INSTITUTO FEDERAL DE CIÊNCIA E TECNOLOGIA DE PERNAMBUCO

Campus Recife

Análise e Desenvolvimento de Sistemas

THOMÁS SOUSA SILVA

OVERJAX:

Um *framework* para facilitar o desenvolvimento de *microservices*

,

Recife

2019

THOMÁS SOUSA SILVA

OVERJAX:

Um *framework* para facilitar o desenvolvimento de *microservices*

Trabalho de conclusão de curso apresentado a coordenação de graduação em Análise e Desenvolvimento de Sistemas do Instituto Federal de Ciência e Tecnologia de Pernambuco, como requisito para obtenção do título de Tecnólogo em Análise e Desenvolvimento de Sistemas.

Orientador: Prof. Dr. Ramide Augusto Sales Dantas.

Recife

2019

Ficha elaborada pela bibliotecária Emmely Cristiny Lopes Silva CRB4/1876

S586o
2019 Silva, Thomás Sousa.
Overjax: um framework para facilitar o desenvolvimento de microservices / Thomás Sousa Silva. --- Recife: O autor, 2019.
65f. il. Color.

TCC (Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas) – Instituto Federal de Pernambuco, Departamento Acadêmico de Controle de Sistemas Eletrônicos - DASE, 2019.

Inclui Referências e glossário

Orientador: Professor Dr. Ramide Augusto Sales Dantas.

1. Orvejax. 2. Java framework. 3. Microservice. I. Dantas, Ramide Augusto Sales (orientador). II. Instituto Federal de Pernambuco. III. Título.

CDD 005 (21ed.)

Trabalho de Conclusão de Curso apresentado pelo aluno **Thomás Sousa Silva** à coordenação de Análise e Desenvolvimento de Sistemas, do Instituto Federal de Pernambuco, sob o título de "***OVERJAX: UM FRAMEWORK PARA FACILITAR O DESENVOLVIMENTO DE MICROSERVICES***", orientado pelo **Prof. Dr. Ramide Augusto Sales Dantas** e aprovado pela banca examinadora formada pelos professores:

Recife, 28 de Março de 2019.

Prof. Dr. Ramide Augusto Sales Dantas
CTADS/DASE/IFPE

Prof. Dr. Paulo Maurício Gonçalves Júnior
CTADS/DASE/IFPE

Prof. Ms. José Paulo da Silva Lima
Faculdade Nova Roma

Thomás Sousa Silva

AGRADECIMENTOS

Primeiramente, um agradecimento mais que especial a toda minha família, por todo o apoio que tive desde sempre.

A toda equipe de Sistemas da Secretaria Estadual de Saúde de Pernambuco, pelos grandes desafios e experiências que tive, isso foi de grande contribuição para o meu desenvolvimento profissional.

Aos amigos, colegas e professores do IFPE-Recife e IFPB-Monteiro, por todas as experiências profissionais e pessoais.

Ao meu antigo orientador José Paulo da Silva Lima, por toda ajuda que tive no meu antigo trabalho.

Ao meu orientador Ramide Augusto Sales Dantas, por todo o apoio na concepção deste trabalho.

A minha mais que amiga “grandona” Ana, por todo carinho e ajuda na construção deste trabalho.

A minha irmãnzinha Thalicia, por toda ajuda na revisão ortográfica e gramatical deste trabalho.

RESUMO

Toda aplicação tem como objetivo resolver um determinado problema, porém para solucioná-lo é preciso fazer muito mais do que criar entidades e regras de negócio. A concepção de uma aplicação exige várias configurações e implementações que não estão ligadas à problemática principal, como tratamento de erros, padronizações e camadas de abstração. O presente trabalho apresenta o desenvolvimento de uma solução que visa reduzir o tempo de desenvolvimento e manutenção de um *microservice* na linguagem Java, por meio da junção de especificações consolidadas no mercado, novas funcionalidades e a definição de padrões de configuração, ao invés de configuração explícita. Como resultado obteve-se um *framework*, que além de facilitar essa configuração inicial, possui vários recursos que podem ser utilizados por toda a aplicação. Entre eles, temos implementações de padronização de formato de resposta de requisições, criação de contexto Java EL global por requisição, interpolador de mensagens internacionalizadas que usa Java EL, manipuladores de exceção, construtores de mensagens internacionalizadas dinâmicas, implementação de filtros de segurança para autenticação e autorização genéricos que suportam qualquer tipo de autenticação, *stateless*, que utilize o *header* HTTP *Authorization*, criação de integração com o gerenciador de identidade de acesso Keycloak, permitindo login único com baixo acoplamento através de *tokens* JWT. Com a intenção de validar o *framework* foi desenvolvido um aplicativo de teste simples denominado *any-address*, que utiliza os recursos do Overjax.

Palavras-chave: Overjax. Java *Framework*. *Microservice*. Redução de trabalho. Redução de tempo de desenvolvimento.

ABSTRACT

Every application has the objective of solving a certain problem or a set of them, but to solve it it is necessary to do much more than to create entities and business rules. The design of an application requires several configurations and implementations that are not linked to the main problem, such as error handling, standardization and abstraction layers. This work presents the development of a project that aims to reduce the time of development and maintenance of a microservice in the Java language, through the combination of specifications consolidated in the market, new features and the definition of configuration patterns, instead of explicit configuration . As a result a framework was obtained, which besides facilitating this initial configuration, has several features that can be used by the entire application. These include standardized implementations of request response format, global Java EL context creation per request, internationalized message interpolator using Java EL, exception handlers, dynamic internationalized message builders, implementation of security filters for authentication and generic authorization that supports any type of authentication, stateless, that use the HTTP Authorization header, creation of integration with the Keycloak access identity manager, allowing single login with low coupling through JWT tokens. With the intention of validating the framework was developed a simple test application named *any-address*, that use the Overjax features.

Keywords: Overjax. Java Framework. Microservice. Work Reducing. Development Time Reducing.

LISTA DE ILUSTRAÇÕES

Figura 1 - Diagrama do modelo de mensagens..... 32
Figura 2 - Exemplo de utilização do *header Authorization* com *token* JWT..... 46
Figura 3 - Tela de administração do Keycloak, com o JSON de configuração..... 50
Figura 4 - Exemplo de *token* decodificado gerado pelo Keycloak.....51
Figura 5 - Endpoints do *microservice any-address*.....54

LISTA DE CÓDIGOS

Código 1 - Exemplo de resposta baseada na especificação JSON API..... 30
Código 2 - Exemplo de *endpoint* que terá sua resposta formatada..... 30
Código 3 - Exemplo de utilização da classe *Logger*..... 31
Código 4 - Exemplo de utilização do MessageBuilder..... 33
Código 5 - Exemplo de arquivo de mensagens..... 33
Código 6 - Fragmento do arquivo pom.xml..... 38
Código 7 - Exemplo de configuração baseada em evento..... 40
Código 8 - Exemplo de configuração utilizando Eclipse microprofile Config..... 40
Código 9 - Exemplo de resposta de *endpoint* com @WrapResponse..... 43
Código 10 - Fragmento da classe ThrowableMapper, mostrando a utilização da *annotation* e o disparo do evento através do método fireAsync..... 44
Código 11 - Fragmento da classe SimpleEntity..... 45
Código 12 - Interface AuthenticationProvider..... 46
Código 13 - Fragmento da classe TokenAuthentication..... 47
Código 14 - Enum que representa os possíveis tipos de resultados de uma autenticação..... 47
Código 15 - Fragmento da classe AuthenticationFilter, ao chamar o método *fire* um evento passando o resultado da autenticação é disparado..... 47
Código 16 - Exemplo de utilização do evento, disparado pela classe AuthenticationFilter..... 47
Código 17 - Exemplo de utilização da annotation @Authorize..... 48
Código 18 - Método da classe SecurityInfo onde é feito o seu registro no ELProcessor..... 48
Código 19 - Exemplo de utilização do evento, disparado pela classe AuthorizationFilter..... 49
Código 20 - Dependências do módulo *overjax-security-keycloak*..... 49
Código 21 - Exemplo de utilização do *overjax-starter*..... 52
Código 22 - Definição do Pom *Parent* no *any-address*..... 55
Código 23 - Definição das dependências do *any-address*..... 55
Código 24 - Classe App do projeto *any-address*..... 55
Código 25 - Fragmento da classe CountryController do projeto *any-address*..... 56
Código 26 - Fragmento da classe IndexController do projeto *any-address*..... 56
Código 27 - Fragmento do arquivo de mensagens do projeto *any-address*..... 57
Código 28 - Definição dos metadados globais do projeto *any-address*..... 57
Código 29 - Fragmento do arquivo web.xml, com a configuração do data source do projeto *any-address*, utilizando variáveis de ambiente..... 58
Código 30 - Fragmento do arquivo de configuração do Keycloak do projeto *any-address*..... 58
Código 31 - Arquivo de configuração do Payara Micro..... 58
Código 32 - Fragmento da classe *State* do projeto *any-address*..... 59
Código 33 - Fragmento da classe CountryService do projeto *any-address*..... 60

LISTA DE TABELAS

Tabela 1 - Quantidade de dependências necessárias em um projeto.....35
Tabela 2 - Quadro comparativo.....36

LISTA DE ABREVIAÇÕES E SIGLAS

API	*Application Programming Interface*
CDI	*Contexts and Dependency Injection*
CEP	Código de Endereçamento Postal
CRUD	Abreviação de *Create Read Update Delete*
HATEOAS	*Hypermedia As The Engine Of Application State*
HTTP	*Hypertext Transfer Protocol*
JAR	Java ARchive
JAX-RS	Java API *for RESTful Web Services*
JDBC	Java *Database Connectivity*
JPA	Java *Persistence* API
JSF	Java *Server Faces*
JSON	JavaScript *Object Notation*
JSON-B	JSON *Binding*
JSON-P	JSON *Processing*
JSP	Java *Server Pages*
JSTL	JSP - *Standard Tag Library*
JTA	Java *Transaction* API
LDAP	*Lightweight Directory Access Protocol*
POJO	*Plain Old Java Object*
REST	*Representational State Transfer*
RESTful	Web services que implementam a arquitetura REST
SAML	*Security Assertion Markup Language*
SES	Secretaria de Saúde do Estado de Pernambuco
SPI	*Service Provider Interface*
URI	*Uniform Resource Identifier*
Web	*World Wide Web*
XPath	XML *Path Language*

SUMÁRIO

1 INTRODUÇÃO 13
1.1 Justificativa 13
1.1.1 Inspiração do Overjax 14
1.2 Objetivo geral 15
1.3 Objetivos específicos 15
1.4 Estrutura do trabalho 15
2 BASES TECNOLÓGICAS 17
2.1 Java 17
2.1.1 Java EE 17
2.1.1.1 JAX-RS 18
2.1.1.2 CDI 18
2.1.1.3 JSON-B 18
2.1.1.4 JSON-P 19
2.1.1.5 Java EL 19
2.2 Maven 20
2.2.1 *Multi-module* 21
2.2.2 Gerenciamento de dependências para projetos *multi-module* 21
2.3 *Microservice* 21
2.4 Eclipse Microprofile 22
2.4.1 Eclipse Microprofile Config 22
2.5 Keycloak 23
2.6 JSON API 24
3 METODOLOGIA 25
3.1 Análise e projeto 25
3.2 Estudos e escolhas de tecnologias 25
3.3 Ambiente de desenvolvimento 26
3.4 Validação do *framework* 27
4 ESPECIFICAÇÃO DO *FRAMEWORK* 28
4.1 Recursos do *framework* 28
4.1.1 Configuração externalizada 28
4.1.2 Gerenciamento de dependências 28
4.1.3 Configuração global do formato de resposta JSON 28
4.1.4 Contexto EL global por requisição 28
4.1.5 Tratamento de exceções 29
4.1.6 Tratamento de exceções delegado 29
4.1.7 *Wrapper* de respostas HTTP 29

4.1.8 Configuração de *log* **30**
4.1.9 Internacionalização **31**
4.1.10 Leitura múltipla do corpo da requisição **31**
4.1.11 Interpolador de mensagens dinâmicas **31**
4.1.12 Modelo de mensagens **32**
4.1.13 *Builder* de mensagens **33**
4.1.14 *Builder* de metadados **33**
4.1.15 Autenticação e autorização **34**
4.1.16 *Starters* **35**
4.2 Comparativo 35
5 DETALHAMENTO DO *FRAMEWORK* 37
5.1 Arquitetura 37
5.2 Módulos 37
5.2.1 Overjax-dependencies **37**
5.2.2 Overjax-annotation **38**
5.2.2.1 @Controller *38*
5.2.2.2 @ExceptionHandling *39*
5.2.2.3 @MappedException *39*
5.2.2.4 @Property *39*
5.2.2.5 @RequestBody *39*
5.2.2.6 @WrapResponse *39*
5.2.3 Overjax-config **40**
5.2.4 Overjax-core **40**
5.2.4.1 Providers *41*
5.2.4.1.1 El *processor provider* 41
5.2.4.1.2 Jsonb *provider* 41
5.2.4.1.3 *Locale provider* 41
5.2.4.1.4 *Logger provider* 41
5.2.4.1.5 *Request body provider* 42
5.2.4.1.6 Formatação de respostas, criação de mensagens e metadados 42
5.2.4.2 Tratamento de exceções *44*
5.2.5 Overjax-data **45**
5.2.6 Overjax-security **45**
5.2.6.1 Filtro de autenticação *45*
5.2.6.2 Filtro de authorização *48*
5.2.7 Overjax-security-keycloak **49**
5.2.8 Overjax-starters **52**
5.2.8.1 Overjax-starter *52*

5.2.8.2 Overjax-starter-data-jpa *52*
5.2.8.3 Overjax-starter-data-jpa-postgres *53*
6 APLICAÇÃO DE TESTE 54
6.1 Configuração 54
6.1.1 Arquivo pom.xml **54**
6.1.2 *Controllers* **55**
6.1.3 Envio de mensagens customizadas **56**
6.1.4 Metadados globais **57**
6.2 Configurações extras 57
6.3 Configurações de *properties* 59
6.4 Domínio do *microservice* 59
6.5 Benefícios 60
7 CONSIDERAÇÕES FINAIS 62
REFERÊNCIAS 63

1 INTRODUÇÃO

Softwares são produtos criados para resolver um determinado problema, porém, em um ambiente empresarial, para chegar nesse produto final é preciso que seja feito uma configuração inicial juntamente com alguma implementação base, como tratamento de erros, padronizações e camadas de abstração. Em outras palavras, é necessário fazer uma arquitetura inicial. Geralmente, esse trabalho inicial não está ligado ao domínio da aplicação.

