PROJETO E DESENVOLVIMENTO DE UMA API PARA O GERENCIAMENTO DE SERVIÇOS DE UM PROFISSIONAL FREELANCER

PATRÍCIO, Diego Juliano*
SOUSA, Marcelo Fernandes de**

*Graduando, Sistemas para Internet
**Doutor, Ciência da Computação

RESUMO

Em razão das dificuldades enfrentadas por profissionais *freelancers* em gerenciar suas prestações de serviços, este trabalho visa apresentar o projeto de implementação e desenvolvimento de um sistema de gerenciamento de serviços que tem como ideia principal auxiliar as tarefas, pagamentos e comunicação do profissional com o cliente. A API foi desenvolvida utilizando a linguagem de programação Java, Framework Spring, Banco de Dados MySQL e H2, dentre outras tecnologias. No mercado atual existem diversas aplicações para profissionais freelancers. Todavia, estes serviços oferecem uma competitividade muito alta para os profissionais, além de que são cobradas taxas pela prestação de serviço e há um prazo longo para o recebimento do pagamento. A aplicação apresentada neste trabalho permite um ambiente mais focado para os serviços de um único profissional, dando-lhe um melhor controle das tarefas e oferecendo um ambiente de fidelização para o cliente.

Palavras-chave: API, Freelancer, Sistema de Gerenciamento de Serviços.

1. INTRODUÇÃO

O gerenciamento de serviços é um dos problemas mais comuns quando nos deparamos com serviços *freelancer*. Muitos profissionais acabam utilizando os serviços de *e-mails* para realizar o controle das demandas solicitadas, o que ocasiona perda de informação, pois são utilizados tanto no uso pessoal quanto no profissional. Existem no mercado sistemas online que visam cobrir este cenário, porém acabam se tornando uma vitrine com inúmeros profissionais concorrendo entre si. Por conseguinte, a proposta deste projeto é uma solução possível e viável, para suprir a carência dos *freelancers* de ter um ambiente focado apenas nos seus trabalhos, no seu portfólio, permitindo aos mesmos o gerenciamento das prestações de serviços, otimizando assim o atendimento de seus clientes.

A necessidade de aprimorar o cumprimento das tarefas do dia a dia e organizá-las em um ambiente seguro fez surgir o crescimento das aplicações *web*. Esta informatização das tarefas pode ser realizada de maneira *online*, utilizando apenas o acesso à internet e o computador. Segundo Sommerville (2011), antes da web os sistemas de *software* eram executados em computadores locais e acessíveis apenas dentro de uma organização. Com a evolução da internet, mais recursos foram incorporados aos *Browsers, o* que permitiu o acesso do sistema através de um navegador *web*.

O presente projeto visa desenvolver um sistema de gerenciamento de ordem de serviços para *freelancers*. O sistema é uma API desenvolvida na linguagem de programação JAVA, utilizando a arquitetura de micros serviços, padrões de projetos DDD, *Framework Spring*, JPA Hibernate, Maven, H2 e *MySQL* para o banco de dados, que permite o cadastro de clientes, serviços, suas categorias e a emissão da ordem de serviços que serão prestadas, assim com a situação de seu pagamento.

O sistema possibilitará o gerenciamento dos serviços solicitados centralizando em um único ambiente todos os detalhes necessários, assegurando o armazenamento correto das informações, evitando falhas na comunicação, trazendo uma melhor acessibilidade ao freelancer pois serão acessados via navegador em qualquer lugar.

2. FUNDAMENTAÇÃO TEÓRICA

O objetivo deste trabalho é apresentar uma solução que permita suprir as necessidades dos profissionais freelancers em ter um ambiente próprio para apresentar seus serviços e gerir seus clientes e suas tarefas.

No atual cenário existe no mercado diversas aplicações para profissionais freelancers. Todavia, estes serviços como: Workana, 99Freelas, GetNinjas, dentre muitos outros. São plataformas nas quais os profissionais competem uns com os outros, além de que são cobradas taxas pela prestação de serviço e há um prazo longo para o recebimento do pagamento.

A identificação do problema surgiu a partir de conversas informais com os mais diversos freelancers de vários meios de atuação. Para completar, utilizou-se da experiência profissional do autor, que atua como freelancer no ramo de marketing digital e web por cerca de 7 anos e compreende todas essas dificuldades apresentadas, o que também serviu como motivação para a criação da aplicação.

Para o desenvolvimento deste projeto foram realizados estudos referentes à diversas tecnologias e ferramentas, além de conceitos que envolvem o processo de desenvolvimento de um software.

2.1 Programação Orientada à Objetos

O desenvolvimento deste projeto aplica o paradigma da Programação Orientada a Objetos (POO), uma forma abstrata de representar o mundo real. Um outro tipo de paradigma é a programação estruturada, esta foca nas ações. Já a POO fita seus esforços no objeto, na entidade (DEITEL, 2011).

Neste paradigma são definidos alguns conceitos importantes a quais serão apresentados a seguir:

- Classes: São coleções de objetos semelhantes, determinados por um grupo de atributos e ações (métodos) de natureza correlata (PUGA, RISSETTI, 2004). Como por exemplo, brigadeiro, trufa e brownie ambos podem ser considerados classes herdadas de doces, porém diferenciam-se em alguns atributos como recheio, tamanho, tipo do chocolate, podendo ser classificado como parte de uma classe superior.
- Objeto: São representações, no domínio da solução, de algum conceito abstrato ou concreto que suporta um incidente (evento ou ocorrência) ou uma interação (transação ou contrato) (PUGA, RISSETTI, 2004).
- Herança: Este é um conceito de reutilizar a classe superclasse ou classe pai herdando seus métodos e atributos. Normalmente, uma classe tem derivações quando, além das características (métodos e atributos) da superclasse, a subclasse possui especificações próprias (DEITEL, 2011).
- Polimorfismo: Este possibilita que um objeto admita o comportamento divergente do que foi estabelecido em sua classe, assim dizendo, a habilidade que o objeto tem de adotar várias formas. Este conceito está relacionado ao de herança (PUGA, RISSETTI, 2004).
- Encapsulamento: Também conhecido como ocultamento de informações, corresponde em esconder detalhes internos de uma classe a um objeto externo. O encapsulamento impede que um programa se torne tão interdependente que uma pequena modificação possa provocar grandes efeitos que se propaguem por todo o sistema (PUGA, RISSETTI, 2004).

- Métodos: São ações que executam tarefas. Eles podem ou não retornar valores e podem ou não receber parâmetros. Regularmente, uma classe possui diversos métodos, que no caso da classe “Atleta” poderiam ser “correr, malhar e descansar" (COAD e YOURDON, 1992).
- Atributos: São as características de um objeto, em outras palavras, a estrutura de dados que vão representar a classe. Os atributos de um objeto representam seu estado, ou seja, o valor de seus atributos em um determinado momento (COAD e YOURDON, 1992).

2.2 Arquitetura de Software

O uso de padrões de arquitetura foi evidenciado através de um livro publicado por Gamma, em 1995, na qual tinha o propósito de proporcionar padrões de projeto para linguagens orientadas a objetos que estavam recém empregadas no mercado (SOMMERVILLE, 2011). Este trabalho está direcionado no padrão *Domain Driven Design* (DDD) devido a sua estrutura de distribuição de código, conforme imagem 1.

