FACULDADE DE TECNOLOGIA DE AMERICANA

Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas

Alfredo Scunderlick Neto

Sistema para gestão de projetos arquitetônicos

Americana, SP

2018

FACULDADE DE TECNOLOGIA DE AMERICANA

Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas

Alfredo Scunderlick Neto

Sistema para gestão de projetos arquitetônicos

Relatório técnico apresentado como exigência parcial para obtenção do título de Tecnólogo em Análise e Desenvolvimento de Sistemas pelo CEETEPS/Faculdade de Tecnologia – FATEC/ Americana.

Americana, SP

2018

Alfredo Scunderlick Neto

Sistema para gestão de projetos arquitetônicos

Relatório técnico apresentado como exigência parcial para obtenção do título de Tecnólogo em Análise eDesenvolvimento de Sistemas pelo CEETEPS/Faculdade de Tecnologia FATEC/ Americana.

Americana, 07 de Dezembro de 2018.

Banca Examinadora:

Evandro Santaclara

Especialista

Faculdade de Tecnologia de Americana - FATEC

Clerivaldo José Roccia

Mestre

Faculdade de Tecnologia de Americana - FATEC

Alberto Martins Junior

Mestre

Faculdade de Tecnologia de Americana - FATEC

FICHA CATALOGRÁFICA – Biblioteca Fatec Americana - CEETEPS
Dados Internacionais de Catalogação-na-fonte

S44s SCUNDERLICK NETO, Alfredo

Sistema para gestão de projetos arquitetônicos. / Alfredo Scunderlick Neto. – Americana, 2018.

73f.

Monografia (Curso de Tecnologia em Análise e Desenvolvimento de Sistemas) - - Faculdade de Tecnologia de Americana – Centro Estadual de Educação Tecnológica Paula Souza

Orientador: Prof. Esp. Evandro Santaclara

1 Desenvolvimento de software I. SANTACLARA, Evandro II. Centro Estadual de Educação Tecnológica Paula Souza – Faculdade de Tecnologia de Americana

CDU: 681.3.05

AGRADECIMENTOS

Aos meus pais, por todo o apoio e incentivo na minha vida profissional. Especialmente ao meu pai, que, como desenvolvedor, também me auxiliou muito na parte técnica.

A Gabriela De Nadai, arquiteta, que tornou este trabalho possível.

A todos os professores da instituição FATEC Americana, que fizeram parte desta jornada. Especialmente ao professor Evandro Santaclara pela dedicação, paciência e ensinamentos.

RESUMO

Este trabalho tem como objetivo mostrar as etapas do desenvolvimento de um sistema para o gerenciamento de projetos de arquitetura. Ideia que surgiu das necessidades apresentadas pela cliente Gabriela De Nadai e sua sócia. Para alcançar esse objetivo foram levantados os requisitos básicos para que fosse possível verificar a viabilidade da utilização e criação desse sistema. Diversas técnicas de engenharia de *software* foram utilizadas nesse processo, como a modelagem do banco de dados, a criação dos diagramas UML, a definição da metodologia de trabalho, a definição de um padrão de projeto, além de todas as ferramentas que se tornam necessárias para a realização desse trabalho, como as linguagens de programação, os *frameworks* e o sistema gerenciador de banco de dados. Com isso, foi possível concluir o sistema com todos os requisitos apresentados inicialmente pela cliente. Apesar disso, continuamos a trabalhar em novas funcionalidades.

Palavras Chave: Gerenciamento de projetos; arquitetura; engenharia de *software*.

ABSTRACT

This work aims to show the stages of the development of a system for the management of architecture projects. Idea that arose from the needs presented by the client Gabriela De Nadai and her partner. To achieve this objective, the basic requirements were raised so that it was possible to verify the feasibility of the use and creation of this system. Several software engineering techniques have been used in this process, such as database modeling, creation of UML diagrams, definition of the work methodology, definition of a design pattern, and all the tools that are necessary for the execution of this work, such as the programming languages, the frameworks and the database manager system. With this, it was possible to complete the system with all the requirements initially presented by the customer. Despite this, we continue to work on new features.

Keywords: *Projects Management; architecture; software engineering.*

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

1. INTRODUÇÃO

Este trabalho apresenta o desenvolvimento de um sistema para escritórios de arquitetura, motivado pelos problemas apresentados pela arquiteta Gabriela De Nadai. Seu objetivo é de melhorar o gerenciamento das etapas dos projetos, dos clientes e manter um controle das alterações realizadas pelos usuários do sistema.

A arquiteta pretende iniciar uma empresa própria junto a sua sócia, e após vivenciar dificuldades no escritório que trabalha atualmente, decidiu se preparar para ter um desempenho melhor no mercado de trabalho.

Segundo a arquiteta, na empresa em que trabalha hoje, os projetos realizados são gerenciados através de planilhas feitas com a ferramenta Excel, método que foi implementado pelos próprios funcionários, mas, ela afirma que nem todos eles a utilizam, ou seja, não existe um padrão para o armazenamento das informações dos projetos. Além disso, qualquer funcionário pode alterar essas planilhas, sem que haja qualquer tipo de registro dessas alterações. Já as informações dos clientes, são armazenadas em uma agenda escrita à mão, logo, é preciso ter essa agenda por perto caso queira alguma informação sobre um cliente específico.

Outro problema observado por ela é com relação aos cálculos de custo dos projetos. O valor de um projeto de arquitetura atualmente é calculado pelas horas de trabalho, dessa forma é bom ter pelo menos uma ideia de quanto tempo foi utilizado no processo de desenvolvimento de cada etapa do projeto, para evitar erros na hora de calcular o valor final. Na empresa em que trabalha atualmente esse custo é calculado por metro quadrado de projeto, um método antigo, pois como não existe uma forma eficiente para registrar a quantidade de horas trabalhadas, não é possível fazer um cálculo exato nos novos padrões de cobrança.

Com esses problemas em mente, o objetivo desse trabalho é desenvolver uma ferramenta mais completa e automática, capaz de facilitar a visualização de todo o processo de produção de projetos, junto ao gerenciamento dos clientes e ao rastreamento de alterações realizadas por usuários do sistema.

2. METODOLOGIA

2.1. *Extreme Programming*

O desenvolvimento deste trabalho visou implementar algumas das práticas da metodologia ágil XP (*Extreme Programming*). Que é um conjunto de práticas para equipes pequenas e médias, desenvolverem *software* baseado em requisitos vagos.

Dentre as práticas utilizadas estão:

- Planejamento – Focar-se em requisitos atuais e não em requisitos futuros.
- Teste – No final do desenvolvimento de uma funcionalidade, a mesma foi testada diversas vezes.
- Cliente presente – Reuniões mensais aconteceram para apresentar o progresso e receber sugestões.
- Código padrão – Padronização da arquitetura e nomenclatura do código.

2.2. Padrão de desenvolvimento

MVC (*Model View Controller*): Arquitetura de desenvolvimento de sistemas dividida em camadas, a fim de manter o projeto melhor estruturado e organizado, com seus conceitos bem divididos.

A utilização desse padrão, permite o isolamento das regras de negócios da lógica de execução e da interface com o usuário. Desta forma, é possível, por exemplo, alterar a interface do usuário sem que haja a necessidade de alterar as regras de negócios, e vice-versa.

Cada uma dessas camadas possui tarefas muito bem definidas. São elas:

- *Controller(*controlador*):* Responsável pela separação em camadas. Esta, faz a ponte entre interfaces e as regras de negócios.
- *View(*visão*):* São as interfaces gráficas do sistema, ou seja, o que o usuário de fato enxerga ao utilizar o mesmo. Estas interfaces possuem eventos atrelados a seus elementos, que ao serem executados, enviam requisições para o controlador.

- *Model*(modelo): São as classes, os moldes de objetos do sistema com seus atributos e métodos. É essa camada que modela o problema que buscamos resolver.
- No caso específico deste projeto, na camada de modelo, foi utilizado também o padrão DAO (*Data Access Object*). Responsável pela execução de comandos DML (*Data Manipulation Language*) no banco de dados do sistema.

O fluxo desse padrão acontece da seguinte forma: o usuário interage com a parte visual, como o clique em um botão, e a *view* envia essa informação ao *controller.* Ele, por sua vez, acessa o *model*. Isso normalmente causa alterações nas informações do mesmo.

Após a requisição ser completamente realizada, o *controller* repassa as informações vindas do *model* para a *view.*

As figuras a seguir exemplificam cada uma das camadas.

Figura 1. Model Cliente

```
@Entity
public class Cliente {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long idCliente;
    private String nome;
    private String email;
    private String telefone;
    private String cpf;
    private String cnpj;
    private String rg;
    private Boolean secundario;

    @OneToOne
    @JoinColumn(name = "idClienteSec", nullable = true)
    private Cliente ClienteSec;

    private TipoCliente tipoCliente;

    @Enumerated(EnumType.STRING)
    private Status status;

    @OneToMany(cascade = CascadeType.ALL,
               orphanRemoval = true)
    private List<Projeto> projetos = new ArrayList<>();

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "idEndereco")
    private Endereco endereco;

    public void addProjeto(Projeto projeto) {
```

Fonte: Autoria própria.

A figura 1 exemplifica um *model* presente no projeto. Neste caso a classe Cliente, com todos os seus atributos.

Figura 2. ClienteController

```
@Controller
public class ClienteController {

    @Autowired
    private ClienteDAO clienteDao;

    @Autowired
    private EnderecoDAO enderecoDao;

    // Abre view de cadastro
    @RequestMapping("/clientes/novo")
    public ModelAndView open() {
        ModelAndView model = new ModelAndView("clientes/novo");
        model.addObject("tiposCliente", TipoCliente.values());
        model.addObject("estados", Estado.values());
        model.addObject("enderecos", enderecoDao.list());
        return model;
    }

    // Abre view de consulta
    @Cacheable(value = "clientList")
    @RequestMapping("/clientes/consulta")
    public ModelAndView list() {
        ModelAndView model = new ModelAndView();
        List<Cliente> lClientes = clienteDao.list();
        model.addObject("clientes", lClientes);
        model.addObject("tiposCliente", TipoCliente.values());
        return model;
    }
```

Fonte: Autoria própria.

A figura 2 mostra um trecho de um dos *controllers* presentes no projeto, com dois métodos e dois atributos. O primeiro método, *open,* busca no banco de dados todos os endereços, os tipos de cliente e os estados nacionais. Os dois atributos são objetos DAO, responsáveis pela comunicação com o banco de dados.

Figura 3. ClienteDAO

```
@Transactional
public class ClienteDAO {

    @PersistenceContext
    private EntityManager manager;

    public void saveClient(Cliente cliente) {
        manager.persist(cliente);
    }

    public List<Cliente> list() {
        return manager.createQuery("select c from Cliente c", Cliente.class).getResultList();
    }


    public boolean findClientByCpf(Cliente pCliente) {

        try {
            Cliente lCliente = manager
                    .createQuery("select distinct(c) from Cliente c" + " where c.cpf = :cpf",
                            Cliente.class)
                    .setParameter("cpf", pCliente.getCpf())
                    .getSingleResult();

            if (lCliente != null)
                return true;
            else
                return false;
        } catch (Exception e) {
            return false;
        }

    }
```

Fonte: Autoria própria.

A figura 3 exibe a classe ClienteDAO, da camada de modelos. Como já descrito anteriormente, é nessa parte do processo que acontece a conexão com o banco de dados.

Figura 4. *View* novo cliente

```
<form:form class="form-horizontal" id="novoCliente" action="${s:mvcUrl('CC#save').build() }" method="POST">
    <div class="box-body">

        <div class="form-group">
            <label class="col-sm-4 control-label">Tipo Cliente*</label>
            <div class="col-sm-4">
                <select name="tipoCliente" id="tipoCliente"
                    class="form-control select2 select2-hidden-accessible"
                    style="width: 100%;" tabindex="-1" aria-hidden="true" required>
                    <c:forEach items="${tiposCliente }" var="tipoCliente">
                        <option value="${tipoCliente}">${tipoCliente.name}</option>
                    </c:forEach>
                </select>
            </div>
        </div>

        <div class="form-group">
            <label for="nome" class="col-sm-4 control-label">Nome*</label>
            <div class="col-sm-4">
                <input type="text" class="form-control" name="nome" id="nome"
                    required>
            </div>
        </div>

        <div class="form-group">
            <label for="nome" class="col-sm-4 control-label">Email*</label>
            <div class="col-sm-4">
                <input type="email" class="form-control"
                    name="email" id="email" required>
            </div>
        </div>
```

Fonte: Autoria própria.

A figura 4 representa um trecho da *view* responsável pelo formulário para inserção de um novo cliente.

2.3. Ferramentas utilizadas

O sistema foi desenvolvido para web, devido a possibilidade de utilizá-lo com um computador simples que possui um navegador *web* e acesso ao servidor ou conexão com a internet, para um possível servidor hospedado na nuvem.

Java: Linguagem utilizada no *back-end,* sendo responsável pela definição das regras de negócio do sistema.

Foram utilizadas as versões 1.8.0_161 do JDK (*Java Development Kit*) e a 1.8.0_171 do JRE (*Java Runtime Environment*). A IDE de desenvolvimento foi o *Eclipse Oxygen.2* versão 4.7.2.

Para o servidor *Java* web, foi utilizado o Tomcat na versão 7.0.

HTML5 ***(Hypertext Markup Language)***: Quinta versão da linguagem chave da internet. Utilizada para apresentação de conteúdo nos navegadores.

Com essa versão, foi possível desenvolver o sistema de maneira responsiva para diversos navegadores e dispositivos, como computadores, tablets e celulares.

CSS3 (Cascading Style Sheets): Utilizada junto ao HTML5 nas páginas web. Responsável pela definição dos estilos do sistema, como cores, fontes, tamanhos, posicionamento, entre outros.

JavaScript: Outra linguagem focada no mundo web, que tem como principal objetivo executar *scripts* de código do lado do cliente (*client-side*), sem a interação direta com o servidor. Dessa forma é possível realizar comunicações assíncronas, alterando o conteúdo da página sem que ela seja recarregada, proporcionando uma melhor experiência para o usuário.

AdminLTE: *Framework* gratuito que consiste em um pacote de recursos, como diversos *templates* HTML, classes CSS e bibliotecas *JavaScript* para o desenvolvimento da parte estética das páginas do sistema.

Esse *framework* possui bibliotecas famosas como o *Jquery, Bootstrap,* FontAwesome, *Select2, InputMask,* entre outros. Todos com a finalidade de facilitar o desenvolvimento *front-end* da aplicação.

MySQL: O banco de dados foi montado com o MySQL, utilizando a IDE HeidiSQL na versão 9.5.0.5196.

XAMPP *Control Panel*: Ferramenta que possibilita a criação de um servidor web local, afim de testar a aplicação no próprio ambiente de desenvolvimento. Essa ferramenta foi utilizada na versão 3.2.2.

SpringMVC: Framework *Java*, utilizado para facilitar o desenvolvimento de aplicações *web*. Ele possui todas as funcionalidades que são necessárias para atender requisições *web*. Formata os dados recebidos, os transformando em objeto Java e entrega esse objeto para ser utilizado no código. Ele faz o mesmo processo de formatação para o envio de respostas. Esse framework foi utilizado na versão 4.3.17.

JSTL (*Java Server Pages Standard Tag Library*): Tecnologia que oferece *tags* que são utilizadas nas *Views* do sistema *web*. Proporcionando funcionalidades cruciais, como por exemplo, laços de repetição e estruturas de decisão.

Apache Maven: Ferramenta para automação de construção e gerenciamento de projetos. Utilizado neste trabalho para a estruturação dos pacotes, e para a injeção de dependências.

AJAX (*Asynchronous JavaScript and XML*): Tecnologia de desenvolvimento web, para deixar as aplicações mais interativas. Normalmente utilizada para troca de pequenas quantidades de informação com o servidor, dessa maneira, deixa as respostas do servidor mais rápidas. Além disso, ela evita que a tela tenha que ser recarregada quando uma nova informação chega do servidor.