Atualmente, existem diversas especificações e implementações de APIs (*Application Programming Interface*), que visam resolver um problema específico relacionado à essa configuração inicial. JPA (Java *Persistence* API) é um exemplo de especificação que pretende facilitar o acesso ao banco de dados. Outro exemplo é o projeto Eclipse Microprofile Config, utilizado para acessar propriedades de configuração.

Essas especificações são renomadas por sua utilidade, mas assim como outras, elas são isoladas, o que exige uma implementação e/ou configuração inicial para integrá-las. Um exemplo seria integra-las para poder configurar o JPA utilizando Eclipse Microprofile Config.

1.1 Justificativa

Implementar uma solução para o exemplo acima demanda esforço e tempo para uma funcionalidade que pode não estar ligada ao domínio da aplicação. Outro problema é fazer a manutenção deste código quando essa configuração inicial é feita de maneira acoplada, para atender as necessidades de uma aplicação específica.

Em um cenário onde só existe uma única aplicação, a manutenção se torna relativamente simples. Porém, o mesmo não ocorre em outro mais realista, onde existem várias aplicações de domínios distintos, mas com uma configuração base semelhante duplicada.

Quando é necessário corrigir um erro ou fazer alguma melhoria na configuração base dessas aplicações, será preciso repetir esse procedimento para todos os outros sistemas, devido ao fato deles não compartilharem uma configuração

base centralizada. O esforço dessa manutenção será multiplicado pelo número de aplicações existentes.

1.1.1 Inspiração do Overjax

Esse trabalho tem como fonte de inspiração as problemáticas enfrentadas ao longo do desenvolvimento do sistema FormaSUS, realizado na Secretaria de Saúde do Estado de Pernambuco (SES). Dentre os problemas enfrentados temos:

- Duplicação de funcionalidades, falta de centralização de código;
- Pouca ou ausente padronização em respostas HTTP (*Hypertext Transfer Protocol*);
- Validação de objetos com regras dinâmicas;
- Interpolação de mensagens dinâmicas;
- Falta de generalização de funcionalidades de CRUD (Abreviação de *Create Read Update Delete*);
- Sanitização de parametros;
- Tratamento de exceções global integrado com resposta padrão;
- Gerenciamento de dependências;
- Ausência de configuração externalizada;
- Falta autenticação descentralizada e stateless.

Nesse período de criação, foram desenvolvidas várias soluções para esses problemas, em que boa parte, não estavam ligados diretamente ao domínio do sistema.

Com o passar do tempo, percebeu-se que essas implementações que não estavam relacionadas diretamente ao domínio do FormaSUS, deveriam ser centralizadas em um projeto a parte, para que pudessem ser utilizadas por outros projetos da Secretaria de Saúde. Devido ao fato de essas funcionalidades possuírem escopo genérico aplicável a vários projetos, surgiu a ideia de criar o projeto público *open source* Overjax (Disponível em ***overjax.seudev.com***), em que a comunidade pudesse contribuir para o desenvolvimento do mesmo.

Como forma de diminuir esse re-trabalho, aumentar a manutenabilidade e reduzir o tempo de desenvolvimento, esse trabalho se propôs a desenvolver uma solução, que resultou no *framework* Overjax, voltada para microservices, que é um

estilo arquitetural que possibilita ampliar a utilidade dessa solução, visto que um microservice pode ser consumido por várias aplicações.

1.2 Objetivo geral

O objetivo principal desse trabalho é desenvolver um *framework* para centralizar as principais configurações iniciais de um microservice, e assim reduzir tempo e custo de desenvolvimento e manutenção.

1.3 Objetivos específicos

Os objetivos específicos desse trabalho são:

- Identificar tecnologias existentes no mercado e suas limitações;
- Selecionar tecnologias que serão utilizadas no desenvolvimento;
- Levantar funcionalidades a serem cobertas pelo *framework;*
- Projetar a arquitetura e a comunicação entre componentes;
- Implementar os componentes de *software* do *framework;*
- Criar aplicação de teste que utilize os principais os recursos desenvolvidos.

1.4 Estrutura do trabalho

O trabalho desenvolvido foi estruturado em sete capítulos, conforme descrito abaixo:

- No primeiro capítulo tem-se a introdução ao tema principal, com a descrição do problema, justificativa e objetivos do presente trabalho;
- No segundo capítulo são apresentadas as bases tecnológicas utilizadas no desenvolvimento do *framework*;
- No terceiro capítulo é apresentado a metodologia utilizada durante o trabalho, descrevendo o processo de desenvolvimento do projeto;
- No quarto capítulo é descrito os recursos do *framework* desenvolvido e apresentado um comparativo entre a solução criada e algumas existentes no mercado;
- No quinto capítulo é apresentado um detalhamento da arquitetura e módulos do *software* desenvolvido;

- No sexto capítulo é apresentado como foi feita a integração de uma aplicação de teste com o Overjax;
- No sétimo capítulo é apresentada a conclusão e possíveis trabalhos futuros.

2 BASES TECNOLÓGICAS

O presente trabalho utiliza como base seis tecnologias para o desenvolvimento do *framework*:

- Java, a linguagem utilizada na programação do *framework*;
- Maven, ferramenta de automação de compilação;
- *Microservice*, estilo arquitetural;
- Eclipse Microprofile, plataforma base de definição que otimiza aplicações Java para arquitetura de *microservices*;
- Keycloak, uma aplicação que permite o login único com o gerenciamento de identidades e o gerenciamento de acesso, destinados a aplicativos e serviços modernos;
- JSON API, uma especificação que define um formato padrão de requisições e respostas HTTP baseadas em JSON (JavaScript *Object Notation)*.

2.1 Java

A linguagem Java teve origem na década de 90, na Sun Microsystems, em um projeto chamado *The Green Project*, que foi iniciado por Patrick Naughton, Mike Sheridan e James Gosling (MENDES, 2009). Hoje em dia é uma da mais utilizadas para criação de aplicação web (CHAN, 2019).

Segundo Mendes (2009), Java é uma linguagem simples orientada a objetos, *multithread*, interpretada, neutra de arquitetura, portável, robusta segura e que oferece alto desempenho.

2.1.1 Java EE

A Java EE (Java *Platform, Enterprise Edition*) é uma plataforma padrão para desenvolver aplicações de grande porte, que inclui diversas especificações (tecnologias) e visa resolver problemas recorrentes do desenvolvimento de aplicações empresariais (FARIA, 2013).

Entre as especificações da Java EE, as utilizadas no desenvolvimento do Overjax são listadas nas subseções a seguir:

2.1.1.1 JAX-RS

JAX-RS (Java API *for RESTful Web Services*) é uma especificação que define um conjunto de APIs para o desenvolvimento de *Web services* de acordo com o estilo arquitetural REST.

Essa especificação tem como objetivo fornecer um conjunto de *annotations* que podem ser usadas com POJOs (*Plain Old Java Object*) para expor recursos *web*. Ela fornece um claro mapeamento entre elementos URI (*Uniform Resource Identifier*) e métodos anotados com annotations correspondentes da API. Sua proposta é fornecer um suporte de alto nível ao protocolo HTTP, sendo suficientemente flexível para suportar variações do protocolo.

JAX-RS é independente de formato de resposta, sendo capaz de suportar qualquer tipo de conteúdo no corpo das respostas HTTP. Além disso, ela também possui independência de container, por ser uma especificação e não uma implementação as aplicações que utilizam JAX-RS podem ser implantadas em qualquer *container web* que tenha suporte a essa especificação (ORACLE, 2017).

2.1.1.2 CDI

CDI (*Contexts and Dependency Injection*) define um poderoso conjunto de serviços complementares que ajudam a melhorar a estrutura do código da aplicação. CDI é a especificação padrão do Java para um padrão de desenvolvimento conhecido com Injeção de dependência, que visa manter o baixo acoplamento entre os componentes da aplicação (ORACLE, 2017).

Essa especificação possui um sofisticado mecanismo de injeção de dependência, que possibilita injetar dependências de maneira segura sem a necessidade de uma configuração complexa. Além disso, a especificação tem um modelo de notificação de eventos e suporte a interceptores para os objetos gerenciados (ORACLE, 2017).

2.1.1.3 JSON-B

Segundo a Oracle (2017), JSON-B (JSON *Binding*) define uma especificação para fazer a ligação entre objetos Java e documentos JSON, seus principais objetivos são:

- Serialização e deserialização para todos documentos JSON compatíveis com o padrão RFC 7159, que define um conjunto de regras de formatação para uma representação portável do dado estruturado;
- Define convenção de mapeamento padrão para classes Java, também permitindo customização desse mapeamento padrão;
- Ser fácil de usar, o uso padrão da API não exige conhecimento prévio do formato de documentos JSON;
- Permitir mapeamento parcial de subconjuntos de documentos JSON para objetos Java e vice e versa;
- Definir ou habilitar integração com implementações da especificação JSON Processing.

2.1.1.4 JSON-P

Segundo a Oracle (2018), JSON-P (JSON *Processing*) é uma API para converter, gerar, transformar e pesquisar mensagens JSON. Ela produz e consome textos no formato JSON em forma de *streaming* permitindo construir modelos de objetos Java. Seus principais objetivos são:

- Definir convenção de mapeamento padrão para classes Java, também permitindo customização desse mapeamento padrão;
- Ser fácil de usar. O uso padrão da API não exige conhecimento prévio do formato de documentos JSON.

2.1.1.5 Java EL

Java EL (Java *Expression Language*) é uma linguagem de expressão que foi originalmente inspirada em ECMAScript e na linguagem XPath. Java EL foi introduzido juntamente com a versão 1.0 da JSTL para tornar fácil a manipulação de dados nas páginas *web* da aplicação, sem ter uma alta complexidade associada com linguagens de programação como Java e Javascript (ORACLE, 2013).

Devido ao seu sucesso, essa API foi movida para especificação JSP 2.0, tornando-se disponível para as páginas JSP, e não apenas para atributos de *tags* JSTL, como ocorria na versão inicial. Java *Server Faces* 1.0 definiu um *framework* para construção de componentes de interface de usuário, que foi construído sobre a

especificação JSP 1.2. Por essa versão não ter uma linguagem de expressão e a versão 2.0 do JSP não atender todas as necessidades que o JSF necessitava, foi criado uma variação do Java EL específica para o JSF.

Após a criação dessa variação, ficou óbvio que era desejável ter uma única linguagem de expressão unificada. Então, a equipe que desenvolve o JSP e o Java *Server Faces* se juntaram e desenvolveram a especificação de uma linguagem de expressão unificada, que pudesse ser desacoplada e aplicável em vários contextos, possuindo uma sintaxe simples e restrita a avaliação de expressões (ORACLE, 2013).

2.2 Maven

Maven foi originalmente iniciado como um tentativa de simplificar o processo de *build* do projeto da turbina Jakarta. O objetivo da equipe desenvolvedora era padronizar o processo de *build* dos projetos, ter uma clara definição do que o projeto é constituído, um caminho fácil para publicar informações do projeto e um caminho para compartilhar JARs através de diversos projetos (APACHE SOFTWARE FOUNDATION, 2018).

Isso resultou em uma ferramenta que agora pode ser usada para construção e gerenciamento de qualquer projeto baseado em Java, tornando o dia-a-dia dos desenvolvedores dessa linguagem de programação mais fácil.

