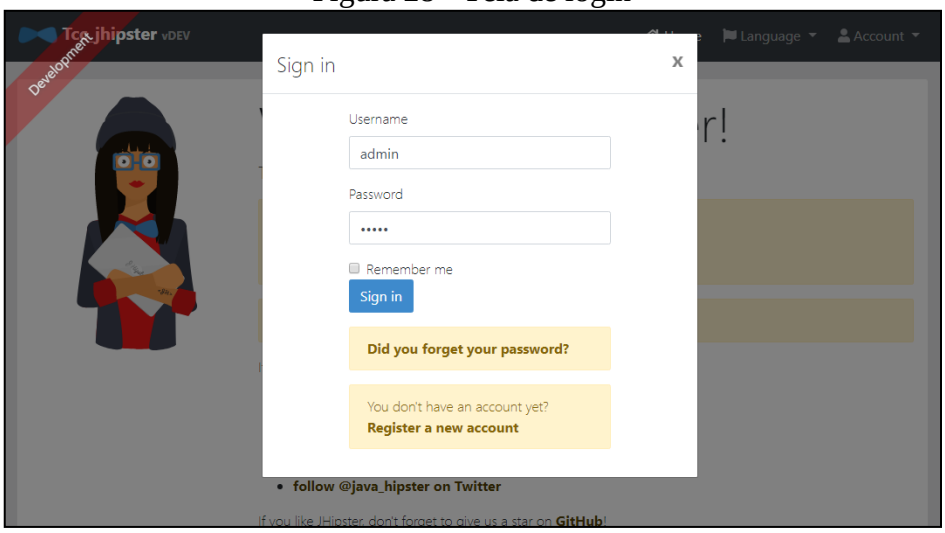
Com o código do Quadro 20 salvo no arquivo `model.jh` citado anteriormente, foi rodado o comando `import-jdl` conforme Figura 27 e logo após a criação dos artefatos a aplicação JHipster pôde ser iniciada. A inicialização da aplicação se dá através de dois comandos, ambos via *prompt* de comando, dentro da pasta do projeto:

a) `mvnw`: inicia a aplicação *backend*, que roda utilizando o Spring Boot. Quando executado após a importação, primeiramente são criadas as tabelas e relacionamentos;

b) `npm start`: inicia a aplicação *frontend*, que no projeto gerado foi escolhido para rodar usando o *framework* Angular.

Após iniciar o *backend* e *frontend*, é exibida uma página de entrada da aplicação gerada pelo JHipster (Figura 28). Nela é possível fazer a autenticação do sistema através do usuário *admin* e senha *admin*.

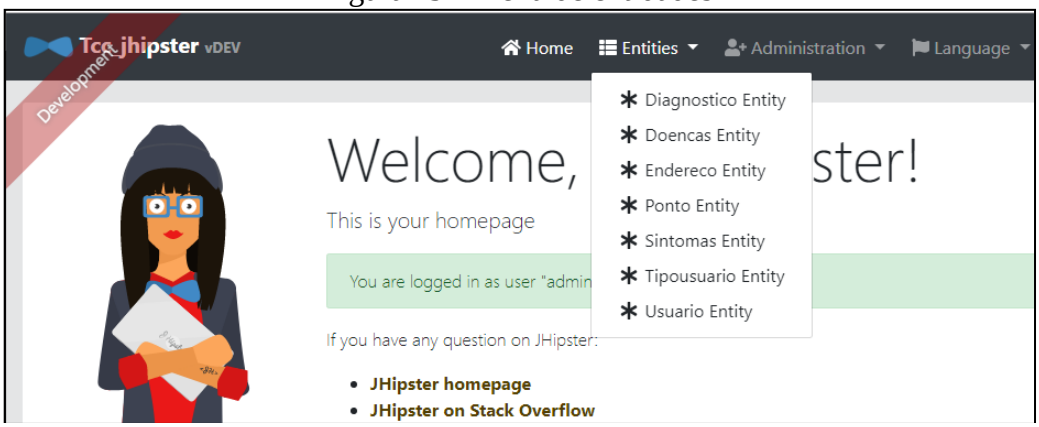
Figura 28 - Tela de login



Fonte: elaborado pelo autor.

Para gerenciar as entidades é disponibilizado um menu, conforme Figura 29, sendo que basta escolher uma entidade para poder fazer a listagem, edição, cadastro e exclusão.

