IAGO SORIANO ROQUE MONTEIRO

PLATAFORMA WEB PARA O ENSINO E APRENDIZAGEM DE LÍNGUA ESTRANGEIRA

São Paulo
2022

IAGO SORIANO ROQUE MONTEIRO

PLATAFORMA WEB PARA O ENSINO E APRENDIZAGEM DE LÍNGUA ESTRANGEIRA

Trabalho apresentado à Escola Politécnica da Universidade de São Paulo para obtenção do Título de Engenheiro da Computação.

Orientador:

Prof. Dr. Jorge Becerra Risco

São Paulo
2022

AGRADECIMENTOS

Dedico este trabalho à minha mãe, Livia Sotis Soriano Roque, que, apesar de todas as dificuldades que enfrentou durante a minha criação, sempre garantiu que eu tivesse boa educação escolar, e que apoiou minhas empreitadas acadêmicas e profissionais mesmo depois de eu me tornar um adulto. Sem você, eu não teria chegado até aqui.

Agradeço ao meu avô, Cecílio Antônio Roque, cujo apoio foi essencial em toda a minha vida escolar, principalmente para a minha manutenção em São Paulo durante os anos da graduação. O senhor sempre foi um modelo de profissional para mim.

Agradeço à minha querida amiga Patrícia Rangel G. do Sacramento, que, de outro Estado do Brasil, me apoiou todos os dias de produção deste trabalho. Seu encorajamento foi essencial nesta caminhada.

Agradeço, finalmente, aos amigos que estiveram comigo desde o primeiro dia em 2018. Nos tornamos engenheiros juntos e foi um grande prazer vê-los crescer como profissionais e como pessoas.

RESUMO

Pessoas interessadas em aprender um novo idioma têm, como possibilidade, além das tradicionais escolas de idiomas e universidades, o estudo autodidata com auxílio de um instrutor particular. O papel do instrutor é o de sugerir atividades que forneçam insumo linguístico suficiente para que o estudante da língua estrangeira tenha contato com o idioma-alvo em sua forma autêntica, e tenha incentivo para produzir suas próprias falas, construções e frases naquele idioma.

É com essa interação em mente que nasceu este trabalho. Este texto detalha as funcionalidades, as tecnologias e os conceitos de engenharia de software envolvidos na produção de um site que conecta instrutores particulares de idioma estrangeiro com estudantes. O site é uma plataforma em que instrutores produzem atividades didáticas de compreensão escrita e oral, e estudantes realizam essas atividades e obtém feedback sobre a sua produção.

Como objetivo do projeto paralelo à resolução do problema do campo da educação, tomou-se um cuidado especial para produzir código e utilizar tecnologias que respeitem bons princípios de engenharia de software, assim como utilizar as tecnologias mais recentes utilizadas no mercado em 2022.

Palavras-Chave ReactJS, NextJS, Typescript, NodeJS, Arquitetura Limpa, Terraform, AWS, CICD, Plataforma de Ensino, Aprendizado de Idiomas, Tecnologia na Educação

ABSTRACT

People who are interested in learning a new language have, as a possibility, besides the traditional routes of language schools and universities, self-studying along with a private tutor. The role of the tutor is that of suggesting activities which will provide comprehensible and authentic input to the student, incentivizing them to produce their own output in the target language.

It is with such interaction in mind that this work was born. This text elaborates on the features, technologies and software engineering concepts involved in the making of a web service that connects foreign language instructors to students. The web service is a platform on which instructors author learning activities, especially those envolving written and oral comprehension, and students are able to complete those activities and get feedback on their language output.

As a parallel objective to delivering value in the field of language learning, some special care to making high quality code that follows good software engineering principles has been taken, as well as to using state-of-the-art and current technologies and practices.

Keywords ReactJS, NextJS, Typescript, NodeJS, Clean Architecture, Terraform, AWS, CICD, Learning Platform, Language Learning, Technology for Education

SUMÁRIO

1 INTRODUÇÃO

1.1 Objetivos

Os objetivos deste trabalho são dois: criar um produto útil no mercado de ensino de idiomas, que possa realmente servir a instrutores e estudantes de línguas estrangeiras, e explorar técnicas modernas de Engenharia de Software, especificamente no âmbito do Desenvolvimento Web, cujas técnicas devem promover um ambiente ágil de desenvolvimento, para se possa produzir aplicações escaláveis e de relativa fácil manutenção.

1.1.1 Aprendizado de Engenharia de Software

No âmbito do aprendizado de engenharia, o objetivo é produzir uma aplicação web que empregue boas práticas de código, em um ambiente de desenvolvimento moderno, que conte com testes automatizados, arquitetura limpa e componentes modularizados no *frontend* e *backend*, e de serviços distribuídos, como *pipelines* de *CI/CD* e alta disponibilidade.

A definição de uma plataforma web utilizada neste trabalho é a de um *SaaS*, ou *Software as a Service*. Isso significa que o cliente acessa o produto através da internet, e usa o seu computador e o seu navegador (instalado em seu computador) para fazer requisições ao serviço. O serviço é um programa de computador que ouve requisições e as responde.

O serviço em si será um programa de computador sendo executado num ambiente de nuvem pública: um datacenter em que empresas que disponibilizam SaaS executam seus programas e os disponibilizam para atender requisições de seus clientes. O sistema em questão está hospedado na nuvem pública da Amazon Web Services, ou AWS.

1.1.2 Objetivo do produto

Pessoas interessadas em aprender um novo idioma têm, como possibilidade, além das tradicionais escolas de idiomas e universidades, o estudo autodidata com auxílio de um instrutor. O papel do instrutor é o de sugerir atividades que forneçam insumo linguístico suficiente para que o estudante da língua estrangeira tenha contato com o idioma-alvo em sua forma autêntica, e tenha incentivo para produzir suas próprias falas, construções e frases naquele idioma.

Neste contexto, o objetivo da aplicação é ter uma plataforma unificada onde pessoas instrutoras de idiomas estrangeiros possam criar atividades para estudantes e acompanhar seu desempenho nessas atividades. Os estudantes realizarão as atividades e produzirão seu próprio conteúdo escrito, que será avaliado pelas pessoas instrutoras usuárias da plataforma.

Instrutores podem utilizar a plataforma para criar atividades pedagógicas de ensino de língua estrangeira, como textos e vídeos seguidos de perguntas. Suas atividades são persistidas pelo sistema e disponibilizadas para usuários estudantes. Estudantes utilizam a plataforma para escolher e realizar as atividades previamente inseridas por instrutores, e ter sua produção escrita no idioma-alvo analisada e corrigida também pelos instrutores que criam as atividades. Assim, a plataforma é um ambiente de criação de material didático, de estudo de língua estrangeira e de avaliação de competências escritas em idioma estrangeiro.

1.2 Conteúdo deste trabalho

1.2.1 Descrição do Produto e do Código

No âmbito da descrição do produto, este documento detalha as telas a que os usuários terão acesso, assim como algumas porções do código que executa as funcionalidades que os usuários vêem concretizadas na tela. Essas descrições encontram-se nos capítulos 3, no qual se discute como se implementou a autenticação na aplicação, e no capítulo 4, em que as funcionalidades relacionadas às atividades didáticas são descritas.

1.2.2 Descrição dos conceitos e tecnologias

Nas seções relacionadas a aspectos técnicos da aplicação, encontra-se uma descrição dos principais conceitos de Engenharia de Software que guiaram a produção da aplicação. Conjuntamente a esses conceitos, encontra-se descrições de algumas das tecnologias utilizadas, aquelas julgadas pelo autor as mais interessantes e pertinentes para ilustrar cada conceito.

No capítulo 2, encontra-se a descrição da arquitetura da aplicação e uma explicação de como os seus componentes interagem no ambiente de nuvem pública.

No capítulo 5, dois conceitos de especial interesse na indústria de desenvolvimento web são destacados: a experiência de desenvolvimento de uma aplicação cliente-servidor que faz uso de microsserviços na seção 5.1 e uma solução para o design de uma aplicação web, a Arquitetura Limpa, na seção 5.2.