Segundo a Apache Software Foundation (2018), o principal objetivo do Maven é permitir que um desenvolvedor compreenda o estado completo de um esforço de desenvolvimento no menor período de tempo. Para conseguir isso, existem várias áreas de preocupação com as quais Maven tenta lidar, entre elas estão:

- Fazer o processo de *build* fácil;
- Prover um sistema de *build* uniforme;
- Prover informações de qualidade do projeto.

Além das tradicionais funcionalidades dessa ferramenta, foram utilizadas no desenvolvimento do *framework* Overjax:

2.2.1 *Multi-module*

Maven suporta a criação de um projeto composto por outros projetos. O mecanismo que manipula projetos multi-módulos é conhecido como *The Reactor*. Essa parte do núcleo do Maven é responsável por coletar todos os módulos disponíveis para construção de maneira ordenada e garantir uma ordem de *build* correta, além de realizar o processo de *build* dos projetos em ordem (APACHE SOFTWARE FOUNDATION, 2018).

Aplicações multi-módulo permitem que o seu código seja dividido em módulos, facilitando o entendimento e a criação de pacotes menores que podem ser utilizados isoladamente ou interligados com algum outro módulo, o compartilhamento de código, e evita duplicações.

2.2.2 Gerenciamento de dependências para projetos *multi-module*

Gerenciamento de dependências é uma das principais funcionalidades do Maven. Essa funcionalidade em projetos *multi-module* torna possível o compartilhamento de dependências entre os sub módulos do projeto, e elimina parcialmente ou completamente a necessidade de configuração de dependências (APACHE SOFTWARE FOUNDATION, 2018).

Seu uso se faz por meio da utilização das *tags dependencies* e *dependency* no arquivo pom.xml. Quando definidas no arquivo pom.xml de um módulo, elimina-se a necessidade de redefinir as *tags* nos sub módulos. Já quando definidas por meio da utilização das *tags dependencyManagement* e *dependency*, não elimina essa necessidade, mas elimina a de definir a versão da dependência nos sub módulos (APACHE SOFTWARE FOUNDATION, 2018).

2.3 *Microservice*

Microservice — também conhecidos como a arquitetura de *microservice* — é um estilo de arquitetura que estrutura uma aplicação como uma coleção de serviços fracamente acoplados, que implementam recursos de negócios. Isso permite *continuous delivery/deployment* de grandes e complexas aplicações e que uma organização evolua sua pilha de tecnologia (RICHARDSON, 2018).

2.4 Eclipse Microprofile

Eclipse Microprofile é uma definição de plataforma base que otimiza projetos Java para uma arquitetura de *microservices*. Essa iniciativa foi criada pela Eclipse Foundation, que tem como objetivo criar especificações para solucionar problemas enfrentados no desenvolvimento e manutenção de *microservices*. Essa plataforma open source é desenvolvida com o apoio da comunidade, juntamente com as empresas que fornecem implementações dessas especificações, o que resulta em uma padronização estabelecida pelo mercado (ECLIPSE, 2018).

Eclipse Microprofile é dividido em vários projetos dentre eles podemos citar:

- Eclipse Microprofile Fault Tolerance
- Eclipse Microprofile Health
- Eclipse Microprofile Jwt Auth
- Eclipse Microprofile Metrics
- Eclipse Microprofile Open Api
- Eclipse Microprofile Opentracing
- Eclipse Microprofile Config

2.4.1 Eclipse Microprofile Config

Essa especificação define um sistema fácil e flexível para configuração de uma aplicação, sendo possível estender esse mecanismo de configuração utilizando SPI (*Service Provider Interface*) de forma portável (ECLIPSE, 2018).

A maioria das aplicações necessitam ser configuradas com base no ambiente que elas estão executando. O Eclipse Microprofile Config é uma API que permite que a configuração da aplicação possa ser alterada sem que seja necessário re-empacotar a aplicação, além de suportar diferentes fontes e formatos de configuração — por exemplo propriedades do sistema, variáveis de ambiente, arquivos de propriedade, arquivos xml e outros datasources (ECLIPSE, 2018).

Assim como todos os demais projetos da plataforma Eclipse Microprofile, essa API foi desenvolvida para ser utilizada em um ambiente de *cloud*. Por isso, ela possui suporte para alteração dinâmica de configuração sem a necessidade de reiniciar a aplicação (ECLIPSE, 2018).

Essa API faz a injeção da configuração por meio da especificação CDI. Com isso é possível criar uma sistema desacoplado de um mecanismo de configuração. Sua forma mais simples de ser utilizada ocorre por meio da *annotation* @ConfigProperty em conjunto com a annotation do CDI @Inject (ECLIPSE, 2018).

2.5 Keycloak

Keycloak é uma solução *open source* de gerenciamento de identidade e acesso. Seu principal objetivo é garantir a segurança de aplicações e serviços com poucas linhas de código. Ela é uma aplicação que fornece autenticação utilizando o mecanismo conhecido como *Single-Sign On* (REDHAT, 2018).

Em um ambiente distribuído, constituído de vários *microservices*, o *Single-Sign On* é ideal, pois permite que usuários se autentiquem uma única vez, sem a necessidade de repetir o processo em cada aplicação. Isso significa que sua aplicação não necessita criar um sistema de *login* e cadastro de usuários, isso já é feito pelo Keycloak. Isso também se aplica ao *logout*, o Keycloak fornece *single-sign out*, o que significa que ao fazer *logout* nessa aplicação, o *logout* servirá em todas as aplicações que utilizam o Keycloak de maneira automática (REDHAT, 2018).

Com essa solução, é possível se autenticar por meio dos protocolos OpenID Connect, OAuth 2.0, SAML 2.0, LDAP e outros, sendo possível utilizar provedores como Google, Facebook, Twitter, GitHub e StackOverflow (REDHAT, 2018).

Essa aplicação foi construída utilizando a linguagem Java, mas seu uso não está restrito a essa linguagem. Além de sua Rest API, ele possui diversos *adapters* para diferentes linguagens e plataformas que permitem sua fácil utilização (REDHAT, 2018).

Além de ser um provedor de autenticação, ele também é um provedor de autorização. Isso permite que você possa gerenciar as permissões de acesso de todas as suas aplicações por meio do console de administração, tendo a possibilidade de criar as restrições necessárias para o bom funcionamento das suas aplicações (REDHAT, 2018).

2.6 JSON API

JSON API é uma especificação que define um formato padrão de requisições e respostas HTTP baseadas em JSON. Ela especifica como um cliente deve solicitar ou modificar um recurso e como o servidor deve responder a essas solicitações (KLABNIK *et al*., 2018).

JSON API é projetado para minimizar tanto o número de requisições quanto a quantidade de dados que são transmitidos entre cliente e servidor. Essa eficiência é alcançada sem comprometer a legibilidade ou flexibilidade (KLABNIK *et al*., 2018).

Essa especificação define um conjunto de convenções. Entre elas podemos citar:

- *Client Responsibilities*
- *Server Responsibilities*
- *Document Structure*
- *Fetching Data*
- *Creating, Updating and Deleting Resources*
- *Query Parameters*
- *Errors*

3 METODOLOGIA

3.1 Análise e projeto

Conforme mencionando anteriormente foi na Secretaria de Saúde do Estado de Pernambuco que a inspiração do Overjax surgiu. A SES mantém e desenvolve diversos sistemas, um deles é o FormaSUS. Realizando manutenção nesses sistemas foi identificado que os problemas não estavam restritos ao FormaSUS, confirmando a ideia de que uma solução centralizada poderia ser aplicável à uma gama de aplicações.

Após essa constatação foi decidido que seria desenvolvido um *framework* que pudesse atender essas necessidades, visto que os *frameworks* existentes no mercado não atendiam por completo essas necessidades. Analisando essas problemáticas juntamente com as soluções desenvolvidas especificamente para o projeto FormaSUS, foi identificado alguns detalhes de implementação considerados bons e ruins, como alto e baixo acoplamento, reuso e facilidade de utilização.

Com base nessas análises foram projetadas novas formas de implementação, até que elas chegassem em um ponto de maturação aceitável, para uma primeira versão. Mais adiante esse *framework* com essas novas soluções recebeu o nome de Overjax.

3.2 Estudos e escolhas de tecnologias

Atualmente, o desenvolvimento de uma aplicação é dividido basicamente em *frontend* e *backend*. O *frontend* é responsável por fazer a interface com o usuário. Já o *backend* tem como função processar as requisições, feitas por meio do *frontend*.

Com essa divisão é possível ter várias aplicações *frontend* que utilizam o mesmo *backend*. Um exemplo disso seria uma aplicação *web* e um aplicativo Android que consomem o mesmo *backend*. O *backend* centraliza todas as regras de negocio da aplicação e disponibilizam *endpoints* para que o *frontend* possa consumir. Geralmente esses *endpoints* são estruturados no padrão REST. Visando tornar o uso do Overjax mais amplo foi decidido que esse projeto seria voltado para *microservices*.

Para desenvolver esse trabalho, foram realizados estudos nas especificações da linguagem Java, com o intuito de encontrar recursos e

funcionalidades que ajudassem a mitigar o problema. A partir do estudo, foi encontrada a especificação JAX-RS, que define APIs para desenvolver *Web services* no estilo REST. Essa especificação é integrada com a especificação CDI, que é integrada com diversas outras especificações do Java. Sabendo disso e dos vários recursos definidos por elas foi decidido que elas fariam parte da base do projeto Overjax.

Também nesses estudos, foi encontrado o projeto Eclipse Microprofile, que é uma plataforma base capaz de otimizar projetos Java para uma arquitetura de *microservices*. Esse projeto é integrado com as duas especificações mencionadas acima, com isso, ele também foi adotado como base do projeto Overjax.

Para que um *microservice* possa ser escalável, ele precisa ser *stateless*. Atualmente, o mercado adota o uso de *tokens* para fazer a segurança da aplicação de forma *stateless*. Com base nisso, foi realizado um estudo para encontrar uma maneira de fazer essa segurança com baixo acoplamento. Neste estudo, foi encontrado o gerenciador de identidade Keycloak. Por essa ferramenta ser *open source* e possuir vários recursos como *single-sign on* e integração com outros provedores de identidade como Google e Facebook, foi decidido que seria feito uma integração com essa ferramenta.

3.3 Ambiente de desenvolvimento

Abaixo estão as definições das principais ferramentas utilizadas durante o processo de desenvolvimento e testes do Overjax:

- Eclipse IDE: É um ambiente de desenvolvimento integrado (IDE), gratuito e *open source*, utilizado para desenvolver o *framework;*
  https://www.eclipse.org/ide
- Git: É um sistema de controle de versão distribuído, gratuito e *open source*, projetado para lidar com tudo, de projetos pequenos a muito grandes, com velocidade e eficiência;
  https://git-scm.com
- GitHub: É uma plataforma, gratuita, de hospedagem de código-fonte com controle de versão usando o Git;
  https://github.com

- Docker: É uma ferramenta, gratuita e *open source*, projetada para facilitar a criação, a implementação e a execução de aplicativos usando contêineres. Os contêineres permitem que um desenvolvedor empacote uma aplicação com todas as partes de que precisa, como bibliotecas e outras dependências, e envie tudo como um único pacote;
  https://www.docker.com
- PostgreSQL: É um sistema de gerenciamento de banco de dados (SGBD), gratuito e *open source*, que utiliza a linguagem SQL (Linguagem de Consulta Estruturada, do inglês *Structured Query Language*) como interface;
  https://www.postgresql.org
- pgAdmin: É uma ferramenta, gratuita e *open source*, para administração e desenvolvimento de bancos de dados PostgreSQL;
  https://www.pgadmin.org
- Keycloak: É uma solução *open source* de gerenciamento de identidade e acesso.
  https://www.keycloak.org
- Payara Micro: É uma plataforma, gratuita e *open source*, de *middleware* para implementações de microsserviços Jakarta EE em contêineres;
  https://www.payara.fish/software/payara-server/payara-micro
- Postman: É uma API *Client*, gratuito, que facilita aos desenvolvedores criar, compartilhar, testar e documentar APIs. Isso é feito, permitindo aos usuários criar e salvar requisições HTTP e HTTPs simples e complexas, bem como ler suas respostas.
  https://www.getpostman.com

3.4 Validação do *framework*

No intuito de validar se o *framework* desenvolvido estava se comportando como esperado definiu-se que seria criado um *microservice* de teste que iria utilizar o Overjax da mesma forma que um outro *microserivce* qualquer pudesse a vir utiliza-lo, atuando como um teste de integração.

4 ESPECIFICAÇÃO DO *FRAMEWORK*

Abaixo estão descritos os principais recursos do Overjax, em seguida temos um comparativo entre a solução criada e algumas existentes no mercado:

4.1 Recursos do *framework*

4.1.1 Configuração externalizada

Todo tipo de configuração definida no Overjax é feita por meio do Eclipse Microprofile Config, a fim de garantir que as configurações possam ser definidas com o uso de várias fontes de configuração, como arquivo de *properties*, variável de ambiente, propriedades do sistema e etc.

