

**ETEC ELIAS NECHAR**  
**DESENVOLVIMENTO DE SISTEMAS**

ESTEVÃO MIGUEL ZANOTI;  
GABRIEL FERNANDES MENONI;  
JÚLIO RANGEL DOS SANTOS PINTO;  
VINÍCIUS RODRIGUES DE ARO.

**ARTIGO**  
INFINITUM

CATANDUVA  
2021

VINÍCIUS RODRIGUES DE ARO  
ESTEVÃO MIGUEL ZANOTI  
JÚLIO RANGEL DOS SANTOS PINTO  
GABRIEL FERNANDES MENONI

**ARTIGO**  
INFINITUM

Trabalho de Conclusão de Curso apresentado à  
escola ETEC Elias Nechar, como parte dos requisitos  
para a obtenção do título de Técnico em  
Desenvolvimento de Sistemas

Orientador: Prof. Alessandro Aparecido Sandrini  
Coorientador: Prof. Sérgio Augusto Pelicano Junior

CATANDUVA  
2021

## RESUMO

Algumas equipes de desenvolvedores encontram dificuldades para coordenar corretamente os seus projetos, e com isso surgem dificuldades, podendo comprometer o fluxo de seu planejamento. O objetivo deste trabalho é o desenvolvimento de um *Bug Tracker*, o qual também é conhecido como Gerenciador de Projetos, para que deste modo, os criadores possam gerir e realizar as tarefas de forma organizada. A proposta funcionara através de uma interface gráfica, que seria comandada por um administrador, sendo ele o responsável por passar as tarefas aos demais participantes. A criação deste, tem como base, uma pesquisa realizada à volta dos desenvolvedores, que dizem que um dos seus maiores problemas sobre a gestão de um projeto em grupo, é a coordenação em equipe e a falta de comunicação. Sendo uma solução para isso, o *Bug Tracker* consegue reunir todas as informações necessárias para que os membros possam se comunicar com clareza e sem problemas, auxiliando assim na construção de melhores projetos.

**Palavras-chave:** *Bug Tracker*. Projetos. Desenvolvedores.

## INTRODUÇÃO

Há tempos vêm sendo falado sobre a crise do *software*, termo cunhado na primeira conferência de engenheiros de *software* da OTAN (Organização do Tratado do Atlântico Norte), em 1968. Edsger Dijkstra, em 1972, referenciou o termo e escreveu sobre a: “Programação tornando-se um problema gigante”. Já em 1987, Barry Boehm relatou sobre a crescente demanda de *software*. Mais adiante, em 1996, Wayt Gibbs produziu um artigo intitulado: “*Software’s Chronic Crisis*”, onde o autor menciona a crise crônica do *software*. E por fim, Marc Andreessen, em 2011, publicou sobre o tema: “*Why Software is Eating the World*”, onde ele aponta que a demanda por *software* está superando cada vez mais a capacidade de produção.

Diminuir a demanda por *software* é claramente uma maneira inviável de resolver o problema, o que nos resta é a alternativa de expandir a capacidade de programadores para desenvolverem sistemas, em outras palavras, amplificar a produtividade da mão de obra.

Segundo a pesquisa “*Pulse of the Profession 2020*” (2020) da PMI (*Project Management Institute*), revela que, em média, 11,4% dos investimentos são desperdiçados devido ao baixo desempenho do projeto. E as organizações que subestimam a gestão de projetos como uma competência estratégica para a mudança de condução, relatam uma média de 67% a mais de projetos fracassados.

De acordo com o estudo “*Software Developers’ Perceptions of Productivity*” realizado com 379 desenvolvedores, conduzido pela Universidade de Zurique, localizada na Suíça, aponta que mais de 80% dos participantes, concordam de alguma forma que saber o número de tarefas (*bugs*, *features*) que eles finalizaram, os ajudam a avaliar sua produtividade pessoal.

Diante dos dados apresentados, percebe-se que a falta de gestão de projetos em uma empresa pode acarretar perda de investimentos e diminuir o desempenho das instituições.

Portanto, indaga-se: Com a alta necessidade da produção por *softwares*, como podemos implementar uma solução para melhorar a gestão das equipes e a qualidade final dos projetos desenvolvidos?

O objetivo deste trabalho é desenvolver um *Bug Tracker* funcional, para que as equipes de desenvolvedores possam se comunicar e realizar projetos com mais qualidade e facilidade. Para isso, deve-se: estabelecer uma boa comunicação entre

os desenvolvedores, aprimorar a qualidade de projetos, empregar uma excelente gestão de tarefas e de fácil acesso, aumentar a satisfação do cliente com o produto, evitar confusões na criação do projeto, e reestruturar a forma de criação entre os desenvolvedores.

No ritmo em que empresas surgem atualmente, o mercado fica cada vez mais competitivo e os consumidores mais exigentes. Neste cenário, a gestão de projetos melhora o desempenho da empresa, e considerando que todo projeto é único, caso ele seja bem executado, pode mudar o modo de como as pessoas veem sua empresa.

A gestão de projetos é fundamental para reduzir os riscos de falhas e controlar todas as etapas envolvidas, bem como garantir a qualidade dos resultados. Assim, é possível gerenciar projetos de forma eficiente, ou seja, atingindo os objetivos e otimizando recursos.

## REFERENCIAL TEÓRICO

O gerenciamento de tarefas é, em suma, uma maneira de identificar, monitorar e progredir as atividades do trabalho que precisam ser realizadas em uma empresa. O objetivo desse tipo de organização é aumentar a praticidade e, conseqüentemente, a produtividade e o desempenho da equipe de trabalho. Para isso, é preciso identificar e organizar cada uma delas de acordo com a sua prioridade. Isso é o que proporcionará a sensação de que você realmente fez o que deveria ter sido feito durante a jornada de trabalho.

“A comunicação é fundamental para a troca de experiências, informações e conhecimentos, principalmente no processo de desenvolvimento de software onde os atores envolvidos devem ocupar posição central, de modo que a interação entre eles proporcione o desenvolvimento satisfatório, atendendo aos requisitos e expectativas dos usuários”, dissertação publicada em 2008 por Francielle Venturini Dalla.

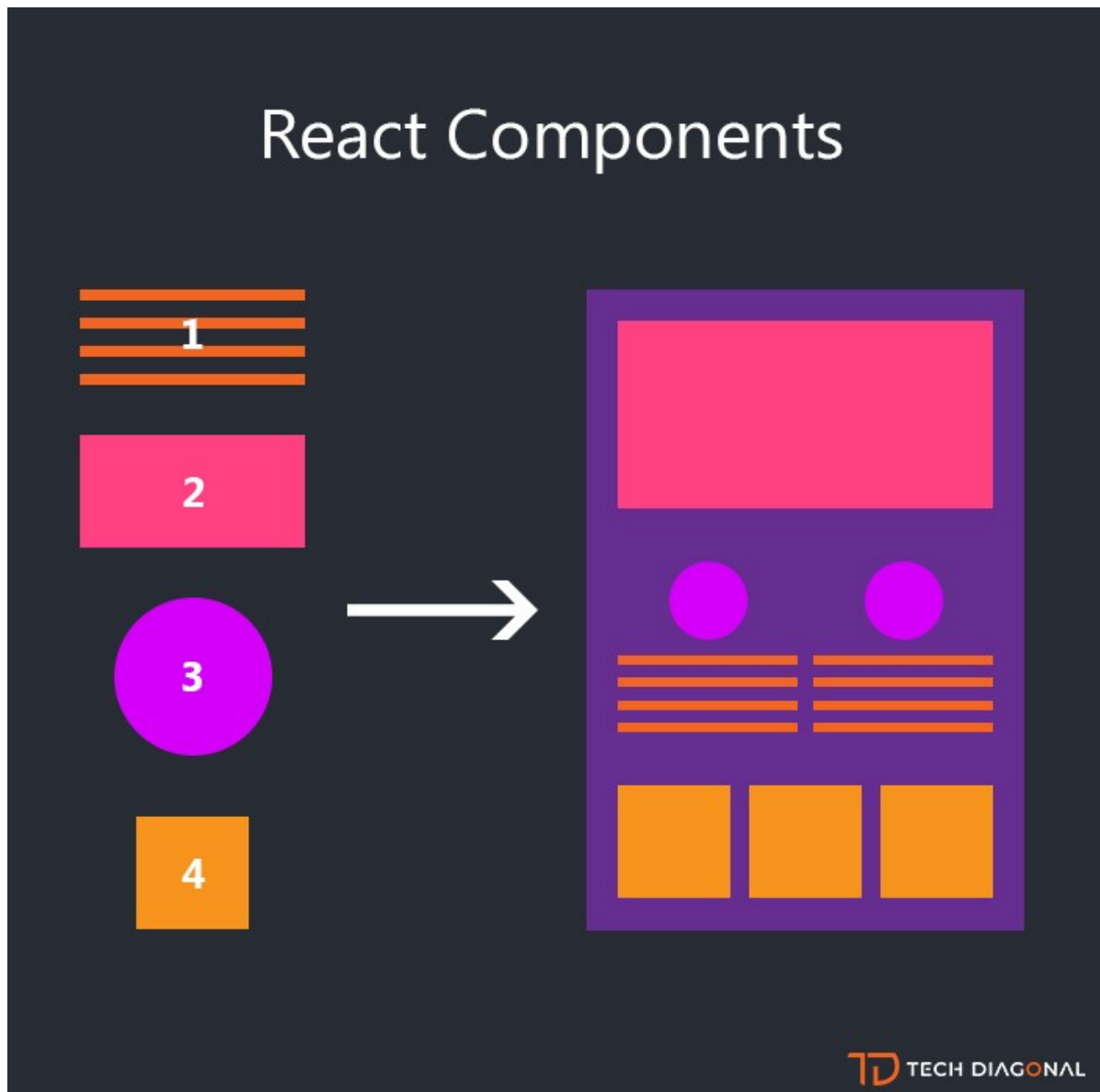
Com o crescimento irrefreável da demanda por software e o sucessivo aumento em sua complexidade, times de desenvolvimento cada vez mais buscam por novas maneiras de se organizar, visando o aumento da produtividade. Visto que, segundo a pesquisa "Pulse of the Profession 2020" (2020) da PMI (Project Management Institute), empresas que negligenciam práticas de gestão de projetos relatam uma média de 67% a mais de projetos falhando, e diante a isso, muitos times viraram seus olhares às ferramentas organizacionais e de gestão de projetos e pessoas.

## MÉTODO

### 1 DESENVOLVIMENTO DO SITE

O design do site foi criado utilizando a plataforma *Figma*, ferramenta para prototipação de sites e aplicativos. O site então foi desenvolvido com o *framework React*, possibilitando a geração de páginas dinâmicas com *JavaScript* e a separação dos elementos da interface de usuário em componentes reutilizáveis.

Figura 1 – Ilustração exemplificando o uso de componentes com *React*.



FONTE: <https://digitalspaces.dev/blog/becoming-a-react-programmer/>.

## 1.1 COLABORAÇÃO EM TEMPO REAL E VERSIONAMENTO DO CÓDIGO

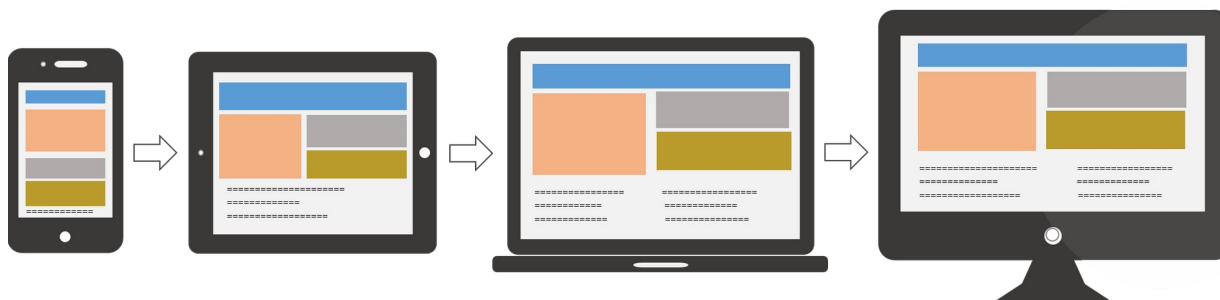
A extensão do *Visual Studio Code* chamada *Live Share*, possibilitou a criação de sessões de colaboração, onde todos da equipe colaboraram em tempo real na criação do código-fonte.

A ferramenta *Git* foi utilizada para versionar o *software*, e um repositório privado foi criado no *GitHub* para a hospedagem do código-fonte, possibilitando o compartilhamento do mesmo entre a equipe.

## 1.2 DESENVOLVIMENTO RESPONSIVO COM A DISCIPLINA MOBILE-FIRST

Para oferecer uma experiência responsiva, empregou-se a disciplina *mobile-first*. Toda a estilização (realizada em CSS) foi inicialmente criada para atender às necessidades de telas de dispositivos móveis, e em seguida, sendo adaptada para resoluções maiores.

Figura 2 – Ilustração demonstrando o processo de desenvolvimento com *mobile-first*.



FONTE: <https://usabilla.com/blog/wp-content/uploads/mobile-first.png>.

