JOÃO MARCOS MARMONTELO

IMPLEMENTAÇÃO DO SISTEMA NACIONAL DE INFORMAÇÕES SOBRE IRRIGAÇÃO

2023

JOÃO MARCOS MARMONTELO

IMPLEMENTAÇÃO DO SISTEMA NACIONAL DE INFORMAÇÕES SOBRE IRRIGAÇÃO

Relatório de Estágio Supervisionado apresentado à Universidade Federal de Lavras como parte das exigências do curso de Sistemas de Informação, para obtenção do título de Bacharel.

Profa. Dra. Renata Teles Moreira
Orientadora

2023

Ficha catalográfica elaborada pela Coordenadoria de Processos Técnicos

da Biblioteca Universitária da UFLA

Marmontelo, João Marcos

IMPLEMENTAÇÃO DO SISTEMA NACIONAL DE INFORMAÇÕES SOBRE IRRIGAÇÃO / João Marcos Marmontelo. – Lavras : UFLA, 2023.

31 p. : il.

Tese (Trabalho de Conclusão de Curso)–Universidade Federal de Lavras, 2023.

Orientadora: Profa. Dra. Renata Teles Moreira.

Bibliografia.

1. TCC. 2. Monografia. 3. Dissertação. 4. Tese. 5. Trabalho Científico – Normas. I. Universidade Federal de Lavras. II. Título.

CDD-808.066

Dedico este trabalho aos meus pais, Gilda de Souza Marmontelo e Hélio Marmontelo

AGRADECIMENTOS

A parte mais importante de finalizar uma jornada é poder sentir todos os momentos vividos, conhecimentos aprendidos, decisões tomadas e problemas enfrentados. Agora o que me resta a fazer é apreciar todos os momentos vividos e agradecer a cada minúsculo detalhe que me trouxe aqui.

Agradeço a minha família, principalmente a minha mãe que sempre me apoiou nas minhas jornadas, ao meu falecido pai que sempre batalhou para que pudesse conquistar uma boa carreira e ao meu irmão que sempre me ajudou nesta jornada.

Aos meus amigos, que além de me motivar, me deram incentivos para nunca desistir dos meus objetivos.

Agradeço também a todos os professores que conheci nesta caminhada, pelo trabalho de passar o conhecimento a frente e mudar pessoas com o aprendizado.

E por fim quero agradecer a Deus por permitir a minha existência neste universo, e todas as bençãos que ele fez nesta jornada.

RESUMO

Este documento possui a finalidade de expor a experiência e conceitos aprendidos ao participar de uma bolsa de estímulo à inovação, cujo principal objetivo é o desenvolvimento de um sistema capaz de registrar e disponibilizar informações sobre projetos públicos de irrigação. Motivado pela necessidade de organizar informações sobre os projetos públicos, o Ministério da Integração e do Desenvolvimento Regional junto com a Fundação de Desenvolvimento Científico e Cultural entraram em parceria para a implementação do Sistema Nacional de Informações Sobre Irrigação.

Palavras-chave: Desenvolvimento de software. Spring Boot. Java.

ABSTRACT

This document is intended to present the experience and concepts learned by participating in an innovation grant, whose main objective is the development of a system capable of registering and providing information on public irrigation projects. Motivated by the need to organize information on public projects, the Ministry of Integration and Regional Development, together with the Foundation for Scientific and Cultural Development, entered into a partnership to implement the National Irrigation Information System.

Keywords: Software development. Spring Boot. Java.

LISTA DE FIGURAS

SUMÁRIO

1 INTRODUÇÃO

Os projetos públicos de irrigação são incentivos do governo que tem objetivo de promover o desenvolvimento socioeconômico de regiões economicamente desfavorecidas e com boa disposição agrícola. Com uma infraestrutura de irrigação, esses projetos procuram aumentar a produtividade agrícola, gerar empregos, melhorar a renda da região e promover inclusão social (GOV, 2020). Os projetos possuem alguns desafios como o custo elevado de implantação e operação, conflitos fundiários e impactos ambientais. Para combater esses desafios é necessário melhorar o planejamento, a gestão e as ações de regularização fundiária e de proteção ambiental.

Sendo previsto na Política Nacional de Irrigação (Lei nº 12.787/2013) [1], o Sistema Nacional de Informações sobre Irrigação (SisNIR) tem o objetivo de coletar, armazenar e recuperar informações referentes à agricultura irrigada, além de manter o cadastro nacional único dos agricultores irrigantes. Os princípios básicos deste sistema são: cooperação institucional, coordenação unificada e acesso da sociedade aos dados e às informações. Assim, o propósito geral deste sistema é ampliar e facilitar o acesso às informações sobre os projetos públicos de irrigação e favorecer o planejamento da expansão da agricultura irrigada (GOV, 2023).

O Ministério da Integração e do Desenvolvimento Regional (MIDR) e a Universidade Federal de Lavras (UFLA) juntaram-se para desenvolver o SisNIR, sendo a Fundação de Desenvolvimento Científico e Cultural (FUNDECC) o órgão interno da UFLA responsável por administrar o projeto SisNIR. A FUNDECC dedicou uma equipe capaz de desenvolver o projeto, composta por membros efetivos internos e bolsistas discentes da UFLA. Nesta equipe estão inclusos cinco programadores, um analista de qualidade e um *Product Owner* (PO). Além destes membros, houve a colaboração de um PO adicional prestou assistência ao PO principal.

O SisNIR é um sistema *Web* multicamada com uma base de dados que é composta de dados estruturados relacionais e georreferenciados. As camadas do sistema são:

- Interface de usuário: que resume em uma aplicação *Web* implementada com a biblioteca *React*, que permite o usuário interagir com o sistema. Esta camada é conhecida pelo nome de *front-end*.

- Regra de negócio: sendo um serviço *Web* implementado com o *framework Java Spring Boot*, que é composto de operações do tipo *REST API*. Este serviço é chamado pela ca-

[1] <https://www.planalto.gov.br/ccivil_03/_Ato2011-2014/2013/Lei/L12787.htm>

mada de interface de usuário e tem responsabilidade de processar as solicitações, gerenciar a lógica de negócios e interagir com o banco de dados. Esta camada é conhecida pelo nome de *back-end*.

- Persistência de dados: camada constituída pelo Sistema de Gerenciamento de Banco de Dados (SGBD) *PostgreSQL* com extensão *PostGis* para suporte a armazenamento de dados geográficos. Esta camada é responsável por armazenar os dados processados e ou solicitados pela regra de negócio, é comumente denominada como banco de dados.

O autor deste documento participou no desenvolvimento do *back-end* e do banco de dados, onde o resultado foi um serviço capaz de realizar as operações de *Create* (Criar), *Read* (Ler), *Update* (Atualizar) e *Delete* (Deletar) - conhecidas pela sigla CRUD - de diversas entidades que constituem os requisitos do sistema. As demandas do SisNIR foram passadas para a equipe de desenvolvedores através de *issues* presentes na plataforma *GitLab*. Além das operações de CRUD, foram implementadas diversas regras de negócios como filtros de busca, alterações em entidades, métodos personalizados, cálculos, entre outras.

Além deste capítulo introdutório, este documento está organizado da seguinte forma: o Capítulo 2 apresenta a organização e o produto desenvolvido; o Capítulo 3 descreve o referencial teórico; o Capítulo 4 apresenta desenvolvimento do trabalho; e, por fim, o Capítulo 5 traz as considerações finais.

2 SOBRE A ORGANIZAÇÃO E O PRODUTO

A Fundação de Desenvolvimento Científico Cultural (FUNDECC)[1] é uma personalidade jurídica, fiscalizada pelo Ministério Público, sendo administrada na forma de estatutos, de direito privado e sem fins lucrativos. A FUNDECC é uma fundação de apoio, situada no campus da Universidade Federal de Lavras, cujas finalidades principais são, apoiar o desenvolvimento de projetos de ensino, pesquisa e extensão, e também projetos de desenvolvimento institucional, científico e tecnológico e de estímulo à inovação (FUNDECC, 2023).