4.1.2 Gerenciamento de dependências

Todas as dependências utilizadas por esse projeto estão centralizadas em um único ponto, isso permite usar tais dependências sem a necessidade de definir o número da versão. Dessa forma, a atualização de versão é garantida facilmente e toda aplicação pode utilizar o mesmo número de versão.

4.1.3 Configuração global do formato de resposta JSON

Requisições que produzem respostas no formato JSON, utilizam uma instância da classe Jsonb para fazer a conversão de objetos Java para objetos JSON. No intuito de tornar configurável esse formato de resposta de maneira global, esse *framework* produz uma instância dessa classe, por meio de CDI.

Essa instância é inicialmente configurada utilizando Eclipse Microprofile Config, em seguida, um evento é disparado permitindo que ela possa ser configurada por quem estiver observando esse evento.

4.1.4 Contexto EL global por requisição

Overjax produz uma instância da classe ELProcessor para cada requisição, o que permite ter um contexto EL global por requisição. Ao produzir essa instância, é disparado um evento de configuração. Desse modo, é possível realizar alguma

configuração adicional de maneira desacoplada. Como por exemplo, adicionar objetos ao contexto EL.

4.1.5 Tratamento de exceções

O *framework* é capaz de tratar exceções dos seguintes tipos:

- ConstraintViolationException
- Exception
- Throwable
- WebApplicationException

Além de tratar as exceções, esses mappers disparam um evento assíncrono para cada exceção tratada, o que permite executar algum processamento dessas exceções de maneira desacoplada, utilizando CDI.

4.1.6 Tratamento de exceções delegado

O *framework* é capaz de delegar a causa de uma exceção para um *mapper* capaz de manipular essa causa. Os tipos de exceções delegados são:

- ConstraintViolationException
- WebApplicationException

Delegar a causa de uma exceção permite que a mensagem de erro, enviada como resposta, seja mais apropriada. Por exemplo, ao tentar persistir uma entidade no banco de dados utilizando JPA com controle de transação via annotation, essa entidade será validada, e caso possua algum dado inválido, uma *ConstraintViolationException* será lançada. Porém, como essa tentativa de persistência foi realizada dentro de uma transação, a exceção será encapsulada em uma exceção de *rollback*.

Sem essa delegação, o usuário teria como resposta uma mensagem de erro genérica, como erro interno no servidor (500). Já com essa funcionalidade, o usuário recebe as mensagens de erro como resposta.

4.1.7 *Wrapper* de respostas HTTP

Quando um usuário faz requisições, ele obtém respostas que variam conforme a requisição. Geralmente uma resposta contém dados, mensagens de erro

e alguma outra informação que venha a complementar os dados principais. Pensando nisso, a comunidade criou a especificação JSON API, que é um formato padrão de respostas HTTP baseadas em JSON.

Essa especificação define que o *body* das respostas deve ser formado por um objeto JSON com apenas três propriedades: data, errors e meta (Código 1).

Código 1 - Exemplo de resposta baseada na especificação JSON API.

```
{
    "data": {
        "foo": "bar"
    },
    "errors": [
        {
            "id": "4910b154-1c4b-43f4-9132-a19b541d5792",
            "code": "54943085",
            "title": "Error title...",
            "detail": "Error detail..."
        }
    ],
    "meta": {
        "meta1": "value1",
        "meta2": "value2"
    }
}
```

Fonte: Elaborado pelo autor, 2018.

Overjax é capaz de produzir respostas neste formato (Código 1) para todo *endpoint* que estiver anotado com a annotation @WrapResponse (Código 2). Os endpoints anotados podem retornar objetos ou lista de objetos que representam os dados (Código 2), e o *framework* ficará responsável por formatar essa resposta. As propriedades *errors* e meta serão criadas automaticamente, caso necessário.

Código 2 - Exemplo de endpoint que terá sua resposta formatada.

```
@GET
@WrapResponse
public Map<String, String> example() {
    return singletonMap("foo", "bar");
}
```

Fonte: Elaborado pelo autor, 2018.

4.1.8 Configuração de *log*

Quando criamos uma aplicação, é importante criar mensagens de *log* para facilitar o rastreamento de fluxos de execução ou erros, por exemplo. O Java possui uma classe chamada *Logger*, que é utilizada com esse propósito. No intuito de facilitar a configuração de uma instância da classe *Logger* e manter essa configuração centralizada, esse *framework* produz instâncias da classe *Logger* já

configuradas. Com isso, será possível usar uma instância dessa classe com o uso da *annotation* @Inject (Código 3).

Código 3 - Exemplo de utilização da classe *Logger*.

```
@Inject
private Logger logger;
```

Fonte: Elaborado pelo autor, 2018.

4.1.9 Internacionalização

Quando uma requisição é feita, é interessante que certas informações retornadas na resposta dessa requisição esteja em idioma apropriado. O protocolo HTTP define o *header Accept-Language*. Sua função é informar ao servidor sobre a linguagem que o requisitante espera que o servidor envie de volta.

Overjax produz uma lista de instâncias da classe *Locale*, ordenada por prioridade, utilizando CDI. Produzindo essa lista, outros pontos da aplicação podem utilizar esses locales, de maneira desacoplada, para formatar as mensagens que serão enviadas como resposta. Essa lista deve suportar locales padrões definidos com o uso de Eclipse Microprofile Config e locales definidos no *header Accept-Language*.

4.1.10 Leitura múltipla do corpo da requisição

Ao fazer uma requisição, seu corpo é lido em forma de *streaming*. Por isso, ela só pode ser feita uma única vez, o que impede que os bytes desse *streaming* possam ser lidos repetidamente para algum tipo de processamento extra. No intuito de tornar possível essa leitura múltipla, o Overjax provê um *array* de *bytes* que represente o corpo da requisição, utilizando CDI.

4.1.11 Interpolador de mensagens dinâmicas

O Java define uma maneira de fazer internacionalização de mensagens utilizando arquivos de propriedades com sufixos, que indicam qual o idioma das mensagens do arquivo. Essa internacionalização é feita por meio da classe *ResourceBundle*.

Uma deficiência dessa classe é internacionalizar mensagens dinâmicas, em que parte dela é definida através de um parâmetro ou uma expressão, por

exemplo. Para resolver essa problemática, o Overjax provê um interpolador de mensagens dinâmicas.

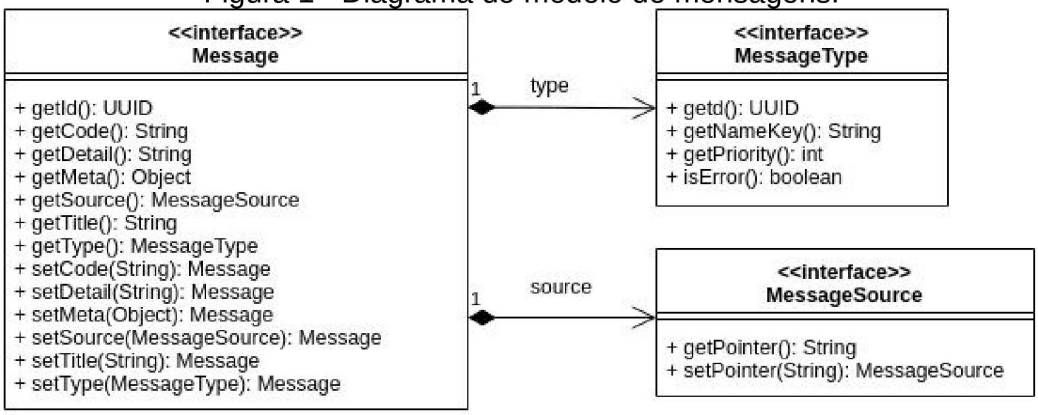
Esse interpolador utiliza a lista de locales, para internacionalizar as mensagens. Além disso, ele usa Java EL, com isso é possível definir parâmetros ou expressões EL nas mensagens. Esses parâmetros serão avaliados dinamicamente com base no que foi adicionado ao contexto global EL da requisição.

Outro recurso deste interpolador é que ele suporta múltiplos arquivos de *properties* (inclusive com o mesmo nome), além de ser possível definir um diretório do *classpath*, em que é armazenado mensagens extensas, cujo nome do arquivo representá a chave da mensagem e o conteúdo do arquivo a mensagem em si. Esse último recurso suporta mensagens com múltiplas linhas.

4.1.12 Modelo de mensagens

Visando padronização de mensagens, foi definido classes modelo que pudessem representá-las (Figura 1). Essas classes foram criadas com base no formato de erro da especificação JSON API, porém com algumas adaptações. Devido ao seu formato genérico, elas podem ser utilizadas para representar mensagens de erro, informação, aviso e sucesso, além de suportar tipos customizados.

Figura 1 - Diagrama do modelo de mensagens.



Fonte: Elaborado pelo autor, 2018.

A padronização de mensagens facilita o entendimento e permite o fácil consumo, devido ao seu formato uniforme. Outra característica desse modelo é o uso de códigos. Utilizar um código único para cada mensagem facilita o rastreamento de fluxos de execução, suporte e documentação.

4.1.13 *Builder* de mensagens

Visando integração entre o interpolador de mensagens e o modelo de mensagens foi desenvolvido um *builder* para construção de mensagens. No modelo definido (Figura 1). Toda mensagem possui um código, este deve ser único e representar uma mensagem específica.

Esse *builder* constrói mensagens por meio de um código (Código 4). Ele é utilizado para encontrar as propriedades da mensagem nos arquivos de mensagens (Código 5), por meio de um interpolador. Por padrão esse *builder* irá procurar as propriedades das mensagens utilizando um prefixo (“message.”), seguido do código, mais um sufixo (“.title”, “.detail”, “.type”), que representa o nome da propriedade (Código 5).

Código 4 - Exemplo de utilização do MessageBuilder.

```
messageBuilder.fromCode("45438");
```

Fonte: Elaborado pelo autor, 2018.

Código 5 - Exemplo de arquivo de mensagens.

```
messages.properties

message.45438.title=Info title...
message.45438.detail=Info detail...
message.45438.type=INFO
```

Fonte: Elaborado pelo autor, 2018.

Com esse *builder*, a criação de mensagens se torna mais fácil. Além disso, ele também é integrado com o wrapper de respostas HTTP. As mensagens criadas utilizando esse *builder*, são reconhecidas pelo *wrapper* e são automaticamente adicionadas a resposta da requisição.

4.1.14 *Builder* de metadados

A especificação JSON API define a propriedade meta, essa propriedade é de propósito geral. Informações que descrevem os dados principais, informações da empresa, informações de contato são exemplos de metadados.

Para facilitar a criação de metadados, foi criado um *builder* para isso. Esse *builder* suporta metadados globais e não globais. Metadados globais são definidos utilizando Eclipse Microprofile Config, e esses são enviados em todas as requisições. Os metadados globais são:

- *Home Page*

- URL de suporte
- *Email* de suporte
- Telefone de suporte
- Versão da aplicação
- Autores
- *Copyright*

Além desses, é possível definir outros metadados. Para isso basta injetar o *builder* e passar o metadado no formato chave valor.

4.1.15 Autenticação e autorização

Overjax possui um módulo que faz a segurança da aplicação de maneira *stateless* com base no *header* HTTP *Authorization*. Seu uso é simples e intuitivo, basta utilizar as annotations @Autenticated e @Authorize sobre um *endpoint* para restringir o acesso a usuários autenticados e autorizados, respectivamente.

Para garantir flexibilidade, a forma de autorização é feita através da avaliação de uma expressão booleana definida como parâmetro da anotação, usando a sintaxe do Java EL. Dessa maneira, é possível criar regras complexas de autorização. Como tradicionalmente a autorização é feita ao utilizar os papéis do usuário, essa abordagem também é suportada.

Esse módulo foi desenvolvido de maneira genérica e a avaliação do valor do *header Authorization* é delegada para um provedor de autenticação, o que permite suporte para qualquer provedor. A criação de um provedor também é simples, basta implementar a *interface* AuthenticationProvider.

Como extensão desse módulo foi implementado um provedor de autenticação que se integra com o Keycloak. Com isso, os usuários podem utilizar *tokens* providos pelo Keycloak para se autenticar e autorizar.

Durante a fase de autenticação e autorização, alguns eventos são disparados, informando o resultado da autenticação e autorização, respectivamente. Com isso, é possível fazer algum tipo de processamento de maneira fácil e desacoplada.

4.1.16 *Starters*

Para tornar o uso desse *framework* ainda mais fácil, foram definidos alguns *starters*. Esses *starters* agrupam configurações de dependências de forma que ao invés de definir várias dependências seja necessário definir apenas uma.

Atualmente, foi definido três *starters*: o principal agrupa as principais dependências necessárias para utilizar o Overjax. Ao utilizar apenas esse *starter* como dependência, já é possível utilizar o Overjax em seu *microservice*. Outro starter define um conjunto de dependências para facilitar o uso de JPA e *Bean Validation*. Como extensão do segundo *starter*, foi definido um terceiro starter que adicionalmente define as dependências necessárias para se conectar a bancos de dados Postgres.