JSON (JavaScript *Object Notation*): Modelo de armazenamento e transmissão de informações no formato de texto, utilizado nas requisições feitas com a tecnologia AJAX. Essa ferramenta foi utilizada da seguinte forma: os dados do formulário HTML são transformados em um objeto JSON através de um método JavaScript. Esse objeto é enviado ao servidor através do AJAX, que, quando chega em um *Controller* Java, o objeto é convertido para uma classe presente no projeto. Essa conversão acontece automaticamente através de recursos fornecidos pelo SpringMVC, basta o objeto JSON possuir o mesmo nome e os mesmos atributos da classe Java.

JPA (*Java Persistence API*): Framework para padronizar a persistência de objetos Java no banco de dados.

Hibernate: Implementação do JPA, utilizado para consultar e persistir objetos. Através dele é possível mapear classes Java para tabelas do banco de dados, e tipos de dados Java para tipos de dados SQL. Esse framework foi utilizado na versão 4.3.11.

3. IMPLEMENTAÇÃO

3.1. Levantamento de requisitos

O levantamento de requisitos foi realizado em reuniões com Gabriela De Nadai, arquiteta e uma das sócias da empresa em questão. Os principais requisitos funcionais identificados foram:

3.1.1. Requisitos funcionais

Os requisitos funcionais levantados nas reuniões foram:

- **Cadastro / Manutenção de usuários do sistema:**

O sistema deve permitir a inclusão, alteração e inativação (remoção lógica) de usuários. Esses serão os responsáveis pela utilização do sistema.

O cadastro deve conter os seguintes atributos: nome, e-mail, senha e permissão.

Existem dois tipos de permissão, administrador e usuários. O usuário terá restrições relacionadas a visualização de dados, enquanto o administrador não terá quaisquer restrições, além disso ele terá acesso a uma tabela com todos os usuários cadastrados no sistema, onde poderá alterar os mesmos.

Um usuário poderá ser inativado e alterado a qualquer momento por um administrador, dessa forma esse usuário ficará impossibilitado de acessar o sistema.

Todos os dados do cadastro serão obrigatórios.

- **Cadastro / Manutenção de clientes:**

O sistema deve permitir a inclusão e alteração de clientes. O cadastro deve conter os seguintes atributos: nome, e-mail, telefone, cpf ou cnpj, rg e endereço.

Foi requisitado a possibilidade de cadastrar duas pessoas em um mesmo cliente (cliente secundários), devido a possibilidade de o cliente ser um casal ou uma empresa. Situação que, segundo a sócia, acontece com frequência.

Dos dados cadastrais, apenas nome, e-mail, telefone e endereço serão obrigatórios. Desses dados, o cpf será único.

Todos os usuários terão acesso a uma tabela com todos os clientes cadastrados no sistema, sendo possível a alteração dos dados de cada um desses clientes.

- **Cadastro / Manutenção de projetos:**

O sistema deve permitir a inclusão e alteração de projetos, sendo que cada projeto deverá ser atrelado a somente um cliente, que possui ou não um cliente secundário.

O cadastro será composto pelos seguintes atributos: cliente, tipo de projeto, medidas do terreno, medidas da construção, categoria do projeto endereço do projeto e observações.

Após a inserção de um novo projeto, o mesmo passa a ter três fases: pré-projeto, anteprojeto e projeto. Cada uma delas composta por diferentes etapas.

Todos os usuários terão acesso a uma tabela com todos os projetos cadastrados no sistema, sendo possível a alteração dos dados de cada um desses projetos.

- **Cadastro / Manutenção de endereços:**

O sistema deve permitir a inclusão e alteração de endereços.

Caso o usuário queira cadastrar um novo endereço, isso será feito junto ao cadastro de clientes e projetos, tendo em vista que um endereço sempre estará atrelado a um cliente ou um projeto. A alteração de endereços poderá ser feita no cadastro de clientes e projetos, ou na consulta dos mesmos.

Este cadastro possui os seguintes campos: cep, rua, bairro, cidade, estado, complemento, condomínio, quadra, lote, cadastro prefeitura e número. Nenhum campo especifico é obrigatório, basta o usuário informar pelo menos um dos campos.

Caso o usuário possua o cep do local, o mesmo, ao ser digitado, buscará através de um *web service,* chamado ViaCep, algumas das informações do endereço, como rua, bairro, cidade e estado.

- **Cadastro / Manutenção de categorias e tipos de projeto:**

O sistema deve permitir a inclusão e alteração de categorias e tipos de projeto.

Esses cadastros poderão ser feitos tanto na tela de cadastro de projetos quanto na própria tela de cadastro de categorias de projetos, o mesmo vale para os tipos de projeto.

Além disso, todos os usuários terão acesso a tabelas com todas as categorias e tipos de projetos cadastradas no sistema, sendo possível a alteração dos dados de cada uma dessas categorias e tipos.

- **Consulta de logs de projetos do sistema**

O sistema deve gravar um histórico das operações realizadas nos projetos cadastrados no sistema, com informações como: qual o usuário responsável pela operação, a data e qual foi a operação. Apenas administradores do sistema poderão ter acesso aos logs.

3.1.2. Requisitos não funcionais

Requisitos não funcionais dizem respeito a como as funcionalidades serão entregues ao usuário do *software*. São requisitos que se esperam de um sistema confiável.

- **Confiabilidade:**

O sistema deve restringir o acesso a informações de acordo com o nível de privilégio de cada usuário. Apenas usuários com privilégio de administrador podem alterar dados sensíveis, como os dados de outros usuários.

Esses dados não devem ser removidos fisicamente do sistema, deve ser feita uma remoção lógica, a fim de manter todos os dados guardados caso sejam necessários no futuro.

- **Interface:**

O sistema deverá apresentar um *layout* amigável e intuitivo, sempre mostrando os caminhos para o usuário, buscando deixar sua experiência o mais agradável possível.

As cores do *layout* devem ser as cores determinadas pela cliente.

- **Conexão:**

A hospedagem do servidor poderá acontecer de duas formas: com um servidor local ou em nuvem.

- **Portabilidade:**

Para utilizar o sistema basta um computador com acesso ao servidor e um navegador *web* instalado.

O sistema também deve ser compatível com navegadores em dispositivos móveis.

3.1.3. Incompatibilidade

- **Navegadores:**

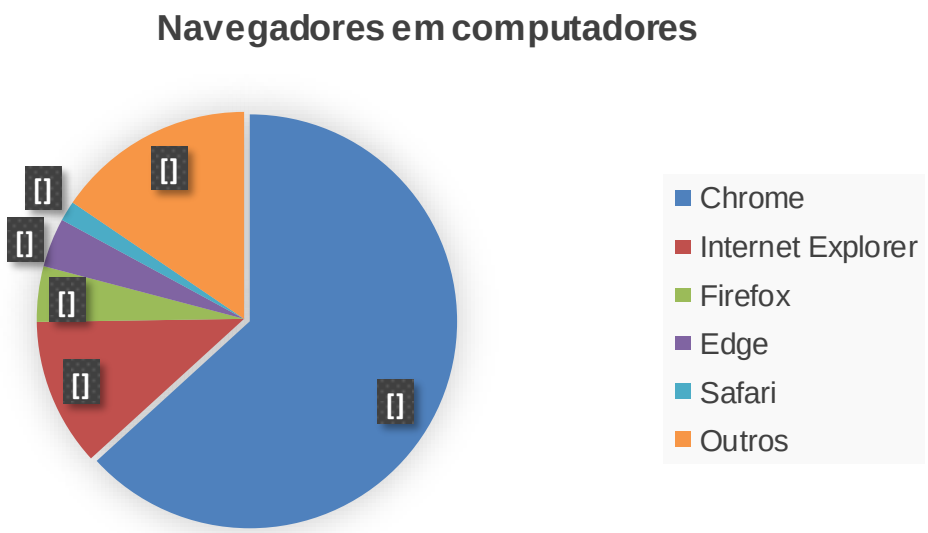
A parte visual do sistema se comporta de formas diferentes entre os navegadores, pois cada um deles interpreta os códigos HTML do seu jeito. Isso, muitas vezes, compromete o *design* e as funcionalidades das páginas *web*.

Foram realizados testes em cinco navegadores, os resultados foram os seguintes:

- **Google Chrome / Microsoft Edge:** Nenhuma falha encontrada, tanto visual quanto funcional. O sistema pode ser utilizado em sua plenitude na versão do navegador para computadores e para celulares
- **Internet Explorer:** Problemas visuais que impedem a utilização de funcionalidades do sistema. Não é possível desfrutar do sistema com esse navegador.
- **Mozila Firefox:** Apenas erros visuais foram encontrados. Os mesmos não impedem a utilização das funcionalidades do sistema.
- **Safari:** Não é possível utilizar o sistema nesse navegador.

Segundo o site netmarketshare.com, a porcentagem de utilização de navegadores no mercado, entre janeiro de 2018 e setembro de 2018, foram as seguintes:

Figura 5. Gráfico de navegadores



Fonte: Autoria própria.

3.2. Definição da equipe e cronograma das atividades

3.2.1. Equipe

A equipe foi composta por apenas uma pessoa. Responsável por todas as etapas desse trabalho.

3.2.2. Cronograma

Nenhuma técnica especifica foi seguida para a elaboração do cronograma de desenvolvimento desse trabalho. Foi utilizado um método de elaboração de *backlogs*, aprendido no ambiente de trabalho.

Dois desses *backlogs* foram criados em planilhas do Google Docs: um para o desenvolvimento do sistema e outro para sua documentação.

O de desenvolvimento foi divido em seis colunas, são elas:

- **Item**: O que deve ser feito.
- **Menu**: Em qual dos menus do sistema o item se encontra.
- **Status**: Como está o andamento do item.
- **Observações**: Descrição de *bugs,* lembretes e anotações em geral.

- **Data início**: Quando foi iniciado o item.
- **Data fim**: Quando o item foi concluído.

A tabela a seguir mostra um trecho desse documento.

Tabela 1. Trecho do backlog de desenvolvimento

BACKLOG TCC					
Item	**Menu**	**Status**	**Observações**	**Data Inicio**	**Data Fim**
Cadastro de novo usuário	Usuários	Ok	Adicionar combo de permissões	29/07	30/07
Validações na tela de login	Login	Ok	Adicionar mensagem de usuario invalido	29/07	29/07
Impedir login de usuários inativos	Login	Ok	Mensagem generica para todos os erros	29/07	30/07
Inativar/Ativar usuários	Usuários	Ok	Fazer com AJAX	30/07	01/08
Mudar mensagens nas data tabels para PT-BR	Todos	Ok	JS ou CSS?	01/08	01/08
Criar menu de clientes	Clientes	Ok		03/08	05/08
Mudar cores dos botões nas datatables	Todos	Ok		30/07	30/07
Implementar Cache na lista de usuários	Usuários	Ok	Melhorar desempenho	01/08	01/08
Impedir que um usuário se inative	Usuários	Ok	Pegar o id do usuario pela session ou springSecurity	01/08	01/08
Modal de alteração de usuário	Usuários	Ok		01/08	01/08

Fonte: Autoria própria.

O backlog da documentação foi definido com as mesmas colunas da tabela acima, com exceção a coluna de “Menu”. Como mostra a tabela abaixo.

Tabela 2. Trecho do *backlog* de documentação

BACKLOG TCC				
Item	**Status**	**Observações**	**Data Inicio**	**Data Fim**
1.Introdução	Ok	Melhorar texto	17/09	10/10
2. Metodologia	Aguardando aprovação	Evandro	10/09	20/10
Extreme programming	Ok		10/09	15/09
Padrão de desenvolvimento	Ok		01/10	15/10
Ferramentas	Ok		10/09	20/10
3. Implementação	Em andamento		16/07	
Requisitos funcionais	Ok		16/07	20/09
Requisitos não funcionais	Aguardando aprovação	Mais algum? - Cliente	16/07	20/09
Atas de reunião	Aguardando info.	Vai precisar? - Evandro	20/07	
Definição da equipe e cronograma das atividades	Em andamento		16/09	
Diagrama de caso de uso	Aguardando aprovação		17/09	
Diagrama de classe	Em andamento		10/10	
Diagrama Entidade Relacionamento	Aguardando aprovação		17/09	

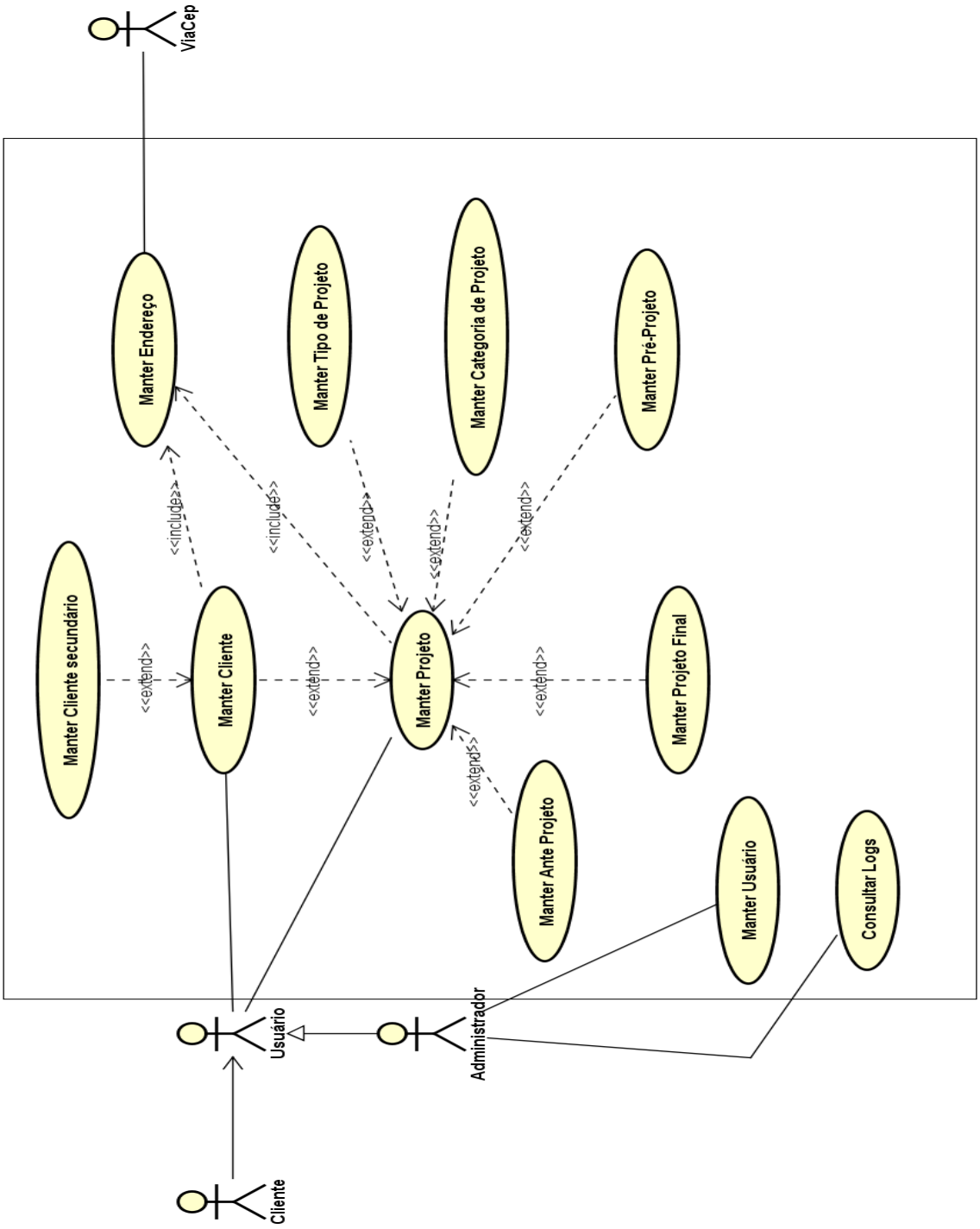
Fonte: Autoria própria.