2.1 Processo Organizacional

Para organizar o processo de desenvolvimento foi utilizado o *framework Scrum*. Esta metodologia tem como característica principal o processo iterativo e incremental, com entregas regulares do produto ao longo do tempo. Assim, o *Product Owner* era responsável de definir e gerenciar as prioridades do projeto e garantir que o produto suprisse as necessidades dos clientes. Os desenvolvedores eram responsáveis por desenvolver o código do sistema. O desenvolvimento foi divido em uma lista de *issues* que ficavam disponíveis em um quadro *Scrum* na plataforma *GitLab*[2]. Uma *issue* é um problema ou tarefa a ser resolvido, e é composta de uma descrição detalhada do que é necessário realizar e o que é esperado. O fluxo de trabalho do quadro *Scrum* está detalhado na Seção 4.4. Além disso, os códigos entregues eram analisados pelo analista de qualidade, garantindo que o projeto atendesse aos requisitos de qualidade.

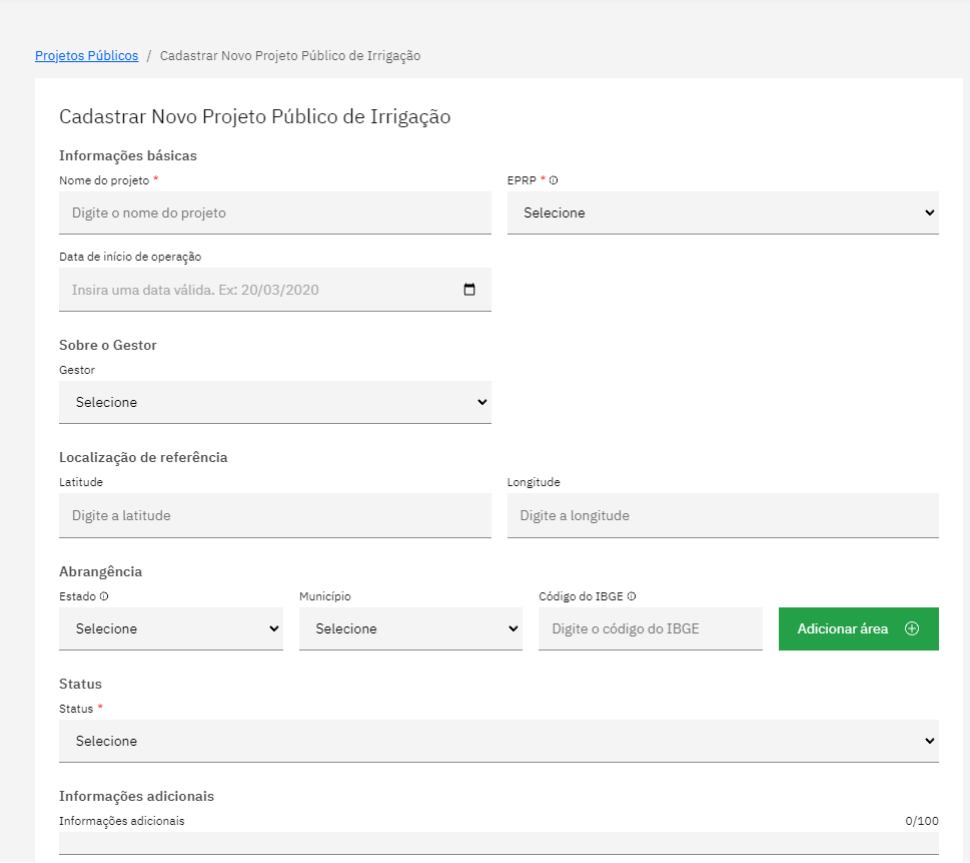
2.2 Produto

O produto final é um sistema *Web* capaz de registrar e exibir informações de diversas entidades, sendo composto por três camadas que se comunicam. A camada *front-end*, exibe paginas para que o usuário possa interagir com o sistema, ao acessar determinada pagina o *front-end* realiza uma ou mais requisições ao *back-end* para obter ou enviar informações. A camada de *back-end* envia ou recebe informações do banco de dados. Assim cada funcionalidade do sistema é composto por um ou mais métodos requisitados. Um exemplo é a tela de cadastro de projetos públicos de irrigação visto na Figura 2.1, nela esta presente um formulário com os campos: Nome do projeto; Entidade Pública Responsável pelo Projeto (EPRP); Data de

[1] <http://www.fundecc.org.br/>
[2] <https://gitlab.com/>

início de operação; Gestor; Latitude e longitude; Status e Informações adicionais. Ao clicar no botão concluir cadastro, o *front-end* fará uma requisição ao *back-end* enviando as informações preenchidas. O *back-end* fica responsável por validar as informações recebidas, persistir-las no banco de dados e retornar para o *front-end* uma mensagem para indicar se a operação foi bem sucedida ou não.

Figura 2.1 – Tela Cadastro de Projeto Público de Irrigação



Fonte: Autor

3 FUNDAMENTAÇÂO TEORICA

Este capítulo tem como objetivo apresentar os conceitos e tecnologias utilizados no projeto descrito neste documento.

3.1 Linguagem de programação Java

Segundo Liang (2019), a linguagem de programação Java é conhecida por ser orientada a objetos, com suporte a *multithreading*, interpretada, livre de arquitetura, portável, robusta, segura e com alto desempenho. Além de Java ser uma linguagem de programação, também está presente uma plataforma composta por uma Interface de Programação de Aplicação, comumente conhecida pela sigla API, e uma máquina virtual.

Além destas características Java possui boa documentação, comunidade ativa de desenvolvedores e alta presença em diversos projetos atuais. E também faz uma checagem em tempo de compilação para identificar linhas de código não alcançáveis e checa variáveis que foram declaradas mas que nunca são utilizadas (FARRELL, 2015).

Java traz detalhes pensados nos programadores e problemas que eles podem encontrar, assim, estes detalhes fazem toda diferença na hora de desenvolver, evitando erros e *bugs* inesperados. Com todas estas características, Java foi designada para ser a linguagem de programação utilizada no *back-end* do projeto SisNIR.

3.2 Framework Spring Boot

Spring é um *framework* de código aberto que foi criado para facilitar o desenvolvimento de aplicações Java, fornecendo recursos e funcionalidades, incluindo, injeção de dependências, controle de transações, arquitetura *Model* (Modelo), *View* (Visão) e *Controller* (Controlador), arquitetura conhecida pela sigla MVC, e testes. A principal vantagem na utilização do *Spring* é poder utilizar os recursos disponíveis para aumentar a produtividade, agilizando o processo de desenvolvimento de aplicações (SPRING, 2023).

Spring Boot oferece uma série de recursos e funcionalidades para a configuração de uma aplicação *Spring*, se comportando como uma extensão do *Spring*. Um dos motivos do surgimento do *Spring Boot* foi a necessidade de configurar de maneira simples os projetos *Springs*, onde o início de um novo projeto é facilitado através de configurações escolhidas pelos programadores e também a flexibilidade de alterá-los em qualquer momento (BIANCHI, 2015).

3.3 API REST

A Transferência de Estado Representacional (REST) abrange uma arquitetura de *software* que define condições de como uma Interface de Programação de Aplicação (API) deve funcionar, possibilitando uma comunicação confiável e de alto desempenho em escala. API representa as regras definidas para poder se comunicar entre os sistemas de *software*. Com isso os desenvolvedores podem criar sistemas capazes de se comunicar de maneira segura. Além disso, essa comunicação ocorre apenas pela exposição da API (AWS, 2023).

Com esta arquitetura, o *back-end* pôde ser realizado de maneira independente do *front-end*, onde o *back-end* é uma API REST com as funcionalidades necessárias que serão utilizadas pelo *front-end*, sem a necessidade da equipe de *front-end* conhecer seu código, mas apenas os detalhes de como consumir a API.

3.4 Insomnia

Para consumir e testar a API resultante do sistema foi utilizada a ferramenta *Insomnia*[1]. Esta ferramenta permite testar os métodos disponíveis de uma API, podendo executar os métodos e visualizar as respostas. Assim, a equipe do *back-end* pôde verificar o comportamento da API independente do *front-end*.

3.5 Swagger

Swagger[2] é uma ferramenta de documentação e design de API para equipes. No projeto esta ferramenta foi utilizada para documentar a API e detalhar os métodos do *back-end* para que a equipe do *front-end* possa visualizar os detalhes de como utilizar o *back-end* sem conhecer o código.