Starters reduzem consideravelmente a configuração de dependências de um projeto. Um projeto que utilize Java EE, JCache, Eclipse Microprofile, Hibernate ORM, Hibernate Validator e Postgres, precisa definir 11 dependências (Tabela 1), enquanto que ao utilizar os starters do Overjax, bastaria definir apenas duas.

Tabela 1 - Quantidade de dependências necessárias em um projeto.

Especificação/Framework	Quantidade de dependências necessárias
Java EE	1
JCache	1
Eclipse Microprofile	4
Hibernate ORM	2
Hibernate Validator	2
Postgres	1
Total	11

Fonte: Elaborado pelo autor, 2018.

4.2 Comparativo

O Overjax não é um concorrente direto do framework Spring nem das especificações do Jakarta EE. A solução desenvolvida é uma extensão das especificações do Java, tendo por função suprir algumas necessidades não abordadas pelo Java (Jakarta EE), Spring e outros *frameworks*.

Na Tabela 2 temos um comparativo entre essas tecnologias, a partir da ótica de um desenvolvedor que irá utilizar uma dessas três tecnologias para criar um

novo projeto, como o any-address por exemplo. Esse comparativo foi feito com base em experiências de trabalho nessas tecnologias.

Tabela 2 - Quadro comparativo.

Recurso	**OVERJAX**	**Spring**	**Jakarta EE**
Configuração externalizada	✓	✓*	×
Integração com CDI	✓	×	✓
Integração nativa com Eclipse Microprofile	✓	×	×
Configuração baseada em eventos	✓	×	×
Gerenciamento de dependências	✓	✓	×
Configuração global do formato de resposta JSON	✓	✓	×
Contexto EL global por requisição	✓	×	×
Tratamento de exceções	✓	✓*	×
Disparo de eventos durante tratamento de exceções	✓	×	×
Tratamento de exceções delegado	✓	×	×
Wrapper de respostas HTTP	✓	×	×
Configuração de log	✓	×	×
Internacionalização	✓	✓	×
Leitura múltipla do corpo da requisição	✓	×	×
Interpolador de mensagens dinâmicas	✓	✓*	×
Modelo de mensagens	✓	×	×
Builder de mensagens	✓	×	×
Builder de metadados	✓	×	×
Autenticação e autorização	✓	✓	✓
Disparo de eventos durante autenticação e autorização	✓	×	×
Integração com Keycloak	✓	✓	✓
Starters	✓	✓	×

*[1] Não utiliza a especificação Eclipse Microprofile Config.

*[2] Não faz o tratamento de ConstraintViolationException.

*[3] Não suporta interpolação de expressões utilizando Java EL.

Fonte: Elaborado pelo autor, 2018.

5 DETALHAMENTO DO *FRAMEWORK*

Overjax é um projeto desenvolvido em java, que tem como base JAX-RS e Eclipse Microprofile. Ele foi criado com o uso de outros *frameworks*, especificações e bibliotecas consolidadas no mercado. Porém, além disso, ele possui implementações próprias, como tratamento de exceções, configuração baseada em eventos e padronização, *wrapper* de resposta, *builder* de mensagens e metadados e mecanismo de autenticação e autorização.

Em paralelo ao Overjax, foi desenvolvido outros projetos que também iriam ser utilizados por ele. Os principais foram *infinity-utils* e *message-interpolator*.

5.1 Arquitetura

Overjax foi criado em uma arquitetura multi-módulo. Para fazer o gerenciamento dessa arquitetura foi utilizado Maven. Em outras palavras, Overjax é um projeto Maven composto de outros sub projetos Maven, também chamados de módulos.

5.2 Módulos

5.2.1 Overjax-dependencies

Esse módulo é constituído por um arquivo pom.xml, nele é definido todas as dependências que são utilizadas pelos demais módulos. Essas dependências foram definidas dentro da *tag dependencyManagement* (Código 6), com o uso de properties para o versionamento. Com isso, os outros módulos e até as aplicações que utilizam o Overjax, podem utilizar as dependências definidas sem a necessidade de especificar a versão.

O benefício dessa estrutura é a centralização do gerenciamento de versão das dependências. Isso garante uma fácil atualização de versão e compatibilidade entre os módulos, já que todos eles utilizam a mesma versão de uma dada dependência.

Código 6 - Fragmento do arquivo pom.xml.

```
                <dependency>
                        <groupId>javax.ws.rs</groupId>
                        <artifactId>javax.ws.rs-api</artifactId>
                        <version>${jaxrs.version}</version>
                        <scope>provided</scope>
                </dependency>

                <dependency>
                        <groupId>javax.cache</groupId>
                        <artifactId>cache-api</artifactId>
                        <version>${jcache.version}</version>
                        <scope>provided</scope>
                </dependency>

                <dependency>
                        <groupId>javax.enterprise</groupId>
                        <artifactId>cdi-api</artifactId>
                        <version>${cdi.version}</version>
                        <scope>provided</scope>
                </dependency>

                <dependency>
                        <groupId>javax</groupId>
                        <artifactId>javaee-api</artifactId>
                        <version>${javaee.version}</version>
                        <scope>provided</scope>
                </dependency>
        </dependencies>
</dependencyManagement>
```

Fonte: Elaborado pelo autor, 2018.

5.2.2 Overjax-annotation

Visando um *framework com pouco* acoplamento, o Overjax foi desenvolvido utilizando annotations. Por meio delas é aplicado semântica aos locais onde elas estão sendo utilizadas para realizar alguma ação. Este módulo foi desenvolvido para agrupar as *annotations* que são comumente utilizadas. Entre elas temos:

5.2.2.1 @Controller

Annotation criada para ser utilizada nas classes de controle dos *microservices* que utilizam Overjax, ela aplica outras *annotations* em cascata, como @RequestScoped e @NameBinding, que são *annotations* da especificação CDI.

Com @NameBinding torna-se possível criar funcionalidades associadas a quem está com a annotation @Controller. Uma possível funcionalidade seria a criação de interceptores.

5.2.2.2 @ExceptionHandling

Um dos recursos do CDI é a emissão de eventos. @ExceptionHandling tem como função marcar semanticamente, que o evento anotado é um evento de manipulação de exceção.

5.2.2.3 @MappedException

Essa *annotation* foi criada para ser um *qualifier*, que é utilizado por classes que implementam a *interface* javax.ws.rs.ext.ExceptionMapper. Assim, é possível injetar as exceções que já foram mapeadas durante uma dada requisição e verificar se uma dada exceção já foi mapeada ou não, o que evita um processamento desnecessário.

5.2.2.4 @Property

Uma *annotation* de propósito geral constituída de uma chave e um valor. Atualmente, seu uso é feito no módulo overjax-security e no overjax-security-keycloak.

5.2.2.5 @RequestBody

Assim como @MappedException, essa annotation também é um *qualifier*, que semanticamente representa que o que vai ser produzido ou injetado é o corpo da requisição. Hoje ela é aplicada no módulo overjax-core.

5.2.2.6 @WrapResponse

Manter um padrão no formato das requisições e respostas de uma aplicação é algo benéfico, pois permite a criação de algo genérico que pode ser aplicado em grande parte da aplicação. JSON API é uma especificação que define um formato de requisições e respostas baseadas em JSON. Tomando como base

isso, foi desenvolvido um manipulador de respostas de requisições para aplicar um formato de resposta padrão.

A *annotation* @WrapResponse tem como propósito indicar quais são os *endpoints* que a manipulação deve ser feita. Essa manipulação é feita no módulo overjax-core.

5.2.3 Overjax-config

Overjax foi desenvolvido por meio de eventos CDI e Eclipse Microprofile Config para criar configurações facilmente alteráveis e totalmente externalizáveis. Esse módulo tem como objetivo principal manter todas as propriedades de configuração que são utilizadas por todos os outros módulos.

Configurações de objetos complexos, como JsonbConfig e ElProcessor são feitas através de eventos (Código 7), já configurações mais simples como *flags*, por exemplo, são feitas utilizando Eclipse Microprofile Config (Código 8).

Código 7 - Exemplo de configuração baseada em evento.

```
public void register(@Observes @Priority(ENUMERATION_SERIALIZER_JSONB_CONFIG) @Config(JSONB) JsonbConfig config) {
    config.withSerializers(this);
    Configs.log(logger, JSONB, "withSerializer", EnumerationSerializer.class.getName());
}
```

Fonte: Elaborado pelo autor, 2018.

Código 8 - Exemplo de configuração utilizando Eclipse microprofile Config.

```
@Inject
@ConfigProperty(name = RESPONSE_METADATA_HOME_PAGE)
private Optional<String> homePage;

@Inject
@ConfigProperty(name = RESPONSE_METADATA_SUPPORT_URL)
private Optional<String> supportUrl;

@Inject
@ConfigProperty(name = RESPONSE_METADATA_SUPPORT_EMAIL)
private Optional<String> supportEmail;

@Inject
@ConfigProperty(name = RESPONSE_METADATA_SUPPORT_PHONE)
private Optional<String> supportPhone;
```

Fonte: Elaborado pelo autor, 2018.

5.2.4 Overjax-core

Esse módulo é o núcleo do projeto, e é constituído por *builders*, *filters*, *converters*, *mappers*, *beans* e *providers*. Atualmente, suas principais funcionalidades são: fornecimento de instâncias através de *providers*, formatação das respostas dos

endpoints (que utilizam a *annotation* @WrapResponse), o tratamento de exceções e criação de mensagens e metadados.

5.2.4.1 Providers

Quando se utiliza a annotation @Inject do CDI, uma instância de uma classe é injetada. E para que isso ocorra, é necessário existir algum *provider*. Por padrão, o CDI é capaz de prover instâncias de classes que possuem um construtor padrão, algumas outras exigem a criação um *provider* customizado.

O core do Overjax fornece alguns *providers*. Entre eles temos:

5.2.4.1.1 El *processor provider*

ElProcessorProvider: Prover uma instância da classe ELProcessor do Java EL para cada requisição, o que permite ter um contexto global isolado por requisição. Esse *provider* dispara um evento de configuração dessa instância, permitindo que ela possa ser configurada de maneira desacoplada.

5.2.4.1.2 Jsonb *provider*

JsonbProvider: Prover instâncias da classe Jsonb, que é utilizada para fazer a conversão de objetos Java para objetos no formato JSON. Assim como o ElProcessorProvider, esse *provider* também dispara um evento de configuração passando a instância criada.

5.2.4.1.3 *Locale provider*

LocaleProvider: Fornece uma lista de *Locales*, em ordem de prioridade, para cada requisição.

5.2.4.1.4 *Logger provider*

LoggerProvider: Cria instâncias da classe *Logger* configurada com base no local em que essa instância será injetada, permitindo ter um ponto central de configuração de *log*.

5.2.4.1.5 *Request body provider*

RequestBodyProvider: Prover um *array* de *bytes* que representa o corpo da requisição, permitindo múltiplas leituras desses *bytes* em diferentes pontos da aplicação.

5.2.4.1.6 Formatação de respostas, criação de mensagens e metadados

Como mencionado anteriormente, a *annotation* @WrapResponse indica quais *endpoints* devem ter suas respostas formatadas. Esse processo de formatação consiste em inserir os dados retornados pelo *endpoint* em um objeto da classe ResponseWrapper. O formato dessa classe foi criado com base na especificação JSON API. A especificação diz que toda resposta deve ser um objeto JSON, semelhante ao exibido no Código 9:

Código 9 - Exemplo de resposta de endpoint com @WrapResponse.

```
{
    "data": {
        "foo": "bar"
    },
    "errors": [
        {
            "id": "f7bf0d15-c00b-4c63-b22c-64e5912a4432",
            "code": "45438",
            "type": {
                "id": "ERROR",
                "name": "Error"
            },
            "title": "Error title...",
            "detail": "Erro detail.."
        }
    ],
    "meta": {
        "meta1": "value1",
        "meta2": "value2",
        "mensagens:": [
            {
                "id": "47bf0d15-c00b-4c63-b22c-64e5912a4433",
                "code": "45438",
                "type": {
                    "id": "INFO",
                    "name": "Info"
                },
                "title": "Info title...",
                "detail": "Info detail.."
            },
            {
                "id": "87bf0d15-c00b-4c63-b22c-64e5912a4434",
                "code": "45438",
                "type": {
                    "id": "WARNING",
                    "name": "Warning"
                },
                "title": "Warning title...",
                "detail": "Warning detail.."
            }
        ]
    }
}
```

Fonte: Elaborado pelo autor, 2018.

A classe ResponseWrapper foi estruturada para atender esse formato. Para facilitar inclusão de metadados foi criado a classe ResponseMetadataBuilder, ela pode ser injetada em algum componente utilizando a annotation do CDI @Inject.

Paralelo a isso, foi desenvolvido um projeto chamado message-interpolator, que usa Java EL para interpolar mensagens de maneira internacionalizada, que por sua vez, foi utilizado como base para criação da classe MessageBuilder. Esse *builder* é responsável por construir mensagens de formato semelhante as mensagens exibidas no Código 9.