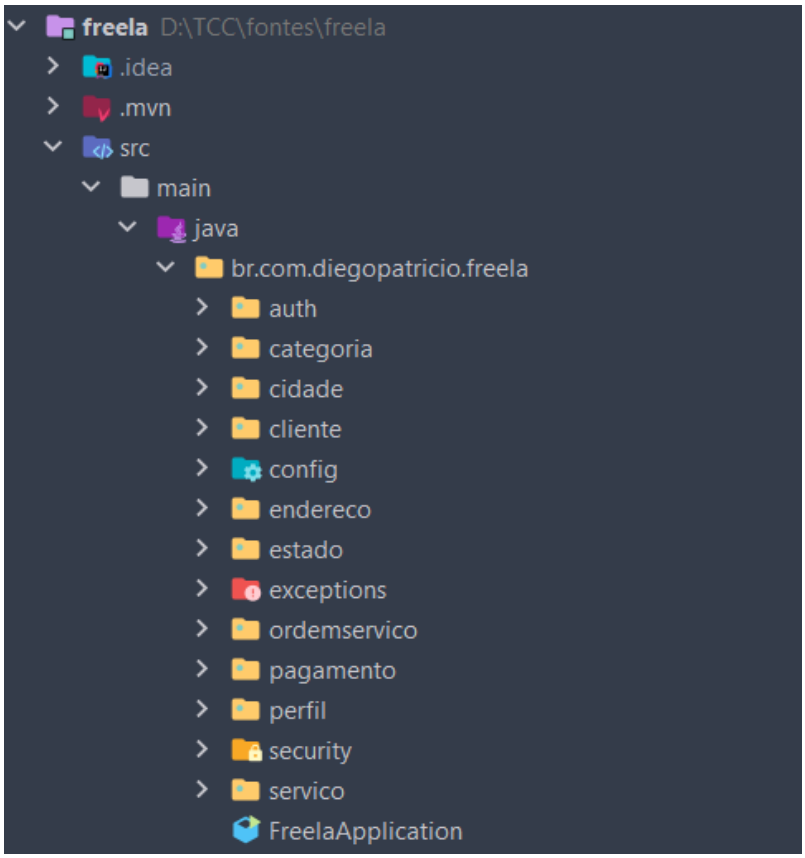


Imagem 1: Estrutura de pacotes do sistema

Fonte: Autor (2022).

O padrão DDD é uma técnica de estruturar a aplicação dirigida no que é de fato mais importante, o domínio do negócio (HAYWOOD, 2009). Estimulando os desenvolvedores a estarem presentes no entendimento do negócio para que construam uma linguagem ubíqua (universal) que exprima o conhecimento do time de negócio para um modelo em domínio (para o código).

Também neste projeto foi aplicado o padrão de arquitetura MVC que consiste em desacoplar e facilitar a troca de informações entre a GUI (*Graphical User Interface*) e a manipulação dos dados fazendo com que as respostas sejam mais rápidas e dinâmicas, conforme apresentado na imagem 2.

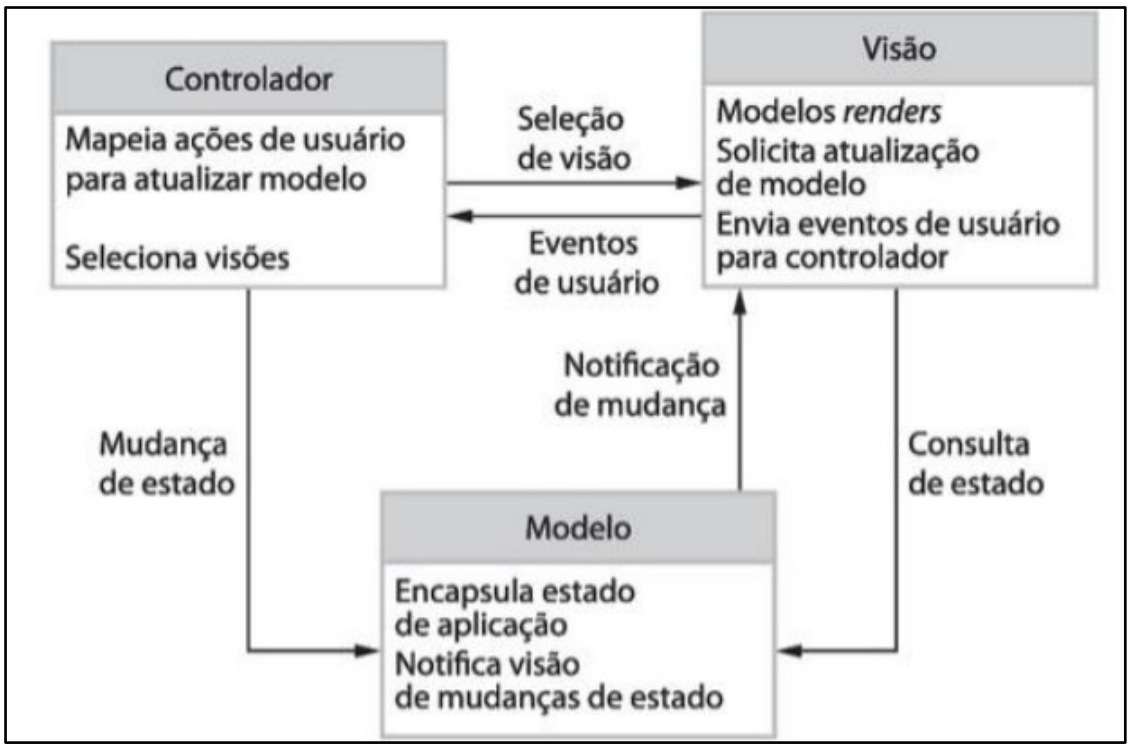


Imagem 2: Diagrama de arquitetura MCV

Fonte: Sommerville (2011).

Esta divisão é realizada por meio de três componentes com uma atuação direta entre eles: *view, model* e *controller* (SOMMERVILLE, 2011).

2.3 Framework Spring

O Spring proporciona recursos para aplicativos baseados em Java para qualquer tipo de plataforma. Dispõe de diversas tecnologias, que simplificam o desenvolvimento de código, permitindo que os desenvolvedores se concentrem na construção da regra de negócio. O *framework* é organizado em módulos que podem

ser agrupados em seus principais recursos em *Core Container*, *Data Access/Integration, Web, AOP (Aspect Oriented Programming)*, Instrumentação e Teste, conforme apresentado na imagem 3.