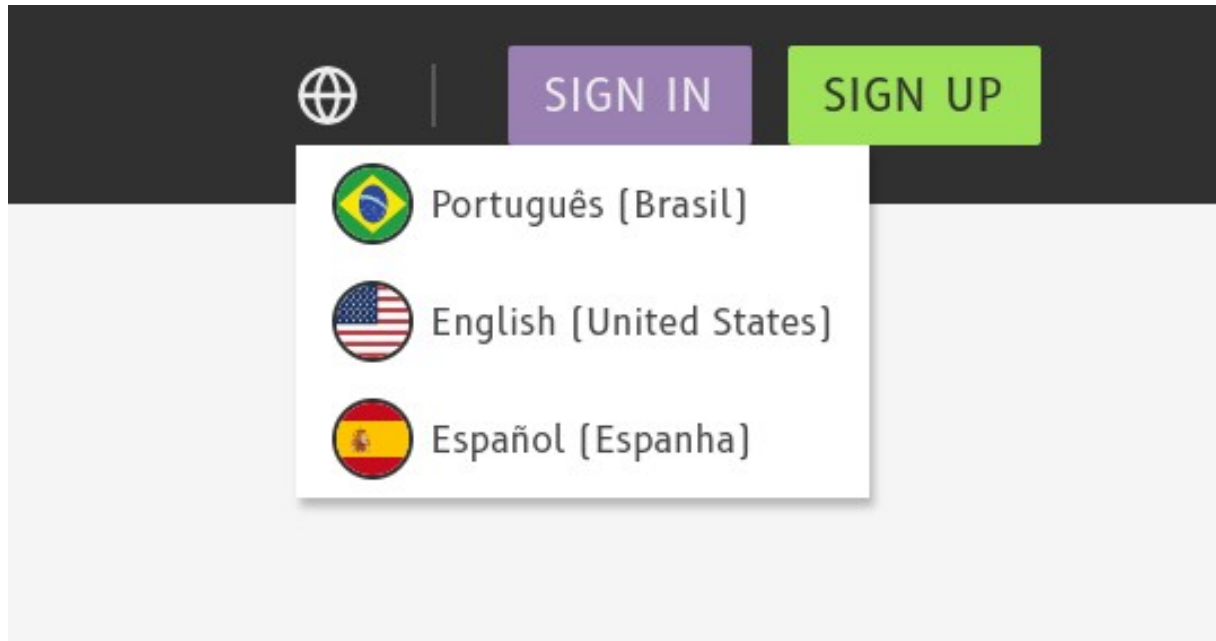
## 1.3 INTERNACIONALIZAÇÃO

A internacionalização é uma técnica utilizada em diversas áreas com o intuito de comercializar produtos e/ou serviços fora do seu mercado local. Em desenvolvimento de *software*, essa técnica se consiste, geralmente, em mostrar mensagens no idioma definido como padrão pelo ambiente do usuário (sistema



operacional; navegador), assim como dando a possibilidade de escolher dentre os idiomas disponíveis. Todo o site está internacionalizado, dando suporte aos idiomas português, inglês e espanhol.

Figura 3 – Exemplo de um menu de seleção de idioma.

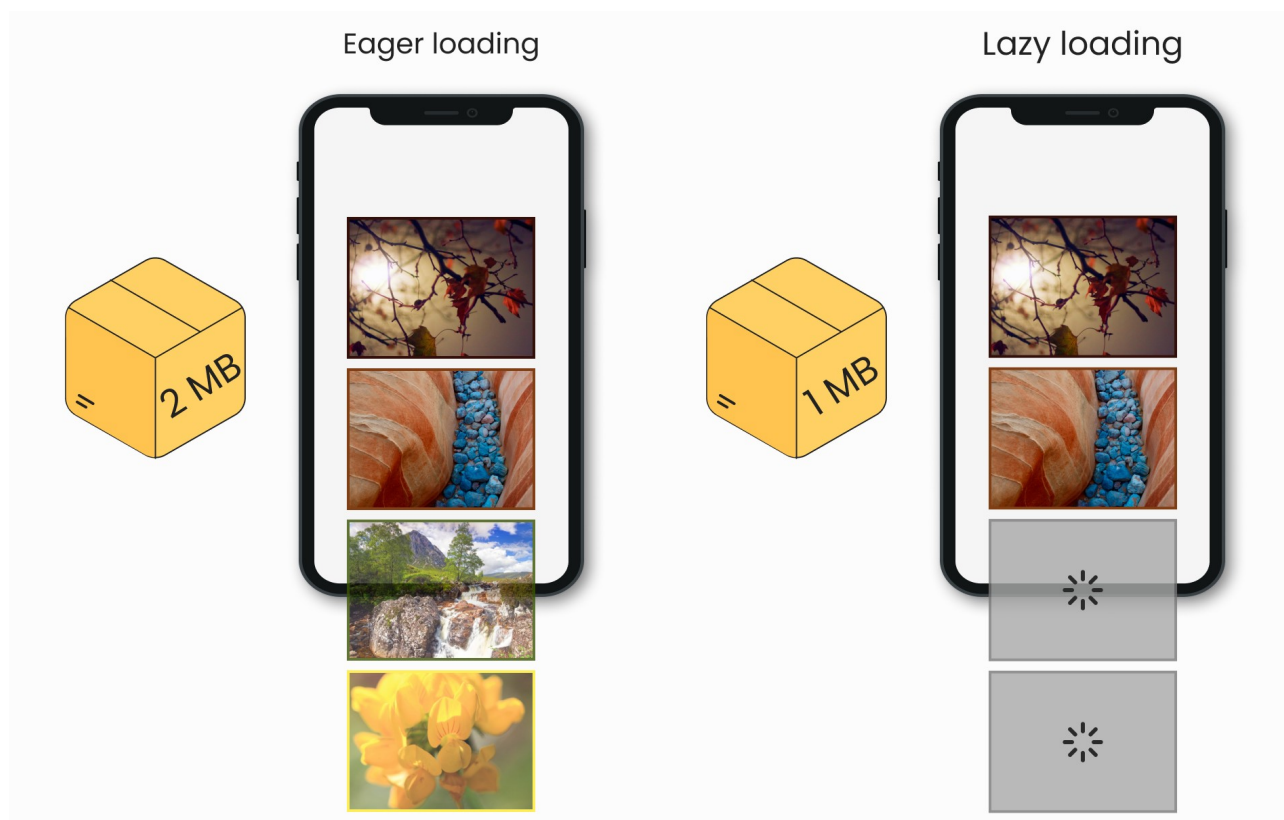


Fonte: Imagem feita pela equipe.

#### 1.4 LAZY LOADING

Para poupar recursos e manter o tempo de carregamento da página o mais rápido possível, foi utilizado a técnica de *Lazy loading*, que consiste em renderizar os elementos na página somente quando são necessários.

FIGURA 4 – Comparação entre *lazy loading* e outras técnicas de carregamento como o *eager loading*



FONTE: Elaborada pelos autores

## 2 DESENVOLVIMENTO DO SERVIDOR

### 2.1 NODE.JS

O *Node.js* foi utilizado na criação do servidor. Consiste em um *software* de código aberto, multiplataforma, que possibilita a execução de código *JavaScript* fora de um navegador da *web*. O *Node.js* permite que os desenvolvedores usem *JavaScript* para escrever ferramentas de linha de comando e *scripts* do lado do servidor, sendo uma das maiores vantagens trazidas pelo *Node.js*, a possibilidade de escrever um código *JavaScript* em todos os lugares, tanto no lado do cliente quanto do servidor, sem a necessidade de aprender outra linguagem de programação.

## 2.2 TYPESCRIPT

*TypeScript* é uma linguagem de programação, fortemente tipada, de código aberto desenvolvida pela *Microsoft*. De acordo com a pesquisa "*2020 Developer Survey*", foi considerada pelo público como a segunda linguagem mais amada, respondida por aproximadamente 65,000 desenvolvedores, conduzida pelo site *Stack Overflow* em 2020.

Tipos fornecem uma maneira de deixar explícito o tipo de uma variável ou descrever a forma de um objeto, fornecendo melhor a documentação e permitindo que o *TypeScript* valide se seu código está funcionando corretamente.

Figura 5 - Exemplo de um código em *TypeScript*.

A imagem mostra um editor de código com o título "TypeScript" e uma seta para baixo. O código é o seguinte:

```
type Person = {  
  name: string;  
  age: number;  
}  
  
const studentName = "Jorge"; // <- inferência de tipos  
  
const student: Person = {  
  name: studentName,  
  age: 16  
}
```

Fonte: Imagem criada pela equipe.

*TypeScript* é um conjunto de ferramentas desenvolvidas dentro do *JavaScript*. No momento da execução, todo código é transpilado para *JavaScript*, devido a isso, é possível ter na mesma base de código, arquivos *TypeScript* e *JavaScript*, o que possibilita a adoção gradual do *TypeScript*.

Figura 6 – Processo de transpilação de um código *TypeScript* para *JavaScript* ES5.

## TypeScript

```
class Duck {  
  name: string = 'Bill';  
  quack(): void {  
    // Quack!  
  }  
}
```



## ECMAScript 5

```
var Duck = (function () {  
  function Duck() {  
    this.name = 'Bill';  
  }  
  Duck.prototype.quack = function () {  
    // Quack!  
  };  
  return Duck;  
})();
```

Fonte: <https://leanpub.com/essentials/typescript/read>

A principal vantagem do *TypeScript* em relação ao *JavaScript* tradicional, é adicionar recursos importantes e úteis para a construção de projetos em larga escala, como tipagem estática, forte e inferida. Também oferece uma melhor integração com editores de textos e *IDEs* (Ambiente de Desenvolvimento Integrado), o que possibilita a descoberta de erros nos estágios iniciais do desenvolvimento. É possível instalá-lo utilizando um gerenciador de pacotes para *NodeJS*, sendo o NPM (*Node package manager*), o mais popular.

## 2.3 ARQUITETURA REST

REST (*Representational State Transfer*) é um conjunto de restrições arquiteturais baseado em princípios que descrevem como os recursos de rede são definidos e endereçados. Esses princípios foram descritos pela primeira vez no ano 2000 por Roy Fielding como parte de sua tese de doutorado, intitulada “*Architectural Styles and the Design of Network-based Software Architectures*”.

Quando um cliente faz uma solicitação a uma API RESTful (API está conforme as restrições da arquitetura REST), essa API transfere uma representação do estado do recurso ao solicitante. Essa informação (ou representação) é entregue via protocolo HTTP.

Para que uma API seja considerada do tipo RESTful, ela precisa estar em conformidade com os seguintes critérios:

- Ter uma arquitetura cliente/servidor formada por clientes, servidores e recursos a serem acessados via protocolo HTTP. As aplicações cliente e servidor devem ser distintas umas das outras, desacopladas.
- Estabelecer uma comunicação *stateless* entre cliente e servidor. Isso significa que toda requisição deve conter todas as informações necessárias para o seu processamento, já que nenhuma informação do cliente é armazenada (como *cookies*, *server-side sessions*) e toda as solicitações são separadas e isoladas.
- Armazenar dados em cache para otimizar as interações entre cliente e servidor.
- Ter um sistema em camadas que organiza os tipos de servidores (responsáveis pela segurança, pelo balanceamento de carga e assim por diante), envolvidos na recuperação das informações solicitadas em hierarquias que o cliente não pode ver.
- Ter uma interface uniforme entre os recursos para que as informações sejam transferidas em um formato padronizado. Toda requisição pelo mesmo recurso deve ser idêntica.

A arquitetura REST é composta de um conjunto de diretrizes que podem ser implementadas conforme necessário. Isso faz com que as APIs REST sejam mais rápidas, leves e escaláveis, o que é ideal para certos tipos de aplicações.

## 2.4 TESTES

O teste de software é uma maneira de avaliar a qualidade do programa e reduzir o risco de falhas, assegurando a qualidade do produto. "Teste de software é o processo de executar um programa com o intuito de encontrar erros." (MYERS, 1979, p. 5, tradução da equipe).

### 2.4.1 Testes automatizados

É possível utilizar ferramentas de automação de testes para escrever scripts com coleções de casos de teste. Uma das vantagens é a possibilidade de executar esses testes repetidas vezes e gerar relatórios exibindo porcentagem de cobertura

dos testes, arquivos com estruturas condicionais não executadas pelos testes, dentre outros detalhes.

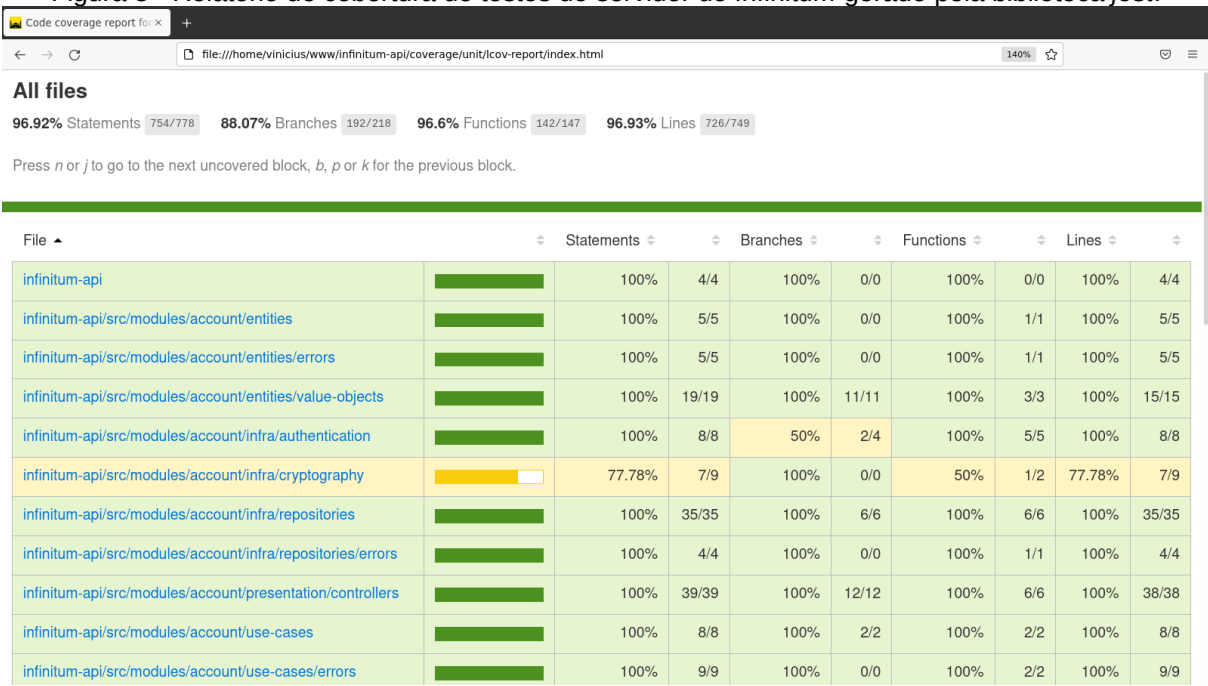
Várias linguagens possuem bibliotecas que facilitam a escrita de scripts de testes, como o *JUnit* para *Java* ou o *xUnit* para *C#*. Foi empregada a biblioteca *Jest* para *Node.js* neste trabalho.