5.2.4.2 Tratamento de exceções

JAX-RS possui a interface ExceptionMapper que permite fazer o tratamento de exceções que ocorreram durante uma requisição. Foi desenvolvido os seguintes mappers:

- ConstraintViolationMapper
- DefaultExceptionMapper
- ThrowableMapper
- WebApplicationExceptionMapper

Todos esses mappers fazem o uso da *annotation* @ExceptionHandling para disparar eventos passando a exceção que está sendo tratada (Código 10).

Código 10 - Fragmento da classe ThrowableMapper, mostrando a utilização da annotation e o disparo do evento através do método fireAsync.

```
@Inject
@ExceptionHandling
private Event<Throwable> throwableEvent;

@Override
protected Response handle(Throwable ex) {
    logger.log(Level.SEVERE, ex.getMessage(), ex);

    throwableEvent.fireAsync(ex);
```

Fonte: Elaborado pelo autor, 2018.

A classe ConstraintViolationMapper faz o tratamento de ConstraintViolationException, essa exceção é lançada quando um objeto que usa *Bean Validation* está inválido. Esse *mapper* envia as violações encontradas para uma instância da classe MessageBuilder, que por sua vez, converte e adiciona as violações como erro no corpo da resposta da requisição.

5.2.5 Overjax-data

Atualmente, esse módulo consiste em duas classes que podem ser utilizadas para simplificar a criação de entidades JPA. A classe AbstractEntity implementa os métodos *equals* e *hashCode* de maneira que subclasses não necessitem implementá-los novamente. Já a classe SimpleEntity (Código 11), é uma subclasse da classe AbstractEntity que já possui definido o campo *id* e *version*.

Código 11 - Fragmento da classe SimpleEntity.

```
@MappedSuperclass
public abstract class SimpleEntity<T extends SimpleEntity<T>> extends AbstractEntity<T, UUID> {

    private static final long serialVersionUID = -1921331366284657616L;

    @Id
    @GeneratedValue
    @Column(nullable = false)
    private UUID id;

    @Version
    private int version;
```

Fonte: Elaborado pelo autor, 2018.

5.2.6 Overjax-security

Para que um *microservice* possa escalar (crescer em um ambiente de cloud) ele tem que ser *stateless*. ou seja, ele não pode armazenar dados na sessão do usuário que está fazendo as requisições. Dados como, quem é o usuário que está acessando, devem ser passados em cada requisição.

Atualmente, uma forma consolidada pelo mercado para transmitir informações do usuário em cada requisição, é por meio de *token* JWT. Com base nisso, foi desenvolvido um módulo genérico para fazer a segurança da aplicação.

A parte central desse modulo ocorre em dois filtros. O primeiro é responsável por fazer o controle de autenticação, garantindo que o usuário que está acessando um *endpoints* é realmente quem ele diz ser. Já o segundo filtro, tem a função de garantir que o usuário que está acessando um dado *endpoint* tenha autorização para acessar esse *endpoint*.

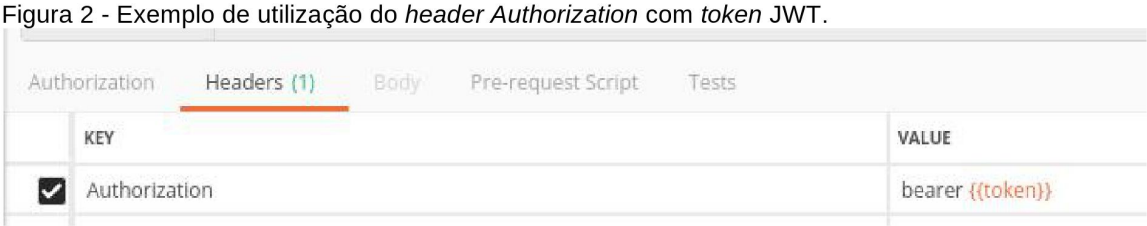
5.2.6.1 Filtro de autenticação

O CDI fornece uma maneira de fazer uma ligação entre um *endpoint* e um filtro de maneira desacoplada através do uso de uma *annotation* customizada. Sabendo disso, foi desenvolvido uma *annotation* chamada @Authenticated e a

classe AuthenticationFilter, para que quem estiver usando esse módulo possa utilizar a *annotation* e indicar que um *endpoint* só deve ser acessado por usuários que estão autenticados.

O protocolo HTTP possui um *header* chamado *Authorization*, que é utilizado para passar informações de autenticação (Figura 2).

Figura 2 - Exemplo de utilização do *header Authorization* com *token* JWT.



Fonte: Elaborado pelo autor, 2018.

Esse overjax-security é um módulo genérico. Nele não é feito nenhum tipo de análise nas informações desse *header*, para identificar se o usuário que está acessando é realmente quem ele diz ser. A realização dessa análise foi delegada para quem implementar a *interface* AuthenticationProvider (Código 12).

Código 12 - Interface AuthenticationProvider.

```
public interface AuthenticationProvider {

    public default String getAuthenticationScheme() {
        return "bearer";
    }

    public String getRealm();

    public default String getToken(String authorizationHeader) {
        if (authorizationHeader != null) {
            authorizationHeader = authorizationHeader.trim();
            String authenticationScheme = getAuthenticationScheme();
            if (authorizationHeader.length() > authenticationScheme.length()) {
                if (authorizationHeader.substring(0, authenticationScheme.length()).equalsIgnoreCase(authenticationScheme)) {
                    return authorizationHeader.substring(authenticationScheme.length()).trim();
                }
            }
        }
        return null;
    }

    public TokenAuthentication validate(String tokenString) throws Exception;

}
```

Fonte: Elaborado pelo autor, 2018.

Quando um usuário tenta acessar um *endpoint* que possui a *annotation* @Authenticated, ele vai passar pelo filtro AuthenticationFilter. Esse filtro envia o valor do *header Authorization* para um AuthenticationProvider, que por sua vez, vai validar e retornar uma instância da classe TokenAuthentication (Código 13). Essa classe representa o resultado da validação.

Código 13 - Fragmento da classe TokenAuthentication.

```
public class TokenAuthentication {

    private final AuthenticationType RESULT_TYPE;
    private final Object TOKEN;
    private final String ERROR_DESCRIPTION;
    private final Exception EXCEPTION;
    private final Set<String> USER_ROLES;
```

Fonte: Elaborado pelo autor, 2018.

Dependendo do tipo de resultado (Código 14), o filtro AuthenticationFilter permite que o usuário acesse o endpoint solicitado ou não.

Código 14 - Enum que representa os possíveis tipos de resultados de uma autenticação.

```
public enum AuthenticationType {

    ACCESS_DENIED,
    PUBLIC_ACCESS,
    NO_AUTHENTICATION,
    INVALID,
    EXPIRED,
    AUTHENTICATED;

}
```

Fonte: Elaborado pelo autor, 2018.

Além de fazer essa filtragem, a classe AuthenticationFilter também dispara eventos com base no resultado da validação (Código 15 e Código 16) e para cada tipo de resultado, um evento é disparado.

Código 15 - Fragmento da classe AuthenticationFilter, ao chamar o método fire um evento passando o resultado da autenticação é disparado.

```
    switch (resultType) {
        case AUTHENTICATED:
            logger.fine("Authentication successful");
            securityInfo.setTokenAuthentication(tokenAuthentication);
            authenticatedUserEvent.fire(tokenAuthentication);
```

Fonte: Elaborado pelo autor, 2018.

Código 16 - Exemplo de utilização do evento, disparado pela classe AuthenticationFilter.

```
public void onAuthenticationEvent(@Observes @Authentication(AUTHENTICATED) TokenAuthentication tokenAuthentication) {
    //...
    System.out.println(tokenAuthentication);
}
```

Fonte: Elaborado pelo autor, 2018.

O valor *AUTHENTICATED* na *annotation* @Authentication (Código 16) indica que esse método será notificado quando a autenticação for bem sucedida.

Para ser notificado dos outros tipos de resultado, basta alterar o valor definido na *annotation* @Authentication.

5.2.6.2 Filtro de authorização

Para fazer o controle de autorização, foi criado o filtro AuthenticationFilter e a *annotation* @Authorize. Esse filtro é dependente do AuthenticationFilter. Quando um usuário faz uma requisição, o filtro AuthenticationFilter disponibiliza uma instância de TokenAuthentication. Nela existe um método chamado getUserRoles, que retorna os papéis que o usuário possui.

Dentre as especificações do Java, existe uma *annotation* chamada @RolesAllowed, que tem um propósito semelhante ao da *annotation* @Authorize, porém essa possui um diferencial. Ela não está limitada a restrição por papéis que o usuário possui. Ao usar @Authorize você deve definir uma expressão booleana dentro de uma *String*. Essa expressão é avaliada usando Java EL (Código 17). Caso o resultado dessa expressão seja verdadeiro, o filtro permite o acesso ao *endpoint*. Caso contrário, a requisição será negada com o código de erro HTTP 401 (não autorizado).

Código 17 - Exemplo de utilização da annotation @Authorize.

```
@GET
@Path("{id}")
@Authorize("$.hasRole('GET_COUNTRY')")
public Response find(UUID id) {
    Country country = countryService.findByID(id);
    return Response.status(country == null ? NOT_FOUND : OK).build();
}
```

Fonte: Elaborado pelo autor, 2018.

Quando um evento de configuração é disparado pela classe ElProcessorProvider, do módulo overjax-core, uma instância da classe SecurityInfo se auto registra no ELProcessor, utilizando o símbolo $ como nome (Código 18).

Código 18 - Método da classe SecurityInfo onde é feito o seu registro no ELProcessor.

```
@SuppressWarnings("unused")
private void register(@Observes @Priority(SECURITY_INFO_EL_PROCESSOR_CONFIG) @Config(EL_PROCESSOR) ELProcessor processor) {
    processor.defineBean("$", this);
    Configs.log(logger, EL_PROCESSOR, "defineBean", "$=" + SecurityInfo.class.getName());
}
```

Fonte: Elaborado pelo autor, 2018.

Devido ao ElProcessorProvider fornecer um ElProcessor global, a instância da classe SecurityInfo está disponível em toda a aplicação durante a

requisição. Com isso, é possível executar seus métodos dentro de uma expressão EL de qualquer endpoint da aplicação.

Assim como no filtro anterior, a classe AuthorizationFilter também dispara eventos. Ela emite um quando o usuário está autorizado, e outro quando ele não está. Para ser notificado desses eventos, deve-se criar um método, como mostra no Código 19:

Código 19 - Exemplo de utilização do evento, disparado pela classe AuthorizationFilter.

```
public void onAuthorizationEvent(@Observes @Authorization(true) TokenAuthorization tokenAuthorization) {
    //...
    System.out.println(tokenAuthorization);
}
```

Fonte: Elaborado pelo autor, 2018.

O booleano *true* na *annotation* @Authorization (Código 19) indica que esse método será notificado quando a autorização for bem sucedida. Utilizando *false*, o tipo de notificação será o inverso.

5.2.7 Overjax-security-keycloak

Esse módulo é uma extensão do overjax-security. Seu propósito é integrar o overjax-security com o Keycloak. Para isso, foi criada uma classe que implementa a interface AuthenticationProvider. O Keycloak é uma ferramenta que possui diversos adapters para diversas linguagens, o que facilita a comunicação com a ferramenta.

Para fazer essa integração foi utilizado as dependências keycloak-servlet-filter-adapter e keycloak-adapter-core, conforme o Código 20.

Código 20 - Dependências do módulo overjax-security-keycloak.

```
<dependencies>
    <dependency>
        <groupId>org.keycloak</groupId>
        <artifactId>keycloak-servlet-filter-adapter</artifactId>
    </dependency>

    <dependency>
        <groupId>org.keycloak</groupId>
        <artifactId>keycloak-adapter-core</artifactId>
    </dependency>
```

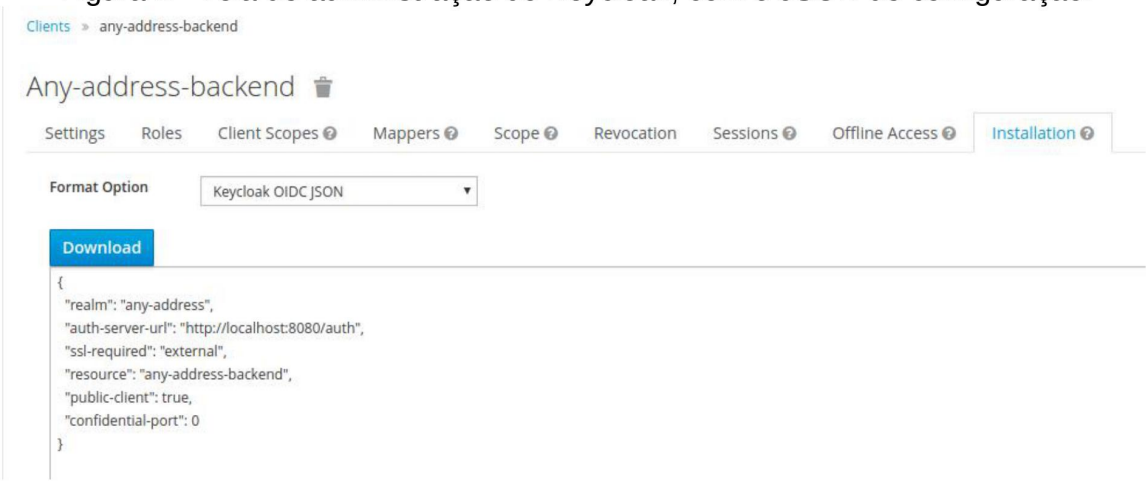
Fonte: Elaborado pelo autor, 2018.