2 ARQUITETURA DO SISTEMA

2.1 Visão Geral

Abaixo encontra-se um diagrama simplificado da arquitetura do sistema. Vê-se que a aplicação segue uma **arquitetura cliente-servidor**, em que o cliente é a aplicação *frontend*, responsável por renderizar *HTML* em um navegador, e por fazer requisições para a aplicação *backend*, que executa as regras de negócio.

O *backend*, por sua vez, é construído utilizando o paradigma de microsserviços, e está dividido em duas aplicações separadas e independentes, cada uma com uma responsabilidade bem definida. Essas aplicações são

- *auhentication-service*, aplicação responsável por todas as funcionalidades relacionadas à autenticação do usuário, que cuida da criação de novos usuários e da distribuição de tokens de autorização para que os usuários tenham permissão para utilizar as funcionalidades da aplicação de domínio.
- *domain-service*, a aplicação de domínio, em que todas as regras de negócio do sistema, que entregam valor para o usuário de alguma maneira, são executadas.

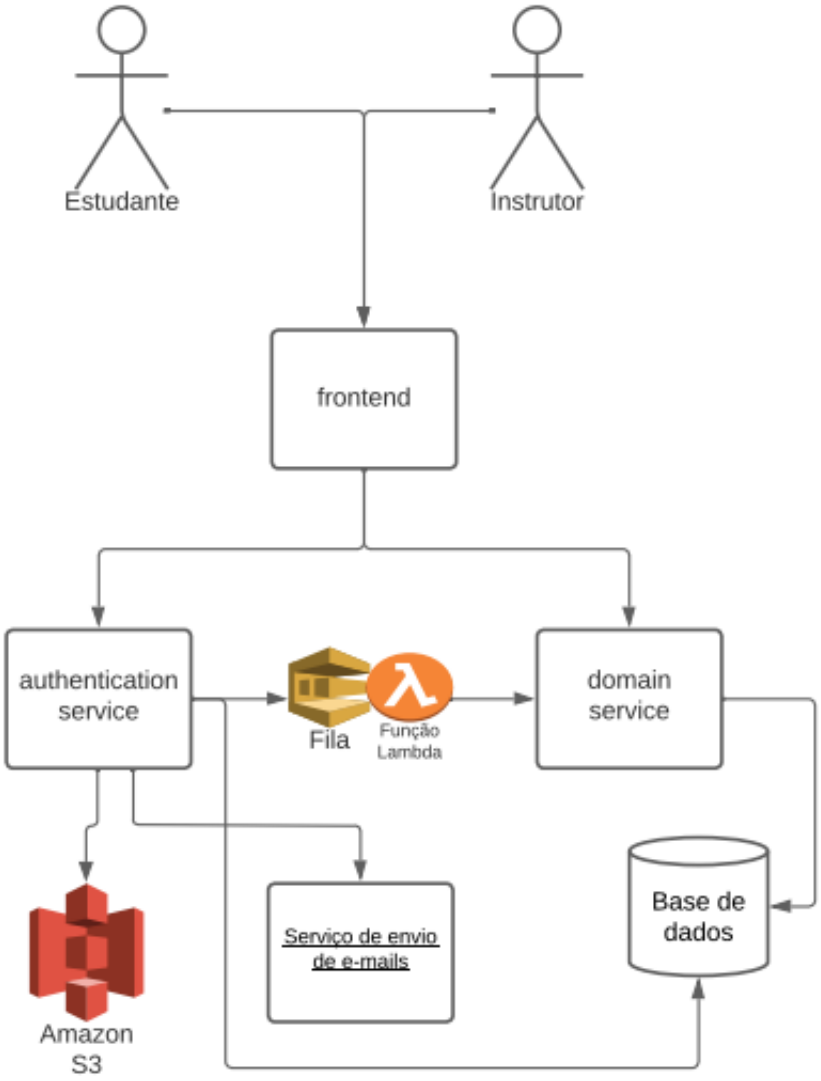


Figura 1: Diagrama simplificado da Arquitetura do sistema

Há três situações em que essas duas aplicações têm que se comunicar entre si:

1. Quando um novo usuário é criado e inserido no sistema. Neste evento, a aplicação de autenticação precisa trazer as informações desse novo usuário à aplicação de domínio, que criará uma representação de um novo instrutor ou estudante e a persistirá em seu banco de dados.

2. Quando um usuário sai do sistema (faz um *sign out*). *Sign out* é o processo inverso a entrar em uma aplicação ou site: as credenciais de acesso do usuário são invalidadas. A aplicação de autenticação atende a essa requisição (a maneira exata como isso acontece é detalhada no capítulo seguinte) e precisa, então, informar a ocorrência desse evento à aplicação de domínio, para que requisições futuras desse usuário não sejam atendidas.

3. Usuário muda sua imagem de perfil. A aplicação de autenticação se encarrega de armazenar e de alterar imagens de perfil. Porém, a aplicação de domínio persiste em seu banco de dados o *url* da última imagem de perfil do usuário para poder exibí-la na tela em algumas situações (ver 4.1). Portanto, quando a imagem é alterada, a aplicação de domínio tem que ser informada da nova *url*.

Para possibilitar essa comunicação, os dois serviços do *backend* se comunicam entre si através de uma fila. Essa é uma maneira assíncrona de comunicação, alternativa a simples requisições/respostas HTTP. A fila armazena mensagens de um serviço emissor e as envia ao serviço receptor, servindo como um intermediário. Se o serviço receptor estiver não responsivo no momento em que a mensagem é enviada, a fila armazena a mensagem e realiza novas tentativas de envio até ser bem-sucedida. Mais detalhes da implementação da fila estão descritos abaixo, em 2.2.1.

Uma vantagem dessa abordagem é permitir ao serviço emissor possa terminar seu processamento e responder ao seu cliente antes da resposta do serviço receptor estar disponível. Uma vez que o serviço receptor receber a mensagem e terminar seu processamento, este pode ou não reponder à fila com uma mensagem nova.

Nos casos de uso de fila desta aplicação, descritos acima, o serviço receptor (aplicação de domínio) não envia novas mensagens à fila após terminar o processamento das mensagens.

Como solução de armazenamento de arquivos, foi empregado o S3, o *Simple Storage Service* da AWS (AWS..., e). Como o nome sugere, este é um serviço que armazena arquivos a baixo custo. Ele é utilizado pela API de autenticação para armazenar as imagens de perfil dos usuários.

Finalmente, um serviço de envio de e-mails chamado Sendgrid (SENDGRID,) é utilizado pela API de autenticação para enviar e-mails de verificação de conta e de redefinição de senha.

2.2 A arquitetura na AWS

Para concretizar a arquitetura descrita na seção anterior, alguns serviços da AWS foram utilizados. Dentre os mais importantes, pode-se mencionar o S3, descrito brevemente acima, o SQS (AWS..., f) com Lambda (AWS..., c) para a mensageria entre os serviços, o serviço de orquestração de containers ECS (*Elastic Container Service*) (AWS..., b), a

nuvem privada VPC (AWS..., g), o banco de dados gerenciado RDS (AWS..., d) e uma instância EC2 (AWS..., a) utilizada como bastion host. Abaixo, encontram-se descrições de como esses serviços são utilizados nesta aplicação. Para mais detalhes de como esses serviços estão configurados, ver os arquivos nas pastas de cada serviço em *infra/modules*, no código fonte deste trabalho.

2.2.1 SQS e Lambda

O serviço de filas simples da AWS, o *SQS* (*Simple Queue Service*) opera sob um mecanismo de *pull*: as mensagens chegam na fila e devem ser lidas pelo serviço de destino. Uma vez lida e processada, o serviço de destino envia uma mensagem para a fila dizendo que a mensagem pode ser removida. Assim, mais nenhum outro serviço que possa estar consumindo mensagens da fila poderá reprocessar a mensagem em questão.

Para que um serviço possa consumir mensagens de uma fila que disponibiliza mensagens sob um mecanismo de *pull*, este precisa fazer *polling*: enviar mensagens periódicas à fila perguntando se há mensagens novas. Porém, *polling* é uma técnica diferente dos *endpoints HTTP* usuais, que são utilizados nas aplicações deste trabalho.

