UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
CURSO DE TECNOLOGIA EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

JOÃO PAULO MERLIN

DESENVOLVIMENTO DE UMA FERRAMENTA DE SCAFFOLDING PARA CRIAÇÃO DE CÓDIGO FONTE PARA FRONT-END

TRABALHO DE CONCLUSÃO DE CURSO

PATO BRANCO
2018

JOÃO PAULO MERLIN

DESENVOLVIMENTO DE UMA FERRAMENTA DE SCAFFOLDING PARA CRIAÇÃO DE CÓDIGO FONTE PARA FRONT-END

Trabalho de Conclusão de Curso de Graduação, apresentado à disciplina de Trabalho de Conclusão de Curso II, do Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas, da Universidade Tecnológica Federal do Paraná, Câmpus Pato Branco, como requisito parcial para obtenção do título de Tecnólogo.

Orientador: Prof. Vinicius Pegorini.

PATO BRANCO
2018

Ministério da Educação
Universidade Tecnológica Federal do Paraná
Câmpus Pato Branco
Departamento Acadêmico de Informática
Curso de Tecnologia em Análise e Desenvolvimento de Sistemas

TERMO DE APROVAÇÃO

TRABALHO DE CONCLUSÃO DE CURSO

DESENVOLVIMENTO DE UMA FERRAMENTA DE SCAFFOLDING PARA CRIAÇÃO DE CÓDIGO FONTE PARA FRONT-END

POR

JOÃO PAULO MERLIN

Este trabalho de conclusão de curso foi apresentado no dia 29 de novembro de 2018, como requisito parcial para obtenção do título de Tecnólogo em Análise e Desenvolvimento de Sistemas, pela Universidade Tecnológica Federal do Paraná. O acadêmico foi arguido pela Banca Examinadora composta pelos professores abaixo assinados. Após deliberação, a Banca Examinadora considerou o trabalho aprovado.

Banca examinadora:

Prof MSc Vinicius Pegorini
Orientador

Profª MSc Andreia Scariot Beulke

Profª Drª Beatriz Terezinha Borsoi

Prof. Dr. Edilson Pontarolo
Coordenador do Curso de Tecnologia em Análise e Desenvolvimento de Sistemas

Profª Drª Beatriz Terezinha Borsoi
Responsável pela Atividade de Trabalho de Conclusão de Curso

A Folha de Aprovação assinada encontra-se na Coordenação do Curso.

RESUMO

MERLIN, João Paulo. Desenvolvimento de uma ferramenta de *scaffolding* para criação de código fonte para *front-end*. 2018. 51f. Monografia (Trabalho de Conclusão de Curso) - Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas, Universidade Tecnológica Federal do Paraná, Câmpus Pato Branco. Pato Branco, 2018.

A Internet tem evoluído no sentido de amplitude de acesso e em termos de melhoria dos recursos que oferece – largura de banda e serviços novos e aprimorados. A Internet passou da simples disponibilização de páginas estáticas para uma plataforma de interação multimídia e ambiente de realização de negócios. A diversidade de recursos oferecidos e de sistemas que executam na Internet faz com que os usuários dos sistemas *web* sejam cada vez mais exigentes, tanto em termos de qualidade dos dados e de interatividade dos sistemas quanto da existência de novos recursos. Para os desenvolvedores isso acarreta uma busca quase que constante pelo aprendizado e uso de novas tecnologias e dos recursos que as linguagens e seus ambientes oferecem. Para que esses desenvolvedores se mantenham produtivos é essencial que eles possam utilizar recursos que facilitem e agilizem o desenvolvimento e o aprendizado de uso desses recursos, como os *frameworks*. O uso dos *frameworks web* tem aumentado entre os desenvolvedores, pois torna o desenvolvimento mais rápido e padronizado. Neste trabalho foi desenvolvido um *plugin* para a geração de código a partir de classes Java para o *framework web* Angular utilizando linguagem TypeScript. O *plugin* desenvolvido além de converter classes Java em código TypeScript também é capaz de gerar CRUDs básicos com base nas *annotations* criadas.

Palavras-chave: *Scaffolding*. Angular. TypeScript.

ABSTRACT

MERLIN, João Paulo Merlin. A *scaffolding* tool do generate front-end source code. 2018. 51f. Monografia (Trabalho de Conclusão de Curso) - Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas, Universidade Tecnológica Federal do Paraná, Câmpus Pato Branco. Pato Branco, 2018.

Internet has evolved in terms of breadth of access and advancement in terms of improving the features it offers - improved bandwidth and new services. Internet has gone from simply providing static pages to a multimedia interaction platform and business environment. The offered diversity of resources and systems running on Internet make users of web systems more and more demanding, both in terms of data quality and interactivity of systems as well as the existence of new resources. For developers this entails an almost constant search for the learning and use of new technologies and the resources that the languages and their environments offer. For these developers to remain productive it is essential that they can utilize resources that facilitate and accelerate the development and learning of these resources, such as frameworks. The use of web frameworks has increased among developers as it makes development faster and more standardized. In this work we have developed a plugin for generating code from Java classes for the Angular web framework in TypeScript language. The plugin developed beyond converting Java classes into TypeScript code is also capable of generating basic CRUDs based on the created annotations.

Keywords: Scaffolding. Angular. TypeScript.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE CÓDIGOS

LISTA DE ABREVIATURAS E SIGLAS

Ajax	*Assynchronous Javascript and XML*
CRUD	*Create, Retrieve, Update and Delete*
CSS	*Cascading Style Sheet*
DOM	*Document Object Model*
HTML	*Hypertext Markup Language*
IDE	*Frameworks, Integrated Development Environment*
MVC	*Model, View, Controller*
REST	*Representational State Transfer*
RIA	*Rich Internet Applications*

SUMÁRIO

1 INTRODUÇÃO

Este capítulo apresenta as considerações iniciais com o contexto de aplicação desenvolvida, os objetivos e a justificativa do trabalho. Por fim, está a organização do texto com a apresentação dos capítulos subsequentes.

1.1 CONSIDERAÇÕES INICIAIS

A Internet é uma ferramenta de comunicação com acesso bastante amplo em termos de território e com muitos recursos. Utilizada para a comunicação entre pessoas e acesso à informação, para a realização de negócios pelas empresas e pessoas físicas e para auxílio na realização de atividades por praticamente qualquer tipo de instituição. Aplicações que executam na *web* possuem vantagens como, por exemplo, custo mais baixo de manutenção, garantia de que a aplicação estará sempre disponível e portabilidade. Por esses e outros motivos muitas empresas estão desenvolvendo aplicações para o ambiente *web* ou migrando seus sistemas legados para poder desfrutar de todas essas possibilidades.

Nos últimos anos, a *web* passou por diversas mudanças e melhorias, no início o principal objetivo era mostrar documentos no formato de hipertexto. Nesse sentido, os navegadores foram evoluindo e a Internet passou para um novo estágio, a chamada Web 2.0. Muitas novidades foram surgindo, dentre elas estão as aplicações ricas para internet (*Rich Internet Applications* - RIAs), possíveis devido ao surgimento da tecnologia *Assynchronous Javascript and XML* (AJAX).

Para desenvolver aplicações web é necessário que o desenvolvedor tenha um bom nível de conhecimento, pois muitos processos são realizados e várias tecnologias são utilizadas para que o produto final seja obtido. Para tentar diminuir essa complexidade, o desenvolvimento *web* tem sido separado em camadas, como cliente e servidor. O servidor é responsável pelo acesso aos dados e por executar as regras de negócio da aplicação, enquanto a camada cliente é responsável por apresentar as informações ao usuário final. Dentro de cada camada existem padrões de projeto (*Design Pattern*) específicos. Na camada de servidor, por exemplo, está o padrão *Model, View, Controller* (MVC).

O desenvolvimento da camada de cliente é, geralmente, obtido como resultado de muito trabalho, pois para criar uma página web é necessário criar códigos *Hipertext Markup Language* (HTML), JavaScript e *Cascading Style Sheet* (CSS). Diante disso, muitos *frameworks* JavaScript foram criados, com o intuito de agilizar esse trabalho. Além dos *frameworks*, também existem geradores de interface, conhecidos como *scaffolding*. Uma ferramenta muito utilizada para esta finalidade é o Yeoman[1].

Os *frameworks* JavaScript são importantes ferramentas de desenvolvimento. Com eles, os desenvolvedores economizam uma quantidade significativa de tempo na construção de páginas *web* quando utilizam *frameworks* apropriados (SCHWARTZ, 2013).

Nesse sentido, o presente trabalho visa aplicar a técnica de *scaffolding* para a geração de código fonte para o Angular que é um *framework front-end web*. O *scaffolding* será aplicado a partir da camada de servidor, que utiliza a linguagem de programação Java. Com isso, pretende-se oferecer aos programadores uma forma mais simples e rápida de criar aplicações *web*.

1.2 OBJETIVOS

A seguir são apresentados os objetivos geral e específicos da realização deste trabalho.

1.2.1 Objetivo Geral

Desenvolver um *plugin* que a partir das classes criadas em Java gere o código TypeScript para utilização em aplicações *web* com o *framework* Angular.

1.2.2 Objetivos Específicos

- Desenvolver um *plugin* para a ferramenta Maven que realize a transformação das camadas de modelo e controle das aplicações Java em código TypeScript para Angular.
- Abstrair toda a camada de comunicação entre cliente e servidor, não sendo

[1] http://yeoman.io/

necessário para o desenvolvedor realizar as chamadas *Representational State Transfer* (REST).

1.3 JUSTIFICATIVA

Muitas empresas comerciais de software estão migrando seus sistemas para a plataforma *web*. O principal motivo para a realização deste trabalho é proporcionar mais agilidade para a criação desse tipo de sistema. Com a camada de serviço da aplicação é possível gerar o código necessário para a comunicação entre as páginas *web* e o servidor de aplicação.

O ritmo dinâmico e acelerado da Internet faz com que os usuários de aplicações *web* sejam cada vez mais exigentes. Ferramentas, princípios, convenções e padrões de desenvolvimento precisam ser utilizados pelos desenvolvedores para aumentar a produtividade, qualidade final e cumprir prazos cada vez menores. *Frameworks, Integrated Development Environment* (IDE), bibliotecas e geradores de código são algumas das ferramentas que programadores utilizam para se tornarem mais produtivos (MAGNO, 2015).