3.6 Gti e GitLab

Git é um Sistema de Controle de Versão (VCS) que tem o objetivo de registrar todas as mudanças realizadas em um projeto, adicionar comentários a cada alteração registrada e reverter o código em qualquer versão do sistema que seja rastreada. Além disso, o *Git* pos-

[1] <https://insomnia.rest/>
[2] <https://swagger.io/>

sui características que favorecem o trabalho colaborativo com diversos integrantes no mesmo projeto (BLISCHAK; DAVENPORT; WILSON, 2016).

Cada projeto rastreado pelo *Git* é chamado de repositório, assim, o *GitLab* [3] é uma plataforma que armazena repositórios. Neste repositório, cada colaborador pode criar uma ramificação do projeto principal para poder publicar suas alterações sem entrar em conflito com as alterações de cada desenvolvedor. Com isso, tem-se um fluxo de trabalho eficiente em projetos com um ou mais colaboradores.

3.7 PostgreSQL

PostgreSQL é um Sistema Gerenciador de Banco de Dados (SGBD) Relacional, que tem por objetivo armazenar informações de qualquer solução tecnológica. Além de armazenar informações, um SGBD é responsável por controlar o acesso de escrita e leitura de informações, gerir o armazenamento de informações e diversas outras responsabilidade (SCHÖNIG, 2020).

3.8 Framework Scrum

Segundo (SCHWABER; SUTHERLAND, 2020) *Scrum* é um *framework* leve e flexível que auxilia equipes e organizações a criar valor de maneira incremental, caracterizado por ciclos curtos e frequentes. *Scrum* oferece várias vantagens como velocidade de entrega, adaptabilidade, aumento da qualidade do produto e melhor satisfação do cliente. O ambiente *Scrum* é composto por três papéis fundamentais que são:

- Um *Product Owner*, responsável por organizar o trabalho, maximizar o valor do produto e definir o *Backlog* do Produto.
- O *Scrum Master*, que auxilia a equipe removendo obstáculos e promovendo um ambiente em que a equipe possa ter sucesso.
- A equipe *Scrum*, que tem objetivo de realizar as tarefas e incrementar o valor do produto.

Além dos papeis, é necessário que ocorram eventos programados com diferentes objetivos que são:

- *Sprint*, um bloco de tempo que possui um conjunto especifico de tarefas que devem ser concluídas.

[3] <https://gitlab.com/>

- Planejamento da *Sprint*, que é uma reunião que ocorre no início de cada *Sprint*, com objetivo de determinar o que pode ser entregue na *Sprint* que estar a iniciar.
- Reuniões diárias, que possuem objetivo de solucionar dúvidas e alinhar informações.
- Retrospectiva, evento realizado ao fim de uma *Sprint*, com o intuito de identificar melhorias e refletir sobre a *Sprint* que chegou ao fim.
- Revisão, evento que ocorre no final de uma *Sprint*, com objetivo de inspecionar o trabalho realizado e obter informações sobre os incrementos, sendo possível gerar mudanças para a próxima *Sprint*, se necessário.

E por fim os artefatos, que são necessários para garantir a transparência, inspeção e adaptação no ambiente *Scrum*, que são:

- *Backlog* do produto: é uma lista de funções novas, aprimoramentos, requisitos de trabalho necessários para criar um produto. Lista criada e mantida pelo PO.
- *Backlog* da *Sprint*: é um conjunto de tarefas do *backlog* do produto que foram designadas a serem realizadas na *Sprint* atual. Tarefas definidas pelo PO.
- Incremento: é o resultado de uma atividade realizada por um ou mais membros da equipe. Os incrementos são compostos por parte do produto.
- Definição de pronto: é um documento contendo uma definição que o incremento esta pronto ou finalizado.

4 DESENVOLVIMENTO

Este capítulo apresenta as atividades desenvolvidas pelo autor no projeto SisNIR.

4.1 Processo seletivo

Para participar da bolsa de estimulo a inovação, foi necessário passar por um processo seletivo. Foram realizadas duas etapas principais: analise de currículo e entrevista. As duas etapas tinham o objetivo de verificar se o candidato tinha experiência no desenvolvimento de software. Após a divulgação do resultado e o envio de alguns documentos, foi iniciado o desenvolvimento do projeto.

4.2 Treinamento

No início do projeto, foram designados materiais de estudo para que a equipe pudesse implementar as primeiras atividades. O tempo dedicado para o treinamento foi de aproximadamente duas semanas. Porém em determinadas atividades foi necessário buscar mais conhecimento, pois houve a necessidade de utilizar determinadas funções que não estavam presentes nos materiais de estudo. O material inicial forneceu instruções sobre o desenvolvimento de APIs utilizando a tecnologia *Spring Boot*, a criação de entidades e seus respectivos CRUDs, a conexão com o SGBD *PostgreSQL*, a utilização das ferramentas de versionamento *Git* e *Gitlab* e, também, como testar as operações de CRUD com a ajuda da ferramenta *Insomnia*.

4.3 Início

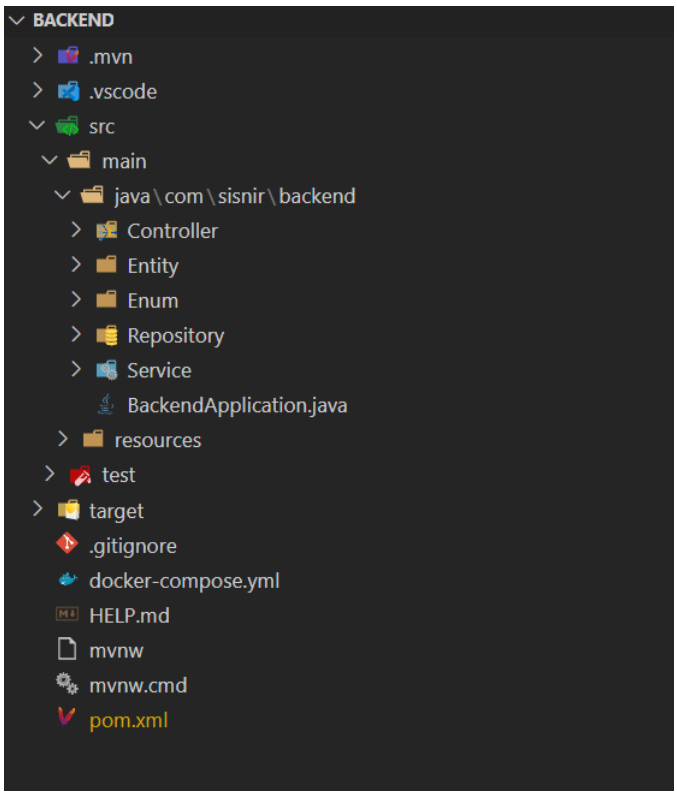
Os primeiros arquivos do projeto, e suas configurações, foram criados por um membro efetivo da FUNDECC. Após o início, a equipe foi orientada a obter o código inicial através de um repositório no *GitLab*. O ambiente de desenvolvimento foi configurado com a instalação de um editor de código e o *Docker* [1]. A Figura 4.1 apresenta a organização de pastas e arquivos definidos no início do projeto. A estrutura do projeto é organizada da seguinte maneira:

- A pasta *main* é a pasta principal que armazena todas as classes codificadas. Essas classes estão separadas em subpastas, cada uma contendo classes de um tipo específico, conforme indicado pelo nome.

[1] <https://www.docker.com/>

- Além da pasta *main*, existe a pasta *test*, que contém todos os testes unitários do projeto.
- Na raiz do projeto, há o arquivo `pom.xml`, que armazena uma lista com todas as dependências necessárias para o projeto.
- Também na raiz do projeto, estão presentes os arquivos de configuração do *Docker*, responsáveis pela configuração e inicialização da aplicação.
- O arquivo `.env` que também está presente na raiz do projeto, é responsável por armazenar variáveis de ambiente que são necessárias para a inicialização da aplicação e conexão com o banco de dados.

Figura 4.1 – Estrutura de pastas e arquivos



Fonte: Autor

4.4 Fluxo de trabalho

Durante o projeto, a equipe utilizou alguns conceitos do *framework Scrum*. O projeto foi desenvolvido em *Sprints*, com duração de trinta dias. Essas *Sprints* eram constituídas pelo conjunto de *issues* disponíveis para desenvolver. Além das *issues* prontas para desenvolver na *sprint*, o projeto possuía um conjunto de *issues* definidas pelos PO (*Backlog* do Produto). Dos

ritos do *Scrum*, o projeto instituiu a reunião diária, que tinha objetivo de esclarecer dúvidas, apresentar o que cada desenvolvedor implementou no dia anterior e o que iria desenvolver ao longo do dia.