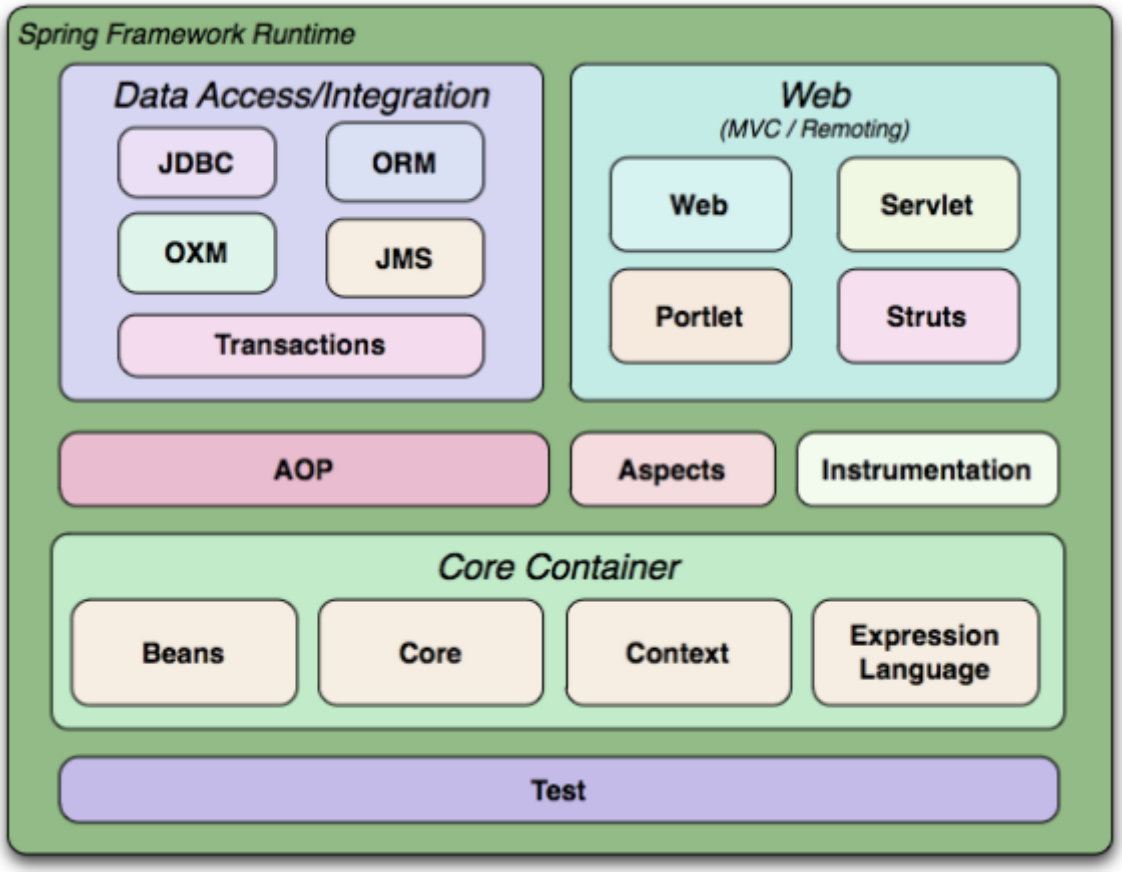


Imagem 3: Diagrama de Organização Spring
Fonte: Pivotal Software (2018).

O objetivo do projeto apresentado é a construção de uma aplicação *Web* utilizando o módulo *Web MVC* na qual é estabelecido métodos que recebem requisições *HTTP* e retornam respostas em *JSON* através de *API RESTful*. Fazendo uso, também do *Spring Data* para fazer a persistência dos dados no *MySQL*, na qual oferece suporte à *Java Database Connectivity* (JDBC) e *Object Relational Mapping* (ORM) com *Hibernate* e *Java Persistence API* (JPA).

Outra etapa bastante importante do projeto na qual o *spring* irá auxiliar será no desenvolvimento de testes unitários para a verificação das funcionalidades de acordo com os requisitos especificados com o *Spring Test*. O *framework* permite a utilização da biblioteca *JUnit* e objetos *mocks*, que são implementações de abstrações de contexto e ambiente de execução, inclusive com a injeção de dependência.

Para algumas operações, conforme definido pela regra de negócio, foi requerido que o usuário da requisição apresentasse um selo de autorização, visto que se trataria de dados sensíveis. Para tanto, a aplicação precisaria verificar o usuário e restringir ou não o seu acesso. Como solução foi utilizado o *Spring Security* com o padrão *JSON Web Token* (JWT) que realiza a autenticação e autorização na API.

A JWT é um padrão definido pelo RFC 7519 para transmitir e armazenar de forma compacta e segura objetos JSON entre aplicações. A seguir é apresentado, na imagem 4, a utilização da implementação realizada.

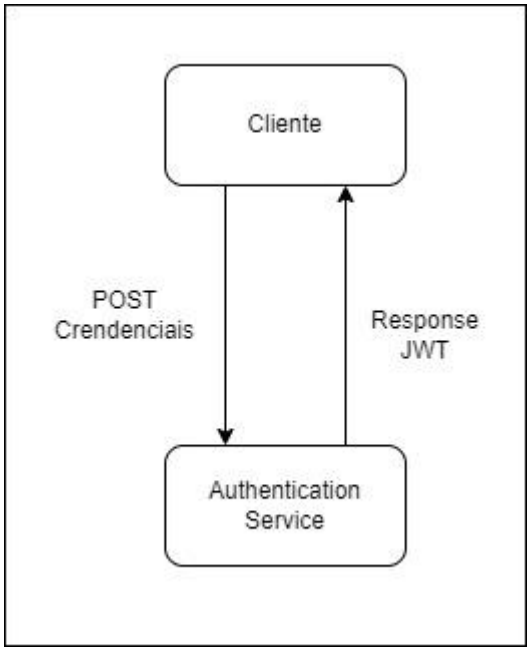


Imagem 4: Diagrama de fluxo JWT

Fonte: Autor (2022)

Os JSON *Web Tokens* consistem em três camadas separadas por pontos (xxxxx.yyyyy.zzzzz) divididas da seguinte maneira. A primeira parte é o *Header* que consiste no tipo do token e o tipo do algoritmo que está sendo usado, como HMAC SHA256 ou RSA. A segunda parte é o *Payload*, que contém as declarações da entidade tratada. E por último a *Signature* que verifica se a mensagem não foi alterada ao longo do percurso e, no caso de *tokens* assinados com chave privada, também verificar a sua veracidade.

3. METODOLOGIA

A metodologia de desenvolvimento adotada para este trabalho é Análise Orientada a Objeto, por tanto a primeira etapa deste trabalho foi a realização do

diagrama de classe realizada na linguagem UML (Linguagem de Modelagem Unificada), uma linguagem padrão orientada à objetos que auxilia na visualização, construção, especificação e documentação do projeto de software.

3.1 Diagrama de classes

Um diagrama de classes detalha os tipos de objetos da aplicação e os relacionamentos que existem entre eles. As classes representam os objetos centrais em um sistema e é apresentada como um retângulo com 3 compartimentos. O primeiro refere-se ao nome da classe, a do meio os atributos e a inferior o comportamento da classe.