Para ter serviços que utilizassem apenas *endpoints HTTP* e evitar de ter que fazer *polling*, foi utilizada a técnica de uma *consumer Lambda*. *Lambda* é o serviço de *Function as a Service* da AWS. Ou seja, são códigos que são executados na nuvem em resposta a eventos configuráveis, sem a necessidade dos clientes (no caso, o autor deste trabalho) gerenciarem máquinas virtuais nem código de aplicações *HTTP* na nuvem. Uma *Lambda* foi configurada para "ouvir"quando novas mensagens chegam à fila e enviar essa mensagem para o endpoint HTTP da API de domínio configurado na mensagem.

```
1  const axios = require("axios");
2
3  exports.handler = async function (event) {
4    const response = { batchItemFailures: [] };
5
6    const promises = event.Records.map(async (record) => {
7      try {
8        const { body, messageAttributes } = record;
9        console.log({ body, messageAttributes });
10       await axios[messageAttributes.httpMethod.stringValue](
11         `https://${process.env.API_URL}/${messageAttributes.httpPath.
               stringValue}`,
12         JSON.parse(body)
```

```
13        );
14      } catch (e) {
15        console.error("error", e);
16        response.batchItemFailures.push({ itemIdentifier: record.messageId
              });
17      }
18    });
19
20    await Promise.all(promises);
21    console.log("response", JSON.stringify(response));
22    return response;
23  };
```

Listing 2.1: Código da função Lambda. Oberva-se que ela obtém da mensagem da fila o método HTTP e a rota para onde enviar a requisição além do corpo da requisição.

2.2.2 Docker

Ambas as aplicações são executadas no ambiente da AWS dentro de containers *Docker*. Um container *Docker* permite que todo o ambiente em volta da aplicação, desde suas dependências até o sistema operacional tenha sempre o mesmo comportamento, seja lá em que máquina a aplicação for executada. (DOCKER,)

Para se executar um container *Docker*, é necessário primeiro definir uma imagem. A imagem contém o *kernel* de um sistema operacional, o código da aplicação a ser executada, e comandos para instalar e executar a aplicação. Seguem abaixo as definições de imagens *Docker* para as duas aplicações.

```
1  FROM public.ecr.aws/bitnami/node:latest
2
3  WORKDIR /app
4
5  COPY ./package*.json ./
6  COPY ./tsconfig.json ./
7  COPY ./packages/auth-web-api/ ./packages/auth-web-api/
8  COPY ./packages/common-core/ ./packages/common-core/
9  COPY ./packages/common-platform/ ./packages/common-platform/
10 COPY ./packages/common-utils/ ./packages/common-utils/
11
12 RUN npm i
13
14 WORKDIR /app/packages/auth-web-api
```

```
15
16 RUN npm run build
17
18 ENTRYPOINT ["npm", "start"]
```

Listing 2.2: Dockerfile da aplicação de autenticação.

```
1  FROM public.ecr.aws/bitnami/node:latest
2
3  WORKDIR /app
4
5  COPY ./package*.json ./
6  COPY ./tsconfig.json ./
7  COPY ./packages/web-api/ ./packages/web-api/
8  COPY ./packages/common-core/ ./packages/common-core/
9  COPY ./packages/common-platform/ ./packages/common-platform/
10 COPY ./packages/common-utils/ ./packages/common-utils/
11
12 RUN npm i
13
14 WORKDIR /app/packages/web-api
15
16 RUN npm run build
17
18 ENTRYPOINT ["npm", "start"]
```

Listing 2.3: Dockerfile da aplicação de domínio.

Nos *Dockerfiles* acima, seguem-se os seguintes passos:

1. Define-se um Sistema Operacional *Linux* com *Node* na sua última versão instalado.
2. Cria-se um diretório chamado */app*, onde o código da aplicação será armazenado.
3. Copiam-se os arquivos relevantes do diretório atual (onde o comando para se construir a imagem *Docker* a partir desse arquivo Dockerfile está sendo executado) para a imagem.
4. Instalam-se as dependências da aplicação.
5. Realiza-se o processo de *build*, que, no caso das tecnologias aqui utilizadas, significa transformar todo o código da linguagem Typescript para a linguagem Javascript.

6. Define-se o comando a ser executado para executar a aplicação, chamado *start*. Para mais detalhes do que faz o comando, ver o arquivo *language-app/packages/auth-web-api/package.json*, no código fonte do projeto.

2.2.3 ECS

ECS é a tecnologia de orquestração de containers da AWS. Orquestração de containers é um conceito importante em arquiteturas de microsserviços. Em uma arquitetura de microsserviços, espera-se que os serviços tenham, todos, certo nível disponibilidade, como 99% ou 99,999% do tempo servindo requisições. Porém, é comum que essas aplicações deixem de funcionar em algum momento, seja por erros de programação, erros causados por entradas inesperadas, ou devido à eventual e momentânea incapacidade da infraestrutura física de responder à carga sobre o sistema. Seja qual for o motivo de eventuais falhas, um orquestrador de containers garante que, a todo momento, um número desejado de instâncias de cada aplicação está sendo executado e está respondendo às requisições.

Neste projeto, o *ECS* garante que há, a todo momento, uma instância da aplicação de autorização e uma instância da aplicação de domínio funcionando. O orquestrador sabe se essas instâncias são capazes de responder requisições, ou se estão "saudáveis", através de um *endpoint HTTP* em */*, chamado *healthcheck*. O orquestrador faz uma requisição a este endpoint e espera receber uma resposta com *HTTP status* 200 (ok). Se for este o caso, o orquestrador conclui que aquela instância está saudável e a contabiliza como uma das instâncias saudáveis daquela aplicação naquele momento. Se qualquer outra resposta retornar (ou nenhuma resposta), o orquestrador descobre que aquela instância não está saudável. Neste caso, a instância é terminada e uma nova é iniciada. Esse processo acontece a cada minuto, e garante que a quantidade de instâncias da aplicação desejada é cumprida.

2.2.4 Rede e VPC

A AWS é chamada de *rede pública*. Este nome é um contraste às *redes privadas*, redes de computadores que se comunicam entre si e cujos endereços de IP não são endereçáveis por agentes fora daquela rede. Devido a essa inacessibilidade, as redes privadas são tidas como altamente seguras.

Assim, para trazer a segurança das redes privadas dos ambientes *on-premise* à rede pública da AWS, esta oferece o serviço de VPC, ou *Virtual Private Cloud*. Uma *private*

cloud, ou nuvem privada, é uma rede privada dentro da rede pública da AWS.

Os recursos computacionais da infraestrutura são provisionados dentro de uma nuvem privada da AWS, e recebem endereços de IP públicos ou privados. Se um recurso deve ser acessível de fora da nuvem privada, este é provisionado em uma **sub-rede pública**, e recebe um endereço de IPv4 endereçável publicamente. Por outro lado, se o recurso não deve ser acessado do lado de fora, este é provisionado numa **sub-rede privada** e recebe apenas um endereço de IP privado.

Na arquitetura deste trabalho, apenas o banco de dados foi provisionado em uma sub-rede privada. Os containers das aplicações, assim como os balanceadores de carga (ver 2.2.5) e o *bastion host* (ver seção a respeito) foram provisionados em sub-redes públicas.

O banco de dados nunca deve ser acessado diretamente por fora da rede, mas sim apenas pelas aplicações, que fazem uso deste para persistência de dados. Portanto, somente poderia estar em uma sub-rede privada.

O *bastion host* deve existir em uma sub-rede pública porque o seu objetivo é de, justamente, ser acessível por computadores de fora da rede.