O fluxo de trabalho foi baseado no quadro de *issues* presente no *GitLab* [2]. Este fluxo foi utilizado como quado *Scrum* pela equipe, com uma pequena modificação na organização de suas colunas:

- Primeira coluna: Coluna "Aberto". Nesta coluna estão presentes todas as *issues* que serão desenvolvidas, porém, ainda não estão prontas para começar a desenvolver, pois o PO define as *issues* que serão implementadas colocando-as na segunda coluna. Esta lista representa o *Product Backlog* do *framework* Scrum antes de começar uma *sprint*.

- Segunda coluna: Coluna "Pronto para desenvolvimento". Os POs transferem as *issues* da primeira coluna para a segunda, indicando que elas estão prontas para serem executadas pelos desenvolvedores. Quando um desenvolvedor assume uma *issue*, ele deve sinalizar que será o responsável por sua execução, evitando assim confusões na equipe e possíveis conflitos sobre quem está realizando determinada *issue*. Após a escolha da tarefa, o desenvolvedor move a *issue* para a terceira coluna.

- Terceira coluna: Coluna "Em desenvolvimento". Nesta coluna encontram-se as *issues* que estão sendo executadas pela equipe de desenvolvedores. O desenvolvedor responsável por cada *issue* deve criar uma *branch* (ramificação) específica para seu trabalho. Isso permite que cada membro da equipe possa progredir em suas tarefas sem interferir no trabalho dos demais.

- Quarta coluna: Coluna "Pendente". Para esta coluna são designadas as *issues* que não foram concluídas, e que possuem algum obstáculo que impeça de serem finalizadas. Esses obstáculos podem variar desde a espera por informações adicionais até a resolução de problemas técnicos. No entanto, assim que o obstáculo for resolvido, a *issue* poderá ser finalizada e movida para a próxima coluna.

- Quinta coluna: Coluna "Esperando *merge* (Mescla)". Ao concluir uma atividade os desenvolvedores solicitam o *merge* do código para a *branch* principal. No entanto, é necessário designar um membro da equipe para analisar o que foi implementado, com o

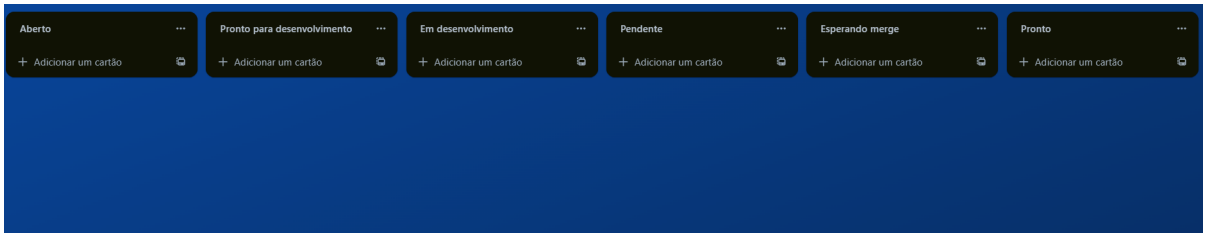
[2] <https://gitlab.com/>

objetivo de identificar possíveis inconsistências e erros. As *issues* analisadas podem ser aprovadas ou reprovadas. As reprovadas devem ser movidas para a terceira coluna (coluna "Em Desenvolvimento") e as aprovadas devem ser movidas para a sexta coluna.

- Sexta coluna: Coluna “Pronto”. Nesta coluna estão presentes as *issues* que foram finalizadas e suas modificações se encontram na *branch* principal. Além disso, esta coluna representa as *issues* que serão enviadas para o servidor de teste, após isso elas são movidas para a sétima coluna.

Na figura 4.2 é apresentado representação do quadro de *issues* com as colunas utilizadas pelos desenvolvedores.

Figura 4.2 – Quadro de *issues* utilizado pelos desenvolvedores



Fonte: Autor

- Sétima coluna: Coluna “Pronto para teste”. Coluna que representa a lista de *issues* que foram enviadas para o servidor de teste. Assim o analista de qualidade pode mover as *issues* para a oitava coluna e inciar os testes.

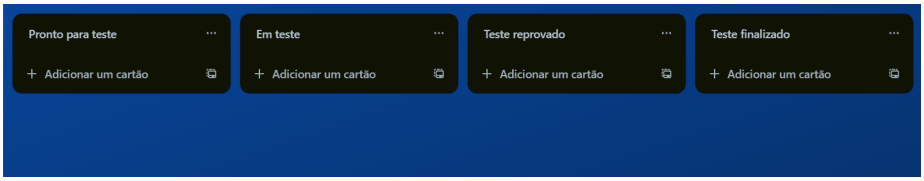
- Oitava coluna: Coluna “Em teste”. Coluna que é composta de *issues* que estão sendo testadas pelo analista de qualidade. Se caso a *issue* passe no teste ela é movida para a décima coluna, caso contrário ela é movida para a nona coluna.

- Nona coluna: Coluna “Teste reprovado”. Coluna que representa as *issues* que não passaram pela analise de qualidade. Neste caso o analista informa o autor da *issue* para ele realizar as correções. O autor da *issue* deve move-la para a terceira coluna e repetir todo o fluxo.

- Décima coluna: Coluna “Teste finalizado”. Coluna composta por todas as *issues* que foram finalizadas e passaram pelo teste de qualidade.

A Figura 4.3 mostra as colunas do quadro que são utilizadas pelo analista de qualidade.

Figura 4.3 – Quadro de *issues* utilizado pelo analista de qualidade



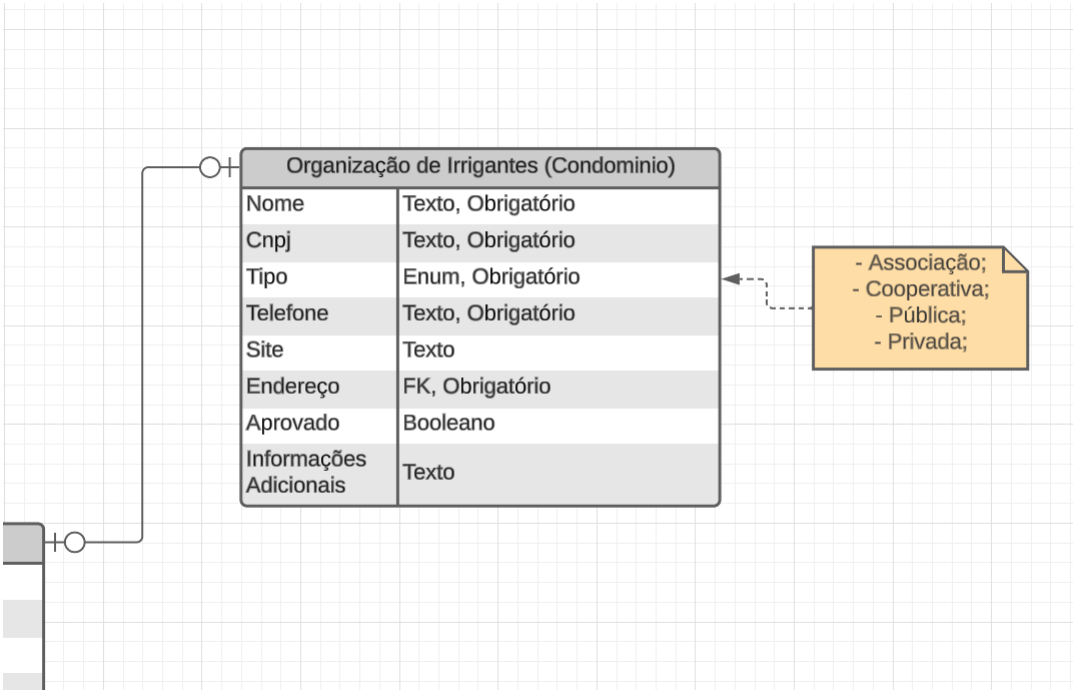
Fonte: Autor

4.5 Atividades realizadas

As primeiras atividades realizadas pela equipe foram de criar as classes do tipo *entity* (entidade). Esse tipo de classe representa os atributos de uma entidade, e, também, uma tabela no banco de dados. Essas classes foram criadas a partir do modelo entidade relacional (MER) disponível na ferramenta *Lucidchart* [3]

Um exemplo de atividade desta etapa foi a criação da *entity* "Organização de Irrigantes". A Figura 4.4 mostra a entidade "Organização de Irrigantes" no MER, e nela está presente todos os seus atributos e seus respectivos tipos, e também os relacionamentos entre as entidades.