Frameworks web estão sendo muito utilizados por serem práticos e ser fácil desenvolver código neles. Por esse motivo, com o desenvolvimento deste trabalho pretende-se gerar código para o *framework web* Angular, que utiliza um código semelhante ao da linguagem Java e que, muitas vezes, é necessário escrever praticamente o mesmo código para atender necessidades semelhantes.

As duas tecnologias (Java e Angular) utilizadas neste trabalho podem ser escritas utilizando o padrão de projetos MVC e por esse motivo se torna possível a utilização da técnica de *scaffolding* para a geração automática de código.

Neste sistema será realizada leitura das classes Java, utilizando as *Applications Programming Interfaces* (*APIs) Reflection e Annotations* que são nativas da linguagem e permitem fazer qualquer operação de forma dinâmica e em tempo de compilação. Uma vez que se têm classes Java lidas e armazenadas na memória, pode-se, por meio de algumas configurações, gerar o código para a aplicação Angular.

1.4 ESTRUTURA DO TRABALHO

O Capítulo 2 apresenta a fundação teórica do trabalho. São apresentados conceitos de *Rich Internet Applications*, em seguida são apresentados conceitos de metaprogramação, as duas principais técnicas da linguagem Java que são *annotations* e *reflections* e, por fim, é apresentado sobre *scaffolding*. No Capítulo 3 são apresentadas as ferramentas, as tecnologias e o método para o desenvolvimento do trabalho. O resultado da realização deste trabalho é apresentado no Capítulo 4, por fim, no Capítulo 5 estão as considerações finais.

2 FUNDAMENTAÇÃO TEÓRICA

Este Capítulo apresenta a fundamentação teórica necessária para o desenvolvimento deste trabalho. A Seção 2.1 enfatiza a importância da utilização de interfaces ricas. A Seção 2.2 elucida os conceitos básicos de metaprogramação e suas principais ferramentas da linguagem Java: Annotations e Reflections. Na Seção 2.3 é apresentada a técnica de *scaffolding* e sua utilização nos *frameworks* Spring e Angular

2.1 RICH INTERNET APPLICATION

O paradigma *web* tradicional é baseado na apresentação de várias páginas (multi páginas) vinculadas e com interação pouco flexível. O surgimento das chamadas Aplicações Ricas para Internet, as RIAs, foi possibilitado pelas tecnologias de desenvolvimento como o Ajax e permite interação mais avançada e sofisticada com o usuário (MARCHETTO et al., 2008).

O desenvolvimento das RIAs trouxe novos desafios aos desenvolvedores (LAWTON 2008). Como exemplo pode ser citado o fato de que as aplicações baseadas em Ajax usam intensivamente comunicação assíncrona e que o programador faz a suposição de que a resposta do servidor será recebida imediatamente após o envio da requisição, com nenhum evento acontecendo entre o envio de cada mensagem. Nessa situação, poderá ocorrer intercalação das mensagens do servidor, chamadas (*callbacks*) trocadas e execução da aplicação sob um estado inválido do *Document Object Model* (DOM) (MARCHETTO et al., 2008).

2.2 METAPROGRAMAÇÃO JAVA

Programas que geram código são, às vezes, chamados de mataprogramas, a escrita de tais programas é chamada metaprogramação (IBM, 2017). Um metaprograma é um programa enriquecido com conectores para modificação de comportamento. Por padrão, todos esses

conectores estão desativados, o que significa que por padrão um metaprograma é semanticamente equivalente ao programa original (DURIEUX et al., 2017). Esses conectores podem ser configurados, ter parâmetros atribuídos de forma a automatizar tarefas de programação, como, as operações de *Create, Retrieve, Update and Delete* (CRUD) com banco de dados.

Frameworks Java atuais provêm mecanismos de extensão mais restritivos, que são as *annotations*. As *annotations* podem ser usadas para comentar o código e os metaprogramas executam bibliotecas em tempo de execução ou extensão de compiladores que executam as ações no código comentado. Embora cada plataforma permita estender a linguagem por meio de APIs e ferramentas correspondentes, ainda há muitas restrições. Por exemplo, somente é possível usar constantes em tempo de compilação como argumentos para anotações Java (SPINELLIS, 2008).

2.3 JAVA ANNOTATIONS API

Em 2004, as anotações (*annotations*) foram introduzidas em Java e desde lá essa técnica de programação tem sido amplamente utilizada. Cada vez mais programas usam *annotations* durante o tempo de compilação para a geração de código (CHODAREV, et al., 2014, LAKATOŠ, PORUBÄN, BACÍKOVÁ, 2013) e para configuração e reflexão em tempo de execução (HAVLICE, 2013).

Apesar de que o uso de *annotations* em ambas as instâncias é frequentemente muito simular, o acesso às *annotations* e modelo de programa, bem como as suas representações, são diferentes. Se o desenvolvedor precisa completar a mesma tarefa em tempo de compilação (*compile*) e de execução (*runtime*), ele precisa estar familiarizado com ambos e deve criar duas versões diferentes do código fonte. Isso ocorre porque o código que usa a API de processamento de *annotations* não pode ser usado em tempo de execução e vice-versa (PIGULA; NOSÁL, 2015).

2.4 JAVA REFLECTIONS API

Análise estática de código orientado a objeto é uma área de pesquisa desafiadora, tornada especialmente desafiadora pelas características de linguagens dinâmicas, conhecida como *reflection*. A Java Reflection API permite aos programadores dinamicamente inspecionar e interagir com outros conceitos de linguagens estáticas tais como classes, campos e métodos, por exemplo, para dinamicamente instanciar objetos, valorar campos e invocar métodos. Essas características dinâmicas da linguagem são úteis, mas o seu uso também causa impacto negativo na acurácia dos resultados de análise estática. Isso ocorre devido à indecidibilidade de resolução de nomes e tipos dinâmicos (LANDMAN; SEREBRENIK; VINJU, 2017).

Java Reflections APIs consistem de objetos que modelam um sistema Java. Esses meta objetos são segmentados em oito classes - *Java.lang.{Class,ClassLoader} and java.lang.reflection.{ Array,Constructor}, {Field,Member,Method,Proxy}* - totalizando 181 métodos públicos. Esses meta-objetos fornecem uma visão dos tipos de sistema em tempo de execução (LANDMAN; SEREBRENIK; VINJU, 2017).

A Figura 1 apresenta uma visão sumarizada da Java Reflection API como uma gramática livre de contexto que define a construção de referências para os meta-objetos.

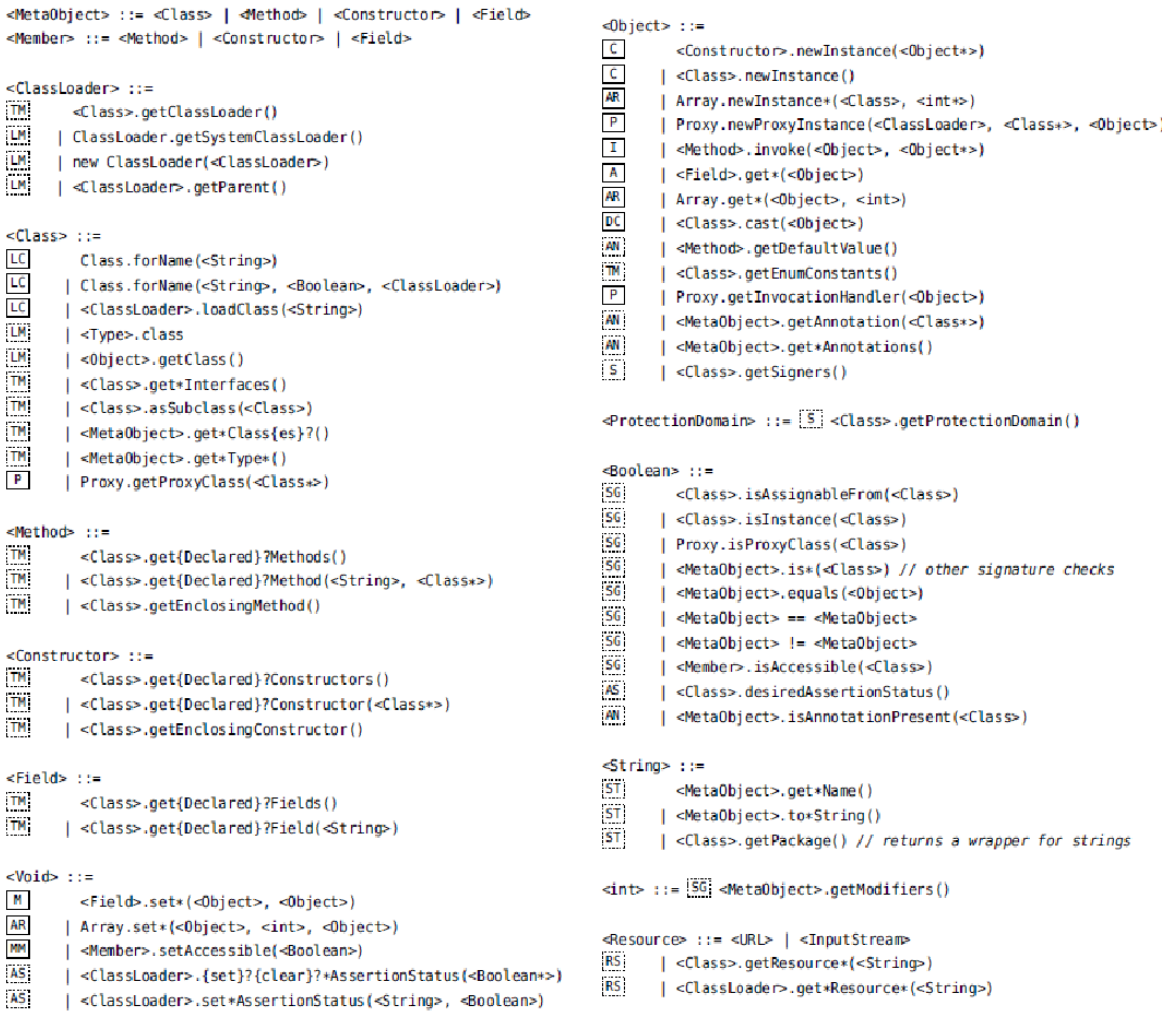

Figura 1 - Gramática de Java Reflection API
Fonte: Landman, Serebrenik e Vinju (2017, p. 508).