Figura 7 – Arquivo com três casos de teste utilizando a biblioteca *jest* no *Node.js*.

```
somar.test.js x
home > vinicius > somar.test.js > ...
1 // ...função `somar` definida previamente ou em outro módulo...
2
3 test("Deve retornar 10 ao somar 5 com 5", () => {
4   const resultado = somar(5, 5);
5
6   expect(resultado).toBe(10);
7 });
8
9 test("Deve retornar uma mensagem de erro caso o primeiro parâmetro não seja um número", () => {
10  const resultado = somar("abcde", 9);
11
12  expect(resultado).toBe("Parâmetros precisam ser numéricos!");
13 });
14
15 test("Deve retornar uma mensagem de erro caso o segundo parâmetro não seja um número", () => {
16  const resultado = somar(23, "edcba");
17
18  expect(resultado).toBe("Parâmetros precisam ser numéricos!");
19 });
20
```

Fonte: Imagem feita pela equipe.

Figura 8 - Relatório de cobertura de testes do servidor do *Infinitem* gerado pela biblioteca *jest*.



Fonte: Imagem feita pela equipe.

## 2.4.2 Desenvolvimento guiado por testes

Durante o desenvolvimento do servidor, foi utilizado a disciplina chamada *Test-driven development* (TDD), ou desenvolvimento guiado por testes, em português, que consiste em escrever testes automatizados antes da implementação de uma função ou classe.

O teste de software é um processo de dupla checagem. Você diz o que deseja que uma função faça ao escrever um teste. Você diz de forma bem diferente quando implementa essa função. Se as duas expressões do algoritmo calculado pela função coincidirem, o código e o os testes estão em harmonia e provavelmente estão corretos. (BECK, 2004, tradução nossa).

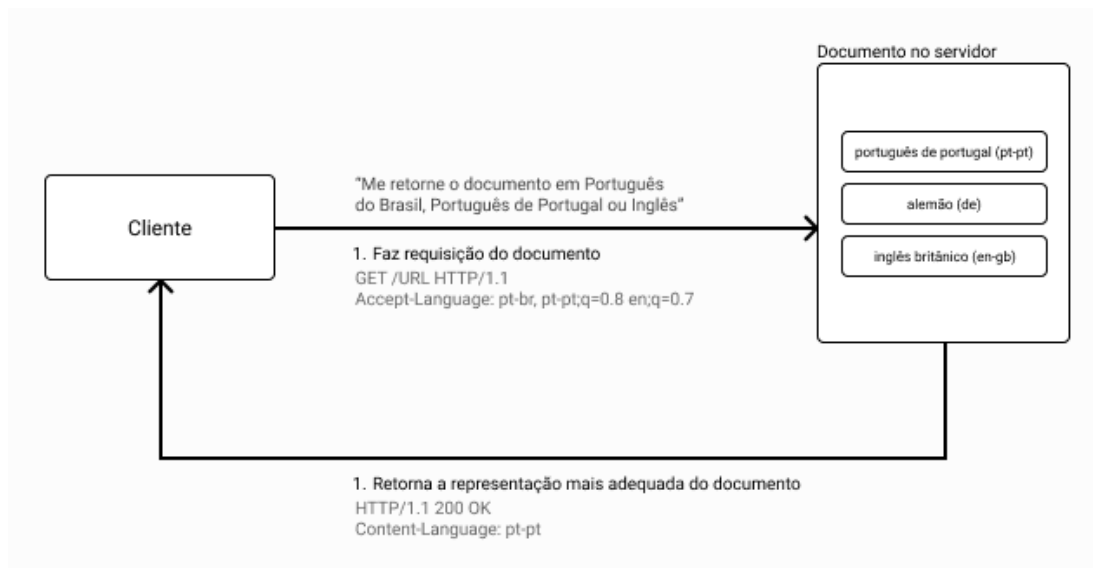
## 2.5 INTERNACIONALIZAÇÃO (I18N) DIRECIONADO AO SERVIDOR

A internacionalização é uma técnica utilizada em diversas áreas com o intuito de comercializar produtos e/ou serviços fora do seu mercado local. Em desenvolvimento de *software*, essa técnica se consiste, geralmente, em mostrar mensagens no idioma definido como padrão pelo ambiente do usuário (sistema operacional; navegador). Todo o servidor está internacionalizado, oferecendo as seguintes opções de idioma: inglês, português e espanhol.

### 2.5.1 Negociação de conteúdo no protocolo HTTP

Negociação de conteúdo é um mecanismo usado pelo protocolo HTTP para servir diferentes representações de um documento, para que o cliente possa pedir pela representação mais adequada (por exemplo, o mesmo documento pode ser servido em diferentes idiomas, diferentes encodificações de texto).

Figura 9 – Ilustração da negociação de conteúdo de idioma.



Fonte: Imagem feita pela equipe.

## 2.5.2 Cabeçalhos *Accept-Language* e *Content-Language*

### 2.5.2.1 *Requisição passando cabeçalho Accept-Language*

Ao fazer uma requisição HTTP, o cliente pode especificar o cabeçalho *Accept-Language* para indicar qual idioma é preferível pelo cliente. Este cabeçalho segue a seguinte sintaxe:

Figura 10 – Exemplo da escolha do idioma.

```
Accept-Language: idiomas [ peso ]
```

Fonte: imagem feita pela equipe.

Os idiomas especificados devem seguir a convenção internacional ISO 639, podendo ser definido também a localidade, seguindo a convenção internacional ISO 3166.



Figura 11 – Convenção internacional ISO 3166.

Código ISO 639	Idioma	Código ISO 3166	País
pt	Português	br	Brasil
da	Dinamarquês	us	Estados Unidos
en	Inglês	gb	Reino Unido

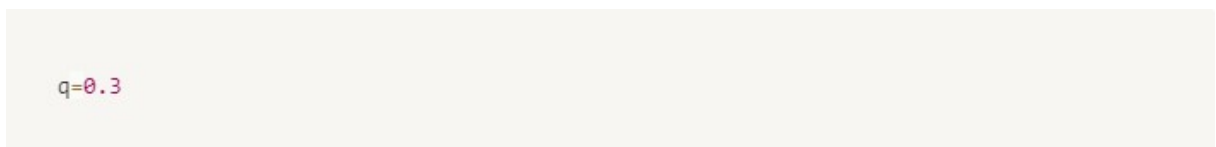
  

Código ISO 639 - ISO 3166	Idioma
pt-BR	Português do Brasil
en-US	Inglês (Estados Unidos)
en-GB	Inglês Britânico

Fonte: Imagem feita pela equipe.

Cada idioma pode ser associado a um valor de qualidade (peso) representando uma estimativa da preferência do usuário pelos idiomas especificados.

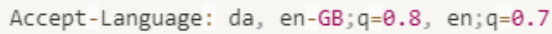
Figura 12 – Valor de qualidade.



Fonte: Imagem feita pela equipe.

Sendo 0.001 o menor valor de qualidade e 1, o maior.

Figura 13 – Exemplo de um cabeçalho Accept-Language válido.



```
Accept-Language: da, en-GB;q=0.8, en;q=0.7
```

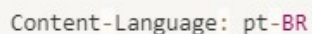
Fonte: Imagem feita pela própria equipe.

O que significaria dizer: "Eu prefiro Dinamarquês, mas aceitarei Inglês Britânico e, em último caso, qualquer outro tipo de Inglês".

#### 2.5.2.2 Resposta retornando cabeçalho Content-Language

Ao receber uma requisição HTTP com o cabeçalho Accept-Language especificado, o servidor deve então selecionar o documento que melhor se adequa ao idioma requerido e retornar o cabeçalho de resposta *Content-Language* informando o idioma utilizado.

Figura 14 – Cabeçalho Content-Language válido.



```
Content-Language: pt-BR
```

Fonte: Imagem feita pela própria equipe.

#### 2.5.3 Servidor não possui o documento em nenhum dos idiomas especificados

Caso os idiomas informados pelo cliente não são oferecidos pelo servidor, o cabeçalho *Accept-Language* pode ser desconsiderado, tratando a resposta como se não estivesse sujeita à negociação de conteúdo ou o código de erro HTTP 406 (Não

aceitável) pode ser enviado como resposta. No entanto, a segunda opção não é recomendada, pois pode acabar impedindo o usuário de acessar o conteúdo que eles ainda assim poderiam visualizar (utilizando um software de tradução, por exemplo).

## 2.6 LOCALIZAÇÃO (L10N)

Localização é o processo de adaptar um produto ou serviço para uso em outros países/culturas.

### 2.6.1 Salvando datas no lado do servidor

Devido aos diferentes fusos horários existentes é preciso dar uma atenção maior à forma como armazenamos datas, para que as informações sejam exibidas corretamente para pessoas em diferentes regiões.

#### 2.6.1.1 *Padrão internacional ISO-8601*

O padrão ISO-8601 define que datas com fuso horário devem ser escritas da seguinte forma:

Figura 15 – Data válida no formato ISO-8601 com fuso horário.

2021-08-30T22:49:09.739-03:00

↗  
Data

↗  
Horário

↑  
Fuso

Fonte: Imagem feita pela equipe.

#### 2.6.1.2 Salvando data sem fuso horário

Ao lidar com datas no lado do servidor, devemos salvá-las ao banco de dados sem um fuso horário atribuído a elas. No padrão ISO-8601, podemos fazer isso substituindo a parte do fuso horário pela letra "Z", o que corresponde ao Tempo Universal Coordenado (abreviado internacionalmente como UTC), do qual se tem como base para calcular todos os outros fusos horários.

Figura 16 - Data no formato ISO-8601 correspondente ao tempo universal coordenado (UTC).

2021-10-30T23:11:09.739Z

↗  
Tempo universal (sem fuso horário)

Fonte: Imagem feita pela equipe.

#### 2.6.1.3 Como o cliente deve lidar com a data retornada do servidor

Ao instanciar a classe *Date* do *JavaScript* no lado do cliente, teremos um objeto contendo informações sobre a data e hora, e com fuso horário definido pelo ambiente (navegador, localidade do sistema operacional etc.).

Se salvarmos uma data com fuso horário de Brasília (GMT-0300) no servidor, caso alguma pessoa em um fuso horário diferente acesse esse recurso do servidor, esta pessoa veria a data no fuso horário de Brasília, mesmo estando em outra região.

Como exemplo, um usuário no Brasil insere a seguinte data no servidor:

Figura 17 – Data no formato ISO-8601 com fuso horário de Brasília.

2021-10-30T23:11:09.739-03:00



Fuso horário do Brasil

Fonte: Imagem feita pela equipe.

Usuário na França, ao solicitar esse recurso no servidor, vê a mesma data com fuso horário de Brasília, quando deveria ver:

Figura 18 – Data no formato ISO-8601 com fuso horário da França.

2021-10-30T23:11:09.739+01:00



Fuso horário da França

Fonte: Imagem feita pela equipe.

## 2.8 SEGURANÇA

### 2.8.1 AUTENTICAÇÃO E AUTORIZAÇÃO COM *TOKEN JWT*

*JWT (Json Web Token)* é um padrão de indústria para realizar autenticação entre duas partes por meio de um *token*.

#### 2.8.1.1 A estrutura de um *token JWT*

Um *token JWT* consiste em uma *string* separada em três partes por pontos. O *token* toma o formato abaixo:

*xxxxx.yyyyyy.zzzzz*

As três partes são: *header*, *payload* e *assinatura*.

##### 2.8.1.1.1 Header

O *header* informa o tipo de *token* utilizado (*JWT*) e o algoritmo utilizado para encriptação da assinatura.

```
{  
  "alg": "RS256",
```

```
"typ": "JWT"
}
```

Esse *JSON* é então encodificado em *Base64* para ser utilizado como a primeira parte do *token*.

#### 2.8.1.1.2 Payload

O *payload* informa dados referentes à própria autenticação, aqui podemos colocar informações sobre o usuário autenticado.

```
{
  "id": "90b33cc6-d0de-45e0-b484-4dfbb4264fce"
}
```

Assim como o *header*, este *JSON* também é encodificado em *Base64* para ser utilizado como a segunda parte do *token*.

