UNIVERSIDADE FEDERAL DO CEARÁ

CAMPUS SOBRAL

CURSO DE GRADUAÇÃO EM ENGENHARIA DE COMPUTAÇÃO

VITORIA NOBRE LIMA

UM RELATO DE USO DOS FRAMEWORKS RUBY ON RAILS, LARAVEL E DJANGO

SOBRAL

2025

VITORIA NOBRE LIMA

UM RELATO DE USO DOS FRAMEWORKS RUBY ON RAILS, LARAVEL E DJANGO

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Computação do Campus Sobral da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Engenharia de Computação.

Orientador: Prof. Dr. Fischer Jônatas Ferreira.

SOBRAL

2025

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Sistema de Bibliotecas
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

L711r Lima, Vitoria Nobre.
Um relato de uso dos frameworks Ruby on Rails, Laravel e Django / Vitoria Nobre Lima. – 2025.
123 f. : il. color.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Campus de Sobral, Curso de Engenharia da Computação, Sobral, 2025.
Orientação: Prof. Dr. Fischer Jônatas Ferreira.

1. Framework. 2. Ruby on Rails. 3. Laravel. 4. Django. 5. Atividades de extensão. I. Título.
CDD 621.39

VITORIA NOBRE LIMA

UM RELATO DE USO DOS FRAMEWORKS RUBY ON RAILS, LARAVEL E DJANGO

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Computação do Campus Sobral da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Engenharia de Computação.

Aprovada em: 25 de fevereiro de 2025

BANCA EXAMINADORA

Prof. Dr. Fischer Jônatas Ferreira (Orientador)
Universidade Federal do Ceará - UFC

Prof. Dr. Carlos Elmano de Alencar e Silva
Universidade Federal do Ceará - UFC

Prof. Dr. Evilásio Costa Júnior
Universidade Federal do Ceará - UFC

À minha mãe, minha primeira professora, que com o giz na mão ensinou não apenas seus alunos, mas também a mim. Ao meu pai que, sob o sol, plantou a semente da árvore que hoje me oferece a sombra. Aos que, mesmo distantes, estiveram presentes através de suas orações.

AGRADECIMENTOS

Agradeço primeiramente a Deus, por suas bênçãos na minha vida e por ter me sustentado ao longo dessa trajetória, e pela sabedoria e oportunidades concedidas a mim.

Agradeço, ainda, à minha mãe Tetilda, ao meu pai José, à minha tia Zulene e aos meus primos, pelo apoio durante todos os momentos, e, principalmente, pelos joelhos dobrados em meu favor. Vocês foram minha base forte para prosseguir.

Aos meus amigos, Emerson Sousa, Gabriel Santiago e Yara Morais pelo apoio e incentivo durante todo o tempo dedicado não somente a este trabalho, como também durante todo o curso. Foram longos anos de lutas, mas ter vocês por perto foi muito importante.

Aos meus colegas de curso por todo o companheirismo durante as disciplinas que cursamos juntos.

Estendo meus agradecimentos a todos os professores que tive durante esses anos de estudo, e em especial ao Prof. Dr. Fischer Jônatas Ferreira, pela sua orientação, paciência e confiança.

Agradeço aos membros da banca, Prof. Dr. Carlos Elmano de Alencar e Silva e Prof. Dr. Evilásio Costa Júnior, pela disponibilidade e colaborações feitas neste trabalho.

À minha gestora de estágio Ana Isabelle pela sua compreensão com meus horários e pelo seu incentivo. Nunca esquecerei os aprendizados que me proporcionou e a confiança que depositou em mim.

Aos meus amigos do setor de Gestão AVA, que, de forma direta ou indireta, contribuíram para que esse momento se tornasse realidade.

RESUMO

Contexto: Durante o desenvolvimento de um projeto para *Web*, há de se compreender e analisar o domínio do problema. Nessa perspectiva, o desenvolvedor de aplicações para *Web* precisa dominar várias tecnologias e suas respectivas ferramentas, bibliotecas e *frameworks*. Contudo, devido à grande variedade de tipos de domínio de sistemas e restrições de projeto, não é fácil a escolha da tecnologia que melhor se adapte às especificações do projeto. Motivação: Contraditoriamente, há pouca literatura acadêmica destinada a fazer comparações entre essas tecnologias para auxiliar as escolhas dos desenvolvedores. Objetivo: O principal objetivo do presente trabalho é realizar um estudo comparativo entre *frameworks* que possibilitem que o *Front-end* e o *Back-end* de um sistema *Web* possam ser implementados juntos. Além disso, tem-se como propósito secundário o desenvolvimento de um *software* que facilite o controle das horas de atividade de extensão de uma Instituição de Ensino Superior (IES). Metodologia: Para alcançar o objetivo principal do trabalho, realizou-se a comparação dos seguintes *frameworks Web*: Ruby on Rails, Laravel e Django, a partir da implementação de um conjunto de funcionalidades em cada um dos *frameworks*, seguido da análise dos *frameworks* a partir de pontos de comparação. Além disso, para alcançar o objetivo secundário, foi projetado e realizada a prototipagem de um *software* de controle das horas de atividade de extensão. Resultados: Como resultado deste trabalho, foram implementadas e comparadas as mesmas funcionalidades utilizando os *frameworks* Ruby on Rails, Laravel e Django, permitindo uma análise de cada um dos *frameworks*. Ademais, foi realizada a prototipagem do *software* de controle das horas de atividade de extensão, que visa beneficiar os docentes e discentes de uma IES. Por fim, foram documentados os pontos de comparação e as lições aprendidas, colaborando, assim, com uma nova perspectiva sobre essas tecnologias, que auxilie os desenvolvedores de aplicações *Web* na difícil tomada de decisão de qual ferramenta utilizar em cada projeto.

Palavras-chave: Framework, Web, Ruby on Rails, Laravel, Django, Software, Atividades de extensão

ABSTRACT

`Context`: During the development of a project for the Web, the problem domain must be understood and analyzed. From this perspective, the Web application developer must master several technologies and their respective tools, libraries, and frameworks. However, due to the wide variety of system domain types and project constraints, choosing the technology that best adapts to the project specifications is not easy. `Motivation`: Contradictorily, little academic literature is designed to compare these technologies to assist developers' choices. `Objective`: The main objective of this work is to carry out a comparative study between frameworks that enable the Front-end and Back-end of a Web system to be implemented together. Furthermore, the secondary purpose is to develop software that facilitates the control of hours of extension activity at a Higher Education Institution (HEI). `Methodology`: This comparison is based on the implementation of a set of functionalities in each framework, followed by an analysis of the framework based on comparison points. Furthermore, to meet the secondary objective, software to control hours of extension activity was designed and prototyped. `Results`: As a result of this work, the same functionalities were implemented and compared using Ruby on Rails, Laravel, and Django, allowing for an in-depth analysis of each framework. Additionally, a prototype of the extension activity hour management software was developed, aiming to benefit teachers and students at HEIs. Finally, the comparison points and lessons learned were documented, providing a new perspective on these technologies and assisting Web application developers in the difficult decision-making process when selecting the most suitable tool for each project.

Keywords: Framework, Web, Ruby on Rails, Laravel, Django, Software, Extension activities

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Considerando o grande avanço tecnológico atual, o uso de sistemas computacionais nas mais diversas áreas e finalidades tornou-se corriqueiro, se não obrigatório (MILETTO; BERTAGNOLLI, 2014), repercutindo em uma demanda tecnológica por parte de pessoas, de empresas e de entidades governamentais maior a cada ano, tanto em volume quanto em complexidade (PRESSMAN; MAXIM, 2021). Dessa forma, hoje as aplicações Web são parte integrante da vida cotidiana, impactando na forma como os desenvolvedores projetam, desenvolvem, mantêm e gerenciam aplicações. (PANWAR, 2024).

Uma etapa crucial durante o desenvolvimento de um projeto para *Web* é a escolha de tecnologias a serem utilizadas que sejam adequadas às especificidades do domínio da aplicação, pois uma escolha errada pode acarretar na falha do projeto, assim devem ser considerados vários aspectos da tecnologia para que a aplicação cumpra os requisitos do projeto. Nessa perspectiva, o desenvolvedor *Web* dispõe de diversas tecnologias e suas respectivas ferramentas, bibliotecas e *frameworks*, entre outros recursos possíveis, que se unem a uma grande variedade de tipos de domínios de sistemas e restrições de projeto, tornando a escolha não trivial (ANJUM; ALAM, 2019).

Paradoxalmente, há poucos trabalhos acadêmicos destinados a fazer comparações entre as tecnologias disponíveis a fim de auxiliar as escolhas dos desenvolvedores. Diante dessa escassez, o principal objetivo do presente trabalho é realizar um estudo comparativo entre *frameworks* para desenvolvimento de sistemas baseados na *Web*, com foco naqueles que permitem que o *Front-end* e o *Back-end* possam ser implementados juntos. Nesse sentido, a escolha dos *frameworks* `Ruby on Rails`, `Laravel` e `Django` para este estudo comparativo se justifica por possuírem arquitetura e abordagem semelhantes. O `Ruby on Rails` foi projetado para compactar a complexidade de aplicações *Web* modernas (RAILS, 2024). Enquanto o `Laravel` se propõe a entregar projetos de qualidade, minimizando a escrita de código manualmente (LARAVEL, 2023). Já o `Django` foi projetado para facilitar a criação de aplicações Web de modo a concluir projetos com mais rapidez e menos código (DJANGO, 2023).

Ademais, espera-se que os resultados obtidos a partir desse estudo comparativo possam auxiliar os desenvolvedores na escolha da tecnologia mais adequada às especificações do projeto, inclusive ampliando a visão dos desenvolvedores sobre esses *frameworks*. Além disso, tem-se como propósito secundário o desenvolvimento de um *software* que facilite o controle das horas de atividade de extensão de uma Instituição de Ensino Superior (IES).

A metodologia utilizada para alcançar o objetivo principal do trabalho foi desenvolver as mesmas funcionalidades utilizando três populares *frameworks* para desenvolvimento Web: `Ruby on Rails`, `Laravel` e `Django`, permitindo a observação das vantagens e desvantagens de cada um deles e a documentação das lições aprendidas desde a instalação até a finalização da implementação das funcionalidades. Já para a obtenção do objetivo secundário, a metodologia consistiu em: levantar os requisitos do *software* de controle de horas de extensão e prototipagem do sistema.

Os resultados obtidos a partir da realização deste trabalho foram, além da documentação das lições aprendidas a partir dos critérios de comparação entre os *frameworks*, a prototipagem do *software* de extensão, desenvolvendo cada tela do sistema. A realização da comparação visa beneficiar os desenvolvedores, já a modelagem do *software* visa beneficiar os docentes e os discentes de uma IES. Além disso, os projetos desenvolvidos ao longo deste trabalho estão disponíveis em repositório remoto através do endereço https://github.com/vitorianobre/projetos-monografia.

O restante do conteúdo está organizado nos demais capítulos da seguinte forma:

- Capítulo 2: apresenta uma visão geral sobre o `Ruby on Rails`, o `Laravel` e o `Django`, discutindo suas especificidades por meio de exemplos práticos e didáticos, além de uma visão sobre as atividades de extensão universitária, discutindo sua importância e objetivo;
- Capítulo 3: detalha a metodologia traçada para alcançar o objetivo principal e secundário presente na proposta;
- Capítulo 4: discute os resultados obtidos durante o processo de comparação entre os *frameworks*;
- Capítulo 5: apresenta a conclusão deste trabalho e sugestões de trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta uma fundamentação teórica sobre os *frameworks* `Django`, `Laravel` e `Ruby on Rails`, descrevendo especificidades e aspectos didáticos das atividades de extensão universitária, na seguinte ordem: a Seção 2.1 apresenta uma visão geral do `Django`, a Seção 2.2 apresenta a arquitetura do `Django`, a Seção 2.3 apresenta um exemplo prático de uso do `Django`, a Seção 2.4 apresenta uma visão geral do `Laravel`, a Seção 2.5 apresenta a arquitetura do `Laravel`, a Seção 2.6 apresenta um exemplo prático de uso do `Laravel`, a Seção 2.7 apresenta uma visão geral do `Ruby on Rails`, a Seção 2.8 apresenta a arquitetura do `Ruby on Rails`, a Seção 2.9 apresenta um exemplo prático de uso do `Ruby on Rails` e, por fim, a Seção 2.10 apresenta uma visão geral sobre as atividades de extensão universitária.

2.1 Visão geral do Framework Django

O `Django`[1] é um *framework Web* gratuito, de código aberto, escrito em `Python`[2], projetado para facilitar a criação de aplicações *Web* de modo a concluir os projetos com mais rapidez e menos código. Foi criado em 2003 por um time de desenvolvimento *Web*, inicialmente como um sistema gerenciador de *sites* jornalísticos. Após criar diversos *sites*, o time começou a reutilizar muitos de seus códigos comuns e padrões de *design*, resultando, em 2005, em um *framework* de desenvolvimento Web de código aberto (BENNETT, 2009) e consequente popularização nos anos seguintes. Dessa forma, o *framework* continuou sendo aprimorado desde a sua primeira versão, incorporando novas funcionalidades, que vão do suporte a diferentes tipos de bancos de dados até a adição de classes, angariando uma comunidade de voluntários com o objetivo de impulsionar o ecossistema de softwares relacionados ao `Django` (DJANGO, 2023).

Além disso, o `Django` se caracteriza por facilitar o rápido desenvolvimento de aplicações que possuem *design* limpo e pragmático, utilizando o conceito DRY (*Don't Repeat Yourself*), ou seja: não repetir código, aproveitando o que já foi feito e não duplicando para evitar redundância (MACIEL, 2020). É também altamente escalável, por ser rápido e flexível para atender às demandas de tráfego mais pesadas, versátil para criar diferentes tipos de aplicações, que vão desde sistemas de gerenciamento de conteúdo até redes sociais e plataformas de computação científica, e seguro, pois fornece um sistema de autenticação de usuário, gerenciamento de contas e senhas, administração de conteúdo e mapeamento do *site* (DJANGO, 2023).

[1] https://www.djangoproject.com/
[2] https://www.python.org/

Assim, para melhor compreender o Django é importante diferenciá-lo do Django REST Framework[3], porque embora ambos sejam *frameworks* escritos em Python, eles possuem diferentes finalidades. Enquanto o Django é um *framework* completo para desenvolvimento de aplicações *Web*, o Django REST Framework é uma biblioteca que se baseia no Django usada especificamente para criar *APIs REST*. Ainda que o Django tenha recursos para criar APIs, o Django REST Framework fornece funcionalidades específicas para facilitar a criação de *APIs REST* de forma eficiente e consistente. Sendo assim, o Django REST Framework funciona como um complemento do Django, ou seja, a instalação deste é requisito para se utilizar aquele (REST, 2024).

O ambiente de desenvolvimento Django é uma instalação necessária para começar a desenvolver e testar aplicações Django. Assim, as principais ferramentas fornecidas pelo *framework* são: um conjunto de *scripts* Python para criação dos projetos e um servidor *Web* local, executado via terminal, usado para testar localmente a aplicação durante o desenvolvimento. Contudo, o computador no qual o Django é executado deve possuir um ambiente de desenvolvimento Python que contenha, pelo menos, um interpretador e um sistema de gerenciamento de pacotes, que podem ser obtidos a partir de qualquer distribuição Python destinada ao Sistema Operacional (SO) utilizado pelo desenvolvedor. Instalados tais recursos, o próximo passo é a criação de um ambiente virtual através do módulo *virtualenv*, que permite isolar as dependências do projeto como um conjunto de pacotes independentes, por meio da execução do comando descrito no *Listing* 2.1.1 no *prompt* de comando do SO em uso.

```
1    pip install virtualenv
```

Listing 2.1.1 – Comando de instalação do módulo *virtualenv*

A seguir, ainda no *prompt* de comando, estando no diretório destinado ao projeto, executa-se o comando do *Listing* 2.1.2 para criar um ambiente virtual, neste caso, chamado *venv*. Dessa forma, após a criação do ambiente, é preciso ativá-lo com o comando mostrado no *Listing* 2.1.3 para que possam ser instalados os pacotes necessários ao projeto. Para desativar uma *virtualenv* utiliza-se o comando *deactivate*.

O próximo passo é a instalação do Django e, para tanto, há três opções: a primeira é usando o comando *pip*, a segunda é a partir do gerenciador de pacotes do SO, e a terceira é usando o código-fonte. A forma recomendada pela própria documentação do Django é utilizando o

[3] https://www.django-rest-framework.org/

```
virtualenv venv
```

Listing 2.1.2 – Criando novo ambiente virtual

```
nome_da_virtualenv\Scripts\Activate
```

Listing 2.1.3 – Ativando ambiente virtual

comando *pip* (DJANGO, 2023) mostrado no *Listing* 2.1.4 que realiza a instalação do *framework*, o qual deve ser executado dentro do ambiente virtual criado.

```
pip install django
```

Listing 2.1.4 – Comando de instalação do `Django`

2.2 Arquitetura do Django

Conforme sua documentação descreve, a arquitetura do `Django` segue o padrão de projeto MTV (*Model, Template, View*), no qual a aplicação é dividida em camadas (DJANGO, 2023), separando estruturalmente a interação com o usuário das regras de negócio, facilitando o mapeamento do domínio através das URLs e tornando o código mais organizado e manutenível. A Figura 1 apresenta uma visão geral da arquitetura utilizada pelo `Django` e a interação entre as camadas *Model*, *Template* e *View*.

Pode-se observar na Figura 1 que todas as camadas são interligadas de modo a realizar um determinado serviço e executar uma tarefa ou comando solicitado pelo usuário da aplicação, por meio do seguinte fluxo: o usuário faz a requisição HTTP pelo *browser* através das rotas tratadas no arquivo *urls.py*, então é executado o método *View* que utiliza o *Model* para acessar o banco de dados e retornar as informações a serem renderizadas pela camada de *Template* e apresentadas ao usuário através do *browser*. Em suma, a camada *Model* é responsável pela regra de negócio, a camada *View* é responsável pela lógica de processamento, e a camada *Template* é responsável pela apresentação dos dados na interface do usuário. A seguir, cada uma dessas camadas é abordada separadamente.

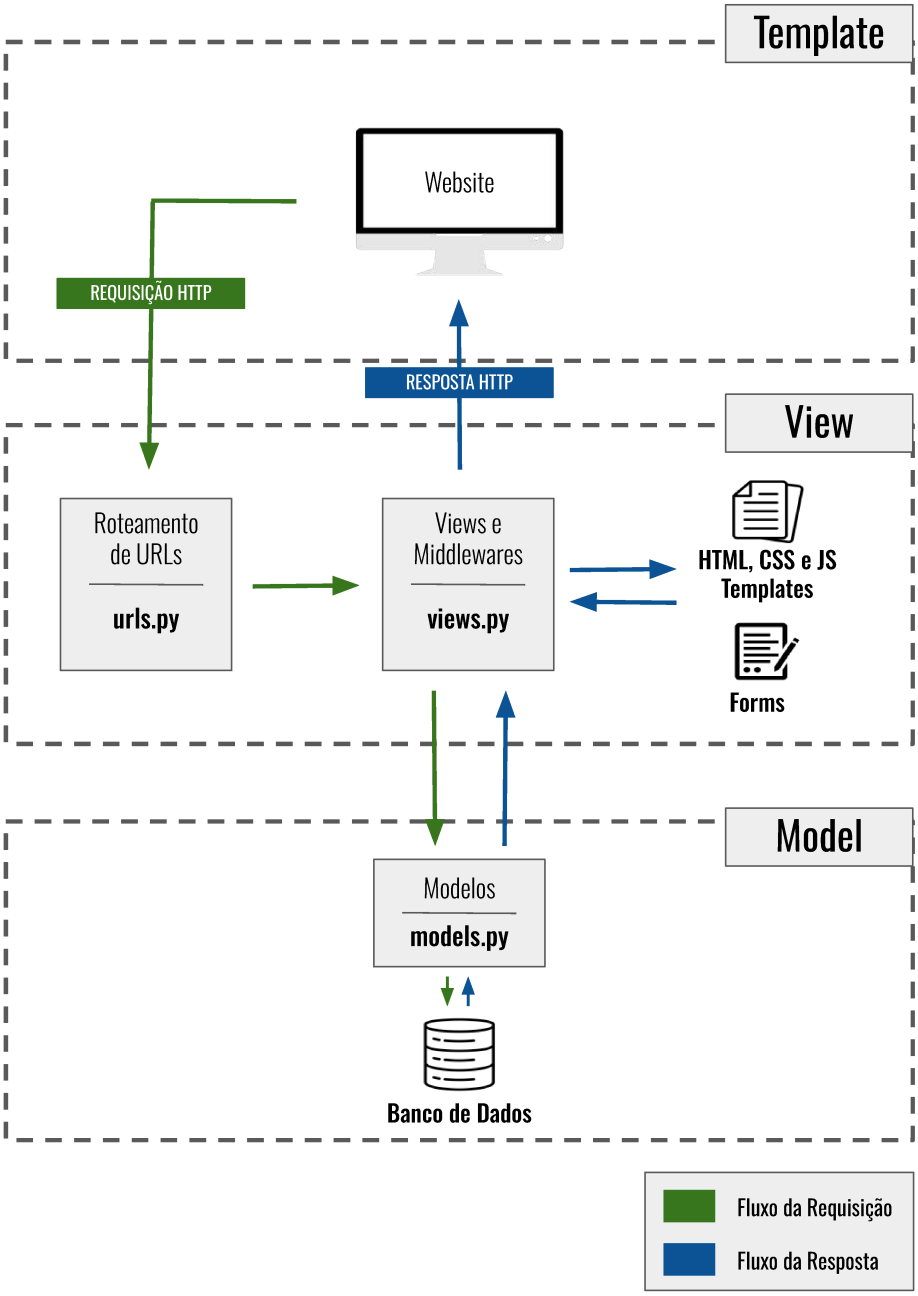


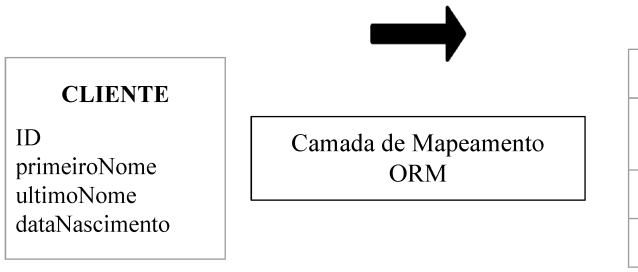
Figura 1 – Arquitetura `Django`. Adaptado de (RAMOS, 2023)

2.2.1 Model

A camada *Model* é responsável por definir a estrutura e o comportamento dos dados a serem armazenados no banco de dados. Os modelos são descritos no arquivo *models.py*, sendo mapeado um modelo correspondente para cada tabela do banco de dados, no qual cada tabela é representada por uma classe `Python` e seus registros são representados como instâncias dessas classes, e cada atributo do modelo representa um campo de dado da tabela. Outra característica dos modelos é a classe que herda de *django.db.models.Model*, a qual permite a definição e

manipulação da estrutura dos dados do banco de dados usando código `Python`, e desempenha papel fundamental na API de dados do `Django`.

A API de acesso ao banco de dados do `Django` é gerada automaticamente e se baseia no padrão *Object-Relational Mapper* (ORM) (DJANGO, 2023), que é um mapeador que estabelece relações entre as tabelas, facilitando o gerenciamento dos dados. A Figura 2 ilustra o suporte nativo oferecido pelo `Django` a esse recurso. Pode-se observar que o `Django` fornece uma abstração de alto nível ao tratar os dados como objetos, cabendo à camada ORM fazer a tradução das operações para *SQL* que, por sua vez, manipula tabelas relacionais. Com isso, o desenvolvedor trabalha com o banco de dados de forma mais eficiente e produtiva.



CLIENTE			
ID	**primeiroNome**	**ultimoNome**	**dataNascimento**
1	João	Silva	12/08/1986
2	Maria	Nascimento	07/12/1993

Tabela Relacional

Figura 2 – *Object-Relational Mapper* (ORM)

Dentre as características do ORM, se destaca o suporte a diferentes tipos de cardinalidades entre as tabelas, como um para um, um para muitos e muitos para muitos, oferecendo também uma API de consulta eficaz para recuperar, filtrar, ordenar e manipular dados no banco de dados. As consultas podem ser realizadas usando métodos encadeados que permitem construir consultas complexas, além de suportar agrupar várias operações de banco de dados em uma única transação e ainda manter a atomicidade da operação, garantindo que todas as operações sejam executadas com sucesso ou revertidas completamente, assegurando a consistência dos dados.

Dessa forma, para manter a consistência entre a definição do modelo e o esquema do banco de dados, além do ORM, o `Django` possui um sistema de migração integrado, as chamadas *migrations*, que nada mais são do que arquivos `Python` responsáveis por gerenciar as alterações no banco de dados ao longo do desenvolvimento do projeto, permitindo manter a conformidade do banco de dados com o modelo à medida que estes são alterados e evoluem. As *migrations* são

projetadas para facilitar a aplicação do modelo nos diferentes bancos de dados suportados pelo `Django`, simplificando o processo de manutenção do banco de dados e fornecendo uma maneira consistente e controlada de evoluir o esquema do banco de maneira conjunta ao crescimento da aplicação.

Além disso, o `Django` mantém um controle interno de todas as *migrations* criadas. Assim, quando uma alteração nos modelos é feita, usa-se o comando *makemigrations* para criar uma nova migração, o qual identifica as alterações e gera um arquivo de migração correspondente que descreve como adequar o banco de dados atual à nova estrutura definida. Para aplicar as mudanças ao banco de dados, usa-se o comando *migrate* que verifica o histórico de *migrations* e aplica apenas as alterações necessárias. Há ainda um sistema de versionamento de *migrations* no `Django`, o qual torna possível reverter uma ou mais *migrations* resultando no retorno a uma versão anterior do banco de dados.

A Figura 3 ilustra o fluxo de trabalho das *migrations*. Inicialmente, são declarados os modelos no arquivo *models.py*, e quando detectada uma alteração no modelo, é feito um versionamento de migrações através do comando *makemigrations*, ou seja, são criados os arquivos de migração no diretório *migrations*. A seguir, as migrações são enviadas ao banco de dados com o comando *migrate*, e, finalmente, as alterações são aplicadas ao banco de dados.

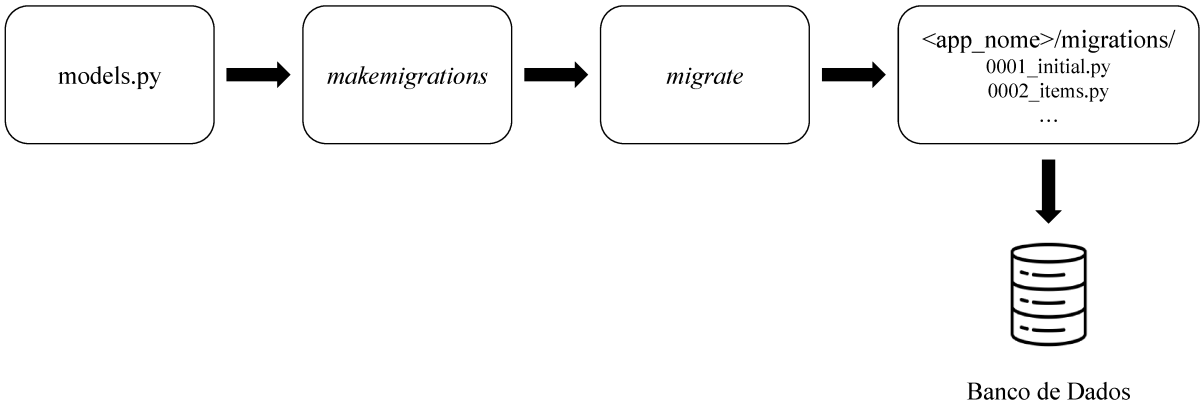


Figura 3 – Fluxo de trabalho das migrações

Embora na prática o `Django` suporte vários Sistemas de Gerenciamento de Banco de Dados (SGBDs), desde que realizados os devidos ajustes recomendados pela respectiva documentação do SGBD, na documentação oficial do `Django` o suporte é atestado apenas para `PostgreSQL`, `MariaDB`, `MySQL`, `Oracle` e `SQLite`, sendo, ainda, recomendado utilizar `SQLite` para projetos pequenos ou para aqueles que não operem em um ambiente de produção (DJANGO, 2023). No entanto, como o `SQLite` tem muitas peculiaridades quando comparado a outros

bancos de dados, é recomendado o desenvolvimento da aplicação com o mesmo banco de dados que se planeja usar em produção.

Por padrão, a configuração do `Django` usa o banco de dados `SQLite`, contudo, para configurar projetos que utilizem outro banco de dados, basta instalar as configurações de conexão de banco de dados apropriadas e alterar os parâmetros no item *DATABASES 'default'* no arquivo *listadecompromissos/settings.py*. Em alguns casos, podem ser necessárias configurações adicionais, tais como *USER*, *PASSWORD* e *HOST*. O *Listing* 2.2.1 apresenta um exemplo de uso do banco de dados `PostgreeSQL` e as respectivas configurações necessárias.

```
1   DATABASES = {
2       "default": {
3           "ENGINE": "django.db.backends.postgresql",
4           "NAME": "meudatabase",
5           "USER": "meudatabaseuser",
6           "PASSWORD": "minhasenha",
7           "HOST": "127.0.0.1",
8           "PORT": "5432",
9       }
10  }
```

Listing 2.2.1 – Exemplo de uso do banco de dados `PostgreeSQL`

2.2.2 View

A camada *View* é responsável por processar as requisições feitas pelos usuários, interagir com o modelo de dados e retornar uma resposta, dependendo, em termos de relacionamento, da camada *Model* para acessar e manipular os dados, permitindo que os modelos forneçam para as *views* as informações a serem renderizadas. Cada *view* retorna um objeto HTTP com o conteúdo da página requisitada e renderiza os *templates*, podendo ainda lidar com formulários através de uma API nativa, autenticação de usuário, redirecionamentos de URLs e outras tarefas relacionadas à interação com o usuário.