Na Figura 1 o "*" dentro de um terminal indica que há zero ou mais outros caracteres, o "*" dentro de um não terminal indica a existência de zero ou mais do referido não terminal e o "{x}?" indica uma parte opcional do terminal.

Uma linguagem consiste essencialmente de uma sequência de caracteres ou símbolos com regras para definir as sequências de símbolos que são válidas para a referida linguagem, ou seja, qual é a sintaxe da linguagem. A interpretação do significado de uma sequência válida de símbolos corresponde à semântica da linguagem. A sintaxe de linguagens é expressa na forma de uma gramática (RICARTE, 2003).

Uma gramática livre de contexto é usada como uma alternativa mais concisa para a

classe de diagramas ou definições de interface. Gramáticas livres de contexto têm sido utilizadas para realizar a análise sintática de linguagens de programação. Nem sempre é possível representar com esse tipo de gramática as restrições que são necessárias para algumas linguagens - por exemplo, exigir que todas as variáveis estejam declaradas antes de seu uso ou verificar se os tipos envolvidos em uma expressão são compatíveis. Entretanto, há mecanismos que podem ser incorporados às ações durante a análise - por exemplo, interações com tabelas de símbolos - que permitem complementar a funcionalidade da análise sintática (RICARTE, 2003).

Cada produção na Figura 1 define um número de alternativas para produzir um objeto do não-terminal definido. A gramática naturalmente agrupa ou retorna tipo que enfatiza a construção de meta objetos (imutáveis) e compreende métodos de objetivo similar usando expressões regulares. Com essa gramática podem ser completamente mapeados os 181 métodos públicos da API inteira em 58 produções, que são posteriormente agrupadas em 17 categorias funcionais, apresentadas no Quadro 1. Na figura 1, as categorias incluídas em um quadrado com bordas sólidas representam as características centrais (*core*) da linguagem dinâmica que simulam partes resolvidas estaticamente. As categorias incluídas em um quadrado com bordas pontilhadas representam APIs de suporte comparáveis ao código de bibliotecas Java normais.

Categoria	**Descrição**
LC Load Class	Entrada para a Reflection API, retorna referências para o meta-objeto a partir de uma *string*. Considerado prejudicial uma vez que pode executar inicializadores estáticos.
LM Lookup Meta Object	Entradas não prejudiciais para a Reflection API retornam referências para o meta-objeto.
TM Traverse Meta Object	Obtém referências para outros meta-objetos relacionados ao meta-objeto em uso no sistema Java.
C Construct Object	Cria uma nova instância de um objeto, equivalente ao construtor Java new <ClassName>()
P Proxy	Proxies são falsas implementações de interfaces, sendo que cada invocação é traduzida para um método *callback* simples. Muito

	prejudicial para análises estáticas, uma vez que não há equivalente estático para essa característica.
A Access Object	Lê o valor de um campo do objeto. Equivalente ao construtor Java obj.field
M Manipulate Object	Muda o valor de um campo. Equivalente ao construtor Java obj.field = newValue
MM Manipulate Meta Object	A única parte mutável da API: altera modificadores de acesso.
I Invoke Method	Invoca um método. Equivalente ao construtor Java recv.method(args)
AR Array	Cria, acessa e manipula *arrays*.
SG Signature	Testa a assinatura de um meta-objeto, por exemplo, se é um campo público.
AS Assertions	Acessa a manipula a *assertion flag* (flag de verificação) de uma classe.
AN Annotations	Acessa e itera anotações.
RS Resources	Lê recursos usando ClassLoader.
ST String representations	Obtém o nome dos elementos de um meta-objeto
S Security	Chamadas relacionadas à segurança
DC Casts	Transmite para uma classe meta-objeto dinamicamente. Equivalente ao construtor Java (Class)obj

Quadro 1 - Categorias funcionais da Java Reflection API
Fonte: Landman, Serebrenik e Vinju (2017, p. 509).

Da perspectiva de análise estática, a *Refection* API introduz características de linguagem dinâmica para uma outra linguagem estaticamente resolvida. A partir dessa perspectiva, a API pode ser dividida em duas partes (de acordo com o tipo de borda do quadro no qual está inserida a sigla que representa a categoria na primeira coluna do Quadro 1). Na primeira parte - no Quadro 1 - estão as características dinâmicas que simulam partes estaticamente resolvidas. Exemplo: o <Method>.invoke API é a equivalente dinâmica do método *inovacation* em Java. A segunda parte - no Quadro 1 - inclui métodos de suporte para

as características dinâmicas da linguagem. Exemplo: obter um método meta-objeto e métodos variados para acessar outros elementos de Java em tempo de execução.

2.5 SCAFFOLDING

A técnica de *scaffolding* gera as estruturas fundamentais para o funcionamento de uma aplicação *web* (MAGNO, 2015). *Scaffolding* é uma técnica suportada por alguns *frameworks* MVC, por meio da qual o programador pode especificar como o banco de dados da aplicação pode ser usado. O compilador ou *framework* usa essa especificação junto com *templates* de código pré-definidos para gerar o código final que a aplicação pode usar para criar, ler, atualizar ou excluir entidades do banco de dados.

A técnica de *scaffolding* cria automaticamente as estruturas mais significativas de um sistema de dados persistentes: o mapeamento entre objetos, banco de dados e interfaces CRUD. Essas estruturas estão intimamente relacionadas entre si e dependem do padrão MVC, portanto a técnica somente poderá ser utilizada para a criação de sistemas CRUD com padrão MVC. Outras ferramentas não relacionadas à *scaffolding* permitem gerar outros componentes, tais como: camada de segurança, integração de sistemas, testes unitários, etc. (ADAMUS et al., 2010).

É recomendado utilizar a técnica de *scaffolding* apenas uma vez, para gerar a estrutura inicial do projeto rapidamente, pois ao longo do desenvolvimento de um projeto, muitas alterações podem ser realizadas em uma classe e essas alterações podem ser sobrescritas com o uso posterior de *scaffolding*. Esse método de desenvolvimento de software é conhecido como prototipação (SOMMERVILLE, 2011).

Dependendo de como os geradores de código são utilizados, a técnica de *scaffolding* pode ser classificada como dinâmica (*run time*) ou estática (*design*). O *scaffolding* dinâmico é a geração e/ou alteração de código-fonte em tempo de execução. O *scaffolding* estático é o mais conhecido e indicado: gera código em tempo de compilação, por meio de comandos no console do *framework* (MAGNO, 2015).

3 MATERIAIS E MÉTODO

A ênfase deste capítulo está em reportar os materiais e o método utilizado para alcançar o objetivo do trabalho. Este capítulo é subdividido em duas seções, sendo uma para os materiais e outra para o método.

3.1 MATERIAIS

O Quadro 2 apresenta a lista de ferramentas e tecnologias utilizadas para o desenvolvimento deste trabalho.

Ferramenta / Tecnologia	Versão	Finalidade
Java	8	Linguagem de programação
TypeScript	2.4	Linguagem de programação
Angular	5	*Framework* JavaScript
Maven	3.5.0	Gerenciador de dependências
Intellij IDEA	2017.2.2	IDE de desenvolvimento

Quadro 2 - Lista de ferramentas e tecnologias

3.1.1 Java

Java é uma linguagem computacional completa, adequada para o desenvolvimento de aplicações baseadas na rede Internet, redes fechadas ou programas *stand-alone* (CAMPIONE, 1996).

A linguagem Java foi desenvolvida na 1ª metade da década de 90 nos laboratórios da Sun Microsystems com o objetivo de ser mais simples e eficiente do que suas predecessoras. O alvo inicial era a produção de software para produtos eletrônicos de consumo (fornos de micro-ondas, agendas eletrônicas, etc.). Um dos requisitos para esse tipo de software é ter código compacto e arquitetura neutra. Java é uma linguagem com muitos recursos para

ambientes distribuídos complexos como a rede Internet. Mas sua versatilidade permite ao programador ir além, oferecendo uma linguagem de programação de uso geral, com recursos suficientes para a construção de uma variedade de aplicativos que podem ou não depender do uso de recursos de conectividade (WUTKA, 1997).

3.1.2 TypeScript

O TypeScript é uma extensão do JavaScript criada para facilitar o desenvolvimento de aplicações. Todo programa em TypeScript é também JavaScript, pois a tradução é viável, porém, o TypeScript possui algumas peculiaridades que o JavaScript não possui. Por exemplo, TypeScript possui módulos, classes e interfaces (BIERMAN; ABADI; TORGERSEN, 2014). A ideia é que, futuramente, desenvolvedores de JavaScript consigam migrar para o TypeScript, visto que a linguagem tende a evoluir.

Neste trabalho, esta tecnologia foi utilizada na transformação dos arquivos Java para linguagem TypeScript.

3.1.3 Framework Angular

Angular é um *framework* de código aberto construído em JavaScript mantido pela Google. Foi originalmente desenvolvido em 2009 por Miško Hevery. Esse *framework* é utilizado para criação de *front-end* de aplicações *web*. Além disso, fornece também um conjunto de funcionalidades que tornam o desenvolvimento *web* mais fácil, tais como o *DataBinding*, *templates*, uso do Ajax, componentes, etc. (PANDA, 2014)

Todos os arquivos TypeScript gerados são compatíveis com as versões do Angular 2 ou mais atuais.

3.1.4 Maven

Na última década, notou-se a necessidade de gerir melhor o desenvolvimento de software para se criar as chamadas “soluções corporativas” a fim de garantir propriedades como: segurança, robustez, acessibilidade, integração, etc. Maven é uma ferramenta utilizada

para gerenciamento de projetos, e suas dependências, é um projeto de código livre mantido pela Apache Software Foundation (SANTOS FILHO, 2008).

Essa ferramenta possibilitou de fato o desenvolvimento do *plugin* que realiza o *scaffolding* das classes Java. O Maven é baseado no conceito de ciclo de vida de compilação, ou seja, ele é responsável por realizar o *build* e a execução dos *plugins* da aplicação.

Para quem constrói um projeto, é necessário aprender um pequeno conjunto de comandos para construir qualquer projeto Maven. Existem três ciclos de vida incorporados: *default*, *clean* e *site*. O ciclo de vida *default* lida com a implantação do projeto, o ciclo de vida *clean* lida com a limpeza do projeto, enquanto o ciclo de vida *site* lida com a criação da documentação do *site* do projeto.