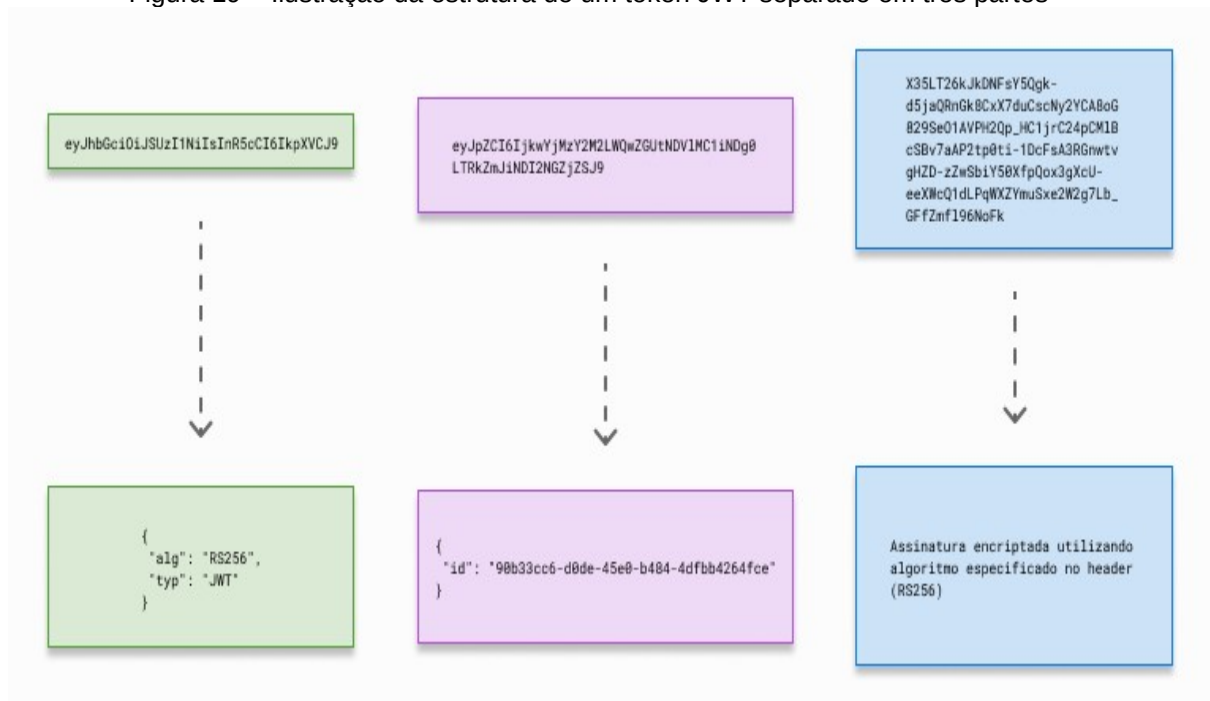
#### 2.8.1.1.3 Assinatura

É a assinatura única de cada *token* que é gerada a partir de um algoritmo de criptografia e tem seu corpo com base no *header*, no *payload* e na chave privada definida pela aplicação.

#### 2.8.1.2 Exemplo de um token JWT

```
eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6IjkwYjM2M2LWQwZGUtNDVI  
MC1iNDg0LTRkZmJiNDI2NGZjZSJ9.X35LT26kJkDNFsY5Qgk-  
d5jaQRnGk8CxX7duCscNy2YCA8oG829SeO1AVPH2Qp_HC1jrC24pCMIbCSBv7aA  
P2tp0ti-1DcFsA3RGnwtvgHZD-zZwSbiY50XfpQox3gXcU-  
eeXWcQ1dLPqWXZYmuSxe2W2g7Lb_GFfZmfl96NoFk
```

Figura 19 – Ilustração da estrutura de um token JWT separado em três partes



Fonte: Imagem feita pela equipe.

#### 2.8.1.2.1 Acessando recursos que necessitam de autorização

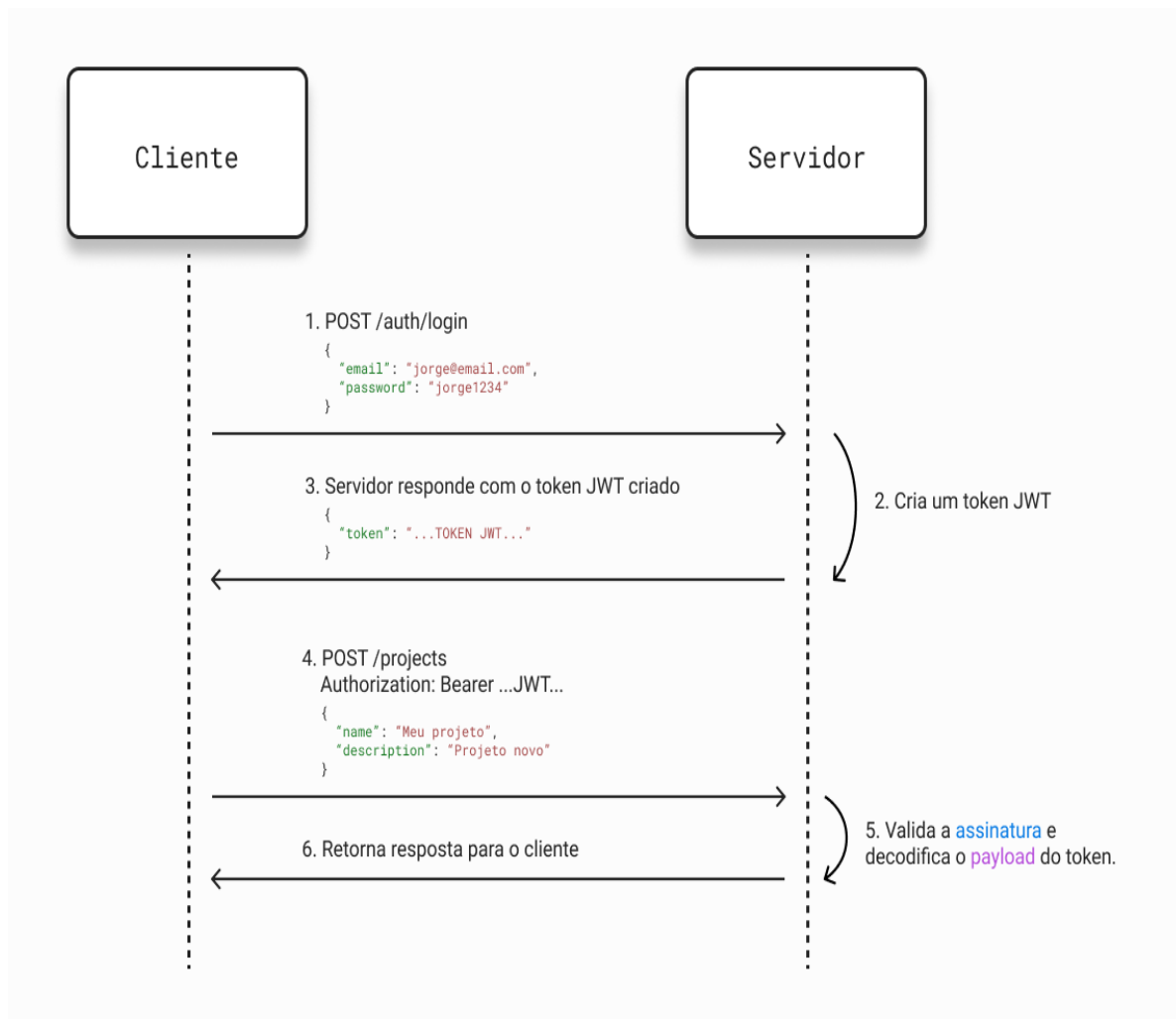
Para acessar recursos que necessitam de autorização, é necessário enviar na requisição *HTTP* o *header Authorization* contendo o *token JWT*. Caso o *token* seja inválido ou o *header* não esteja presente, o servidor responderá com o código de status 401 (Não autorizado).

#### 2.8.1.2.2 Como obter um token JWT

Para obter um *token JWT* deve ser feito uma requisição ao servidor informando *e-mail* e senha, caso os dados informados estejam corretos, o servidor retornará um objeto *JSON* contendo o *token*. O cliente então fica responsável em guardar esse *token* para enviar em futuras requisições.

Figura 20 – Ilustração de como funciona o processo de emissão de um token JWT para autorização do cliente





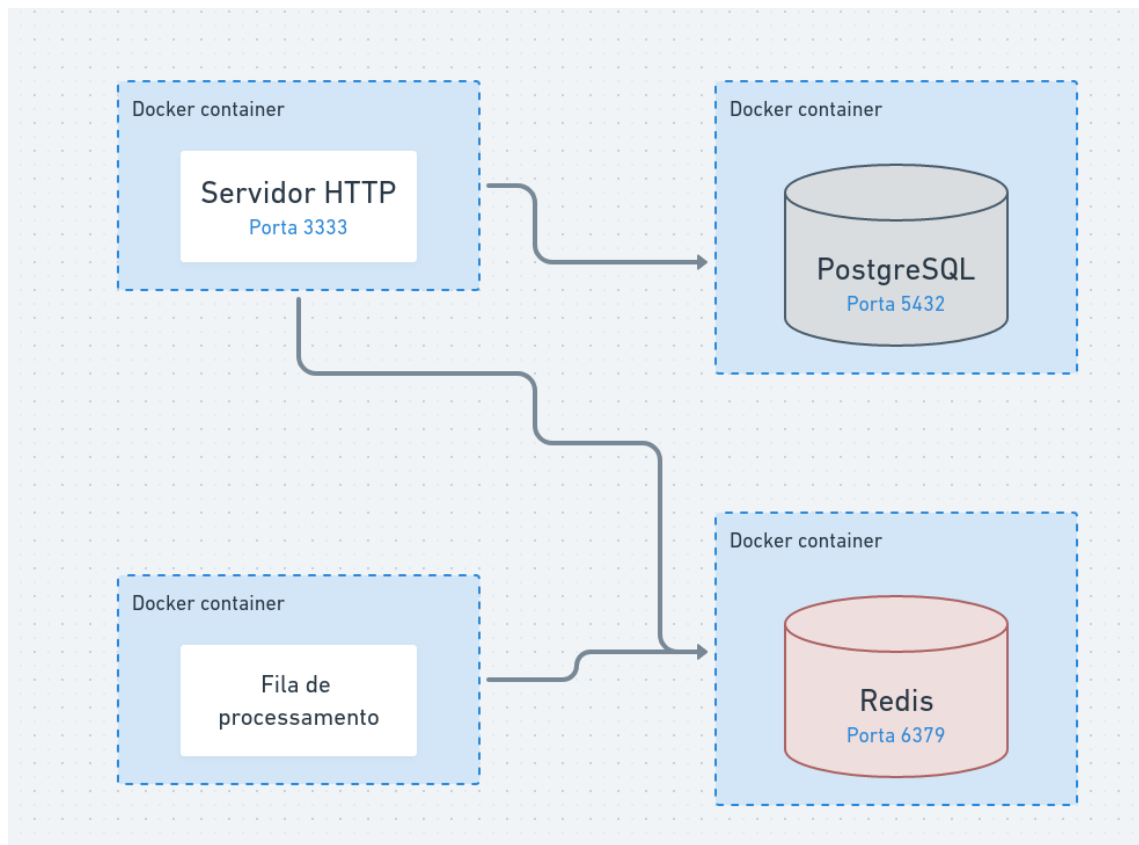
Fonte: Imagem feita pela equipe.

## 2.9 Containerização com Docker e Docker Compose

### 2.9.1 DOCKER

*Docker* é um sistema de virtualização que compartilha do Sistema Operacional da máquina para oferecer ambientes de execução isolados.

Figura 21 - Ilustração dos contêineres Docker utilizados no Infinitum.



Fonte: Imagem feita pela equipe.

### 2.9.2 Imagem Docker

Uma imagem *Docker* permite a execução de código dentro de um contêiner de forma declarativa que o *Docker* pode ler para subir um contêiner.

Figura 22 - Imagem *Docker* rodando o servidor do Infinitum

```
1 FROM node:14.17.5-alpine
2 WORKDIR /usr/app
3 COPY . .
4 RUN yarn
5 EXPOSE 3333
6 CMD ["yarn", "run:dev"]
```

Fonte: Imagem feita pela equipe

### 2.9.3 Docker Compose

*Docker Compose* é uma ferramenta de Infraestrutura como Código (abreviado em inglês como *IaC*) que permite a orquestração de contêineres para aplicações *multi-contêineres*.

## 2.7 ARQUITETURA

A arquitetura de *software* é, simplesmente, a organização de um sistema. Essa organização inclui todos os componentes, como eles interagem entre si, o ambiente em que operam e os princípios usados para projetar o *software*.

"A arquitetura de um software é a forma dada ao sistema por aqueles que o constroem. Essa forma é dada pela divisão do sistema em componentes, o arranjo desses componentes, e as maneiras pelas quais esses componentes se comunicam entre si." (MARTIN, 2017, p. 149, tradução pela equipe)

Arquiteturas de *software* buscam fornecer uma metodologia econômica que torne mais fácil desenvolver código de qualidade com menos dependências, e que torne mais fácil a manutenção do código.