No `Django` é possível definir uma *view* como uma função ou como uma classe. Ao definir uma *view* como função, usa-se a sintaxe de funções do `Python`, que recebe como argumento um objeto de requisição e retorna uma resposta HTTP. Já na definição como classe, a *view* herda uma das classes de visualização fornecidas pelo `Django`, como, por exemplo, a "*View*" ou "*TemplateView*", sendo essa uma abordagem útil quando há necessidade de utilização

dos recursos adicionais fornecidos pelas classes de visualização. Dessa forma, a escolha da abordagem para definição de uma *view* depende das necessidades do projeto e dos tipos de recursos a serem implementados. Ao definir uma *view*, esta deve ser mapeada a uma URL específica, permitindo que o `Django` encaminhe as requisições do usuário para a *view* correta.

No *Listing* 2.2.2 é apresentado um exemplo de definição de *view* como uma função, o qual apresenta a seguinte estrutura: na Linha 1 é feita a importação do módulo *django.http* para enviar a resposta HTTP, na Linha 3 é feita a definição da função, as Linhas 4 a 6 correspondem ao escopo da função e na Linha 7, é definido o retorno da função. Já no *Listing* 2.2.3, é apresentado um exemplo de definição de *view* como uma classe e cuja estrutura é: na Linha 2 é feita a importação do módulo *django.view* responsável por criar *views* baseadas em classes, na Linha 4 é feita a declaração da classe herdada de *View* e na Linha 5 é definido um método que retorna, na Linha 9, uma instância HTTP.

```
1  from django.http import HttpResponse
2
3  def new_view(request):
4      # Lógica da view
5
6      # Resposta da view
7      return HttpResponse("Hello, world!")
```

Listing 2.2.2 – Exemplo de definição de *view* como função

```
1  from django.http import HttpResponse
2  from django.views import View
3
4  class NewView(View):
5      def get(self, request):
6          # Lógica da view para a requisição GET
7
8          # Resposta da view
9          return HttpResponse("Hello, world!")
```

Listing 2.2.3 – Exemplo de definição de *view* como classe

As requisições do usuário são processadas a partir da URL que este desejar acessar, cabendo ao próprio `Django` fazer o roteamento das requisições através do arquivo *urls.py*, que é um módulo `Python` que mapeia padrões URL e funções `Python`, relacionando-as. Todo esse processo é representado esquematicamente na Figura 4, seguindo o seguinte fluxo: quando o

usuário requisita uma página, o `Django` percorre cada um dos padrões URL na ordem do código até encontrar a URL que corresponda à requisição quando, então, a *view* associada é processada, e o *template* é renderizado para, por fim, a página solicitada ser enviada como resposta ao usuário. Embora seja feito o teste com cada um dos módulos, essa operação é rápida, pois os caminhos são compilados em expressões regulares. Se nenhuma das URLs corresponder à requisição, o `Django` executa uma *view* especial de exceção 404.

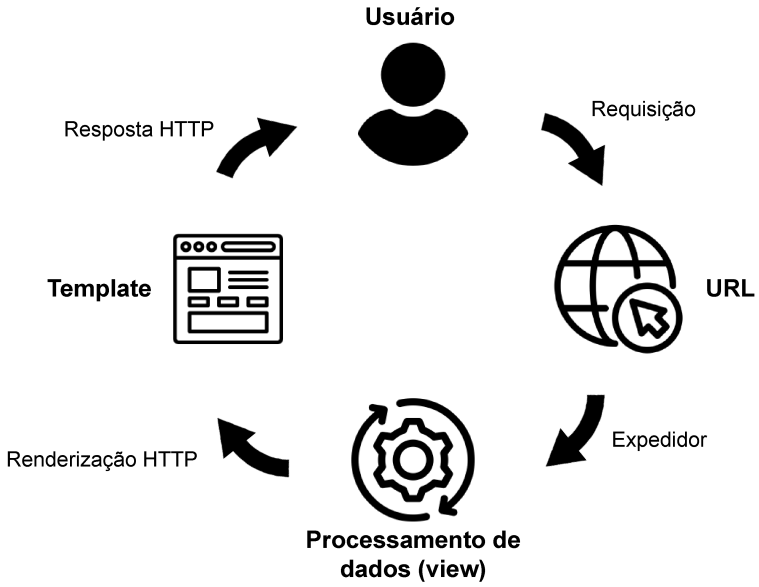


Figura 4 – Fluxo de processamento de requisições URLs. Adaptado de (ANKUSH_953, 2016)

Durante o fluxo de processamento no `Django`, antes de chegar à *view* correspondente, as requisições passam por componentes leves e de baixo nível que processam os dados de entrada e saída e que são chamados de *middlewares*, os quais podem atuar desde o gerenciamento de requisições até o tratamento de erros, inclusive modificando globalmente a requisição, executando ações adicionais ou até mesmo interrompendo o seu processamento, caso seja necessário. A Figura 5 ilustra a interação da requisição e da resposta com os *middlewares*, cujos passos são: a requisição passa pelos *middlewares* listados no arquivo *settings.py* seguindo a ordem determinada na lista, em seguida a *view* correspondente é executada, permitindo ao servidor enviar a resposta à requisição, mas não sem antes essa passar novamente pelos *middlewares*, percorrendo o caminho inverso até ser enviada de volta ao usuário (HOLOVATY; KAPLAN-MOSS, 2009).

Os *middlewares* são implementados como classes que definem métodos específicos, chamados durante o processamento das requisições, os quais podem realizar ações específicas antes, durante ou após o processamento das *views* (HOLOVATY; KAPLAN-MOSS, 2009). As configurações de *middlewares* são especificadas no arquivo *settings.py* na forma de lista e a

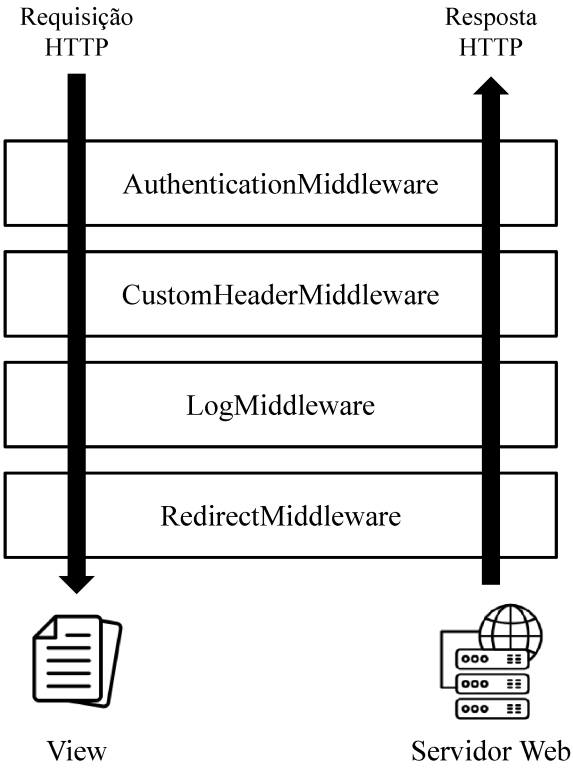


Figura 5 – Diagrama de *Middleware*. Adaptado de (BOUCHER, 2016)

ordem em que estão listados é a ordem de execução durante o processamento das requisições. A ordem de execução é importante, pois cada *middleware* tem a possibilidade de modificar a requisição e a resposta antes de passá-las para o próximo *middleware*. Embora ao criar um projeto, certos *middlewares* sejam adicionados automaticamente, é possível adicionar novos, remover ou reorganizar os *middlewares* de acordo com as necessidades do projeto.

2.2.3 Template

A camada *Template* é responsável pela renderização das páginas da *Web* a partir dos dados recebidos da camada *View*, obtidos por intermédio do mecanismo de renderização do `Django`, retornando para o cliente a página com as informações requisitadas, garantindo a separação da lógica de apresentação da lógica de negócios, promovendo uma melhor organização do código. Estruturalmente, um *template* é um arquivo de texto que define a estrutura da página exibindo informações obtidas na *view*, através da junção entre HTML e código `Python`, permitindo inserir dinamicamente informações na página através da substituição de marcações.

Dessa forma, as marcações no *template* são elementos especiais que permitem a inserção dinâmica de dados e a criação de estruturas condicionais, podendo ser de dois tipos: *tags* ou filtros. Enquanto as *tags* permitem controlar o fluxo de execução e a estrutura do conteúdo

exibido, os filtros permitem manipular valores de variáveis diretamente nos *templates*. O `Django` oferece uma ampla variedade de *tags* e filtros embutidos, e também é possível escrever *tags* e filtros personalizados para atender às necessidades específicas do projeto. A documentação oficial dispõe de uma lista completa e detalhada de todas as *tags* e filtros disponíveis. (DJANGO, 2023)

Há, ainda, três maneiras para utilizar as marcações apresentadas no *Listing* 2.2.4. A primeira forma é a variável de *template*, a qual é delimitada por chaves duplas "{{ }}", e permite exibir dados de uma variável obtidos pela *view*, como exemplificado na Linha 1. A segunda forma é o *filter* que também é representado por chaves duplas, porém contém dentro das chaves o caractere *pipe* "{{ | }}" conforme a Linha 3, adicionando uma formatação à informação que será renderizada e permitindo o encadeamento de filtros. A terceira forma é a *tag expression* delimitada por "{% %}", exemplificada nas Linhas 5 a 8, que permite executar comandos `Python` dentro dos *templates*.

```
1    <p>{{ aluno.nome }}</p>
2
3    <p>{{ aluno.data_de_nascimento | date:'d/m/Y'}}</p>
4
5    {% for aluno in alunos %}
6          <p>{{ aluno.nome }}</p>
7          <p>{{ aluno.matricula }}</p>
8    {% endfor %}
```

Listing 2.2.4 – Exemplos de marcações de *template*

Adicionalmente o `Django` se utiliza do conceito de "herança de *templates*" que permite reduzir a redundância nos *templates* definindo apenas o que é único. Assim, é possível ter uma página *base.html* como um "esqueleto", semelhante a que é apresentada no *Listing* 2.2.5, que contenha os elementos comuns às páginas do *site*. Ainda neste *template*, é definida uma série de blocos para que *templates* filhos possam preenchê-los. No *Listing* 2.2.6 é apresentado um exemplo de *template* filho em que a *tag* {% extends %} indica a herança do *template* pai. O uso da "herança de *templates*" permite criar múltiplas versões de um *site* com diferentes *templates* base como, por exemplo, a criação de versões mobile de uma aplicação *Web* originalmente desenvolvida para computador.

Por fim, o `Django` interpreta arquivos adicionais como imagens, arquivos `JavaScript` ou arquivos `CSS` como arquivos estáticos que, por padrão, são armazenados em uma pasta cha-

```
<html>
<head>
    <title>{% block title %}{% endblock %}</title>
</head>
<body>
    {% block content %}{% endblock %}
</body>
</html>
```

Listing 2.2.5 – Exemplo de *template base.html*

```
{% extends "base.html" %}

{% block title %}Lista de alunos de {{ semestreIngresso }}{%
↪  endblock %}

{% block content %}
<h1>Lista de alunos de {{ semestreIngresso }}</h1>

{% for aluno in alunos %}
    <p>{{ aluno.nome }}</p>
    <p>Matrícula: {{ aluno.matricula }}</p>
{% endfor %}
{% endblock %}
```

Listing 2.2.6 – Exemplo de *template* filho

mada *static* localizada dentro do diretório *app*. No *Listing* 2.2.7 é apresentada, na Linha 1, a *tag {% load static %}* que é utilizada para referenciar o uso de arquivos estáticos, já nas Linhas 7 e 9 exemplos de arquivos são incluídos na página, sendo eles, respectivamente, uma imagem e um arquivo `JavaScript`.

```
{% load static %}
<html>
<head>
    <title>Título da página</title>
</head>
<body>
    <img src="{% static 'images/sitelogo.png' %}" alt="Logo">

    <script src="{% static 'script.js' %}"></script>
</body>
</html>
```

Listing 2.2.7 – Exemplo de uso de arquivos estáticos

Dessa forma, as páginas de exibição de erro podem ser facilmente substituídas quando há necessidade de um comportamento personalizado. Inclusive, a documentação do `Django` (DJANGO, 2023) sugere ter uma página de erro consistente para todo o *site*, de tal forma que o `Django` irá capturar a página e retornar como página de erro padrão da aplicação juntamente com um código de erro HTTP. Além disso, dispõe dos *middlewares* exclusivos para tratamento de erros, ideais para detectar URLs incorretas.

2.2.4 Vantagens do Django

Interface Admin: o `Django` dispõe de uma interface de administração nativa semelhante a um *website*, como mostra a Figura 6, a qual é personalizável através da classe *ModelAdmin* e permite aos usuários administradores do banco de dados modificar ou remover objetos de forma amigável, dispensando a necessidade de criação de uma interface. O usuário administrador é ativado por padrão no modelo de projeto usado pelo comando *startproject*. Para criar um novo usuário, usa-se o comando *createsuperuser*. Já o *site* de administração, pode ser acessado através da URL conectada /admin/.

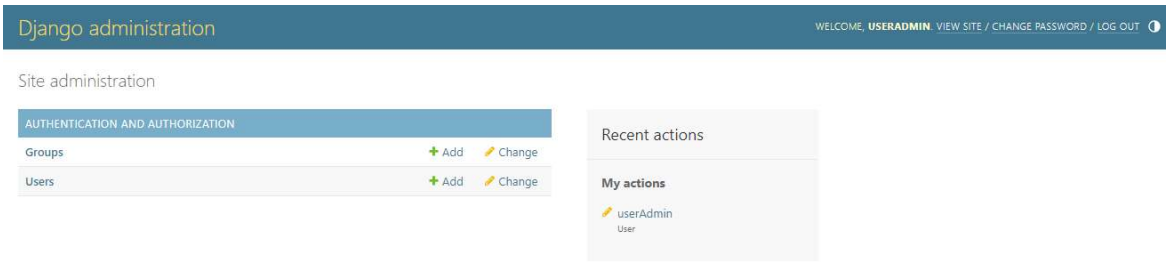


Figura 6 – Fragmento de tela inicial do site de administração

Sistema de autenticação de usuário: o `Django` possui um sistema de autenticação e autorização de usuário que lida com contas de usuário, grupos, permissões e sessões baseadas em *cookies* (DJANGO, 2023), permitindo verificar as credenciais do usuário e definir a permissão de acesso de cada um (DOCS, 2024). A estrutura de autenticação visa ser genérica e flexível, permitindo criar, por exemplo, URLs e formulários, a partir do zero ou chamando a API fornecida pelo *framework* para efetuar o *login* do usuário (DOCS, 2024). Além disso, permite personalizar a autenticação de acordo com a necessidade do projeto a ser desenvolvido (DJANGO, 2023).

2.3 Exemplo prático com Django

O exemplo apresentado nesta seção é uma aplicação de Lista de Compromissos, desenvolvida com o *framework* `Django`, que permite cadastrar, editar, visualizar e excluir atividades por meio de uma página *Web*. Através do uso de um banco de dados, a descrição da atividade fornecida pelo usuário, bem como a data e a hora nas quais o cadastro foi feito, são armazenadas de forma persistente, ou seja, tais informações não são perdidas mesmo após recarregar a página. As Seções 2.3.1 e 2.3.2 descrevem todo o desenvolvimento dessa aplicação.

2.3.1 Preparação do ambiente

Após a instalação do *framework*, para inicializar um novo projeto, executa-se o comando do *Listing* 2.3.1 que, neste caso, inicializa um projeto chamado *listadecompromissos*. Nesse comando, o termo *startproject* cria automaticamente um diretório interno chamado, também, de *listadecompromissos*, além de arquivos de inicialização, cada qual com um propósito específico. Tais arquivos de inicialização são ilustrados na Figura 7, que contém os nomes dos arquivos e a organização dos diretórios.

```
1    django-admin startproject listadecompromissos
```

Listing 2.3.1 – Inicializando projeto

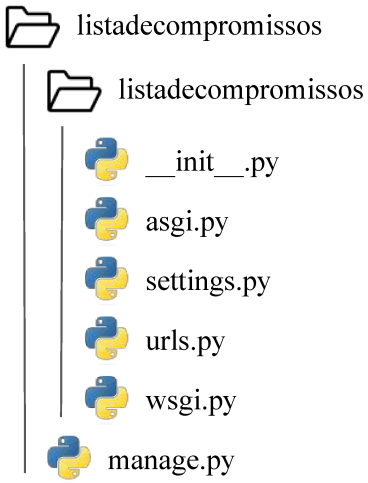


Figura 7 – Arquivos de Inicialização

O diretório *listadecompromissos/* é um contêiner para o projeto e só contém o *script*

`Python` *manage.py*, utilitário de linha de comando para tarefas administrativas, e um diretório homônimo que constitui o pacote para o projeto, composto pelos seguintes arquivos:

- *__init__.py*: arquivo vazio que serve para indicar ao `Python` que naquele diretório contém código executável;
- *settings.py*: módulo de configuração do projeto;
- *urls.py*: contém o mapeamento das URLs internas do projeto como um índice do *site*;
- *asgi.py* e *wsgi.py*: pontos de integração para servidores *Web* compatíveis com ASGI e WSGI, respectivamente.

Para iniciar o servidor *Web* local de hospedagem do projeto para testes, é preciso acessar o diretório *listadecompromissos/* exterior por meio do terminal e executar o arquivo *manage.py* seguido da tarefa *runserver*, conforme o *Listing* 2.3.2. Em seguida, deve-se acessar, através do *browser*, o endereço http://127.0.0.1:8000, que acarreta a exibição da tela inicial padrão de uma aplicação `Django`, conforme a Figura 8, a qual contém instruções básicas para personalizar projetos e *links* úteis com informações extras.

```
1    python manage.py runserver
```

Listing 2.3.2 – Inicializando servidor *Web* local

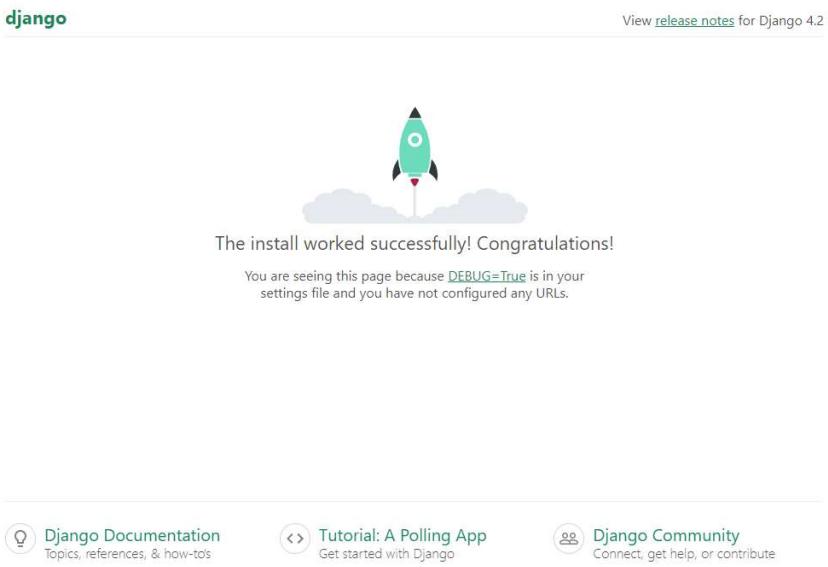


Figura 8 – Tela inicial de uma aplicação `Django`

A partir de agora o projeto está configurado para, então, trabalhar com as *apps*, que são aplicações *Web* que fazem algo específico, como, por exemplo, o gerenciamento das tarefas

da Lista de Compromissos. O projeto é, então, uma coleção de configurações e *apps* para um determinado *site*, podendo um mesmo projeto conter várias *apps*.

2.3.2 Implementação do exemplo

As *apps* podem ficar em qualquer diretório dentro do caminho de busca dos módulos `Python`. Neste projeto, a *app* será criada no mesmo diretório do arquivo *manage.py* para ser importada como um módulo de nível superior, para isso, navega-se, através do terminal, para o mesmo diretório que o *manage.py* se encontra e executa-se o comando do *Listing* 2.3.3 para criar uma *app* referente às tarefas da lista de compromissos, isto é, um novo diretório chamado "tarefas". Em seguida, cria-se uma nova *view* com o código `Python` do *Listing* 2.3.4 no arquivo *tarefas/views.py*, que adiciona uma função *index* com uma resposta HTTP como retorno. Por enquanto, o retorno HTTP da *view* criada será usado apenas para testar se o acesso está correto.

```
1       python manage.py startapp tarefas
```

Listing 2.3.3 – Criando *app* "tarefas"

```
1   from django.http import HttpResponse
2
3   def index(request):
4       return HttpResponse("Hello, world!")
```

Listing 2.3.4 – Criando *view* de teste

Para chamar uma *view* é preciso mapear a URL criando um arquivo *urls.py* no diretório de tarefas e mapear a rota *index* através da função *path* importada da *django.urls*, como mostra o *Listing* 2.3.5, no qual são passados dois argumentos obrigatórios, *route* e *view*, e um argumento opcional *name*. A *route* é uma *string* que descreve a URL, a *view* é uma função utilizada para executar quando a URL requisitada combina com o argumento *route*, e o argumento *name* é utilizado para referenciar a *view* em qualquer outro lugar do projeto.

A nova *app* deve ser mapeada no projeto importando a função *include* de *django.urls* no arquivo *listadecompromissos/urls.py* e inserir *include()* na lista *urlpatterns*, conforme o *Listing* 2.3.6, permitindo referenciar outras *URLconfs*. A seguir, o servidor deve ser novamente inicializado através do comando exibido no *Listing* 2.3.2 e, em seguida, navega-se até a página http://127.0.0.1:8000/tarefas/ através do *browser* para verificar se a *view* está sendo exibida

```
    from django.urls import path
    from . import views

    urlpatterns = [
        path("", views.index, name="index"),
    ]
```

Listing 2.3.5 – Mapeando *view index*

corretamente, devendo ser apresentada a mensagem "*Hello, world!*", enviada como retorno da *view*.

```
    from django.contrib import admin
    from django.urls import include, path

    urlpatterns = [
        path("tarefas/", include("tarefas.urls")),
        path("admin/", admin.site.urls),
    ]
```

Listing 2.3.6 – Mapeando a *app* "tarefas"

No arquivo *listadecompromissos/settings.py* estão as configurações de banco de dados, além da configuração do *INSTALLED_APPS* a qual possui os nomes de todas as aplicações `Django` ativas. A documentação do `Django` (DJANGO, 2023) detalha a configuração padrão contida ao inicializar o projeto. Assim, Para utilizar essas aplicações é necessário criap as tabelas do banco de dados, pois algumas das aplicações fazem uso de pelo menos uma tabela do banco de dados. Para isso, executa-se o comando *migrate* conforme o *Listing* 2.3.7 que irá analisar a configuração *INSTALLED_APPS* e criar as tabelas necessárias no banco de dados de acordo com as configurações do banco de dados, que neste projeto trata-se do `SQLite`, dada a simplicidade da aplicação.

```
        python manage.py migrate
```

Listing 2.3.7 – Criando tabelas no banco de dados

O projeto "Lista de compromissos" é composto por apenas um modelo chamado "Tarefa", o qual possui dois campos: o conteúdo da tarefa e a data de criação. Para isso, no arquivo *tarefas/models.py* adiciona-se o conteúdo do *Listing* 2.3.8. Pode-se observar que "Tarefa"

é uma classe simples do Python na qual cada campo é representado por uma instância da classe *Field*, sendo que o *CharField* é usado para os campos do tipo caractere e requer obrigatoriamente um *max_length*, já o *DateTimeField* para campos do tipo data/hora, indicando ao Django o tipo de dado contido em cada campo. Um *Field* pode ter vários argumentos opcionais, colocados a depender da necessidade da aplicação. As instâncias "conteudo" e "criado" são os nomes do campo que o banco de dados irá utilizar para nomear as colunas.

```
1    from django.db import models
2    class Tarefa (models.Model):
3        conteudo=models.CharField(max_length=300)
4        criado= models.DateTimeField(auto_now_add=True)
5        def __str__ (self):
6           return self.coteudo
```

Listing 2.3.8 – Criando modelo "Tarefas"

Ademais, para incluir a *app* "Tarefas" no projeto, é necessário referenciar a classe de configuração da *app*, *TarefasConfig*, na configuração de *INSTALLED_APPS*. Para tanto, como a classe *TarefasConfig* está no arquivo *tarefas/apps.py*, basta adicionar o caminho *tarefas.apps.TarefasConfig* na lista de *INSTALLED_APPS*, como mostrado no *Listing* 2.3.9. Deve-se, ainda, incluir a *app* "Tarefas" através do comando *makemigrations* seguido do nome do diretório, como mostrado no *Listing* 2.3.10 para gerar uma *migration* da alterações feitas no modelo. Para aplicar as alterações ao banco de dados, o comando do *Listing* 2.3.7 deve ser executado novamente.

```
1    INSTALLED_APPS = [
2        'django.contrib.admin',
3        'django.contrib.auth',
4        'django.contrib.contenttypes',
5        'django.contrib.sessions',
6        'django.contrib.messages',
7        'django.contrib.staticfiles',
8        'tarefas',
9    ]
```

Listing 2.3.9 – Adicionando caminho à lista "INSTALLED_APP"

```
1        python manage.py makemigrations tarefas
```

Listing 2.3.10 – Incluindo *app* “Tarefas” no banco de dados

Para criar um usuário administrador que possa acessar o *site* de administração, o comando *createsuperuser* deve ser executado conforme demonstrado pelo *Listing* 2.3.11. Em seguida, no terminal, digita-se o nome de usuário desejado, um endereço de *e-mail* e uma senha. Para acessar o *site* de administração através do *browser*, basta navegar até o endereço http://127.0.0.1:8000/admin/ e então é apresentada uma tela de *login* contendo um formulário de *login*, que dá acesso ao sistema, como exemplifica a Figura 9. Ao acessar o sistema com a conta de superusuário criada, será exibida a página inicial do *site* de administração, na qual há alguns tipos de conteúdos editáveis, incluindo grupos e usuários, cujas funcionalidades são fornecidas pelo *framework* de autenticação do `Django`.

```
1        python manage.py createsuperuser
```

Listing 2.3.11 – Criando superusuário

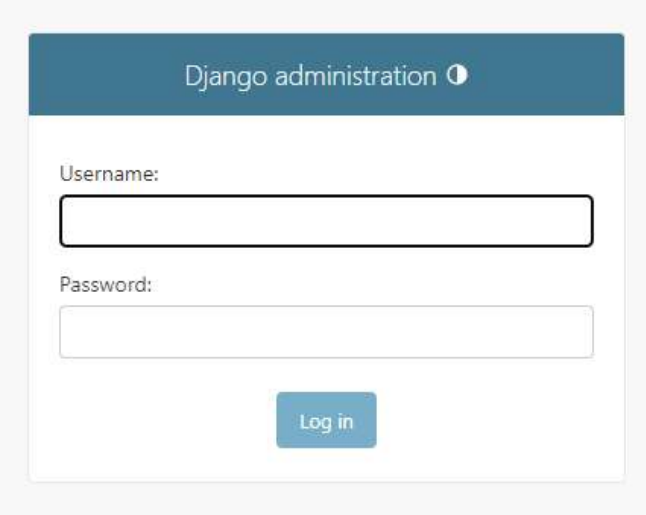


Figura 9 – Fragmento de tela de *login* do site de administração

Para tornar a *app* “Tarefas” visível na página principal do *site* de administração, deve-se editar o arquivo *tarefas/admin.py* e registrar a *app*, conforme o código apresentado no *Listing* 2.3.12. Na Linha 4, a função *register()* registra a *app* para ser gerenciada pelo *site* de administração. A *app* deve ser então exibida no *site* de administração ao recarregar a página no

browser.

```
1    from django.contrib import admin
2    from .models import Tarefa
3
4    admin.site.register(Tarefa)
```

Listing 2.3.12 – Ativando *app* “Tarefas” no site de administração

Já para criar os *templates*, é necessário criar um diretório chamado “*templates*”, e nele criar um outro diretório chamado “tarefas”, no qual, finalmente, é criado um arquivo *index.html* contendo o código apresentado no *Listing* 2.3.13. O conteúdo dessa página, delimitado pela *tag* <body> é composto por um formulário, nas Linhas 10 a 14, e por uma estrutura de repetição, Linhas 17 a 22. O formulário, que utiliza o método *POST*, é composto por um *token* (Linha 11) de proteção contra ataques de falsificação de solicitações entre sites, por um *input* que recebe o conteúdo da tarefa e por um botão que submete o formulário. Já a estrutura de repetição percorre a variável tarefa exibindo seu conteúdo e data de criação, e fornece *links* que direcionam para as URLs de edição e exclusão da tarefa, respectivamente, as quais são identificadas pelo valor *tarefa.id* passado como argumento.

```
<html lang="pt-br">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width,
    ↪  initial-scale=1.0">
    <title>Lista de Compromissos</title>
</head>
<body>
    <form method="POST">
        {% csrf_token %}
        <input type="text" name='conteudo'>
        <button type="submit">Enviar</button>
    </form>
    {% for tarefa in tarefa %}
        <p>{{tarefa.conteudo}}</p>
        <p><a href="{% url 'editar_tarefa' tarefa.id
        ↪  %}">Editar</a></p>
        <p><a href="{% url 'delete' tarefa.id %}">Excluir</a></p>
        <p>{{tarefa.criado}}</p>
    {% endfor %}
</body>
</html>
```

Listing 2.3.13 – Conteúdo da página *index.html*

O próximo passo é atualizar o conteúdo da *view index* no arquivo *tarefas/views.py*, conforme mostra o *Listing* 2.3.14, o qual exibe as tarefas cadastradas. Assim, quando uma nova tarefa é adicionada, ela é salva no banco de dados e o usuário é redirecionado à mesma página.

```
from django.shortcuts import render, redirect, get_object_or_404
from .models import Tarefa
def index (request):
    tarefa=Tarefa.objects.all()
    if request.POST:
        tarefa=request.POST.get('conteudo')
        tarefa=Tarefa(conteudo=tarefa)
        tarefa.save()
        return redirect('index')
    context={
        'tarefa':tarefa,
    }
    return render(request, 'tarefas\index.html',context)
```