3.3. Ferramenta para documentação do sistema

3.3.1. Caso de uso

O desenvolvimento do diagrama a seguir foi possível a partir dos requisitos funcionais levantados nas reuniões feitas com a cliente. O mesmo foi desenvolvido com a utilização da ferramenta UML, Astah *Community.*

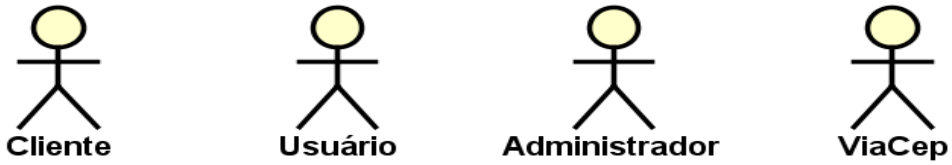
Figura 6. Diagrama de Caso de Uso



Fonte: Autoria própria.

Os atores envolvidos no diagrama acima são os seguintes:

Figura 7. Atores



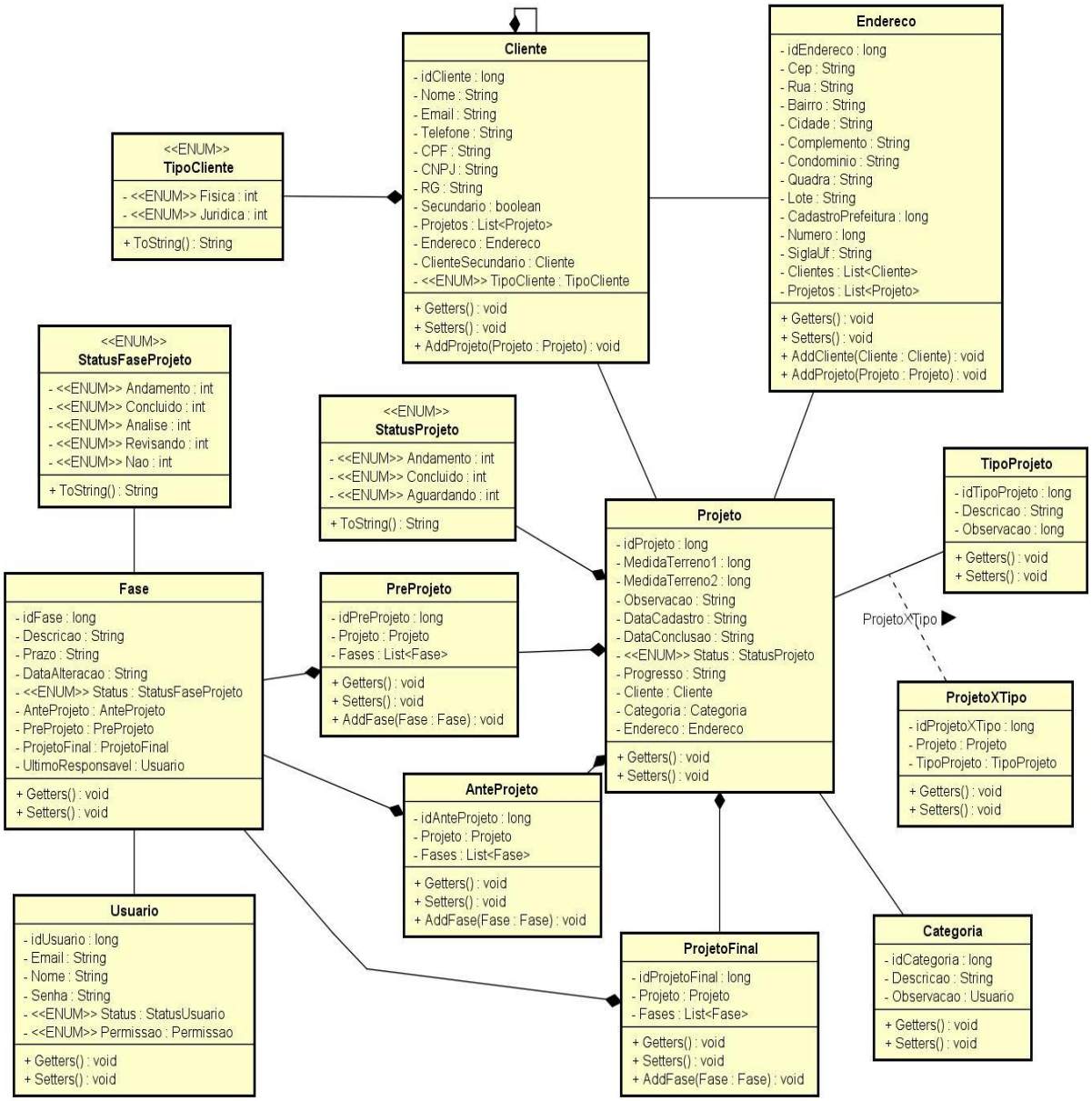
Fonte: Autoria própria.

- **Cliente:** Não tem acesso direto ao sistema, mas é através de informações passadas por esse ator que se torna possível alimentar o sistema.
- **Usuário:** Responsável pela utilização das funcionalidades do sistema.
- **Administrador:** Além de possuir as mesmas responsabilidades de um usuário, um administrador é quem faz a manutenção dos usuários do sistema.
- **ViaCep:** *Web Service* utilizado para cadastrar endereços.

3.3.2. Classe

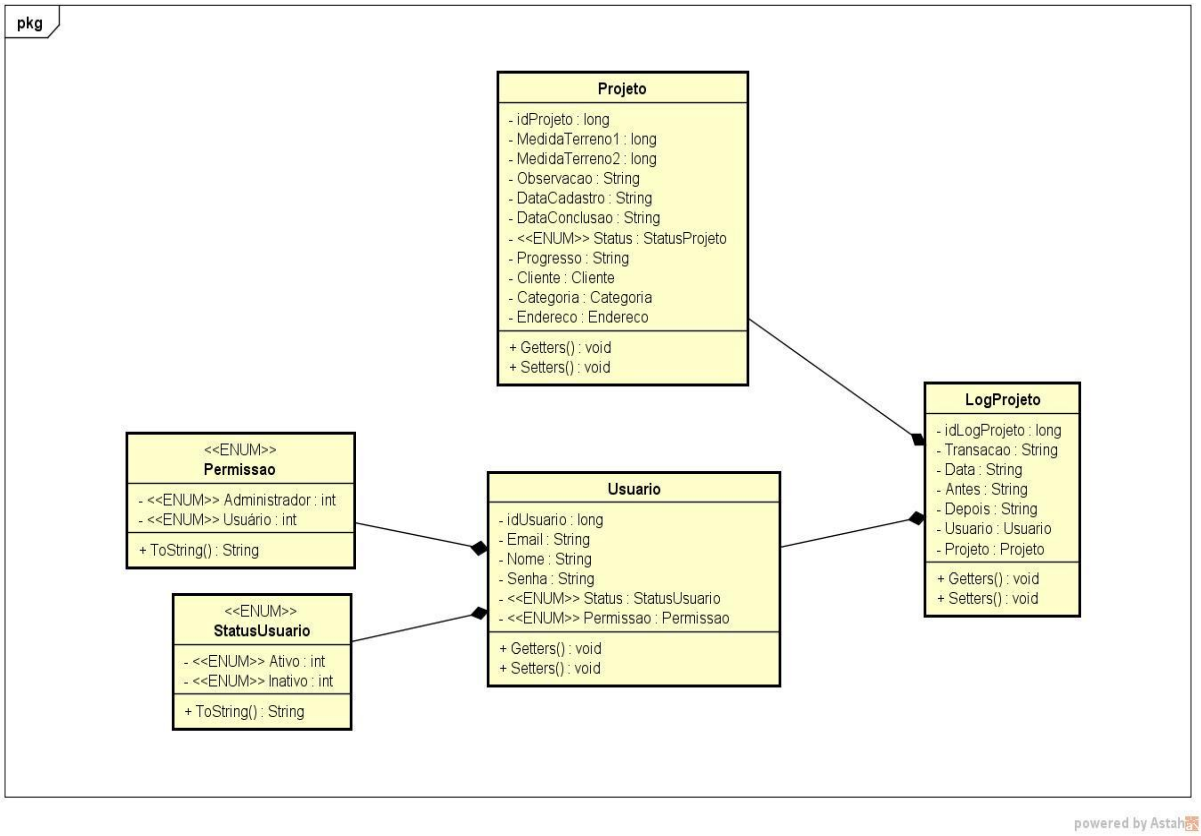
Os diagramas a seguir possuem as entidades presentes na modelagem do banco mais as classes presentes somente no universo Java. Essas classes que não se encontram no banco são do tipo ENUM (conjunto de valores pré-definidos), que se tornam bem eficazes com o Hibernate.

Figura 8. Diagrama de classe



Fonte: Autoria própria.

Figura 9. Diagrama de classe de logs



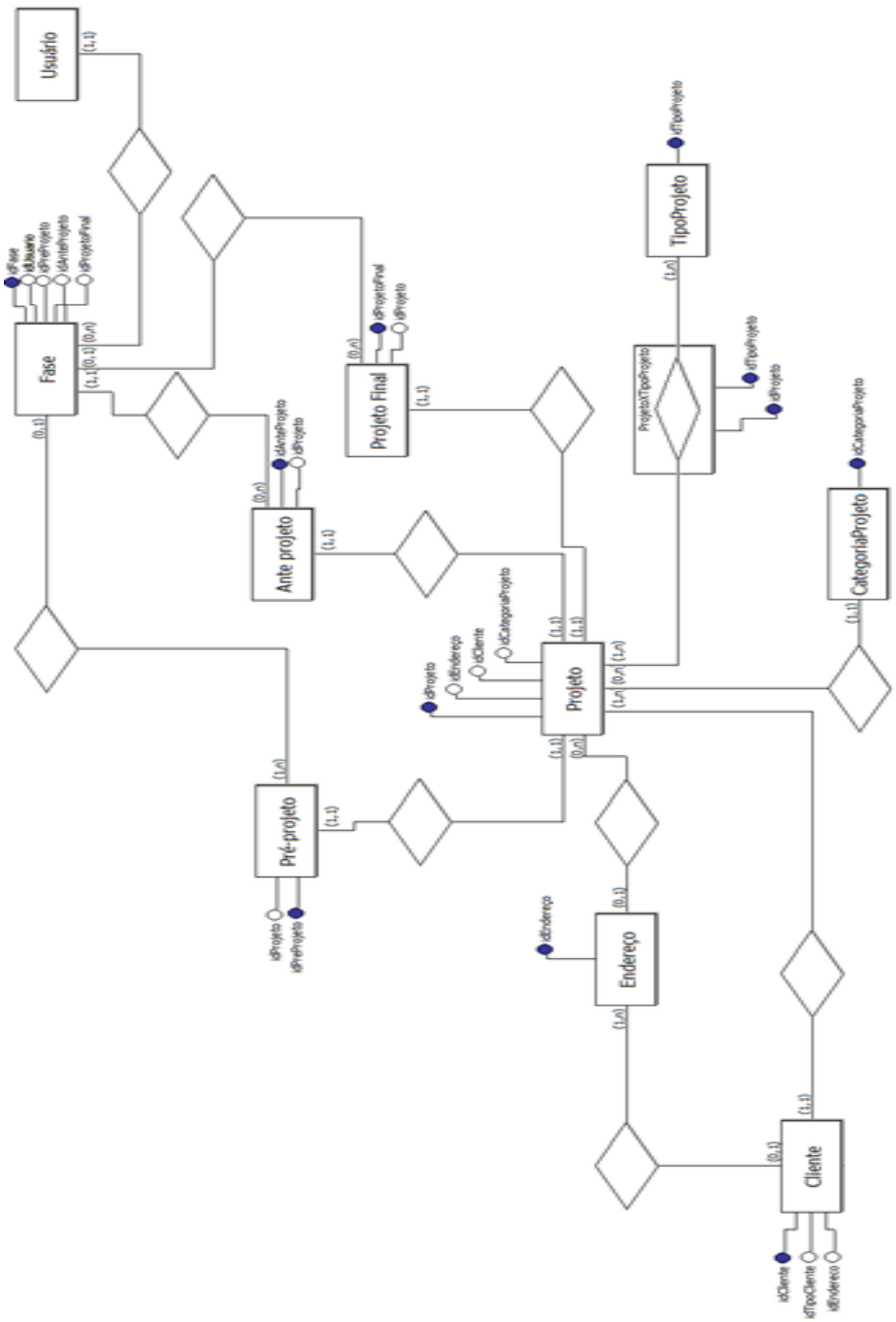
Fonte: Autoria própria.

3.4. Modelagem do banco de dados

3.4.1. Diagrama Entidade Relacionamento

Realizado baseando-se nas necessidades analisadas após as reuniões e durante o desenvolvimento do sistema. O mesmo foi realizado com a ferramenta BrModelo.

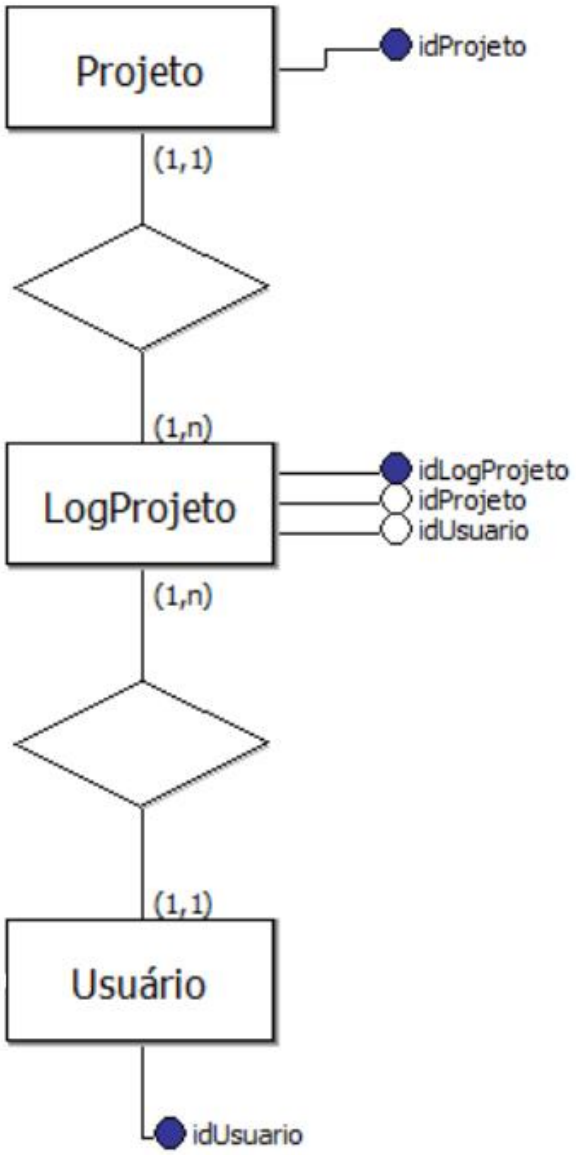
Figura 10. Diagrama Entidade Relacionamento



Fonte: Autoria própria.

O diagrama a seguir representa os relacionamentos envolvidos no processo de armazenamento de *logs* de projetos.