Nas últimas décadas, muitas arquiteturas de *software* surgiram, e apesar de suas diferenças, grande parte dessas buscam um mesmo objetivo, que é o desacoplamento de regras de negócio vitais da aplicação de detalhes de

implementação. Cada uma dessas arquiteturas produz sistemas que têm as seguintes características:

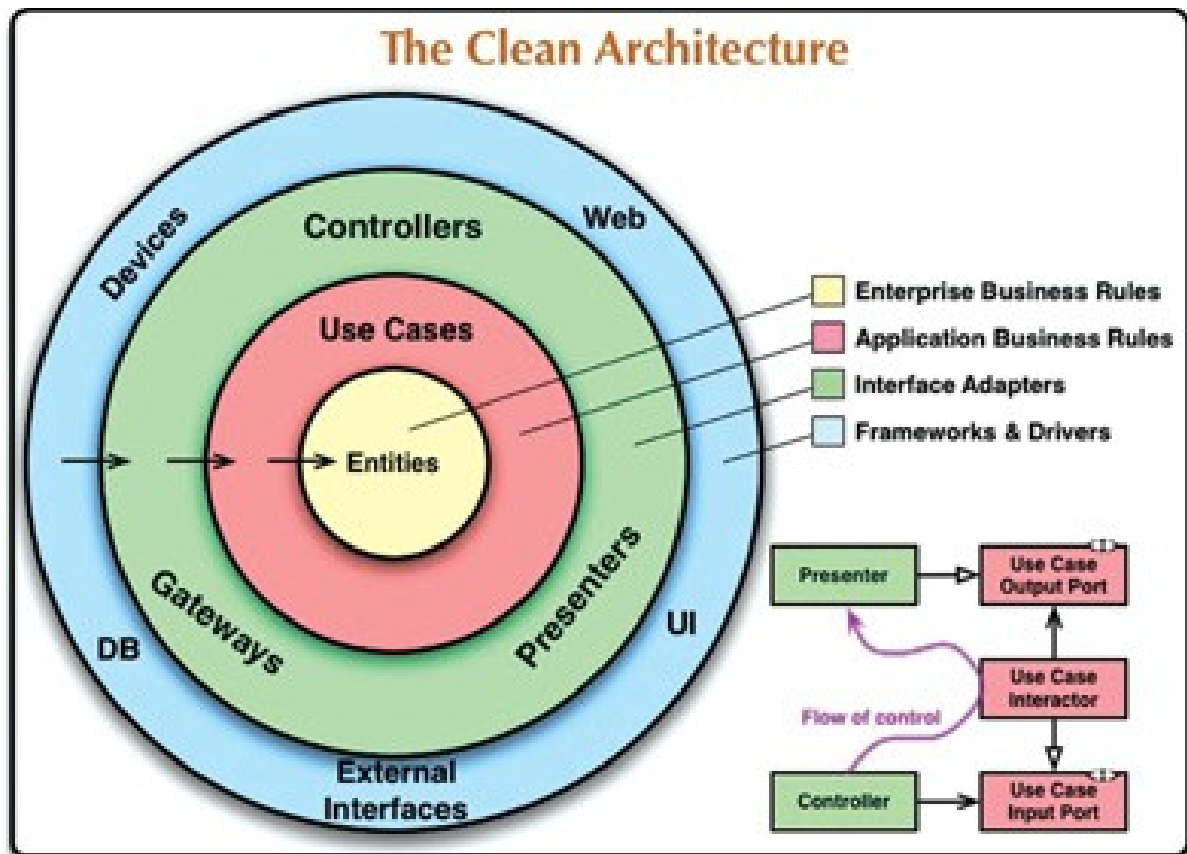
- Independente de bibliotecas externas: A arquitetura não depende da existência de alguma biblioteca cheia de funcionalidades. Isso permite que você use bibliotecas como ferramentas, ao invés de forçá-lo a fazer mudanças no seu sistema devido a suas restrições limitadas.
- Testável: As regras de negócio podem ser testadas sem a interface de usuário, banco de dados, servidor *web* ou qualquer outro elemento externo.
- Independente da interface de usuário: A interface de usuário pode ser modificada facilmente, sem afetar outras partes do sistema. Uma interface *web* pode ser substituída pelo terminal, por exemplo, sem afetar as regras de negócio.
- Independente do banco de dados: Você pode trocar o *MySQL* pelo *PostgreSQL*, um banco de dados relacional por um não-relacional. Um banco de dados pelo sistema de arquivos, salvando os dados em um arquivo *JSON*. As regras de negócio independem do banco de dados.

### 2.7.1 Clean Architecture

Arquitetura limpa é uma filosofia de *design* de *software* que separa os elementos de um sistema em camadas. Essa separação fornece aos desenvolvedores uma maneira de organizar o código de forma que encapsule a lógica de negócio e a mantenha isolada dos detalhes de implementação. Foi criada por Robert C. Martin e compartilha de similaridades com outras arquiteturas de *software*, como a Arquitetura Hexagonal, criada por Alistar Cockburn.

Não há um número exato de camadas a serem seguidas, porém o criador da arquitetura a planejou em quatro camadas: camada de entidades, de casos de uso, de apresentação e de infraestrutura. A imagem abaixo mostra a ordem das dependências, das camadas exteriores para as camadas interiores:

Imagem 23 - Imagem ilustrativa da separação de camadas na arquitetura limpa.

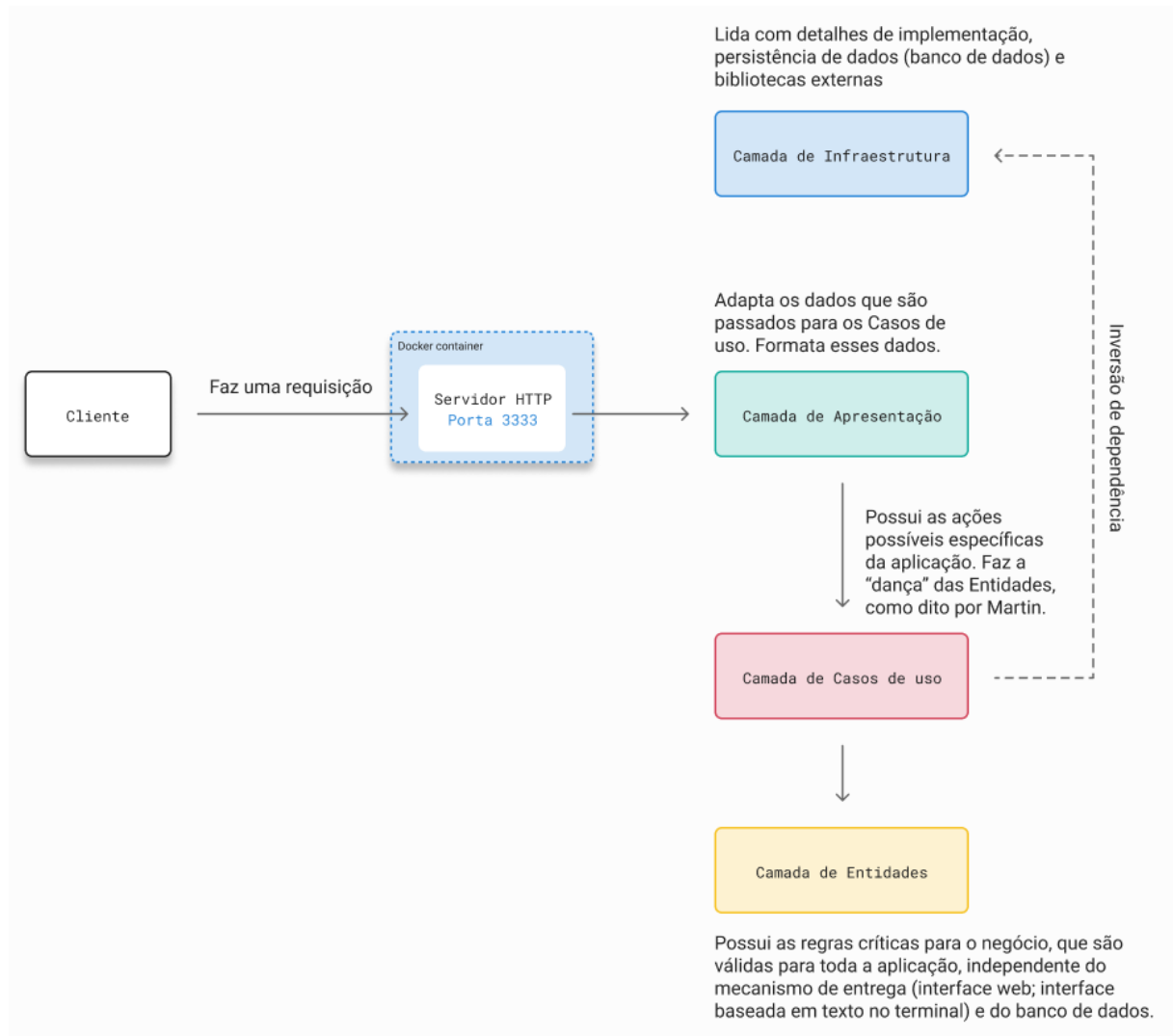


Fonte: Figura retirada do livro *Clean Architecture: A Craftsman's Guide to Software Structure and Design*, página 205.

Quanto mais para dentro do círculo olharmos, encontraremos regras de negócio vitais à nossa aplicação, que dificilmente mudarão ao passar do tempo. Já nos círculos exteriores, encontraremos *frameworks* e bibliotecas de terceiros, e partes do nosso sistema que mudam com frequência, por motivos não tão urgentes (como a interface de usuário, que muda a todo momento por questões estéticas).

A regra principal da arquitetura limpa, chamada de regra de dependência, é que as dependências do código só podem ser movidas das camadas externas para dentro. O código nas camadas internas não pode ter conhecimento das funções nas camadas externas. As variáveis, funções e classes (quaisquer entidades) que existem nas camadas externas não podem ser mencionadas nos níveis mais internos. Recomenda-se que os formatos de dados também fiquem separados entre as camadas.

Figura 24 - Direção do fluxo de dados na Arquitetura Limpa.



Fonte: Imagem feita pela equipe.

#### 2.7.1.1 Camada de entidades

As entidades encapsulam as Regras de Negócios Críticas para toda a empresa. Uma entidade pode ser um objeto com métodos ou um conjunto de estruturas de dados e funções. [...] elas têm menos probabilidade de mudar quando algo externo muda. Por exemplo, você não esperaria que esses objetos fossem afetados por uma mudança na navegação da página ou segurança. (MARTIN, 2017, p. 206, tradução da equipe)

As Regras de Negócios Críticas são regras vitais ao funcionamento do negócio, e existiriam mesmo que não houvesse um sistema para automatizá-las.

#### 2.7.1.1.1 Utilizando Controle de Acesso Baseado em Cargo (RBAC) para lidar com permissões dentro de Projetos

Para lidar com as permissões que o participante terá dentro de um projeto, foi utilizado o Controle de Acesso Baseado em Cargo (abreviado para RBAC, em inglês). Cada usuário possui um cargo associado a si para cada projeto do qual participa. Existem quatro cargos possíveis:

- Dono do projeto: Possui todas as permissões, somente um usuário pode possuir essa função, e será o usuário responsável por criar o projeto.
- Administrador: Possui permissão para administrar os *tickets* do projeto, assim como os membros.
- Membro: Possui permissão para administrar os *tickets* do projeto.
- Espectador: Possui permissão para visualizar o projeto e seus *tickets*, somente.

Ao realizar qualquer operação (criar *ticket*, atualizar *ticket*, adicionar membro etc.), o cargo do usuário no projeto é consultado, e então é verificado se aquele cargo possui a permissão necessária. Caso o usuário não possua permissão, a operação não terá efeito e um erro será retornado.

#### 2.7.1.2 Camada de casos de uso

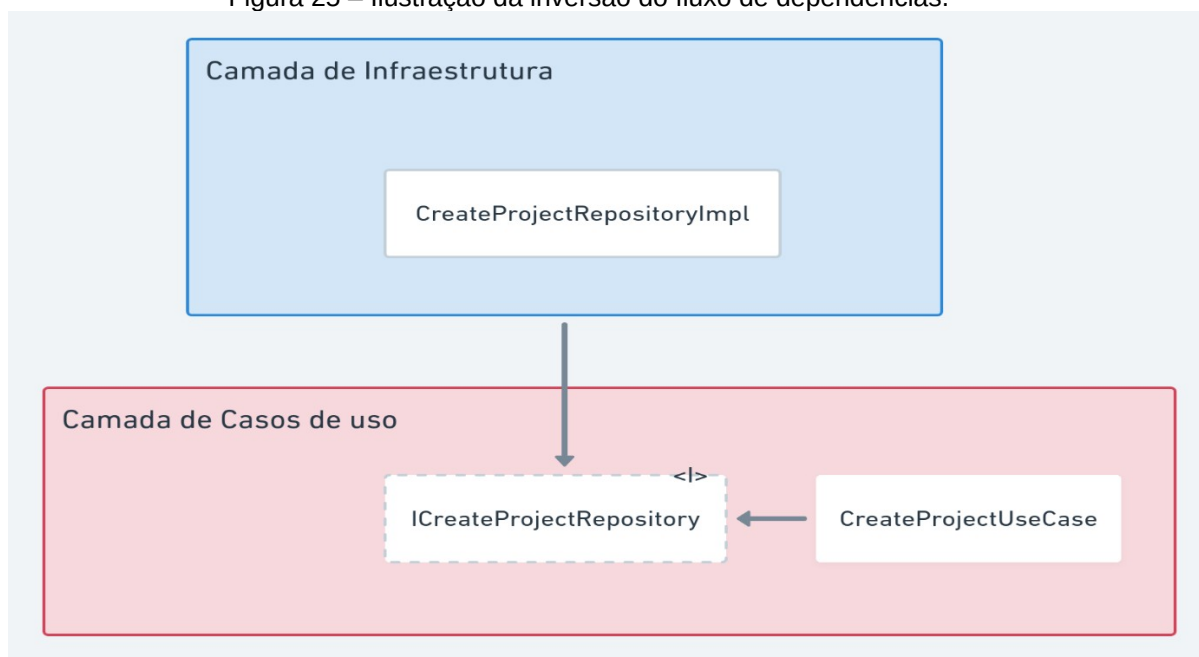
"Um caso de uso é a descrição da maneira como um sistema automatizado é usado. Ele especifica a entrada a ser fornecida pelo usuário, a saída a ser retornada ao usuário e as etapas de processamento envolvidas na produção dessa saída." (MARTIN, 2017, p. 196, tradução da equipe)

Não esperamos que mudanças nesta camada afetem as Entidades. Também não esperamos que essa camada seja afetada por mudanças nas externalidades, como o banco de dados ou a interface de usuário. A camada de casos de uso é isolada dessas questões. (MARTIN, 2017, p. 207, tradução da equipe)

#### 2.7.1.2.1 Inversão de dependência

Dado que os Casos de uso executam as ações do nosso sistema (criar um projeto, adicionar membro ao projeto, entre outros), é necessário utilizar um meio para persistir esses dados (banco de dados), porém esse é um detalhe de implementação que não cabe à essa camada, e que será implementado nas camadas exteriores, como na camada de infraestrutura. Como disse Martin (2017, p. 206, tradução da equipe), as dependências do código-fonte devem apontar apenas para dentro, em direção às camadas interiores. Portanto, podemos inverter a dependência à camada de infraestrutura ao definirmos uma interface e programarmos para essa interface.