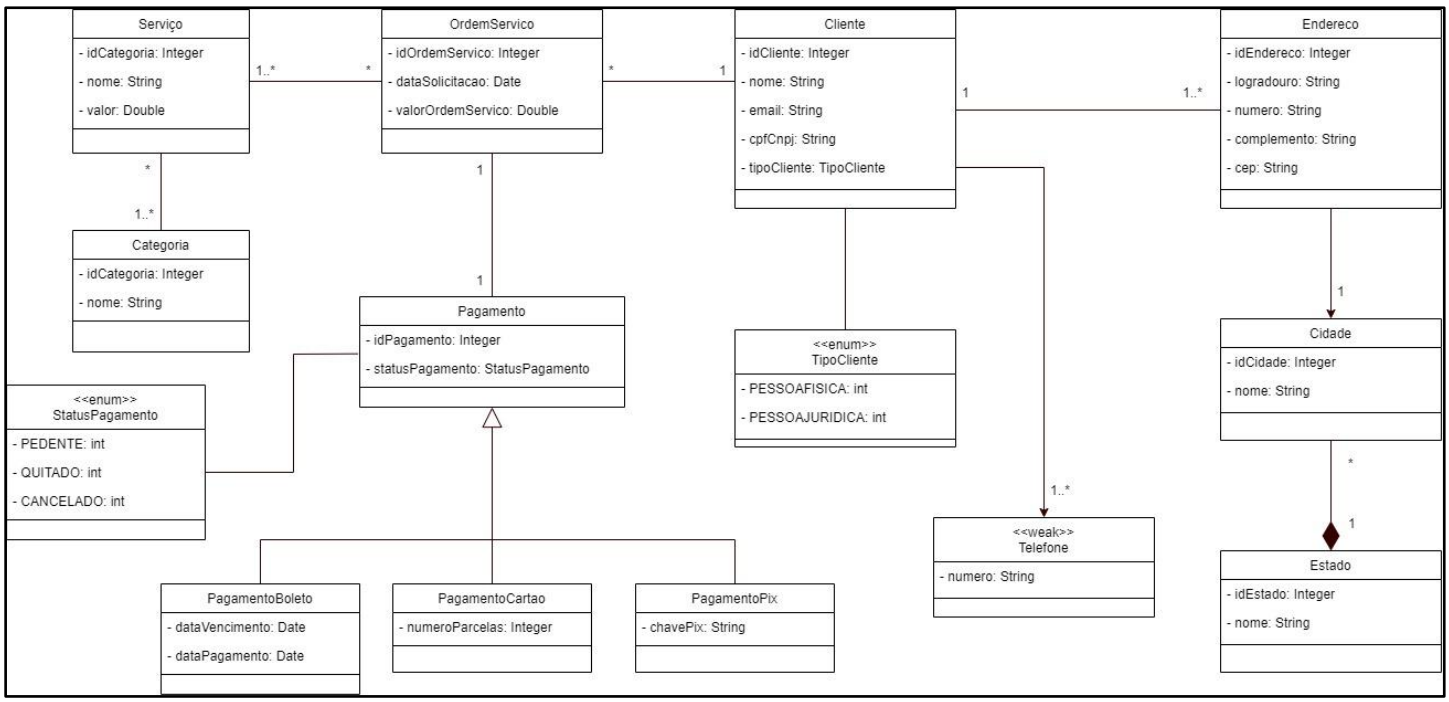


Imagem 5: Diagrama de Classes da API FREELA

Fonte: Autor (2022)

Na imagem 5, apresentada acima, está representado todo o fluxo de relacionamento que as entidades exercem na aplicação, assim como suas associações.

3.2 Requisitos Funcionais

O Documento de Definição de Requisitos (DDR) tem a finalidade de apresentar o conteúdo dos requisitos funcionais e as regras que serão aplicadas às funcionalidades propostas, assim como seus relacionamentos e dependências. Os

requisitos funcionais foram identificados, com base na regra de negócio do sistema de gerenciamento de serviços, para as funcionalidades apresentadas na tabela 1 abaixo.

É considerado como requisito funcional as obrigações da aplicação. Há possibilidade de variações conforme o tipo de produto a ser desenvolvido e quem será a sua audiência. Quando direcionado a usuários, a aplicação deve ter uma abstração maior para que seja possível o entendimento do usuário, e quando escritos para o sistema, devem contemplar todas entradas e saídas possíveis, incluindo suas exceções (SOMMERVILLE, 2011).

ID	Requisito Funcional	Descrição
[RF01]	Manter Serviço	O sistema permite incluir, alterar, listar e excluir informações de serviços como: nome, valor, categoria.
[RF02]	Manter Categoria	O sistema permite incluir, alterar, listar e excluir categorias.
[RF03]	Manter Cliente	O sistema permite incluir, alterar, listar e excluir informações de clientes como: nome, email, cpf/cnpj, endereço, telefone.
[RF04]	Manter Ordem de Serviço	O sistema permite incluir, atualizar, listar e excluir ordem de pagamento.

Tabela 1: Tabela de requisitos funcionais

Fonte: Autor (2022)

3.3 Requisitos Não Funcionais

Os requisitos não-funcionais são aqueles que não se relacionam diretamente com as funções do sistema, no entanto, fazem referências a conceitos como confiabilidade, tempo de resposta, segurança, dentre outras restrições impostas aos serviços ofertados pelo sistema (SOMMERVILLE, 2011).

ID	Requisitos Não Funcionais
[RNF01]	A API deverá ser desenvolvida em JAVA utilizando o framework *Spring Boot*.
[RNF02]	A aplicação deve seguir os princípios de arquitetura de micro serviço para garantir eficiências nas manutenções futuras.
[RNF03]	Deve-se utilizar o Spring Security para fornecer proteção no armazenamento de senhas e transferências de dados.

Tabela 2: Tabela de requisitos não funcionais

Fonte: Autor (2022)

Conforme apresentado na tabela 2, estão descritos os requisitos não funcionais levantados para a API FREELA.

3.4 Caso de Uso

O intuito principal de um diagrama de caso de uso é descrever as interações de um usuário com o sistema. A personificação de usuários e funcionalidades são realizadas através de distintos papéis (PRESSMAN, 2016).

Considerando os requisitos descritos anteriormente, é possível modelar as funcionalidades do sistema proposto. Para isso, inicialmente, é apresentado na tabela 3 o caso de uso do sistema.

Atores	Clientes
Pré Condições	Cliente Cadastrado
Visão Geral	Este caso de uso consiste no processo de contratação de serviços e fechamento da OS (ordem de serviço) por parte do cliente.
Cenário Principal de Sucesso	
1. [OUT] O sistema informa os nomes de todas categorias. 2. [IN] O cliente seleciona a categoria e informa o serviço desejado. 3. [OUT] O sistema informa nome e preço dos serviços que se enquadram na pesquisa. 4. [IN] O cliente seleciona um serviço para adicionar a ordem se serviço. 5. [OUT] O sistema exibe a OS com as informações da OS são: nome, serviço contratado, forma de pagamento, status do pagamento e valor total da ordem de serviço. 6. [IN] O cliente informa que deseja finalizar a ordem de serviço.	

Cenários Alternativos
5. Pagamento Variante 5.1.: Pagamento com boleto [IN] O cliente informa que deseja pagar com boleto. Variante 5.2: Pagamento com cartão [IN] O cliente informa que deseja pagar com cartão e informar a quantidade de parcelas. Variante 5.3: Pagamento com Pix [IN] O cliente informa que deseja pagar com pix e informar a chave para pagamento.

Tabela 3: Tabela de caso de uso

Fonte: Autor (2022)

4. DESENVOLVIMENTO

A estrutura da aplicação foi desenhada baseada nos requisitos e casos de uso levantados, como apresentado na documentação em *swagger* (apêndice A), no qual permitiu um maior entendimento do objetivo do negócio e alinhamento entre tecnologias a serem utilizadas. Com um *backend* desacoplado do *frontend* torna possível adicionar qualquer *cliente* sem a necessidade de manutenção ou adição de código, facilitando também a integração entre interfaces.