Os containers das aplicações, contudo, deveriam estar em sub-redes privadas, como boa prática de segurança. Porém, estes foram provisionados em sub-redes públicas por uma questão de gerenciamento de custos. Acontece que, para que os containers das aplicações estejam em redes privadas, (e, portanto, mais seguros, por não terem IPs públicos), é necessário provisionar IPs públicos a essas máquinas, para que elas respondam às requisições dos usuários. Esse processo é chamado de *NAT Translation* e requer o provisionamento de um componente da AWS chamado *NAT Gateway*. Porém, a AWS cobra por byte de dado que é transferido pelos containers através do *Gateway*, o que aumentou o custo de operação da infraestrutura em cerca de 5 vezes. Portanto, para uma fase de prototipação, foi escolhido manter os containers em sub-redes públicas, em que esse processo não se faz necessário. A segurança não está severamente comprometida, contudo, pois, apesar desses containers terem IPs públicos, eles contam também com *firewalls* (*AWS Security Groups)*, que garantem que nenhum IP pode acessá-los por SSH, e que qualquer IP pode acessar somente a porta 3006, em que a aplicação está sendo executada.

2.2.5 Balanceadores de Carga

Balanceadores de carga são um tipo de *proxy reverso*. São servidores que interceptam as requisições direcionadas aos servidores da aplicação e as redirecionam segundo alguma regra ou após algum processamento.

Neste trabalho, há um balanceador de carga por ambiente (ver 5.5.1), e seu propósito é de receber requisições e as direcionar à aplicação correta. Para isso, foram configuradas as seguintes regras no balanceador de carga (*Application Load Balancer*):

1. Requisições destinadas a *[staging.]api.language-app.isrm.link/auth-web-api* são direcionadas à aplicação de autenticação.
2. Requisições destinadas a *[staging.]api.language-app.isrm.link/web-api* são direcionadas à aplicação de domínio.

O Balanceador de carga também é responsável por garantir o acesso à aplicação via HTTPS.

2.2.6 Postgre e RDS

PostgreSQL é um banco de dados relacional *open-source* altamente popular no mercado. Este é o banco de dados utilizado para persistência pelas aplicações neste projeto. (POSTGRE,)

RDS é um serviço da AWS que gerencia uma máquina virtual com algum banco de dados instalado. O serviço cuida da instalação e atualização da instância do banco de dados, restando ao desenvolvedor apenas configurar o *schema* dos dados na instância, que tem alta disponibilidade.

Foi escolhido o RDS pois a AWS oferece, no seu plano gratuito de um ano, 24horas por dia de uma instância PostgreSQL numa máquina do tipo *db.t3.micro* (nomenclatura para máquinas virtuais particular da AWS). Esta é uma das máquinas com menos memória e processamento disponíveis na AWS, mas é o suficiente para uma aplicação em fase de prototipação.

Numa arquitetura de microsserviços, o ideal é que cada aplicação conte com sua própria instância de banco de dados. Porém, neste projeto, para que fosse possível permanecer dentro dos limites da oferta gratuita da AWS, todos os quatro bancos de dados (um por aplicação por ambiente: pré-produção e produção) estão localizados na mesma

instância. Para modificar essa arquitetura para uma versão mais profissional, com quatro instâncias independentes, bastaria adicionar mais instâncias RDS ao código de definição de infraestrutura e passar suas *urls* para cada serviço do ECS.

3 FLUXO DE AUTENTICAÇÃO

Nas seções deste capítulo estão apresentadas as telas, e as funcionalidades presentes no fluxo de autenticação da aplicação.

A autenticação do sistema conta com tokens JWT como mecanismo de autenticação, além de uma *versão de token*, que controla se o usuário tem permissão para acessar os recursos do sistema com seu *token* atual.

3.1 Tokens JWT

A autenticação utiliza tokens JWT, que se tratam de um método padrão RFC 7519 aberto para atestar identidades entre entidades de maneira segura na internet. Os tokens são criados pelo serviço de autenticação, que os produz utilizando dois elementos: os dados que o token irá conter, e um segredo (uma string de caracteres aleatórios), armazenado de maneira segura como variável de ambiente no servidor em que a aplicação é executada. Esse segredo tem a função de garantir a *integridade* do token: se alguém que não conhece o segredo alterar os conteúdos do token, o servidor reconhecerá que os token foi corrompido e não o aceitará.

No fluxo de autenticação, quando um usuário entra no sistema, o serviço de autenticação cria um JWT token com informações relevantes para o serviço de domínio, como o id do usuário e versão atual do token de autenticação. Esse token é devolvido para o usuário e armazenado no seu navegador. A aplicação do *frontend*, então, utiliza esse token para realizar as requisições subsequentes ao serviço de domínio.

O conteúdo do token em si é descoberto, o usuário pode ler o conteúdo do token. Ou seja, o token não é um mecanismo de criptografia. Porém, o usuário não é capaz de alterar o conteúdo do token e ter suas requisições subsequentes atendidas, pois o usuário não conhece o segredo com o qual o token foi criado. Apenas as aplicações do *backend* conhecem o segredo.

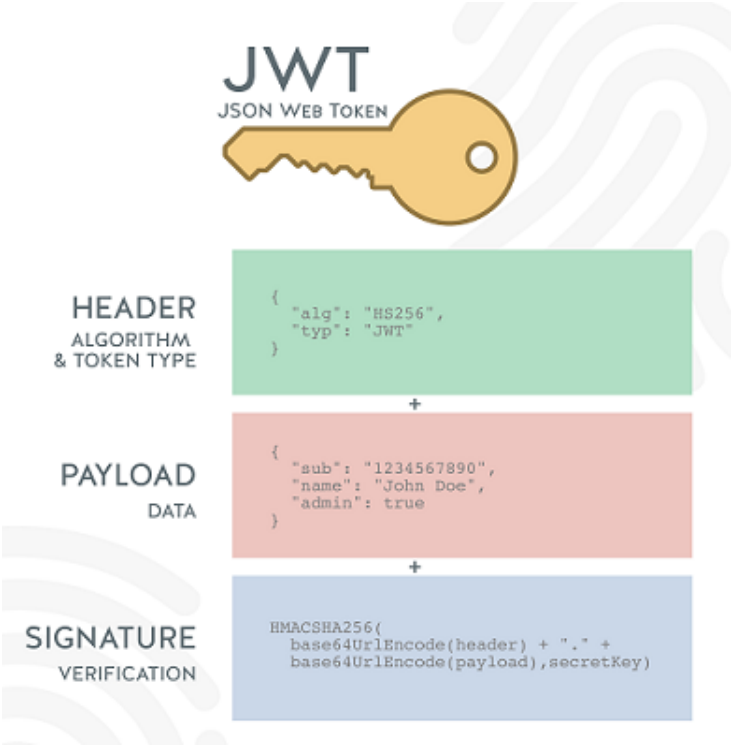


Figura 2: Anatomia de um token JWT (MACORATTI, 2019)

3.1.1 Versionamento de Tokens

O serviço de autenticação emite tokens que carregam dentro de si informações que identificam o usuário, como o seu id, e a versão da sua autenticação, utilizada para verificar se a sessão ativa do usuário ainda é válida.

Os tokens têm uma versão, um dado numérico que começa em 0 no evento de *sign up*, quando o usuário cria uma conta, e é persistida no servidor de autenticação. O endpoint de *sign in* emite tokens em sua versão atual.

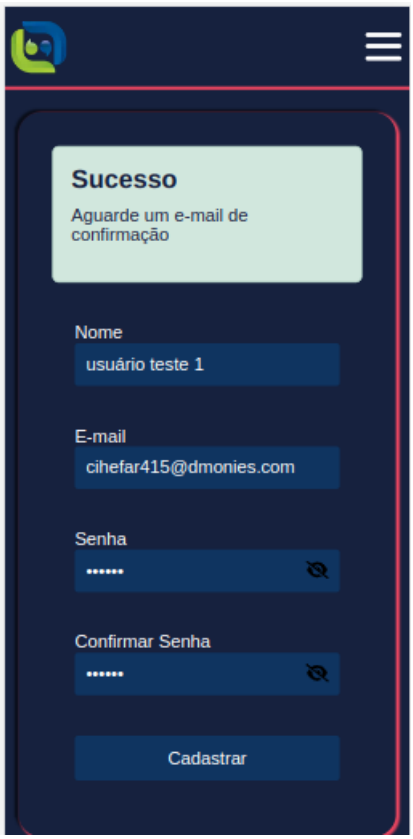
O propósito do versionamento de tokens de autenticação é invalidar *sign ins* realizados anteriormente. Os casos de uso de *sign out* e de troca de senha incrementam a versão do token associada ao usuário. Esses casos de uso pertencem à aplicação de autenticação, e esta emite um evento à aplicação de domínio, informando-a sobre a nova versão de token de autenticação do usuário em questão.