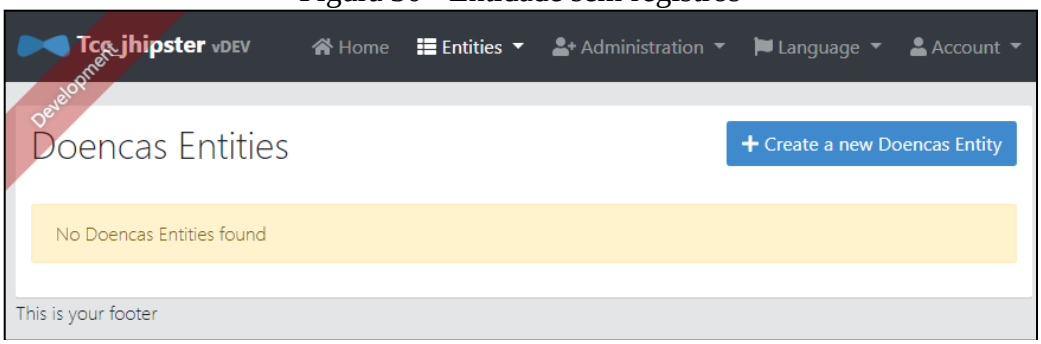
Figura 29 - Menu de entidades



Fonte: elaborado pelo autor.

A Figura 30 mostra a visão de uma entidade sem registros cadastrados.

Figura 30 - Entidade sem registros



Fonte: elaborado pelo autor.

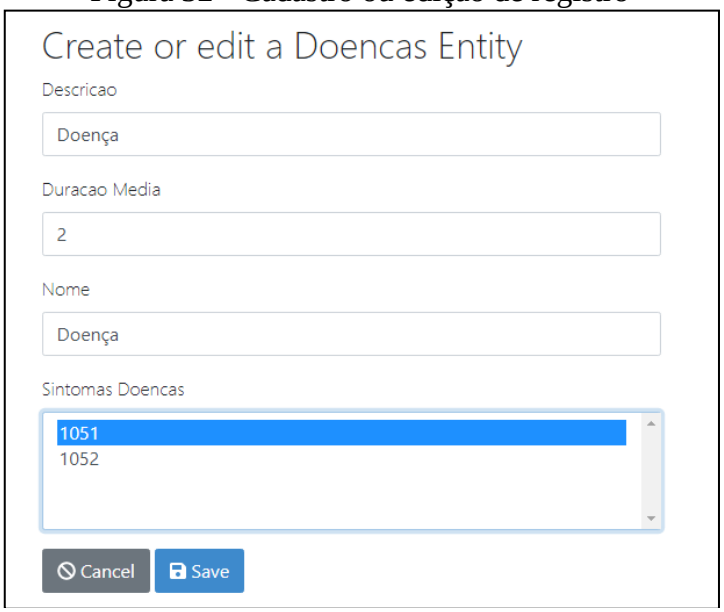
A Figura 31 mostra a visão da listagem de entidades cadastradas.

Figura 31 - Listagem de registros

Fonte: elaborado pelo autor.

A Figura 32 mostra a visão do cadastro ou edição de uma entidade. No JHipster, por padrão no *frontend*, os relacionamentos são criados na forma de caixa de seleção, e o valor exibido é o id do registro, o que torna a escolha um pouco confusa.