A aplicação FREELA é uma API *Web RESTful* desenvolvida em Java, com o *framework Spring*, utilizando o paradigma orientado a objetos. O projeto dispõe da ferramenta *Maven* para o gerenciamento de dependências (conforme apresentado no apêndice B) e utiliza para implementar a API *Servlet* de Java o servidor *Tomcat*. As classes controladoras recebem as requisições do *frontend/cliente* que processam a lógica de negócio nas classes de serviços que acessam as classes de repositório, as quais podem manipular o banco de dados, conforme imagem 6.

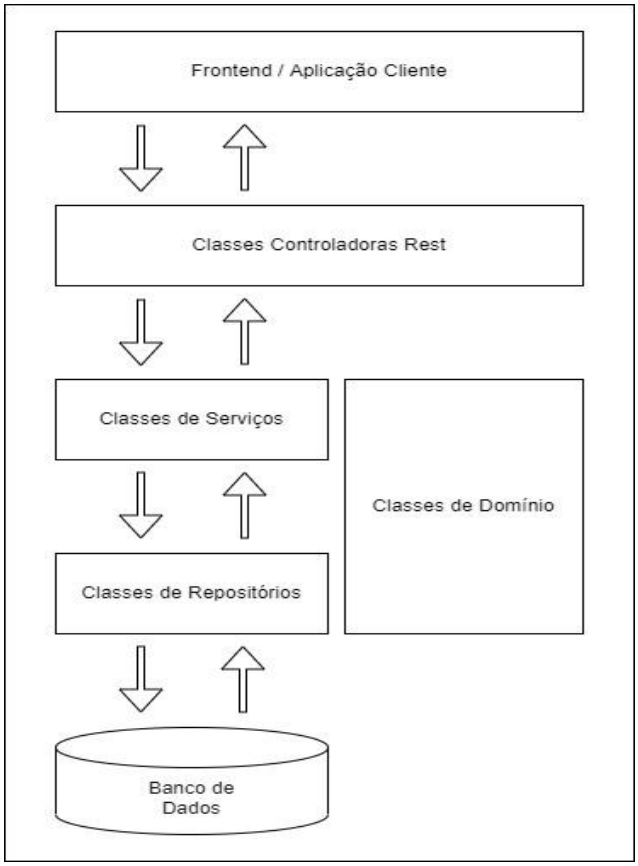


Imagem 6: Diagrama de Aplicação REST

Fonte: Autor (2022)

Neste projeto está sendo utilizado o banco de dados MySQL em ambiente de produção e o H2 em ambiente de teste, sendo o acesso a essas bases, gerenciado pelo *Spring Data* e as consultas executadas com o *Hibernate*.

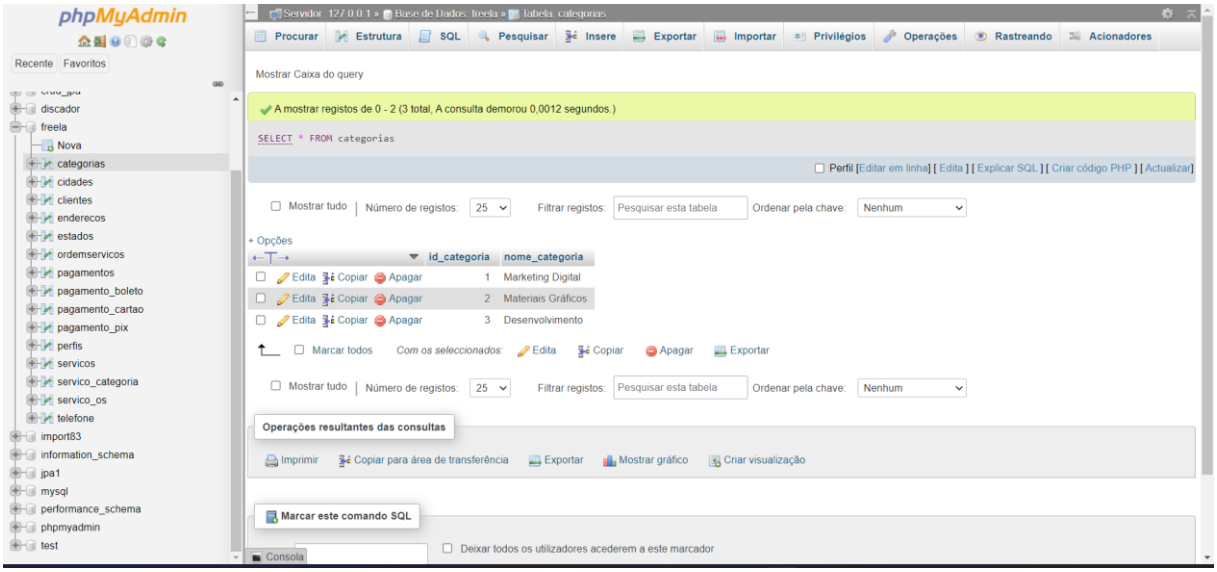


Imagem 7: Banco de Dados MySQL

Fonte: Autor (2022)

A API é um serviço REST que transfere dados no formato JSON. A comunicação é feita através do protocolo HTTPS através dos métodos GET que busca uma informação na base de dados, POST que insere uma nova informação na base de dados, PUT que faz à atualização em alguma informação já existente na base de dados e DELETE que exclui um registro na base de dados.

Na API foi desenvolvido o CRUD (*Create, Read, Update, Delete*), um acrônimo para as maneiras de se operar em informação armazenada, de categorias, clientes, serviços e ordem de serviço conforme à regra de negócio estabelecida.

Para realizar o armazenamento (*Create*) as requisições são passadas pelo método POST. O *Spring* auxilia neste processo, pois basta apenas utilizar a anotação *@PostMapping* em cima do método desejado e determinar um corpo na requisição através da anotação *@RequestBoy* seguido do tipo do objeto pretendido no parâmetro do método, conforme na imagem 8.

```
@PostMapping
public ResponseEntity<Void> cadastrarCategoria(@Valid @RequestBody CategoriaDTO categoriaDTO){
    Categoria categoria = service.fromDTO(categoriaDTO);
    categoria = service.cadastrarCategoria(categoria);
    URI uri = ServletUriComponentsBuilder.fromCurrentRequestUri().path("/{id}")
            .buildAndExpand(categoria.getIdCategoria()).toUri();
    return  ResponseEntity.created(uri).build();
}
```

Imagem 8: *Response* do método cadastrar categoria

Fonte: Autor (2022)

Para poder resgatar a informação cadastrada é utilizada a anotação *@GetMapping* a qual pode ser atribuída o valor, geralmente a variável id, para buscar um objeto em específico, caso contrário, retornará uma lista com todos os objetos salvos como apresentado na imagem 9.

```java
@GetMapping
public ResponseEntity<List<CategoriaDTO>> listarCategorias(){
    List<Categoria> listaCategorias = service.listarCategorias();
    List<CategoriaDTO> listaDTO = listaCategorias.stream().map(CategoriaDTO::new).collect(Collectors.toList());
    return ResponseEntity.ok().body(listaDTO);
}

@GetMapping(value = "/{id}")
public ResponseEntity<Categoria> busarCategoria(@PathVariable Integer id){
    Categoria categoria = service.buscarCategoria(id);
    return ResponseEntity.ok().body(categoria);
}
```