Listing 2.3.14 – Atualizando *view index*

Além disso, outra *view* deve ser adicionada conforme o código do *Listing* 2.3.15 no arquivo *tarefas/views.py*, abaixo da *view index*, recebendo um identificador da tarefa como argumento indicando a tarefa que deve ser excluída. Na Linha 2, a variável "tarefa" recebe do banco de dados a tarefa que possui o *id* correspondente; na Linha 3, a tarefa é excluída do banco de dados e, por fim, na Linha 4, o usuário é redirecionado à *view index* onde a página será renderizada com as tarefas restantes no banco. A nova *view excluir_tarefa* deve então ser mapeada no arquivo *tarefas/urls.py* incluindo na lista *urlpatterns* outra rota, como no *Listing* 2.3.16, através da função *path()* que recebe como argumento o *id* da tarefa que será excluída.

```
1    def excluir_tarefa(request,id):
2        tarefa=Tarefa.objects.get(id=id)
3        tarefa.delete()
4        return redirect('index')
```

Listing 2.3.15 – Criando nova *view excluir_tarefa*

```
1    path('delete/<int:id>/', views.excluir_tarefa, name='delete')
```

Listing 2.3.16 – Mapeando rota para excluir tarefa

Ademais, para adicionar uma *view* de edição de tarefa, deve-se definir uma nova função *editar_tarefa*, conforme o *Listing* 2.3.17, que recebe como argumento o *id* da tarefa a ser editada. A função *editar_tarefa* recebe como argumento uma requisição e o *id* da tarefa que será editada, e em seu escopo busca a tarefa pelo *id* e altera o conteúdo da tarefa de acordo com as informações enviadas pelo formulário através do método POST, em seguida, salva as alterações no banco de dados e retorna à página inicial da aplicação.

```
def editar_tarefa(request, id):
    tarefa = get_object_or_404(Tarefa, id=id)
    if request.method == 'POST':
        conteudo = request.POST.get('conteudo')
        tarefa.conteudo = conteudo
        tarefa.save()
        return redirect('index')
    context = {
        'tarefa': tarefa,
    }
    return render(request, 'tarefas/editar.html', context)
```

Listing 2.3.17 – Criando nova *view editar_tarefa*

Em seguida, no diretório *templates/tarefas* deve ser adicionado um novo *template editar.html* com o conteúdo apresentado no *Listing* 2.3.18, que cria o formulário de edição. Além disso, a nova rota de edição deve ser mapeada no arquivo *urls.py* de acordo com a Linha 7 do *Listing* 2.3.19, que recebe a url e a *view* a ser exibida ao acessar a rota.

```
<!DOCTYPE html>
<html lang="pt-br">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
    ↪ initial-scale=1.0">
    <title>Editar tarefa</title>
</head>
<body>
    <form method="POST">
        {% csrf_token %}
        <label for="conteudo">Editar Tarefa:</label>
        <input type="text" name="conteudo" id="conteudo" value="{{
        ↪ tarefa.conteudo }}">
        <button type="submit">Salvar</button>
    </form>
</body>
</html>
```

Listing 2.3.18 – Conteúdo da página *editar.html*

Por fim, para estilizar a página *index.html* utiliza-se um arquivo estático, criando novos diretórios *tarefas/static/tarefas* e adicionando um arquivo *style.css* com o conteúdo apresentado no *Listing* 2.3.20, que altera a cor de fundo da página.

```
from django.urls import path
from . import views

urlpatterns = [
    path("", views.index, name="index"),
    path('delete/<int:id>/',views.excluir_tarefa,name='delete'),
    path('editar/<int:id>/', views.editar_tarefa,
    ↪  name='editar_tarefa'),
]
```

Listing 2.3.19 – Mapeando rota para editar tarefa

```
body {
    background-color: #E7F5CF;
}
```

Listing 2.3.20 – Exemplo de conteúdo da página *style.css*

Em seguida, deve-se adicionar o conteúdo do *Listing* 2.3.21 na *tag* <header> do arquivo *index.html* para referenciar o arquivo de estilização, e executar novamente o comando de inicialização do servidor, apresentado anteriormente no *Listing* 2.3.2. Dessa forma, o projeto "Lista de compromissos" finalizado resultará na Figura 10.

```
{% load static %}

<link rel="stylesheet" href="{% static 'tarefas/style.css' %}">
```

Listing 2.3.21 – Referenciando estilização no HTML

Figura 10 – Tela do projeto finalizado em `Django`

2.4 Visão geral do Framework Laravel

O Laravel[4] é um *framework Web* gratuito, de código aberto, escrito em PHP[5], com sintaxe expressiva e elegante (LARAVEL, 2023), projetado com o objetivo de fornecer ao desenvolvedor uma experiência agradável e recursos poderosos, sendo uma ferramenta completa para codificação (CARDOSO, 2021). Este *framework* se propõe a entregar projetos de qualidade, minimizando a escrita manual de código por meio da utilização de métodos e classes, fazendo-se necessário pouco código nativo PHP e utilizando-se de linhas de comando para gerar arquivos (GABARDO, 2017).

Desenvolvido por Taylor Otwell, o Laravel teve sua primeira versão lançada em junho de 2011 e contando com, além de um ORM personalizado, chamado de *Eloquent*, roteamento baseado em *closures*, função que permite passar objeto e parâmetro para outras funções e métodos, atribuir uma variável, ou até mesmo serialização. Possuía, ainda, auxiliares para criação de formulários, validações e autenticações (STAUFFER, 2019).

Dessa forma, Otwell se dispôs a desenvolver seu próprio *framework* a partir de sua insatisfação com o *framework* CodeIgniter[6], um dos primeiros *frameworks* PHP que teve inspiração no Ruby on Rails e era simples e fácil de usar, com uma boa documentação e forte comunidade, porém, em termos de avanços tecnológicos, demorou a incorporar novos recursos das versões seguintes do PHP (STAUFFER, 2019). A partir da versão Laravel 4, sua estrutura passou a extrair componentes do Symfony[7], outro *framework* PHP, focado em padrões de design corporativo e comércio eletrônico. Embora tenha sido inspirado em Symfony, com sua constante evolução, o Laravel passou a ser um *framework* independente e cresceu em recursos (STAUFFER, 2019).

O Laravel se caracteriza por apresentar uma estrutura progressiva capaz de lidar com o crescimento do projeto de maneira a atender aplicações simples ou até mesmo lidar com aplicações *Web* de cargas de trabalho corporativas. É também escalável por permitir lidar com centenas de milhões de solicitações por mês. Além disso, ele combina com os melhores pacotes do ecossistema PHP, buscando oferecer um *framework* mais robusto e amigável para o desenvolvedor, resultando em uma comunidade com milhares de desenvolvedores que contribuem para a sua estruturação (LARAVEL, 2023).

[4] https://laravel.com/
[5] https://www.php.net/
[6] https://codeigniter.com/
[7] https://symfony.com/

Assim, os valores mais fortes do Laravel são aumentar a velocidade e a felicidade do desenvolvedor, fornecendo código e recursos claros, simples e bonitos que o ajudem a escrever códigos claros e duradouros. Ter a felicidade do desenvolvedor como preocupação primária impactou no estilo do *framework* e no progresso da tomada de decisões. Isso porque o Laravel se concentra em uma curva de aprendizado superficial, ou seja, não é necessário um profundo conhecimento da linguagem para que seja desenvolvida uma aplicação, visando minimizar as etapas entre o início de um novo projeto até sua publicação. As tarefas mais comuns na construção de aplicações *Web*, como interação com bancos de dados e autenticações, entre outras, são simplificadas pelos componentes que o Laravel fornece (STAUFFER, 2019).

O Laravel tem como requisito de instalação a linguagem PHP, podendo ser instalada através do pacote gratuito XAMPP[8], e seu gerenciador de dependências, Composer[9]. Após a instalação das dependências, executa-se o comando do *Listing* 2.4.1 que instala o Laravel na máquina e cria um novo projeto, neste caso, sendo chamado de "novo-projeto".

```
1   composer create-project --prefer-dist laravel/laravel novo-projeto
```

Listing 2.4.1 – Comando de instalação do Laravel e criação de um novo projeto

2.5 Arquitetura do Laravel

A arquitetura do Laravel segue o padrão de projeto MVC (*Model*, *View*, *Controller*), no qual as camadas de lógica e apresentação são separadas para manter uma maior organização do código, além de vantagens no ciclo de vida do projeto (GABARDO, 2017). Esse padrão arquitetural se popularizou através do Ruby on Rails, além de outras ferramentas e convenções que tiveram influência no desenvolvimento de aplicações *Web*, principalmente no que diz respeito à rapidez (STAUFFER, 2019).

Conforme a Figura 11 apresenta, o fluxo de dados da aplicação é iniciado por meio de uma requisição HTTP, feita pelo usuário, que é mapeada através do mapeamento das rotas e encaminhada para um *Controller*, que processa a requisição e envia dados ao *Model* para realizar a interação com o banco de dados. Ao final desta etapa, o controlador envia os dados obtidos para a camada *View*, responsável pela apresentação dos dados em tela ao usuário.

[8] https://www.apachefriends.org/pt_br/index.html
[9] https://getcomposer.org/

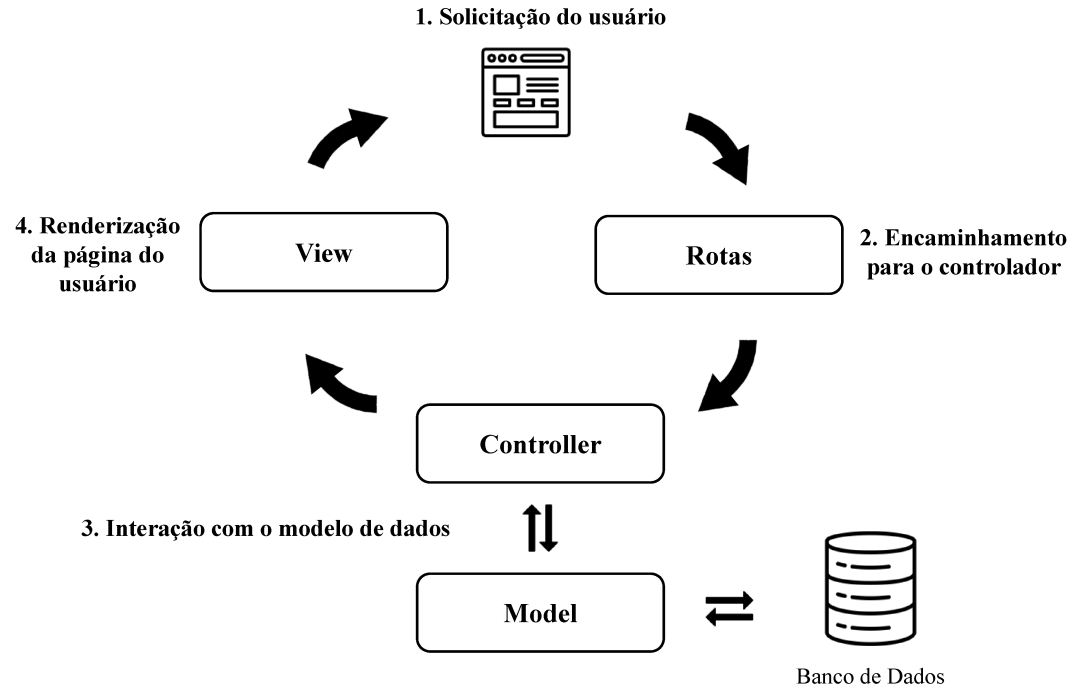


Figura 11 – Arquitetura `Laravel`. Adaptado de (PATEL, 2023a).

2.5.1 Rotas

As rotas mapeiam as requisições HTTP solicitadas pelo usuário e as redirecionam para um controlador apropriado através de uma lista de roteamento. Embora esta não seja, de fato, uma camada do modelo MVC, é a abordagem mais comum para tratar de requisições HTTP (GABARDO, 2017). No `Laravel`, a classe responsável por definir as rotas da aplicação e carregar automaticamente os arquivos de rotas que ficam no diretório */routes* é a *Route Service Provider*.

Por padrão do `Laravel`, os arquivos de rotas são divididos em *web.php* e *api.php*. No arquivo *web.php* estão as rotas acessadas pelo usuário final, ou seja: são requisições filtradas por *middlewares* que fornecem recursos como controle de sessão e encriptação de *cookies*. O termo *middleware* caracteriza uma camada intermediária que possibilita a comunicação entre aplicações distribuídas (MACIEL; ASSIS, 2004). No arquivo *api.php* estão as rotas para API (*Application Programming Interface*), se houver.

Há duas maneiras de definir rotas no `Laravel`, conforme apresentado no *Listing* 2.5.1: *Route Verbs* e *Route Handling*. Nas Linhas 1 a 3 é apresentado um exemplo de rota definida por *Route Verbs*, que utiliza a requisição do tipo *GET* acessada pela URL raiz "/" e executa uma função anônima que retorna uma *view* chamada "welcome". É possível ainda passar argumentos através das rotas. Na Linha 5, o parâmetro "nome" é passado para

a função anônima através da rota, e a instrução *echo* na Linha 6 é usada para exibir conteúdo HTML na página de retorno. Já ao utilizar o *Route Handling* para definir uma rota, pode-se reduzir até centenas de milissegundos de cada solicitação para um arquivo com grande lógica de roteamento (STAUFFER, 2019). Para isso, passa-se o nome do *controller* e do método como uma *string* no lugar do fechamento, como mostrado na Linha 9. Isto quer dizer ao `Laravel` para passar solicitações ao método *index* do *controller app/Http/Controllers/WelcomeController*. Assim como no *Route Verbs*, o *Route Handling* possui os mesmos parâmetros e será tratado da mesma forma, sendo essa uma maneira alternativa para definir rota.

```
1    Route::get('/', function () {
2        return view('welcome');
3    });
4
5    Route::get('/aluno/{nome}', function ($nome) {
6        echo "<h1>Aluno: $nome </h1>";
7    });
8
9    Route::get('/', 'WelcomeController@index');
```

Listing 2.5.1 – Definindo rotas

Em ambas as maneiras de definir rotas, para tornar os parâmetros de rota opcionais, inclui-se um ponto de interrogação (?) após o nome do parâmetro, conforme o *Listing* 2.5.2. Neste caso, é necessário fornecer um valor *default* para a variável correspondente do parâmetro de rota.

```
1    Route::get('users/{id?}', function ($id = 'fallbackId') {
2     // Escopo da função
3    });
```

Listing 2.5.2 – Definindo parâmetros de rota opcionais

2.5.2 Controller

A camada *Controller* é responsável por agrupar a lógica de negócios, recebendo as requisições por meio das rotas. Além disso, consome e envia registros para os modelos, e envia os dados de exibição às *views* (GABARDO, 2017). Os *controllers* são armazenados no diretório *app/Http/Controllers*, e podem agrupar a lógica de tratamento de requisições em uma única classe

como, por exemplo, uma classe *UserController* que pode manipular as requisições relacionadas aos usuários como mostrar, criar, atualizar e excluir usuários do banco de dados (LARAVEL, 2023).

O *Artisan* é um conjunto de ações de linha de comando integradas capaz de executar *migrations*, criar usuários e outros registros no banco de dados, além de muitas outras tarefas (STAUFFER, 2019). Para gerar um novo *controller*, por linha de comando, utiliza-se o comando *Artisan make:controller* seguido do nome do *controller*, como no *Listing* 2.5.3, que, neste exemplo, cria um novo *controller* chamado *User Controller*.

```
1    php artisan make:controller UserController
```

Listing 2.5.3 – Criando novo *controller*

O `Laravel` oferece uma convenção para as rotas CRUD chamada *resource controller*. Através de linha de comando, como no *Listing* 2.5.4, um *controller* é gerado contendo um método para cada operação de recurso disponível e rotas prontas para uso. A Tabela 1 apresenta todas as ações disponíveis no *resource controller*, em que cada ação espera uma chamada padrão de URL utilizando o HTTP *Verb* específico. A primeira coluna corresponde ao HTTP *Verb*, a segunda coluna corresponde às URLs, a terceira coluna corresponde ao método do *controller*, e por fim, a quarta coluna corresponde à descrição de cada ação. Para vincular automaticamente as rotas listadas na Tabela 1 aos métodos de ação do *controller* é utilizado o recurso no arquivo *web.php*, como apresentando no *Listing* 2.5.5.

```
1    php artisan make:controller UserController --resource
```

Listing 2.5.4 – Utilizando *resource controller*

```
1    Route::resource('tasks', 'TasksController');
```

Listing 2.5.5 – Referenciando rota com o *resource controller*

A ideia dos *middlewares* é que funcionem como uma camada que envolve a aplicação inspecionando as solicitações e aceitando ou rejeitando-as com base no que foi projetado (STAUFFER, 2019), como exemplifica a Figura 12. Os *middlewares* estão localizados no diretório *app/Http/Middleware*, e por meio deles é possível, por exemplo, verificar se o

Tabela 1 – Métodos *Resource controller*. Adaptado de (STAUFFER, 2019).

HTTP Verb	URL	Método do Controller	Descrição
GET	users	index( )	Exibe todos os usuários
GET	users/create	create( )	Exibe um formulário de criação de um novo usuário
POST	users	store( )	Aceita a submissão do formulário de criação de um novo usuário
GET	users/{user}	show( )	Exibe um usuário
GET	users/{user}/edit	edit( )	Edita um usuário
PUT/PATCH	users/{user}	update( )	Aceita o formulário de edição de um usuário
DELETE	users/{user}	destroy( )	Exclui um usuário

usuário está autenticado, e em caso de correspondência, permitir que a requisição prossiga para a aplicação, ou caso contrário redirecionar o usuário para a tela de *login*.

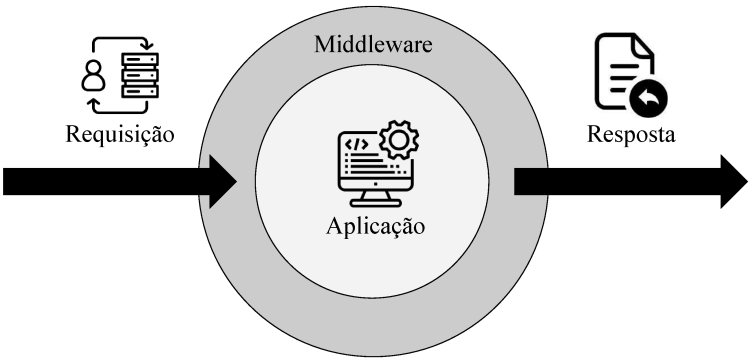


Figura 12 – *Middleware* no `Laravel`. Adaptado de (STAUFFER, 2019).

Além disso, os *middlewares* podem ser aplicados na lógica da aplicação pelas rotas ou diretamente no *controller*. O *Listing* 2.5.6 apresenta um exemplo de uso de *middleware* na rota da aplicação que realiza a verificação de autenticação do usuário por meio do *middleware auth* antes de executar o método *show*. Semelhantemente, no *Listing* 2.5.7 é apresentado um exemplo de uso de *middleware* no *controller* que, neste caso, aplica o *middleware* a todos os métodos do *controller* que houver.

```
1    Route::get('profile', [UserController::class,
     ↪  'show'])->middleware('auth');
```

Listing 2.5.6 – Exemplo de *middleware* atribuído à rota do controlador

Assim, todas as rotas e *controllers* devem retornar ao navegador uma resposta à

```
use Illuminate\Http\Request;

class UserController extends Controller
{
    public function __construct()
    {
        $this->middleware('auth');
    }

    public function show($id)
    {
        // Lógica para mostrar um usuário específico
    }
}
```

Listing 2.5.7 – Exemplo de *middleware* atribuído ao controlador

solicitação do usuário, de modo que o Laravel fornece várias maneiras diferentes de retornar essas respostas como *strings* e *arrays*, porém, normalmente, o retorno das ações de rotas são as *views* (STAUFFER, 2019).

2.5.3 View

A *View* é a camada de exibição da aplicação, responsável por separar a lógica do *controller* de sua apresentação (LARAVEL, 2023), sendo elas arquivos de saída, armazenados no diretório *resources/views*, que podem conter, a depender da necessidade do projeto, JSON ou XML, e, o mais comum da *Web*, HTML (STAUFFER, 2019). No Laravel há dois formatos de *views*: PHP simples ou *Blade templates*. O arquivo com a extensão *.php* será renderizado com o mecanismo PHP, já o arquivo com extensão *.blade.php* será renderizado com o mecanismo *Blade*.

Os arquivos que utilizam PHP simples possuem a sintaxe padrão da linguagem e escrevem PHP diretamente no arquivo para gerar o HTML, como apresentado no *Listing* 2.5.8, utilizando a *tag* de abertura *<?php ?>*. Neste exemplo, será renderizada uma mensagem com o nome do usuário sempre que a condicional *if($user)* for verdadeira.

Os arquivos *Blade* oferecem maiores funcionalidades quando comparados com o PHP simples, fornecendo uma maneira limpa e concisa de trabalhar com estruturas de controle PHP (LARAVEL, 2023). Essas funcionalidades permitem, além do uso de herança de *template*, ou seja, as *views* podem herdar seções de outras facilitando a reutilização de código, o uso de

```
1     <h1>Bem-vindo</h1>
2     <p>Esse é um exemplo de view utilizando PHP simples</p>
3     <?php if($user): ?>
4         <p>Olá, <?php echo $user->name; ?>!</p>
5     <?php endif; ?>
```

Listing 2.5.8 – Exemplo de uso da sintaxe PHP simples

componentes e *slots*, que são, respectivamente, estruturas reutilizáveis úteis para componentização de partes comuns da aplicação, como cabeçalhos e rodapés, e partes variáveis injetadas no componente que permitem flexibilizar o conteúdo dos componentes, além do uso de diretivas para *loops* e condicionais, ou seja, instruções especiais que permitem incorporar código PHP no arquivo de *template* (STAUFFER, 2019). O *Listing* 2.5.9 apresenta um exemplo de arquivo *Blade* que exibe em tela cada um dos itens de uma lista, dispostos através da *tag* . Além disso, o uso de chaves ({{ }}), na Linha 4, é utilizado para encapsular código PHP e imprimir em tela o valor da variável.

```
1     <h1>Bem-vindo</h1>
2     <p>Esse é um exemplo de view utilizando Blade</p>
3     @foreach($items as $item)
4         <li>{{ $item }}</li>
5     @endforeach
```

Listing 2.5.9 – Exemplo de uso da sintaxe do *Blade*

O mecanismo de *template Blade*, ao contrário de outros mecanismos de *template* PHP, não restringe a usar código PHP simples nas *views*, pois, ao invés de serem executados diretamente pelo servidor *Web*, são compilados em código PHP puro e armazenados em *cache* até serem modificados, ou seja, não é adicionada nenhuma carga adicional ao desempenho da aplicação (*overhead*) (LARAVEL, 2023).

Para criar uma nova *view* através do terminal, utiliza-se o comando *Artisan make:view*, como no *Listing* 2.5.10, que cria uma *view* chamada *users* com a extensão *.blade.php*. Em seguida, depois de criar a *view* é possível exibi-la em tela através da rota utilizando o auxiliar global *view*, como no *Listing* 2.5.11. Neste exemplo, o primeiro argumento corresponde ao nome do arquivo *view*, e o segundo argumento é um *array* de dados disponíveis dentro da *view*, chamado *name*. Como alternativa para passar um *array* de dados completo como argumento da função auxiliar *view*, é possível usar o método *with*, como no *Listing* 2.5.12, que retorna uma

instância do objeto *view* para permitir o encadeamento de métodos antes de retornar a *view*.

```
1    php artisan make:view users
```

Listing 2.5.10 – Comando de exemplo para criar uma *view*

```
1    Route::get('/', function () {
2        return view('user', ['name' => 'Maria']);
3    });
```

Listing 2.5.11 – Exibindo *view* de exemplo através da rota

```
1    Route::get('/', function () {
2        return view('user')
3            ->with('name', 'João')
4            ->with('occupation', 'Professor');
5    });
```

Listing 2.5.12 – Utilizando método *with* na exibição da *view* de exemplo

2.5.4 Model

A camada *Model* do padrão MVC possui os objetos que manipulam os dados da aplicação através do ORM. No `Laravel`, o pacote *Eloquent* é o ORM responsável por implementar o *Model* e se destaca pela sua simplicidade (STAUFFER, 2019). Com o *Eloquent* cada tabela do banco de dados tem uma classe correspondente usada para interagir com essa tabela, permitindo recuperar, inserir, atualizar e excluir registros no banco (LARAVEL, 2023). Os arquivos do *Model* estão localizados no diretório *app/Models* e estendem a classe *Illuminate/Database/Eloquent/Model*.

Além disso, no `Laravel`, é possível definir várias conexões de bancos de dados e posteriormente escolher qual delas será usada por padrão, através da interface fornecida pelo *Eloquent*, permitindo a flexibilidade para trabalhar com bancos de dados (STAUFFER, 2019). O suporte para múltiplas conexões permite ainda usar diferentes conexões de banco para dois tipos diferentes de dados, como, por exemplo, ler dados de um e gravar em outro.

Dessa forma, o `Laravel` torna a interação com o banco de dados simples, e suporta uma variedade de bancos usando `SQL` bruto, um construtor de consultas fluente e o *Eloquent*.

O *framework* fornece suporte, inicialmente, para os bancos de dados: MariaDB[10], MySQL[11], PostgreSQL[12], SQLite[13] e SQL Server[14] (LARAVEL, 2023). As configurações para serviços de bancos de dados estão localizadas no arquivo *config/database.php*. Neste arquivo, é possível definir as conexões com o banco e especificar a conexão padrão da aplicação.

O *Eloquent* fornece um gerenciamento para múltiplos relacionamentos entre tabelas, facilitando o trabalho com esses relacionamentos. Os relacionamentos são definidos como métodos nas classes de modelo do *Eloquent*, e a definição de relacionamento serve como construtor de consultas, fornecendo um poderoso recurso de encadeamento e consulta de métodos. O construtor de consultas de banco de dados do Laravel fornece uma interface propícia e fluente para criar e executar consultas no banco de dados, e funciona com todos os sistemas de banco suportados pelo Laravel (LARAVEL, 2023).

No Laravel são usadas *migrations* como um controle de versão do banco de dados (LARAVEL, 2023). Uma *migration* é um arquivo cuja lógica possibilita criar, modificar ou excluir, por exemplo, uma tabela do banco de dados, e é executada em ordem por data, tornando possível reverter uma *migration* executada. Uma classe de migração contém dois métodos: *up* e *down*. O método *up* é usado para a criação de novas tabelas, colunas ou índices no banco de dados. Já o método *down* reverte as operações realizadas pelo método *up*.

2.5.5 Vantagens do Laravel

Starter Kits: o Laravel fornece *kits* de autenticação e inicialização de aplicativos para oferecer vantagem inicial na sua construção. Esses *kits* estruturam o aplicativo com as rotas, *controllers* e *views* necessárias para registrar e autenticar os usuários (LARAVEL, 2023). Para iniciar rapidamente o sistema de autenticação, o Laravel fornece um *Starter Kit* chamado *Laravel Breeze* que apresenta uma implementação mínima e simples (LARAVEL, 2023) dos recursos de autenticação, como *login*, registro, redefinição de senha e verificação de *e-mail*. A camada *view* do *Laravel Breeze* é composta por *templates Blade* estilizados com Tailwind CSS[15]. Outro *Starter Kit* disponibilizado é o *Laravel Jetstream* que aumenta a funcionalidade do *Laravel Breeze* com recursos que incluem autenticação em dois fatores, gerenciamento de

[10] https://mariadb.org/
[11] https://www.mysql.com/
[12] https://www.postgresql.org/
[13] https://www.sqlite.org/
[14] https://www.microsoft.com/pt-br/sql-server
[15] https://tailwindcss.com/

sessão, dentre outros.

Manipulador de erros: o `Laravel` já possui configurado, ao iniciar um novo projeto, o tratamento de erros e exceções, como por exemplo, as páginas de erro padrão para códigos de *status* HTTP, como 404 e 500 que estão localizadas no diretório *resources/views/erros*. Os erros são registrados pela classe *App/Exceptions/Handler* e em seguida exibidos em tela, enquanto no arquivo *config/app.php*, a opção *debug* determina o quanto de informação sobre o erro será exibida ao usuário. Por padrão, esta opção é configurada como variável de ambiente no arquivo *.env*. Durante o desenvolvimento do projeto, a variável pode ser definida como *true*, porém em ambiente de produção, o valor deve sempre ser *false* para evitar expor valores de configuração confidenciais ao usuário final da aplicação (LARAVEL, 2023).

2.6 Exemplo prático com Laravel

O exemplo apresentado nesta seção é uma aplicação de Sistema de Gerenciamento de Contatos, desenvolvida com o *framework* `Laravel`. Nela é possível adicionar contatos e visualizá-los na página, além da possibilidade de editar e excluir o contato da lista. Através do uso de um banco de dados, as informações não são perdidas mesmo após recarregar a página. As Seções 2.6.1 e 2.6.2 descrevem todo o desenvolvimento da aplicação.