Figura 32 - Cadastro ou edição de registro



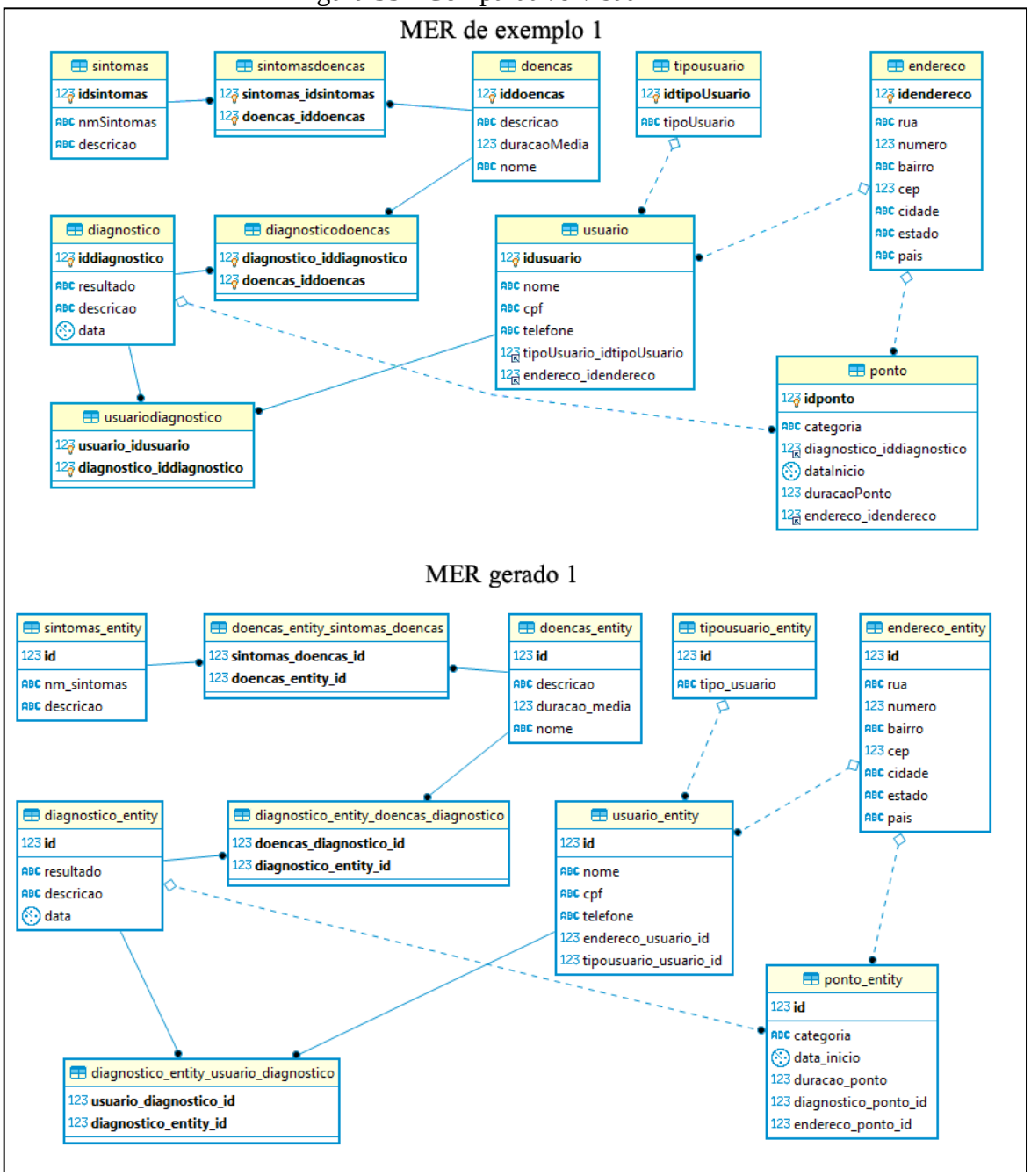
Fonte: elaborado pelo autor.

Com base nas informações listadas, fica evidenciado que o JHipster criou uma aplicação completa a partir das tabelas, campos e relacionamentos modelados no MER de exemplo 1.

Nesse caso, permite-se que sejam listados, cadastrados, editados e excluídos registros das tabelas contempladas.

Neste estudo de caso, a aplicação foi gerada numa base de dados distinta da origem, ou seja, o JHipster ficou responsável por criar as tabelas, campos, relacionamentos e demais objetos do banco. Dessa forma, outra maneira de validar o protótipo, foi comparando o MER da aplicação gerada, Figura 33, com o MER de exemplo 1. O que mudou foram apenas os nomes das tabelas, relacionamentos e chaves primárias, porém as ligações entre as tabelas permaneceram as mesmas.

Figura 33 - Comparativo visual MER 1

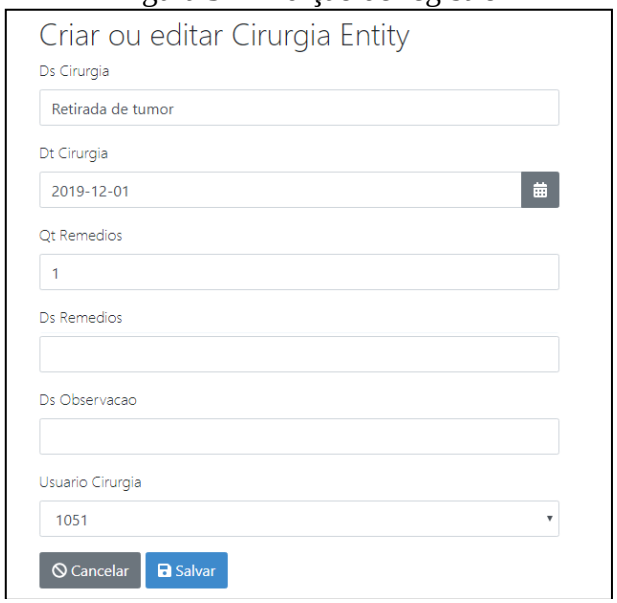


Fonte: elaborado pelo autor.