Posteriormente, no momento em que a API de domínio receber uma requisição, ela vai verificar o número de versão contido no token. Se tiver havido um logout ou troca de senha desde a emissão desse token, a versão do token que consta no banco de dados da aplicação de domínio será diferente daquela contida no token. Isso diz à aplicação que o token não é mais válido. A aplicação não atende a requisições com tokens de autenticação

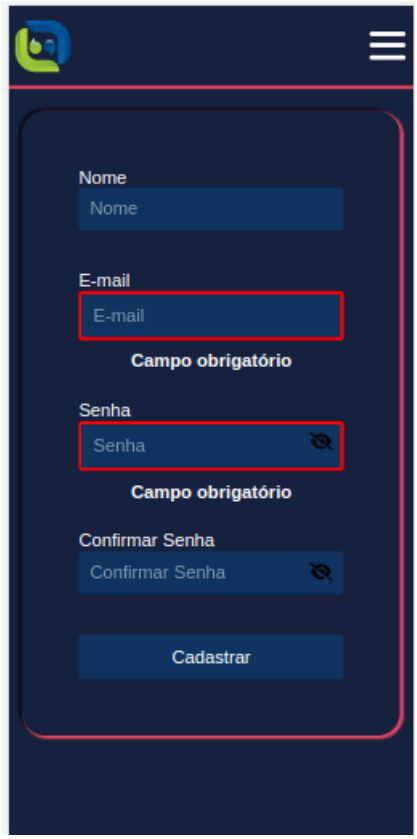
inválidos.

3.2 Criação de novo usuário

Nesta tela, o usuário insere seu nome, e-mail, senha e uma confirmação de senha. Esses dados são validados e enviados para o serviço de autenticação, no *endpoint HTTP* POST /signup.



(a) Criação de conta bem-sucedida



(b) Usuário tenta criar conta com dados inválidos e a interface indica onde está o problema

No código que lida com essa requisição, temos os dados do formulário, mais o idioma preferido do usuário (obtido pelo *dropdown* na barra de navegação) como entradas do endpoint.

O código vai confirmar se o e-mail ainda não está cadastrado (e retornar um erro se já estiver), verificar se a senha e a confirmação de senha são idênticas, e, então, criar um usuário novo. O usuário é criado sem um *role* definidio, ou seja, o usuário ainda não é nem instrutor e nem estudante. O usuário também recebe uma imagem de perfil genérica, e sua senha é persistida após ser criptografada criptografada, já que armazenar senhas de maneira visível no banco de dados seria uma falha de segurança.

Então, um serviço de e-mails envia um e-mail de verificação de conta (ver 3.3) no idioma selecionado para o e-mail que acaba de ser cadastrado, e um token para a subsequente verificação de conta é persistido.

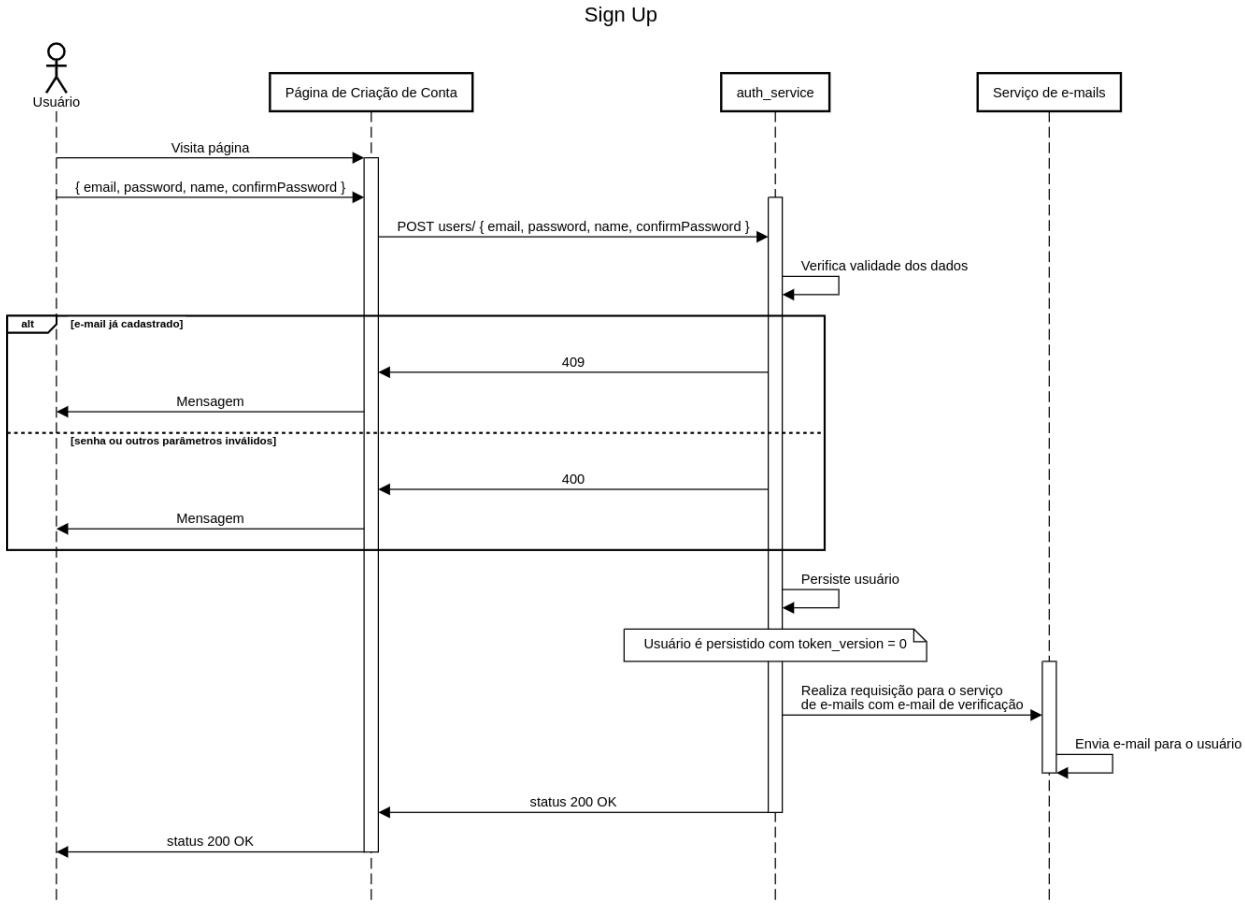


Figura 4: Diagrama de Sequência do fluxo de criação de conta

3.3 Usuário confirma sua conta

O e-mail enviado no fluxo descrito acima contém um link que leva para a página de verificação de conta.