3.1.5 Intellij IDEA

O Intellij IDEA é uma IDE de desenvolvimento, sua primeira versão foi lançada em Janeiro de 2001 e foi uma das primeiras IDEs Java disponíveis com recursos avançados de navegação por código e refatoração de código integrados.

3.2 MÉTODO

A seguir são apresentadas as principais atividades realizadas para a realização do trabalho:

a) Levantamento de requisitos

O levantamento de requisitos iniciou com uma conversa com vários programadores de diversas áreas de desenvolvimento *web, desktop*, analistas de requisitos e *designers* de interfaces. Esses profissionais relataram que poderia haver uma maneira de automatizar grande parte do processo produtivo, principalmente os que trabalham diretamente com criação de interfaces *web*.

b) Análise

Com base nos requisitos estabelecidos, foi gerado o diagrama classes para o sistema.

c) Projeto

Nesta fase foram estabelecidos alguns prazos, definidas as prioridades e foram

realizadas estimativas de tempo.

d) Implementação

Na implementação do sistema algumas tarefas foram criadas para seguir de forma contínua, por exemplo: criação das configurações para execução do *plugin*, criação dos objetos, leitura das classes Java, escrita dos arquivos para o *front-end*, entre outras.

e) Testes

Os testes foram feitos localmente utilizando várias versões do Spring Boot e também do Angular.

4 RESULTADOS

Este capítulo apresenta o que foi obtido como resultado do trabalho que é o desenvolvimento de um *plugin* Maven para a geração de código em *TypeScript* para Angular a partir de código Java.

4.1 ESCOPO DO SISTEMA

O projeto proposto tem por objetivo gerar automaticamente código para *front-end* a partir da leitura de *Annotations* no código de *back-end* escrito em Java. Inicialmente foi desenvolvido um *plugin* para Maven, podendo futuramente ser gerado *plugin* para Gradle.

Será necessário realizar a criação de *Annotations* para que o programador possa mapear suas entidades e *end-points* da aplicação, para poder executar o *plugin* que irá gerar o código fonte necessário.

O uso deste *plugin* funcionará da seguinte maneira: o programador vai configurar seu projeto Maven e deverá incluir as dependências necessárias para compilar o código fonte. Em seguida, o programador irá adicionar em suas entidades uma *Annotation* que identifica aquela classe como 'compilável', ou seja, que gerará código para o *front-end*. Também pode ser gerado o código para comunicação entre cliente e servidor. Para isso, o programador deve anotar suas classes e métodos que disponibilizam uma API REST. Nesta anotação alguns parâmetros são esperados, como, por exemplo, a *Uniform Resource Locator* (URL) que irá ser solicitada e o método da chamada Ajax.

A estrutura do código gerado seguirá os padrões definidos pelo Google Inc. no *framework* Angular. Serão geradas todas as classes necessárias para a criação de uma tela. Exemplo: existe uma classe no *back-end* chamada 'Usuario' e uma classe chamada 'UsuarioController' responsável pelas requisições Ajax. Com isso, será gerado para o *front-end* uma pasta chamada 'usuario' dentro da qual estarão algumas classes *TypeScript*, são elas: 'usuario.ts', 'usuario-module.ts', 'usuario-service.ts', 'usuario-component.ts', 'usuario-component.html' e 'usuario-component.scss'.

4.2 MODELAGEM DO SISTEMA

O projeto foi dividido em 3 camadas, são elas:

a) Parser

É responsável por ler as classes Java e suas propriedades usando reflexões. Usará classes responsáveis por processar os tipos de dados correspondentes. Por exemplo, se o tipo da propriedade for um “List<Pessoa>” a classe Pessoa também deverá ser analisada.

b) Compiler

Transforma todos os modelos Java mapeados para os modelos Typescript. Ele irá utilizar uma tabela de propriedades para analisar os atributos, gerando, assim, sempre o tipo compatível com a linguagem. O Quadro 3 mostra os principais tipos de variáveis que o *compiler* irá resolver durante o procedimento de conversão.

Java type	**TypeScript type**
Object	any
byte, Byte, short, Short, int, Integer, long, Long	number
float, Float, double, Double	number
boolean, Boolean	boolean
char, Character	string
String	String
BigDecimal, BigInteger	number
Date	Date, string ou number (configurável)
UUID	string
T[]	T[]
Collection<T>	T[]
Map<String, T>	{ [index: string]: T }

Quadro 3 - Mapeamentos de tipos

c) Emitter

Com os modelos lidos e processados, essa camada é responsável por escrever e salvar

os arquivos TypeScript.

Para cada classe ou interface Java, o gerador exibe a classe ou interface correspondente. O nome dos arquivos é baseado no nome simples da classe Java, podendo ser configurados prefixos e sufixos no *plugin*.

O resultado esperado deverá seguir o exemplo apresentado nas Listagens 1 e 2, as quais apresentam a classe Java original e a classe que deve ser gerada em TypeScript, respectivamente.

```java
public class Usuario {
   private Long id;
   private String nome;
   private String email;
   private Boolean ativo;
   private List<String> permissoes;
}
```

Listagem 1 - Classe Usuario.java

```typescript
export class Usuario {
   id: number;
   nome: string;
   email: string;
   ativo: boolean;
   permissoes: string[];
}
```

Listagem 2 - Classe Usuario.ts

Na Figura 2 é apresentado o diagrama das classes que representam um objeto modelo. Todas as entidades anotadas com a *annotation* '@TsModel', serão lidas pelo Parser e gerados novos objetos.

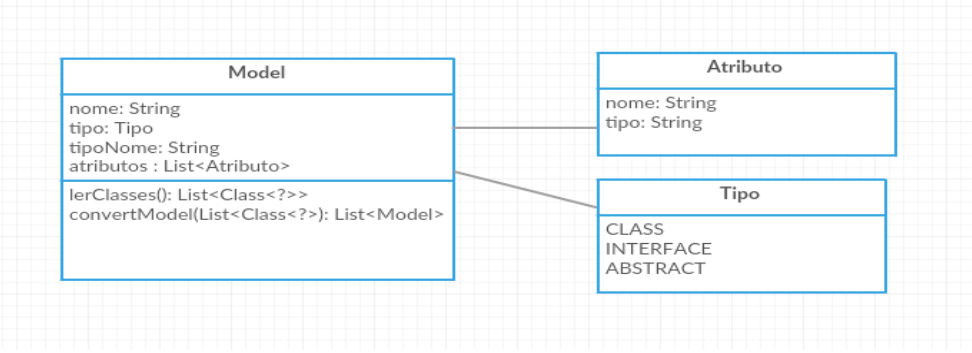

Figura 2 - Objetos que representam o arquivo TypeScript

Assim que todas as classes forem lidas, o *plugin* realiza a conversão dos tipos de dados Java para os tipos TypeScript, conforme apresentado no Quadro 3.

Foi utilizado um *enum* com os tipos de dados conhecidos, como String, Integer, Boolean etc. Todos os atributos passarão por uma validação utilizando o 'instanceof' da classe. Se for um atributo conhecido, ele será alterado, caso não seja conhecido, deverá procurar se esse atributo não é um relacionamento que utiliza outra classe anotada com '@TsModel'. Se mesmo assim o Compiler não encontrar o tipo, será utilizado por *default* o tipo 'any'.

O próximo passo que o *plugin* realiza é pegar esses objetos gerados e gravar nos arquivos TypeScript.

Para a gravação dos arquivos, foi utilizado um *template engine* chamado Freemarker. Este *framework* permite que seja criado *templates* para criação dos arquivos TypeScript, possibilitando, ainda, a passagem dos objetos por parâmetros para renderizar o arquivo corretamente, como pode ser observado no exemplo apresentado na Figura 3.

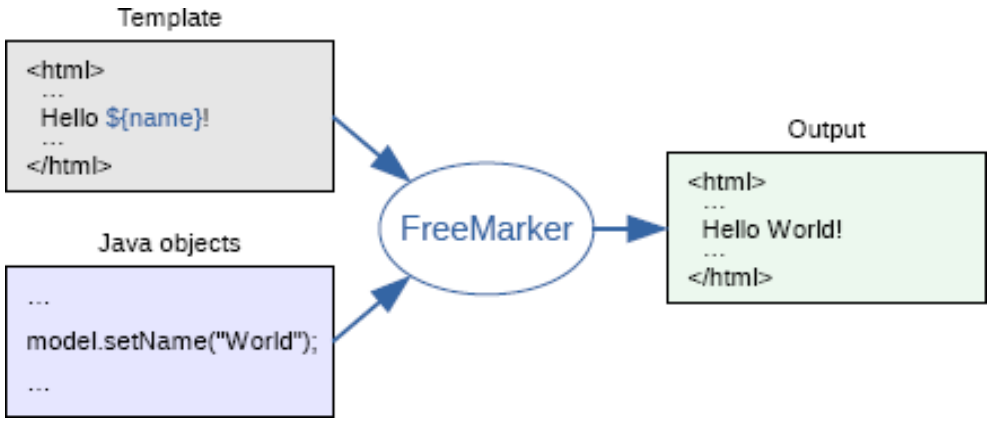

Figura 3 - Exemplo de uso do Freemarker em HTML.

Para que todas essas funcionalidades sejam executadas, foi criado um *plugin* Maven, que basicamente executa um comando quando o projeto for compilado. Além desse *plugin*, também deverá ser adicionada uma dependência com as *annotations* que poderão ser utilizadas.

As principais partes da configuração do projeto são:

- groupId - é a identificação do projeto e deve ser único. O groupId deve seguir as regras do nome do pacote java, isso significa que ele deve começar com um nome de domínio invertido que você controla.
- artifactId - é o nome do *jar* sem a versão.
- version - é a versão do plugin.
- goal - é o nome do *mojo* que será executado dentro do *plugin*.
- phase - é a fase do ciclo de vida do *maven*, *process-classes* significa que irá processar as classes após a compilação da aplicação.

Na Listagem 3 é possível visualizar um exemplo como o *plugin* desenvolvido neste trabalho deverá ser configurado em um novo projeto.

```
<plugin>
   <groupId>com.typescript-generator</groupId>
   <artifactId>typescript-generator-maven-plugin</artifactId>
   <version>1.0.0</version>
   <executions>
      <execution>
         <id>generate</id>
         <goals>
            <goal>generate</goal>
         </goals>
         <phase>process-classes</phase>
      </execution>
   </executions>
</plugin>
```

Listagem 3 - Configuração do *plugin maven*.