3.3.1.2 Validação a partir do MER de exemplo 2

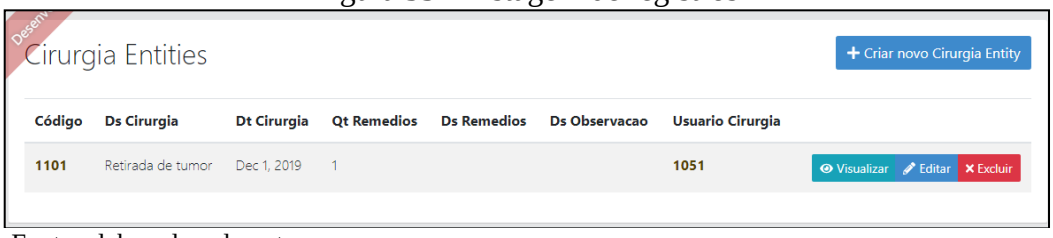
O processo de validação do protótipo a partir do MER de exemplo 2, foi similar ao feito no exemplo anterior, com a diferença de que o banco de dados usado foi o SQL Server. O padrão de telas segue exatamente o mesmo conforme mostrado na Figura 34 e Figura 35.

Figura 34 - Edição de registro



Fonte: elaborado pelo autor.

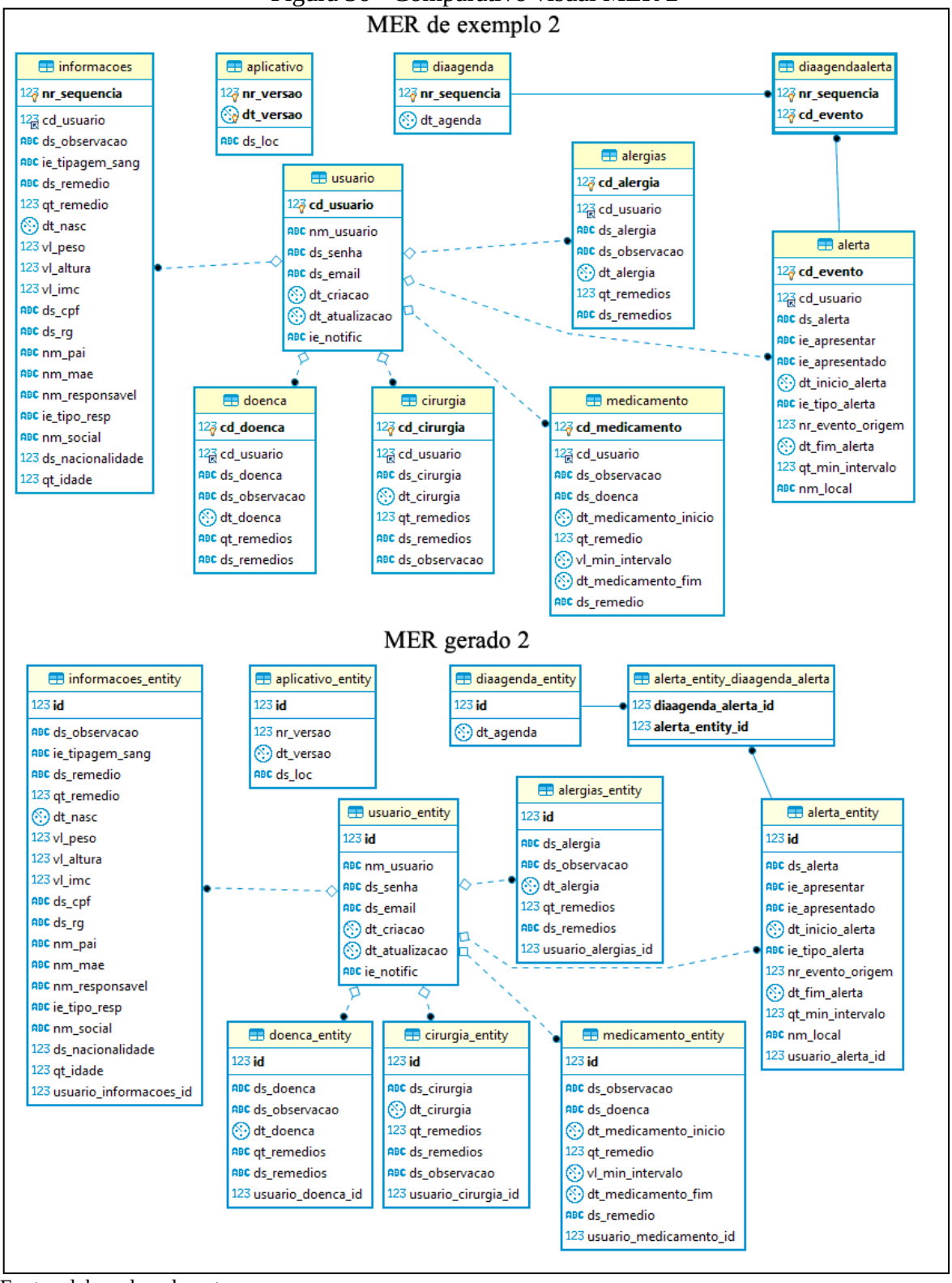
Figura 35 - Listagem de registros



Fonte: elaborado pelo autor.