Figura 11. Diagrama de Entidade Relacionamento de *Logs*

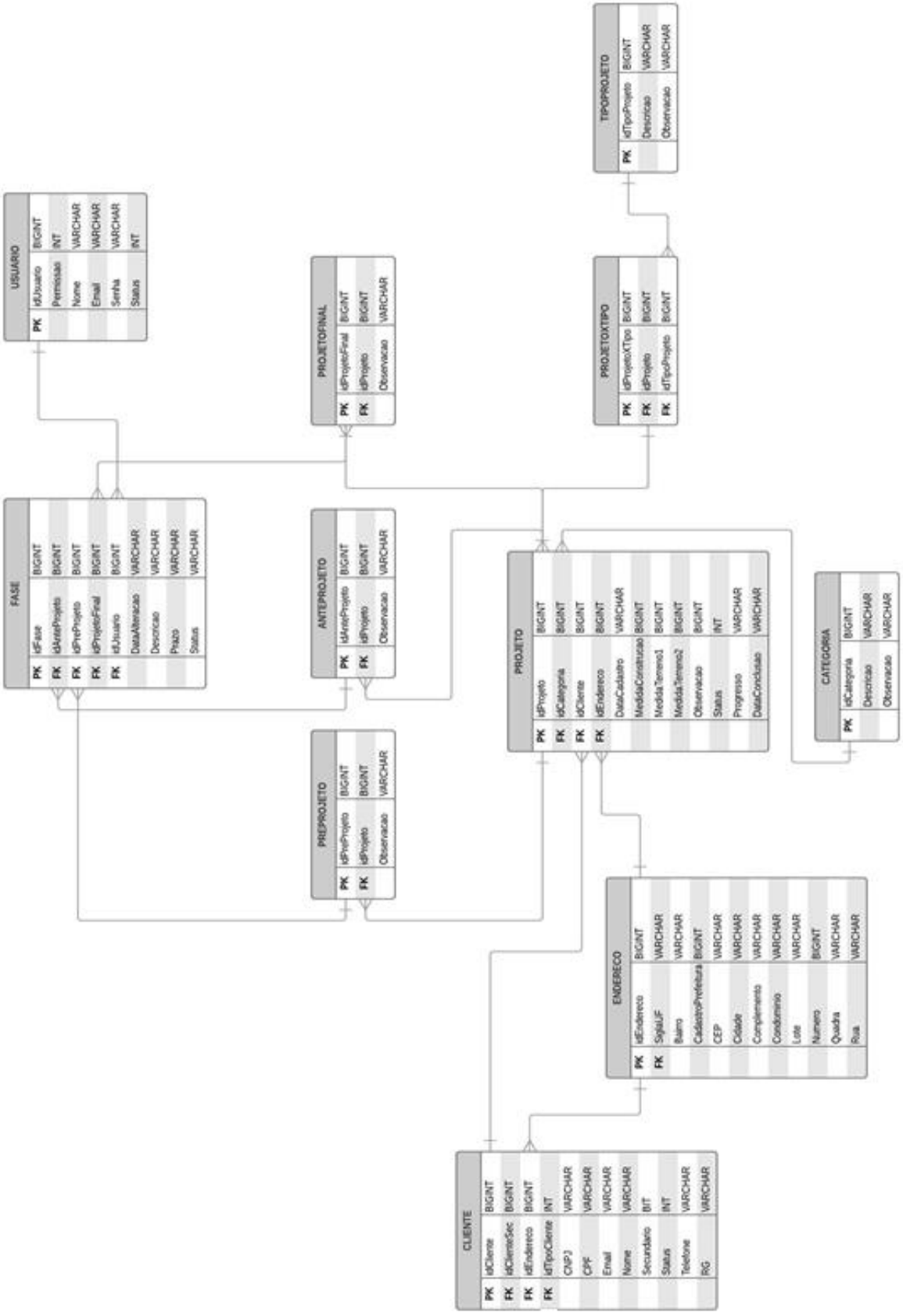


Fonte: Autoria própria.

3.4.2. Diagrama Lógico

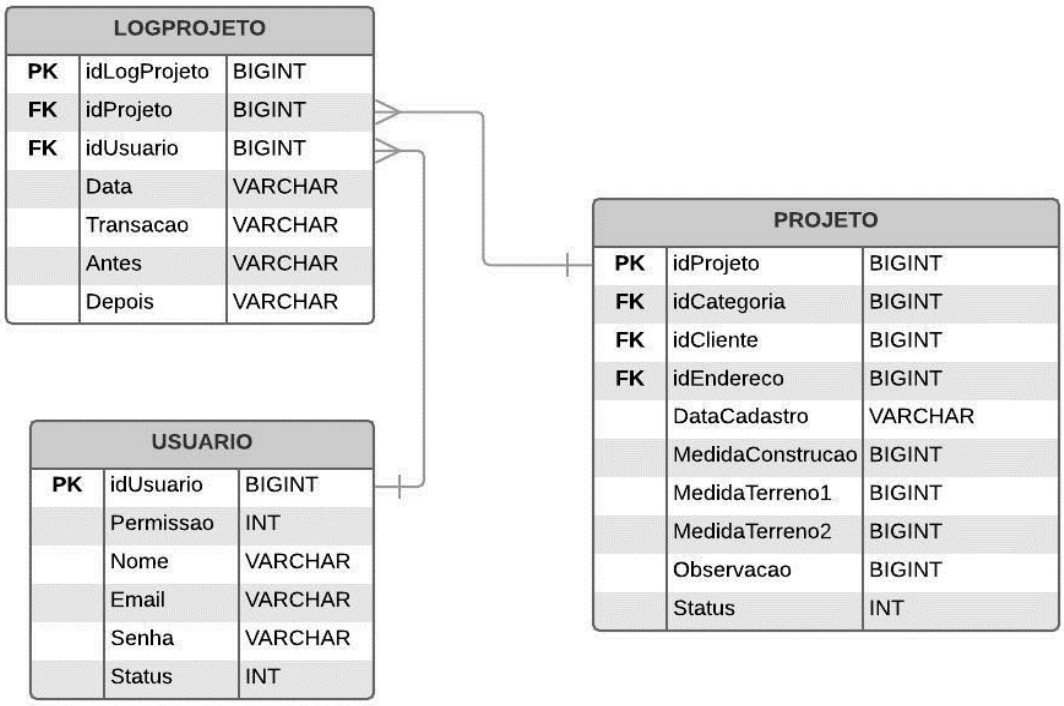
Além de possuir as mesmas entidades e relacionamentos do diagrama anterior, contém todos os atributos de cada uma delas. Os diagramas a seguir foram realizados com a ferramenta Lucidchart.

Figura 12. Diagrama lógico



Fonte: Autoria própria.

Figura 13. Diagrama lógico de logs



Fonte: Autoria própria.

3.4.3. JPA e Hibernate

Configurações

Para utilização dessas ferramentas, foi necessário a injeção de suas dependências através do Maven. A figura 14 mostra essas dependências.

Figura 14. Dependências do Hibernate, JPA e do driver para conexão com o banco de dados.

```
<!-- dependencias hibernate e jpa -->
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-entitymanager</artifactId>
    <version>${hibernate.version}</version>
</dependency>
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>${hibernate.version}</version>
</dependency>
<dependency>
    <groupId>org.hibernate.javax.persistence</groupId>
    <artifactId>hibernate-jpa-2.1-api</artifactId>
    <version>1.0.0.Final</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-orm</artifactId>
    <version>${spring.version}</version>
</dependency>

<!-- dependencia driver mysql -->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>${mysql.connector.version}</version>
</dependency>
```

Fonte: Autoria própria.

Além disso, foi necessário a criação da classe JPA*Configuration,* responsável por mapear os *models* do projeto para que o *framework* consiga interpreta-los. Essa classe também é encarregada por realizar a conexão com o banco de dados, através de um *driver*. E é com ela que dizemos ao JPA que iremos utilizar o Hibernate nas persistências de dados.

Cada uma dessas configurações foi realizada em métodos específicos, como mostra a figura 15.

Figura 15. Classe JPA*Configuration*

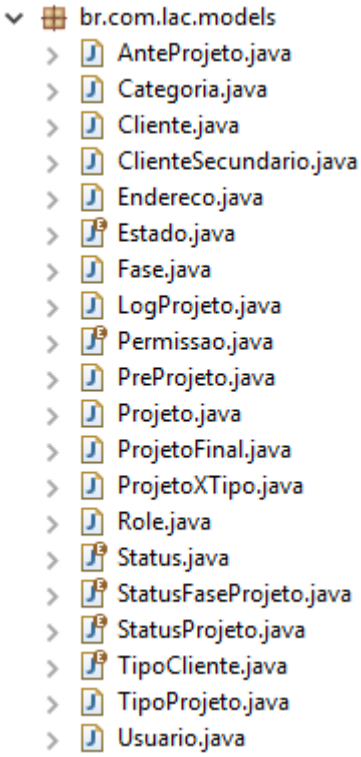
```
@Bean
public LocalContainerEntityManagerFactoryBean entityManagerFactory(DataSource dataSource) {

    LocalContainerEntityManagerFactoryBean factoryBean = new LocalContainerEntityManagerFactoryBean();

    JpaVendorAdapter vendorAdapter = new HibernateJpaVendorAdapter();
    factoryBean.setJpaVendorAdapter(vendorAdapter);

    factoryBean.setPackagesToScan("br.com.lac.models");

    factoryBean.setDataSource(dataSource);
    factoryBean.setJpaProperties(additionalProperties());

    return factoryBean;
}

private Properties additionalProperties() {
    Properties properties = new Properties();
    properties.setProperty("hibernate.dialect", "org.hibernate.dialect.MySQL5Dialect");
    properties.setProperty("hibernate.show_sql", "true");
    properties.setProperty("hibernate.hbm2ddl.auto", "update");
    return properties;
}

@Bean
@Profile("dev")
private DriverManagerDataSource dataSource() {
    DriverManagerDataSource dataSource = new DriverManagerDataSource();
    dataSource.setUsername("root");
    dataSource.setPassword("");
    dataSource.setUrl("jdbc:mysql://localhost:3307/lac_arquitetura");
    dataSource.setDriverClassName("com.mysql.jdbc.Driver");
    return dataSource;
}
```

Fonte: Autoria própria.

Muitas dessas configurações são padronizadas, por isso não será explicado cada uma delas, mas serão esclarecidas as principais, como a presente no método *dataSource*. É nele que dizemos ao JPA as informações para conexão com o banco de dados da aplicação, passando como parâmetros o usuário, senha, caminho e o driver necessário para esta conexão.

O método *entityManagerFactory* possui em uma de suas linhas o caminho para o JPA mapear os *models* do sistema, como mostram as figuras 16 e 17.

Figura 16. *Models* do projeto



Fonte: Autoria própria.

Figura 17. Linha com a definição do caminho para os *models*

```
factoryBean.setPackagesToScan("br.com.lac.models");
```

Fonte: Autoria própria.

Com esses *frameworks* configurados, é possível utilizar anotações (trechos com prefixo @, interpretados pelo compilador) nos *models* para que o Hibernate os crie no banco de dados como tabelas, caso essas tabelas não existam no mesmo. Dessa forma ganhamos tempo para focar no desenvolvimento da aplicação. Segue um exemplo prático disso na figura 18 e 19.

Figura 18. Tabela Categoria no banco de dados

#	Nome	Tipo de dados	Tamanho/Itens
1	idCategoria	BIGINT	20
2	descricao	VARCHAR	255
3	observacao	VARCHAR	255

Fonte: Autoria própria.

Figura 19. Atributos do *model* Categoria

```
@Entity
public class Categoria {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long idCategoria;
    private String descricao;
    private String observacao;
```

Fonte: Autoria própria.

As figuras acima demonstram como o Hibernate traduz uma classe Java para uma tabela de bando de dados, através do uso de anotações.

A anotação *@Entity* diz ao Hibernate que essa classe deve ser considerada uma entidade, logo, ela deve existir no banco de dados com a mesma estrutura presente no *model*. Dessa forma, a tabela criada tem os mesmos atributos presentes na classe com seus tipos também traduzidos para o banco.

A anotação *@Id* define a chave primaria da tabela em questão. E a *@GeneratedValue*, a forma como essa chave vai se comportar.

Utilização

Com o *model* configurado, agora é possível a criação de seu DAO para a manipulação do conteúdo da tabela gerada pelo Hibernate. A figura 20 demonstra o DAO para o *model* exibido anteriormente.

Figura 20. CategoriaDAO

```
@Repository
@Transactional
public class CategoriaDAO {

    @PersistenceContext
    private EntityManager manager;

    public void saveCategory(Categoria pCategoria) {
        manager.persist(pCategoria);
    }

    public List<Categoria> list(){
        return manager.createQuery("select c from Categoria c", Categoria.class).getResultList();
    }

    public Categoria getById(Long pId) {
        return manager.createQuery("select c from Categoria c where c.idCategoria = :id", Categoria.class)
                .setParameter("id", pId).getSingleResult();
    }
```

Fonte: Autoria própria.

O atributo do tipo *EntityManager* é o gerenciador de entidades, esse, fornecido pelo JPA, realiza a persistência, ou seja, a inserção, alteração e requisição de objetos no banco de dados.

Os métodos montam as queries executadas pelo banco de dados. Tratando-se do Hibernate, o recurso utilizado nessas queries é o HQL, acrônimo para Hibernate *Query Language*. Essa tecnologia permite utilizar objetos Java nos comandos SQL, pois, dada a quantidade de relacionamentos envolvidos, poderiam

gerar comandos muito extensos e mais difíceis de manter. Além disso, essa ferramenta é totalmente orientada a objetos, logo, consegue interpretar técnicas como herança e polimorfismo.

A figura 21 mostra uma consulta com a utilização do HQL, com relacionamentos feitos diretamente por objetos e não por campos.

Figura 21. *Query* utilizando HQL

```
Long lCount = (Long)manager.createQuery("select count(f) from Fase f"
        + " left join f.anteProjeto"
        + " left join f.preProjeto"
        + " left join f.projetoFinal"
        + " where f.status != :status"
        + " and (f.anteProjeto.projeto.idProjeto = :id"
        + " or f.preProjeto.projeto.idProjeto = :id"
        + " or f.projetoFinal.projeto.idProjeto = :id)")
        .setParameter("status", StatusFaseProjeto.Nao)
        .setParameter("id", pId)
        .getSingleResult();
```

Fonte: Autoria própria.

3.5. Desenvolvimento

Configurações iniciais

Como relatado anteriormente, o projeto utiliza o framework SpringMVC, e para a utilização do mesmo são necessárias algumas configurações básicas.

Primeiramente, foram feitas as injeções das dependências para a utilização do framework. Isso foi feito através do arquivo pom.xml. É com ele que o Maven baixa e configura o SpringMVC para o projeto. A figura abaixo mostra um trecho deste arquivo.

Figura 22. Pom.xml

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>4.3.17.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.apache.tomcat</groupId>
    <artifactId>tomcat-servlet-api</artifactId>
    <version>7.0.30</version>
    <scope>provided</scope>
</dependency>
<dependency>
    <groupId>javax.servlet.jsp</groupId>
    <artifactId>jsp-api</artifactId>
    <version>2.1</version>
    <scope>provided</scope>
</dependency>

<dependency>
    <groupId>javax.servlet.jsp.jstl</groupId>
    <artifactId>jstl-api</artifactId>
    <version>1.2</version>
    <exclusions>
        <exclusion>
            <groupId>javax.servlet</groupId>
            <artifactId>servlet-api</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

Fonte: Autoria própria.

Depois do Maven baixar essas dependências, é criado dentro do projeto uma pasta chama Maven *Dependencies*, contendo o que foi colocado no arquivo pom.xml.

A figura abaixo mostra um trecho desta pasta.

Figura 23. Maven Dependencies

Fonte: Autoria própria.

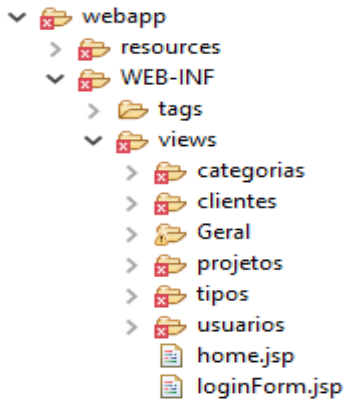
Além da injeção das dependências, para utilizar o SpringMVC foi necessário a configuração de algumas classes especificas, são elas:

- AppWebConfiguration.java: mapeia os pacotes do projeto, para que o SpringMVC encontre os *controllers*.
- ServletSpringMVC.java: responsável por configurar o servidor Tomcat para que ele passe as requisições para o SpringMVC.

Estrutura do projeto

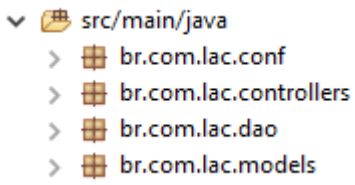
Para manter o projeto organizado foram criadas pastas para cada uma das camadas do padrão adotado. Como mostra a figura a seguir.

Figura 24. Pastas contendo a parte visual do



Fonte: Autoria própria.

Figura 25. Pacotes contendo classes Java



Fonte: Autoria própria.