O Diagrama de classes exibido na Figura 4 mostra a estrutura do projeto. A classe GeradorPrincipal é responsável por ler e carregar para memória todas as classes do projeto que possuam a *annotation* '@TsModel' em sua declaração.

A classe Parser é responsável pela conversão das classes carregadas pelo GeradorPrincipal para um objeto chamado Model, neste momento também é realizada a conversão dos tipos de variáveis Java para TypeScript. Dentro da camada de conversão

existem classes para converter os objetos de acordo com o tipo de arquivo que será gerado, por exemplo, o arquivo HTML terá um *converter* específico, assim como os demais arquivos.

Após o carregamento e a conversão finalizada, a classe TemplateGerador é invocada passando todos os objetos convertidos, para que seja realizada a escrita dos novos arquivos e finalizada a operação.

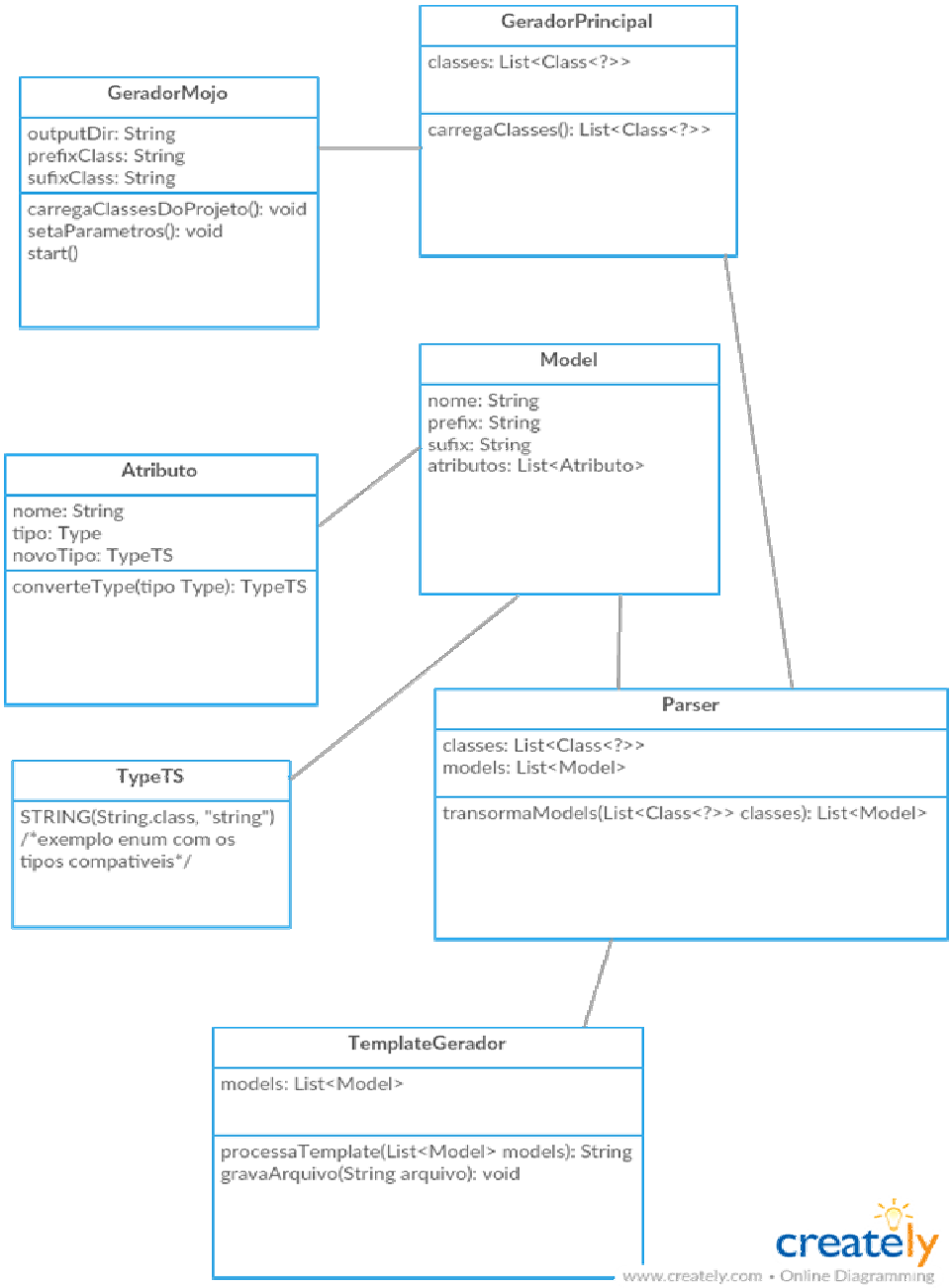

Figura 4 - Diagrama de classes

Na Figura 5 é exibido o diagrama de atividades que representa o funcionamento do *framework*. O sistema irá identificar todas as classes com a *Annotation* específica para conversão dos atributos para TypeScript, após a leitura e a conversão dos tipos correspondentes o *plugin* irá gerar os arquivos e então gravá-los no diretório configurado.

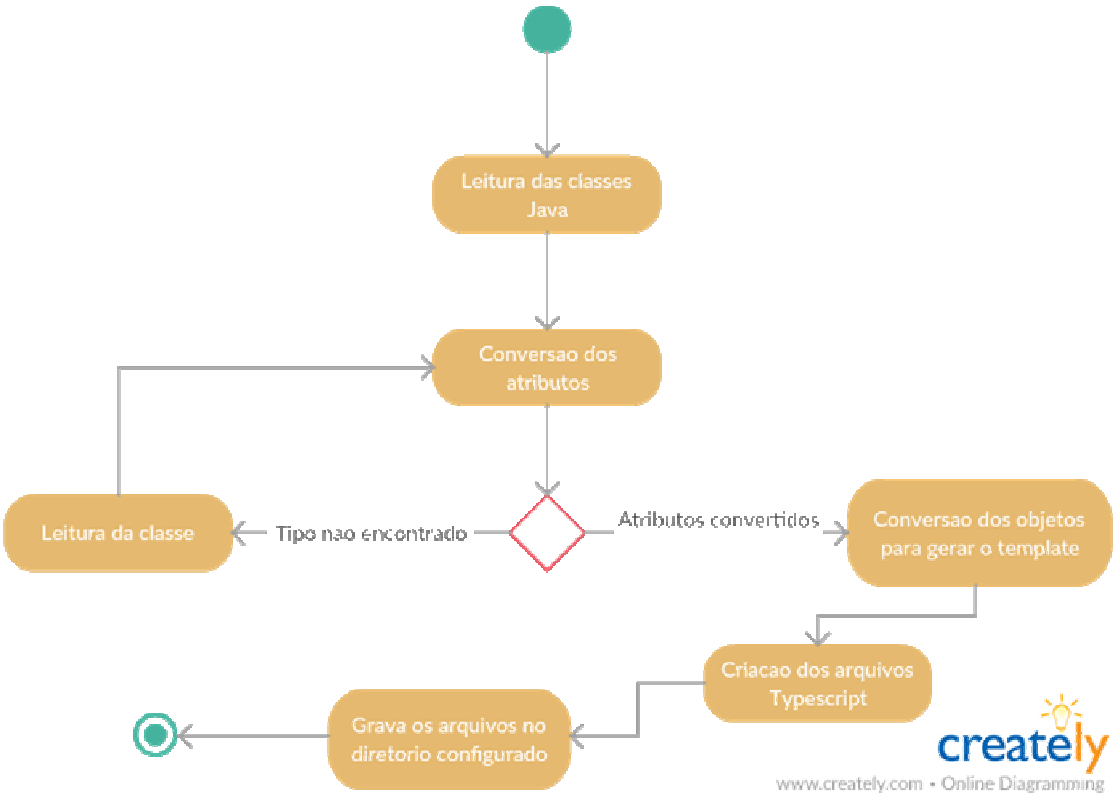

Figura 5 - Diagrama de atividade

4.3 APRESENTAÇÃO DO SISTEMA

Por se tratar de um *framework*, não há telas de interação com o usuário, porém algumas telas de CRUD poderão ser geradas automaticamente, além de todas as classes utilizadas no *front-end.*

O *framework* é iniciado a partir do seguinte comando Maven:

```
clean install typescript-generator:generate
```

Lembrando que é necessário configurar corretamente o *plugin* no arquivo pom.xml. Com este comando a aplicação irá limpar, compilar e gerar os arquivos TypeScript.

A Figura 6 representa a listagem de usuários gerada a partir da classe Usuario.java, apresentada na Listagem 1. Nesta tela, é possível incluir um novo registro e alterar ou remover registros existentes. Nessa listagem, todos os atributos da entidade são mostrados na tabela, exceto os campos marcados com a Annotation @TsIgnore.

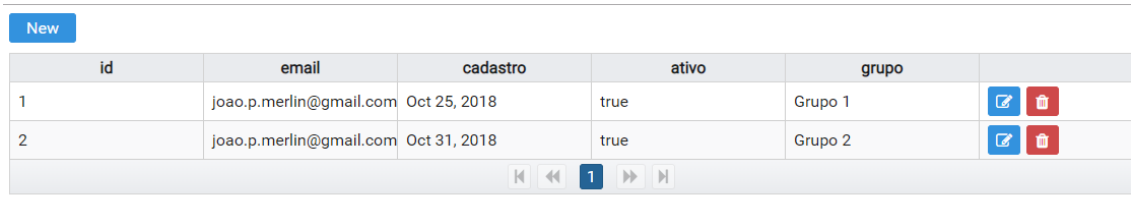

Figura 6 - Listagem principal

As telas de cadastro funcionam de maneira semelhante: o usuário preenche o formulário e em seguida clica no botão save ou cancel. A Figura 7 apresenta um formulário de usuário gerado automaticamente a partir da entidade *User*. Neste exemplo o campo 'id' está desativado, pois neste atributo está sendo utilizada a *Annotation* @GeneratedValue.

Todos os componentes são gerados de acordo com o tipo de variável utilizada, por exemplo: o campo cadastro é um Date, então um componente do tipo Calendar é gerado.