Figura 4.4 – MER da entidade Organização de Irrigantes



Fonte: Autor

Na Figura 4.5 é descrito o código da classe "Organização de Irrigantes". As principais características dessa classe são os atributos, conforme o MER, as *annotations* (anotações)

[3] <https://www.lucidchart.com/pages/>

`@Column`, `@Entity` e `@OneToOne`. `@Entity` indica que esta classe será transformada em uma tabela no banco de dados, `@Column` indica que o atributo será uma coluna dentro da tabela no banco de dados, e `@OneToOne` indica que o atributo é uma outra *entity* e terá uma relação de um para um. Além destas características está presente um atributo chamado "TipoOrganizacaoIrrigantes", este atributo é um *enum* que é uma classe do tipo enumeradora, esta classe é um conjunto de valores do tipo texto, e este atributo apenas poderá possuir os valores definidos no *enum*.

Figura 4.5 – *Entity* Organização de Irrigantes

```java
@Entity
@Data
public class OrganizacaoDeIrrigantes implements Serializable{
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false)
    private String nome;

    @Column(nullable = false)
    private String cnpj;

    @Column(nullable = false)
    private TipoOrganizacaoDeIrrigantes tipo;

    @Column(nullable = false)
    private String telefone;

    @Column
    private String site;

    @OneToOne(cascade = CascadeType.REMOVE)
    @JoinColumn(name = "enderenco_id")
    private Endereco endereco;

    @Column
    private Boolean aprovado;

    @Column
    private String informacoesAdicionais;

}
```

Fonte: Autor

Após as primeiras atividades, foi realizado o desenvolvimento dos CRUDs, referentes às entidades criadas. Para isso, foi necessário criar mais três classes para cada CRUD. Cada classe tem uma responsabilidade dentro do projeto. A entidade "Organização de Irrigantes" será utilizada neste texto como exemplo para explicar estas classes. Por motivo de segurança, os códigos e atributos apresentados neste documento não são reais, apenas semelhantes.

A primeira classe é do tipo *controller* (controladora). Seguindo o exemplo da entidade "Organização de Irrigantes", tem-se a classe `OrganizacaoDeIrrigantesController.java`. Esta classe possui a responsabilidade de disponibilizar os métodos públicos que serão utilizados por outros sistemas, sendo o único meio de comunicação externo. Assim, tem-se os métodos disponíveis, o tipo de retorno de cada método e seus parâmetros, conforme mostrado na Figura 4.6. Além disso nesta figura esta presente as anotações, `@PostMapping`, `@PutMapping`, `@GetMapping` e `@DeleteMapping`. `@PostMapping` indica que o método será do tipo POST, e significa que será adicionado um novo elemento no banco de dados. `@PutMapping` indica que o método será do tipo PUT, onde o comportamento será de alteração de algum registro no banco de dados. `@GetMapping` indica que o método será do tipo GET, método responsável por retornar um ou vários elementos do banco de dados. `@DeleteMapping` define que o método será do tipo DELETE, com o comportamento de remover um elemento do banco de dados.

Figura 4.6 – *Controller* da Organização de Irrigantes

```java
@RestController
@RequestMapping("/organizacao-de-irrigantes")
public class OrganizacaoDeIrrigantesController {

    @Autowired
    private OrganizacaoDeIrrigantesService organizacaoDeIrrigantesService;

    @PostMapping
    @Transactional
    public ResponseEntity<OrganizacaoDeIrrigantes> salvaOrganizacaoDeIrrigantes(
            @RequestBody OrganizacaoDeIrrigantes organizacao) {
        return organizacaoDeIrrigantesService.salvaOrganizacaoDeIrrigantes(organizacao);
    }

    @PutMapping("/{id}")
    @Transactional
    public ResponseEntity<OrganizacaoDeIrrigantes> editaOrganizacaoDeIrrigantes(@PathVariable Long id,
            @RequestBody OrganizacaoDeIrrigantes organizacaoNova) {
        return organizacaoDeIrrigantesService.editaOrganizacaoDeIrrigantes(id, organizacaoNova);
    }

    @GetMapping
    public ResponseEntity<Page<OrganizacaoDeIrrigantes>> retornaTodasOrganizacoesDeIrrigantes(Pageable pageable) {
        return organizacaoDeIrrigantesService.retornaTodasOrganizacoesDeIrrigantes(pageable);
    }

    @GetMapping("/{id}")
    public ResponseEntity<OrganizacaoDeIrrigantes> retornaOrganizacaoDeIrrigantesPorId(@PathVariable Long id) {
        return organizacaoDeIrrigantesService.retornaOrganizacaoDeIrrigantesPorId(id);
    }

    @DeleteMapping("/{id}")
    public ResponseEntity<String> deletaOrganizacaoDeIrrigantes(@PathVariable Long id) {
        return organizacaoDeIrrigantesService.deletaOrganizacaoDeIrrigantes(id);
    }
}
```

Fonte: Autor

A segunda classe é do tipo *repository* (Repositório). Tem-se a classe `OrganizacaoDeIrrigantesRepository.java`, onde sua funcionalidade é a de comunicar com o banco de dados. Para isso é necessário estender uma interface que traz os principais métodos necessários para a comunicação com o banco de dados. Além de possuir diversos métodos, a interface realiza a criação automática das tabelas no banco de dados, mapeando as características das entidades. Na Figura 4.7 é mostrado o código da classe, incluindo apenas a declaração da classe, pois os métodos necessários são herdados da classe *JpaRepository*.

Figura 4.7 – *Repository* da Organização de Irrigantes

```
import org.springframework.data.jpa.repository.JpaRepository;

import com.sisnir.backend.Entity.OrganizacaoDeIrrigantes;

public interface OrganizacaoDeIrrigantesRepository extends JpaRepository<OrganizacaoDeIrrigantes, Long> {

}
```

Fonte: Autor

A terceira classe é do tipo *service* (Serviço). Tem-se, então, a classe `OriganizacaoDeIrrigantesService.java`, onde é realizada a implementação da regra de negócio do sistema, e, pode estar contido verificações, atribuições, cálculos entre outras funcionalidades que variam de acordo com os requisitos. Na Figura 4.8 é exibida a declaração da classe e o método correspondente a função de salvar de um CRUD. Devido ao tamanho dos métodos, neste documento foi descrito apenas um deles.

Figura 4.8 – *Service* da Organização de Irrigantes

```
@Service
public class OrganizacaoDeIrrigantesService {
    @Autowired
    OrganizacaoDeIrrigantesRepository organizacaoDeIrrigantesRepository;

    @Autowired
    EnderecoRepository enderecoRepository;

    public ResponseEntity<OrganizacaoDeIrrigantes> salvaOrganizacaoDeIrrigantes(OrganizacaoDeIrrigantes organizacao) {

        Optional<Endereco> optionalEndereco = enderecoRepository.findById(organizacao.getEndereco().getId());
        if (optionalEndereco.isPresent()) {
            organizacao.setEndereco(optionalEndereco.get());
        } else {
            organizacao.setEndereco(endereco:null);
        }

        return ResponseEntity.status(HttpStatus.CREATED).body(organizacaoDeIrrigantesRepository.save(organizacao));
    }
```

Fonte: Autor

Cada CRUD necessita de quatro classes distintas. A relação entre estas classes é descrita a seguir.

A classe do tipo *controller* possui métodos públicos, estes métodos possuem parâmetros que podem ser identificadores e objetos do tipo *entity*. Os métodos da *controller* são responsáveis por realizar a chamada aos métodos da classe *service* passando os parâmetros.

A classe do tipo *service* processa as informações obtidas pelos parâmetros e utiliza os métodos disponíveis da classe *repository* e por fim retorna uma resposta para a *controller*.

4.6 Demais atividades

Após a implementação das *entities* e seus CRUDs, foram adicionados diversas *issues* ao projeto. Devido às solicitações dos clientes, foram necessárias modificações em algumas entidades e suas relações. Os POs, então, atualizaram o MER e criaram *issues* para os desenvolvedores implementarem. As alterações envolvendo atributos simples foram resolvidas diretamente, enquanto as que exigiam tratamento específico ou envolviam relacionamentos se mostraram mais complexas, necessitando de grandes modificações.