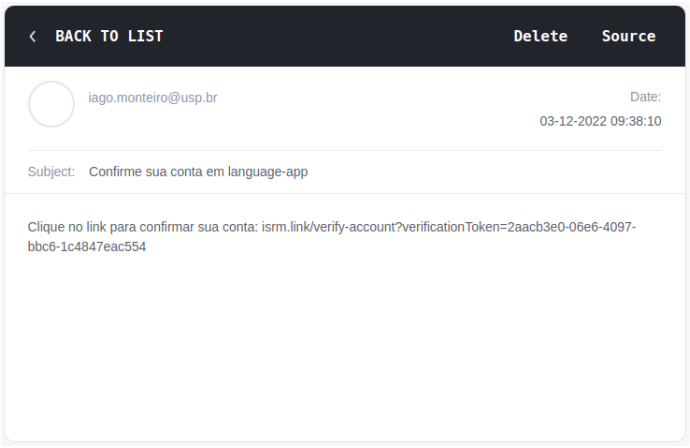
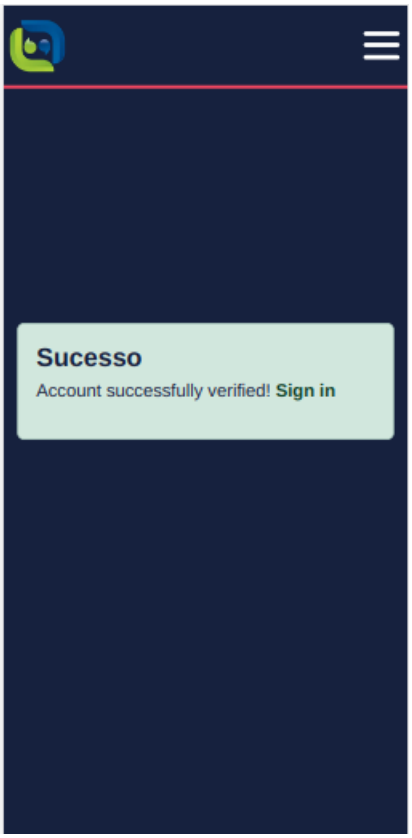


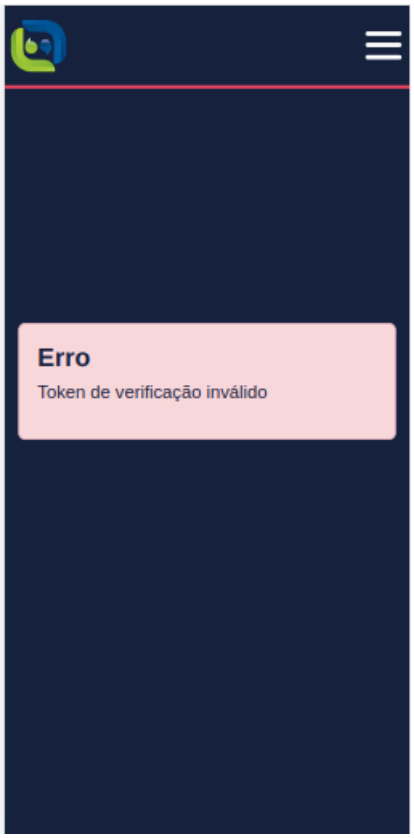
Figura 5: Diagrama de Sequência do fluxo de criação de conta

O usuário não precisa fazer nada mais após clicar no link, pois o *endpoint* de validação de conta (PATCH verification-tokens/:token) recebe uma requisição com o token contido na *url* assim que a página é aberta.

Se o token for válido (será, se o usuário não tiver alterado a *url* enviada por e-mail), uma mensagem de confirmação é exibida. Caso contrário, há uma mensagem de erro.



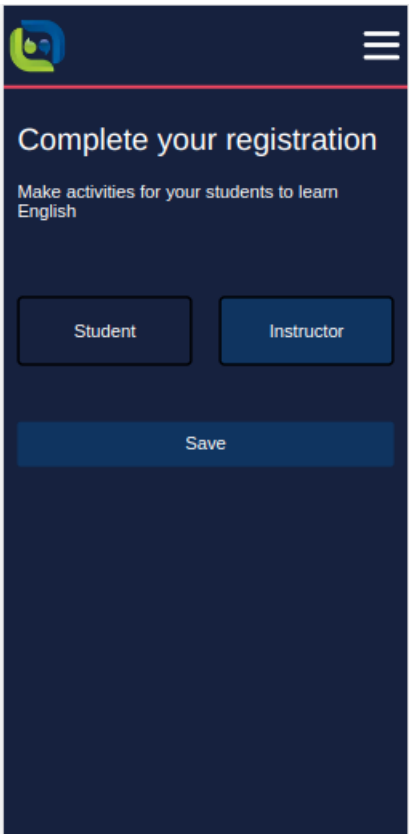
(a) Token de verificação de conta enviado ao se clicar na url é válido e a conta do usuário foi verificada



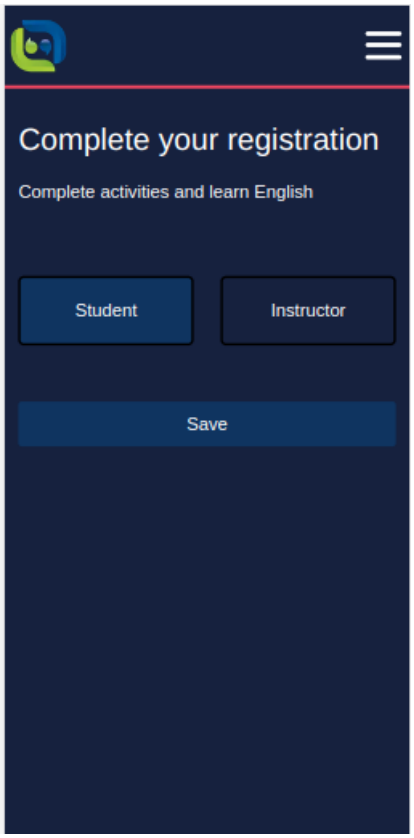
(b) oken de verificação de conta enviado ao se clicar na url **não** é válido e a conta do usuário **não** foi verificada

3.4 Usuário define seu papel

/dashboard é a *url* para a *área do usuário*, em que instrutores e estudantes encontram botões contendo tudo aquilo que podem realizar na plataforma. Porém, para que a aplicação saiba qual configuração de painel (ou área do usuário) exibir, é necessário definir se o usuário será um instrutor ou um estudante. Essa escolha é realizada assim que o usuário acessa sua área após confirmar sua conta.



(a) Usuário prestes a escolher ser um instrutor



(b) Usuário prestes a escolher ser um estudante

Uma vez que a seleção é realizada, o cliente realiza uma requisição para PATCH /users, informando o papel (ou *role*) escolhido.

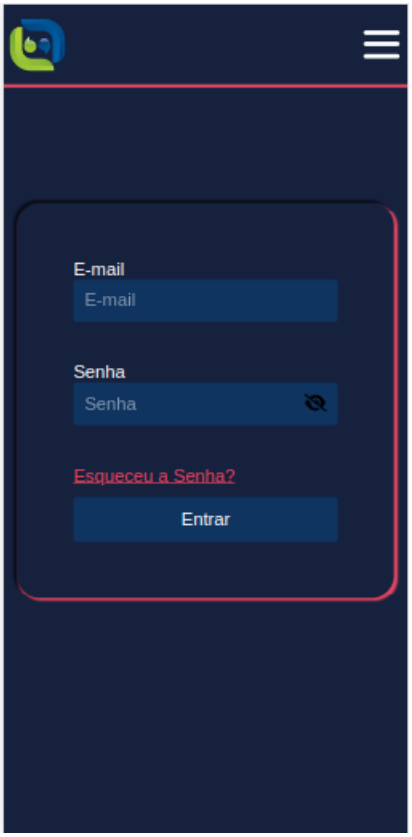
O código que lida com essa requisição valida se o texto enviado como *role* é válido, e, então, atualiza o campo *role* do usuário no seu banco de dados.

Neste momento ocorre a primeira comunicação entre aplicações *backend* da arquitetura de microsserviços. Com o usuário definido, é hora do servidor de domínio persistir esse novo usuário, além de criar uma entidade nova correspondente (uma nova entidade *Instructor* ou *Student*). Para tanto, uma mensagem com as informações do usuário são enviadas para a fila.

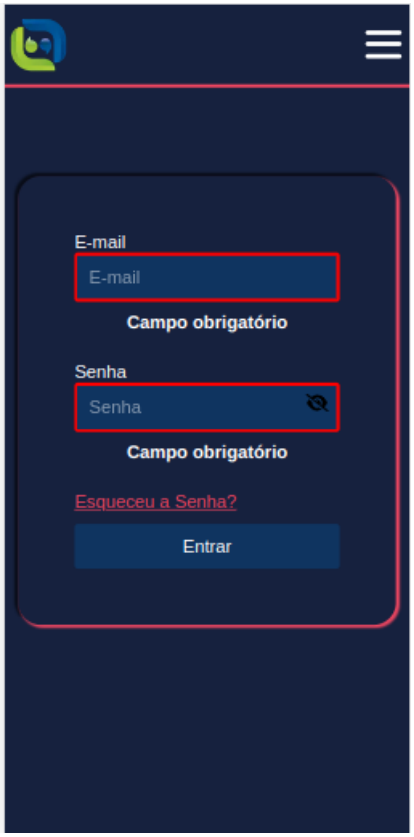
A função Lambda que consome as mensagens da fila envia as informações do usuário para o *endpoint* POST users da API de domínio.