Figura 25 – Ilustração da inversão do fluxo de dependências.



Fonte: Imagem criada pela equipe.

#### 2.7.1.3 Camada de apresentação

Segundo Martin (2017. p. 207, tradução da equipe), o *software* na camada de apresentação converte dados do formato mais conveniente para os casos de uso e entidades, para o formato mais conveniente para alguma agência externa, como o banco de dados ou a *web*.



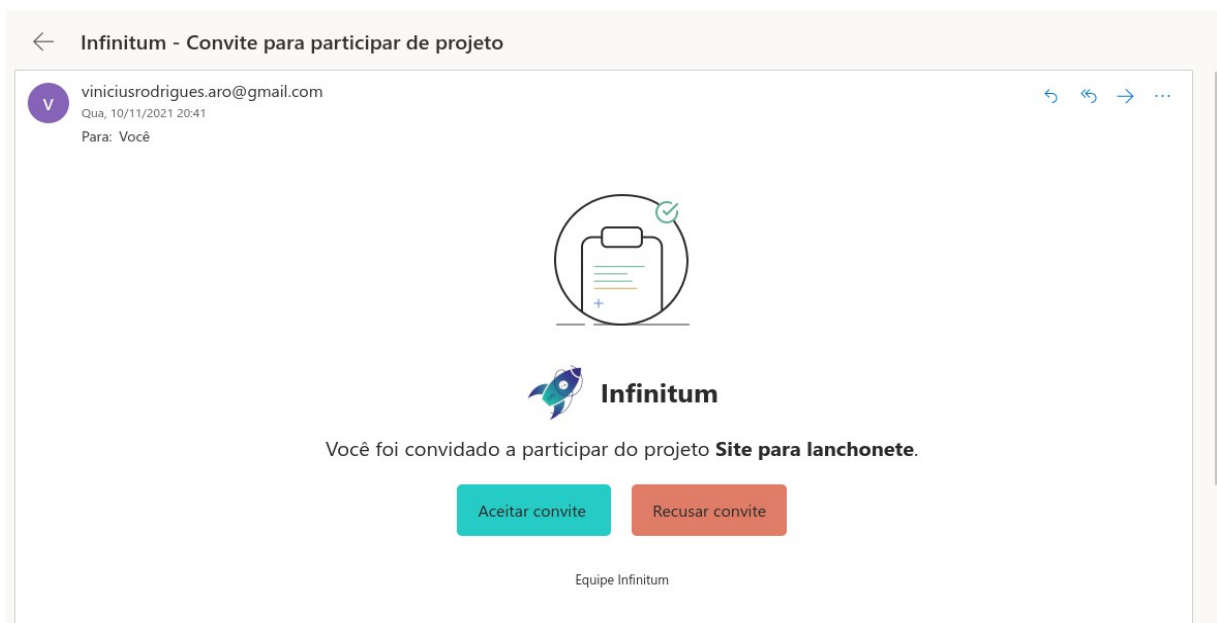
#### 2.7.1.4 Camada de infraestrutura

De acordo com Martin (2017, p. 207, tradução da equipe), a camada mais externa é geralmente composta de bibliotecas externas e ferramentas como o banco de dados e o *framework web*. Geralmente você não escreve muito código nesta camada, a não ser o código que organiza a comunicação com o próximo círculo interno.

##### 2.7.1.4.1 Enviando e-mails com a biblioteca *Nodemailer*

Para realizar o envio de e-mails foi utilizado a biblioteca chamada *Nodemailer*. É possível autenticar-se com uma conta do *Gmail* e enviar e-mails através dela.

Figura 26 - E-mail de convite para projeto.



Fonte: Imagem feita pela equipe.

Figura 27 - E-mail de aviso de expulsão.



Fonte: Imagem feita pela equipe.

#### 2.7.1.4.2 Fila de processamento de *background jobs* com a biblioteca *Bull*

Uma fila de processamento pode suavizar a carga de uma aplicação em um momento de pico de processamento. A biblioteca de filas escolhida foi a *Bull*, biblioteca rápida e robusta que utiliza de uma conexão com um banco de dados em memória *Redis*.

Embora seja possível implementar filas diretamente usando um banco de dados em memória como *Redis*, esta biblioteca fornece uma *API* que cuida de todos os detalhes de baixo nível e enriquece a funcionalidade básica do *Redis* para que casos de uso mais complexos possam ser tratados facilmente. (Documentação do *bull*, tradução da equipe.)

#### 2.7.1.4.3 Criando uma fila

Uma fila pode ser criada ao instanciar a classe *Bull*:

Figura 28 – Criação da fila ao instanciar a classe *Bull*

```
const emailDeliveryQueue = new Bull('email-delivery-queue');
```

Fonte: Imagem feita pela equipe.

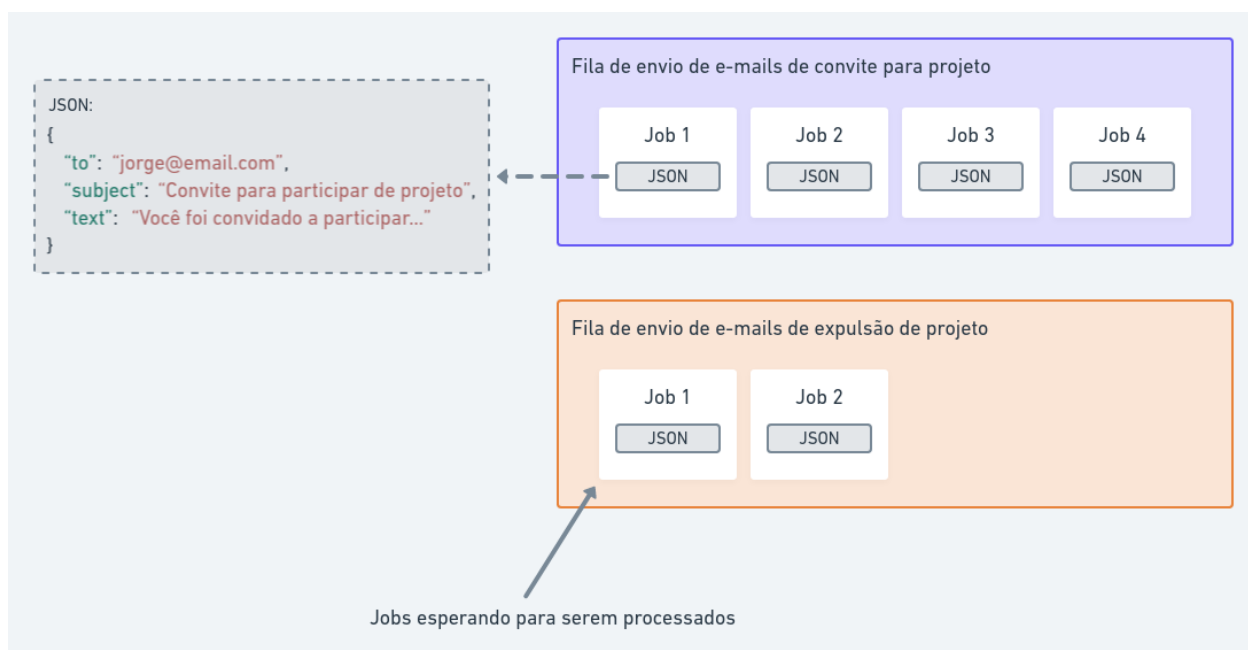
Uma instância de fila pode normalmente ter três funções principais diferentes: Um *job producer*, um *worker* e/ou um *event listener*.

Embora uma determinada instância possa ser usada para as 3 funções, normalmente o *producer* e o *worker* são divididos em várias instâncias. Uma determinada fila, sempre referida por seu nome de instanciação (*email-delivery-queue* no exemplo acima), pode ter muitos *producers*, *workers* e *listeners*.

#### 2.7.1.4.4 Jobs

*Jobs* são tarefas a serem processadas pela fila, como o envio de um e-mail, por exemplo. Ao adicionar *jobs* à uma fila é possível passar um objeto *JSON* com informações que podem ser utilizadas na hora do processamento do *job*, por um *worker*.

Figura 29 - Ilustração de diferentes filas populadas com *background jobs*.



Fonte: Imagem feita pela equipe.

#### 2.7.1.4.5 Producer

Um *producer* é um programa *Node.js* que adiciona trabalhos à uma fila, como o programa abaixo:

Figura 30 – Exemplo de um programa *Node.js* que faz o papel de um producer.

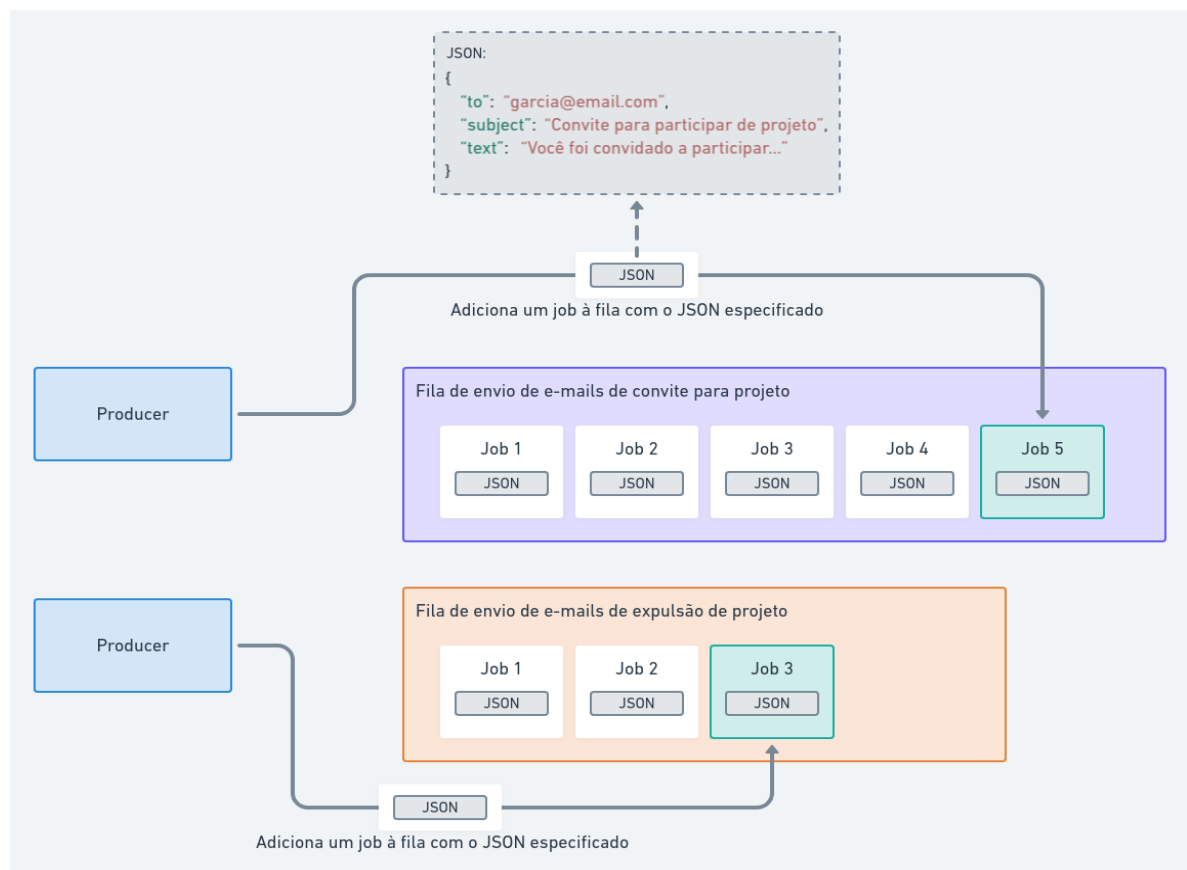
```
JavaScript ▾  
  
const emailDeliveryQueue = new Bull('email-delivery-queue');  
  
// adicionando job à fila  
const job = await emailDeliveryQueue.add({  
  to: 'jorge@email.com',  
  subject: "Convite para participar de projeto",  
  text: "Você foi convidado a participar..."  
});
```

Fonte: Imagem feita pela equipe.