Os pacotes da figura 25 são das camadas *controller e model*, ou seja, as camadas que ficam “por baixo dos panos”, que os usuários não veem ao utilizar o sistema. O primeiro pacote armazena classes de configuração para o desenvolvimento do projeto. O segundo contém todos os *controllers* do sistema, o terceiro os DAOs, e o quarto os *models.*

As pastas da figura 24 contém todos os componentes das *views.* A pasta *resources*, por exemplo, armazena todos os arquivos CSS e Java Script da aplicação. A pasta *tags,* contém um arquivo denominado pageTemaplate.tag, que é a base para todas as *views* do sistema. Esse arquivo possui todas as importações necessárias para a utilização dos arquivos presentes na pasta *resources*. Isto foi feito para manter o código HTML o mais limpo e objetivo possível. Além de importar todos os CSS e Java Scripts necessários, esse arquivo monta a estrutura de todas as telas do sistema, com o cabeçalho, o menu lateral e o rodapé.

Segurança com SpringSecurity

O SpringSecurity é um recurso fornecido pelo Spring para facilitar o desenvolvimento de mecanismos de autenticação e autorização, que implementa um filtro de requisições.

Para utilização dessa ferramenta foram injetadas suas dependências no arquivo pom.xml e implementada a classe de configuração, SecurityConfiguration.java. Com isso, foi possível bloquear as telas para usuários não autenticados. Por exemplo, se alguém digitar uma url para uma tela do sistema sem estar autenticado, esta pessoa será redirecionada para a tela de login.

O SpringSecurity também possibilitou a criptografia de senhas do tipo *hash*, ao serem recebidas pelo servidor, com a classe BCryptPasswordEncoder. A figura 26 mostra a configuração para o uso desta classe, e a figura 27 mostra a utilização da mesma.

Figura 26. Método responsável pela configuração do *Encoder*

```
@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
    auth.userDetailsService(usuarioDao).passwordEncoder(new BCryptPasswordEncoder());
}
```

Fonte: Autoria própria.

Figura 27. Utilização do *Encoder*

```
private BCryptPasswordEncoder passwordEncoder = new BCryptPasswordEncoder();

public void saveUser(Usuario pUsuario) {
    String hasSenha = passwordEncoder.encode(pUsuario.getSenha());
    pUsuario.setSenha(hasSenha);
    manager.persist(pUsuario);
}
```

Fonte: Autoria própria.

A figura a seguir exibe como a senha foi inserida no banco de dados após ser criptografada. Na figura 28, a senha é o conjunto dos seguintes caracteres, “123456”.

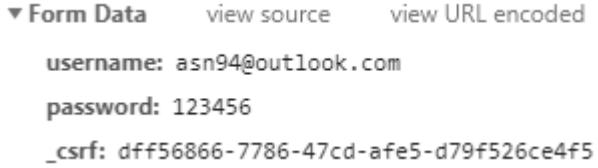
Figura 28. Senha criptografada

Fonte: Autoria própria.

Além disso, o SpringSecurity fornece proteção contra ataques CSRF, sigla em inglês para *Cross Site Request Forgery*. Esse tipo de ataque consiste na danificação ou roubo de dados de um usuário autenticado em um serviço *web*. Isso acontece através de um script, executado quando a vítima entra em um site malicioso. Dessa forma, o atacante consegue redirecionar o envio dos dados deste usuário para um servidor de sua escolha.

Para combater esse tipo de ataque o SpringSecurity fornece um *token* crsf, enviado em toda requisição feita pelo usuário. Dessa forma, o servidor sempre sabe quem de fato, está fazendo a requisição.

Figura 29. Dados da requisição enviada ao servidor



Fonte: Autoria própria.

Desempenho com *cache*

Para melhorar o desempenho do sistema foi utilizado o recurso conhecido como *Cache*, que consiste no armazenamento de dados no contexto da aplicação. Ou seja, esses dados ficam no navegador do usuário, dessa forma, quando certas informações forem requisitadas, a aplicação já as possui, evitando consultas repetitivas ao banco de dados.

Para implementar essa tecnologia no projeto foi utilizado um *framework* fornecido pela Google, o Guava. O mesmo foi baixado pelo Maven através da injeção de suas dependências e da configuração de um gerenciador de cache na classe AppWebConfiguration, como mostra a figura 30.

Figura 30. Configuração do *cacheManager*

```
public CacheManager cacheManger() {
    CacheBuilder<Object, Object> builder = CacheBuilder.newBuilder().maximumSize(100).expireAfterAccess(5,
            TimeUnit.MINUTES);
    GuavaCacheManager manager = new GuavaCacheManager();
    manager.setCacheBuilder(builder);
    return manager;
}
```

Fonte: Autoria própria.

Com essa configuração feita, é possível utilizar a anotação @Cachable em métodos dos *controllers* para dizer ao Spring quando utilizar o *cache*.

Figura 31. Utilização da anotação *Cachable*

```
// Abre view de consulta
@Cacheable(value = "clientList")
@RequestMapping("/clientes/consulta")
public ModelAndView list() {
    ModelAndView model = new ModelAndView();
    List<Cliente> lClientes = clienteDao.list();
    model.addObject("clientes", lClientes);
    model.addObject("tiposCliente", TipoCliente.values());
    return model;
}
```

Fonte: Autoria própria.

Como mostra a figura 31, todos os dados retornados pelo método *list ()* ficarão guardados na memória do navegador do usuário. Logo, sempre que o usuário acessar uma tela que aciona esse método, o sistema não precisa ir até o

banco de dados para buscar as informações necessárias, elas já estarão no próprio contexto da aplicação.

Para dizer ao Spring quando limpar essa informação, basta informar em qual método você deseja fazer isso. A imagem a seguir mostra isso na prática. O método *save*() recebe a anotação @CacheEvict, passando o nome da informação que o *controller* deve apagar do cache. Isso se torna necessário pois, quando um novo usuário é salvo, as informações do banco de dados são atualizadas, então a informação presente no *cache* deve ser trocada.

Figura 32. Utilização da anotação *CacheEvict*

```
// Salva novo cliente
@CacheEvict(value = "clientList", allEntries = true)
@RequestMapping(value = "/clientes/novo", method = RequestMethod.POST)
public ModelAndView save(Cliente pCliente,
                         ClienteSecundario pClienteSec,
                         @RequestParam String pEndereco,
                         @RequestParam String pEnderecoSec,
                         RedirectAttributes redirectAttributes) {
```

Fonte: Autoria própria.

3.6. Telas do sistema

Login

O banco de dados base do sistema já será criado com um usuário administrador padrão, pois o sistema não possui uma tela em que novos usuários possam se cadastrar. Os únicos capazes de inserir novos utilizadores são os administradores do sistema.

Figura 33. Tela de login

Fonte: Autoria própria.

Caso a senha ou e-mail sejam digitados incorretamente o sistema passa uma mensagem de erro genérica. A mesma mensagem, também é mostrada quando um usuário inativo tenta se logar.

Figura 34. Erro apresentado na tela de login

Fonte: Autoria própria.

Menu

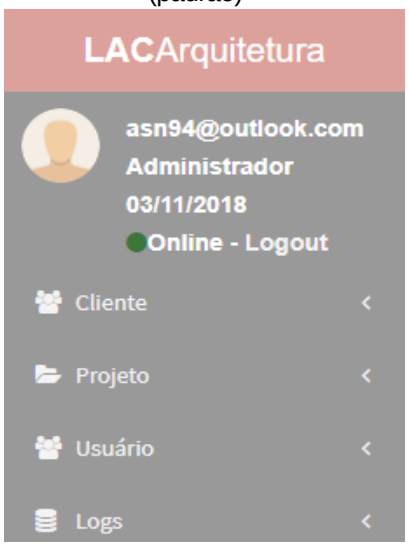
O menu lateral é a principal forma para o usuário navegar pelo sistema. Esse é divido em quatro categorias, são elas: cliente, projeto, usuário e *log*. Sendo os dois últimos exclusivos para usuários com permissões de administrador.

Na parte superior são exibidas informações sobre o usuário autenticado. Seu e-mail, permissão e a data do dia atual, além de um botão para efetuar o *logout*.

Duas opções de *layout* estão disponíveis, expandido e retraído. Sendo a segunda opção ideal para *mobiles,* ou para o usuário ter mais espaço no navegador *desktop*. Isso fica a critério do mesmo.

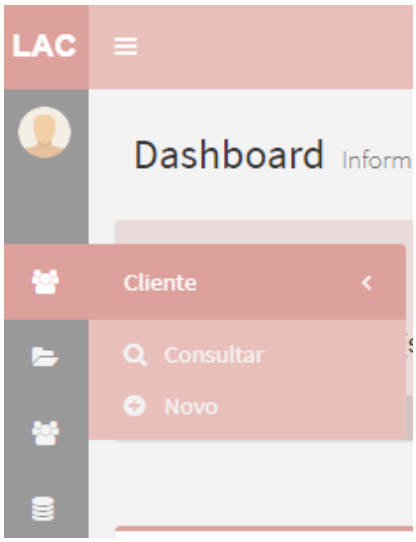
A figura 36 mostra o menu padrão, expandido e a figura 35, o menu retraído.

Figura 36. Menu expandido (padrão)



Fonte: Autoria própria.

Figura 35. Menu retraído



Fonte: Autoria própria.

Dashboard (*home*)

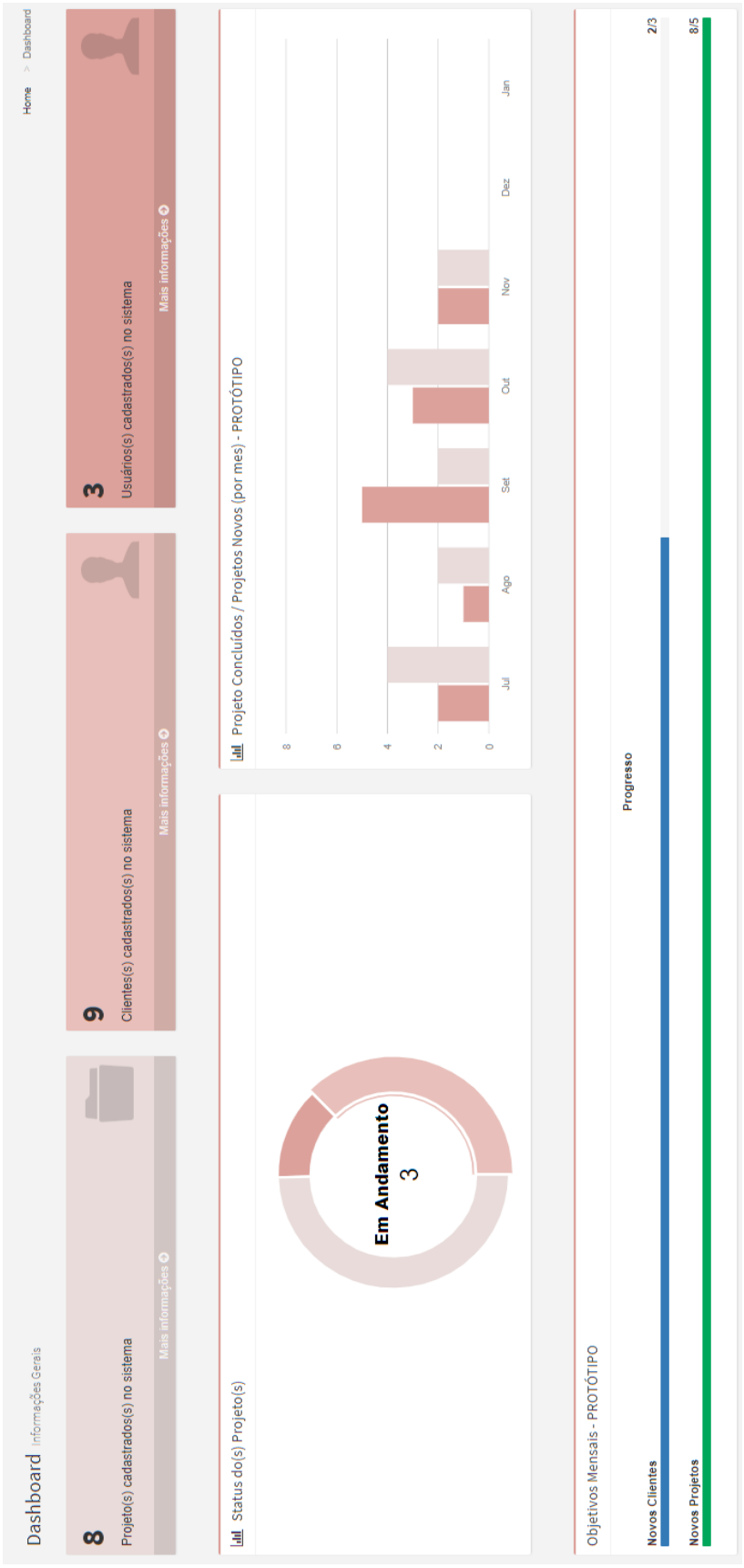
Tela que apresenta algumas informações sobre os dados presentes no sistema como a quantidade de projetos cadastrados, os usuários, os clientes, dois gráficos e os objetivos mensais.

O gráfico de status exibe quantos projetos estão concluídos, em andamento e aguardando. O gráfico de projetos concluídos e novos projetos (ainda em desenvolvimento), irão mostrar os dados baseados nos meses do ano.

A parte de objetivos mensais (também em desenvolvimento) define quais as metas do mês atual.

A figura a seguir mostra essa tela.

Figura 37. Tela dashboard (home)



Fonte: Autoria própria.

Clientes

Este menu possui duas opções: consulta e novo. A figura a seguir representa o formulário para o cadastro de um novo cliente e, caso exista, o cliente secundário atrelado a esse cadastro, como especificado nos requisitos funcionais.

Figura 38. Formulário de novo cliente

Fonte: Autoria própria.

Todos os campos seguidos do caractere “*”, são obrigatórios. Os demais campos são opcionais.

Ao clicar no botão “Cliente secundário”, um segundo formulário idêntico ao presente na figura acima aparece, tendo apenas um campo obrigatório: o nome. Além disso, o campo “Tipo Cliente” é sempre preenchido com a opção “Pessoa física”.

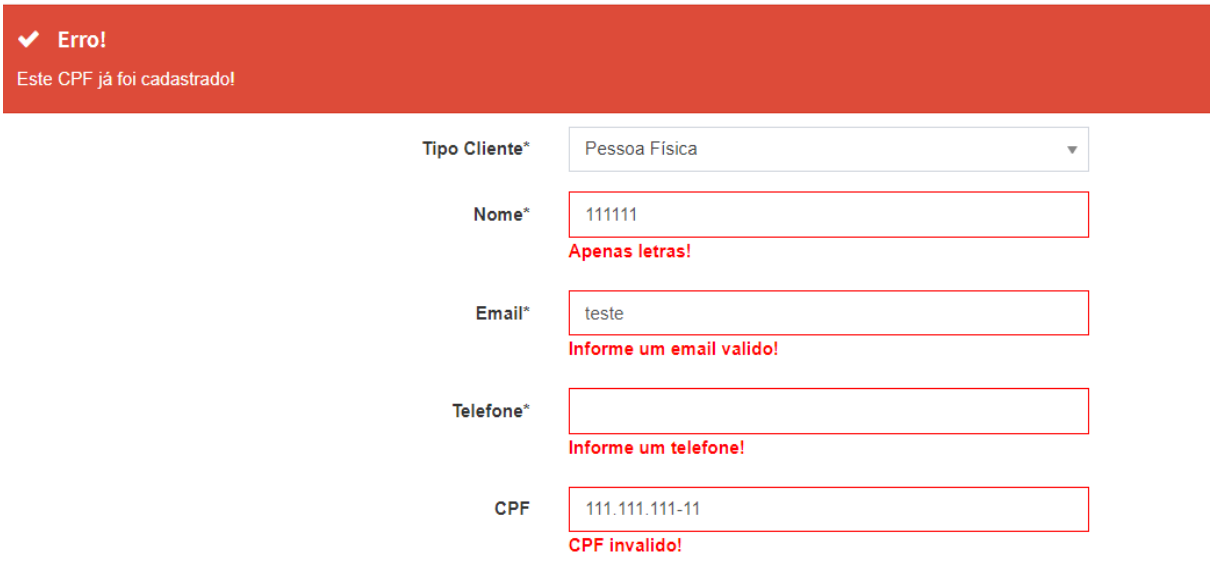
Campos e validações

- **Tipo Cliente**: Possui duas opções, pessoa física e pessoa jurídica.
- **Nome**: Este campo não aceita caracteres numéricos.
- **E-mail**: Validação baseada no caractere “@”.
- **Telefone**: O campo possui duas máscaras, de telefone residencial e celular, basta digitar um digito a mais para ativar a segunda.