3.5 Emissão de tokens

Nesta tela, o usuário insere suas credenciais para entrar no sistema. Seu e-mail e senha tem seus formatos validados e são enviados para POST /sessions da aplicação de autenticação.



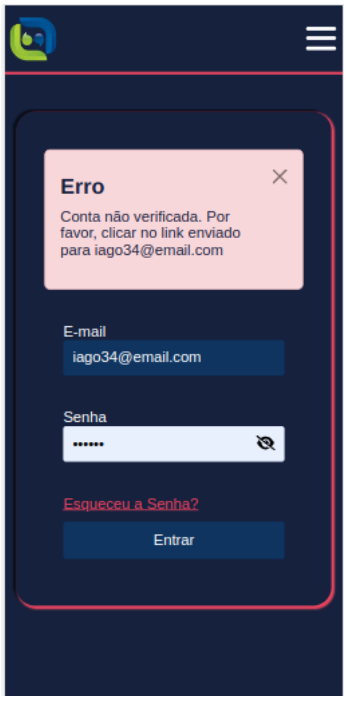
(a) Tela de entrar no sistema antes de qualquer interação



(b) Tela de entrar no sistema depois de uma validação da interface

No *backend*, o *endpoint* verifica se os campos de e-mail e senha foram informados, e se a senha, depois de criptografada, corresponde àquela salva no banco de dados para este e-mail. Verifica-se, também, se o usuário já verificou sua conta. Sem verificar a conta previamente, é impossível entrar no sistema.

Se todas as informações estiverem corretas, um novo token é emitido, contendo a *tokenVersion* associada ao usuário no momento. Se nenhum *sign out* ou pedido de troca de senha tiver sido realizado até o momento, essa versão terá valor 0.



(a) Tela de entrar no sistema depois de uma tentativa mal-sucedida de entrar no sistema

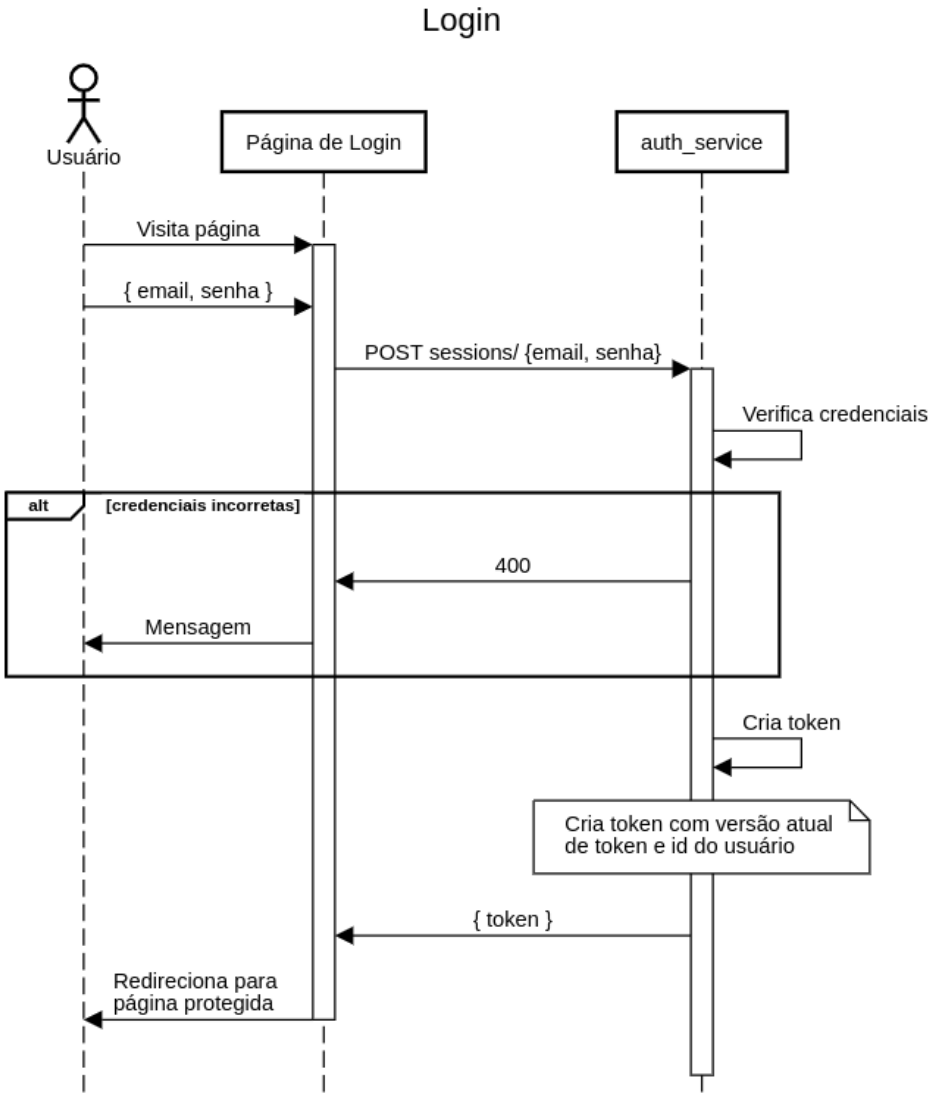


Figura 10: Diagrama de Sequência do fluxo de emissão de token de autenticação

3.6 Sign out: usuário sai do sistema

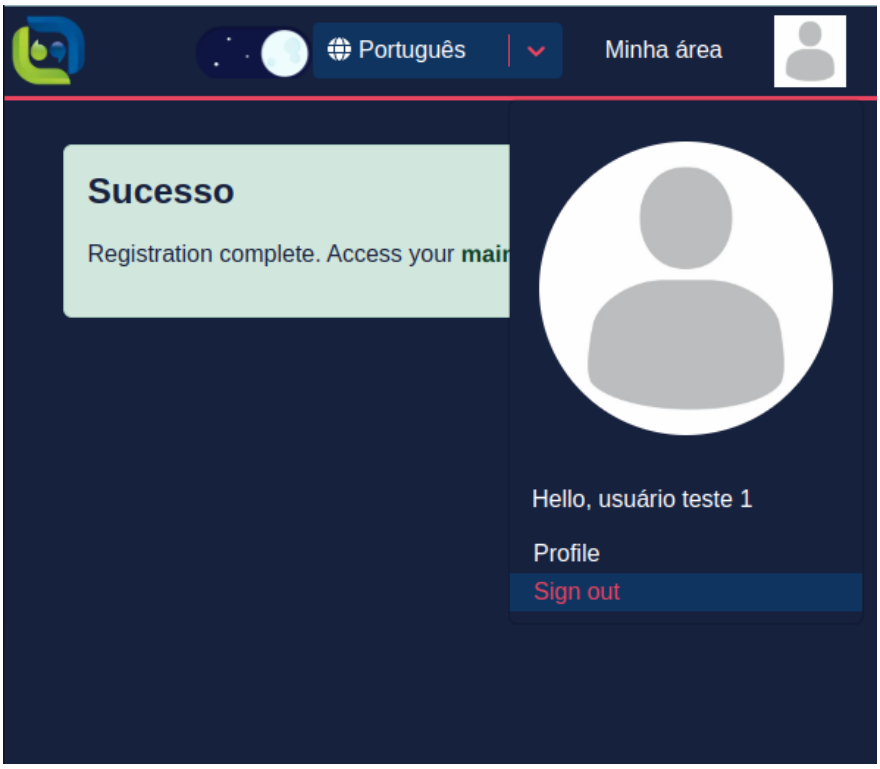


Figura 11

Ao clicar no botão ”Sair”na barra de navegação, uma requisição é enviada para PATCH /sessions. No *backend*, a versão atual do token é incrementada, e a API de domínio recebe uma requisição (através da fila) em PATCH /users, trazendo a nova versão. Na API de domínio, então, a versão do token é atualizada.