Um exemplo de modificação foi a de alterar o nome da entidade "Organização de Irrigantes" para "Gestor Distrito". Devido a diversas classes utilizarem métodos correspondente ao CRUD desta entidade, foi necessário alterar todas as classes que utilizavam de alguma maneira algo desta entidade.

Além de modificações, os clientes solicitaram funcionalidades novas no decorrer do projeto. Uma funcionalidade nova, por exemplo, foi a de adicionar filtros de busca na entidade "Projeto Público de Irrigação". Esta funcionalidade está inclusa nas diversas *issues* que necessitaram de mais estudo, pois os filtros foram adicionados na classe *repository* da entidade, e necessitavam de conhecimento da linguagem SQL. Conforme é mostrado na Figura 4.9, tem-se a *repository* referente ao "Projeto Público de Irrigação" onde está presente o método de busca por parâmetros.

Figura 4.9 – *Repository* do Projeto Público de Irrigação

```
@Repository
public interface ProjetoPublicoIrrigacaoRepository extends BaseRepository<ProjetoPublicoIrrigacao, Long> {

    @Query("select ppi from ProjetoPublicoIrrigacao ppi where ppi.deleted = false and" +
            "(?1 IS NULL OR LOWER(ppi.nome) LIKE '%'||LOWER(?1)||'%') and" +
            "(?2 IS NULL OR ppi.status = ?2) and" +
            "(?3 IS NULL OR ppi.etapa = ?3) and" +
            "(?4 IS NULL OR ppi.eprp IN (select e from EPRP e where e.id = ?4))")
    Page<ProjetoPublicoIrrigacao> findByParameters(Pageable pageable, String nome, String status, String etapa, Long id_EPRP);

}
```

Fonte: Autor

Durante o desenvolvimento do sistema, os clientes perceberam a necessidade de adicionar verificações e condições, que foram passadas para os desenvolvedores adicionarem nas classes *services* de determinadas entidades. Por exemplo, o cálculo de diversos atributos, que antes eram passados pelo *front-end*, mas que fazia mais sentido ser calculado no *back-end* e armazenado no banco de dados.

4.7 Produto

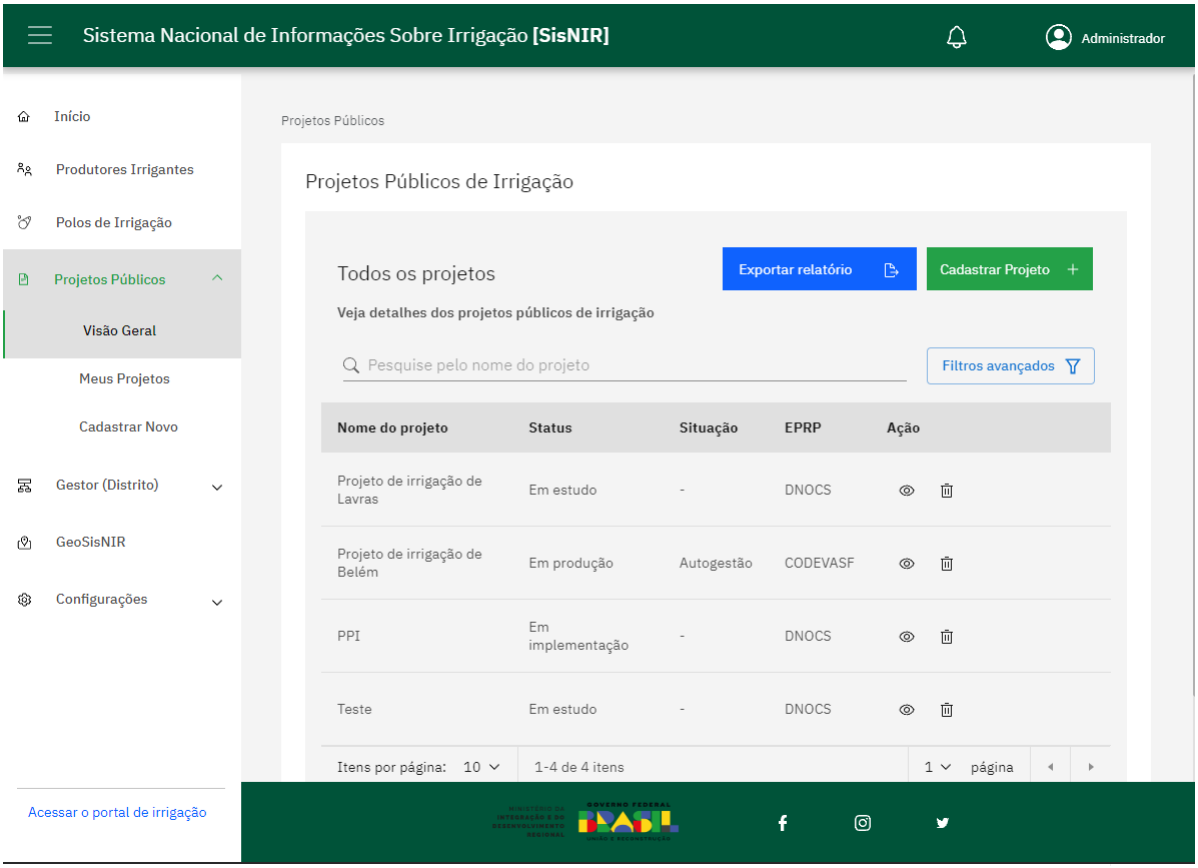
O produto resultante foi um sistema Web multicamada com diversas funcionalidades. A camada *front-end* foi implementada por uma equipe terceirizada, e foi utilizado o *framework React* que tem sua linguagem base *JavaScript*. O *back-end* e o banco de dados foi implementado pela equipe em que o autor estava incluso. Devido à quantidade de opções e funcionalidades, será apresentado apenas uma parte das funcionalidades do sistema.

A Figura 4.10 mostra a tela de visão geral dos projetos públicos de irrigação e nela estão presentes as seguintes funcionalidades:

- Menu lateral: fornece acesso a outras telas do sistema, como início, produtores irrigantes, polos de irrigação, etc.
- Botão exportar relatório: gera uma planilha contendo um relatório geral de todos os projetos públicos.
- Campo de pesquisa: campo que permite a busca de projetos públicos pelo nome.
- Botão filtros avançados: abre um submenu contendo diversos filtros de busca.
- Lista de projetos públicos: lista todos os projetos públicos cadastrados e informações como: nome do projeto, status, situação, Entidade Pública Responsável pelo Projeto (EPRP) e ação. Na parte de ação estão presentes os botões visualizar e deletar, representado por um olho e uma lixeira respectivamente.

Nesta Tela foi necessário uma requisição ao *back-end* para obter todos os projetos públicos do banco de dados para a exibição da lista.

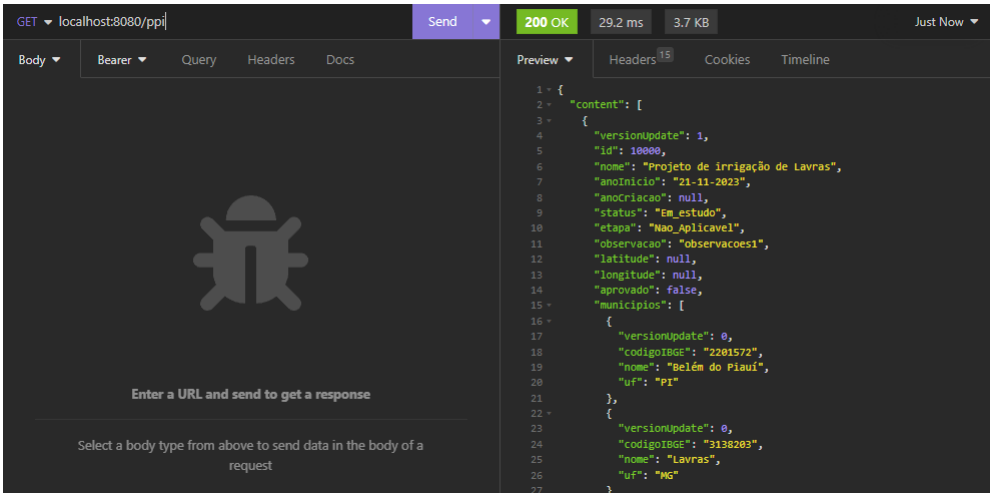
Figura 4.10 – Tela Visão Geral de Projetos Públicos de Irrigação



Fonte: Autor

A Figura 4.11 exibe a requisição “listar todos os projetos públicos”, mesmo método necessário na Figura 4.10, porém exibido pela ferramenta *Insomnia*, que foi utilizada pelos desenvolvedores para testar os códigos desenvolvidos antes do *front-end* estar disponível. Nela está incluso o tipo do método (GET), o caminho do método, que é `localhost:8080/ppi`, e a direita está presente o conteúdo e o status da requisição.