- **CPF**: Validação baseada no cálculo de verificação. Além disso é checado no banco de dados se esse CPF já foi cadastrado.
- **CNPJ**: Ao selecionar a opção “Pessoa jurídica” no tipo de cliente, o campo CPF da lugar ao campo CNPJ, que também é validado com a utilização do cálculo.
- **RG**: Máscara apenas para documentos do estado de São Paulo.
- **Endereço**: Será explicado ainda neste capítulo.

A figura a seguir mostra exemplos das validações citadas acima.

Figura 39. Validações no cadastro de clientes



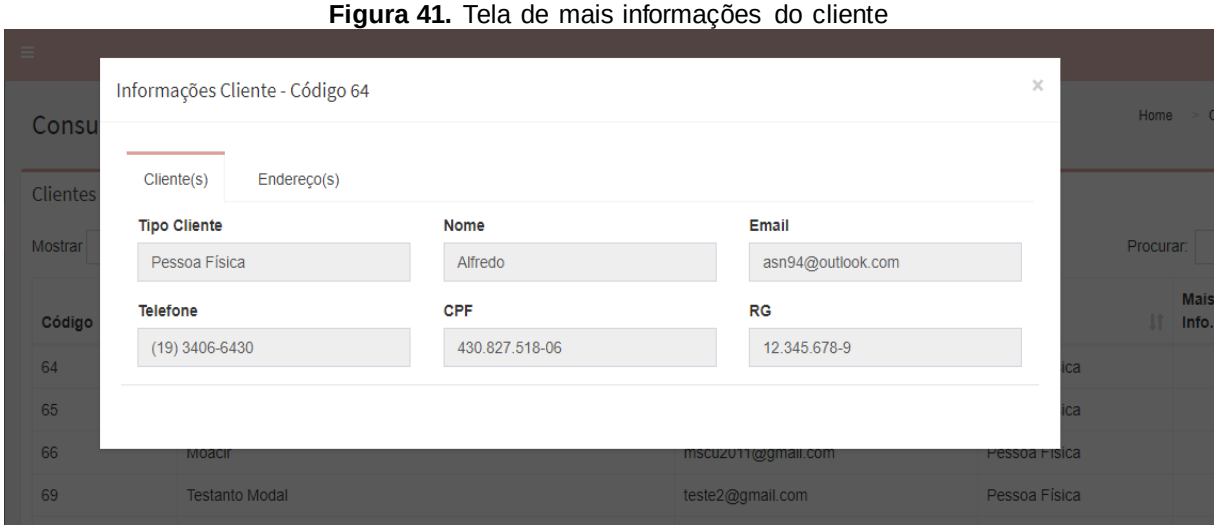
Fonte: Autoria própria.

Figura 40. Tela de consulta de clientes

Fonte: Autoria própria.

A tela de consulta de clientes, mostrada na figura anterior, exibe todos os clientes cadastrados no sistema, possibilitando a visualização de todas as informações e a alteração das mesmas. Isso acontece ao clicar nos ícones apresentados na coluna “Mais Info.” e “Alterar”. Ao clicar na opção de mais informações, é apresentado um *modal* que não possibilita qualquer alteração nos dados. Ao clicar na opção alterar, o mesmo *modal* é apresentado, mas com a possibilidade de alterar as informações.

Figura 41. Tela de mais informações do cliente



Fonte: Autoria própria.

Essa tela possui um campo “Procurar”, que possibilita o usuário filtrar os resultados apresentados. Qualquer um dos campos pode ser usado nesse filtro.

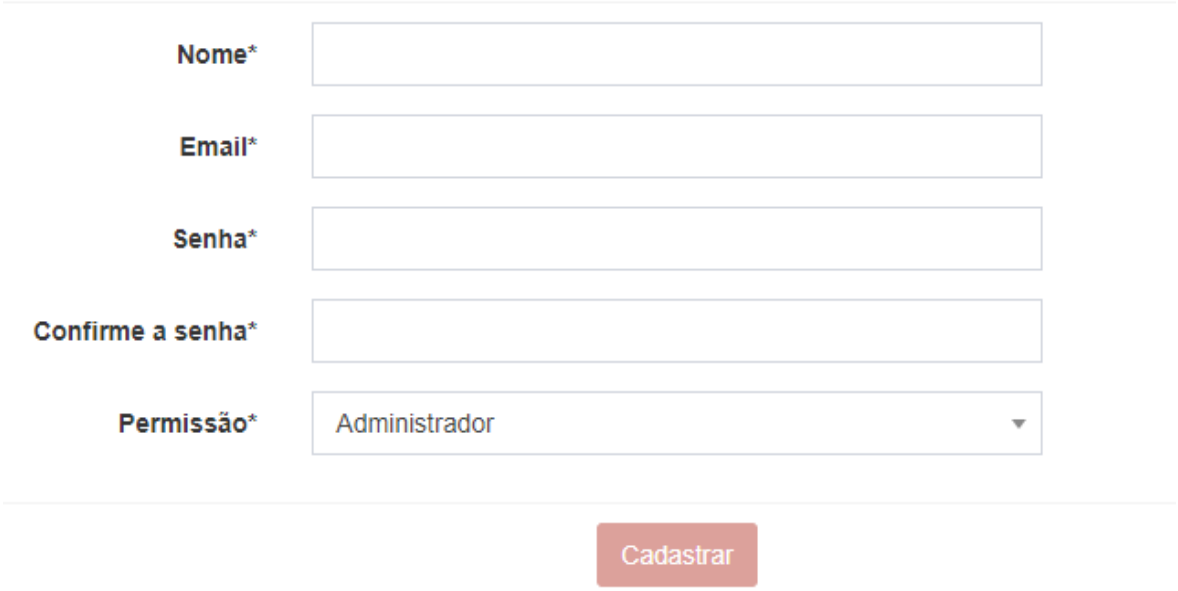
No canto inferior direito existe um botão flutuante, com a função de abrir o formulário de cadastro de um novo cliente.

Usuários

Assim como o menu descrito no item anterior, o de usuário possui as mesmas opções: novo e consulta. Lembrando que essa tela é exclusiva para administradores do sistema.

É nessa tela que os administradores cadastram e alteram usuários. A figura a seguir exibe o formulário para cadastro de um novo usuário. As regras de campos obrigatórios são as mesmas do item anterior.

Figura 42. Formulário de novo usuário



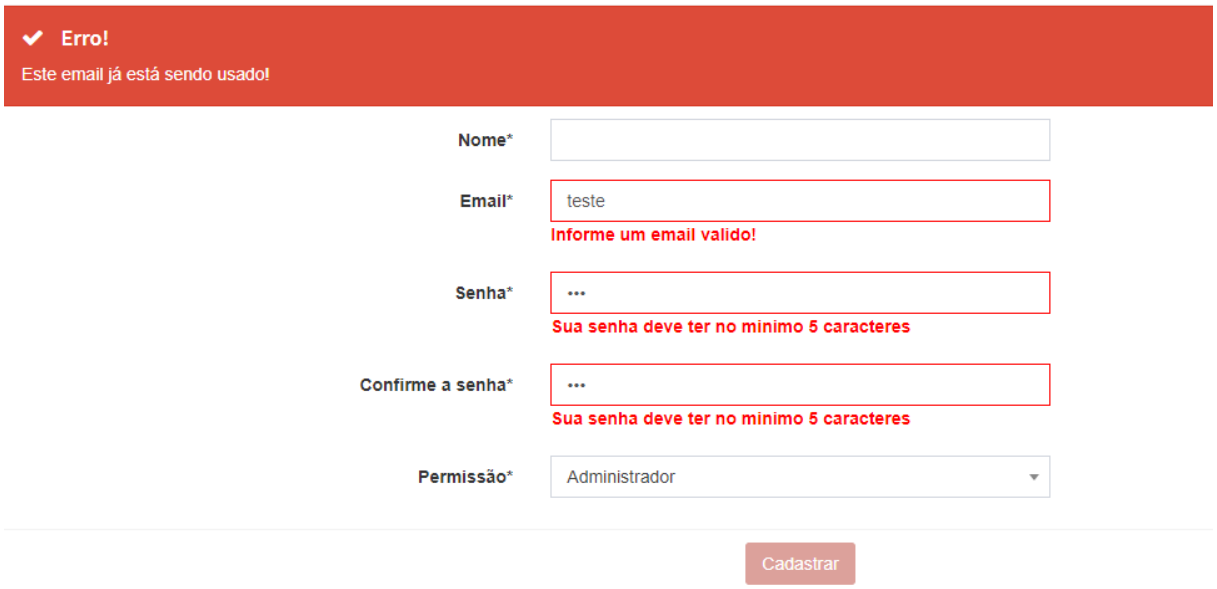
Fonte: Autoria própria.

Campos e validações

- **Nome**: Este campo não aceita caracteres numéricos.
- **E-mail**: Validação baseada no caractere “@”. O mesmo é checado no banco de dados, para garantir que não existam dois usuários com o mesmo e-mail.
- **Senha**: O campo deve possuir pelo menos cinco caracteres.
- **Confirme a senha**: Este campo deve ter o mesmo conteúdo do campo senha.
- **Permissão**: O campo possui duas opções, “Administrador” e “Usuário”, como descrito no capítulo dos requisitos funcionais.

A figura a seguir mostra exemplos das validações citadas acima.

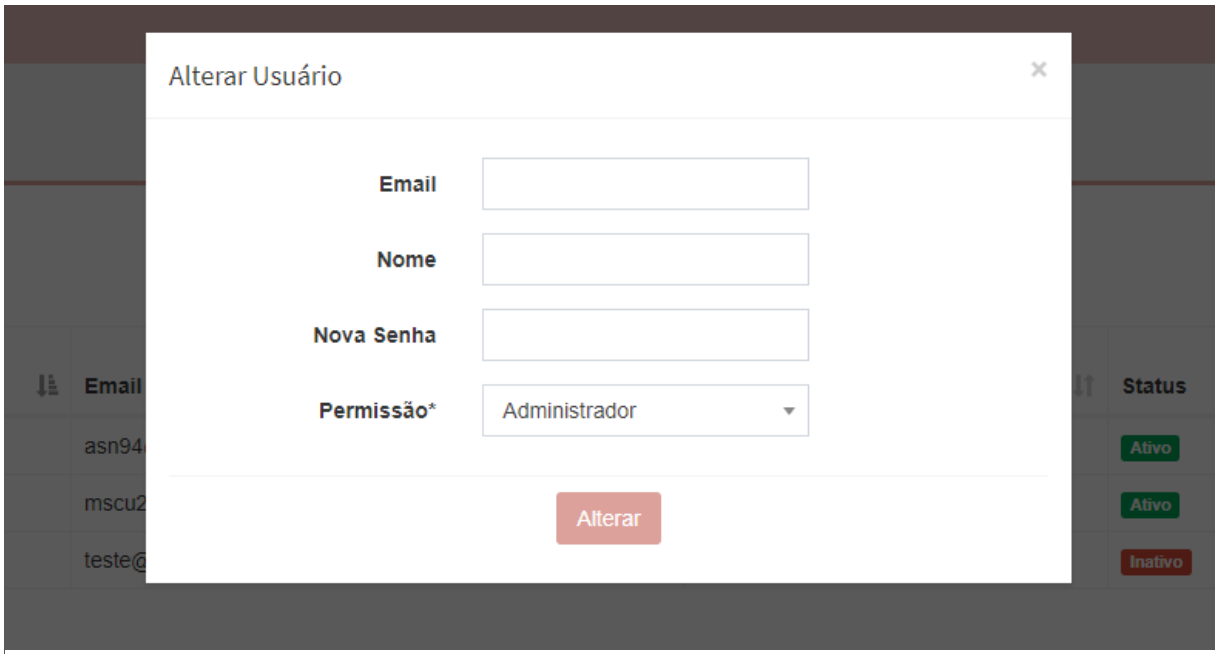
Figura 43. Validações no cadastro de usuários



Fonte: Autoria própria.

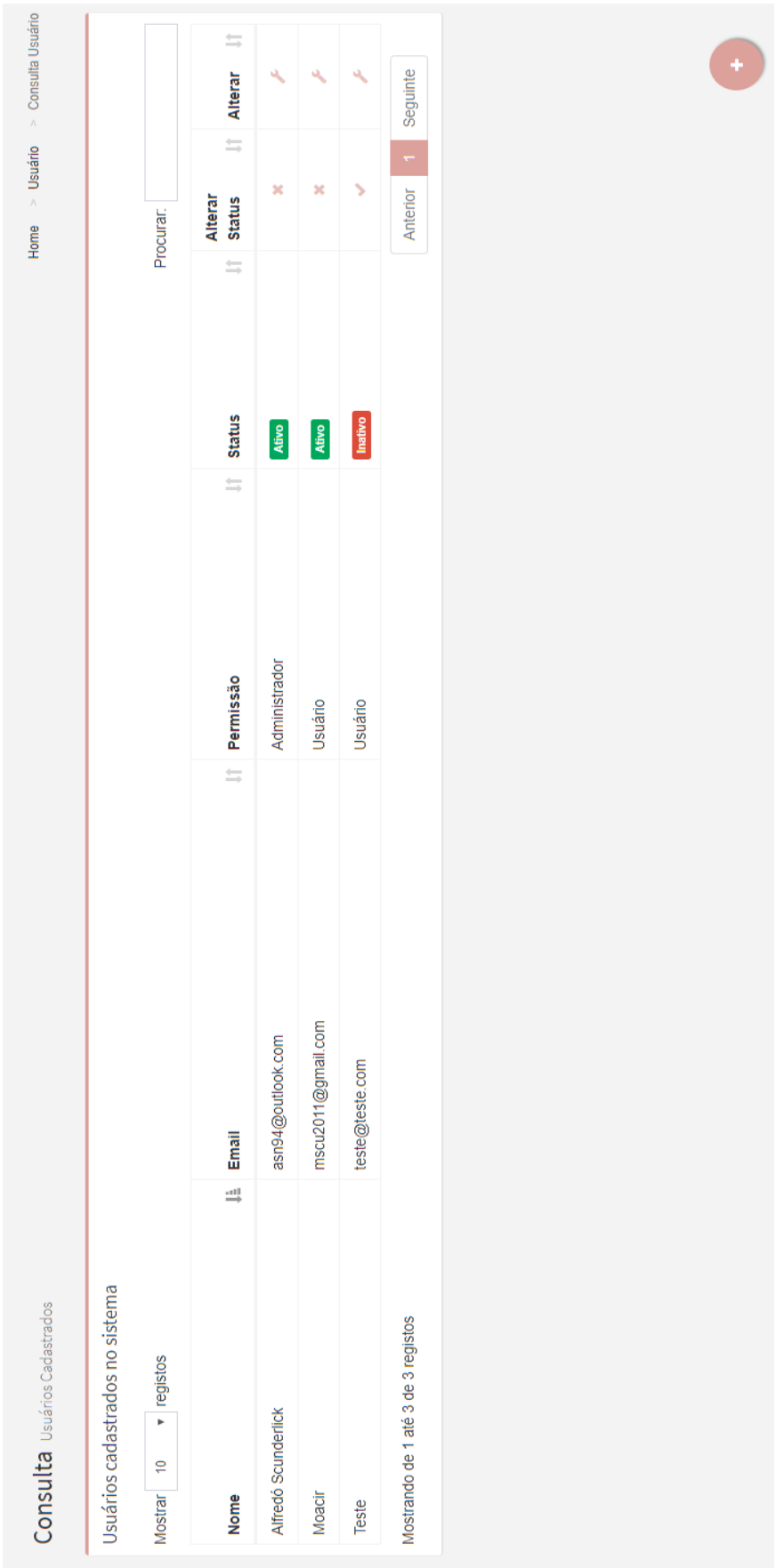
A tela de consulta de usuários segue o mesmo padrão da tela apresentada anteriormente, com o botão flutuante e a possibilidade de filtragem das informações. O que difere é a operação “Alterar Status”, que ao ser clicada, inativa ou ativa o usuário. Caso o usuário seja inativado, esse perde o acesso ao sistema. Essa operação é feita através do uso da tecnologia Ajax.