A Figura 36 mostra a comparação visual entre o MER original e o MER gerado pelo JHipster, comprovando que as ligações continuam as mesmas, e o que muda são apenas os nomes das tabelas e relacionamentos, uma vez que o JHipster possui um padrão para a nomenclatura.

Figura 36 - Comparativo visual MER 2



Fonte: elaborado pelo autor.

3.3.2 Discussões

Conforme o Quadro 21, os trabalhos detalhados na seção 2.3 possuem características similares, e de alguma forma se propõem a auxiliar na criação de uma aplicação web, quais sejam Klug (2007), OpenXava (2019) e Rodriguez-Echeverria et al. (2016).

Como observado no Quadro 21, os trabalhos de OpenXava (2019) e Rodriguez-Echeverria et al. (2016) disponibilizam uma aplicação web completa, já o trabalho de Klug (2007) se propõe a ser uma ferramenta que gera apenas parte de uma aplicação web, necessitando de um projeto existente em que os artefatos gerados precisam ser adicionados ao projeto. O protótipo desenvolvido neste trabalho produz artefatos suficientes para que o JHipster possa levantar uma aplicação web completa, sem a necessidade de um projeto já existente.

No que se refere ao uso de módulos de segurança (autenticação e autorização) em OpenXava (2019) é necessário adquirir uma licença, as ferramentas de Rodriguez-Echeverria et al. (2016) não contempla esses módulos em sua estrutura. A ferramenta de Klug (2007) necessita que esses módulos sejam implementados em sua estrutura de projeto existente. Este trabalho usa o JHipster como gerador da aplicação, sendo que esses módulos fazem parte de seu núcleo de funcionamento.

Klug (2007) se propõe a ser uma ferramenta que gera os artefatos necessários para realizar o CRUD das entidades a partir de uma base de dados, as ferramentas de OpenXava (2019) e Rodriguez-Echeverria et al. (2016) contemplam a geração do CRUD sem extrair informação alguma da base. O protótipo desenvolvido, em conjunto do JHipster, contempla a geração do CRUD a partir da base de dados.

De todas as ferramentas analisadas, Openxava (2019) e Rodriguez-Echeverria et al. (2016) foram as que mais apresentaram características importantes para uma ferramenta que tenha como objetivo gerar uma aplicação web completa, o que não ocorre com a ferramenta elaborada por Klug (2007). Para suprir essa demanda, o protótipo desenvolvido neste trabalho atingiu o objetivo de prototipar um gerador de aplicações web utilizando a ferramenta JHipster, abordada na seção 2.2, para gerar uma aplicação web completa a partir de uma base de dados existente.

Como demonstrado na etapa de validação, na seção 3.3.1, o protótipo desenvolvido neste trabalho atendeu boa parte das características exibidas no Quadro 21. Entretando um ponto que o protótipo não contempla é o uso dos dados da base existente na aplicação gerada, pois o JHipster acaba gerando uma nova base. Este problema não ocorre no trabalho de Klug (2007),

que acaba usando toda a estrutura do banco de dados informado. As ferramentas desenvolvidas por Openxava (2019) e Rodriguez-Echeverria et al. (2016) também se aproveitam dos dados da base configurada, uma vez que não geram nenhum tipo de artefato a partir do banco de dados.

Quadro 21 - Comparativo entre os trabalhos correlatos

Características	Klug (2007)	OpenXava (2019)	Rodriguez-Echeverria et al. (2016)	Aguiar (2019)
Geração de aplicação web completa	Não	Sim	Sim	Sim[3]
Uso dos dados da base existente	Sim	Sim	Sim	Não
Geração a partir de uma base de dados existente	Sim	Não	Não	Sim
Depende de um projeto existente	Sim	Não	Sim	Sim
Módulo de segurança (autenticação e autorização)	Não	Versão paga	Sim	Sim[3]

Fonte: elaborado pelo autor (2019).

[3] A responsabilidade de gerar a aplicação completa, gerenciar a autenticação e atutorização é do JHipster.

4 CONCLUSÕES

Este trabalho apresentou o desenvolvimento de um protótipo de gerador de aplicação web com JHipster, em que a aplicação é gerada a partir de uma base de dados existente. Os objetivos propostos por este trabalho foram alcançados e através da implementação foi possível criar um mapeamento das estruturas de bases de dados existentes e convertê-las para o modelo que o JHipster compreende, ou seja, na linguagem JDL. Para tanto, uma pequena API acabou sendo desenvolvida para fazer a geração do código JDL.