Um *job* é apenas um objeto *JavaScript*. A função que irá processar esse *job* terá acesso a esse objeto.

Veja abaixo uma ilustração de como funciona o processo de adição de um *job* à fila, a partir pelo *producer*.

Figura 31 - Ilustração do papel de um job producer na biblioteca Bull.

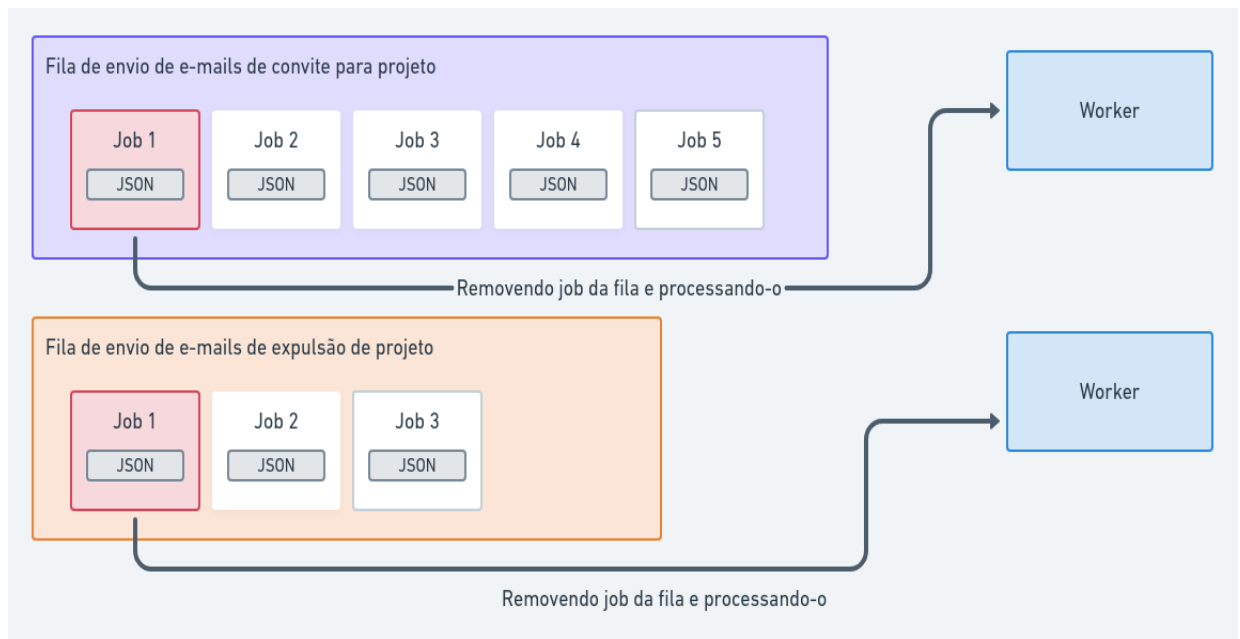


Fonte: Imagem feita pela equipe.

#### 2.7.1.4.6 Worker

Um *worker* (ou *consumer*) é quem processará os *jobs* existentes na fila, como ilustra a imagem abaixo:

Figura 32 - Ilustração geral de como funciona a biblioteca *Bull*.



Fonte: Imagem feita pela equipe.

Um *worker* é um programa *Node.js* que define a função que processará os *jobs*, como o programa abaixo:

Figura 33 – Ilustração da função que processará o jobs.

```
JavaScript ▾  
  
const emailDeliveryQueue = new Bull('email-delivery-queue');  
  
// declarando uma função que processará os jobs desta fila  
emailDeliveryQueue.process((job) => {  
  console.log(`Processando job ${job.id}`);  
  return;  
});
```

Fonte: Imagem feita pela equipe.

A propriedade *job.data* possui o objeto *JavaScript* especificado pelo *producer* ao adicionar o *job*.

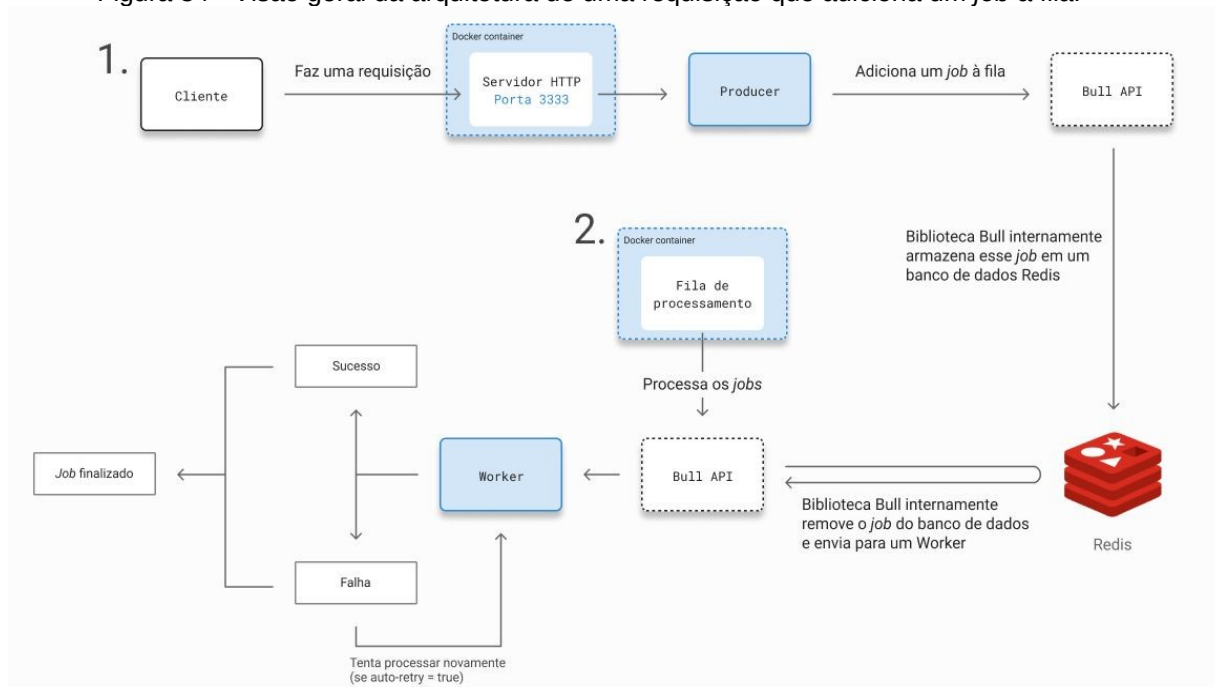
*Workers* podem ser executados no mesmo ou em diferentes processos. O *Redis* atuará como um ponto comum e, desde que um *worker* ou *producer* possa se conectar ao *Redis*, eles poderão cooperar no processamento dos *jobs*.

#### 2.7.1.4.7 Visão geral da arquitetura de uma requisição que adiciona um job à fila

Ao fazer uma requisição ao servidor que delega processamento à uma fila de *jobs* (como quando é necessário o envio de algum e-mail), o processo ocorre em duas etapas:

1. O cliente faz uma requisição ao servidor *HTTP*, que por sua vez adiciona um *job* à fila através da interface da biblioteca *Bull*, que internamente armazena um registro em um banco de dados *Redis*.
2. Rodando em outro container *Docker*, está a aplicação que processará a fila, novamente chamando métodos definidos pela interface da biblioteca *Bull*, que internamente removerá o registro do banco de dados e enviará o *job* para que um *Worker* o processe.

Figura 34 - Visão geral da arquitetura de uma requisição que adiciona um *job* à fila.



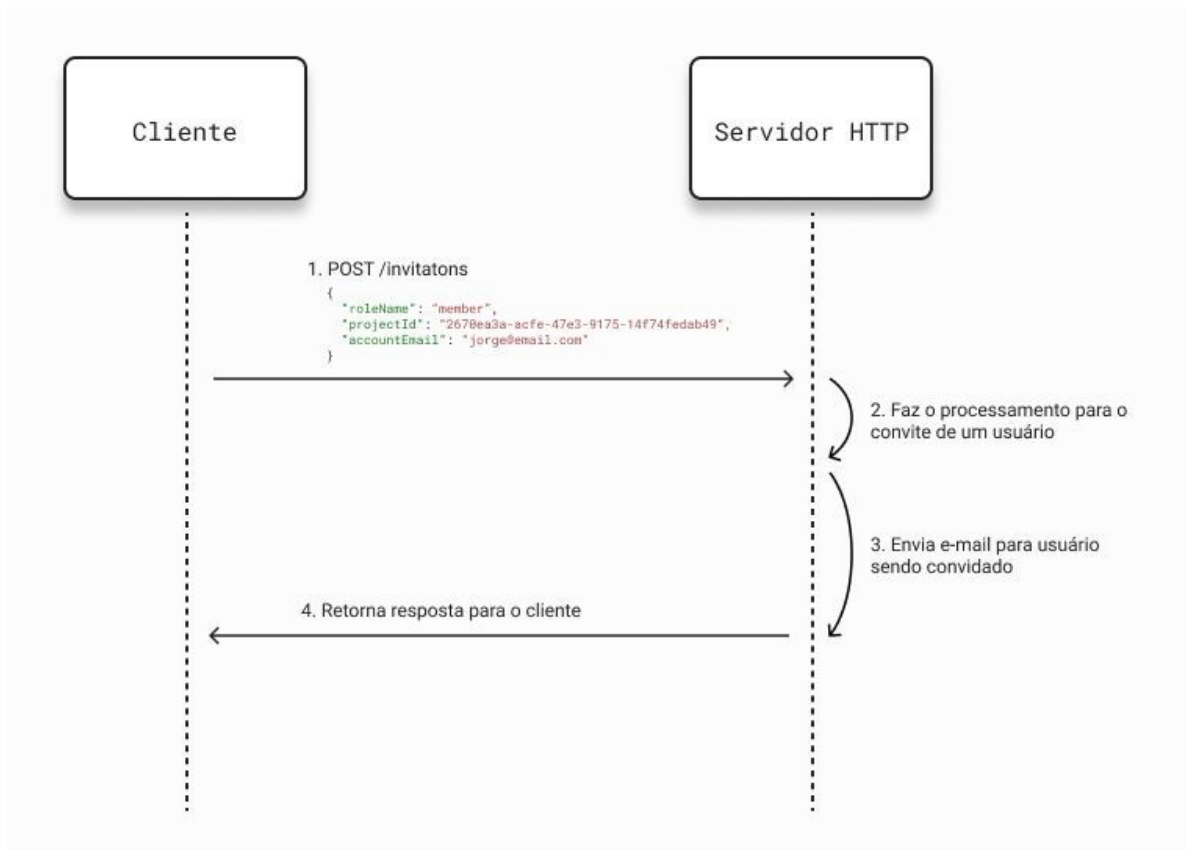
Fonte: Imagem feita pela equipe.

#### 2.7.1.4.8 Otimização: enviando e-mails utilizando a fila de processamentos

Caso o envio de e-mail fosse feito de forma assíncrona, seria necessário esperar o tempo de envio do e-mail (o que pode levar um tempo considerável), para

retornar uma resposta ao cliente, o que pode acarretar uma experiência ruim para o cliente consumidor da *API*.

Figura 35 - Ilustração da relação cliente/servidor tradicional, de forma assíncrona.



Fonte: Imagem feita pela equipe.

Para diminuir o tempo de resposta de requisições que fazem o envio de e-mail, a função responsável por enviar o e-mail é colocada em fila e delegada para ser executada por outro processo, dado que um administrador do projeto não precisa esperar o tempo de envio de um e-mail de convite (ou expulsão) para saber que a operação ocorreu com sucesso.

Figura 36 - Ilustração da relação cliente/servidor delegando processamento não urgente de imediato (envio de e-mail) para uma fila de processamento.





Para conectar-se ao banco de dados pelo servidor, foi utilizado a biblioteca *Knex.js*, que permite a conexão com diversos bancos de dados SQL, também oferece uma interface para criação de consultas SQL ao encadear métodos, como pode-se observar abaixo:

Figura 38 - Exemplo de uma consulta SQL utilizando o *query builder Knex.js*.

JavaScript ▾

```
const issues = await connection("issue")
  .leftJoin("account", "issue.assigned_to_account_id", "=", "account.id")
  .select(
    "issue.id as issue_id",
    "issue.title as issue_title",
    "issue.description as issue_description",
    "issue.completed as issue_completed",
    "issue.created_at as issue_created_at",
    "issue.expires_at as issue_expires_at",
    "account.email as account_email"
  )
  .where({ issue_group_id: "3717b34b-a6cf-4030-9b5c-6f0004da88cb" })
  .orderBy("issue.created_at");
```

Fonte: Código retirado e modificado da base de código do *Infinitum*.

Por oferecer uma abstração em cima de consultas SQL "cruas", o *Knex.js*, internamente, protege contra os ataques do tipo *SQL Injection*.