Figura 44. Tela de alteração de usuário



Fonte: Autoria própria.

Figura 45. Tela de consulta de usuários



Fonte: Autoria própria.

Endereços

Não existe um menu para endereços. Os mesmos podem ser cadastrados e alterados através das telas de cadastro de usuários e projetos, como mostra a figura abaixo.

Figura 46. Campo endereço

Fonte: Autoria própria.

Ao clicar no ícone a esquerda, um *modal* para cadastrar um novo endereço é exibido.

Figura 47. Tela de cadastro de endereços

Fonte: Autoria própria.

Dos campos presentes neste formulário, nenhum é obrigatório. Para que ele seja valido, basta o usuário preencher pelo menos uma das informações.

Quando o campo CEP é informado e validado, uma requisição Ajax é feita para o *web service* ViaCEP, que traz as informações rua, bairro, cidade e estado, e as coloca no formulário através da ferramenta Jquery.

O cadastro do endereço também é realizado por Ajax. Ao se dar por confirmada a inserção do novo endereço, essa opção é automaticamente adicionada no *combo box*.

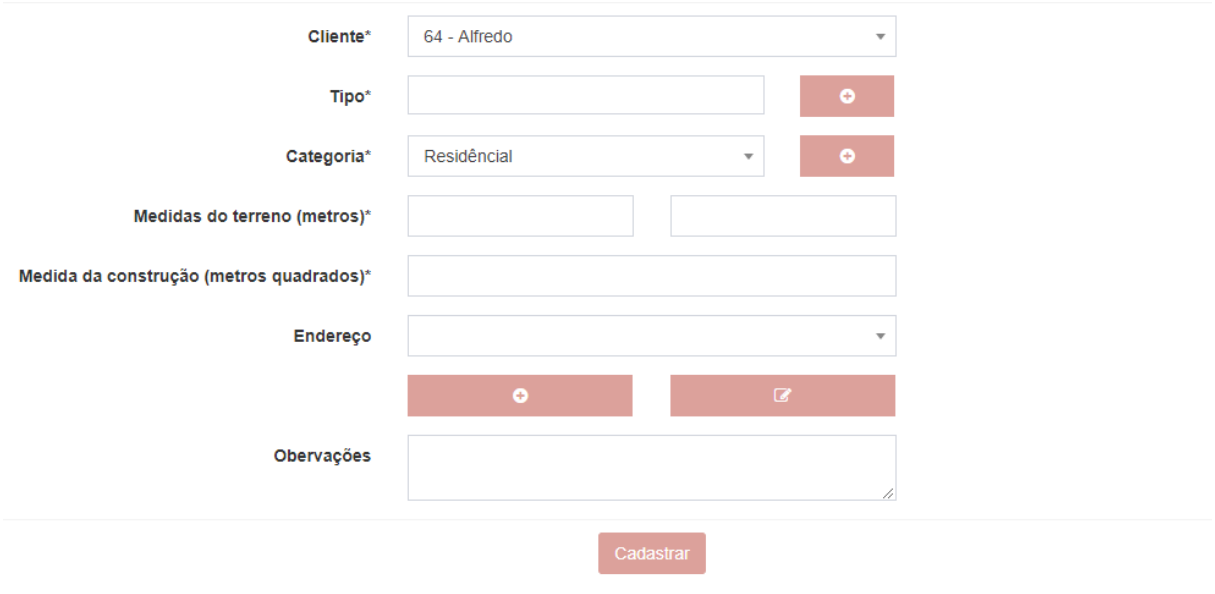
Para realizar a alteração de um endereço basta selecioná-lo e clicar no botão a direita. O mesmo *modal* será exibido, mas com os campos preenchidos.

Projetos

Esta é a tela mais relevantes do sistema, onde os usuários cadastram novos projetos e controlam o andamento de cada um deles.

O cadastro, assim como no cadastro de clientes, possui o campo endereço, que permite a adição de novos endereços caso necessário. Além desse campo, existem mais dois campos que funcionam da mesma forma, “Tipo” e “Categoria”. Ambos possuem seus próprios menus, mas é possível adicionar novas opções a esses campos no próprio cadastro de projetos.

Figura 48. Tela de cadastro de projetos

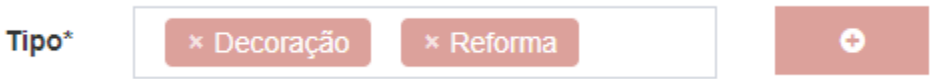


Fonte: Autoria própria.

Campos e validações

- **Cliente**: Contém todos os clientes primários cadastrados no sistema. Este *combo box,* assim como todos os outros, possui um filtro para conveniência do usuário.
- **Tipo**: O banco de dados base terá por padrão quatro tipos já cadastrados: decoração, reforma, nova construção e acompanhamento de obras. Como dito anteriormente, é possível a adição de novos tipos, seja pelo menu “Tipo Projeto” ou pelo formulário de novo projeto. O cadastro de tipos não permite tipos com o mesmo nome. Cada projeto pode ter mais de um tipo, como mostra a figura a seguir.

Figura 49. Campo tipo



Fonte: Autoria própria.

- **Categoria**: Da mesma forma que o campo tipo, este campo terá três opções padrão já cadastradas no banco de dados: residencial, comercial e industrial. O cadastro de categorias segue o mesmo padrão descrito no campo anterior, porém apenas uma categoria pode ser selecionada.
- **Medidas do terreno**: Aceita apenas valores numéricos.
- **Medida da construção**: Aceita apenas valores numéricos.
- **Endereço**: Funciona conforme explicado na página 56.
- **Observações**: Caso o arquiteto deseje fazer alguma anotação, esse é o espaço para isso.

A consulta de projetos segue o mesmo padrão das outras telas de consulta apresentadas anteriormente. A grande diferença é o botão “Mais Info.”, que leva o usuário a tela de detalhes do projeto, onde o mesmo é capaz de fazer alterações em cada uma das fases do projeto. Essa tela se encontra na figura a seguir.

Figura 50. Tela de consulta de projetos

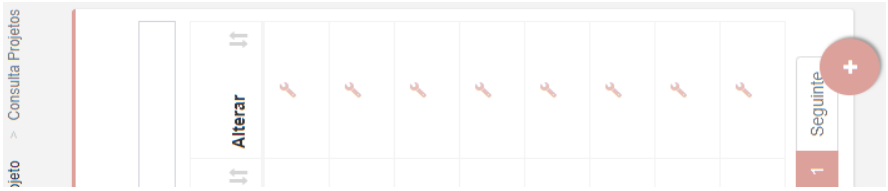
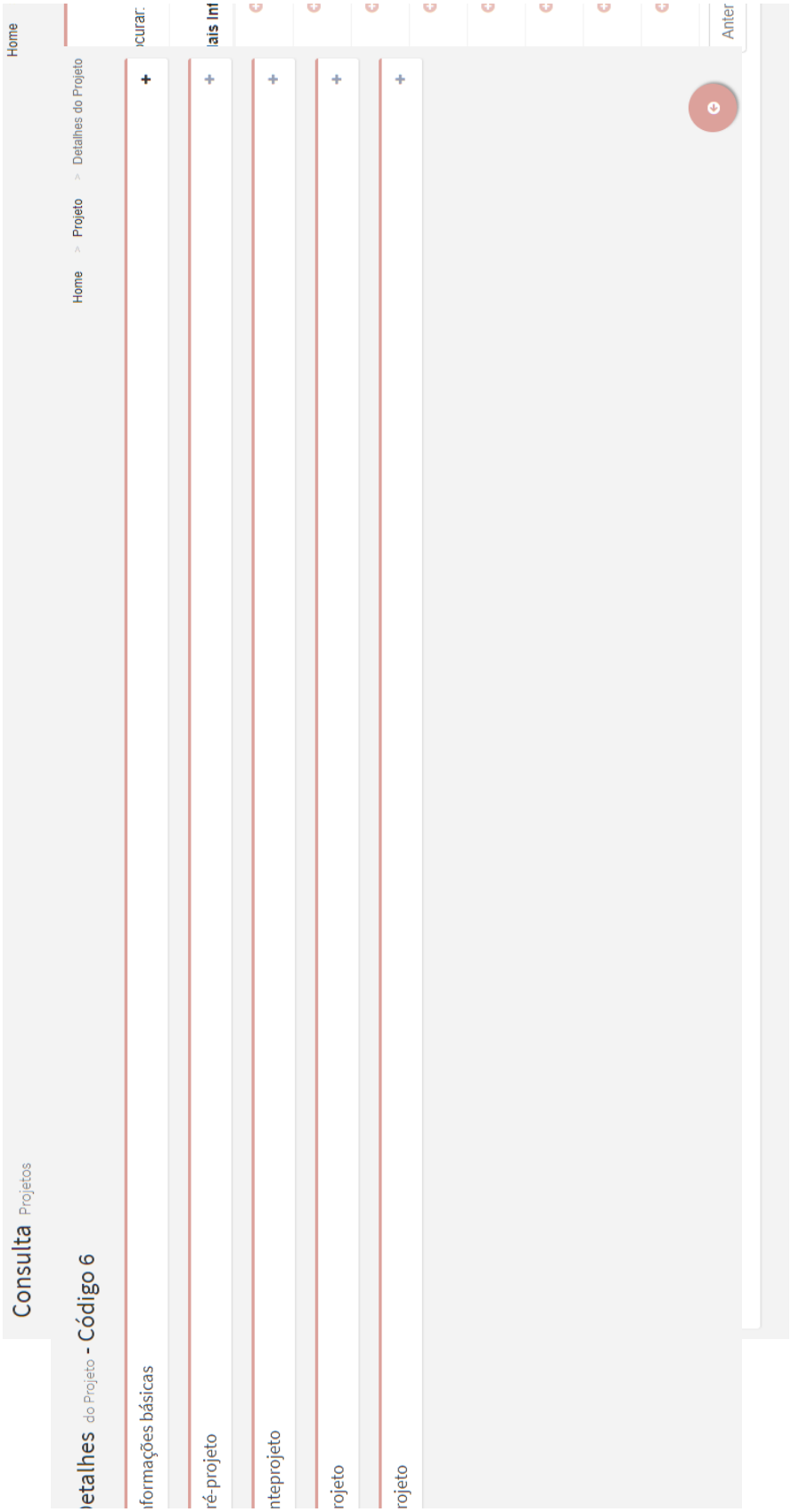


Figura 51. Tela de detalhes do projeto



Fonte: Autoria própria.

A figura anterior mostra a tela de detalhes do projeto, que é dividida em cinco partes que, inicialmente aparecem retraídas. O usuário é quem escolhe qual dessas partes deseja visualizar. As partes pré-projeto, anteprojeto e projeto são compostas por fases, essas compostas por quatro campos: prazo, status, último responsável e data de alteração. Os dois primeiros podem ser alterados pelos usuários, enquanto os outros dois são automaticamente atualizados através de requisição Ajax, quando a fase sofre alguma mudança.

O campo status é o responsável pela realização do cálculo que monta a coluna progresso. Ele pega todas as fases que estão concluídas e divide pelas fases que não tem o status “Não se aplica”. Essas são ignoradas. Quando o projeto chega a marca de 100% de progresso, seu status é alterado para concluído e é gravada a data dessa conclusão.

A seguir será mostrado a composição de cada uma das partes presentes na tela de detalhes.

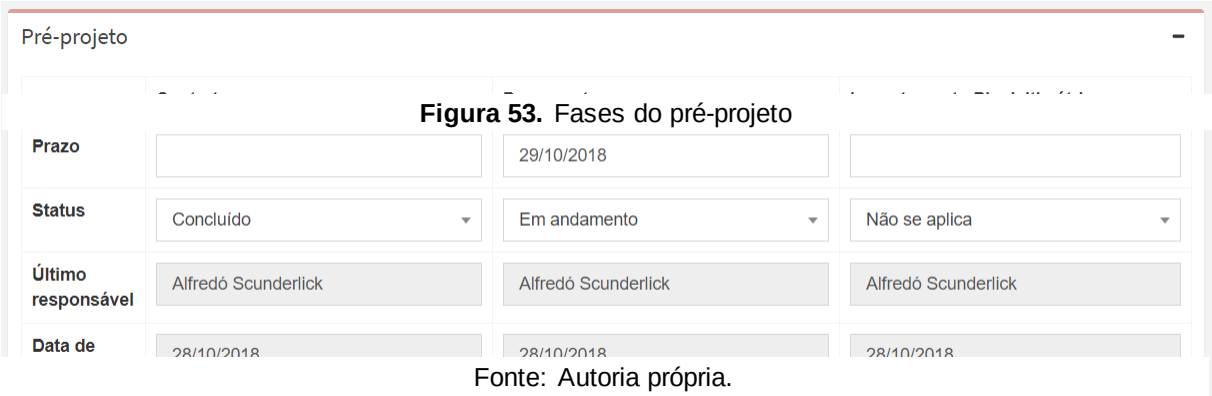
- **Informações básicas**: Consiste na apresentação dos dados presentes no cadastro do projeto. Esses não podem ser alterados nesta tela.

Figura 52. Informações básicas do projeto

Fonte: Autoria própria.

- **Pré-projeto**: Parte inicial do desenvolvimento de um projeto. Essa é

Figura 53. Fases do pré-projeto

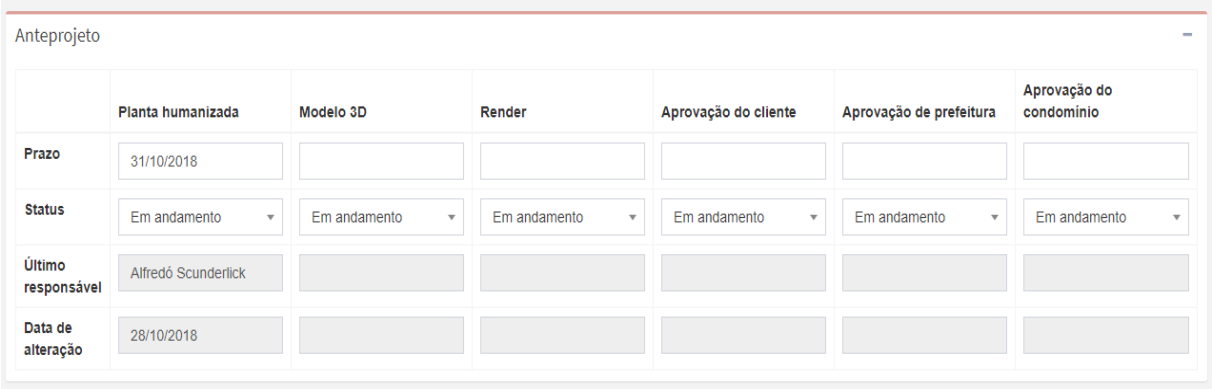


Fonte: Autoria própria.

composta por três fases: contrato, documentos e levantamento planialtimétrico.

- **Anteprojeto**: Parte intermediaria, composta por seis fases: planta humanizada, modelo 3D, render, aprovação do cliente, aprovação de prefeitura, aprovação do condomínio.