2.6.1 Preparação do ambiente

Para a implementação desse projeto, ao instalar o `Laravel` conforme apresentado no *Listing* 2.4.1, deve-se nomear o projeto como "gerenciamento-de-contatos" substituindo o termo "novo-projeto" apresentado no comando. Uma vez tendo criado o novo projeto, é necessário navegar até o diretório que contém o projeto e executar o comando do *Listing* 2.6.1 para iniciar o servidor local através do *Laravel Artisan*. Em seguida, deve-se acessar o endereço *http://localhost:8000* através do navegador, onde será exibida a página inicial padrão de uma aplicação `Laravel` contendo links úteis e informações como as versões do `Laravel` e do `PHP` utilizadas no projeto, conforme mostra a Figura 13.

```
1    php artisan serve
```

Listing 2.6.1 – Inicializando servidor local do `Laravel`

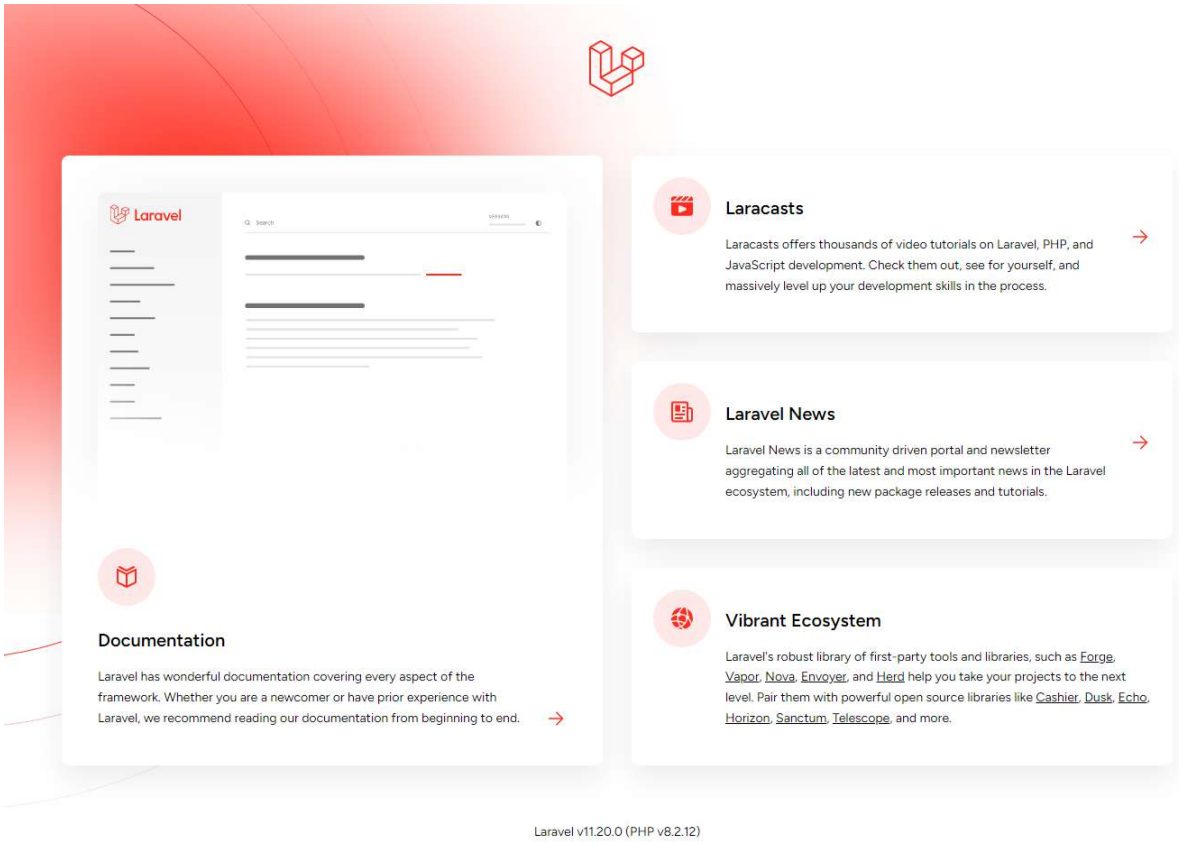


Figura 13 – Tela inicial de uma aplicação `Laravel`

2.6.2 Implementação do exemplo

No arquivo *routes/web.php*, onde são definidas as rotas da aplicação, há uma rota responsável por exibir a página inicial padrão do projeto, que deve ser substituída por uma nova rota chamada */contatos*, conforme o *Listing* 2.6.2 que utiliza o *resource controller* e recebe como argumentos, além da URL, o nome do *controller*, *ContatosController*. Conforme a Linha 4, é necessário que o controlador seja referenciado dentro do arquivo *routes*.

```
1  <?php
2
3  use Illuminate\Support\Facades\Route;
4  use App\Http\Controllers\ContatosController;
5
6  Route::resource('contatos', ContatosController::class);
```

Listing 2.6.2 – Configuração da rota contatos

Para criar o *controller ContatosController* referenciado na rota criada, executa-se no terminal o comando do *Listing* 2.6.3 que, além de criar um novo *controller*, adiciona os métodos

CRUD por padrão. O arquivo *app/Http/Controllers/ContatosController.php* gerado a partir da execução do comando contém uma classe que estende da classe *Controller* nativa do `Laravel` e engloba a lógica do *controller*.

```
1    php artisan make:controller ContatosController --resource
```

Listing 2.6.3 – Criando um novo *controller*

Ao escopo do método *index* é adicionado o conteúdo do *Listing* 2.6.4, que contém um *array* de contatos que, por enquanto, será enviado estaticamente para a *view* de retorno da função. Para criar a *view*, adiciona-se um novo arquivo ao diretório *resources/views* com o nome *listar-contatos.blade.php*. O código correspondente à estrutura da página, apresentado no *Listing* 2.6.5, deve ser colocado no arquivo criado, o qual contém um formulário que recebe o conteúdo para criação de novos contatos e exibe os contatos existentes em uma lista. Na Linha 12, o *token @csrf* é colocado para que o *middleware* de proteção possa validar a solicitação *POST* do formulário.

```
1    public function index() {
2        $contatos = [
3            'João',
4            'Maria'
5        ];
6        return view('listar-contatos', ['contatos' => $contatos]);
7    };
```

Listing 2.6.4 – Adicionando escopo do método *index*

Dada a simplicidade do projeto, o banco de dados utilizado neste exemplo será o `SQLite`, pois este banco de dados necessita de poucas configurações de uso. Assim, para configurar o banco de dados, criar-se um arquivo *database.sqlite* no diretório *database*, e no arquivo de configurações de variáveis de ambiente, *vendor/.env*, apaga-se as configurações DB já contidas no arquivo e é mantida apenas a *DB_CONNECTION* recebendo o valor *sqlite*. Para criar uma nova tabela no banco de dados, executa-se o comando *make:migration* seguido do nome da *migration*, que neste exemplo é chamada de *create_contatos_table*, como no *Listing* 2.6.6. Em seguida é gerado um arquivo no diretório *database/migrations* abaixo dos arquivos advindos com a inicialização do projeto.

No arquivo gerado pela *migration* já há uma coluna *id* de auto incremento e uma

```
<!DOCTYPE html>
<html lang="pt-br">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width,
    ↪ initial-scale=1.0">
    <title>Lista de Contatos</title>
</head>
<body>

    <form action="/contatos" method="POST">
    @csrf
    <input type="text" name='nome' placeholder="Nome">
    <input type="text" name="telefone" id="telefone"
    ↪ placeholder="Telefone">
    <button type="submit">Enviar</button>
</form>

    <ul>
        @foreach ($contatos as $contato)
            <li>
                {{ $contato->nome }} - {{ $contato->telefone }}
            </li>
        @endforeach
    </ul>
</body>
</html>
```

Listing 2.6.5 – Conteúdo da página *listar-contatos.blade.php*

```
php artisan make:migration create_contatos_table
```

Listing 2.6.6 – Criando nova *migration*

função *timestamp* que cria as colunas *created_at* e *updated_at*. Usando o código mostrado no *Listing* 2.6.7, são adicionadas à tabela novas colunas correspondentes ao nome e ao telefone do contato. Na Linha 2 o campo da coluna armazena um nome de, no máximo, 150 caracteres. Na Linha 3 a coluna criada armazena o número de telefone com tipo de dado *string* de até 20 caracteres e permitindo que o campo seja nulo. Para aplicar as definições do modelo, é executado o comando *migrate* do *Listing* 2.6.8.

Para criar um novo modelo através do terminal, executa-se o comando do *Listing* 2.6.9, que gera um novo arquivo no diretório *app/Models* chamado *Contato.php*. O modelo

```
1     $table->id();
2     $table->string('nome', 150);
3     $table->string('telefone', 20)->nullable();
4     $table->timestamps();
```

Listing 2.6.7 – Adicionando novas colunas à tabela

```
1     php artisan migrate
```

Listing 2.6.8 – Executando *migrate*

criado estende a classe *Models* e permite utilizar métodos para manipular o banco de dados. O *Eloquent* irá associar o modelo criado à tabela do banco de dados de mesmo nome no plural criada anteriormente. Além disso, o modelo deve ser importado no arquivo do *controller*, como no *Listing* 2.6.10, para que o `Laravel` o reconheça.

```
1     php artisan make:model Contato
```

Listing 2.6.9 – Criando novo modelo

```
1     use App\Models\Contato;
```

Listing 2.6.10 – Importando modelo

No arquivo *Http/Controller/ContatosControler.php*, após importar o modelo *Contato*, é adicionado o conteúdo do *Listing* 2.6.11 no escopo do método *store* para salvar os contatos no banco de dados. Esse método recebe como argumento uma requisição, e nas Linhas 2 e 3 é atribuído às variáveis *nomeContato* e *telefoneContato* os valores correspondentes do *input* através da requisição. Nas Linhas 5 e 6, as variáveis são atribuídas ao modelo e, em seguida, o contato é salvo no banco de dados. O código da Linha 9 redireciona o usuário à página inicial de exibição de contatos.

```
1     public function store(Request $request) {
2         $nomeContato = $request->input(key:'nome');
3         $telefoneContato = $request->input(key: 'telefone');
4         $contato = new Contato();
5         $contato->nome = $nomeContato;
6         $contato->telefone = $telefoneContato;
7         $contato->save();
8
9         return redirect(to:'/contatos');
10    }
```

Listing 2.6.11 – Salvando contatos no banco de dados

A exibição dos contatos, que inicialmente foi declarada como estática, agora precisa ser modificada para que os dados exibidos sejam obtidos diretamente do banco. Para tanto, o código do *Listing* 2.6.12 é adicionado no escopo do método *index* no arquivo *Http/Controller/-ContatosControler.php*, que na linha 2 faz uma *query SELECT* pelo ORM do `Laravel` para obter todos os contatos salvos no banco e exibe-os na *view listar-contatos*. Agora, no método *index* os contatos estão sendo obtidos do banco de dados através de uma consulta `SQL` usando a sintaxe do *Eloquent*.

```
1     public function index() {
2         $contatos = Contato::all();
3
4         return view('listar-contatos', ['contatos'=>$contatos]);
5     }
```

Listing 2.6.12 – Obtendo e exibindo contatos do banco de dados

Para implementar a funcionalidade de edição de contato, algumas alterações devem ser feitas no arquivo *resources/views/listar-contatos.blade.php*, conforme o *Listing* 2.6.13. Na Linha 22 o elemento *li* da lista recebe, além do nome e o telefone do contato, um *link* de redirecionamento para uma nova *view* a ser criada posteriormente.

No arquivo do *controller Http/Controller/ContatosControler.php*, no método *edit*, é adicionado o conteúdo do *Listing* 2.6.14, que recebe como argumento o *id* do contato a ser editado, e em seu escopo, realiza a busca desse contato no banco de dados. O retorno da busca é armazenado na variável *contato* e o método retorna a nova *view*. No diretório *resource/views* deve ser criado um novo arquivo *editar-contato.blade.php* contendo o código do *Listing* 2.6.15, que possui um formulário de edição do contato utilizando o método HTTP *PUT*.

```
<!DOCTYPE html>
<html lang="pt-br">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width,
    ↪ initial-scale=1.0">
    <title>Lista de Contatos</title>
</head>
<body>

    <form action="/contatos" method="POST">
        @csrf
        <input type="text" name='nome' placeholder="Nome">
        <input type="text" name="telefone" id="telefone"
        ↪ placeholder="Telefone">
        <button type="submit">Enviar</button>
    </form>

    <ul>
        @foreach ($contatos as $contato)
        <li>
            {{ $contato->nome }} - {{ $contato->telefone }}
            <a href="/contatos/{{ $contato->id }}/edit">Editar</a>
        </li>
        @endforeach
    </ul>
</body>
</html>
```

Listing 2.6.13 – Adicionando *link* de edição na página inicial

```
public function edit($id) {
    $contato = Contato::findOrFail($id);
    return view('editar-contato', compact('contato'));
}
```

Listing 2.6.14 – Buscando contato no banco de dados

No arquivo *Http/Controller/ContatosControler.php* adiciona-se ao escopo do método *update* o conteúdo do *Listing* 2.6.16, que recebe como argumento uma requisição e o *id* do contato que será editado. Em seu escopo, na Linha 2, o contato é buscado no banco através do *id* e em seguida os dados do contato são atualizados com aqueles enviados através do formulário de edição. Ao final, as alterações são salvas no banco de dados e o usuário é redirecionado para a página inicial da aplicação.

```
<!DOCTYPE html>
<html lang="pt-br">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width,
    ↪ initial-scale=1.0">
    <title>Editar Contato</title>
</head>
<body>
    <h1>Editar Contato</h1>

    <form action="/contatos/{{ $contato->id }}" method="POST">
        @csrf
        @method('PUT')

        <input type="text" name="nome" value="{{ $contato->nome }}"
        ↪ placeholder="Nome">
        <input type="text" name="telefone" value="{{
        ↪ $contato->telefone }}" placeholder="Telefone">
        <button type="submit">Salvar Alterações</button>
    </form>
</body>
</html>
```

Listing 2.6.15 – Conteúdo da página *editar-contato.blade.php*

```
public function update() {
    $contato = Contato::findOrFail($id);

    $contato->nome = $request->input('nome');
    $contato->telefone = $request->input('telefone');
    $contato->save();

    return redirect('/contatos');
}
```

Listing 2.6.16 – Editando os dados e salvando no banco

Por fim, para implementar a funcionalidade de excluir um contato da lista, no arquivo *resources/views/listar-contatos.blade.php* deve ser adicionado um formulário utilizando o método HTTP *DELETE*, como na Linha 24 do *Listing* 2.6.17, que possui um botão de exclusão. No *controller Http/Controller/ContatosControler.php* deve ser adicionado ao escopo do método *destroy* o conteúdo do *Listing* 2.6.18, que recebe como argumento o *id* do contato que será excluído. Ao final da exclusão o usuário é redirecionado à página inicial.

```
<!DOCTYPE html>
<html lang="pt-br">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width,
    ↪ initial-scale=1.0">
    <title>Lista de Contatos</title>
</head>
<body>

    <form action="/contatos" method="POST">
        @csrf
        <input type="text" name='nome' placeholder="Nome">
        <input type="text" name="telefone" id="telefone"
        ↪ placeholder="Telefone">
        <button type="submit">Enviar</button>
    </form>

    <ul>
        @foreach ($contatos as $contato)
        <li>
            {{ $contato->nome }} - {{ $contato->telefone }}
            <a href="/contatos/{{ $contato->id }}/edit">Editar</a>
        </li>

        <form action="/contatos/{{ $contato->id }}" method="POST"
        ↪ style="display:inline;">
            @csrf
            @method('DELETE')
            <button type="submit" onclick="return confirm('Tem
            ↪ certeza que deseja excluir este
            ↪ contato?')">Excluir</button>
        </form>
        @endforeach
    </ul>
</body>
</html>
```

Listing 2.6.17 – Adicionando formulário de exclusão na página inicial

```
    public function destroy($id) {
        $contato = Contato::findOrFail($id);
        $contato->delete();

        return redirect('/contatos');
    }
```

Listing 2.6.18 – Método de exclusão de contato

Ao final do processo de desenvolvimento da aplicação, a página inicial resultante é mostrada na Figura 14. Para adicionar estilização à página inicial, criar-se um novo arquivo *public/css/style.css* com o conteúdo do *Listing* 2.6.19, e na página inicial adiciona-se no escopo da *tag head* o código do *Listing* 2.6.20. Para estilizar outras páginas da aplicação, o processo é análogo. Assim, o projeto de Sistema de Gerenciamento de Contatos tem a aparência mostrada na Figura 15.

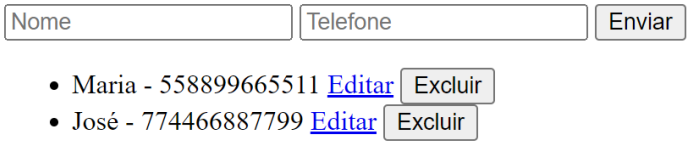


Figura 14 – Fragmento da Página inicial da aplicação

```
    body {
        background-color: #c4e6db;
    }
```

Listing 2.6.19 – Estilização da página inicial

```
    <link rel="stylesheet" href="{{ asset('css/style.css') }}">
```

Listing 2.6.20 – Referenciando arquivo CSS na página inicial

Figura 15 – Fragmento da Página inicial da aplicação estilizada

2.7 Visão geral do Framework Ruby on Rails

O `Ruby on Rails`[16] (doravante, `Rails`) é um *framework Web* gratuito e de código aberto, escrito na linguagem dinâmica orientada a objetos `Ruby`[17], somando uma comunidade de mais de seis mil contribuintes com o código, projetado para compactar a complexidade das aplicações *Web* modernas (RAILS, 2024) tornando-as fáceis de desenvolver, implantar e manter (RUBY; THOMAS, 2023), além de permitir alimentar sites de alto tráfego (VISWANATHAN, 2008). Lançada sua primeira versão em 2004, conforme sua evolução, `Rails` passou a ter um grande conjunto de bibliotecas associadas, entendido como Ecossistema `Rails` (RUBY; THOMAS, 2023), e dispõe de uma estrutura *Full-stack* com ferramentas necessárias ao desenvolvimento de aplicações *Web* no *Fron-end* e no *Back-end*, tais como: renderização de *templates* HTML, atualização de banco de dados, enfileiramento de *jobs* para trabalho assíncrono e fornecimento de sólidas proteções de segurança contra ataques comuns (RAILS, 2024).

Além disso, o `Rails` se mostra uma estrutura promissora para desenvolvimento ágil de aplicações *Web*, se baseando em seis princípios: simplicidade, capacidade de expansão, testabilidade, produtividade, reutilização e capacidade de manutenção. A simplicidade relaciona-se com sua característica de *Scaffolding*, que cria uma base da estrutura com padrões de configuração estabelecidos, reduzindo a complexidade de desenvolvimento e economizando tempo (MICHAEL *et al.*, 2007). A capacidade de expansão permite que o projeto seja escalável de modo que o desenvolvedor possa aprofundar aspectos em particular para maior robustez da aplicação. A testabilidade do `Rails` permite a criação de testes automatizados que facilitam a identificação de erros no código. A produtividade facilita a rápida iniciação de uma tarefa, o que o próprio criador do *framework*, David Heinemeier Hansson, chama de "compressão conceitual" (RUBY; THOMAS, 2023). A reutilização e a capacidade de manutenção de código baseiam-se no princípio de DRY (MICHAEL *et al.*, 2007), tornando o *software* mais confiável e fácil de entender e manter (THOMAS; HUNT, 2019).

Uma vez tendo instalado o `Ruby`, para instalar o `Rails`, deve-se executar no terminal o comando do *Listing* 2.7.1 para que o *framework* seja instalado através do gerenciador de pacotes *RubyGems*. Esse gerenciador de pacotes facilita a criação, compartilhamento e instalação de bibliotecas, sendo instalado por padrão com o `Ruby` (RUBY, 2024). É possível verificar se a instalação foi bem-sucedida executando o comando do *Listing* 2.7.2, onde o retorno esperado é o

[16] https://rubyonrails.org/
[17] https://www.ruby-lang.org/pt/

nome do *framework* seguido da versão instalada, como exemplificado na Linha 2.

```
1    gem install rails
```

Listing 2.7.1 – Comando de instalação

```
1    rails --version
2    % Rails 7.1.0
```

Listing 2.7.2 – Comando de verificação da instalação

2.8 Arquitetura do Ruby on Rails

O *framework* `Rails` segue o padrão arquitetural MVC (*Model*, *View*, *Controller*), que separa a aplicação em três camadas interconectadas (VERMA, 2014), as quais abrangem a interface de usuário, as funcionalidades e o conteúdo de informações (PRESSMAN; MAXIM, 2021). A camada *Model* lida com os dados da aplicação, regras de negócio e lógica de processamento específicos. A camada *View* gera a interface de usuário e possibilita a apresentação do conteúdo e lógica da aplicação. A camada *Controller* gerencia o acesso às camadas *Model* e *View*, coordenando o fluxo de dados entre elas (PRESSMAN; MAXIM, 2021).

No `Rails` as duas principais subestruturas que compõem sua arquitetura são *Action Pack* e *Active Record*, que estabelecem a arquitetura fundamental do padrão MVC para uso do *framework*. Por sua vez, o *Action Pack* é subdividido em três módulos: *Action Dispatch*, *Action Controller* e *Action View* (RUBY; THOMAS, 2023).

A Figura 16 ilustra a arquitetura do *framework* `Rails`, destacando a área em azul para representar os submódulos que compõem o *Action Pack*. Nessa arquitetura, a requisição é feita pelo usuário e enviada para o Servidor *Web*, sendo encaminhada para o *Action Dispatcher*, responsável pelo roteamento da requisição ao controlador apropriado. No *Action Controller* a requisição é enviada através de operações ao *Active Model* e, em seguida, ao *Active Record*. A consulta é realizada ao banco de dados, que, por sua vez, devolve os dados solicitados ou informações sobre erros ao buscar os dados, e a resposta é enviada de volta ao *Action Controller* que agora é responsável por renderizar os dados, enviando-os ao *Action View*. Dentro do *Action Controller*, a requisição pode ser ainda redirecionada por um controlador para outra URL, como,

por exemplo, o usuário ser direcionado para outra página após a realização do *login*. Já no *Action View* é feita a exibição da página para o usuário com os dados da requisição.

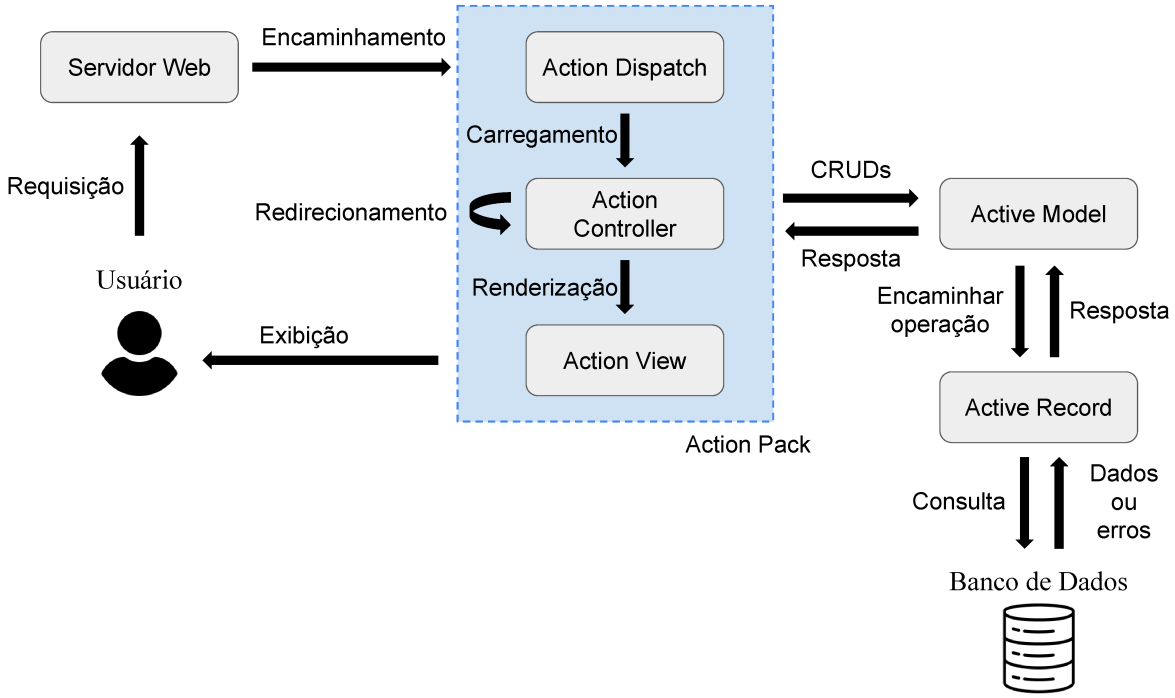


Figura 16 – Arquitetura `Ruby on Rails`. Adaptado de (PATEL, 2023b)

2.8.1 Action Pack

O módulo *Action Pack* abrange as camadas *Controller* e *View* da arquitetura MVC, e a sua subdivisão de módulos em *Action Dispatch*, *Action Controller* e *Action View* fornece suporte para processamento de solicitações e geração de respostas de saída. Embora estes módulos estejam agrupados em um único componente, o `Rails` fornece uma separação clara no código para controle e lógica de apresentação (RUBY; THOMAS, 2023).

O Servidor *Web* padrão do `Rails`, *Puma*, permite abrir uma janela do navegador e, no endereço http://localhost:3000, visualizar e navegar pelas páginas da aplicação, de modo que o servidor não necessite ser reiniciado após alterações feitas nos arquivos, fazendo com que as mudanças sejam captadas automaticamente e exibidas em tela.

O *Action Dispatch* lida com o roteamento das requisições feitas pelo usuário através do Servidor *Web*, analisando e fazendo o processamento relacionado ao HTTP, decodificação de parâmetros, manipulação de lógica de *cache*, *cookies* e sessões. O roteamento se refere às rotas que são responsáveis por definir o caminho percorrido pelo usuário dentro da aplicação, determinando as páginas exibidas e as ações executadas.

O `Rails` permite duas maneiras de definir rotas: uma maneira abrangente e outra conveniente, podendo ainda combinar ambas as abordagens. A forma abrangente permite definir um mapeamento direto de URLs baseado na correspondência padrão, ou seja, definir rotas específicas para diferentes URLs baseadas em expressões regulares (RAILS, 2024). O *Listing* 2.8.1 apresenta um exemplo da forma abrangente de criação de rotas para manipular *posts* de um *blog*, passando o *id* como parâmetro na requisição. A Linha 2 apresenta a rota que utiliza o método *get* para exibir o *post* em tela, na Linha 3 a rota permite a edição do conteúdo do *post*, e na Linha 4 a rota que utiliza o método *delete* exclui o *post*. Já a forma conveniente permite definir rotas baseadas em modelos, gerando rotas necessárias para as *actions*: *index*, *show*, *new*, *edit*, *create*, *update* e *destroy* (RAILS, 2024). Utilizando o mesmo exemplo dos *posts* de um *blog*, o *Listing* 2.8.2 apresenta o caso em que o próprio `Rails` gera automaticamente as rotas necessárias para manipular os *posts*. Após o processamento das rotas, o *Action Dispatch* envia a requisição para o controlador correspondente.

```
1  Rails.application.routes.draw do
2    get '/posts/:id', to: 'posts#show', as: 'post'
3    get '/posts/:id/edit', to: 'posts#edit', as: 'edit_post'
4    delete '/posts/:id', to: 'posts#destroy'
5  end
```

Listing 2.8.1 – Exemplo de definição de rotas da maneira abrangente

```
1  Rails.application.routes.draw do
2    resources :posts
3  end
```

Listing 2.8.2 – Exemplo de definição de rotas da maneira conveniente

O *Action Controller* recebe as solicitações enviadas do *Action Dispatch* e retorna uma visualização para o *Action View*, assim, cada controlador processa uma ação ou objeto e envia a visualização para o navegador. Há ainda, para cada controlador, um diretório associado dentro do diretório *app/views* contendo os arquivos de *template* que serão exibidos em tela a partir da ação do controlador associado (RAILS, 2024).

Além disso, o *Action Controller* pode executar ações através de um pré-filtro (tradução livre) chamado *before_action*, que executa um método antes de executar as ações, e de um pós-filtro (tradução livre) chamado *after_action*, que executa um método adicional após a execu-

ção das ações (MICHAEL *et al.*, 2007). Um exemplo de uso do *before_action* é apresentado no *Listing* 2.8.3 para verificar a autenticação de usuário através do método *authenticate_user!*. Se o usuário estiver autenticado, a ação de exibir perfil, mostrada na Linha 4, poderá ser executada. Já o *Listing* 2.8.4 apresenta um exemplo do uso de *after_action* registrando a hora que determinada ação foi executada. Na Linha 12, a hora atual é registrada após a execução da *action create*.

```
1   class ProfilesController < ApplicationController
2     before_action :authenticate_user!
3
4     def show
5       @user = current_user
6     end
7   end
```

Listing 2.8.3 – Exemplo de uso de *before_action*

```
1   class PostsController < ApplicationController
2     after_action :log_action_time, only: [:create]
3
4     def create
5       # Lógica da action
6     end
7
8     private
9
10    def log_action_time
11      Rails.logger.info "Ação 'create' foi executada em #{Time.now}"
12    end
13  end
```

Listing 2.8.4 – Exemplo de uso de *after_action*

Após o processamento das ações, o *Action Controller* estabelece a conexão com o *Active Record* e fornece dados do banco para a renderização da página solicitada pelo usuário através do submódulo *Action View* responsável por exibir a resposta da requisição no navegador, cuja saída final é uma composição de três elementos: *templates*, *partials* e *layouts*.

Os *templates* são arquivos que permitem a escrita de código `Ruby` dentro do HTML, além de oferecer suporte, também, à linguagem de marcação XML (RAILS, 2024). O *Listing* 2.8.5 mostra um exemplo de *template* que utiliza um *loop* para exibir nomes através das *tags* de incorporação regulares (<% %>) que são usadas para executar código `Ruby` que não necessita

de retorno.

```
1  <h1>Nomes</h1>
2  <% @pessoas.each do |pessoa| %>
3    Nome: <%= pessoa.nome %><br>
4  <% end %>
```

Listing 2.8.5 – Exemplo de *template*

Já os *partials* são partes reutilizáveis de código renderizados na *view* de modo a evitar duplicação de código, podendo ser renderizados mais de uma vez na aplicação, e são nomeados com um prefixo sublinhado (_) (RAILS, 2024), como mostra o *Listing* 2.8.6, ao qual cria uma classe *posts* que contém o título e o corpo do *post*. Já o *Listing* 2.8.7 exemplifica a renderização de um *partial* em uma *view*.

```
1  <div class="post">
2    <h2><%= post.title %></h2>
3    <p><%= post.body %></p>
4  </div>
```

Listing 2.8.6 – Exemplo de um arquivo *partial*

```
1  <h1>Posts</h1>
2  <%= render @post %>
```

Listing 2.8.7 – Exemplo de renderização de um *partial*

Por sua vez, os *layouts* são usados para renderizar *templates* comuns a mais de uma *view* (RAILS, 2024), como, por exemplo, a estrutura comum do HTML que envolve cabeçalhos, rodapés, etc. Como mostra o *Listing* 2.8.8, na Linha 11, o método *yield* indica o local do conteúdo da *view* específica. Assim, ao combinar os três elementos *template*, *partial* e *layout*, é gerada uma *view* que será renderizada, e seu conteúdo é preenchido em tempo de execução.