A etapa de validação do projeto também se mostrou eficaz. Foi possível evidenciar que a aplicação web foi gerada e executada corretamente de acordo com os padrões do JHipster, e que as entidades geradas seguiam a estrutura da base original, bem como seus relacionamentos.

A principal contribuição da criação do protótipo é a possibilidade de escalar uma aplicação web a partir de um modelo de entidade relacionamento em pouco tempo, bem como na prototipação de uma nova aplicação que já tenha seu modelo definido. Da mesma forma, a nível acadêmico, o protótipo pode ser usado em disciplinas como Banco de Dados e Projeto de Software I e II, facilitando a criação das aplicações.

Como citado no final da seção 3.3, a principal limitação do protótipo é não aproveitar os dados da base existente usada como fonte para geração da aplicação web, pois o JHipster cria uma nova base e ignora os dados da base de referência. Outras limitações são: ignorar campos não nulos; não leva em consideração o tamanho de campos do tipo string; não exibe uma descrição intuitiva para os relacionamentos das tabelas no *frontend* (exibe apenas o id).

4.1 EXTENSÕES

Sugere-se como extensões do protótipo apresentado:

a) fazer com que o JHipster rode na mesma base usada como entrada para a aplicação, para que os dados existentes possam ser aproveitados;
b) permitir nos relacionamentos, que o usuário escolha qual campo será exibido no *frontend*, ao invés do id da tabela, para que seja possível ter uma visualição mais clara do registro que está sendo escolhido;
c) gerar código JDL com validações nos campos, como validar campos não nulos, tamanho de string, para garantir a integridade dos dados persistidos;
d) expandir a API de acesso a funcionalidades do JHipster, como rodar direto do protótipo, para facilitar a experiência do usuário, sem ser necessário usar linha de comando.

REFERÊNCIAS

BRAMBILLA, Marco; FRATERNALI, Piero. **Interaction Flow Modeling Language**: Model-Driven UI Engineering of Web and Mobile Apps with IFML. United States: Morgan Kaufmann, 2014. 422 p. (The MK/OMG Press).

DEITEL, H. M.; DEITEL, P. J. **Java**: como programar. 6. ed. Tradução Edson Furmankiewicz. São Paulo: Pearson, 2005.

DEVMEDIA. **Introdução ao Node.js**: Básico ao Avançado com o Node.js. 2019. Disponível em: <https://www.devmedia.com.br/nodejs/>. Acesso em: 01 dez. 2019.

FOWLER, Martin; PARSONS, Rebecca. **Domain-specific languages.** Boston: Addison-Wesley, 2011. 580 p. ISBN 0-321-71294-3

FRANKY, Maria Consuelo; PAVLICH-MARISCAL, Jaime A. Improving implementation of code generators: A regular-expression approach. In: CONFERENCIA LATINOAMERICANA EN INFORMATICA (CLEI), 38., 2012, Medellin. **Conferencia Latinoamericana En Informatica (CLEI)**. Medellin: Ieee, 2012. p. 1 - 10.

GHOSH, Debasish. **DSLs in ACTION**. Greenwich: Manning, 2011. 377 p.

GONÇALVES, Edson. **Dominando Java Server Faces e facelets utilizando Spring 2.5, Hibernate e JPA**. Rio de Janeiro: Ciência Moderna, 2008. 368 p.

HERRINGTON, Jack. **Code generation in action**. Greenwich, CT: Manning, 2003. 342 p.

JHIPSTER. **JHipster**. 2019. Disponível em: <https://www.jhipster.tech/>. Acesso em: 13 abr. 2019.

KLUG, Maicon. **Gerador de Código JSP Baseado em Projeto de Banco de Dados**. Orientador: Profª. Joyce Martins. 2017. 121 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) - Universidade Regional de Blumenau, Blumenau, 2017. Disponível em: <https://bu.furb.br//docs/MO/2008/329221_1_1.pdf>. Acesso em: 19 abr. 2019.

LIMA JUNIOR, Walter Teixeira. **O surgimento da nova camada complexa da Web e a apropriação doméstica das tecnologias digitais conectadas**. 2013. Disponível em: <https://www.researchgate.net/publication/258245252_O_surgimento_da_nova_camada_complexa_da_Web_e_a_apropriacao_domestica_das_tecnologias_digitais_conectadas>. Acesso em: 1 dez. 2019.