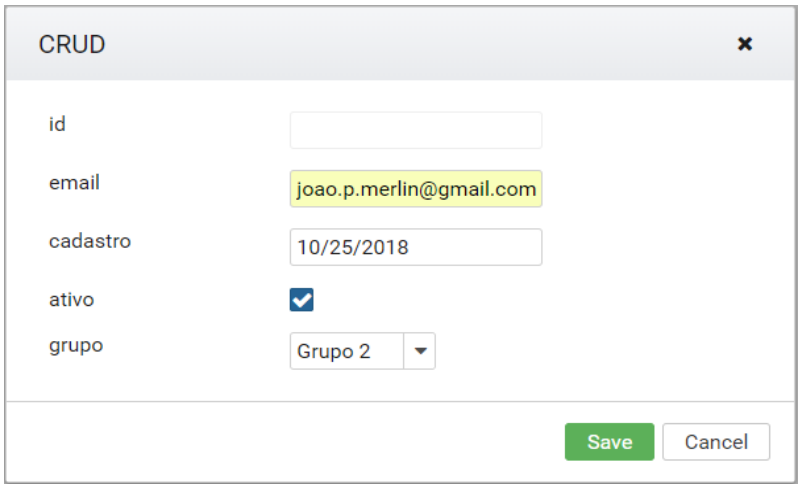

Figura 7 - CRUD

Para cada tipo de dados de um atributo, um componente Angular correspondente é renderizado. A Figura 8 representa o componente Calendar, utilizado quando um atributo do tipo Date é utilizado.

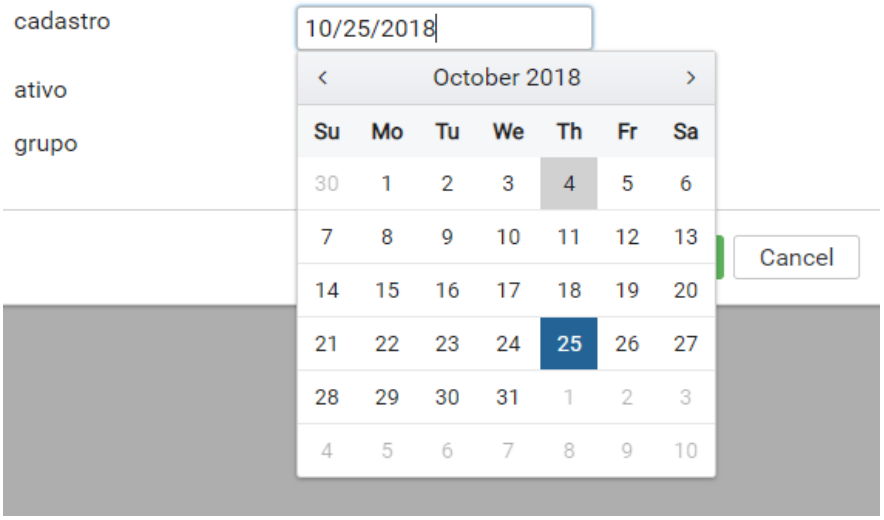

Figura 8 - Componente Calendar

A Figura 9 representa o componente Checkbox, utilizado quando um atributo do tipo Boolean é utilizado.

Figura 9 - Componente Checkbok

Na Figura 10 é possível visualizar o componente Dropdown, utilizado quando há um relacionamento com outra entidade. Neste caso a entidade relacionada precisa estar anotada com *@TsModel*.

Figura 10 - Componente Dropdown

Na listagem do CRUD, quando a opção excluir é selecionada uma confirmação é mostrada na tela, conforme Figura 11.

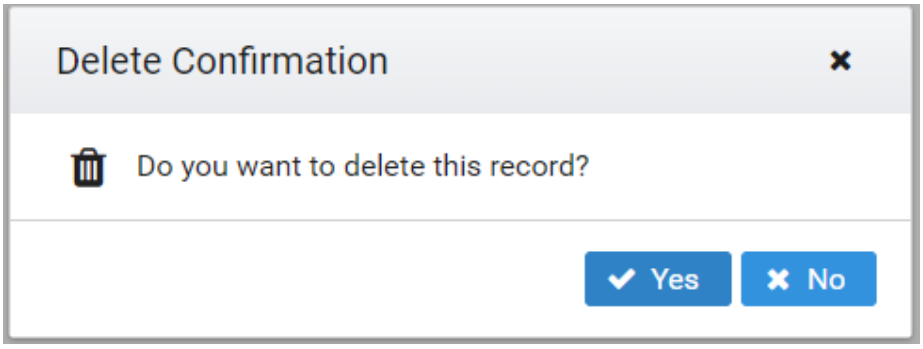

Figura 11 - Confirmação remover item

Todos os CRUDs existentes no sistema são gerados seguindo o mesmo padrão, sendo que, para cada um deles, serão gerados todos os arquivos necessários para o correto funcionamento do *front-end.* Para a entidade User serão gerados os arquivos: ‘user.ts’, ‘user-module.ts’, ‘user-service.ts’, user-component.ts’, ‘user-component.html’ e ‘user-component.scss’.

O arquivo user.ts é responsável por representar a entidade User do servidor. O arquivo user-service.ts é responsável pela comunicação entre o cliente e o servidor, nele estão todos os métodos existentes no *controller* Java. Os arquivos user-component.ts, user-componente.html e user-component.scss são responsáveis pelo controle e visualização da interface. Por fim, o arquivo user-module.ts é responsável por agrupar todos esses arquivos, possibilitando exportar esses componentes para outros projetos.

Seguindo o exemplo da classe Usuario, temos a classe UsuarioController responsável pelos métodos de CRUD, seguindo esta classe, o plugin irá gerar todos os arquivos citados acima. Na listagem 4 temos o exemplo do CRUD de usuário e nas seguintes listagens os arquivos que foram gerados para o Angular.

```java
@TsComponent(value = User.class, crud = true)
@RestController
@RequestMapping("user")
public class UserController {

    @Autowired private UserData data;

    @GetMapping
    @TsCrudFindAll
    public List<User> findAll() { return data.findAll(); }

    @GetMapping("{id}")
    @TsCrudFindOne
    public User findOne(@PathVariable("id") Long id) { return data.findOne(id); }

    @PostMapping
    @TsCrudSave
    public User save(@RequestBody User user) { return data.save(user); }

    @DeleteMapping("{id}")
    @TsCrudDelete
    public void delete(@PathVariable("id") Long id) { data.delete(id); }

}
```

Listagem 4 – Classe UsuarioController

A listagem 5 mostra a classe de serviço gerada para o exemplo acima, esta classe é responsável pela comunicação entre a aplicação Angular e o servidor.

```typescript
@Injectable()
export class UsuarioService {

  constructor(private http: HttpClient) {
  }

  public delete(id: number): Observable<void> {
    return this.http.delete<void>(`/user/${id}`);
  }

  public save(arg0: Usuario): Observable<Usuario> {
    return this.http.post<Usuario>(`/user`, arg0);
  }

  public findAll(): Observable<Usuario[]> {
    return this.http.get<Usuario[]>(`/user`);
  }

  public findOne(id: number): Observable<Usuario> {
    return this.http.get<Usuario>(`/user/${id}`);
  }

}
```

Listagem 5 – Classe UsuarioService

A listagem 6 mostra a classe que gerencia a tela de CRUD também gerada conforme os métodos existentes na classe UsuarioController. Além dos métodos existentes na classe, alguns são gerados para poder interagir com a tela, exemplo: new e edit.

```
@Component({
  selector: 'app-usuario',
  templateUrl: './usuario.component.html',
  styleUrls: ['./usuario.component.css']
})
export class UsuarioComponent implements OnInit {

  listUsuario: Usuario[];
  selectedUsuario = new Usuario();
  showModal = false;
  grupoOptions: Grupo[] = [];

  constructor(private usuarioService: UsuarioService,
              private grupoService: GrupoService,
              private confirmationService: ConfirmationService) {
    this.grupoService.findAll().subscribe( options: e => this.grupoOptions = e);
  }

  ngOnInit(): void {
    this.findAll();
  }

  delete(id: number) {
    this.confirmationService.confirm({
      message: 'Do you want to delete this record?',
      accept: () => {
        this.usuarioService.delete(id).subscribe( options: e => {
          this.findAll();
        });
      }
    });
  }

  save(arg0: Usuario) {
    this.usuarioService.save(arg0).subscribe( options: e => {
      this.findAll();
      this.showModal = false;
    });
  }

  findAll() {
    this.usuarioService.findAll().subscribe( options: e => {
      this.listUsuario = e;
    });
  }

  findOne(id: number) {
    this.usuarioService.findOne(id).subscribe( options: e => {
    });
  }

  new() {
    this.selectedUsuario = new Usuario();
    this.showModal = true;
  }

  edit(usuario: Usuario) {
    this.selectedUsuario = JSON.parse(JSON.stringify(usuario));
    this.showModal = true;
  }
}
```

Listagem 6 – Classe UsuarioComponent

A listagem 7 mostra a parte da listagem dos dados no arquivo HTML, neste caso todos os atributos lidos na classe Usuario são listados na tabela.

```
<p-table [value]="listUsuario" [paginator]="true" [rows]="10">
  <ng-template pTemplate="header">
    <tr>
      <th>id</th>
      <th>email</th>
      <th>cadastro</th>
      <th>ativo</th>
      <th>grupo</th>
      <th></th>
    </tr>
  </ng-template>
  <ng-template pTemplate="body" let-item>
    <tr>
      <td>{{item.id}}</td>
      <td>{{item.email}}</td>
      <td>{{item.cadastro | date}}</td>
      <td>{{item.ativo}}</td>
      <td>{{item.grupo.nome}}</td>
      <td>
        <button type="button" pButton (click)="edit(item)" icon="fa-edit"></button>
        <button type="button" pButton (click)="delete(item.id)" icon="fa-trash" class="ui-button-danger"></button>
      </td>
    </tr>
  </ng-template>
</p-table>
```

Listagem 7 – Arquivo HTML da classe UsuarioComponent (Listagem)

A listagem 8 mostra o formulário gerado. O formulário e a listagem estão no mesmo arquivo HTML. O formulário é gerado em uma modal, e só é apresentada quando for solicitado uma inclusão ou alteração de um registro.