Dessa forma, para evitar sobrecarregar os *templates* com código `Ruby`, há classes auxiliares chamadas de *helpers*, localizadas no diretório *app/helpers*, que lidam com tarefas comuns como, por exemplo, formulários, formatação de datas e manipulação de *strings* (RAILS, 2024). O *Listing* 2.8.9 apresenta um exemplo de arquivo *helper* chamado *uppercase*, que converte *strings* em letras maiúsculas, e o *Listing* 2.8.10 exemplifica a renderização do *helper uppercase*, em cuja Linha 2 exibe o resultado esperado.

```
<!DOCTYPE html>
<html>
<head>
  <title>Minha página</title>
  <%= csrf_meta_tags %>
  <%= csp_meta_tag %>
  <%= stylesheet_link_tag 'application', media: 'all',
  ↪ 'data-turbolinks-track': 'reload' %>
  <%= javascript_include_tag 'application', 'data-turbolinks-track':
  ↪ 'reload' %>
</head>
<body>
  <%= yield %>
</body>
</html>
```

Listing 2.8.8 – Exemplo de renderização de um *layout*

```
module StringHelpers
  def uppercase(string)
    string.upcase
  end
end
```

Listing 2.8.9 – Exemplo de um arquivo *helper*

```
<%= uppercase("hello world") %>
# HELLO WORLD
```

Listing 2.8.10 – Exemplo de renderização de um *helper*

2.8.2 Active Record

O *Active Record* representa a camada *Model* do MVC e contém a lógica de negócios da aplicação conectando as tabelas do banco de dados com sua representação em classes `Ruby` (RAILS, 2024) e transformando as requisições que vêm do *Action Controller* para comandos SQL (MICHAEL *et al.*, 2007). O módulo *Active Record* segue o padrão ORM, onde as classes estão ligadas diretamente às tabelas do banco de dados, e cada registro é um objeto da classe e cada coluna representa um atributo (MICHAEL *et al.*, 2007), permitindo representar modelos e dados, além de executar operações no banco de dados de maneira orientada a objetos (RAILS, 2024).

Como uma de suas filosofias, o `Rails` possui convenções de configuração apresenta-

das pela própria documentação, e ao segui-las é esperado que haja pouca ou nenhuma escrita de outras configurações. Um exemplo dessa convenção é a pluralização dos nomes das classes para associar à respectiva tabela do banco de dados como, por exemplo, para uma classe *User* será criada uma tabela chamada *users*. Para nomes compostos utiliza-se a forma *CamelCase*. Neste caso, a classe *BookClub* corresponde à tabela *book_clubs*. Além disso, é possível ainda realizar uma associação manual como, por exemplo, associar a classe *Usuario* a uma tabela chamada *usuarios*. Neste caso, `Rails` não associará automaticamente, sendo necessária a configuração manual para associar a classe à tabela correspondente (RAILS, 2024).

O acrônimo CRUD é usado para verbos que operam com dados, são eles: *Create*, *Read*, *Update* e *Delete*. Nativamente, `Rails` permite implementar a funcionalidade CRUD, através do *Active Record*, sem necessidade de configuração, criando automaticamente os métodos que permitem ler e manipular registros do banco de dados. O módulo permite ainda validar o estado de um modelo antes de gravar um dado no banco de dados. Para isso são usados métodos como *create* e *update*, que retornam *false* para métodos inválidos e nenhuma operação é realizada no banco (RAILS, 2024).

2.8.3 Vantagens do Ruby on Rails

Banco de dados: o `Rails` se conecta com diferentes bancos de dados, dentre eles estão `SQLite3`, `DB2`, `MySQL`, `PostgreSQL` e `SQL Server`, sendo necessária, com exceção do `SQLite3`, a instalação de um *driver* de banco de dados (RUBY; THOMAS, 2023). Para cada projeto é possível definir bancos de dados diferentes, de acordo com a capacidade de gerenciamento. Por exemplo, é possível definir um banco de dados para desenvolvimento, outro para teste e outro para produção (MICHAEL *et al.*, 2007).

Convenção sobre configuração: `Rails` possui um conjunto de convenções que são padrões nativos para evitar configurações extras e aumentar a produtividade. Além disso, as convenções facilitam a aprendizagem da linguagem para iniciantes, tornando possível desenvolver aplicações simples com pouca experiência (RAILS, 2024).

2.9 Exemplo prático com Ruby on Rails

O exemplo apresentado nesta seção é uma aplicação de um "*Blog* Simples", desenvolvida com o *framework* `Rails`, onde é possível adicionar *posts* com título e conteúdo, e

listá-los na página principal, além de ser possível editar e excluir cada *post* individualmente. Através do uso de um banco de dados, as informações não serão perdidas mesmo após recarregar a página. As Seções 2.9.1 e 2.9.2 descrevem todo o desenvolvimento da aplicação.

2.9.1 Preparação do ambiente

Após a instalação do `Rails`, deve-se instalar o banco de dados `SQLite3`, que será utilizado no desenvolvimento do projeto. Por padrão `Rails` possui *scripts* projetados para facilitar o desenvolvimento de aplicações, chamados de geradores (RAILS, 2024). Dessa forma, ao iniciar um novo projeto, a utilização de um gerador de aplicações fornece a base necessária de configuração manual. Para isso, no terminal, deve-se navegar até o diretório do projeto e executa-se o comando do *Listing* 2.9.1, o qual cria um projeto que no exemplo aqui apresentado é chamado de *Blog*.

```
1    rails new blog
```

Listing 2.9.1 – Inicializando novo projeto

O novo diretório que contém o projeto possui várias pastas e arquivos gerados que compõem a aplicação, de modo que cada um possui a sua função. As principais pastas são:

- *app*: contém as camadas do modelo MVC;
- *bin*: contém os *scripts* de inicialização da aplicação, além de *scripts* usados para, dentre outros, configurar e atualizar a aplicação;
- *config*: contém as rotas da aplicação;
- *db*: contém o esquema do banco de dados, bem como as *migrations*;
- *public*: contém os arquivos estáticos;
- *README.md*: arquivo de instruções para uso da aplicação.

Após a criação do novo projeto, deve-se navegar até o diretório principal pelo terminal e executar o comando do *Listing* 2.9.2 a fim de inicializar o Servidor *Web*. Em seguida, no navegador, utilizando o endereço http://localhost:3000, é possível visualizar a página inicial do projeto, como exemplifica a Figura 17. A página inicial do `Rails` em uma nova aplicação é chamada de *smoke test* (teste de fumaça), que garante que o *software* esteja corretamente configurado para exibir uma página (RAILS, 2024).

```
ruby bin\rails server
```

Listing 2.9.2 – Inicializando o servidor

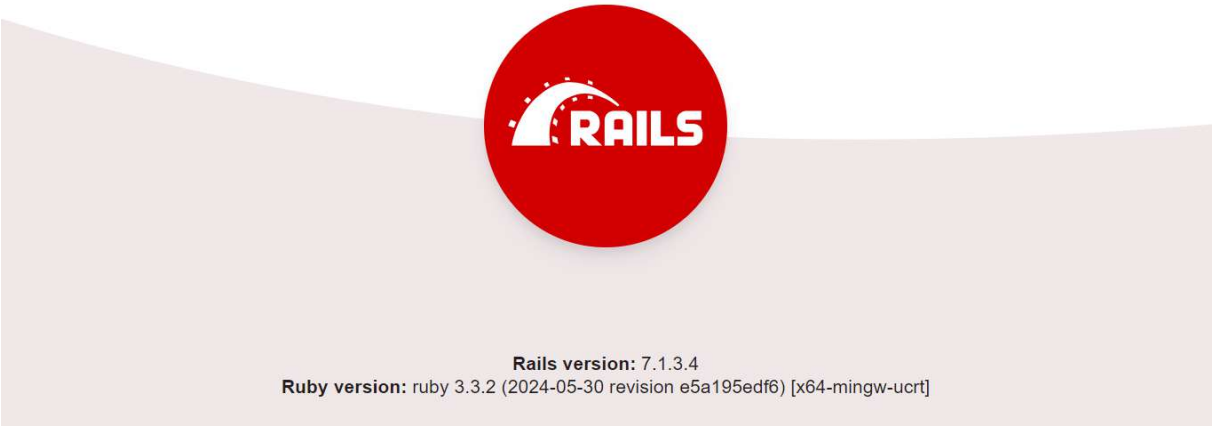


Figura 17 – Página Inicial do `Ruby on Rails`

2.9.2 Implementação do exemplo

Inicialmente é criada uma rota no arquivo *config/routes.rb* que mapeia as solicitações *GET /posts* para a *index* como ação de um controlador, de acordo com o código apresentado no *Listing* 2.9.3. Assim, para criar o *controller*, no terminal, é utilizado o comando do *Listing* 2.9.4, que cria um *controller* chamado *PostsController* e sua ação *index*.

```
Rails.application.routes.draw do
  get "/posts", to: "posts#index"
end
```

Listing 2.9.3 – Criando uma nova rota

```
rails generate controller Posts index
```

Listing 2.9.4 – Criando um novo *controller*

Após a execução do comando, o `Rails` gera vários arquivos e o primeiro a ser editado é o *app/controllers/posts_controller.rb*, que possui em seu código uma ação *index* de escopo, conforme o *Listing* 2.9.5. Quando uma ação não renderiza explicitamente uma *view*,

o `Rails`, por convenção de configuração, irá renderizar uma *view* correspondente ao nome do *controller* e da ação (RAILS, 2024).

```
1  class PostsController < ApplicationController
2    def index
3    end
4  end
```

Listing 2.9.5 – Exemplo de arquivo *PostsController*

As *views* estão localizadas no diretório *app/views*, e a ação *index* é renderizada no arquivo *app/views/posts/index.html.erb* que contém o conteúdo apresentado no *Listing* 2.9.6. Posteriormente, o conteúdo padrão apresentado na página será substituído pelo conteúdo principal da aplicação.

```
1  <h1>Posts#index</h1>
2  <p>Find me in app/views/posts/index.html.erb</p>
```

Listing 2.9.6 – Exemplo de conteúdo do arquivo *index*

Para que a página *index* criada a partir da rota *posts* seja exibida como página inicial, é necessário adicionar essa rota como raiz do projeto, incluindo, no arquivo *config/routes.rb*, a rota *root* no topo do bloco *Rails.application.routes.draw*, conforme apresentado na Linha 2 do *Listing* 2.9.7. A partir disso, a nova página inicial da aplicação deverá conter o fragmento apresentado na Figura 18 com um texto padrão escrito pelo próprio `Rails`.

```
1  Rails.application.routes.draw do
2    root "posts#index"
3
4    get "/posts", to: "posts#index"
5  end
```

Listing 2.9.7 – Adicionando a rota *root*

Figura 18 – Fragmento da página inicial do projeto após a adição da rota *root*

Após a definição do *controller* e da *view*, é feita a criação do *model* da aplicação, seguindo o padrão arquitetural MVC. Em `Rails` os modelos são representados como uma classe e, por convenção, a nomenclatura dos modelos é no singular, pois o modelo instanciado representa um único registro de dados (RAILS, 2024). Um novo modelo pode ser definido executando no terminal a linha de comando do *Listing* 2.9.8, que determina a criação de dois campos, *string* e *text*, respectivamente, para o título e o corpo do *post*.

```
1    rails generate model Post title:string body:text
```

Listing 2.9.8 – Criando modelo *Post*

As *migrations* são um recurso que permite escrever modificações no esquema do banco de dados ao longo do tempo de maneira consistente utilizando uma DSL `Ruby` ao invés de SQL puro. A DSL (*Domain-Specific Language*) é uma linguagem desenvolvida dentro do próprio `Ruby` para descrever as alterações feitas no banco de dados, na qual cada *migration* pode ser interpretada como uma nova "versão" do banco de dados, sendo possível reverter alterações para qualquer ponto do histórico de versões. O arquivo de migração *db/migrate/<timestamp>_create_posts.rb* contém o código que cria no banco de dados a tabela *posts* com os campos desejados. Para que a migração seja implementada é necessário executar o comando do *Listing* 2.9.9, que exibe no terminal uma mensagem confirmando a criação da tabela a partir da *migration*.

```
1    rails db:migrate
```

Listing 2.9.9 – Executando migração

Para o presente projeto são implementadas as operações CRUD, onde a primeira operação a ser implementada é a *Read*, de leitura, realizando a alteração na ação *index* do arquivo *app/controllers/posts_controller.rb* conforme o *Listing* 2.9.10. Na Linha 3 a variável de instância *posts* é declarada precedida do símbolo @ e recebe o modelo e o método *all*, que retorna todos os registros da tabela associada ao modelo *Post*.

As variáveis de instância podem ser acessadas pela *view* no arquivo *app/views/posts/index.html.rb*, substituindo todo o conteúdo do arquivo pelo conteúdo HTML do *Listing* 2.9.11, no qual a Linha 4 faz uso da variável de instância *@posts* para exibir em tela todos os *posts* gravados no banco de dados. O código HTML apresenta ainda, na Linha 1, o título

```
1  class PostsController < ApplicationController
2    def index
3      @posts = Post.all
4    end
5  end
```

Listing 2.9.10 – Adicionando método *index*

da página, seguido, na Linha 3, de uma lista não ordenada que exibe o título de cada *post*. A simbologia <% %> e <%= %> são *tags* regulares que permitem a escrita de código Ruby dentro do HTML.

```
1  <h1>Blog</h1>
2
3  <ul>
4    <% @posts.each do post %>
5      <li>
6        <%= post.title %>
7      </li>
8    <% end %>
9  </ul>
```

Listing 2.9.11 – Conteúdo da página de exibição dos *posts*

Como boa prática, utilizando o fundamento de DRY, é recomendado criar uma *view* compartilhada *partial* com a estrutura comum de uma página HTML, através da criação de um arquivo *app/view/posts/_page_structure.html.erb* com o conteúdo do *Listing* 2.9.12, adicionando, na Linha 11, conteúdo dinamicamente. Dessa forma, todas as páginas da aplicação farão uso do *partial*, inicialmente aplicado na página *app/view/posts/index.html.rb*, como apresenta o *Listing* 2.9.13.

Antes de implementar as demais operações CRUD, é interessante criar uma *view* para visualização individual de cada um dos *posts* criados, começando pela adição de uma nova rota, no arquivo *config/routes.rb*, que mapeia a ação do *controller*, conforme a Linha 5 do *Listing* 2.9.14. Além disso, a nova rota possui um parâmetro *:id* a fim de permitir buscar no banco de dados o *post* correspondente.

```
<!DOCTYPE html>
<html>
<head>
  <title>"Blog"</title>
  <%= stylesheet_link_tag 'application', media: 'all' %>
  <%= javascript_pack_tag 'application' %>
  <%= csrf_meta_tags %>
</head>
<body>
  <div class="container">
    <%= yield %>
  </div>
</body>
</html>
```

Listing 2.9.12 – *Partial* da estrutura comum das páginas HTML

```
<%= render "page_structure" %>
<h1>Blog</h1>

<ul>
  <% @posts.each do post %>
    <li>
      <%= post.title %>
    </li>
  <% end %>
</ul>
```

Listing 2.9.13 – Conteúdo da página de exibição dos *posts* utilizando *partial*

```
Rails.application.routes.draw do
  root "posts#index"

  get "/posts", to: "posts#index"
  get "/posts/:id", to: "posts#show"
end
```

Listing 2.9.14 – Criando nova rota *show*

Essa busca pode ser implementada através da inclusão de uma nova ação no arquivo *app/controllers/posts_controller.rb* do *controller*, como exemplificado pela ação *show* do *Listing* 2.9.15, a qual utiliza o método *find*, que recebe o parâmetro *:id*, recebido como argumento, retorna para a variável de instância *@post* o *post* correspondente. Encontradas as informações do *post* desejado é preciso estruturá-las para exibição em tela, o que pode ser feito através da criação do novo arquivo *app/views/posts/show.html.erb* com o conteúdo apresentado no *Listing* 2.9.16,

que exibe em tela o título e o conteúdo do *post* e um *link* redirecionando à página inicial. Por fim, para redirecionar da página inicial à página de exibição do *post* correspondente, é necessária a modificação no arquivo *app/views/posts/index.html.erb* adicionando a *tag a*, conforme o *Listing* 2.9.17.

```
1   class PostsController < ApplicationController
2     def index
3       @posts = Post.all
4     end
5     def show
6       @post = Post.find(params[:id])
7     end
8   end
```

Listing 2.9.15 – Criando nova ação *show* no *controller*

```
1   <%= render "page_structure" %>
2   <h1><%= @post.title %></h1>
3
4   <p><%= @post.body %></p>
5
6   <ul>
7     <li>
8       <a href="/posts">
9           Voltar para página inicial
10      </a>
11    </li>
12  </ul>
```

Listing 2.9.16 – Criando página de exibição de *post*

```
<% content_for :title, "Blog" %>

<%= render "page_structure" %>
<h1>Blog</h1>

<ul>
  <% @posts.each do post %>
    <li>
        <a href="/posts/<%= post.id %>">
            <%= post.title %>
        </a>
    </li>
  <% end %>
</ul>
```

Listing 2.9.17 – Adicionando *link* aos títulos dos *posts*

Por padrão, na aplicação, as entidades que combinam rotas, ações de *controller* e *views* para realizarem operações CRUD são chamadas de recursos. Nesta aplicação, os *posts* são entendidos como um recurso, e para se utilizar dele, o `Rails` fornece um método chamado *resources* que mapeia todas as rotas convencionais para um recurso, realizando uma substituição no arquivo *config/routes.rb* conforme o *Listing* 2.9.18. A inspeção das rotas mapeadas é feita através do comando *rails routes*, cujo retorno é ilustrado no *Listing* 2.9.19.

```
Rails.application.routes.draw do
  root "posts#index"

  resources :posts
end
```

Listing 2.9.18 – Criando o método *resources*

```
1   rails routes
2  #    Prefix Verb   URI Pattern                  Controller#Action
3  #      root GET    /                            posts#index
4  #     posts GET    /posts(.:format)             posts#index
5  #  new_post GET    /posts/new(.:format)         posts#new
6  #      post GET    /posts/:id(.:format)         posts#show
7  #           POST   /posts(.:format)             posts#create
8  # edit_post GET    /posts/:id/edit(.:format)    posts#edit
9  #           PATCH  /posts/:id(.:format)         posts#update
10 #           DELETE /posts/:id(.:format)         posts#destroy
```

Listing 2.9.19 – Mapeando rotas criadas pelo método *resources*

Após a substituição no arquivo *config/routes.rb*, é necessário ainda alterar o arquivo *app/views/posts/index.html.erb*, conforme o *Listing* 2.9.20, que adiciona o *helper link_to* (Linha 7) para renderizar um *link*, o qual tem como seu primeiro argumento o texto do *link* e como segundo argumento o destino do *link*.

```
1   <%= render "page_structure" %>
2   <h1>Blog</h1>
3
4   <ul>
5     <% @posts.each do post %>
6       <li>
7           <%= link_to post.title, post %>
8       </li>
9     <% end %>
10  </ul>
```

Listing 2.9.20 – Adicionando o *helper link_to*

A seguir a implementação da operação *Create* do CRUD é feita em etapas. Inicialmente, conforme o *Listing* 2.9.21, no arquivo *app/controllers/posts_controller.rb*, abaixo da ação *show*, é implementada a ação *new*, que instancia um novo *post*, mas não o salva, sendo isto atribuído à ação *create*, instanciada em seguida. Na Linha 13 é feita a tentativa de salvar o *post*. Se a operação for realizada com sucesso, a ação redireciona o navegador para a página inicial listando o novo *post*, conforme a Linha 14. Se o salvamento falhar, conforme a Linha 16, a ação exibe novamente o formulário com código de *status* 422.

A etapa de criação de um novo *post* envolve, ainda, um formulário para obtenção dos dados, que pode ser criado utilizando um recurso chamado construtor de formulário (*form builder*), que permite construir um formulário totalmente configurado com uma quantidade

```
1   class PostsController < ApplicationController
2     def index
3       @post = Post.all
4     end
5     def show
6       @post = Post.find(params[:id])
7     end
8     def new
9       @post = Post.new
10    end
11    def create
12      @post = Post.new(title: "...", body: "...")
13      if @post.save
14        redirect_to root_path
15      else
16        render :new, status: :unprocessable_entity
17      end
18    end
19  end
```

Listing 2.9.21 – Criando ações *new* e *create* no *controller*

mínima de código, conforme exemplifica o *Listing* 2.9.22, que constitui o conteúdo do novo arquivo *app/views/posts/new.html.erb*, no qual o método auxiliar *form_with*, da Linha 4, instancia um construtor de formulário e os métodos *label*, *text_field* e *text_area* são usados para gerar elementos apropriados para o formulário.

```
1   <%= render "page_structure" %>
2   <h1>Novo Post</h1>
3
4   <%= form_with model: @post do form %>
5     <div>
6       <%= form.label "Título" %><br>
7       <%= form.text_field :title %>
8     </div>
9
10    <div>
11      <%= form.label "Conteúdo" %><br>
12      <%= form.text_area :body %>
13    </div>
14
15    <div>
16      <%= form.submit "Salvar" %>
17    </div>
18  <% end %>
```

Listing 2.9.22 – Formulário de criação de novo *post*

Os dados obtidos através do formulário são enviados para a ação *create* por parâmetros de *hash* junto com os parâmetros de rota, podendo ser acessados, no caso do título, por *params[:post][:title]*, e no caso do corpo, por *params[:post][:body]*. Porém, esse tipo de acesso individual aos valores é suscetível a erro, então usa-se um único *hash* contendo os valores junto com um filtro chamado *Strong Parameters* para filtrar os parâmetros e evitar que um usuário mal-intencionado envie campos extras no formulário ou substitua dados privados. Para implementar o *Strong Parameters*, pode-se adicionar um método privado no final do escopo da classe, no arquivo *app/controllers/posts_controller.rb*, conforme o *Listing* 2.9.23. Além disso, na Linha 11, no método *create*, o argumento é alterado para o método privado *post_params*.

Outra etapa do processo de implementação do método *Create* é a validação dos dados de entrada do usuário. Para isso, o `Rails` fornece um recurso que auxilia nessa validação chamado *validations*, que verifica o objeto de modelo antes de salvar no banco de dados. Se durante a verificação houver falha, o salvamento é abortado e mensagens de erro correspondentes são apresentadas em tela. A implementação desse recurso, através do arquivo *app/models/post.rb*, é ilustrada no *Listing* 2.9.24, que na Linha 2 valida a variável *title* tornando obrigatória a presença de pelo menos um caractere. Analogamente, na Linha 3, a validação determina a necessidade de conteúdo, porém com um mínimo de 10 caracteres.

Naturalmente, a fim de que os testes de validação repercutam visualmente, o arquivo

```ruby
class PostsController < ApplicationController
  def index
    @post = Post.all
  end
  def show
    @post = Post.find(params[:id])
  end
  def new
    @post = Post.new
  end
  def create
    @post = Post.new(post_params)
    if @post.save
      redirect_to root_path
    else
      render :new, status: :unprocessable_entity
    end
  end

  private
    def post_params
      params.require(:post).permit(:title, :body)
    end
end
```

Listing 2.9.23 – Adicionando *Strong Parameters*

```ruby
class Post < ApplicationRecord
    validates :title, presence: { message: "O post deve conter um
    ↪ título" }
    validates :body, presence: { message: "O conteúdo do post deve
    ↪ conter no mínimo 10 caracteres" }, length: { minimum: 10,
    ↪ message: "O conteúdo do post deve conter no mínimo 10
    ↪ caracteres" }
end
```

Listing 2.9.24 – Adicionando *Validations*

app/views/posts/new.html.erb também deve ser modificado para que exiba uma mensagem de erro para *title* e *body*, conforme as Linhas 8 e 16 do *Listing* 2.9.25, através do método *full_messages_for* que retorna a mensagem de erro.

```
<%= render "page_structure" %>
<h1>Novo Post</h1>

<%= form_with model: @post do form %>
  <div>
    <%= form.label "Título" %><br>
    <%= form.text_field :title %>
    <% @post.errors.full_messages_for(:title).each do message %>
      <div><%= message %></div>
    <% end %>
  </div>

  <div>
    <%= form.label "Conteúdo" %><br>
    <%= form.text_area :body %><br>
    <% @post.errors.full_messages_for(:body).each do message %>
      <div><%= message %></div>
    <% end %>
  </div>

  <div>
    <%= form.submit "Salvar" %>
  </div>
<% end %>
```

Listing 2.9.25 – Adicionando método *full_messages_for* ao formulário de criação de novo *post*

Para finalizar a implementação do método *Create*, é preciso fornecer ao usuário o meio necessário para executar o formulário, o que é feito adicionando no arquivo *app/views/-posts/index.html.erb* um *link* direcionado para a página de criação de um novo *post* conforme a Linha 4 do *Listing* 2.9.26. A partir da criação de novos *posts*, a página inicial deve corresponder ao exemplo da Figura 19.

```
1     <%= render "page_structure" %>
2     <h1>Blog</h1>
3
4     <%= link_to "Criar novo Post", new_post_path %>
5
6     <ul>
7       <% @posts.each do post %>
8         <li>
9             <%= link_to post.title, post %>
10        </li>
11      <% end %>
12    </ul>
```

Listing 2.9.26 – Adicionando *link* para criação de um novo *post* na página inicial

Figura 19 – Fragmento da página inicial do projeto após a adição do *link* de criação de novo *post*

Para implementar o método *Update* do CRUD, assim como no método *Create*, o processo é dividido em etapas. Inicialmente, adiciona-se as ações *edit* e *update* no *controller*, ao arquivo *app/controllers/posts_controller.rb*, abaixo da ação *create*, conforme o *Listing* 2.9.27. Na Linha 25, a ação *edit* busca no banco de dados o *post* correspondente ao *id* enviado como argumento e o armazena na variável de instância @*post*. Já na Linha 24, os dados do formulário são validados e, se a atualização for bem-sucedida, a ação redireciona o usuário para a página de exibição do *post* editado; caso contrário, a ação exibe o formulário com uma mensagem de erro.

```ruby
class PostsController < ApplicationController
  def index
    @post = Post.all
  end
  def show
    @post = Post.find(params[:id])
  end
  def new
    @post = Post.new
  end
  def create
    @post = Post.new(post_params)
    if @post.save
      redirect_to root_path
    else
      render :new, status: :unprocessable_entity
    end
  end
  def edit
    @post = Post.find(params[:id])
  end
  def update
    @post = Post.find(params[:id])
    if @post.update(post_params)
      redirect_to @post
    else
      render :edit, status: :unprocessable_entity
    end
  end

  private
    def post_params
      params.require(:post).permit(:title, :body)
    end
end
```

Listing 2.9.27 – Criando ações *edit* e *update* no *controller*

O formulário de atualização do método *edit* é análogo ao do método *new*, até mesmo em código através do novo arquivo *app/views/posts/_form.html.erb* cujo conteúdo é mostrado no *Listing* 2.9.28. Observa-se que a única diferença para o arquivo *app/views/posts/new.html.erb* é a substituição do uso da variável de instância *@post* por *post* como uma recomendação de boa prática do próprio *framework*, tornando-se em uma *view* compartilhada, ou seja, um *partial*, o que repercutirá nos arquivos *app/views/posts/new.html.erb* e *app/views/posts/edit.html.erb*, como mostram o *Listing* 2.9.29 e o *Listing* 2.9.30.

```
<%= form_with model: post do form %>
  <div>
    <%= form.label "Título" %><br>
    <%= form.text_field :title %>
    <% post.errors.full_messages_for(:title).each do message %>
      <div><%= message %></div>
    <% end %>
  </div>

  <div>
    <%= form.label "Conteúdo" %><br>
    <%= form.text_area :body %><br>
    <% post.errors.full_messages_for(:body).each do message %>
      <div><%= message %></div>
    <% end %>
  </div>

  <div>
    <%= form.submit "Salvar" %>
  </div>
<% end %>
```

Listing 2.9.28 – *Partial* do formulário de edição e criação de um *post*

```
<%= render "page_structure"  %>
<h1>Novo Post</h1>

<%= render "form", post: @post %>
```

Listing 2.9.29 – Atualizando página de criação de novo *post* utilizando *partial*

```
<%= render "page_structure"  %>
<h1>Editar Post</h1>

<%= render "form", post: @post %>
```

Listing 2.9.30 – Atualizando página de edição de *post* utilizando *partial*

Para dar acesso à funcionalidade de atualização do *post*, é adicionado a sua página de visualização um *link* para edição, no arquivo *app/views/posts/show.html.erb*, como no *Listing* 2.9.31. O *link* para retornar à página inicial também é alterado para uso do *link_to*, conforme a Linha 7. A página de visualização deve corresponder ao exemplo da Figura 20.