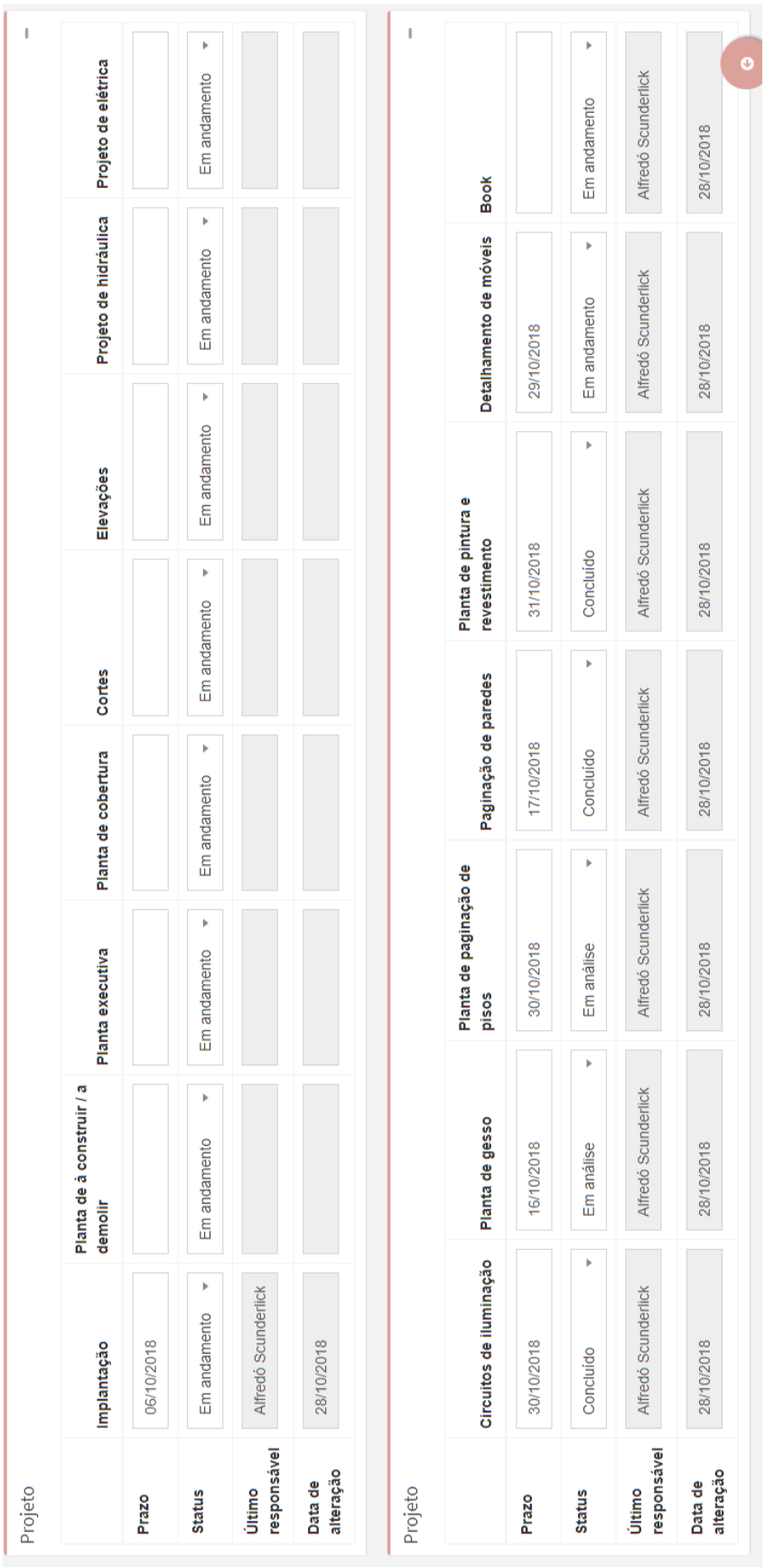
Figura 54. Fases do anteprojeto



Fonte: Autoria própria.

- **Projeto**: Parte final, composta por quinze fases. Devido a quantidade de campos, a tabela foi dividida em duas partes para evitar o uso de *scroll*, facilitando a visualização do usuário.

Figura 55. Fases finais do projeto



Fonte: Autoria própria.

Log de projetos

Como descrito no capítulo dos requisitos funcionais, o sistema possui uma tela que possibilita os administradores visualizarem as operações realizadas nos projetos cadastrados, desde inserções e alterações dos dados básicos, até alterações de fases. Esses *logs* são salvos automaticamente quando uma dessas operações é realizada por qualquer que seja o usuário.

Como citado acima, existem três tipos de *logs*, cada um deles é composto por:

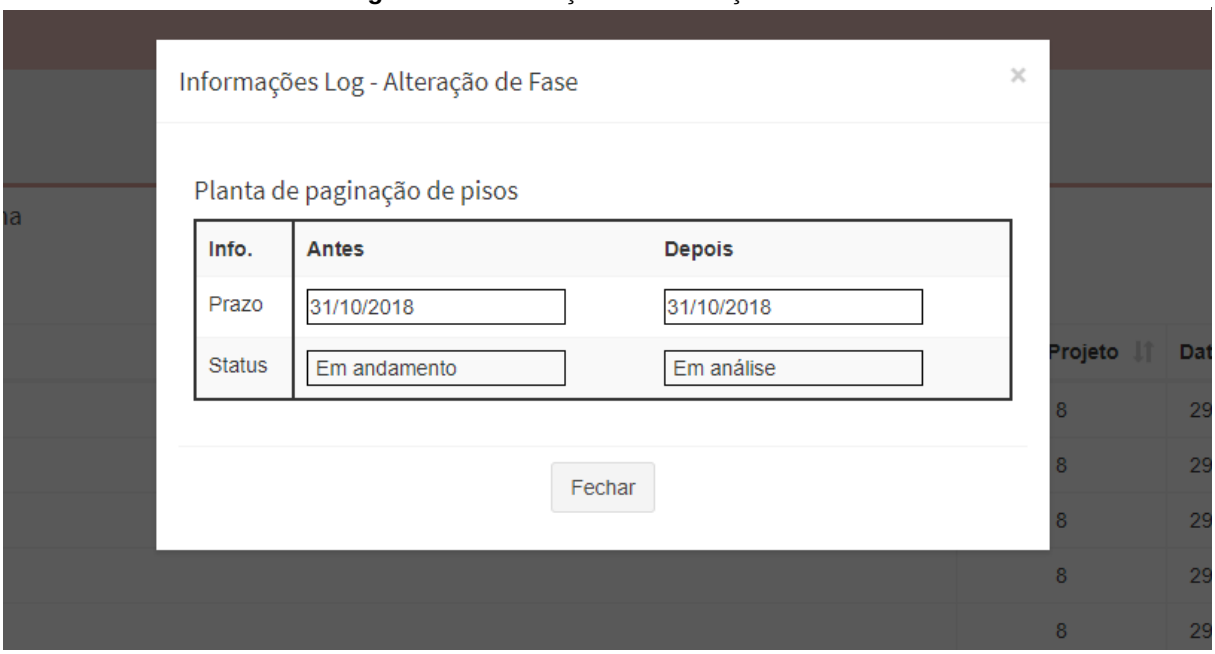
- **Usuário**: Quem foi o responsável pela operação.
- **Código do projeto:** Qual projeto foi cadastrado/alterado.
- **Data:** Quando essa operação foi realizada.
- **Transação:** O tipo do *log.* São eles:
  - **Cadastro:** Novo projeto.
  - **Alteração de dados básicos:** Mudanças feitas nos dados básicos do projeto, como as medidas do terreno, medida da construção, observações e status.
  - **Alteração de fases:** Mudanças feitas nos dados de alguma fase de um projeto, como o prazo e o status.
- **Antes:** Como estavam as informações antes da alteração.
- **Depois:** Como ficaram as informações depois da alteração.

A tela de *logs* é composta por uma tabela, que segue o mesmo padrão das demais presentes no sistema. Para visualizar as informações que foram alteradas, basta clicar no ícone presente na coluna “Mais Info.”, como mostram as figuras a seguir.

Figura 56. Consulta de logs

Fonte: Autoria própria.

Figura 57. *Informações da alteração de fase*



Fonte: Autoria própria.

3.7. Infraestrutura de tecnologia da informação

Necessidade de infraestrutura de redes

É necessária uma rede local, seja ela cabeada ou *wi-fi,* caso seja definido o uso de um servidor físico.

Necessidade de link de internet

A única funcionalidade do sistema que depende de conexão com a internet é a busca pelo cep, com a utilização do *web service* ViaCep. Caso não exista esse link, essa funcionalidade retorna um erro, mas não impossibilita a utilização do sistema, pois ainda é possível informar manualmente os dados do endereço.

Se futuramente for escolhido hospedar o servidor na nuvem, torna-se necessário a aquisição de um serviço de internet.

Necessidade de aquisições de licenças de softwares

Não há necessidade da aquisição de nenhuma licença de *software*, tendo em vista que é possível hospedar e rodar o sistema com *softwares* livres. A única coisa que é aconselhada é a utilização de navegadores testados previamente nesse trabalho, como o Google Chrome ou o Microsoft Edge (exclusivo para Windows 10 nos desktops). Então, se o usuário preferir o navegador da Microsoft, o mesmo terá que adquirir uma licença do Windows 10.

4. CONSIDERAÇÕES FINAIS

O projeto atingiu os resultados esperados, baseados nos requisitos levantados no início do processo, mas esse trabalho é apenas o começo do desenvolvimento desse sistema. Junto a arquiteta Gabriela De Nadai, pretendemos continuar aperfeiçoando o sistema. Inclusive possuímos uma lista das melhorias que esperamos implementar num futuro próximo.

O objetivo é ter o maior número de funcionalidades possível para atender ao máximo as necessidades de um escritório de arquitetura, possibilitando uma futura comercialização desse sistema.

Por enquanto já foram definidas as seguintes melhorias, são elas:

- **Tela de mudança de senha**: Atualmente apenas um administrador pode mudar a senha de outro usuário. A ideia é criar uma dela para que qualquer usuário possa trocar sua senha.
- **Controle de horas/minutos**: Adicionar um atributo nos projetos para que os usuários coloquem as horas ou minutos utilizados em cada fase. Dessa forma será possível calcular quanto tempo foi necessário para concluir o projeto como um todo, possibilitando definir melhor o valor do serviço prestado.
- **Escolher quais fases irão constituir um projeto**: Permitir que os usuários escolham quais fases o projeto terá. Dessa forma a tela de detalhes ficará mais objetiva, sem a necessidade de mudar o status para “Não se aplica” quando uma fase não for relevante para um projeto.
- **Orçamentos**: Montar orçamentos de decoração, reforma e outros tipos de projeto. Com a possibilidade de gerar um relatório para ser repassado ao cliente.
- **Agenda de reuniões**: Possibilidade de cadastro de reuniões e tarefas a serem realizadas. Possivelmente na forma de um calendário interativo ou de uma tabela como as outras já presentes no sistema.

Muito foi aprendido no processo de desenvolvimento deste trabalho. É possível afirmar que o sistema supre as necessidades iniciais apresentadas pela

cliente, automatizando tarefas que antes eram feitas de maneira manual e muitas vezes sem qualquer tipo de controle.

5. REFERÊNCIAS

PRESSMAN, Roger S **Engenharia de Software**. 5 ed. Rio de Janeiro: McGraw-Hill. 2002.

ELMASRI, Ramez; NAVATHE, Shamkant B. **Sistemas de Banco de Dados**. 4 ed. São Paulo: Pearson Addison Wesley. 2005.

LARMAN, Craig. **Utilizando UML e padrões**. 2 ed. Porto Alegre: Bookman. 2004.

CAELUM. **Java para Desenvolvimento Web**. Disponível em: < https://www.caelum.com.br/apostila-java-web/>. Acesso em: 20 jun. 2018.

CAELUM. **Desenvolvimento Web com HTML, CSS e JavaScript**. Disponível em: <https://www.caelum.com.br/apostila-html-css-javascript/>. Acesso em: 30 jun. 2018.

CAELUM. **Java e Orientação a Objetos**. Disponível em: <https://www.caelum.com.br/apostila-java-orientacao-objetos/>. Acesso em: 05 jul. 2018.

VIACEP. **Exemplo com Jquery**. Disponível em: < https://viacep.com.br/exemplo/jquery/>. Acesso em: 03 ago. 2018.

ALURA. **Entendendo a Web por baixo dos panos**. Disponível em: <https://cursos.alura.com.br/course/http-fundamentos>. Acesso em: 20 jun. 2018.

ALURA. **HTML5 e CSS3: Suas primeiras páginas da Web**. Disponível em: <https://cursos.alura.com.br/course/introducao-html-css>. Acesso em: 20 jun. 2018.

ALURA. **Fundamentos de Java na Web**. Disponível em: <https://cursos.alura.com.br/course/servlet-3-e-fundamentos-web>. Acesso em: 20 jun. 2018.

ALURA. **Tags para facilitar o desenvolvimento JSP**. Disponível em: <https://cursos.alura.com.br/course/jstl>. Acesso em: 15 jul. 2018.

ALURA. **SpringMVC: Criando aplicações Web**. Disponível em: <https://cursos.alura.com.br/course/spring-mvc-1-criando-aplicacoes-web>. Acesso em: 15 jul. 2018.

ALURA. **SpringMVC: Integração, cache, segurança e templates**. Disponível em: <https://cursos.alura.com.br/course/springmvc-2-integracao-cache-seguranca-e-templates>. Acesso em: 01 ago. 2018.

MIHALCEA, Vlad. **The best way to map a @OneToMany relationship with JPA and Hibernate**. Disponível em: <https://vladmihalcea.com/the-best-way-to-map-a-onetomany-association-with-jpa-and-hibernate/>. Acesso em: 05 ago. 2018.

DEVMEDIA. **Ajax com Jquery: Trabalhando com requisições assíncronas**. Disponível em: <https://www.devmedia.com.br/ajax-com-jquery-trabalhando-com-requisicoes-assincronas/37141>. Acesso em: 30 jul. 2018.

DEVMEDIA. **Introdução ao Padrão MVC**. Disponível em: <https://www.devmedia.com.br/introducao-ao-padrao-mvc/29308>. Acesso em: 10 set. 2018.

DEVMEDIA. **JSON Tutorial**. Disponível em: <https://www.devmedia.com.br/json-tutorial/25275>. Acesso em: 10 set. 2018.

APÊNDICE A – ATA DE REUNIÃO 01/2018

ATA DE REUNIÃO
n: 01/2018

Data: 18/06/2018
Horário de Início: 20:35
Horário de Termino: 22:00
Local: Rua Maria Regina Delben Astorri, 54
Participantes: Alfredo Scunderlick Neto – Programador Júnior
Gabriela De Nadai – Arquiteta
Pauta: Situação atual da gestão dos projetos, principais funcionalidades.

Situação atual da gestão dos projetos: Discutido como é feito a gestão dos projetos na empresa em que Gabriela De Nadai trabalha. Houve a exibição da planilha utilizada pelos funcionários e uma explicação detalhada de como funciona o processo de execução de um projeto arquitetônico.
Principais funcionalidades: Definidos os principais requisitos para o sistema, incluindo quais telas seriam necessárias, seus campos, suas validações e aparências.
Decidido que, a toda semana, a cliente será apresentada com o andamento do desenvolvimento do sistema, incluindo a utilização do mesmo, afim de alinhar seus desejos com o que está sendo feito.

APÊNDICE B – ATA DE REUNIÃO 02/2018

ATA DE REUNIÃO
n: 02/2018

Data: 30/07/2018
Horário de Início: 19:00
Horário de Termino: 20:00
Local: Rua Maria Regina Delben Astorri, 54
Participantes: Alfredo Scunderlick Neto – Programador Júnior
Gabriela De Nadai – Arquiteta
Pautas: Apresentação do protótipo, dúvidas e definição das próximas etapas.

Apresentação do protótipo: Exibido para a cliente algumas das telas desenvolvidas, muitas, sem suas principais funcionalidades, feitas apenas para aprovação dos layouts.
Decidido a mudanças das cores principais, reorganização das informações nas telas de cadastro.

Dúvidas e definição das próximas etapas: Tiradas as dúvidas com relação aos campos que deveriam compor cada uma das telas de cadastro e suas validações. A arquiteta apresentou esquemas da tela de cadastro de projetos e da tela das fases de cada um deles para a próxima etapa do desenvolvimento do sistema.

APÊNDICE C – ATA DE REUNIÃO 03/2018

ATA DE REUNIÃO
n: 03/2018

Data: 06/09/2018
Horário de Início: 20:00
Horário de Termino: 21:20
Local: Rua Maria Regina Delben Astorri, 54
Participantes: Alfredo Scunderlick Neto – Programador Júnior
Gabriela De Nadai – Arquiteta
Pauta: Tipos e categorias de projeto, tela das fases do projeto.

Tipos e categorias de projeto: Discutido sobre a necessidade de uma tela de cadastro para novas categorias e tipos de projeto.
Decidido como seriam essas telas.

Tela das fases do projeto: Discutido a melhor forma para dividir as fases do projeto em uma só tela.
Decidido o layout final para essa tela, dividindo o projeto final em duas partes.