Imagem 9: *Response* do método listar e buscar categoria

Fonte: Autor (2022)

Já para poder atualizar o objeto cadastrado é usado o método PUT, *@PutMapping,* que também receberá um valor para identificar o objeto em questão. Neste método pode-se aplicar dois parâmetros: um identificador com a anotação *@PathVariable* e a *@RequestBody* para o conteúdo a qual deseja atualizar, seguindo o modelo da imagem 10.

```java
@PutMapping(value = "/{id}")
public ResponseEntity<Void> atualizarCategoria(@Valid @RequestBody CategoriaDTO categoriaDTO, @PathVariable Integer id){
    Categoria categoria = service.fromDTO(categoriaDTO);
    categoria.setIdCategoria(id);
    categoria = service.atualizarCategoria(categoria);
    return ResponseEntity.noContent().build();
}
```

Imagem 10: *Response* do método atualizar categoria

Fonte: Autor (2022)

Para o método DELETE, deve-se utilizar a anotação *@DeleteMapping*, atribuindo um valor que se refere ao identificador do objeto e informar este ID com a anotação *@PathVariable* para buscar o recurso alvo, seguindo o modelo da imagem 11.

```java
@DeleteMapping(value = "/{id}")
public ResponseEntity<Void> deletarCategoria(@PathVariable Integer id){
    service.deletarCategoria(id);
    return ResponseEntity.noContent().build();
}
```

Imagem 11: *Response* do método deletar categoria

Fonte: Autor (2022)

4.1 Executando a aplicação

Conforme regra de negócio estabelecida todo usuário que se cadastrar na API é configurado como cliente, mas é possível alterar seu perfil para administrador caso o profissional freelancer necessite inserir outra pessoa para gerenciar o sistema.

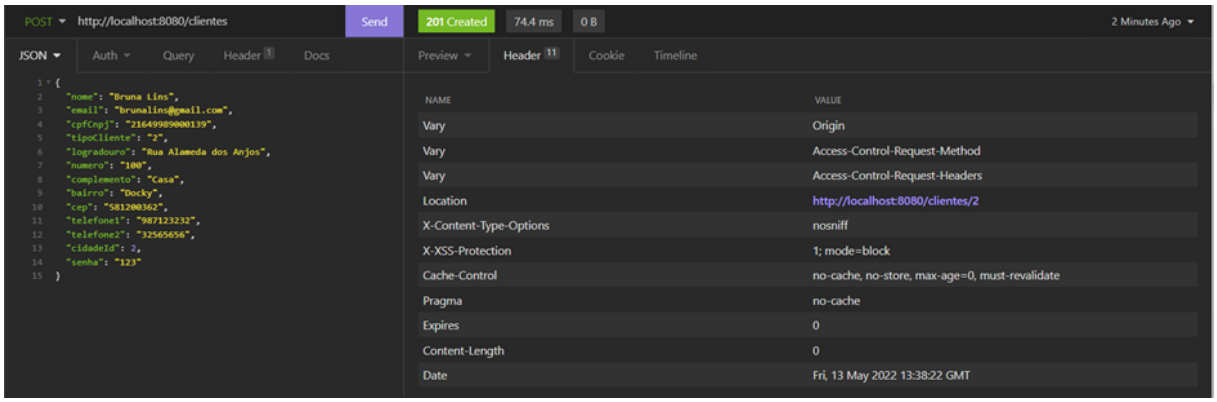


Imagem 12: Cadastrando um cliente

Fonte: Autor (2022)

Para cadastrar qualquer usuário pode se utilizar do *Frontend* preencher um formulário a qual será consumido pela API FREELA passando as informações no corpo da requisição e será retornado a *URI* com o usuário cadastrado, conforme imagem abaixo.

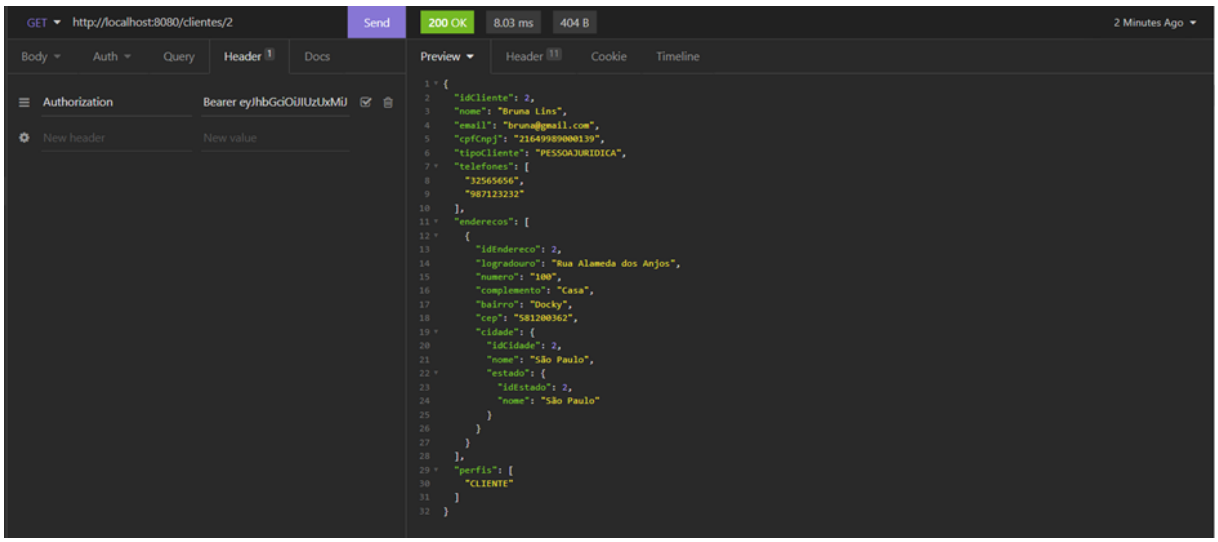


Imagem 13: Resgatando informações de um cliente

Fonte: Autor (2022)

Para atualizar e buscar um cliente é necessário passar o *ID*. Esta ação será exclusiva do perfil de administrador e do próprio cliente logado no sistema.