#### 2.7.1.4.10 Migrations

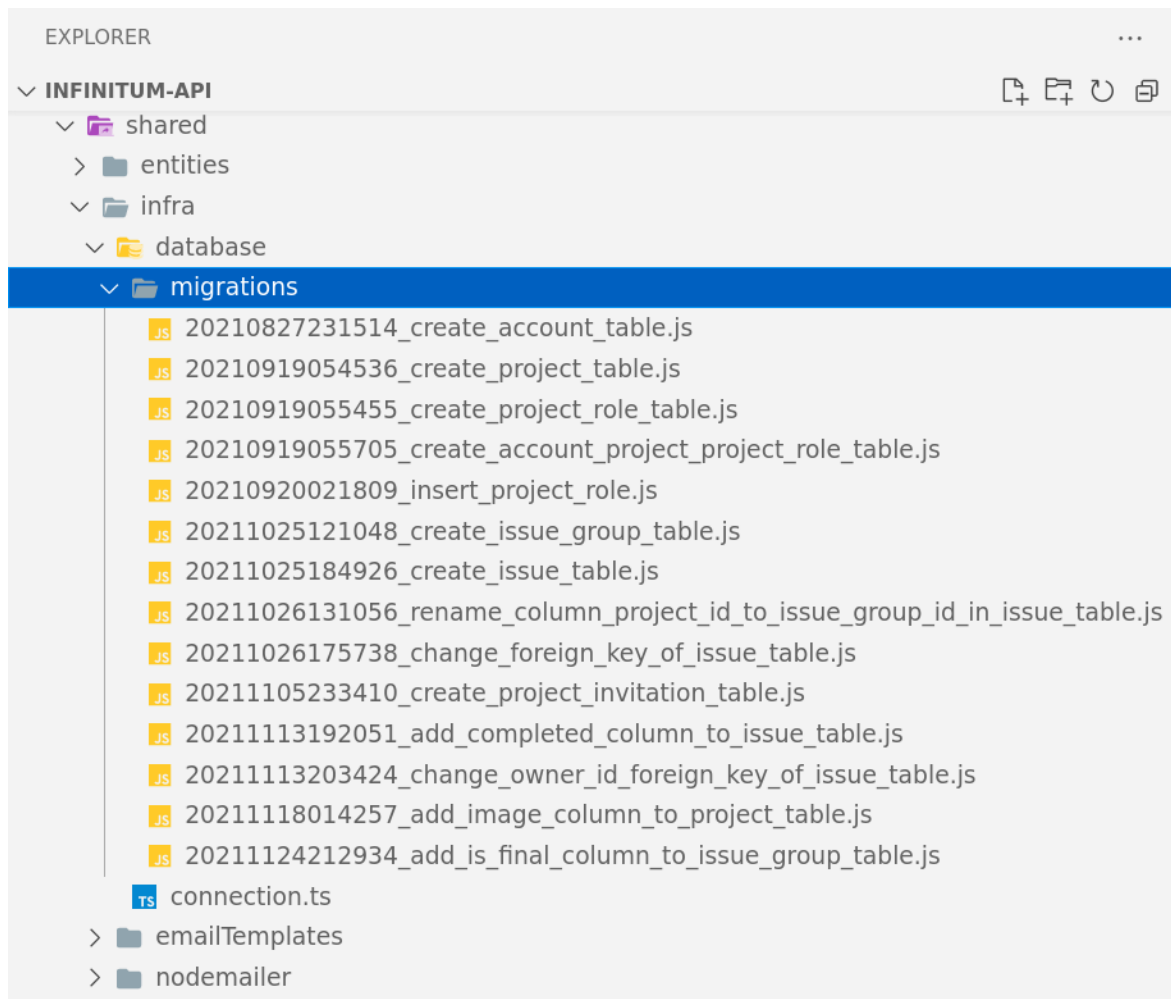
A estratégia de design chamada de *migrations* consiste, basicamente, em escrever *scripts* para fazer alterações ao banco de dados e mantê-los salvos sequencialmente na base de código, para que qualquer pessoa possa configurar o banco de dados em sua máquina no mesmo estágio que a última versão do desenvolvimento. Foi feita a utilização dessa estratégia para manter as mudanças ao banco de dados sincronizadas entre os membros da equipe, permitindo a cada participante configurá-lo em sua máquina com a simples execução de *scripts*.

*Pramod Sadalage* da empresa *ThoughtWorks* criou uma estratégia de design de banco de dados incremental elegante:

- Comece com um banco de dados vazio.
- Adicione todas as tabelas e colunas com *scripts* automatizados que também migram quaisquer dados existentes conforme necessário.
- Numere sequencialmente os *scripts* para que um banco de dados em qualquer estágio anterior possa ser levado para qualquer estágio posterior, executando os *scripts*.

Uma versão específica do código espera uma versão específica do *design* do banco de dados. A implantação de uma nova versão do sistema envolve a implementação do novo código e a execução de quaisquer *scripts* de *design* / migração. (BECK, 2004, tradução da equipe).

Figura 39 - Pasta de *migrations* com arquivos adicionando tabelas/colunas que formam o estágio final do banco de dados relacional utilizado no *Infinitum*, os arquivos estão sequenciados por data de criação.



Fonte: Imagem retirada da funcionalidade de *file explorer* da *IDE Visual Studio Code*.

#### 2.7.1.11 Banco de dados em memória

*Redis (Remote Dictionary Server)* é um banco de dados de código-fonte aberto, em memória, que segue o paradigma de armazenamento de dados chave-valor. Foi utilizado para o processamento da fila de envio de e-mails.

Figura 40 - Ilustração do paradigma de armazenamento de dados chave-valor.



Fonte: Imagem feita pela equipe.

Por ser um banco de dados em mem ria, as opera  es de leitura e escrita s o muito mais r pidas se comparadas ao armazenamento em HDD ou SSD.

No *Infinitem*, o banco *Redis* est  rodando em um container *Docker*, orquestrado pelo *Docker Compose*.

#### 2.7.1.5 Camada principal

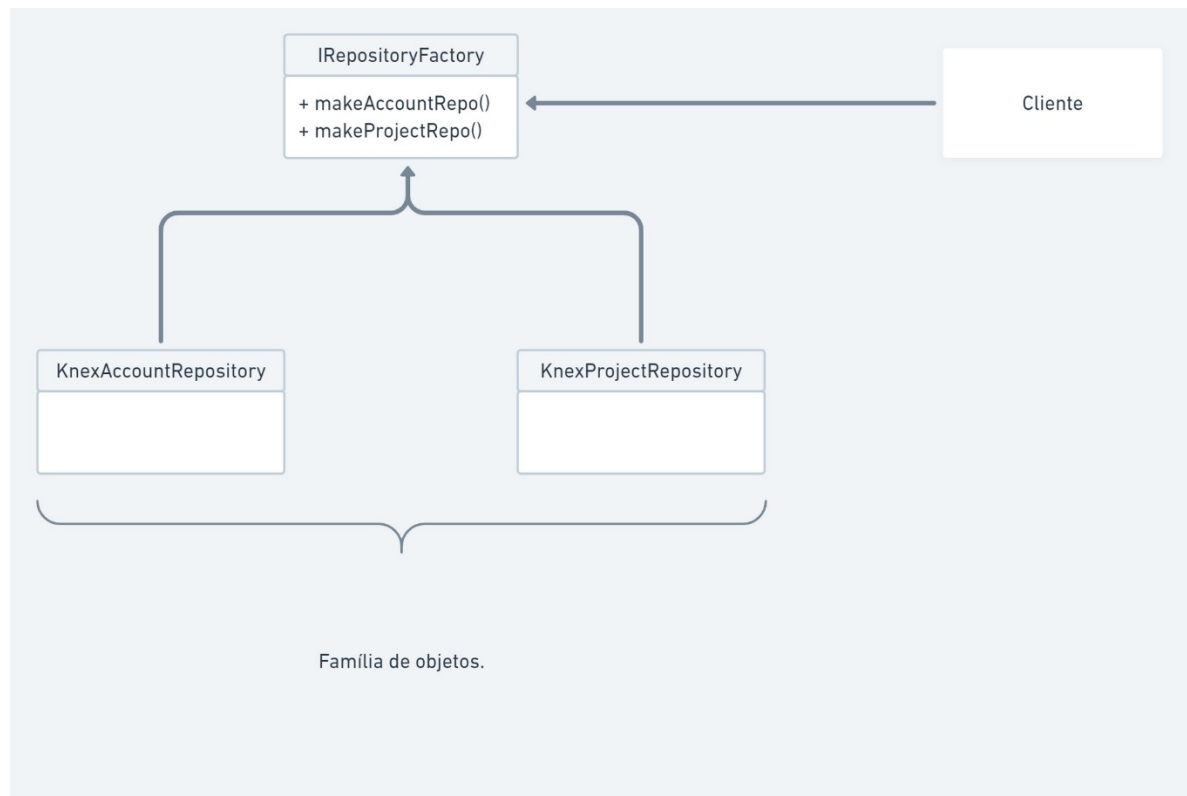
Dado que definimos diferentes interfaces na camada de casos de uso, precisamos injetar implementa  es concretas para compor esses objetos. A camada principal   descrita por Martin como a camada "mais suja" da aplica  o, por depender de todas as outras camadas.

Como a camada principal n o possui nenhuma l gica ou regra de neg cio, mas sim instancia  o de classes concretas e composi  o de objetos, ela n o   citada junto com as 4 camadas descritas por Martin.

##### 2.7.1.5.1 Instanciando fam lias de objetos com o padr o de projeto Abstract Factory

O padr o de projeto *Factory* define uma classe respons vel pela instancia  o de um objeto, ao adicionar uma interface em cima disso, temos o projeto *Abstract Factory*, que permite a instancia  o de fam lias de objetos relacionados.

Figura 41 - Ilustração de como funciona uma Abstract Factory. Imagem feita pela própria equipe.



Fonte: Imagem feita pela equipe.

## REFERÊNCIAS

<https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/meyer-fse-2014.pdf>

<https://www.pmi.org/learning/library/forging-future-focused-culture-11908>

<https://repositorio.ufsm.br/handle/1/4532>

<https://proj4.me/blog/gestao-de-tarefas>

<https://www.asplan.com.br/entenda-a-importancia-da-gestao-de-tarefas-na-contabilidade/#:~:text=O%20gerenciamento%20de%20tarefas%20%C3%A9%2C%20basicamente%2C%20uma%20maneira,produtividade%20e%20o%20desempenho%20da%20equipe%20de%20trabalho>

<https://en.wikipedia.org/wiki/Node.js>

<https://nodejs.org/en/>

<https://www.redhat.com/pt-br/topics/api/what-is-a-rest-api>

<https://www.ibm.com/cloud/learn/rest-apis>

<https://restfulapi.net/statelessness/>

<https://www.service-architecture.com/articles/web-services/representational-state-transfer-rest.html>

<https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>

<https://www.typescriptlang.org/><https://leanpub.com/essentials/typescript/readg.org/>

<https://insights.stackoverflow.com/survey/2020#technology-most-loved-dreaded-and-wanted-languages-loved>

<https://leanpub.com/essentials/typescript/read>

<https://pt.wikipedia.org/wiki/TypeScript>

<https://www.docker.com/>

[https://pt.wikipedia.org/wiki/Docker\\_\(software\)](https://pt.wikipedia.org/wiki/Docker_(software))

<https://searchitoperations.techtarget.com/definition/Docker-image>

BECK, Kent. Extreme Programming Explained: Embrace Change. 2ª edição. Estados Unidos da América: Addison-Wesley, 2004.

<https://www.postgresql.org/>

<https://www.statista.com/statistics/809750/worldwide-popularity-ranking-database-management-systems/>

<https://blog.nodeswat.com/implement-access-control-in-node-js-8567e7b484d1>

<https://redis.io/>

<https://github.com/OptimalBits/bull/tree/develop/docs>

<https://knexjs.org/>

<https://www.intel.com.br/content/www/br/pt/analytics/in-memory-database.html>

<https://web.archive.org/web/20150905081110/http://www.objectmentor.com/resources/articles/isp.pdf>

<https://whatis.techtarget.com/definition/clean-architecture>

<https://www.castsoftware.com/glossary/what-is-software-architecture-tools-design-definition-explanation-best>

MARTIN, Robert C. Clean Architecture: A Craftsman's Guide to Software Structure and Design. 1ª edição. Estados Unidos da América: Pearson, 2017.

GAMMA, Erich. et al. Design Patterns: Elements of Reusable Object-Oriented Software 1ª edição. Estados Unidos da América: Addison-Wesley, 1994.

<https://datatracker.ietf.org/doc/html/rfc7231>

<https://www.rfc-editor.org/rfc/rfc5646#section-2.1>

[https://developer.mozilla.org/en-US/docs/Web/HTTP/Content\\_negotiation](https://developer.mozilla.org/en-US/docs/Web/HTTP/Content_negotiation)

<https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Accept-Language>

[https://en.wikipedia.org/wiki/List\\_of\\_ISO\\_639-1\\_codes](https://en.wikipedia.org/wiki/List_of_ISO_639-1_codes)

<https://www.iso.org/iso-3166-country-codes.html>

[https://pt.wikipedia.org/wiki/Tempo\\_Universal\\_Coordenado](https://pt.wikipedia.org/wiki/Tempo_Universal_Coordenado)

[https://en.wikipedia.org/wiki/Coordinated\\_Universal\\_Time](https://en.wikipedia.org/wiki/Coordinated_Universal_Time)

[https://en.wikipedia.org/wiki/ISO\\_8601](https://en.wikipedia.org/wiki/ISO_8601)

<https://www.freecodecamp.org/news/synchronize-your-software-with-international-customers/>

<https://www.docker.com/>

[https://pt.wikipedia.org/wiki/Docker\\_\(software\)](https://pt.wikipedia.org/wiki/Docker_(software))

<https://searchitoperations.techtarget.com/definition/Docker-image>