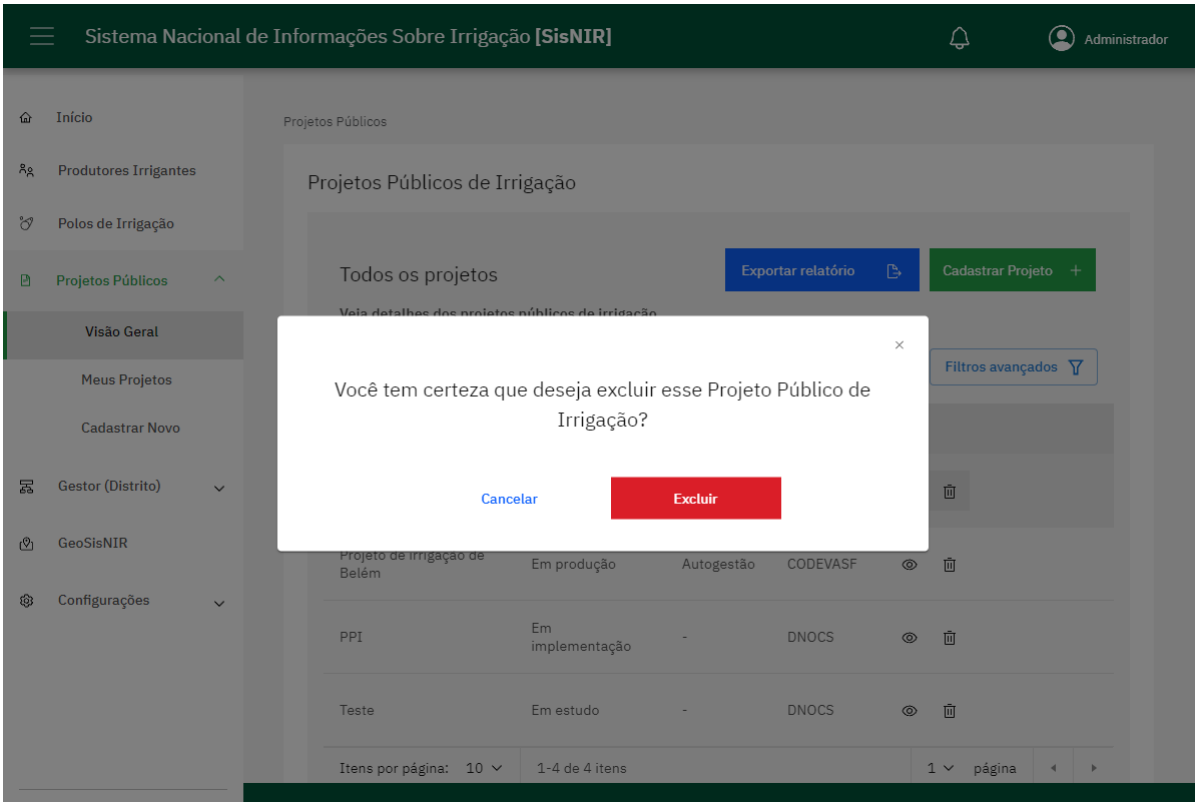
Figura 4.11 – Requisição GET na ferramenta *Insomnia*



Fonte: Autor

A Figura 4.12 mostra a janela “exclui projeto público”, exibida ao clicar no botão representado por uma lixeira. Nela, tem-se os botões “Cancelar” que fecha a janela, e o botão “Excluir”, que faz uma requisição ao *back-end* solicitando a exclusão da entidade, confirmando a remoção do projeto público.

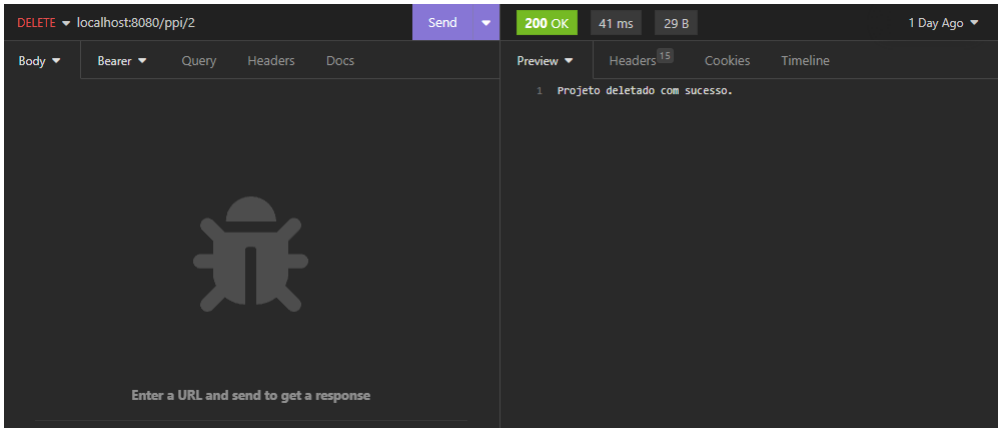
Figura 4.12 – Tela Excluir Projeto Público



Fonte: Autor

Na Figura 4.13 está presente a mesma requisição mencionada na Figura 4.12 porém feito no *Insomnia*, onde está presente o tipo (DELETE) da requisição, o caminho (`localhost:8080/ppi/2`), o status e o conteúdo do retorno.

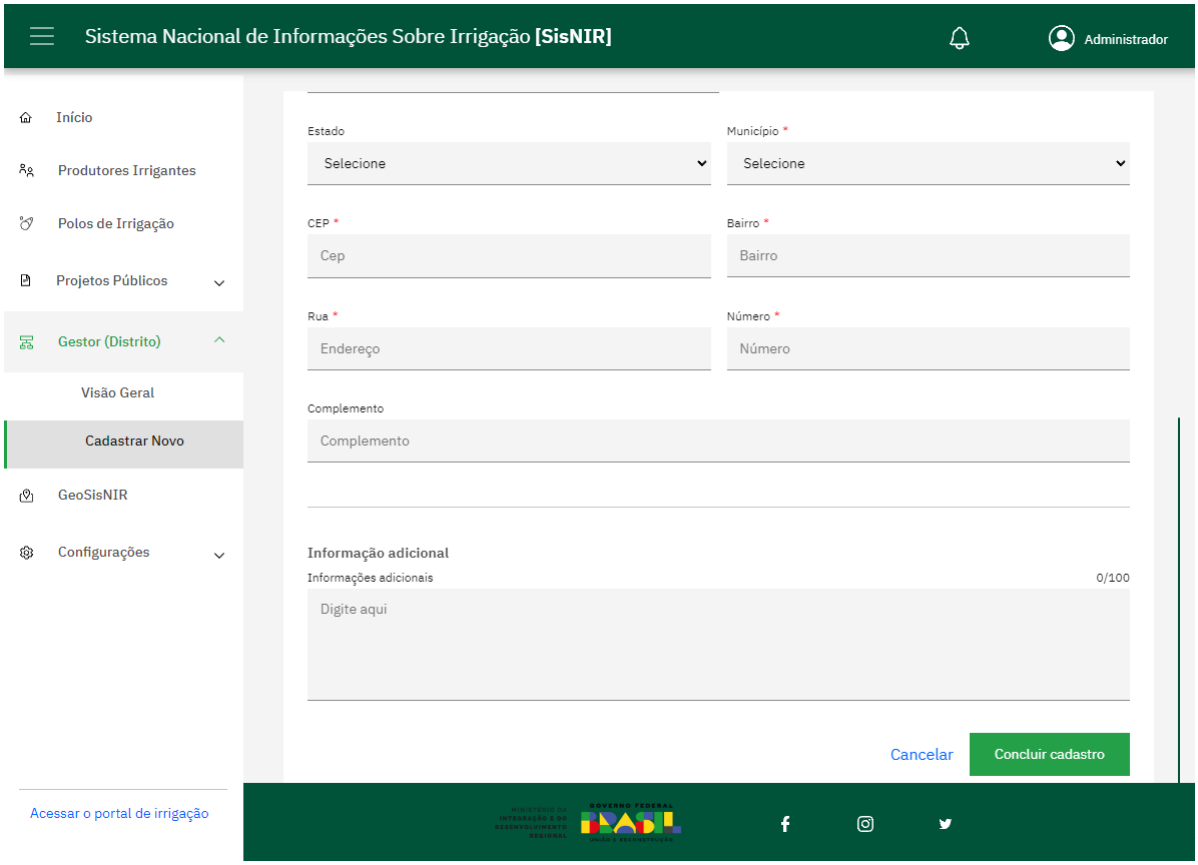
Figura 4.13 – Requisição DELETE na ferramenta *Insomnia*