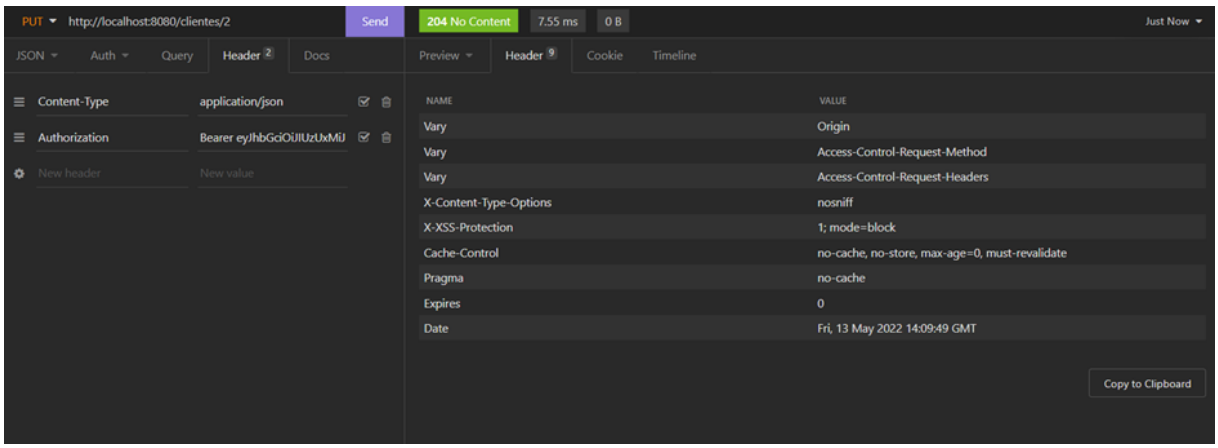


Imagem 14: Atualizando informações de um cliente

Fonte: Autor (2022)

Já para os métodos de listar e deletar cliente só poderá ser acessado pelo administrador. Para isto deve-se ser efetuado o *login* resgatar o *token* informado e passar como parâmetro no Header das requisições conforme apresentado abaixo.

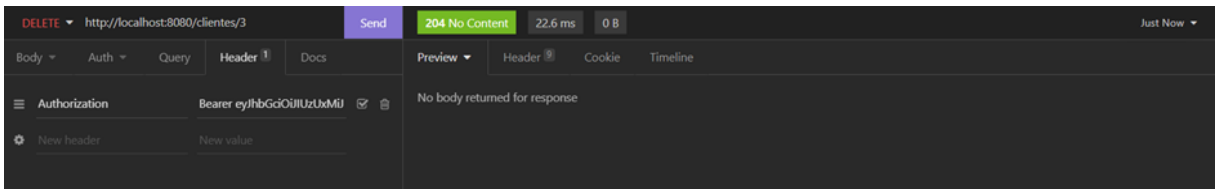


Imagem 15: Deletando um cliente

Fonte: Autor (2022)

A entidade Categoria e Serviços só poderão ser cadastradas, atualizadas e deletas apenas por um perfil administrador. Mas para os métodos *GET* e listar e buscar é aberta a todos que possuírem a *uri.*

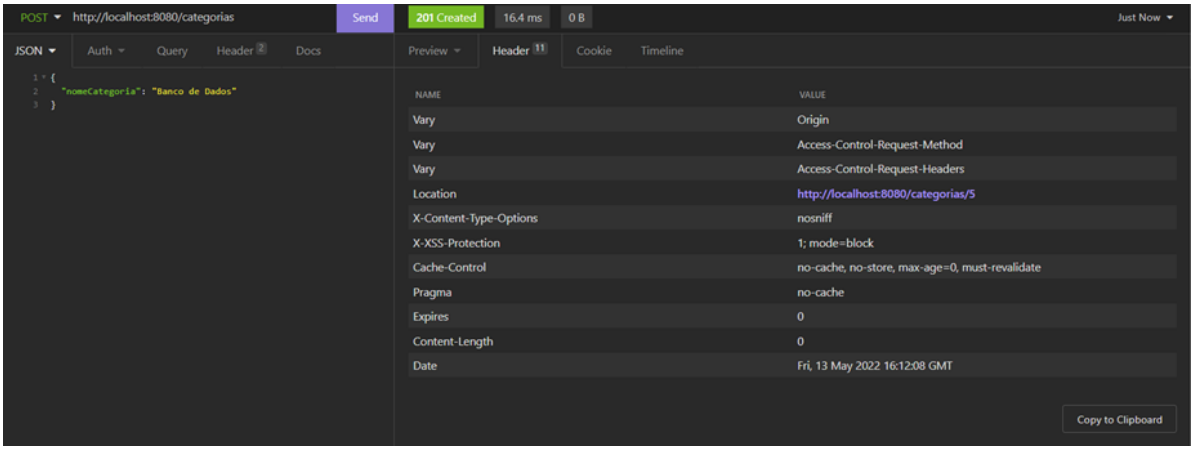


Imagem 16: Cadastrando uma Categoria

Fonte: Autor (2022)

Para a ordem de serviço a regra de negócio permite que apenas clientes logados, passando o *token* de acesso, possa registrar uma ordem de serviço, assim também, como resgatá-la. Para os métodos *PUT* que atualiza o status do pagamento e o *DELETE* que exclui uma ordem de serviço está reservado apenas para o administrador, assim como também método *GET* para listar todas as ordens de serviço. O cliente pode visualizar todas as suas OS através do *path ordemservicos/page* que lista uma paginação em ordem decrescente de todas as ordens de serviços por ele contratada.

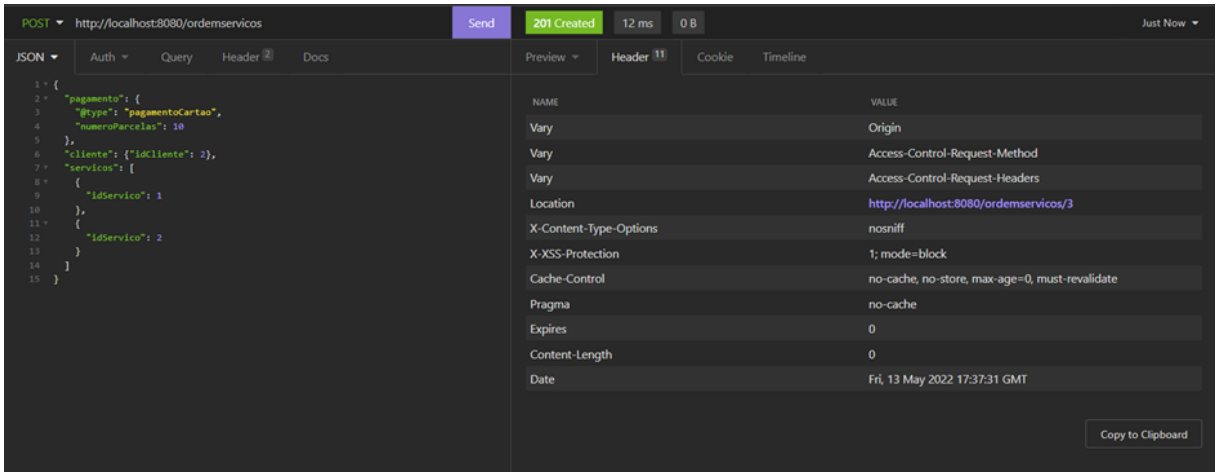


Imagem 17: Cadastrando Ordem de Serviço

Fonte: Autor (2022)

Toda a demonstração da execução de cada *path* da aplicação e a explicação das regras de negócios com suas respectivas tratativas de exceção foi realizada em vídeo e hospedada na plataforma de vídeo *YouTube* podendo ser acessada através

do link: *https://www.youtube.com/watch?v=TqnU4Nr2np4*. Assim sendo, é possível verificar a API em funcionamento.

5. CONSIDERAÇÕES FINAIS

Como muitos freelancers não possuem um sistema próprio para gerenciar suas prestações de serviços, este sistema visa atender este mercado carente, uma vez que configura uma necessidade existente. Uma forma de colaborar para que profissionais tenham um maior controle sobre as suas prestações de serviço, otimizando o processo de prestação de serviços e fidelização do cliente.