Para finalizar a implementação das operações CRUD, é implementada a operação *Delete* para a exclusão de um recurso. Para isso, são necessárias apenas uma rota e uma ação

```
1     <%= render "page_structure"  %>
2     <h1><%= @post.title %></h1>
3
4     <p><%= @post.body %></p>
5
6     <ul>
7       <li><%= link_to "Voltar para página inicial", root_path %></li>
8       <li><%= link_to "Editar post", edit_post_path(@post) %></li>
9     </ul>
```

Listing 2.9.31 – Adicionando *link* para edição do *post*

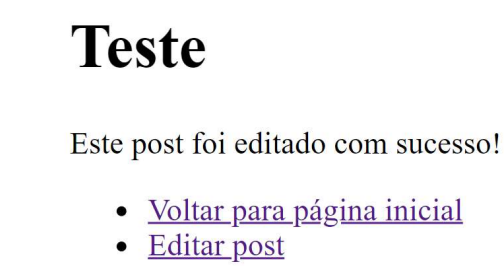


Figura 20 – Fragmento da página de visualização de um *post*

do *controller*. Com o uso do *resources*, o `Rails` já mapeia a rota *DELETE /posts/:id* para as solicitações da ação *destroy*. Então, no arquivo *app/controller/posts_controller.rb*, abaixo da ação *update*, é adicionado o método *destroy*, como no *Listing* 2.9.32. Na Linha 31, a ação busca no banco de dados o *post* correspondente ao *id* passado como argumento; na Linha 32 chama o método *destroy* para excluir o *post* do banco de dados, e na Linha 33 o usuário é redirecionado à página inicial.

```
1   class PostsController < ApplicationController
2     def index
3       @post = Post.all
4     end
5     def show
6       @post = Post.find(params[:id])
7     end
8     def new
9       @post = Post.new
10    end
11    def create
12      @post = Post.new(post_params)
13      if @post.save
14        redirect_to root_path
15      else
16        render :new, status: :unprocessable_entity
17      end
18    end
19    def edit
20      @post = Post.find(params[:id])
21    end
22    def update
23      @post = Post.find(params[:id])
24      if @post.update(post_params)
25        redirect_to @post
26      else
27        render :edit, status: :unprocessable_entity
28      end
29    end
30    def destroy
31      @post = Post.find(params[:id])
32      @post.destroy
33      redirect_to root_path, status: :see_other
34    end
35
36    private
37      def post_params
38        params.require(:post).permit(:title, :body)
39      end
40  end
```

Listing 2.9.32 – Criando ação *destroy* no *controller*

Para disponibilizar a opção de exclusão ao usuário, adiciona-se um *link* "Excluir post" no arquivo *app/views/posts/show.html.erb*, conforme o *Listing* 2.9.33, na Linha 9, que executa o método *delete* e envia ao usuário uma confirmação de exclusão do *post*. A nova página

de visualização de *post* deverá corresponder ao exemplo da Figura 21.

```
1   <%= render "page_structure"  %>
2   <h1><%= @post.title %></h1>
3
4   <p><%= @post.body %></p>
5
6   <ul>
7     <li><%= link_to "Voltar para página inicial", root_path %></li>
8     <li><%= link_to "Editar post", edit_post_path(@post) %></li>
9     <li><%= link_to "Excluir post", post_path(@post), data: {
10                      turbo_method: :delete,
11                      turbo_confirm: "Você tem certeza que deseja
                        ↪  excluir esse post?"
12                    } %></li>
13  </ul>
```

Listing 2.9.33 – Adicionando *link* para exclusão do *post*

Figura 21 – Fragmento da página de visualização de um *post* após adição do *link* de exclusão

Por fim, para estilizar a página *index*, cria-se um arquivo estático *app/assets/stylesheets/index.css* com o código do *Listing* 2.9.34, que apresenta um exemplo de estilização da página, podendo ser adicionadas outras alterações conforme desejado.

```
1   body {
2       background-color: #c4e6db;
3   }
```

Listing 2.9.34 – Estilização da página *index*

Assim, para aplicar a estilização na página, é inserido o código da Linha 1 no topo da página *app/view/posts/index.html.erb*, conforme o *Listing* 2.9.35. Para estilizar as demais

páginas da aplicação, segue-se o mesmo padrão. Assim, o projeto "*Blog* Simples" resultará como na Figura 22.

```
<%= stylesheet_link_tag 'index.css' %>

<%= render "page_structure" %>
<h1>Blog</h1>

<%= link_to "Criar novo Post", new_post_path %>

<ul>
  <% @posts.each do post %>
    <li>
        <%= link_to post.title, post %>
    </li>
  <% end %>
</ul>
```

Listing 2.9.35 – Adicionando arquivo de estilização na página inicial

Figura 22 – Fragmento da página inicial estilizada

2.10 Atividades de Extensão

O uso da palavra "extensão" se origina no processo educacional da Inglaterra, na segunda metade do século XIX, em Oxford e Cambridge, a partir da iniciativa de professores das duas universidades para encontrar uma maneira de atender às necessidades de comunidades do entorno da universidade, resultando na ministração de palestras para mulheres, trabalhadores urbanos e rurais, e ensinando sobre literatura, questões sociais e agrícolas (FIGUEIREDO, 2020). Atualmente, as atividades de extensão universitária são compreendidas como ações na forma de componentes curriculares integrados à matriz curricular que contemple o diálogo construtivo por meio da interação da universidade com a comunidade buscando articular pesquisa,

ensino e extensão para a formação dos discentes, sendo realizadas fora da delimitação física da IES (BRASIL, 2018).

A pesquisa, o ensino e a extensão no ensino superior são responsáveis pela formação de capital intelectual de qualquer país, pois beneficiam o sujeito em formação, que melhora e aumenta seu aprendizado, bem como à comunidade que se beneficia dos serviços prestados pela IES, de modo que a pesquisa não se resumem em apenas pesquisa de revisão de literatura, contemplando também a pesquisa de campo e contribuindo para a construção do conhecimento com vivência na prática, tornando eficiente o processo de aprendizagem (FIGUEIREDO, 2020).

A extensão universitária promove, também, uma formação acadêmica eficaz para atuação do discente no mercado de trabalho, para que este se torne capaz de encontrar soluções para problemas em sua área de atuação profissional de maneira crítica e analítica, promovendo a habilidade de aprender enquanto pratica na comunidade o que foi estudado em sala de aula, pois o objetivo da IES deve ser formar cidadãos éticos e humanísticos para o mercado de trabalho e para a vida (FIGUEIREDO, 2020).

Segundo a Resolução Nº 7, de 18 de dezembro de 2018, as atividades de extensão são regulamentadas pelas Diretrizes para a Extensão na Educação Superior Brasileira considerando os documentos normativos próprios dos cursos de graduação compondo, no mínimo, 10% (dez por cento) do total da carga horária curricular estudantil, se inserindo nas seguintes modalidades: programas, projetos, cursos e oficinas, eventos e prestação de serviços (BRASIL, 2018). Assim, como forma de reconhecimento formativo dos estudantes a participação nas atividades de extensão deve ser adequadamente registrada, documentada e analisada, permitindo ao discente a obtenção de créditos curriculares ou carga horária equivalente após a devida avaliação (BRASIL, 2018). A fim de que haja o devido registro dos documentos, surge a demanda de um *software* dedicado à submissão, avaliação e certificação da documentação dos discentes.

3 METODOLOGIA

Este capítulo apresenta a metodologia utilizada no trabalho, na seguinte ordem: a Seção 3.1 apresenta uma visão geral do *software* desenvolvido, além de apresentar os casos de uso, os atores, o diagrama de caso de uso e a prototipagem do *software*. Já a Seção 3.2 apresenta a implementação das funcionalidades, a discussão dos pontos de comparação e, por fim, a documentação das lições aprendidas.

3.1 Software de Extensão

Durante a realização deste trabalho, desenvolveu-se um *software* que viabiliza a avaliação de documentos comprobatórios da participação de discentes em projetos de extensão, tendo por objetivo apoiar as atividades de extensão e auxiliar os docentes na avaliação da documentação. Dessa maneira, o estudante submete a documentação que certifica sua participação em um ou mais projetos de extensão e a banca avaliadora formada por docentes do curso verifica a veracidade dos documentos. Caso aprovada a documentação, as horas solicitadas serão devidamente registradas em um banco de dados e integralizadas no histórico escolar do discente, sendo esta aplicação incorporada ao sistema de gestão acadêmica da IES, para que haja permissão de acesso à lista de docentes e discentes do curso e para que a aplicação seja administrada pela própria IES.

Com a finalidade de planejamento do *software*, utilizou-se a Resolução Nº 7 que regula as atividades de extensão como um guia de apoio na etapa precedente ao desenvolvimento, denominada levantamento de requisitos. Para isso, foram realizadas reuniões com o então vice-coordenador do curso de Engenharia de Computação da UFC do Campus de Sobral, adotando uma visão analítica para identificar as necessidades do *software*, de modo a garantir uma boa usabilidade e eficiência do sistema. Outrossim, a disposição dos elementos na interface de interação com o usuário foi estrategicamente identificada para proporcionar uma interface intuitiva.

A partir dessas discussões, foram definidos os requisitos do *software* alinhando a proposta do sistema às demandas da IES. Dessa forma, o levantamento de requisitos teve por objetivos: identificar problemas, propor soluções viáveis e especificar um conjunto preliminar de abordagens para diferentes cenários. A seguir, são detalhadas as etapas de planejamento do *software* de extensão.

Caso de Uso: um caso de uso conta com uma descrição geral sobre o desempenho de um usuário final em uma série de papéis possíveis, além de ser observada a interação do usuário com o sistema sob determinadas circunstâncias, ou seja: um caso de uso representa o software do ponto de vista do usuário final (PRESSMAN; MAXIM, 2021). Dessa forma, os casos de uso foram determinados a partir da definição dos atores.

Atores: a primeira etapa que envolve a criação do caso de uso é a definição do conjunto de atores, que são os possíveis papéis que o usuário final pode desempenhar enquanto o sistema opera, ou seja: se comunica com o sistema e que é externo ao sistema em si. A partir disso, para definir os atores envolvidos, foram considerados os docentes e discentes do curso de Engenharia de Computação da UFC do Campus de Sobral, bem como seus níveis de permissões de acesso, isto é, cada ator possui um conjunto específico de funcionalidades que atendem às suas necessidades de uso.

Diagrama de Caso de Uso: de modo a ilustrar as funcionalidades, projetou-se o diagrama de caso de uso, que detalha as interações entre os atores e o *software* através do uso de símbolos e conectores para descrever as funcionalidades e suas interações com o sistema. Nesse diagrama, foram listadas todas as funcionalidades, além de detalhar quais atores possuem permissão de acessá-las, proporcionando uma visão clara e detalhada das operações. Essa abordagem facilita, ainda, modificações futuras ou expansão do *software*.

Descrição do Caso de Uso: para melhor detalhar o diagrama de caso de uso, foi realizada uma descrição textual das funcionalidades, apresentando uma identificação única para cada uma, bem como os respectivos requisitos funcionais, detalhando o comportamento esperado do sistema mediante as interações com o usuário.

Prototipagem do Software: os protótipos representam uma versão simplificada do *software* que permite implementar aspectos e funcionalidades projetadas para o sistema (SOARES, 2008), além de possibilitar uma validação da usabilidade do usuário antes da implementação em código. A partir disso, para auxiliar a visualização da interface, foi realizada a prototipagem do *software* de extensão, elaborando e testando a interface do usuário de modo visual e interativo utilizando a ferramenta de *design* `Figma`[1]. Assim, através dessa ferramenta, foram criadas as telas de acesso para cada ator, incluindo as respectivas funcionalidades de acesso.

[1] https://www.figma.com/

3.2 Comparação dos frameworks

Implementação das funcionalidades: o desenvolvimento do *software* teve o intuito de servir como base de comparação entre os *frameworks* `Ruby on Rails`, `Laravel` e `Django`. Dessa forma, embora tenha sido projetado por completo, o objetivo principal não buscava a implementação integral da aplicação. Em vez disso, foram escolhidas duas funcionalidades importantes para serem implementadas utilizando cada um dos *frameworks*, permitindo uma análise comparativa entre eles.

Discussão dos pontos de comparação: nesta etapa, foram analisados e discutidos aspectos dos *frameworks* a partir da implementação das funcionalidades escolhidas. Os pontos de comparação escolhidos foram determinados com o objetivo de avaliar a eficiência e usabilidade dos *frameworks* estudados. Os pontos de comparação considerados foram:

- **Curva de aprendizado:** nível de dificuldade observado para começar a desenvolver um projeto com cada *framework*;
- **Tamanho dos arquivos:** impacto do *framework* no tamanho final do projeto, incluindo dependências e arquivos gerados durante o desenvolvimento;
- **Verbosidade do *framework*:** quantidade de código necessário para implementar uma tarefa;
- **Documentação de apoio:** qualidade e suporte oferecido pelos materiais e tutoriais disponíveis em canais oficiais da tecnologia;
- **Comunidade para tirar dúvidas:** atividade da comunidade de desenvolvedores, além da facilidade de encontrar suporte em fóruns *online*;
- **Camadas de segurança:** ferramentas e padrões de segurança implementados por cada tecnologia para proteger os dados e navegação na aplicação;
- **Utilização no mercado:** popularidade e adoção dos *frameworks* no mercado considerando o uso em projetos reais e comerciais.

Esses critérios permitiram uma avaliação geral do *framework*, fornecendo uma base sólida para comparar as vantagens e desafios na implementação de sistemas semelhantes ao projetado.

Documentação das lições aprendidas: por fim, foram registrados os principais aprendizados adquiridos ao longo da realização deste trabalho. A partir da implementação das funcionalidades nos diferentes *frameworks*, foi possível destacar aspectos que influenciam diretamente no processo de desenvolvimento. Dessa maneira, o registro dessas lições permitiu,

não apenas aprimorar o desenvolvimento do *software* neste trabalho, como também fornecer guias para projetos futuros, contribuindo com a tomada de decisão em relação à escolha da tecnologia de desenvolvimento.

A Figura 23 apresenta o fluxo adotado na realização da metodologia deste trabalho seguindo uma abordagem estruturada do sistema. Inicialmente, realizou-se o levantamento de requisitos, etapa onde são definidas as funcionalidades do *software*. Em seguida, elaborou-se o diagrama de caso de uso, onde são representadas graficamente as interações entre os atores e o *software*. A partir do diagrama realizou-se a prototipagem do sistema, permitindo uma visualização de sua interface. A fase seguinte foi a implementação das funcionalidades escolhidas, para, em seguida, realizar a comparação entre os *frameworks*, analisando, a partir de pontos de comparação, as vantagens e desafios apresentados por cada um. Por fim, documentou-se as lições aprendidas durante a realização do trabalho.

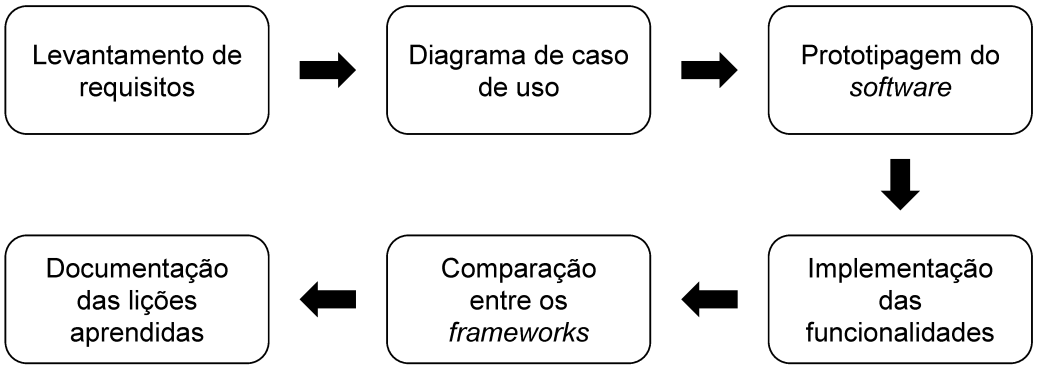


Figura 23 – Fluxo metodológico

4 RESULTADOS

Este capítulo apresenta os resultados obtidos a partir da implementação da metodologia detalhada no Capítulo 3, na seguinte ordem: a Seção 4.1 apresenta os atores do caso de uso, a Seção 4.2 apresenta o diagrama e a descrição do caso de uso, a Seção 4.3 apresenta a prototipagem do sistema, a Seção 4.4 apresenta a implementação das funcionalidades, a Seção 4.5 apresenta os pontos de comparação entre os *frameworks* e, por fim, a Seção 4.6 apresenta a documentação das lições aprendidas.

4.1 Atores

O *software* de extensão foi projetado para dispor as funcionalidades acessíveis por hierarquia, como apresenta a Tabela 2 por meio da definição dos atores e suas respectivas descrições. Assim, a partir dessa diferenciação entre os papéis é possível garantir que cada usuário tenha acesso apenas às funcionalidades pertinentes à sua atuação dentro do *software*, evitando acessos indevidos entre usuários.

Tabela 2 – Descrição dos atores de casos de uso

Ator	Descrição
Aluno	Estudante do curso
Professor	Professor do curso
Banca	Professor avaliador das atividades submetidas
Coordenador	Coordenador do curso

Dessa forma, a definição dos atores é fundamental para estruturar, de forma clara e eficiente, as permissões do sistema, isto é, o ator “Aluno” representa o estudante que submete a documentação comprobatória de participação nas atividades de extensão, enquanto o ator “Banca” refere-se ao professor avaliador desses documentos. Por sua vez, o ator “Coordenador” representa o coordenador do curso e possui o maior nível de permissão dentro do *software*, permitindo a gestão e organização do processo de avaliação.

Portanto, essa estrutura hierárquica contribui para uma navegação mais intuitiva e segura, garantindo a execução de funções apenas por usuários autorizados, além de reduzir o risco de inconsistências no fluxo de dados e na interação entre os usuários.

4.2 Diagrama de Caso de Uso

O diagrama de caso de uso é uma ferramenta utilizada na modelagem do fluxo de eventos do *software* como uma representação visual das interações entre os atores e as funcionalidades determinadas no levantamento de requisitos. Assim, para o *software* de extensão, o diagrama foi construído baseado no levantamento feito junto ao vice-coordenador do curso de Engenharia de Computação da UFC - Campus Sobral.

Com essas definições, foi possível estruturar o diagrama de caso de uso, mostrado na Figura 24, onde cada ator é representado, cada elipse corresponde a uma funcionalidade específica do sistema, e cada linha de conexão indica quais funcionalidades são acessíveis para cada ator.

Um exemplo presente no diagrama é a funcionalidade “Gerar relatório”, que estende a funcionalidade “Consultar histórico de atividades” do ator “Aluno”. Isso significa que, ao consultar o histórico de suas atividades, o estudante pode gerar um relatório de todas as atividades já submetidas. Essa estrutura do diagrama permitiu visualizar com clareza a interação entre os usuários e suas respectivas permissões, além de garantir o mapeamento de cada funcionalidade dentro do projeto.

Ainda, no diagrama de caso de uso, é possível observar que, além de “Gerar relatório”, outras funcionalidades estão associadas por meio da relação *extend*, como “Adicionar arquivos” e “Salvar rascunho”, indicando que as ações executadas a partir dessas funcionalidades são opcionais ou complementares. Outrossim, o ator “Coordenador” tem acesso a funcionalidades de gestão, como “Montar banca”, “Cadastrar novo coordenador” e “Manter aluno”, reforçando sua responsabilidade administrativa e diferenciando-o dos demais atores.

Também, no diagrama de caso de uso, é ilustrado um dos fluxos mais importantes do *software*, o da avaliação das atividades submetidas pelo aluno, refletindo o funcionamento do processo acadêmico de avaliação e integralização das horas de atividade de extensão. Esse processo ocorre em três etapas principais:

1. Na funcionalidade “Consultar demandas de análise”, o professor da banca avaliadora acessa a lista de atividades pendentes de avaliação;
2. Já na funcionalidade “Avaliar solicitação do aluno”, após a análise da documentação, o professor pode deferir ou indeferir uma atividade;
3. Por fim, na funcionalidade “Consolidar atividade”, a avaliação é consolidada e exibida para o aluno.

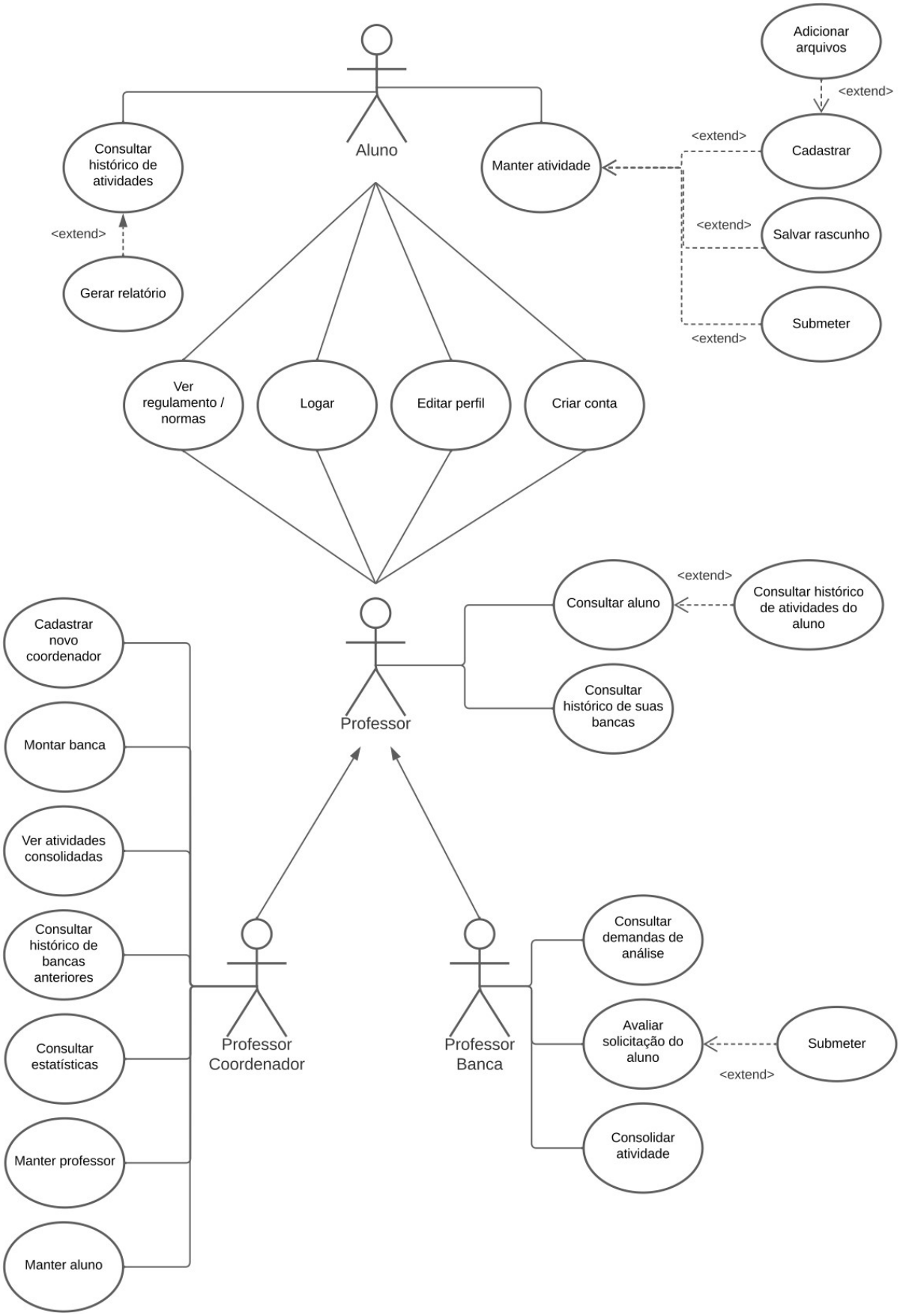


Figura 24 – Diagrama UML de caso de uso

Descrição do Caso de Uso: a Tabela 3 apresenta a descrição detalhada das funcionalidades do diagrama de caso de uso, onde cada funcionalidade possui um ID único, um nome descritivo, um ator responsável e a descrição dos requisitos funcionais que definem o comportamento esperado do *software*, por exemplo, a funcionalidade de ID 10, "Submeter", tem como requisito funcional a permissão para que o aluno submeta uma atividade para avaliação.

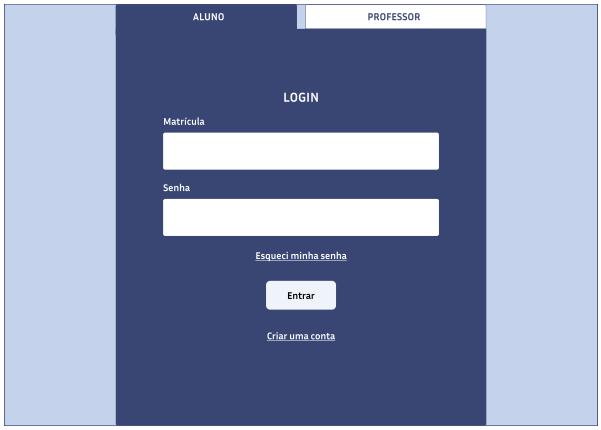
Tabela 3 – Descrição das funcionalidades do diagrama de caso de uso

ID	Nome	Ator	Requisitos Funcionais
01	Manter Atividade	Aluno	O sistema deve permitir a realização de alterações nas atividades.
02	Consultar histórico de atividades	Aluno	O sistema deve permitir a visualização das atividades classificadas pela situação (PENDENTES, SUBMETIDAS, INTEGRALIZADAS, INDEFERIDAS)
03	Gerar relatório	Aluno	O sistema deve gerar uma relatório, no formato PDF, contendo os dados do aluno e seu histórico de atividades.
04	Ver regulamento	Aluno, Professor	O sistema deve exibir o regulamento das atividades de extensão.
05	Adicionar arquivos	Aluno	O sistema deve permitir a anexação de arquivos à uma atividade.
06	Logar	Aluno, Professor	O sistema deve realizar a validação do usuário quando este se conectar na aplicação.
07	Editar perfil	Aluno, Professor	O sistema deve permitir a realização das alterações nos dados informados (p. ex. nome, e-mail). O sistema deve permitir a redefinição de senha.
08	Cadastrar	Aluno	O sistema deve permitir o cadastro, leitura e atualização de uma atividade.
09	Salvar rascunho	Aluno	O sistema deve permitir que o aluno possa armazenar as atividades enquanto estiver preenchendo, sem submetê-las. O sistema deve permitir a realização das alterações na atividade.
10	Submeter	Aluno	O sistema deve permitir a submissão das atividades cadastradas para avaliação da banca.
11	Consultar aluno	Professor	O sistema deve permitir a busca de perfil do aluno solicitado. O sistema deve informar os dados básicos do aluno solicitado (p. ex. nome, matrícula, e-mail) e seu histórico de atividades.
12	Consultar histórico de suas bancas	Professor	O sistema deve realizar a busca por bancas já participadas pelo professor que solicitou. O sistema deve mostrar o histórico de atividades avaliadas pelo professor que solicitou.
13	Consultar demandas de análise	Banca	O sistema deve mostrar as atividades pendentes de avaliação.
14	Avaliar solicitação do aluno	Banca	O sistema deve permitir a avaliação das atividades submetidas.
15	Submeter	Banca	O sistema deve permitir a submissão das avaliações. O sistema deve permitir o acesso da avaliação a outros professores da banca.
16	Montar banca	Coordenador	O sistema deve permitir que sejam escolhidos os professores para compor a banca de avaliação.
17	Ver atividades consolidadas	Coordenador	O sistema deve exibir atividades avaliadas pela banca.
18	Criar conta	Aluno, Professor	O sistema deve permitir a criação de uma conta para validação do usuário. O sistema deve armazenar os dados do usuário cadastrado.
19	Consultar histórico de atividades do aluno	Professor	O sistema deve exibir o histórico de atividades submetidas pelo aluno.
20	Consultar histórico de bancas anteriores	Coordenador	O sistema deve exibir o histórico de professores participantes de bancas anteriores. O sistema deve exibir o histórico de atividades avaliadas por bancas anteriores.
21	Consultar estatísticas	Coordenador	O sistema deve exibir estatísticas básicas do Curso de Engenharia da Computação. O sistema deve exibir estatísticas básicas dos alunos por semestre.
22	Manter professor	Coordenador	O sistema deve permitir ativar ou inativar um professor.
23	Manter aluno	Coordenador	O sistema deve permitir ativar ou inativar um aluno.
24	Cadastrar um novo coordenador	Coordenador	O sistema deve permitir o cadastro de um novo coordenador.
25	Consolidar atividades	Banca	O sistema deve exibir para o aluno a avaliação feita pela banca na atividade submetida.

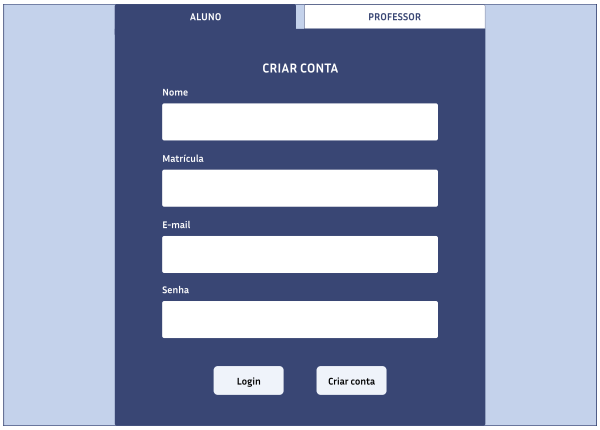
4.3 Prototipagem do sistema

A prototipagem do *software*, realizada com a ferramenta `Figma`, tem como objetivo modelar a interface de usuário antes da implementação definitiva. Esse processo permite a visualização da disposição dos elementos, a navegabilidade entre as telas e a organização das permissões de acesso conforme a hierarquia definida na definição do caso de uso. Além disso, as telas foram projetadas considerando a experiência do usuário, de modo a garantir uma interface intuitiva e acessível para os diferentes atores do *software*. Dessa forma, cada ator possui um conjunto específico de funcionalidades e, consequentemente, acessa diferentes interfaces de acordo com suas permissões de acesso.