Fonte: Autor

A Figura 4.14 mostra a tela de cadastro de projeto público de irrigação. Nela tem-se o menu lateral e os campos necessários para o cadastro. Ao preencher todas as informações e clicar no botão de concluir o cadastro, o *front-end* faz uma requisição ao *back-end*, passando os dados do projeto público para o *back-end* processar e salvar no banco de dados.

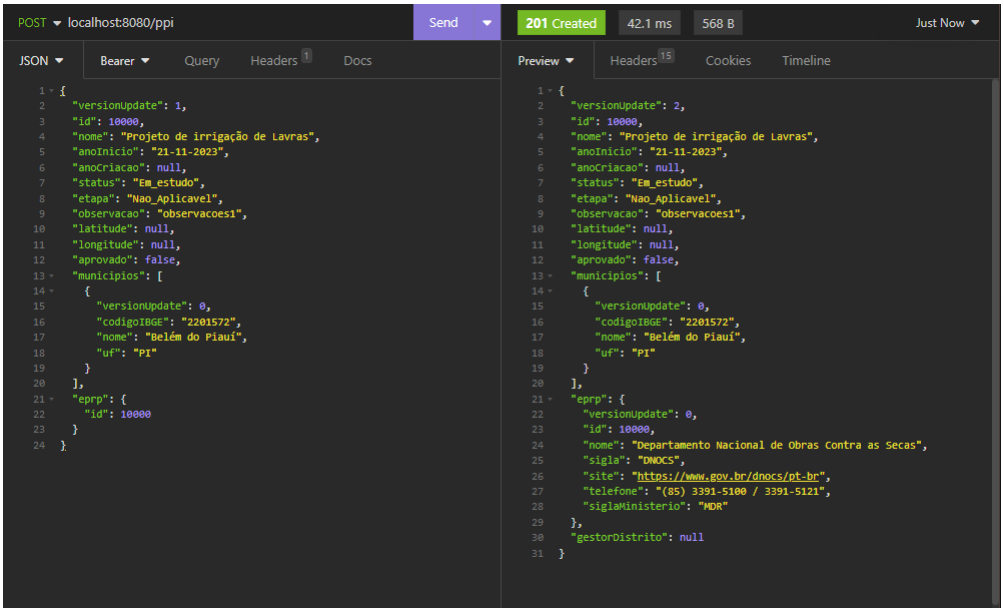
Figura 4.14 – Tela Cadastro de Projeto Público



Fonte: Autor

A Figura 4.15 exibe a ferramenta *Insonmia* com a mesma requisição da Figura 4.14, que solicita o cadastro de um projeto público para o *back-end*. A requisição é do tipo POST, o caminho é (`localhost:8080/ppi`) e o corpo da requisição contém as informações do projeto público. Para realizar esta requisição, é necessário enviar as informações na parte *body*. Ao realizar esta requisição, o *back-end* fará uma verificação das informações e enviará para o banco de dados.

Figura 4.15 – Requisição POST na ferramenta *Insomnia*



Fonte: Autor

5 CONCLUSÃO

Este documento descreveu a experiência e os conhecimentos adquiridos pelo autor ao participar do projeto SisNIR, com foco na implementação da regra de negócio e a persistência de dados. Além disso, este documento relatou como foi a organização do fluxo de trabalho, como foi o desenvolvimento do projeto com exemplos de atividades realizadas pelo autor, a metodologia e as ferramentas utilizadas para gerenciar e organizar a produtividade.

No decorrer do desenvolvimento o autor pôde ter suas habilidades de programação ampliadas, devido a presenciar as diversidades de um projeto, onde teve que lidar com erros, decisões a fazer em prol do sistema, alterações de última hora, falhas de comunicação entre a equipe, entre outras adversidades. Contudo, o projeto teve seu sucesso e o autor pode expandir seu conhecimento tanto no desenvolvimento de software quanto no de cooperação e comunicação.

Ao realizar as tarefas o autor pode colocar em prática o conteúdo das seguintes disciplinas: engenharia de software, introdução a sistemas de banco de dados, introdução aos algoritmos, estruturas de dados, programação orientada a objetos, entre outras. A disciplina introdução a sistemas de banco de dados contribuiu para tomar decisões na hora de criar o banco de dados e as consultas personalizadas. A disciplina práticas de programação orientada a objetos, contribuiu no desenvolvimento dos códigos, pois a linguagem Java é orientada a objetos. A disciplina estruturas de dados, contribuiu para a manipulação dos dados e objetos, pois em determinadas situações foram necessários utilizar os conceitos vistos em sala. Além do conteúdo das disciplinas vistos em prática, o autor teve contato com a criação e utilização de APIs, conteinerização, versionamento de código e testes unitários.

A equipe teve grande sucesso no desenvolvimento do projeto, apesar de alguns pontos negativos que merecem a atenção para melhorar a produtividade de futuros projetos desenvolvidos pela FUNDECC. A falta de um administrador de banco de dados e de um líder técnico, por exemplo, foram pontos que mostraram ser necessários em alguns momentos. Para evitar esses pontos negativos é preciso ter uma equipe que possua um líder técnico com grande experiencia e um administrador de banco de dados que auxilie nas tomadas de decisões.

Apesar dos desafios, o projeto teve seu objetivo atingido. A cooperação e a união da equipe foram fundamentais para o sucesso. Não houve grandes conflitos que atrapalhassem o desenvolvimento, e os membros se apoiavam mutuamente, o que facilitou o trabalho.

REFERÊNCIAS

AWS. **O que é API RESTful?** 2023. Disponível em: <https://aws.amazon.com/pt/what-is/restful-api>. Acesso em: 22/11/2023.

BIANCHI, E. A. Sistema de envio e recebimento de mensagens para plataforma android utilizando spring boot e google cloud messaging. Universidade Tecnológica Federal do Paraná, 2015.

BLISCHAK, J. D.; DAVENPORT, E. R.; WILSON, G. A quick introduction to version control with git and github. **PLoS computational biology**, v. 12, n. 1, p. e1004668, 2016.

FARRELL, J. **Java programming**. [S.l.]: Course Technology Press, 2015.

FUNDECC. **Sobre a FUNDECC**. 2023. Disponível em: <http://www.fundecc.org.br/sobre-a-fundecc/>. Acesso em: 22/11/2023.

GOV. Projetos de irrigação. **Ministério da Integração e do Desenvolvimento Regional**, fev. 2020. Disponível em: <https://www.gov.br/mdr/pt-br/assuntos/irrigacao/projetos-de-irrigacao>. Acesso em: 22/11/2023.

GOV. **Sistema Nacional de Informações sobre Irrigação**. 2023. Disponível em: <https://www.gov.br/mdr/pt-br/assuntos/irrigacao/sistema-nacional-de-informacoes-sobre-irrigacao>. Acesso em: 22/11/2023.

LIANG, Y. D. **Introduction to Java programming and data structures: comprehensive version**. [S.l.]: Pearson, 2019.

SCHÖNIG, H.-J. **Mastering PostgreSQL 13: Build, administer, and maintain database applications efficiently with PostgreSQL 13**. [S.l.]: Packt Publishing Ltd, 2020.

SCHWABER, K.; SUTHERLAND, J. The 2020 scrum guide. **Scrum Org, November**, 2020.

SPRING. **Spring Framework**. 2023. Disponível em: <https://spring.io>. Acesso em: 22/11/2023.