Foram apresentadas a análise de negócio, o diagrama de classe com suas entidades e seus relacionamentos, caso de uso e a definição de requisitos funcionais e não funcionais que estão inseridos na aplicação.

5.1 TRABALHOS FUTUROS

Este projeto teve como foco o desenvolvimento *Backend* de um sistema de gerenciamento de tarefas para freelancers. Posteriormente a API será hospedada no *Heroku,* uma plataforma em nuvem para fazer *deploy* de aplicações e será desenvolvido o *Frontend*, na qual consumirá a API desenvolvida para executar a regra de negócio e com isso obter o gerenciamento dos serviços.

REFERÊNCIAS

COAD, P., YURDON, E. Análise baseada em objetos. Rio de Janeiro, 1992

DEITEL, Paul; DEITEL, Harvey. Como programar. 6. ed. São Paulo: Pearson Education, 2011.

HAYWOOD, D. Domain-Driven Design Using Naked Objects. [S.l.]: Pragmatic Bookshelf, 2009.

JSON Web Token. Auth0, Inc, 2018. Disponível em: https://jwt.io/introduction/. Acesso em: 01/05/2022.

Módulos Spring Framework. Pivotal Software, Inc, 2018. Disponível em: https://docs.spring.io/spring/docs/3.0.0.M4/reference/html/ch01s02.html. Acesso em: 08/03/2022.

PRESSMAN, Roger; MAXIM, Bruce. Engenharia de Software. 8. ed. Porto Alegre: AMGH, 2016.

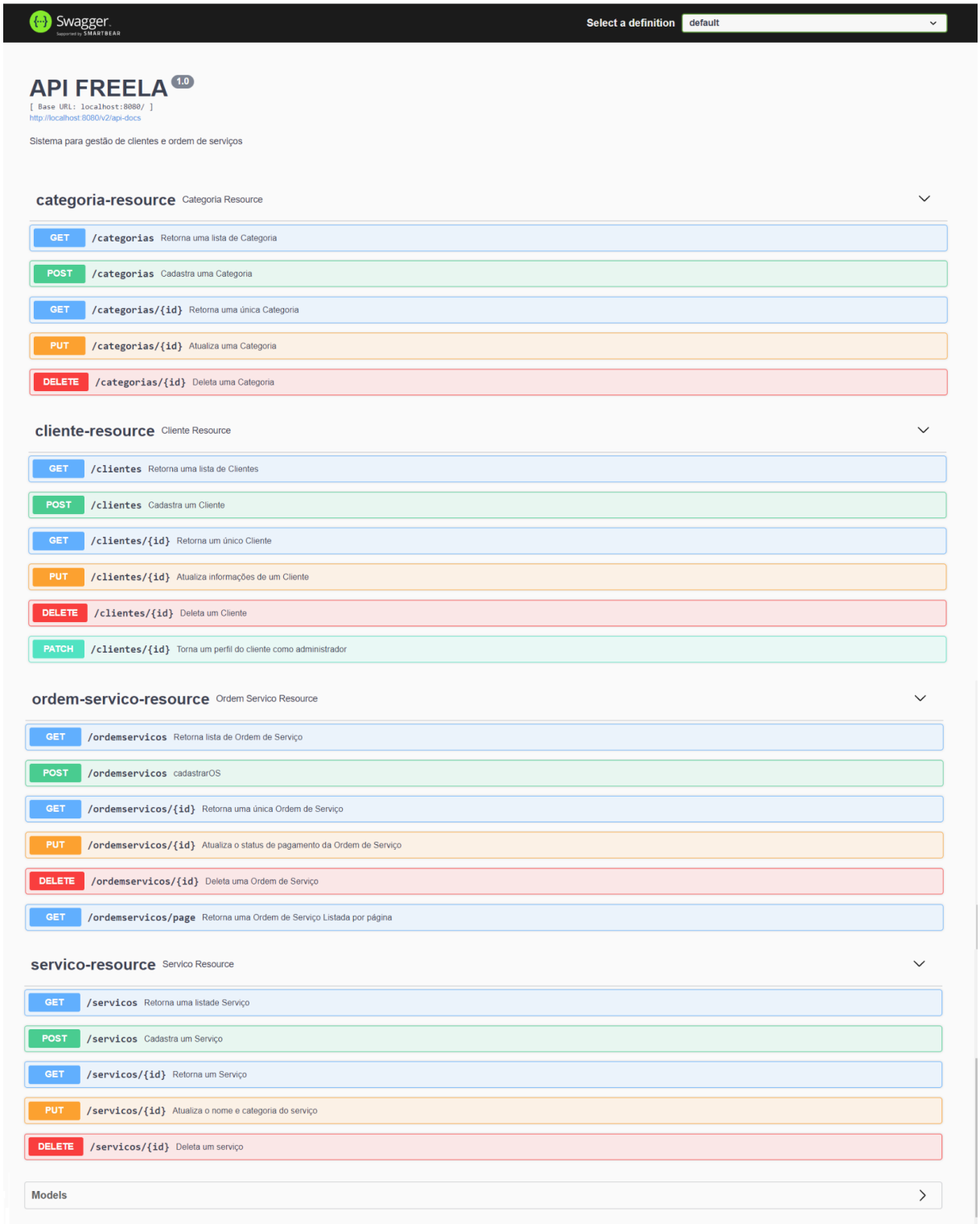
PUGA, Sandra; RISSETTI, Gerson. Lógica de programação e estrutura de dados com aplicações em Java. São Paulo: Pearson Education do Brasil, 2004.

SOMMERVILLE, Ian. Engenharia de Software. 10. ed. São Paulo : Pearson Prentice Hall, 2011.

Spring Framework. Pivotal Software, Inc, 2018. Disponível em: https://spring.io/. Acesso em: 08/03/2022.

APÊNDICE A - DOCUMENTAÇÃO DA API

A documentação da API FREELA encontra-se disponível em *swagger* do link: *http://localhost:8080/swagger-ui/index.html*.



APÊNDICE B - DEPENDÊNCIAS

As dependências utilizadas no desenvolvimento da aplicação se encontram detalhadas abaixo. O código apresentado abaixo faz parte do arquivo pom.xml gerenciado pelo maven.

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <version>1.18.22</version>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jdbc</artifactId>
    </dependency>
    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <scope>runtime</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-devtools</artifactId>
    </dependency>
```

```
        <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-validation</artifactId>
        </dependency>
        <dependency>
                <groupId>io.springfox</groupId>
                <artifactId>springfox-swagger2</artifactId>
                <version>3.0.0</version>
        </dependency>
        <dependency>
                <groupId>io.springfox</groupId>
                <artifactId>springfox-swagger-ui</artifactId>
                <version>3.0.0</version>
        </dependency>
        <dependency>
                <groupId>io.springfox</groupId>
                <artifactId>springfox-boot-starter</artifactId>
                <version>3.0.0</version>
        </dependency>
        <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-security</artifactId>
        </dependency>
        <dependency>
                <groupId>io.jsonwebtoken</groupId>
                <artifactId>jjwt</artifactId>
                <version>0.7.0</version>
        </dependency>
</dependencies>
```