Todos os componentes são gerados de acordo com os atributos encontrados na classe Usuario, também é feita uma verificação dos tipos de atributos, no exemplo a seguir temos um Date, onde é gerado um componente de seleção de data, e um Boolean, onde é gerado um componente do tipo checkbox.

```
<p-dialog header="CRUD" [(visible)]="showModal" [modal]="true" [width]="600">

  <div class="ui-g">
    <div class="ui-g-4">id</div>
    <div class="ui-g-8">
      <input type="number"
             pInputText
             name="id"
             disabled
             [(ngModel)]="selectedUsuario.id"/>
    </div>
  </div>

  <div class="ui-g">
    <div class="ui-g-4">email</div>
    <div class="ui-g-8">
      <input type="text"
             pInputText
             name="email"
             [(ngModel)]="selectedUsuario.email"/>
    </div>
  </div>

  <div class="ui-g">
    <div class="ui-g-4">cadastro</div>
    <div class="ui-g-8">
      <p-calendar name="cadastro"
                  appendTo="body"
                  [(ngModel)]="selectedUsuario.cadastro"></p-calendar>
    </div>
  </div>

  <div class="ui-g">
    <div class="ui-g-4">ativo</div>
    <div class="ui-g-8">
      <p-checkbox name="ativo"
                  binary="true"
                  [(ngModel)]="selectedUsuario.ativo"></p-checkbox>
    </div>
  </div>

  <div class="ui-g">
    <div class="ui-g-4">grupo</div>
    <div class="ui-g-8">
      <p-dropdown name="grupo"
                  appendTo="body"
                  [options]="grupoOptions"
                  optionLabel="nome"
                  [autoDisplayFirst]="false"
                  [(ngModel)]="selectedUsuario.grupo"></p-dropdown>
    </div>
  </div>

  <p-footer>
    <button pButton type="button" (click)="save(selectedUsuario)" label="Save" class="ui-button-success"></button>
    <button pButton type="button" (click)="showModal = false" label="Cancel" class="ui-button-secondary"></button>
  </p-footer>
</p-dialog>
```

Listagem 8 – Arquivo HTML da classe UsuarioComponent (Formulário)

4.4 IMPLEMENTAÇÃO DO SISTEMA

Para a implementação deste projeto foi necessária a utilização de módulos Maven, para evitar a repetição de código e facilitar a utilização.

O projeto foi separado nos seguintes módulos: *Annotation*, *Core* e *MavenPlugin*, como mostrado na Figura 12.

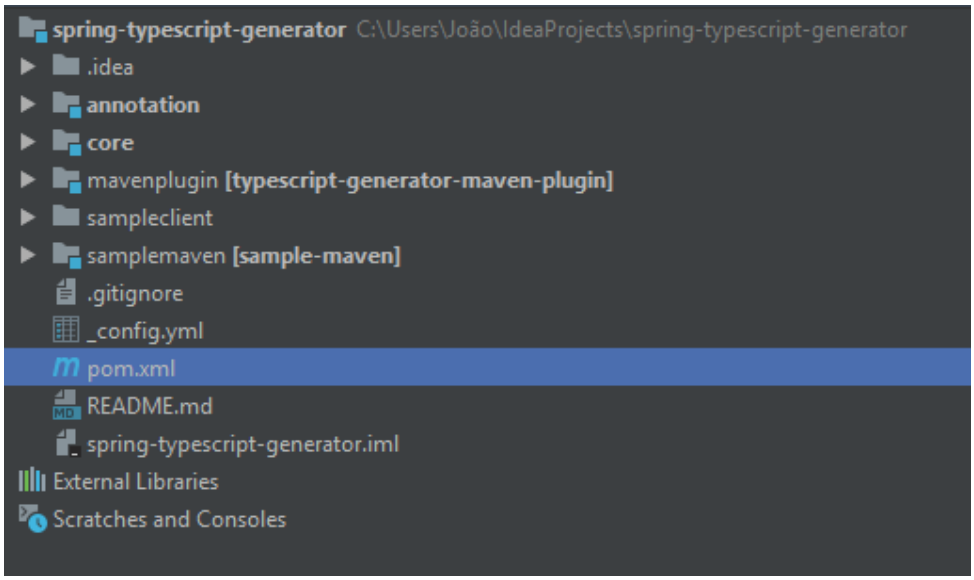

Figura 12 - Estrutura do projeto

4.4.1 Módulo Annotation

O módulo *Annotation* possui todas as anotações necessárias para o mapeamento das entidades, podendo assim ser gerado um artefato jar apenas com estas dependências.

A Figura 13 mostra todas as *annotations* criadas. As principais *annotations* criadas são responsáveis por identificar quais classes Java devem ser processadas. Para cada tipo de arquivo deve ser utilizada a *annotation* específica, por exemplo:

- TsComponent: deve ser utilizada nas classes Java que possuem métodos REST. Esta *annotation* possui ainda uma opção que habilita a geração de um CRUD automático. As *annotations* TsCrudDelete, TsCrudFindAll, TsCrudFindOne e TsCrudSave devem

ser utilizados nos métodos de delete (exclusão), buscar todos, buscar por ID e salvar, respectivamente. A Listagem 9 mostra o código fonte desta *annotation*.

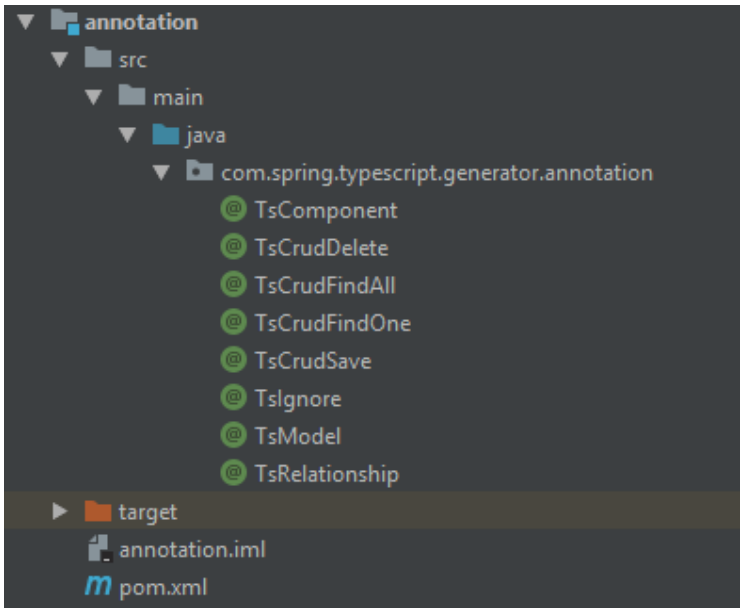

Figura 13 - Módulo Annotations

```java
import java.lang.annotation.*;

/**
 * Created by joao on 17/08/17.
 */

@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE})
public @interface TsComponent {

    Class<?> value();

    boolean crud() default false;
}
```

Listagem 9 - Annotation TsComponent

- TsModel: deve ser utilizado nas entidades Java. As *annotations* TsIgnore e

TsRelationship devem ser utilizadas para ignorar um atributo Java e realizar um relacionamento do tipo muitos para um, respectivamente. Na Listagem 10 é exibido o código fonte desta *annotation*.

```
import java.lang.annotation.*;

/**
 * Created by joao on 17/08/17.
 */

@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE})
public @interface TsModel {

    String value() default "";
}
```

Listagem 10 - Annotation TsModel

4.4.2 Módulo Core

Este é o principal módulo do projeto, pois é nele que estão todas as regras e os *templates* utilizados para o *scaffolding* das classes Java. Neste módulo foi utilizado um padrão semelhante ao MVC, utilizando apenas o *Model* e o *Controller*, pois neste caso não existe a camada de *View*. A Figura 14 representa a estrutura do módulo Core.

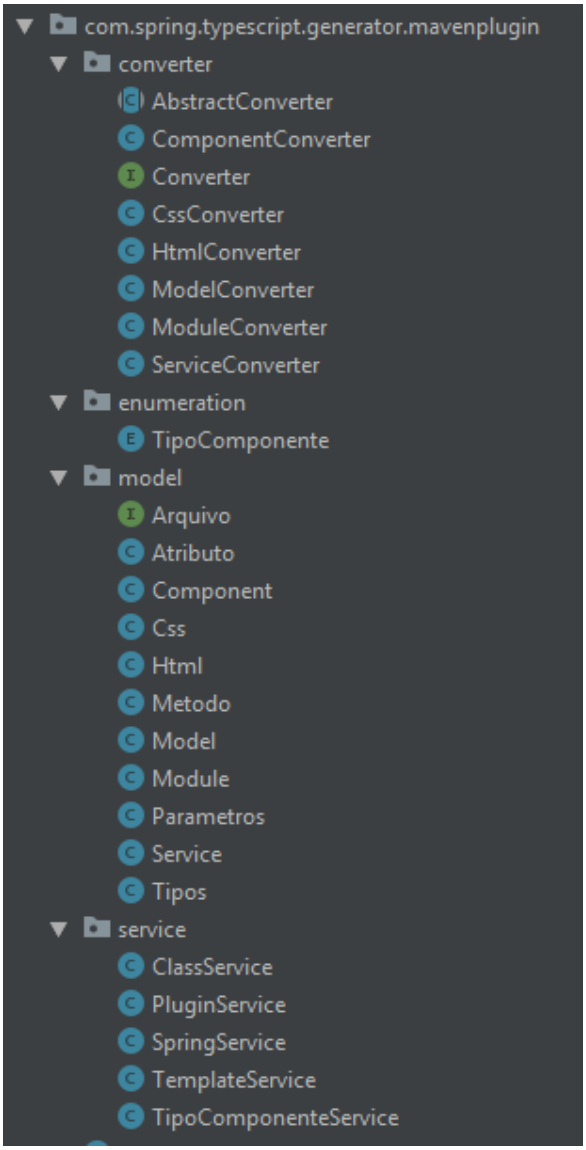

Figura 14 - Módulo Core

Todo processo de *scaffolding* inicia com a chamada do método *build* da classe *PluginService*. Essa classe é responsável por invocar todos os Converters responsáveis por ler, e gerar os arquivos TypeScript, como mostrado na Listagem 11.

```java
public class PluginService {

    private ClassService classService = new ClassService();
    private TemplateService templateService = new TemplateService();
    private Converter<Model> converterModel = new ModelConverter();
    private Converter<Service> converter = new ServiceConverter();
    private Converter<Component> converterComponent = new ComponentConverter();
    private Converter<Module> converterModule = new ModuleConverter();
    private Converter<Html> converterHtml = new HtmlConverter();
    private Converter<Css> converterCss = new CssConverter();

    public void buid(Parametros parametros) {

        List<Model> models = converterModel.converter(classService.getModels());
        models.forEach(model -> templateService
                .build(model, templateName: "model", model.getFolder(), parametros));

        List<Service> services = converter.converter(classService.getComponents());
        services.forEach(service -> templateService
                .build(service, templateName: "service", service.getFolder(), parametros));

        List<Component> components = converterComponent.converter(classService.getComponents());
        components.forEach(component -> templateService
                .build(component, templateName: "component", component.getFolder(), parametros));

        List<Module> modules = converterModule.converter(classService.getComponents());
        modules.forEach(module -> templateService
                .build(module, templateName: "module", module.getFolder(), parametros));

        List<Html> htmls = converterHtml.converter(classService.getComponents());
        htmls.forEach(html -> templateService
                .build(html, templateName: "html", html.getFolder(), parametros));

        List<Css> csss = converterCss.converter(classService.getComponents());
        csss.forEach(css -> templateService
                .build(css, templateName: "css", css.getFolder(), parametros));
    }
}
```