LIMA, Carlos Filipe da Silva. **Full Stack Application Generation for Insurance Sales based on Product Models.** 2016. 165 f. Dissertação (Mestrado) - Curso de Mestrado em Engenharia Informática, Instituto Superior de Engenharia do Porto, Porto, 2016. Disponível em: <https://recipp.ipp.pt/bitstream/10400.22/10829/1/DM_CarlosLima_2016_MEI.pdf>. Acesso em: 10 out. 2019.

LIVRAGHI, Marco. **Automatic generation of web based crud applications.** 2016. 113 f. Dissertação (Mestrado) - Curso de Master Of Science In Computer Engineering, Politecnico di Milano, Milano, 2016. Disponível em: <https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&cad=rja&uact=8&ved=2ahUKEwjp_Ied0ZTmAhUfFLkGHUmLBrcQFjAAegQIBBAC&url=https%3A%2F%2Fwww.politesi.polimi.it%2Fbitstream%2F10589%2F125742%2F1%2F2016_09_Livraghi.pdf&usg=AOvVaw3fnSaEvTsQL1sh8rUZiHk->. Acesso em: 15 set. 2019.

MAGNO, Danillo Goulart. **Aplicação da Técnica de Scaffolding para a Criação de Sistemas CRUD**. 2015. 99 f. Dissertação (Mestrado em Ciência e Tecnologia da Computação) – Universidade Federal de Itajubá, Itajubá, 2015. Disponível em: <http://repositorio.unifei.edu.br/xmlui/handle/123456789/197 >. Acesso em: 20 out. 2019.

MASCARENHAS, Maicon. **Linguagens de Domínio Específico**, Rio de Janeiro, 2017. Disponível em: <https://dcc.ufrj.br/~fabiom/dsl/Aula02.pdf>.Acesso em: 19 abr. 2019.

OPENXAVA. **OpenXava**. 2019. Disponível em: <https://www.openxava.org/>. Acesso em: 13 abr. 2019.

PEREIRA, Caio Ribeiro. **Node.js**: Aplicações web real-time com Node.js. São Paulo: Casa do Código, 2013. 185 p.

RAIBLE, Matt. **The JHipster mini-book**. Birmingham: C4Media, 2019. 170 p. ISBN: 978-1-329-63814-3. Disponível em: < https://www.infoq.com/minibooks/download/jhipster-mini-book-5?minBookCampRef= >. Acesso em 15 set. 2019.

RED HAT. **O que é transformação digital e como alcançá-la**. [S. l.], 2017. Disponível em: <https://www.redhat.com/pt-br/topics/digital-transformation/what-is-digital-transformation>. Acesso em: 4 jun. 2019.

RODRIGUEZ-ECHEVERRIA, Roberto et al. AutoCRUD - Automating IFML Specification of CRUD Operations. In: INTERNATIONAL WORKSHOP ON AVANCED PRACTICES IN MODEL-DRIVEN WEB ENGINEERING, 12., 2016, Roma. **Proceedings International Workshop on Avanced practices in Model-Driven Web Engineering**. Roma: Scitepress, 2016. p. 307 - 314.

ROGERS, David L. **Transformação digital**: repensando o seu negócio para a era digital. Tradução Afonso Celso da Cunha Serra. 1. ed. - São Paulo: Autêntica Business, 2017. 336p.

SILVA NETO, José Antonio da; VILAR, Rodrigo. **Geração de código baseado em metamodelos para segurança de sistemas de informação web**. 2018. 14 f. TCC (Graduação) - Curso de Sistemas de Informação - Bacharelado, Universidade Federal da Paraíba, Rio Tinto, 2018. Disponível em: <https://si.dcx.ufpb.br/wp-content/uploads/2019/03/TCC-Jos%C3%A9-Antonio.pdf>. Acesso em: 4 jun. 2019.

SHINDE, Kshitija; SUN, Yu. Template-Based Code Generation Framework for Data-Driven Software Development. In: 2016 4TH INTL CONF ON APPLIED COMPUTING AND INFORMATION TECHNOLOGY, 4., 2016, Las Vegas. **Proceedings Intl Conf on Applied Computing and Information Technology**. Las Vegas: Ieee, 2017. p. 55 - 60.

VOELTER, Markus. **DSL Engineering**: Designing, Implementing and Using Domain-Specific Languages. Stuttgart: Createspace Independent Publishing Platform, 2013. 558 p.

APÊNDICE A – Descrição dos Casos de Uso

Este Apêndice apresenta no Quadro 22 a descrição do Caso de Uso principal do protótipo desenvolvido neste trabalho.