A Figura 25 apresenta as telas de "*Login*" (Figura 25a), correspondente à funcionalidade de ID 06, e "Criar conta" (Figura 25b), correspondente à funcionalidade de ID 18, projetadas para permitir a autenticação dos usuários, e que possuem duas abas que separam os perfis entre Aluno e Professor, onde os atores "Professor", "Banca" e "Coordenador" acessam o sistema pela aba Professor. Essa separação foi projetada para simplificar o fluxo de entrada do *software* assegurando que cada ator tenha acesso apenas às telas correspondentes às suas permissões.



(a) Tela "Login"

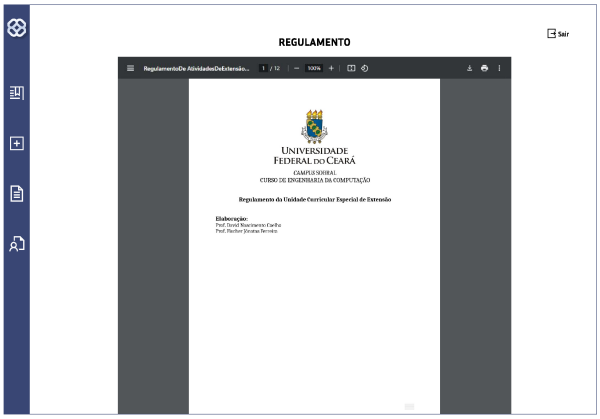


(b) Tela "Criar conta"

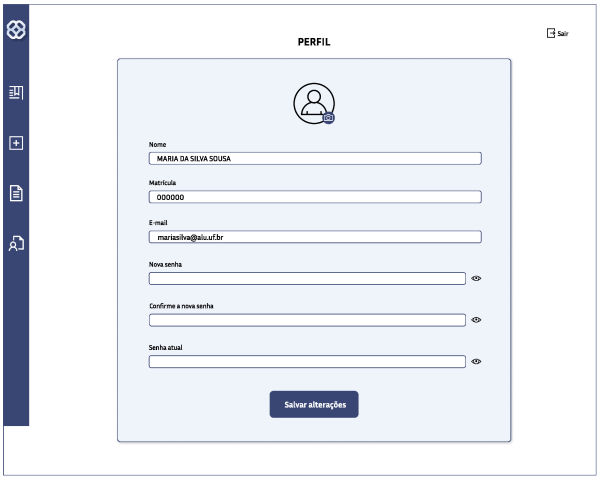
Figura 25 – Telas de acesso ao sistema

Dessa forma, para todos os perfis há uma permissão de acesso comum às páginas "Ver regulamento" e "Perfil", apresentadas na Figura 26. A página "Ver regulamento" (Figura 26a) tem como objetivo disponibilizar ao usuário o regulamento das atividades de extensão em um documento no formato *PDF*, de modo a permitir que alunos e professores consultem as orientações de submissão e avaliação das atividades, correspondendo à funcionalidade de ID

04, “Ver regulamento”. Já a página “Perfil” (Figura 26b) permite que o usuário possa editar informações cadastradas no sistema, como atualização de *e-mail* e senha, além de realizar o *upload* de uma foto de perfil. Assim, essa página de “Perfil” correponde à funcionalidade de ID 07 na Tabela de descrição do diagrama de caso de uso.



(a) Tela “Ver regulamento”



(b) Tela “Perfil”

Figura 26 – Telas comuns aos perfis

A interface do usuário Aluno foi projetada de modo a facilitar o gerenciamento de suas atividades de extensão. Para tal, a visualização das páginas possui um menu lateral que dá acesso rápido às principais funcionalidades: “Minhas atividades”, correspondente à funcionalidade de ID 02, e “Cadastrar atividade”, correspondente à funcionalidade de ID 08, além das páginas de acesso comum aos perfis. Esse *design* garante que o aluno gerencie suas atividades de forma prática, reduzindo o tempo necessário para localizar informações importantes dentro do sistema.

Assim, a página “Minhas atividades” (Figura 27) foi projetada para oferecer ao aluno uma visão clara do andamento de suas submissões. Sendo assim, as atividades são classificadas em quatro categorias, permitindo acompanhar detalhadamente a situação de cada atividade submetida:

- **Pendentes:** atividades que ainda não foram submetidas oficialmente, ou seja, permanecem em rascunho. Essa situação corresponde à funcionalidade de ID 09;
- **Submetidas:** atividades enviadas para análise da banca avaliadora, correspondente à funcionalidade de ID 10;
- **Integralizadas:** atividades aprovadas, cujas horas de extensão foram integralizadas;
- **Indeferidas:** atividades que não foram aprovadas pela banca avaliadora, que recebem um

feedback indicando os motivos de indeferimento.

Figura 27 – Tela “Minhas atividades”

Além da visualização categorizada das atividades, ao selecionar o botão “Visualizar”, na página “Minhas atividades”, o *software* direciona o usuário para a página “Visualizar atividade” (Figura 28), onde é possível consultar detalhes da submissão, como informações gerais da atividade cadastrada, *feedback* da banca avaliadora, no caso das atividades integralizadas e indeferidas, e a opção de realizar o *download* do arquivo de comprovação de participação na atividade, enviado durante a submissão.

Ainda na página “Minhas atividades”, o sistema permite ao aluno gerar um relatório, correspondente à funcionalidade de ID 03, com os detalhes de todas as suas atividades cadastradas. Como mostrado na Figura 29, essa funcionalidade pode ser acessada através do botão “Gerar relatório”, localizado no canto superior direito da página. Esse relatório contém detalhes das atividades do aluno, incluindo a lista de todas as suas atividades organizadas por situação (pendentes, submetidas, integralizadas e indeferidas) e a carga horária correspondente a cada atividade. Essa funcionalidade facilita ao aluno acompanhar o histórico de sua participação em

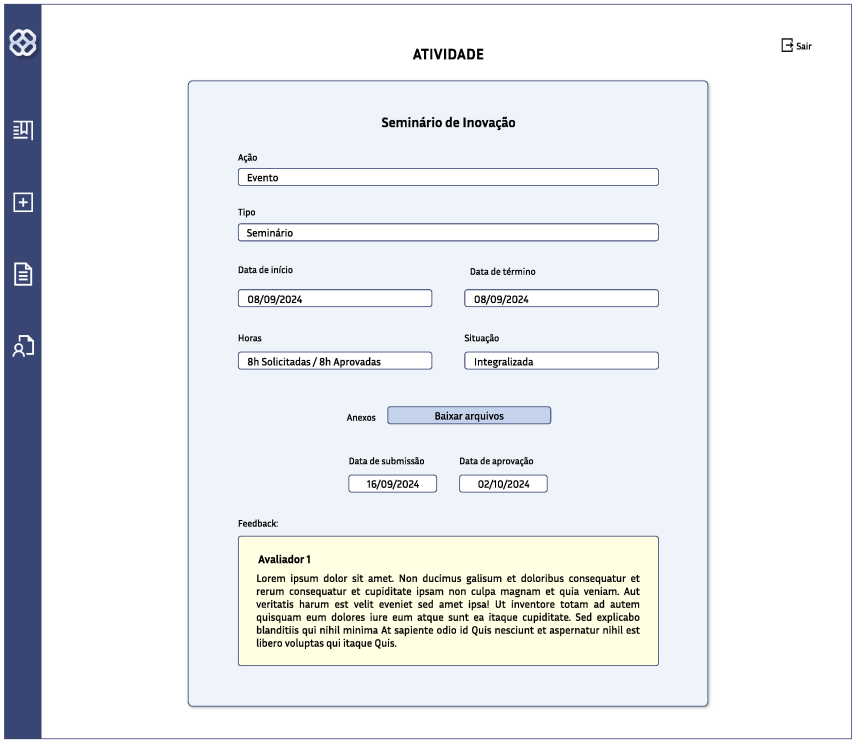


Figura 28 – Tela de visualização de atividade

atividades de extensão, além de gerar um documento de registro oficial de suas atividades para fins acadêmicos e administrativos.

Já a página “Cadastrar atividade” (Figura 30) foi projetada para permitir o cadastro de novas atividades de extensão por meio de um formulário onde o aluno anexa a documentação comprobatória de sua participação e pode ainda salvar como rascunho antes da submissão definitiva. Além disso, campos de preenchimento como o título da atividade, quantidade de horas de participação e a data de início e término da atividade fazem parte do formulário de cadastro.

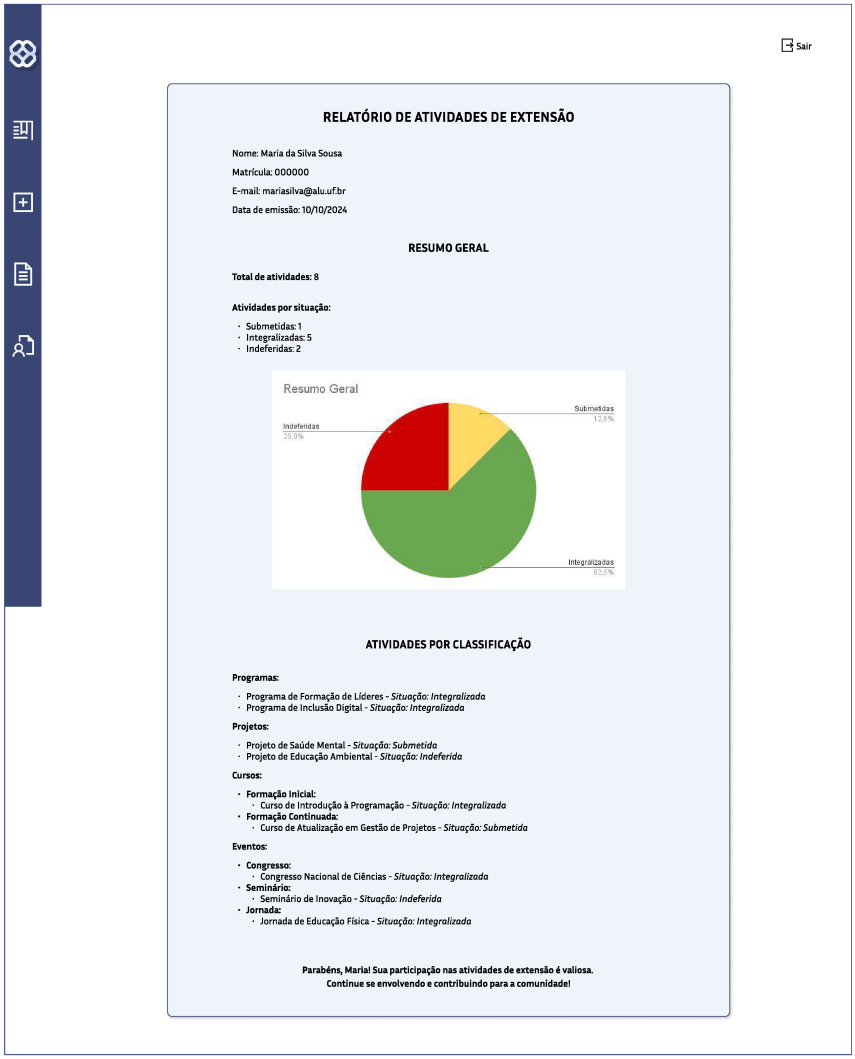


Figura 29 – Relatório das atividades do Aluno

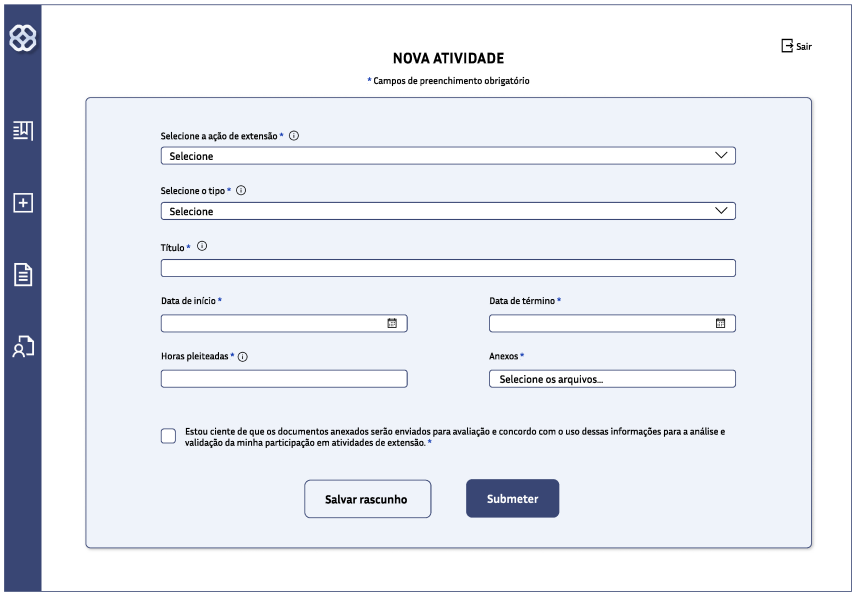


Figura 30 – Tela “Cadastrar atividade”

Nas páginas do usuário Professor há um menu lateral de acesso rápido à página “Área do professor”, como mostrado na Figura 31. Essa página foi projetada para centralizar as funcionalidades “Consultar aluno”, ID 11, e “Consultar histórico de suas bancas”, ID 12.

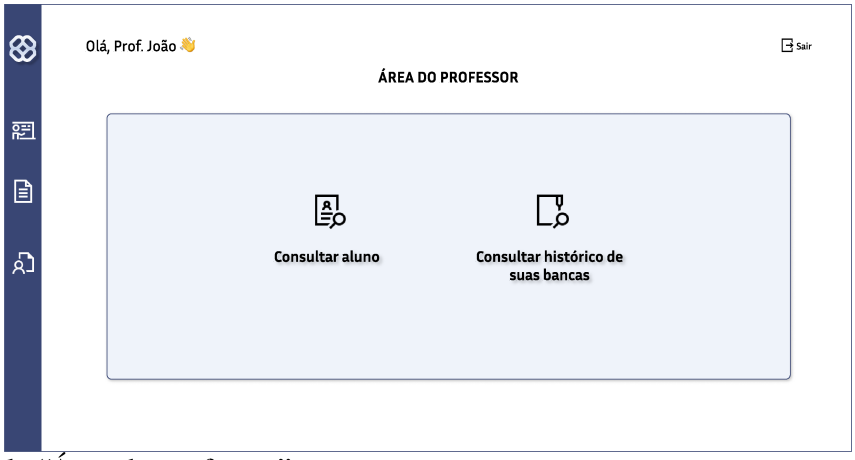


Figura 31 – Tela “Área do professor”

Assim, ao clicar no botão “Consultar aluno”, o professor é direcionado a uma página que lhe permite consultar um aluno a partir do nome e/ou matrícula, como mostrado na Figura 32. Após localizar o aluno desejado, o professor pode visualizar as informações do aluno, bem como suas atividades, categorizadas em integralizadas e indeferidas. Ao clicar no botão “Visualizar” localizado ao lado da atividade, o professor tem acesso detalhado à atividade em uma página semelhante à visualização do aluno.

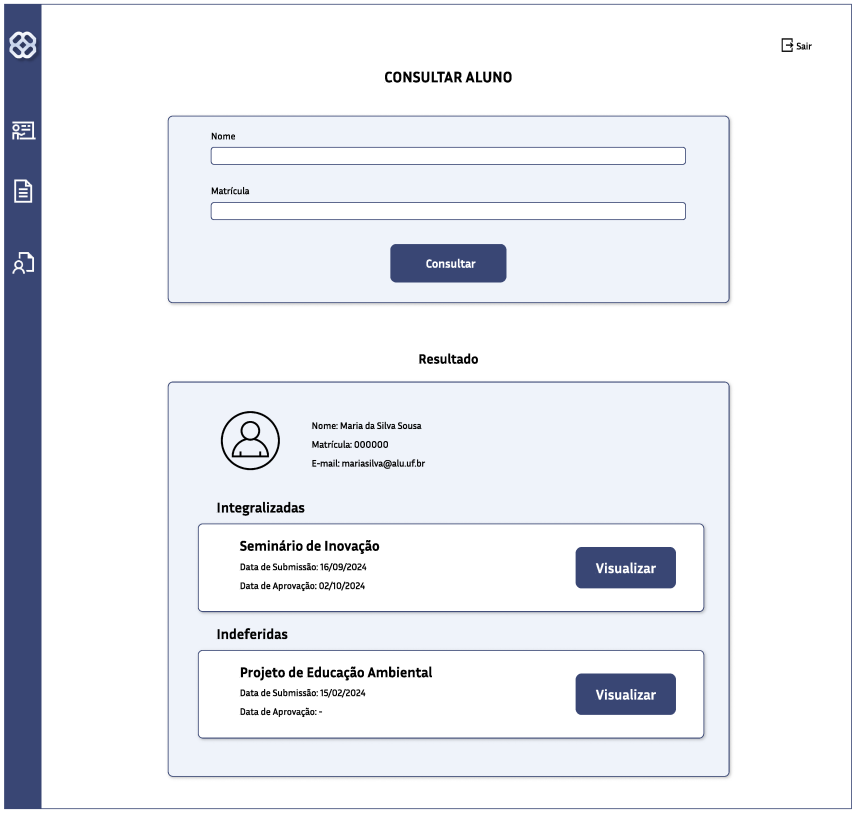


Figura 32 – Tela “Consultar aluno”

Ao clicar no botão “Consultar histórico de suas bancas” o professor pode pesquisar as bancas das quais participou, utilizando filtros de busca como ano, semestre e/ou título da atividade, conforme mostrado na Figura 33. Na parte inferior da página, os resultados de busca são exibidos, permitindo a visualização dos demais participantes da banca e da lista de atividades avaliadas pela respectiva banca, além do botão de visualização de atividade.

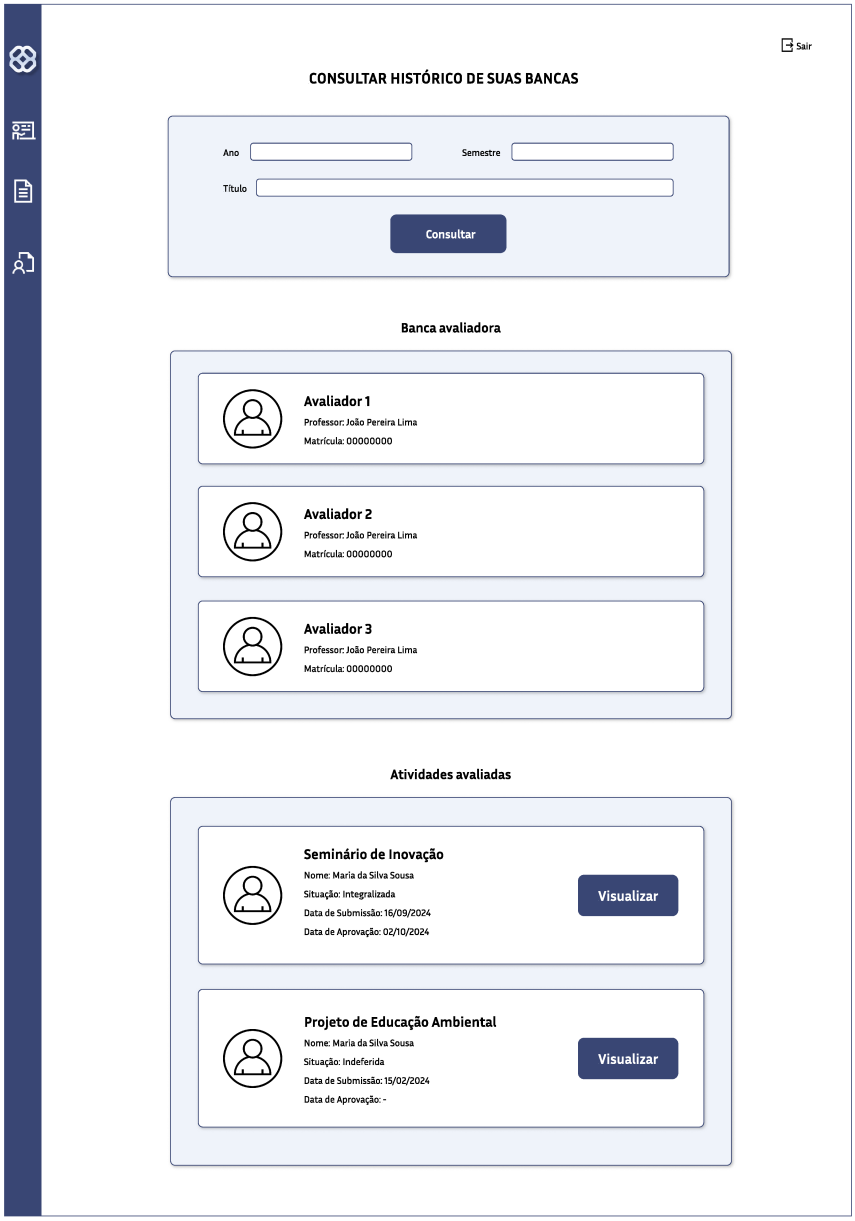


Figura 33 – Tela “Consultar histórico de suas bancas”

Já nas páginas do usuário “Banca” a interface apresenta um menu de acesso à página “Área do professor” com as mesmas visualizações do usuário “Professor”, além do acesso à página “Área do avaliador”, conforme mostrado na Figura 34. Essa página “Área do Avaliador” foi projetada para centralizar o processo de avaliação das solicitações submetidas por um aluno. O professor avaliador pode buscar por uma palavra-chave contida no título da atividade ou visualizar todas as atividades, correspondendo à funcionalidade de ID 13, organizadas em duas categorias: Atividades Pendentes, àquelas que aguardam por avaliação; e Atividades Avaliadas, atividades já avaliadas. Dentro da visualização de Atividades Pendentes, o avaliador tem acesso a detalhes da atividade, bem como à avaliação de outros membros da banca. Após a análise da atividade, o avaliador pode adicionar seu *feedback* ao aluno, fornecendo justificativas ou

sugestões, caso necessário, e concluir sua avaliação clicando nos botões “Deferir” ou “Indeferir”, correspondendo à funcionalidade de ID 15.

Figura 34 – Tela “Área do avaliador”

Por fim, o usuário Coordenador, semelhante ao usuário Professor Banca, tem acesso à página “Área do Professor” e possui os mesmos acessos dos demais docentes. No entanto, além desses acessos, o Coordenador, por possuir permissões administrativas exclusivas, este tem acesso à página “Área do Coordenador”, mostrada na Figura 35, projetada para concentrar as ferramentas de gerenciamento administrativo, permitindo ao Coordenador consultar histórico de todas as bancas (ID 20), consultar usuário (professor ou aluno) (ID 22 e 23), consultar estatísticas (ID 21), montar nova banca (ID 16) e cadastrar um novo coordenador (ID 24).

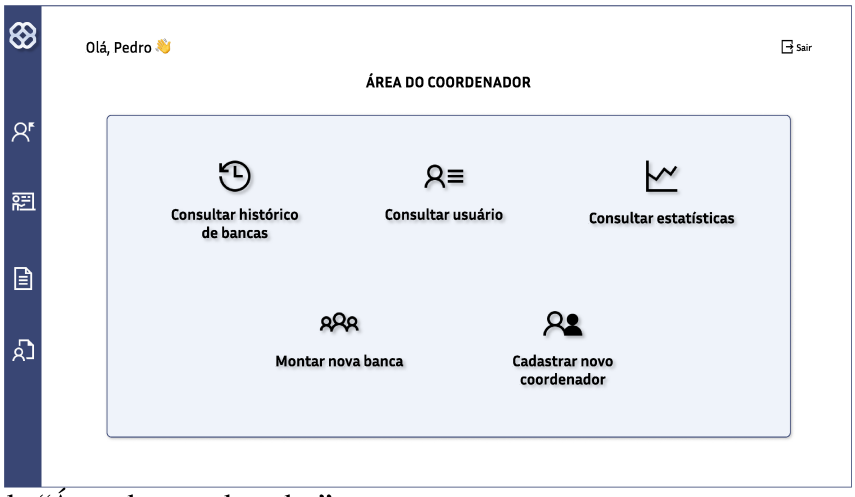


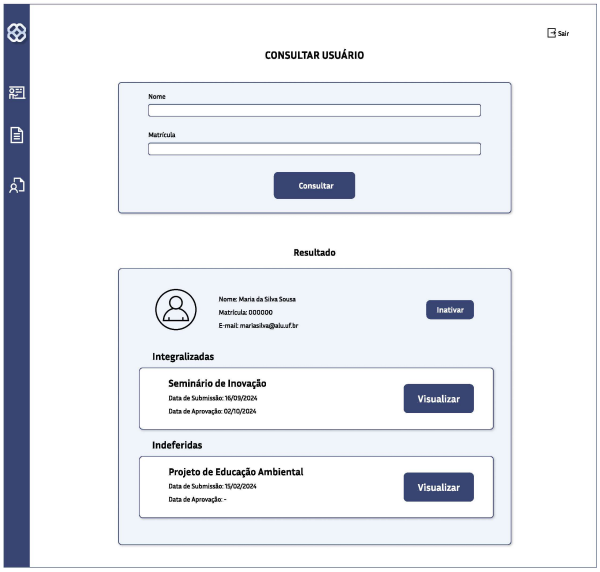
Figura 35 – Tela “Área do coordenador”

A página “Consultar histórico de bancas” é semelhante à página “Consultar histórico de suas bancas” disponível no perfil do Professor. No entanto, é diferenciada por permitir o acesso ao histórico de todas as bancas registradas no sistema, não se limitando apenas à banca da qual o usuário fez parte. Essa funcionalidade permite ao Coordenador acompanhar detalhadamente as avaliações realizadas.

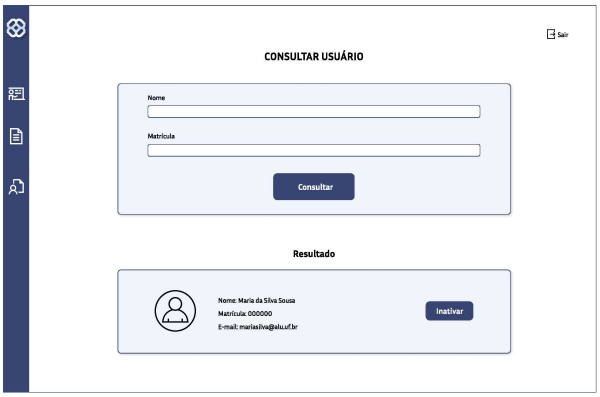
Já a página “Consultar usuário”, mostrada na Figura 36, permite ao Coordenador buscar por informações de usuários, sejam eles alunos (Figura 36a) ou professores (Figura 36b), através do uso de filtros como nome ou matrícula. Ainda na página “Consultar usuário”, no canto superior direito da seção de resultados da página, há o botão “Ativar/Inativar”, correspondendo às funcionalidades de ID 22 e 23, que permite ao Coordenador habilitar ou desabilitar o acesso de um usuário ao sistema.

Ainda no perfil do Coordenador, a página “Consultar estatísticas”, conforme mostrada na Figura 37, oferece um *dashboard* interativo, criado com a ferramenta de `Genially`[1], que apresenta, em uma visualização, os dados estatísticos do sistema, como a quantidade total de

[1] https://genially.com/pt-br/



(a) Tela “Consultar usuário” para o resultado de aluno



(b) Tela “Consultar usuário” para o resultado de professor

Figura 36 – Telas de consulta de usuário

atividades submetidas e gráficos de comparação entre os semestres, auxiliando o Coordenador no acompanhamento e análise das atividades de extensão no curso.

Figura 37 – Tela “Consultar estatísticas”

Além disso, a página “Montar nova banca” exibe um formulário de cadastro de novas bancas avaliadoras incluindo os campos de vigência da banca (ano e semestre letivo, e data de início e término de validade), além dos campos de busca dos avaliadores, como mostrado na Figura 38.

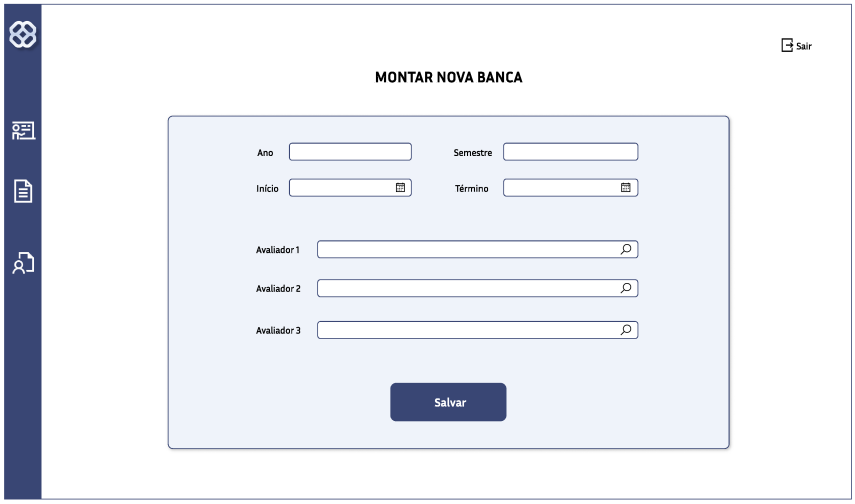


Figura 38 – Tela “Montar nova banca”

Por fim, a página “Cadastrar novo coordenador” permite ao atual Coordenador ceder o acesso de Coordenador a outro usuário. Essa página contém um formulário que inclui os campos de busca do usuário, consentimento de acesso de permissão e senha do usuário para efetivar o cadastro, mostrado na Figura 39.

Figura 39 – Tela “Cadastrar novo coordenador”

4.4 Implementação das funcionalidades

A partir do levantamento das funcionalidades do *software*, foram escolhidas para serem implementadas utilizando os *frameworks* estudados as funcionalidades "Cadastrar" (ID 08), que permite inserir as informações da atividade no formulário, e "Submeter" (ID 10), que permite que as atividades cadastradas sejam submetidas à avaliação. Além disso, foi utilizado o banco de dados `SQLite` em todos os projetos, pois não necessita de configuração manual, já que os próprios *frameworks* o utilizam como padrão quando não há especificação de outro banco de dados.