Listagem 11 - Classe PluginService

Como pode-se observar na Listagem 6, para cada tipo de arquivo existe um conversor e um *template* específico. Será utilizada a classe Model para exemplificar o processo de *scaffolding*. Na Listagem 12 é possível entender como é realizada a leitura das classes.

```java
public Model getModel(Class<?> aClass) {
    Model model = new Model();
    model.setNome(aClass.getSimpleName());

    TsModel tsModel = aClass.getAnnotation(TsModel.class);
    if (!tsModel.value().isEmpty()) {
        model.setNome(tsModel.value());
    }

    model.setModifier(Modifier.isAbstract(aClass.getModifiers()) ? "abstract " : "");

    Arrays.asList(aClass.getDeclaredFields()).forEach(field -> {
        field.setAccessible(true);

        if (!field.isAnnotationPresent(TsIgnore.class)) {

            Atributo atributo = new Atributo();
            atributo.setNome(field.getName());
            atributo.setTipo(getType(field, model));
            atributo.setTipoComponente(TipoComponenteService.getTipoComponente(field));

            if (field.isAnnotationPresent(GeneratedValue.class)) {
                atributo.setDisabled(true);
            }

            if (field.isAnnotationPresent(TsRelationship.class)) {
                TsRelationship relationship = field.getAnnotation(TsRelationship.class);
                atributo.setRelationShip(true);
                atributo.setRelationShipLabel(relationship.label());
            }

            model.addAtributo(atributo);
        }
    });

    return model;
}
```

Listagem 12 - Classe ModelConverter

Na Listagem 7, o método *getModel()* recebe uma classe por parâmetro e nele é realizada a leitura por meio de *Reflection* para a criação do objeto Model. Todos os atributos da classe são percorridos para poder identificar o nome e o tipo de variável utilizado, para a partir disso descobrir qual é a classe TypeScript que deverá ser utilizada.

Após esse processo, o arquivo *typescript* é gerado utilizando um Template Engine. A Listagem 13 mostra o *template* e a Listagem 14 mostra a classe responsável por renderizar o *template* e gravar o arquivo em disco.

```
<#list model.imports as imp>
import {${imp.nome}} from '../${imp.folder}/${imp.nomeArquivo}';
</#list>
<#if model.imports?has_content>

</#if>
export ${model.modifier}class ${model.nome} {

  <#list model.atributos as attr>
  ${attr.nome}: ${attr.tipo};
  </#list>

}
```

Listagem 13 - Template Model

```
public class TemplateService {

    public void build(Arquivo arquivo, String templateName, String folder, Parametros parametros) {
        try {
            HashMap<String, Object> params = new HashMap<>();
            params.put(templateName, arquivo);

            Configuration cfg = new Configuration(Configuration.VERSION_2_3_26);
            cfg.setDefaultEncoding("UTF-8");
            cfg.setTemplateExceptionHandler(TemplateExceptionHandler.RETHROW_HANDLER);

            InputStream modelTemplate = getClass()
                    .getResourceAsStream(String.format("/%s.ftl", templateName));

            Template template = new Template( name: null, new InputStreamReader(modelTemplate), cfg);

            String dir = parametros.getDestino() + "/" + folder;
            new File(dir).mkdir();
            File file = new File( pathname: dir + String.format("/%s.%s", arquivo.getNomeArquivo(),
                    arquivo.getExtensao()));

            try (Writer writer = new FileWriter(file)) {
                template.process(params, writer);
            }
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

Listagem 14 - Template Service

4.4.3 Módulo Maven Plugin

O módulo Maven Plugin estende um AbstractMojo que permite que um *plugin* Maven seja executado utilizando um comando, neste caso 'typescript-generator:generate', em que 'typescript-generator' é o nome do plugin e 'generate' é o nome do Mojo Maven.

Esse módulo possui apenas uma classe e nela é invocado as classes do módulo Core. O motivo da separação do módulo Core, é que assim é possível criar *plugins* para outras tecnologias como, por exemplo, Gradle, sem precisar repetir todo o código fonte do módulo Core.

A Listagem 15 mostra como funciona a classe GenerateMojo. O método *execute()* é o padrão executado quando uma classe implementa a interface *Mojo*, ou seja, quando a execução do *plugin* é invocada, esse método é invocado automaticamente. Ele é responsável por iniciar o processo de *scaffolding,* invocando as classes que estão no módulo *core*. Ainda nesta classe é possível definir os parâmetros que poderão ser passados para o *plugin*. O parâmetro *destination é* responsável por armazenar o diretório em que os arquivos gerados serão gravados.

```java
@Mojo(name = "generate",
        defaultPhase = LifecyclePhase.PROCESS_CLASSES,
        threadSafe = true)
public class GenerateMojo extends AbstractMojo {

    @Component
    private MavenProject project;

    @Component
    private PluginDescriptor descriptor;

    @Parameter
    private String destination;

    public void execute() {
        try {
            List<String> runtimeClasspathElements = project.getRuntimeClasspathElements();
            ClassRealm realm = descriptor.getClassRealm();

            for (String element : runtimeClasspathElements) {
                File elementFile = new File(element);
                realm.addURL(elementFile.toURI().toURL());
            }

            new PluginService().buid(new Parametros(destination));
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

Listagem 15 - Classe GenerateMojo

5 CONCLUSÃO

Neste trabalho foi desenvolvido um *plugin* Maven capaz de gerar código fonte para *front-end* utilizando a técnica de *scaffolding*, para as tecnologias Java para o *back-end* e Angular para o *front-end*.

Todos os objetivos do trabalho foram alcançados, apesar de sofrerem alterações durante o desenvolvimento. No decorrer do projeto novas ideias surgiram, então foram adicionadas novas rotinas como, por exemplo, a geração de CRUDS básicos.

O desenvolvimento iniciou com o estudo de plugins Maven e como os utilizar em projetos Java. Também, foi necessário estudar sobre Java Reflections para que o objetivo fosse alcançado.

As principais dificuldades encontradas durante o desenvolvimento do trabalho foram quanto a utilização dos plugins Maven, o ciclo de vida e a maneira de depurar (debugger) essas classes. Ainda, entre as maiores dificuldades estão as diversas maneiras de ler os metadados das classes Java utilizando Reflections.

Como trabalhos futuros poderão ser desenvolvidas novas opções para a geração de CRUDs genéricos, novos componentes e a utilização de mestre detalhe. Também poderá ser desenvolvido esse *plugin* para aplicações Java que utilizam o gerenciador de pacotes Gradle.

REFERÊNCIAS

CAMPIONE, Mary; WALRATH, Kathy. **The Java tutorial**: object-oriented programming for the Internet. SunSoft Press, 1996.

CHODAREV, Sergej; LAKATOŠ, Dominik; PORUBÄN, Jaroslav; KOLLÁR, Jan. **Abstract syntax driven approach for language composition**. Central European Journal of Computer Science, v. 4, n. 3, p. 107–117, 2014.

DURIEUX, Thomas; CORNU, Benoit; SEINTURIER, Lionel; MONPERRUS, Martin. **Dynamic patch generation for null pointer exceptions using metaprogramming**. IEEE, p. 349-358, 2017.

HAVLICE, Zdeněk. **Auto-Reflexive software architecture with layer of knowledge based on UML models**. International Review on Computers & Software, v. 8, n. 8, 2013.

IBM. **The art of metaprogramming, Part 1**: introduction to metaprogramming. Disponível: <https://www.ibm.com/developerworks/library/l-metaprog1/l-metaprog1-pdf.pdf>. Acesso em: 26 out. 2017.

LAKATOŠ, Dominik; PORUBÄN, Jaroslav; BACÍKOVÁ, Michaela. **Declarative specification of references in DSLs**. In: 2013 Federated Conference on Computer Science and Information Systems, ser. FedCSIS 2013, Sept 2013, p. 1527–1534.

LANDMAN, Davy; SEREBRENIK, Alexander; VINJU, Jurgen J. **Challenges for static analysis of java reflection – literature review and empirical study**. 2017 IEEE/ACM 39th International Conference on Software Engineering, 2017, p. 507-518.

MAGNO, Danillo Goulart. **Aplicação da técnica de *scaffolding* para a criação de sistemas CRUD**. Dissertação (mestrado) Programa de Pós-Graduação em Ciência e Tecnologia da Computação, Universidade Federal de Itajubá, 2015.

PIGULA, Peter; NOSÁL, Milan. Unified **Compile-time and runtime Java annotation processing**. In: Federated Conference on Computer Science and Information Systems, v. 5, p. 965–975, 2015.

RICARTE, Ivan Luiz Marques. **Gramáticas livres de contexto**. Disponível em: <http://www.dca.fee.unicamp.br/cursos/EA876/apostila/HTML/node43.html>. Acesso em: 26 out. 2017.

SCHWARTZ, Mathias. **Design and analysis of web application framework**. 2013. Disponível em: <https://pure.au.dk/ws/files/54599671/PH.D_dissertation_Mathias_Schwarz.pdf> Acesso em: 23 de ago. 2017.

SPINELLIS, Diomidis. Rational metaprogramming: tools of the trade. **IEEE Software**, p. 78-

79, 2008.

WUTKA, Mark. **Java:** técnicas profissionais. Berkeley, 1997.

SANTOS FILHO, Walter dos. **Introdução ao Apache Maven**. Belo Horizonte: Eteg. 2008.