Uma das classes existentes nessas dependências é a AdapterTokenVerifier. Ela possui um método chamado verifyToken, que é responsável por validar um dado *token*. Para fazer isso, é necessário que seja

passada uma instância da classe KeycloakDeployment. Essa classe possui a configuração necessária para a validação do *token*.

Para que a integração funcione, é necessário ter o serviço do Keycloak configurado e em execução. No console de administração do Keycloak existe uma aba chamada *Installation* (Figura 3). Nela é possível obter a configuração necessária exigida pela instância da classe KeycloakDeployment.

Figura 3 - Tela de administração do Keycloak, com o JSON de configuração.



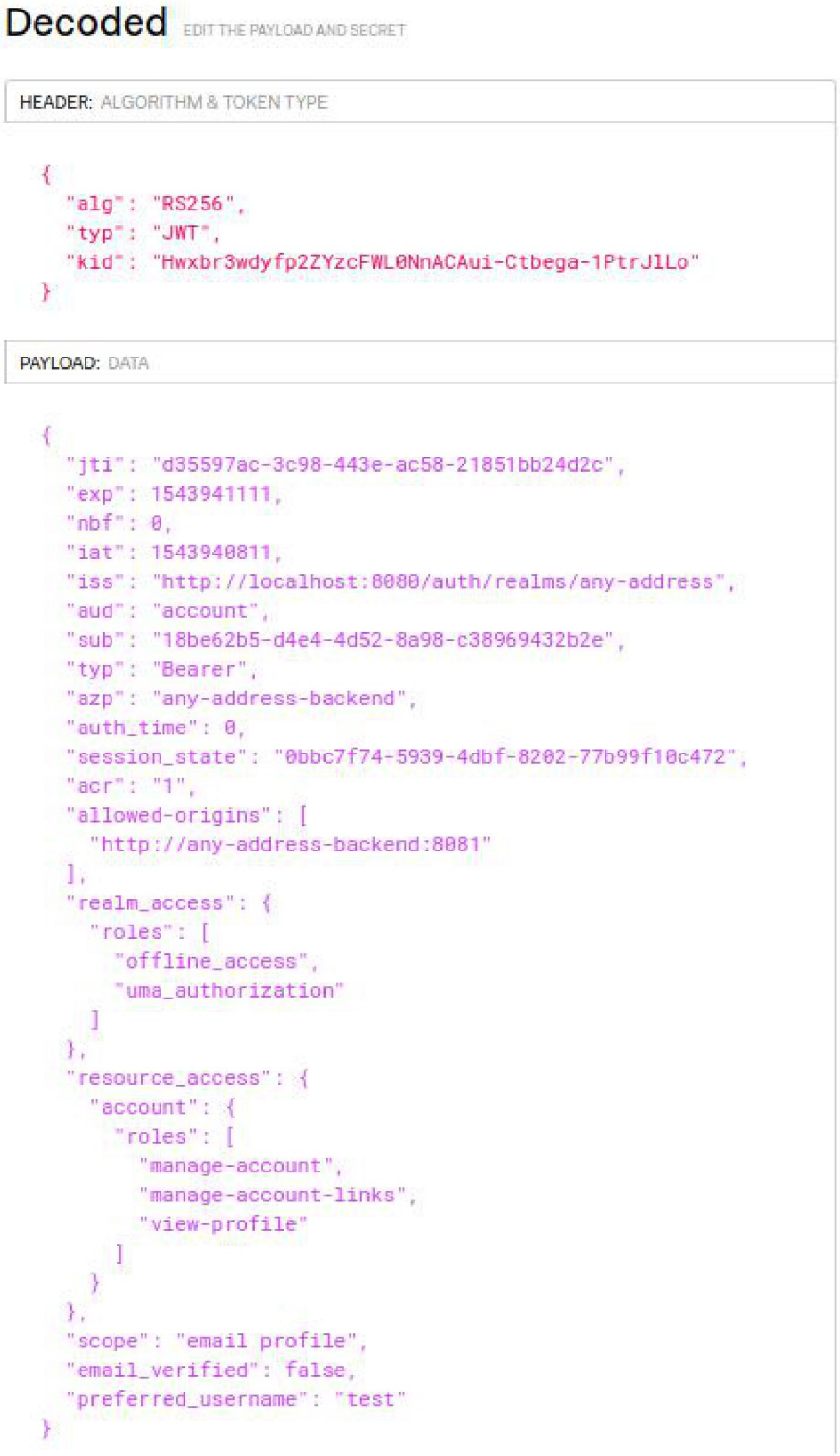
Fonte: Elaborado pelo autor, 2018.

Esse módulo faz o uso da especificação Eclipse Microprofile Config e assim, é possível realizar a configuração do KeycloakDeployment e dos papéis de usuário que serão considerados. Para construir uma instância da classe KeycloakDeployment, é necessário passar o JSON de configuração (Figura 3) em forma de InputStream. Na classe ConfigProperties do módulo overjax-core existe a constante SECURITY_KEYCLOAK_DEPLOYMENT. Nela é definida a property padrão que será utilizada para obter o caminho do arquivo em que se encontra o JSON.

O Keycloak é constituído em *realms* ou reinos. Um reino é um conjunto de clientes, que seria o equivalente a um conjunto de aplicações ou *microservices* de uma organização, por exemplo. Nele é possível definir para cada usuário, os papéis que ele possui no reino, que devem ser aplicados para todos os clientes, assim como também é possível definir papéis para clientes específicos (Figura 4).

Também na classe ConfigProperties existe a constante SECURITY_KEYCLOAK_RESOURCES_ACCESS. Nela é definida a *property* padrão que será utilizada para obter quais são os clientes que devem ter seus papéis considerados.

Figura 4 - Exemplo de *token* decodificado gerado pelo Keycloak.



Fonte: JWT, 2018.

Na Figura 4, existe a propriedade “resource_access”. Nela são definidas as permissões que o usuário possui por *client*. Nesse exemplo, o usuário possui as

permissões “manage-account”, “manage-account-links” e “view-profile”, para o client “account”.

A property SECURITY_KEYCLOAK_RESOURCES_ACCESS foi criada para dar flexibilidade e permitir que seja definido zero ou N “resource_access” a serem considerados.

5.2.8 Overjax-starters

Para facilitar o desenvolvimento de *microservices* que irão utilizar Overjax, foi criado alguns módulos extras para reduzir a necessidade de importação de várias dependências. O módulo overjax-starters é a base para criação dos demais *starters*, e consiste em apenas um simples arquivo pom.xml.

5.2.8.1 Overjax-starter

Esse módulo é o principal *starter*, e é um Pom *Parent* (Código 21), que tem como função definir as principais dependências exigidas para o funcionamento do Overjax, com adição de algumas especificações do Java e do projeto Eclipse Microprofile.

Código 21 - Exemplo de utilização do overjax-starter.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>com.seudev.overjax</groupId>
        <artifactId>overjax-starter</artifactId>
        <version>0.0.1-SNAPSHOT</version>
    </parent>
```

Fonte: Elaborado pelo autor, 2018.

5.2.8.2 Overjax-starter-data-jpa

Módulo constituído de um arquivo pom.xml, cuja a função é definir um conjunto de dependências necessárias para criar um microservice que utilize JPA. Dentre as dependências declaradas temos:

- hibernate-validator
- hibernate-validator-cdi
- hibernate-core

- hibernate-c3p0
- overjax-data

5.2.8.3 Overjax-starter-data-jpa-postgres

Extensão do overjax-starter-data-jpa com a adição do *driver* JDBC para bancos de dados PostgresSQL.

6 APLICAÇÃO DE TESTE

Para validar se o *framework* desenvolvido estava se comportando como esperado, foi desenvolvido uma *microservice* de teste chamado *any-address*. Sua função é armazenar informações de país, estado, cidade, CEP e endereço (Figura 5). Para dar vida a esse *microservice* foi utilizado o Payara Micro em conjunto com Docker, PostgresSQL, pgAdmin e Keycloak.

Figura 5 - Endpoints do *microservice any-address*

Fonte: Swagger, 2019.

6.1 Configuração

Para criar esse *microservice* utilizando o Overjax foi necessário fazer as configurações descritas nas subseções a seguir.

6.1.1 Arquivo pom.xml

Abaixo é descrito o que é necessário definir no arquivo pom.xml:

- Definir como Pom *Parent* (Código 22) o módulo *overjax-starter;*
- Adicionar como dependências os módulos *overjax-security-keycloak* e *overjax-starter-data-jpa-postgres* (Código 23).

Código 22 - Definição do Pom Parent no any-address.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>com.seudev.overjax</groupId>
        <artifactId>overjax-starter</artifactId>
        <version>0.0.1-SNAPSHOT</version>
    </parent>
```

Fonte: Elaborado pelo autor, 2018.

Código 23 - Definição das dependências do any-address.

```xml
    <dependencies>
        <dependency>
            <groupId>com.seudev.overjax</groupId>
            <artifactId>overjax-security-keycloak</artifactId>
        </dependency>

        <dependency>
            <groupId>com.seudev.overjax</groupId>
            <artifactId>overjax-starter-data-jpa-postgres</artifactId>
        </dependency>
    </dependencies>
</project>
```

Fonte: Elaborado pelo autor, 2018.

6.1.2 *Controllers*

Abaixo é descrito o que é necessário para criação dos *controllers:*

- Criação da classe App (Código 24), para definir o contexto raiz;
- Adição da *annotations* @Controller (Código 25) para indicar que a classe é um *controller*;
- Adição da *annotation* @WrapResponse (Código 25) para indicar que as respostas das requisições devem ser formatadas;
- Adição da *annotation* @Authenticated (Código 25), para indicar que os *endpoints* necessitam de autenticação;
- Adição da annotation @Authorize (Código 25) para indicar que o *endpoint* necessita de autorização.

Código 24 - Classe App do projeto any-address.

```java
@ApplicationPath("/")
public class App extends Application {

}
```

Fonte: Elaborado pelo autor, 2018.

Código 25 - Fragmento da classe CountryController do projeto any-address.

```
@Controller
@WrapResponse
@Authenticated
@Path("/api/v1/country")
@Produces(APPLICATION_JSON)
@Consumes(APPLICATION_JSON)
public class CountryController {

    @Context
    private UriInfo uriInfo;

    @EJB
    private CountryService countryService;

    @GET
    @Path("{id}")
    @Authorize("$.hasRole('GET_COUNTRY')")
    public Response find(UUID id) {
        Country country = countryService.findByID(id);
        return Response.status(country == null ? NOT_FOUND : OK).build();
    }
```

Fonte: Elaborado pelo autor, 2018.

6.1.3 Envio de mensagens customizadas

Abaixo é descrito o que é necessário para enviar mensagens:

- Adição da *annotation* @WrapResponse para indicar que as respostas das requisições devem ser formatadas;
- Declaração de atributo do tipo MessageBuilder mais adição da *annotation* @Inject (Código 26);
- Executar chamada ao método fromCode passando o código da mensagem (Código 26);
- Definir mensagem no arquivo de *properties* *src/main/resources/messages/messages.properties* (Código 27).

Código 26 - Fragmento da classe IndexController do projeto any-address.

```
@Inject
private MessageBuilder messageBuilder;

@GET
public Response index() {
    Message message = messageBuilder.fromCode("index", false);
    return Response.ok(message).build();
}
```

Fonte: Elaborado pelo autor, 2018.

Código 27 - Fragmento do arquivo de mensagens do projeto any-address.

```
message.index.title=Any Address
message.index.detail=A microservice to store country, state, city, CEP and address informations.
message.index.type=INFO
```

Fonte: Elaborado pelo autor, 2018.

6.1.4 Metadados globais

Abaixo é descrito o que é necessário para configurar os metadados globais:

- Adição das variáveis de ambiente no arquivo *docker-compose.yaml* (Código 28).

Código 28 - Definição dos metadados globais do projeto any-address.

```
OVERJAX_RESPONSE_METADATA_HOME_PAGE: "https://seudev.com"
OVERJAX_RESPONSE_METADATA_SUPPORT_EMAIL: "contact@seudev.com"
OVERJAX_RESPONSE_METADATA_APPLICATION_VERSION: "0.0.1"
OVERJAX_RESPONSE_METADATA_AUTHORS: "Thomás Sousa Silva"
OVERJAX_RESPONSE_METADATA_COPYRIGHT: "Copyright © 2018-2018 Seudev"
```

Fonte: Elaborado pelo autor, 2018.

6.2 Configurações extras

Para que o *microservice* any-address pudesse ser executado, foram necessárias outras configurações, que não estão ligadas ao Overjax. Entre elas temos: o Keycloak, banco de dados do Keycloak, banco de dados do any-address e o servidor de backend Payara Micro.