4.4.1 Implementação das funcionalidades utilizando Ruby on Rails

Na implementação das funcionalidades "Cadastrar" e "Submeter" utilizando o *framework* `Ruby on Rails` foram criadas três rotas, conforme apresenta o Listing 4.4.1. A rota *root* exibe a *view index* que apresenta em tela as atividades cadastradas, como a tela "Minhas atividades", apresentada na prototipagem do *software*; já as demais rotas são responsáveis, respectivamente, por exibir o formulário, como na tela de cadastro de atividade da prototipagem, e enviar os dados preenchidos no formulário.

```
1    Rails.application.routes.draw do
2      resources :atividades, only: [:index, :show]
3      root 'atividades#index'
4      get "/cadastrar", to: "atividades#new"
5      post "/cadastrar", to: "atividades#create"
6    end
```

Listing 4.4.1 – Rotas da aplicação - Rails

Em seguida, no *controller*, conforme apresenta o Listing 4.4.2, foram definidas cinco ações, sendo elas: a ação *index* filtra as atividades de acordo com a situação em que estão, a ação *show* busca no banco a atividade e renderiza a página, a ação *new* exibe o formulário de cadastro de uma nova atividade, a ação *create* cria a atividade de acordo com as informações preenchidas no formulário e, por fim, a ação atividade_params é passada como parâmetro no momento de salvar a atividade, permitindo a correspondência entre os dados e o banco de dados.

As atividades estão dispostas na página inicial separadamente em:

- Submetidas: aguardando aprovação pela banca avaliadora;

```ruby
class AtividadesController < ApplicationController
def index
  @submetidas = Atividade.where(situacao: 'Em análise')
  @integralizadas = Atividade.where(situacao: 'Aprovada')
  @indeferidas = Atividade.where(situacao: 'Indeferida')
end
def show
  @atividade = Atividade.find(params[:id])
  render 'atividades/exibir-atividade'
end
def new
  @atividade = Atividade.new
  render 'atividades/cadastrar-atividade'
end
def create
  @atividade = Atividade.new(atividade_params)
  if params[:atividade][:acao_extensao] == 'Curso'
    @atividade.tipo = params[:atividade][:tipo_curso]
  elsif params[:atividade][:acao_extensao] == 'Evento'
    @atividade.tipo = params[:atividade][:tipo_evento]
  else
    @atividade.tipo = nil
  end
  if @atividade.save
    redirect_to root_path, notice: 'Atividade cadastrada com
    ↪  sucesso!'
  else
    render 'atividades/cadastrar-atividade'
  end
end
def atividade_params
  params.require(:atividade).permit(:acao_extensao, :tipo, :horas,
  ↪  :anexo)
end
end
```

Listing 4.4.2 – Controller da aplicação - Rails

- Integralizadas: aprovadas pela banca avaliadora;
- Indeferidas: reprovadas pela banca avaliadora.

O arquivo *model* do projeto define a obrigatoriedade dos campos, além da definição da data de submissão da atividade como a data correspondente ao dia em que é enviada, conforme apresenta o Listing 4.4.3. Para simular as atividades “Integralizadas” e “Indeferidas” foi utilizado o comando *rails console* para alterar os campos *situacao* e *data_aprovacao* no banco de dados.

Além disso, na página *index*, é possível ainda visualizar os detalhes de cada tarefa, ao

```ruby
class Atividade < ApplicationRecord
    has_one_attached :anexo
    validates :acao_extensao, presence: true
    validates :horas, presence: true, numericality: { only_integer:
    ↪  true, greater_than: 0 }
    before_create :set_data_submissao
    private
    def set_data_submissao
        self.data_submissao ||= Date.today
    end
end
```

Listing 4.4.3 – Model da aplicação - Rails

clicar sobre a atividade desejada, sendo então redirecionado à página de visualização de atividade referente às informações cadastradas durante a submissão da atividade, como na tela "Visualizar atividade" apresentada na prototipagem do *software*. No *link* "Baixar anexo" é possível realizar o *download* da documentação de comprovação de participação na atividade de extensão enviada pelo discente e ao clicar no *link* "Voltar" o usuário é redirecionado à página inicial.

Ainda na página inicial, ao clicar no *link* "Cadastrar atividade" o usuário é redirecionado à página de formulário para cadastrar uma nova atividade. Nesse formulário, o primeiro campo recebe a Ação de Extensão de acordo com o documento de regulamento das atividades de extensão utilizado na etapa de levantamento de requisitos, sendo elas: Programa, Projeto, Curso, Evento. Excepcionalmente as ações Curso e Eventos possuem subcategorias, sendo as subcategorias da ação Curso: Formação inicial e Formação continuada, e as subcategorias da ação Evento sendo: Congresso, Seminário, Encontro e Simpósio. Dessa forma, quando selecionada alguma dessas ações, é exibido um novo campo chamado tipo.

Por fim, para armazenar os documentos comprobatórios de participação do aluno enviado por anexo, foi utilizado o *Active Storage* que facilita o *upload* de arquivos. O *Active Storage* armazena os arquivos localmente no diretório *storage* garantindo um processo de *upload* de arquivo mais seguro.

4.4.2 Implementação das funcionalidades utilizando Laravel

Já para a implementação das funcionalidades utilizando o *framework* `Laravel`, foram criadas cinco rotas conforme apresenta o Listing 4.4.4. A rota principal "/" exibe a página inicial da aplicação. Em seguida, a rota /cadastrar utiliza dois métodos, sendo eles *create*, usado

para exibir o formulário de cadastro, e *store*, que recebe os dados do formulário e salva no banco de dados. Já a rota atividades/id exibe a atividade selecionada, identificada pelo parâmetro "id". Finalmente, a rota download/id é dedicada ao *download* do anexo enviado no cadastro da atividade e utiliza o parâmetro "id" para buscar o arquivo no banco de dados.

```php
<?php
use Illuminate\Support\Facades\Route;
use App\Http\Controllers\AtividadeController;

Route::get('/', [AtividadeController::class,
    'index'])->name('atividades.index');
Route::get('/cadastrar', [AtividadeController::class,
    'create'])->name('atividades.create');
Route::post('/cadastrar', [AtividadeController::class,
    'store'])->name('atividades.store');
Route::get('/atividades/{id}', [AtividadeController::class,
    'show'])->name('atividades.show');
Route::get('/download/{id}', [AtividadeController::class,
    'download'])->name('atividade.download');
```

Listing 4.4.4 – Rotas da aplicação - Laravel

No *controller*, apresentando no Listing 4.4.5, foram definidas cinco ações, sendo elas: a ação *index* que realiza a separação das atividades de acordo com a exibição nas *views*, filtrando entre submetidas, integralizadas e indeferidas; a ação *create* que exibe a *view* de cadastro que contém o formulário; a ação *store* que recebe a requisição de cadastro da nova atividade, inicialmente verificando se o arquivo foi anexado, e em seguida cadastrando a nova atividade no banco; a ação *show* que exibe uma atividade a partir do "id" enviado como argumento; a ação *download* que é responsável pelo *download* do arquivo anexado no cadastro da atividade ao clicar no botão "Baixar anexo" na página de exibição da atividade.

```php
<?php
class AtividadeController extends Controller {
    public function index() {
        $submetidas = Atividade::where('situacao', 'Em
        ↪ análise')->get();
        $integralizadas = Atividade::where('situacao',
        ↪ 'Integralizada')->get();
        $indeferidas = Atividade::where('situacao',
        ↪ 'Indeferida')->get();
        return view('index', compact('submetidas', 'integralizadas',
        ↪ 'indeferidas')); }
    public function create() { return view('cadastrar-atividade'); }
    public function store(Request $request) {
        if ($request->hasFile('anexo')) {
            $anexoPath = $request->file('anexo')->store('anexos',
            ↪ 'public');
        } else {
            return redirect()->back()->withErrors(['anexo' => 'O
            ↪ arquivo deve ser enviado.']); }
        $atividade = Atividade::create([
            'acao_extensao' => $request->input('acao_extensao'),
            'tipo' => $request->input('tipo-curso') ??
            ↪ $request->input('tipo-evento'),
            'horas' => $request->input('horas'),
            'anexo' => $anexoPath ]);
        return redirect('/')->with('success', 'Atividade cadastrada
        ↪ com sucesso!'); }
    public function show($id) {
        $atividade = Atividade::findOrFail($id);
        return view('exibir-atividade', compact('atividade')); }
    public function download($id) {
        $atividade = Atividade::findOrFail($id);
        if ($atividade->anexo) {
            $filePath = storage_path('app/public/' .
            ↪ $atividade->anexo);
            if (file_exists($filePath)) {
                return response()->download($filePath);
            } else {
                return redirect()->back()->with('error', 'Anexo não
                ↪ encontrado.'); }}
        return redirect()->back()->with('error', 'Anexo não
        ↪ disponível.'); }}
```

Listing 4.4.5 – Controller da aplicação - Laravel

Em seguida, no arquivo *model*, mostrado no Listing 4.4.6, são definidos os campos presentes no banco de dados, e, por padrão, determina o preenchimento do campo “situacao”

como “Em análise”, além da função *boot* que adiciona automaticamente a data de submissão e a data de aprovação de acordo, respectivamente, com a data em que a atividade é cadastrada e a data em que a atividade foi avaliada pela banca.

```php
<?php
namespace App\Models;
use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;
class Atividade extends Model {
    use HasFactory;
    protected $fillable = [
        'acao_extensao',
        'tipo',
        'horas',
        'anexo',
        'data_submissao',
        'situacao',
        'data_avaliacao'
    ];
    protected $attributes = [
        'situacao' => 'Em análise',
    ];
    public static function boot() {
        parent::boot();
        static::creating(function ($model) {
            $model->data_submissao = now();
        });
        static::updating(function ($model) {
            if ($model->isDirty('situacao') && $model->situacao ===
            ↪ 'Integralizada') {
                $model->data_avaliacao = now();
            }
        });
    }
}
```

Listing 4.4.6 – Model da aplicação - Laravel

Semelhante à implementação utilizando `Rails` (Seção 4.4.1), as atividades na página inicial estão dispostas separadamente entre submetidas, integralizadas e indeferidas. Para isso, utilizou-se o comando *php artisan tinker* para modificar manualmente o campo “situacao” das atividades a fim de simular atividades “Integralizadas” e “Indeferidas”.

4.4.3 Implementação das funcionalidades utilizando Django

Na implementação utilizando Django, há dois tipos de arquivos de rotas: *urls.py* na pasta padrão de mesmo nome do projeto e *urls.py* do app criado, chamado "atividades". No Listing 4.4.7 é apresentado o arquivo de rotas da pasta padrão que define a rota da página *admin* do Django e a rota "atividades/" da página inicial do projeto.

```
1    from django.contrib import admin
2    from django.urls import path, include
3    urlpatterns = [
4        path('admin/', admin.site.urls),
5        path('atividades/', include('atividades.urls')),
6    ]
```

Listing 4.4.7 – urls.py da pasta padrão da aplicação - Django

O arquivo *admin.py*, responsável por definir a configuração da interface administrativa do Django, é apresentado no Listing *listing:software-admin-django*, onde o modelo Atividade é importado e a tabela é dividida em cinco colunas: acao_extensao, get_tipo, horas, data_submissao.

```
1    from django.contrib import admin
2    from .models import Atividade
3    class AtividadeAdmin(admin.ModelAdmin):
4        list_display = ('acao_extensao', 'get_tipo', 'horas',
         ↪  'data_submissao')
5        list_filter = ('acao_extensao', 'horas', 'data_submissao')
6        def get_tipo(self, obj):
7            if obj.acao_extensao == "Curso":
8                return obj.tipo_curso or "-"
9            elif obj.acao_extensao == "Evento":
10               return obj.tipo_evento or "-"
11           return "-"
12       get_tipo.short_description = "Tipo"
13   admin.site.register(Atividade, AtividadeAdmin)
```

Listing 4.4.8 – Admin da aplicação - Django

Já o Listing 4.4.9 apresenta o arquivo de rotas do app "atividades". A rota principal "/", exibe a view que lista todas as atividades cadastradas. Enquanto a rota "cadastrar/" exibe o formulário de cadastro de uma nova atividade. Já a rota "<int:id>/" exibe a atividade

correspondente ao “id” passado por parâmetro de rota.

```
from django.urls import path
from . import views
app_name = 'atividades'
urlpatterns = [
    path('', views.lista_atividades, name='lista_atividades'),
    path('cadastrar/', views.cadastrar_atividade,
    ↪  name='cadastrar_atividade'),
    path('<int:id>/', views.exibir_atividade,
    ↪  name='exibir_atividade'),
]
```

Listing 4.4.9 – Rotas da aplicação - Django

No arquivo *views.py* foram definidas três *views*, como apresentado no Listing 4.4.10. A *view* “lista_atividades” é responsável por filtrar as atividades por situação e renderizar o *template* “index.html”. Já a *view* “cadastrar_atividade” renderiza o formulário e valida a resposta do cadastro, salvando as informações no banco de dados. E, por fim, a *view* “exibir_atividade” recebe como argumento o “id” da atividade e renderiza o *template* “exibir-atividade.html” com os dados da atividade requisitada.

```
    from django.shortcuts import render, get_object_or_404, redirect
    from .models import Atividade
    from .forms import AtividadeForm
    def lista_atividades(request):
        submetidas = Atividade.objects.filter(situacao='Em análise')
        integralizadas =
        ↪  Atividade.objects.filter(situacao='Integralizada')
        indeferidas = Atividade.objects.filter(situacao='Indeferida')
        context = {
            'submetidas': submetidas,
            'integralizadas': integralizadas,
            'indeferidas': indeferidas
        }
        return render(request, 'atividades/index.html', context)
    def cadastrar_atividade(request):
        if request.method == "POST":
            return redirect("atividades:lista_atividades")
        return render(request, "atividades/form.html")
    def exibir_atividade(request, id):
        atividade = get_object_or_404(Atividade, pk=id)
        return render(request, 'atividades/exibir-atividade.html',
        ↪ {'atividade': atividade})
```

Listing 4.4.10 – Views da aplicação - Django

Finalmente, no arquivo *models.py* é definido o modelo de dados da aplicação, especificando os campos da tabela que a ser armazenada no banco de dados, como mostrado no Listing 4.4.11. O modelo inclui campos, como *acao_extensao*, *anexo* e *situacao*, dentre outros. Além disso, o modelo garante que apenas valores válidos possam ser atribuídos aos campos da tabela.

```python
from django.db import models
class Atividade(models.Model):
    ACAO_EXTENSAO_CHOICES = [
        ('Programa', 'Programa'),
        ('Projeto', 'Projeto'),
        ('Curso', 'Curso'),
        ('Evento', 'Evento'),
    ]
    TIPO_CURSO_CHOICES = [
        ('Formação inicial', 'Formação inicial'),
        ('Formação continuada', 'Formação continuada'),
    ]
    TIPO_EVENTO_CHOICES = [
        ('Congresso', 'Congresso'),
        ('Seminário', 'Seminário'),
        ('Encontro', 'Encontro'),
        ('Simpósio', 'Simpósio'),
    ]
    SITUACAO_CHOICES = [
        ('Em análise', 'Em análise'),
        ('Integralizada', 'Integralizada'),
        ('Indeferida', 'Indeferida'),
    ]
    acao_extensao = models.CharField(max_length=20,
    ↪  choices=ACAO_EXTENSAO_CHOICES)
    tipo_curso = models.CharField(max_length=30,
    ↪  choices=TIPO_CURSO_CHOICES, blank=True, null=True)
    tipo_evento = models.CharField(max_length=30,
    ↪  choices=TIPO_EVENTO_CHOICES, blank=True, null=True)
    horas = models.PositiveIntegerField()
    anexo = models.FileField(upload_to='anexos/', blank=True,
    ↪  null=True)
    data_submissao = models.DateField(auto_now_add=True)
    situacao = models.CharField(max_length=50, default="Em análise")

    def __str__(self):
        return f"{self.acao_extensao} - {self.horas}h"
```

Listing 4.4.11 – Model da aplicação - Django

4.5 Pontos de Comparação

A partir da implementação das funcionalidades, foi possível realizar as comparações entre os *frameworks* utilizando os seguintes pontos:

- **Curva de aprendizado:** do ponto de vista da linguagem de programação em que foram desenvolvidos os *frameworks*, visto que a dificuldade da linguagem impacta na curva de

aprendizado, o Django apresenta uma melhor curva de aprendizado dada a facilidade e ampla utilização apresentada pela linguagem Python. Seguido pelo Laravel dada a popularidade da linguagem PHP. Por último Rails, desenvolvido utilizando Ruby, menos popular que os demais;

- **Tamanho dos arquivos:** o tamanho dos arquivos de um *framework* pode influenciar na manutenção a longo prazo. Ao comparar o tamanho final do projeto em cada um dos *frameworks* Django apresenta o menor tamanho com 446 KB, seguido do projeto em Rails com 14,9 MB e, por fim, o projeto em Laravel com 727 MB;
- **Verbosidade do framework:** a verbosidade reflete a simplicidade relacionada ao *framework*. Dessa forma, o Rails e o Laravel apresentam uma abordagem mais concisa, enquanto Django prioriza uma clareza e legibilidade, exigindo mais detalhes na escrita de código. Assim, a escolha da tecnologia quanto à verbosidade deve considerar um equilíbrio entre clareza e manutenibilidade a longo prazo.
- **Documentação de apoio:** as documentações dos *frameworks* Django e Rails possuem um tutorial de auxílio para implementação de um pequeno projeto inicial a fim de que o desenvolvedor conheça o básico do *framework*, sendo a documentação do Django considerada a mais clara e didática. Já a documentação do Laravel não segue uma ordem lógica de implementação de um projeto para iniciantes da linguagem, dificultando a estruturação de um projeto inicial; porém, apresenta exemplos genéricos para tópicos presentes na documentação. Ainda, o idioma padrão das três documentações comparadas é o inglês, porém a documentação do Django possui tradução para o português brasileiro, facilitando aos desenvolvedores não fluentes na língua inglesa;
- **Comunidade para tirar dúvidas:** os *frameworks* utilizados possuem comunidade dedicada ao estudo da linguagem e disponível para tirar dúvidas, tendo Rails e Django o diferencial de ter uma comunidade brasileira dedicada.
- **Camadas de segurança:** os principais ataques em uma aplicação *Web* são SQL Injection, XSS (*Cross-Site Scripting*) e CSRF (*Cross-Site Request Forgery*). O SQL Injection é um ataque de inserção de código malicioso em consultas SQL que pode destruir o banco de dados. O XSS é um ataque onde o invasor injeta código malicioso na aplicação que será executado na máquina do usuário. Já o CSRF é um ataque de envio malicioso de solicitações não autorizadas de um usuário para um *site*, se utilizando da confiança no navegador de um usuário autenticado. Assim, utilizando como critérios de segurança os

principais ataques em aplicação *Web*, o desempenho dos *frameworks* é apresentado na Tabela 4.

Tabela 4 – Comparação da camada de segurança entre os frameworks

Segurança	Rails	Laravel	Django
SQL Injection	ORM Active Record fornece métodos de consulta que evita SQL Injection	Eloquent ORM que fornece métodos de consulta que evita SQL Injection	Evita SQL Injection separando o código de consulta dos parâmetros de consulta
XSS	Tem prevenção XSS automática com helpers	Tem prevenção XSS automática com Blade	Tem prevenção XSS automática com templates
CSRF	Proteção através do módulo RequestForgeryProtection	Proteção através da geração automática de tokens por sessão	Proteção integrada

Após a análise das camadas de segurança dos três *frameworks*, pode-se afirmar que todos oferecem boa segurança contra os principais ataques em uma aplicação *Web*, porém o `Django` se destaca por possuir proteção automática contra esses ataques.

- **Utilização no mercado:** segundo o Survey StackOverflow 2024[2], em uma pesquisa cujo objetivo é elencar as ferramentas e tecnologias mais usadas atualmente pelos desenvolvedores, para a pergunta "Em quais estruturas e tecnologias da *Web* você realizou um extenso trabalho de desenvolvimento no ano passado?", de um total de 38132 respostas, 11,4% dos desenvolvedores responderam o *framework* `Django`, 8,6% responderam `Laravel` e 5,2% responderam `Ruby on Rails`. Ainda na mesma pesquisa foi questionado "Em quais (tecnologias *Web*) deseja trabalhar no próximo ano?", e para 5010 respostas, 14,3% para `Django`, 4,8% para `Laravel` e 1,9% para `Ruby on Rails`. Além disso, segundo o Statista[3] em uma pesquisa que busca elencar os *frameworks Web* mais usados entre os desenvolvedores em todo o mundo em 2024, de um total de 48.503 entrevistados, 12% responderam `Django`, 7,9% responderam `Laravel` e 4,7% responderam `Ruby on Rails`. A partir dessas pesquisas, pode-se observar um maior uso do *frameworks* `Django` no mercado, seguido pelo `Laravel` e `Ruby on Rails`. Ademais, `Django` aponta uma maior perspectiva de crescimento para os próximos anos quando comparado aos outros dois *frameworks*.

4.6 Lições aprendidas

Ao longo deste estudo, foram realizadas comparações entre *frameworks* utilizados para desenvolvimento de aplicações *Web* que integrem *Front-end* e *Back-end*, além do planeja-

[2] https://survey.stackoverflow.co/2024/
[3] https://www.statista.com/

mento de um *software* de controle de extensão para uma IES. Baseado na análise comparativa, foi possível observar vantagens de cada *framework* e realizar prototipagem das telas do *software* de extensão. Com isso, foram identificadas as seguintes lições aprendidas:

1. **Importância da escolha do *framework* no desenvolvimento:** a escolha correta do *framework* de desenvolvimento pode proporcionar um melhor aproveitamento das ferramentas disponíveis. O `Django` dispõe de uma interface administrativa que auxilia o gerenciamento do *Back-end* da aplicação. O `Laravel` dispõe de uma vasta biblioteca de pacotes que podem agilizar o desenvolvimento do projeto. O `Rails`, por sua vez, possui uma sintaxe intuitiva e pouca configuração manual inicial. Assim, é de grande importância o alinhamento do *framework* às necessidades do projeto, garantindo que os recursos disponíveis otimizem o desenvolvimento do *software*.
2. **Importância da comunidade e do suporte:** o estudo revelou que a comunidade e o suporte são de suma importância na produtividade e resolução de problemas durante o desenvolvimento do projeto. Enquanto o `Laravel` e o `Django` possuem grandes comunidades, documentação oficial detalhada, o `Rails` possui uma comunidade experiente, mas certas soluções de problemas podem ser dificultosas de achar uma estratégia de solução por se tratar de um *framework* com menor penetração no mercado. Essa experiência demonstrou que escolher um *framework* com comunidade engajada e com bom suporte reduz o tempo de desenvolvimento e facilita na superação de percalços.
3. **A importância do planejamento do *software*:** ao projetar e prototipar o *software* de extensão, foi evidenciada a importância do planejamento no desenvolvimento do *software*. A criação detalhada das funcionalidades e determinação dos atores e casos de uso permitem uma visão clara do sistema e seu fluxo de funcionamento. Além disso, a prototipagem das telas possibilita uma posterior validação da experiência do usuário. Dessa forma, um planejamento bem estruturado de um *software* melhora a organização do projeto e aumenta a agilidade de desenvolvimento.

5 CONCLUSÕES E TRABALHOS FUTUROS

Todos os *frameworks* analisados se apresentam poderosas ferramentas de desenvolvimento e a escolha dependerá do contexto de uso. Sendo assim, conforme os resultados obtidos, pode-se afirmar que cada um possui características particulares que o tornam adequados em diferentes contextos.

O `Laravel` possui vantagem na quantidade de recursos disponíveis em seu ecossistema, ideal para projetos cujo maior objetivo é o aumento da agilidade de desenvolvimento. O `Rails` se mostra mais vantajoso para projetos que priorizam rapidez em desenvolvimento e poucas convenções de configuração manuais, além de facilitar escalabilidade e manutenção de código. A principal vantagem do `Django` é sua interface administrativa que auxilia e facilita a administração do banco de dados da aplicação. É ideal para desenvolvedores que buscam implementar projetos focados no *Back-end* com pouca escrita de código de consulta.

Para o contexto de aplicação do *software* de controle de horas de extensão, projetado durante a realização deste trabalho, destaca-se o *framework* `Django` com sua página de administração que facilita na manipulação dos arquivos comprobatórios enviados pelos discentes, além de ser menor em tamanho de projeto e apresentar proteção automática contra os principais ataques em aplicações *Web*.

Este trabalho permitiu comparar os *frameworks* `Ruby on Rails`, `Laravel` e `Django`, além da modelagem de um sistema de integração de horas de extensão, destacando a importância da avaliação de requisitos para um projeto na escolha da tecnologia de implementação. Como trabalhos futuros, sugere-se a implementação do sistema com todas as suas funcionalidades utilizando um ou mais *frameworks* e posterior análise de desempenho aprofundada, avaliando sua performance utilizando os mesmos ou mais critérios de comparação.

REFERÊNCIAS

ANJUM, N.; ALAM, S. A comparative analysis on widely used web frameworks to choose the requirement based development technology. **IARJSET**, v. 6, p. 16–24, 09 2019.

ANKUSH_953. **Views In Django | Python**. 2016. Acessado em 19 de Maio de 2023: https://www.geeksforgeeks.org/views-in-django-python/.

BENNETT, J. **Practical Django Projects**. Apress, 2009. ISBN 9781430219385. Disponível em: https://books.google.com.br/books?id=4KArzQEACAAJ.

BOUCHER, J. **Django Middleware: An overview of middleware**. 2016. Acessado em 19 de Maio de 2023: https://jodyboucher.com/blog/django-middleware-overview.

BRASIL. **Resolução CNE/CES nº 7, de 18 de dezembro de 2018**. 2018. Acessado em 23-junho-2023, disponível em: https://normativasconselhos.mec.gov.br/normativa/view/CNE_RES_CNECESN72018.pdf.

CARDOSO, L. d. C. **Frameworks Back End**. [*S. l.*]: Editora Saraiva, 2021.

DJANGO. **Django**. 2023. Acessado em 28-maio-2023, disponível em: https://www.djangoproject.com/.

DOCS, M. W. **Tutorial Django Parte 8: Autenticação de usuário e permissões**. 2024. Acessado em 06-setembro-2024, disponível em: https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Django/Authentication.

FIGUEIREDO, S. C. G. de. Atividades de extensão: A curricularização da extensão no ensino superior. **PRODUÇÃO ACADÊMICA E PLURALIDADE**, p. 229, 2020.

GABARDO, A. C. **Laravel para ninjas**. [*S. l.*]: Novatec Editora, 2017.

HOLOVATY, A.; KAPLAN-MOSS, J. **The definitive guide to Django: Web development done right**. [*S. l.*]: Apress, 2009.

LARAVEL. **Laravel**. 2023. Acessado em 15-junho-2023, disponível em: https://laravel.com/.

MACIEL, F. de B. **Python e Django: desenvolvimento web moderno e ágil**. Alta Books, 2020. ISBN 9786555200973. Disponível em: https://books.google.com.br/books?id=Td_aDwAAQBAJ.

MACIEL, R. S. P.; ASSIS, S. R. de. **Uma solução para o desenvolvimento de aplicações distribuídas**. [*S. l.*]: CienteFico, 2004.

MICHAEL, B.; KIRCHBERG, P. *et al.* Ruby on rails. **IEEE software**, v. 6, n. 24, p. 105–108, 2007.

MILETTO, E. M.; BERTAGNOLLI, S. de C. **Desenvolvimento de Software II: Introdução ao Desenvolvimento Web com HTML, CSS, JavaScript e PHP-Eixo: Informação e Comunicação-Série Tekne**. [*S. l.*]: Bookman Editora, 2014.

PANWAR, V. Web evolution to revolution: Navigating the future of web application development. **International Journal of Computer Trends and Technology**, v. 72, p. 34–40, 02 2024.

PATEL, J. **10 Reasons Why Laravel is Better Than Other PHP Frameworks?** 2023. Acessado em 24-junho-2023, disponível em: https://www.monocubed.com/blog/why-laravel-php-framework/.

PATEL, M. **Ruby on Rails Architecture | Everything You Need To Know**. 2023. Acessado em 07 de Abril de 2024: https://www.rorbits.com/ruby-on-rails-architecture/.

PRESSMAN, R. S.; MAXIM, B. R. **Engenharia de software-9**. [*S. l.*]: McGraw Hill Brasil, 2021.

RAILS, R. on. **Ruby on Rails**. 2024. Acessado em 29-fevereiro-2024, disponível em: https://rubyonrails.org/.

RAMOS, V. **DESENVOLVIMENTO WEB COM PYTHON E DJANGO: INTRODUÇÃO**. 2023. Acessado em 20 de Abril de 2023: https://pythonacademy.com.br/blog/desenvolvimento-web-com-python-e-django-introducao.

REST, D. **Django REST**. 2024. Acessado em 26-agosto-2024, disponível em: https://www.django-rest-framework.org/.

RUBY. **Ruby**. 2024. Acessado em 12-junho-2024, disponível em: https://www.ruby-lang.org/pt/.

RUBY, S.; THOMAS, D. **Agile Web Development with Rails 7**. [*S. l.*]: Pragmatic Bookshelf, 2023.

SOARES, B. C. Requisitos para utilização de prototipagem evolutiva nos processos de desenvolvimento de software baseado na web. **Belo Horizonte: UFMG**, 2008.

STAUFFER, M. **Laravel: Up & Running: A Framework for Building Modern PHP Apps**. O'Reilly Media, 2019. ISBN 9781492041184. Disponível em: https://books.google.com.br/books?id=HcqPDwAAQBAJ.

THOMAS, D.; HUNT, A. **The Pragmatic Programmer: your journey to mastery**. [*S. l.*]: Addison-Wesley Professional, 2019.

VERMA, A. Mvc architecture: A comparative study between ruby on rails and laravel. **Indian Journal of Computer Science and Engineering (IJCSE)**, v. 5, n. 5, p. 196–198, 2014.

VISWANATHAN, V. Rapid web application development: a ruby on rails tutorial. **IEEE software**, IEEE, v. 25, n. 6, p. 98–106, 2008.