Quadro 22 - Descrição do caso de uso UC01

Caso de uso:	UC01 – Gerar entidades e relacionamentos na linguagem JDL a partir de uma base de dados existente
Descrição:	Permitir ao ator gerar o código JDL (entidades e relacionamentos) equivalentes às tabelas e relacionamentos de uma base existente
Atores:	Usuário
Pré-condição:	Acesso via rede aos servidores de banco de dados.
Fluxo principal:	1. O protótipo apresenta tela de configuração do banco de dados, contendo as informações: a) tipo de banco de dados; b) nome ou endereço IP do servidor do banco de dados; c) porta do banco de dados; d) nome da base de dados; e) schema da base de dados; f) usuário e senha da base dados; g) botão para avançar para a próxima etapa. 2. O protótipo apresenta tela de carregamento dos metadados da base 3. O protótipo apresenta tela para ajustar os relacionamentos, contendo as informações: a) caixa de seleção de tabelas; b) lista de relacionamentos da tabela com caixa de seleção do tipo de relacionamento; c) botão para avançar para a próxima etapa; d) botão para voltar para a etapa de configuração do banco de dados. 4. O protótipo apresenta tela com o resultado da conversão dos metadados da base de dados em código equivalente da JDL, contendo as informações: a) caixa de texto editável com o código gerado; b) botão para lançar o código gerado na ferramenta *JDL Studio*; c) botão para salvar o código gerado em arquivo; d) botão para voltar para a etapa de ajuste de relacionamentos.

Fonte: elaborado pelo autor.

APÊNDICE B – INSTALAÇÃO DO JHIPSTER

Para que o JHipster possa ser utilizado é necessária a instalação do Node.js, que é um ambiente de execução JavaScript, assíncrono e orientado a eventos de código aberto, onde os desenvolvedores usam JavaScript para escrever scripts do lado do servidor (DEVMEDIA, 2019).

Pereira (2013, p. 7) afirma que assim como o Maven, o Node.js possui seu próprio gerenciador de pacotes, o NPM (Node Package Manager). Com o NPM é possível instalar dependências aos projetos que usam Node.js, bem como instalar outras ferramentas que também dependem do Node.js para executar.

Conforme descrito na seção 2.2, o JHipster é um gerador baseado no Yeoman, ambos são pacotes gerenciados pelo NPM. Portanto, para instalar o Yeoman e JHipster é necessário acessar um prompt de comando e digitar os comandos exibidos no Quadro 23.

Quadro 23 - Instalação do Yeoman e JHipster

```
npm install -g yo
npm install -g generator-jhipster
```

Fonte: elaborado pelo autor.

APÊNDICE C – DEPLOY A PARTIR DO CÓDIGO GERADO

O Quadro 24 exibe os comandos necessários para iniciar uma aplicação JHipster e importar o código gerado pelo protótipo desenvolvido.

Quadro 24 - Iniciando uma aplicação JHipster e importando o arquivo gerado

```
jhipster
? Which *type* of application would you like to create? Monolithic
application (recommended for simple projects)
? What is the base name of your application? app
? What is your default Java package name? com.aguilasa.app
? Do you want to use the JHipster Registry to configure, monitor and
scale your application? No
? Which *type* of authentication would you like to use? JWT
authentication (stateless, with a token)
? Which *type* of database would you like to use? SQL (H2, MySQL,
MariaDB, PostgreSQL, Oracle, MSSQL)
? Which *production* database would you like to use? PostgreSQL
? Which *development* database would you like to use? PostgreSQL
? Do you want to use the Spring cache abstraction? Yes, with the Ehcache
implementation (local cache, for a single node)
? Do you want to use Hibernate 2nd level cache? Yes
? Would you like to use Maven or Gradle for building the backend? Maven
? Which other technologies would you like to use? (deixar em branco)
? Which *Framework* would you like to use for the client? Angular
? Would you like to use a Bootswatch theme (https://bootswatch.com/)?
Default JHipster
? Would you like to enable internationalization support? Yes
? Please choose the native language of the application English
? Please choose additional languages to install Portuguese (Brazilian)
? Besides JUnit and Jest, which testing frameworks would you like to use?
Gatling, Protractor
? Would you like to install other generators from the JHipster
Marketplace? No
import-jdl model.jh
```

Fonte: elaborado pelo autor.

Para realizar o *deploy* em produção, basta digitar o comando exibido no Quadro 25 em um terminal. Este comando empacota tanto o *backend*, quanto o *frontend* em um arquivo de extensão *.jar*. O arquivo JAR fica disponível na pasta *target/* do projeto.

Quadro 25 - Comando para deploy em produção

```
mvnw -Pprod clean verify
```

Fonte: elaborado pelo autor.

Para executar a aplicação não é necessário fazer o *deploy* em um servidor de aplicação, basta executá-lo diretamente do arquivo JAR gerado (Quadro 26).

Quadro 26 - Executar backend diretamente do arquivo JAR

```
java -jar <nome_do_arquivo>.jar
```

Fonte: elaborado pelo autor.