Com a intenção de tornar essas configurações extras mais fáceis, utilizou-se o Docker. Para isso, foi criado um arquivo *docker-compose.yaml* e nele foram definidos todos os serviços necessários para execução do *any-address*. Já no arquivo *docker-compose.override.yaml*, foi definido configurações extras desses serviços, como usuário e senha de acesso (Código 29).

Código 29 - Fragmento do arquivo web.xml, com a configuração do data source do projeto any-address, utilizando variáveis de ambiente.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
    version="3.1">

    <data-source>
        <name>java:app/any-address</name>
        <class-name>com.mchange.v2.c3p0.ComboPooledDataSource</class-name>
        <property>
            <name>dataSourceName</name>
            <value>${ENV=DATABASE_POOL_NAME}</value>
        </property>
        <property>
            <name>jdbcUrl</name>
            <value>${ENV=DATABASE_URL}</value>
        </property>
        <property>
            <name>user</name>
            <value>${ENV=DATABASE_USER}</value>
        </property>
        <property>
            <name>password</name>
            <value>${ENV=DATABASE_PASSWORD}</value>
        </property>
        <property>
            <name>maxPoolSize</name>
            <value>${ENV=DATABASE_MAX_POOL_SIZE}</value>
        </property>
```

Fonte: Elaborado pelo autor, 2018.

Para integração do Keycloak foi criado o arquivo *services/any-address-backend/config/.secrets/overjax.security.keycloak.deployment* (Código 30), cujo arquivo recebe o mesmo nome da property padrão utilizado pelo Overjax. Para esse arquivo ser reconhecido, foi adicionado o *services/any-address-backend/config/post-boot-commands.txt* (Código 31), que é reconhecido pelo servidor Payara Micro.

Código 30 - Fragmento do arquivo de configuração do Keycloak do projeto any-address.

```json
{
  "realm": "any-address",
  "realm-public-key": "MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAn...",
  "auth-server-url": "http://localhost:8080/auth",
  "ssl-required": "external",
  "resource": "any-address-backend",
  "public-client": true,
  "confidential-port": 0
}
```

Fonte: Elaborado pelo autor, 2018.

Código 31 - Arquivo de configuração do Payara Micro.

```
set-config-secrets-dir --directory=/etc/config/.secrets
```

Fonte: Elaborado pelo autor, 2018.

6.3 Configurações de *properties*

Devido a todas as configurações do Overjax terem sido feitas por meio do Eclipse Microprofile Config, essas configurações podem ser definidas de várias formas. A escolhida foi o uso de variável de ambiente, mas ele não está limitado a essa forma. Inclusive, essa especificação suporta mescla de várias fontes de configuração.

6.4 Domínio do *microservice*

Para representar o domínio, foram criadas algumas entidades que fazem o uso de *annotations* do JPA e *Bean Validation* (Código 32). Junto a isso, foram produzidas classes de serviço que fazem a persistência e acesso aos dados, utilizando o EntityManager em conjunto com transações JTA, por meio de *annotations* (Código 33).

Código 32 - Fragmento da classe *State* do projeto any-address.

```
@Entity
@Table(uniqueConstraints = {
        @UniqueConstraint(name = "uq_state_abbreviation", columnNames = {"country", "abbreviation"}),
        @UniqueConstraint(name = "uq_state_name", columnNames = {"country", "name"})
})
public class State extends SimpleEntity<State> {

    private static final long serialVersionUID = 6076755231220716567L;
    private static final int ABBREVIATION_LENGTH = 5;
    private static final int NAME_LENGTH = 50;

    @NotBlank
    @Size(max = ABBREVIATION_LENGTH)
    @Column(name = "abbreviation", nullable = false, length = ABBREVIATION_LENGTH)
    private String abbreviation;
```

Fonte: Elaborado pelo autor, 2018.

Código 33 - Fragmento da classe CountryService do projeto any-address.

```
@LocalBean
@Stateless
@Transactional
@ValidateOnExecution(type = ALL)
public class CountryService extends AbstractService {

    @Transactional(SUPPORTS)
    public List<Country> findAll(CountryFilter filter) {
        CriteriaQuery<Country> query = cb.createQuery(Country.class);
        Root<Country> from = query.from(Country.class);

        List<Predicate> predicates = new ArrayList<>(2);
        if (filter.getAbbreviation() != null) {
            predicates.add(cb.like(from.get(Country_.abbreviation), filter.getAbbreviation()));
        }

        if (filter.getName() != null) {
            predicates.add(cb.like(from.get(Country_.name), filter.getName()));
        }

        query.where(predicates.toArray(new Predicate[0]));

        return entityManager.createQuery(query)
                .getResultList();
    }
```

Fonte: Elaborado pelo autor, 2018.

6.5 Benefícios

A criação do *microservice any-address* utilizando o *framework* Overjax, propiciou alguns benefícios, dentre eles temos:

- **Configuração do arquivo pom.xml reduzida**. Com apenas duas dependências foi possível utilizar o Keycloak, como camada de segurança, o JPA, integrado com o banco de dados PostgreSQL e outras especificações do Java como EJB, *Bean Validations* e JAX-RS;
- **Padronização do formato de resposta das requisições**. Com o uso da annotation @WrapResponse foi possível padronizar o formato de resposta dos *endpoints*, permitindo integração de outros benefícios do Overjax, como tratamento de exceções, envio de metadados globais e criação de mensagens dinâmicas internacionalizadas;
- **Keycloak como camada de segurança**. Utilizando as annotations @Autenticated foi possível definir quais endpoints exigirão autenticação. Com @Authorize foi definido a regra de autorização de cada *endpoint*. O uso dessas annotations não está limitado ao Keycloak;
- **Configuração externalizada independente de fonte de configuração**. Toda configuração do *framework* foi definida utilizando Eclipse Microprofile Config, com

isso é possível definir as configurações em variáveis de ambiente, arquivos de propriedades, diretórios ou qualquer outra fonte de configuração customizada;

- **Microservice stateless**. Devido o uso de JWT e configuração externalizada esse *microservice* pode ser facilmente hospedado em um ambiente de *cloud*, pois não armazena estado e possui flexibilidade para alterar as configurações com base no ambiente, sem a necessidade de alterar o código;
- **Redução de tempo e custo de desenvolvimento**. Devido os recursos do Overjax foi possível concentrar os esforços no domínio do *microservice*, aumentando assim a produtividade;
- **Redução de tempo e custo de manutenção**. O Overjax possui diversos recursos que são necessários para diversas aplicações. Utiliza-los propicia o reuso, o que evita a necessidade de re-implementação e possívelmente a realização de alguma correção, com isso os esforços de manutenção são voltados apenas para o dominio do *microservice*.

7 CONSIDERAÇÕES FINAIS

Neste trabalho foram realizados estudos sobre as especificações da linguagem Java, como CDI, JAX-RS, Java EL e JCache. Em conjunto a isso, houve pesquisa também sobre *microservices* para entender as problemáticas e benefícios dessa arquitetura, e como as especificações do projeto Eclipse Microprofile, ajudam a resolver problemas enfrentados nesse modelo arquitetural voltado para nuvem.

Esses estudos resultaram no conhecimento necessário para o desenvolvimento do *framework* Overjax. Essa solução foi inspirada nas problemáticas enfrentadas no desenvolvimento do FormaSUS, que foi realizado na Secretaria de Saúde do Estado de Pernambuco. Seu objetivo é centralizar as principais configurações iniciais de um *microservice*, e assim reduzir tempo e custo.

Esse *framework* centraliza diversos recursos que são aplicáveis à uma gama de aplicações. Utilizar esses recursos causa uma redução no tempo e custo de desenvolvimento e manutenção, pois com o reuso desses recursos os esforços são concentrados no domínio da aplicação.

Overjax é um *framework* que ainda tem muito a crescer, mas desde já, ele atende a um conjunto de necessidades recorrentes em grande parte dos projetos de pequeno e grande porte. Ele foi desenvolvido no intuito de ser utilizado em ambiente escalável, e mesmo que ele ainda não esteja pronto para isso, o Overjax possui um grande potencial.

Esse trabalho é totalmente aberto para comunidade (*open source*). Visando o crescimento desse projeto, ainda existem melhorias a serem feitas. Por exemplo, a criação de documentação, testes automatizados e a implantação em um ambiente de integração contínua, como o GitLab CI.

Outras funcionalidades podem ser adicionadas a ele, como a criação de um *handler* que sanitize os parâmetros das requisições, gerador de *hyperlinks* dinâmicos para uso do padrão HATEOAS, mecanismo de padronização de consultas baseado na especificação JSON API, entre outros.

Todo o código fonte do projeto Overjax e da aplicação de teste pode ser encontrado por meio dos links ***overjax.seudev.com***, ***any-address.seudev.com***, respectivamente.

REFERÊNCIAS

APACHE SOFTWARE FOUNDATION. **Apache Maven project**: introduction. 2018. Disponível em: <https://maven.apache.org/what-is-maven.html>. Acesso em: 26 nov. 2018.

APACHE SOFTWARE FOUNDATION. **Apache Maven project**: guide to working with multiple modules. 2018. Disponível em: <https://maven.apache.org/guides/mini/guide-multiple-modules.html>. Acesso em: 27 nov. 2018.

APACHE SOFTWARE FOUNDATION. **Apache Maven project**: introduction to the dependency mechanism. 2018. Disponível em: <https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html>. Acesso em: 26 nov. 2018.

CHAN, R. **The 10 most popular programming languages, according to the Facebook for programmers**. 2019. Disponível em: <https://www.businessinsider.com/the-10-most-popular-programming-languages-according-to-github-2018-10>. Acesso em: 31 mar. 2019.

ECLIPSE. **Microprofile:** frequently asked questions. 2018. Disponível em: <https://microprofile.io/faq/>. Acesso em: 27 nov. 2018.

ECLIPSE. **Configuration for MicroProfile**. 2018. Disponível em: <https://microprofile.io/project/eclipse/microprofile-config>. Acesso em: 27 nov. 2018.

FARIA, T. **Java EE 7 com JSF, PrimeFaces e CDI. 1. ed.** AlgaWorks. 2013.

KLABNIK, S. *et al*. **Latest specification (v1.0)**. 2018. Disponível em: <https://jsonapi.org/format/>. Acesso em: 26 nov. 2018.

MENDES, R. D. **Programação Java com ênfase em orientação a objetos. 1. ed.** São Paulo: Novatec. 2009.

ORACLE. **Expression language specification**: version 3.0 final release. 2013. Disponível em: <https://download.oracle.com/otn-pub/jcp/el-3_0-fr-eval-spec/EL3.0.FR.pdf>. Acesso em: 26 nov. 2018.

ORACLE. **JAX-RS:** Java™ API for RESTful Web services. 2017. Disponível em: <https://download.oracle.com/otn-pub/jcp/jaxrs-2_1-final-eval-spec/jaxrs-2_1-final-spec.pdf>. Acesso em: 26 nov. 2018.

ORACLE. **JSR 365 contexts and dependency injection for Java 2.0**. 2017. Disponível em: <https://download.oracle.com/otn-pub/jcp/cdi-2-final-spec/cdi-spec.pdf>. Acesso em: 26 nov. 2018.

ORACLE. **JSON-B:** Java™ API for JSON binding version 1.0. 2017. Disponível em: <https://download.oracle.com/otn-pub/jcp/json_b-1-final-eval-spec/jsonb_spec_1.0.pdf>. Acesso em: 26 nov. 2018.

ORACLE. **Java API for JSON processing**. 2018. Disponível em: <https://javaee.github.io/jsonp/>. Acesso em: 26/11/2018.

REDHAT. **Keycloak**. 2018. Disponível em: <https://www.keycloak.org/about.html>. Acesso em: 26 nov. 2018.

RICHARDSON, C. **What are microservices?**. 2018. Disponível em: <https://microservices.io/>. Acesso em: 15 nov. 2018.

GLOSSÁRIO

Annotation	Representado pela inicial @, é um recurso da Plataforma Java
Build	Processo de construção/compilação de uma aplicação
Deserialização	É o processo de conversão de um fluxo de *bytes* em um objeto
ECMAScript	Linguagem de programação baseada em scripts
Framework	Abstração que une códigos comuns entre vários projetos de software provendo uma funcionalidade genérica
Login	Termo em inglês que significa ter acesso a uma conta de um sistema
Logout	Termo em inglês que significa terminar uma sessão de acesso a uma conta de um sistema
Multithread	Capacidade que o sistema operacional possui de executar várias threads simultaneamente
OAuth	Protocolo de autorização para APIs *web*
OpenID Connect	Camada de autenticação sobre o OAuth
RFC 7159	The JavaScript *Object Notation* (JSON) *Data Interchange Format*
Rollback	*Reversão de uma operação*
Serialização	É o processo de conversão de um objeto em um fluxo de *bytes*
Streaming	Transmissão contínua de distribuir informação
Javascript	Linguagem de programação interpretada
Web services	Serviço web que fornece alguma funcionalidade através da *web*