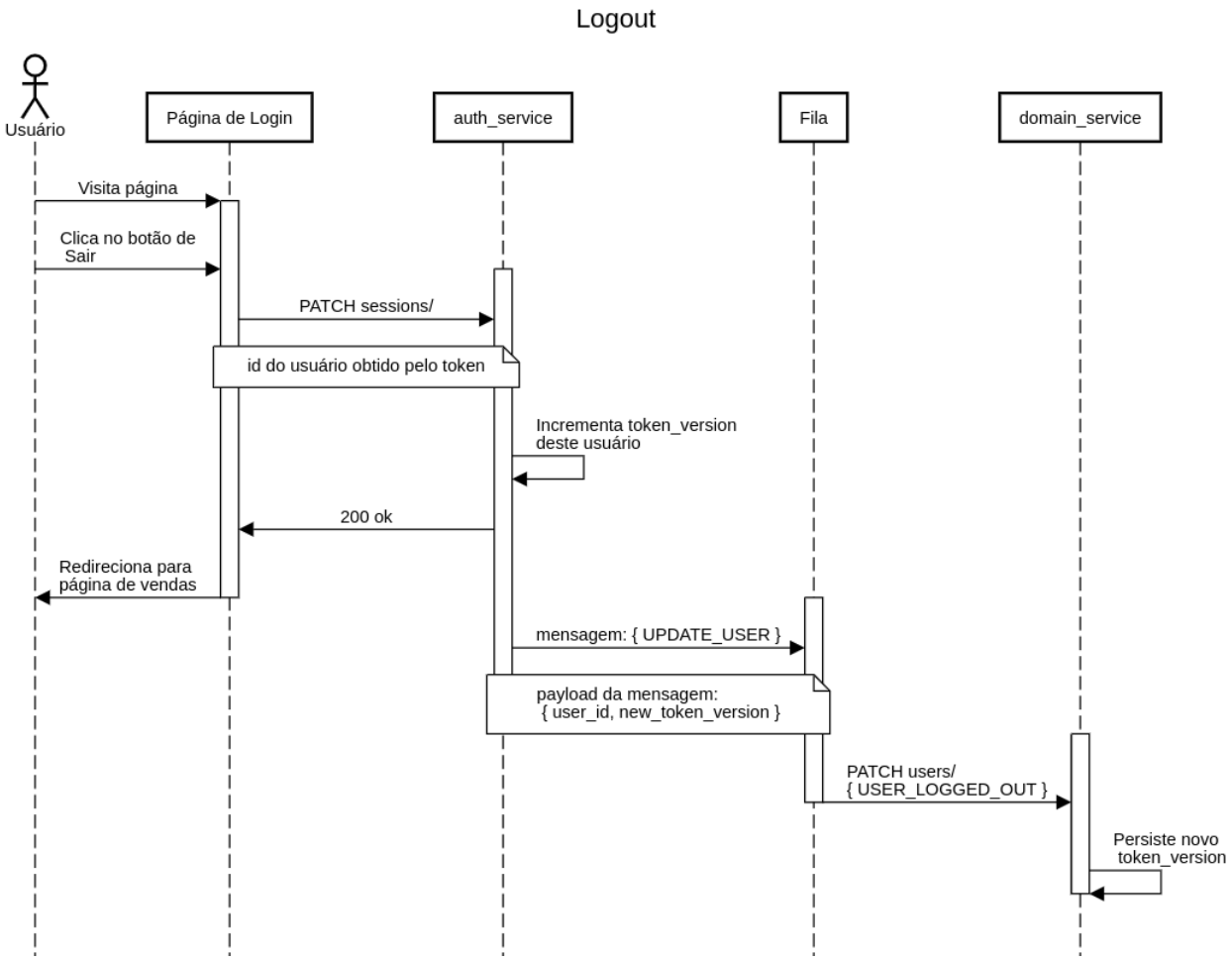


Figura 12: Diagrama de Sequência do fluxo de término de sessão

3.7 Requisição de Troca de senha

Este é o início do fluxo de troca de senha. Tudo começa com o botão de "Esqueci minha senha", presente no formulário de Entrar no sistema.

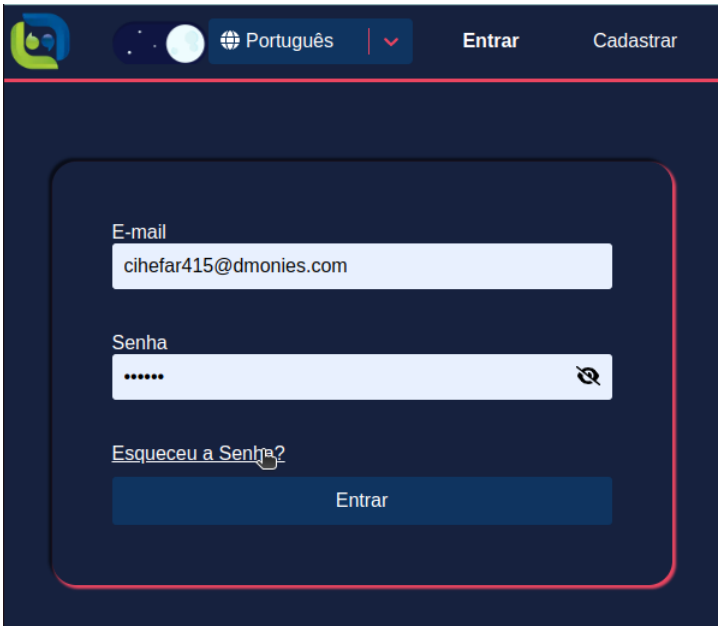


Figura 13: Usuário com o *mouse* sobre o botão que inicia o fluxo de troca de senha

Ao clicar nesse botão, o usuário é redirecionado à página "Esqueci minha senha". Nessa página, o usuário insere seu e-mail. O e-mail é enviado para POST /forgot-password-tokens. Ali, verifica-se se o e-mail pertence a um usuário válido. Se for o caso, um token de troca de senha é gerado. O token tem um campo *data de validade*, cujo valor é dali a três horas. Dessa maneira, a troca de senha só será possível dentro de três horas do início do fluxo.

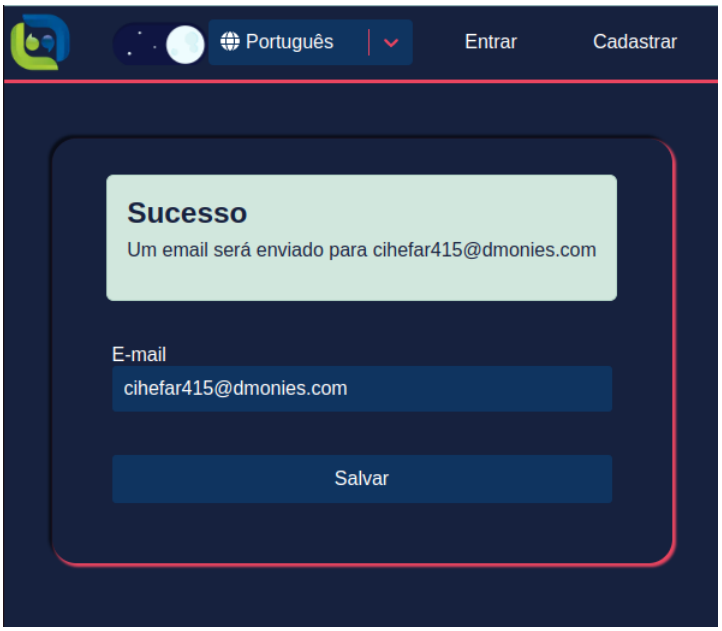


Figura 14: Usuário recebe confirmalão de que o fluxo de troca de senha foi iniciado

Então, um e-mail com um link para a troca efetiva da senha é enviado ao usuário. O

link é para a página de troca de senha, e contém o token gerado.

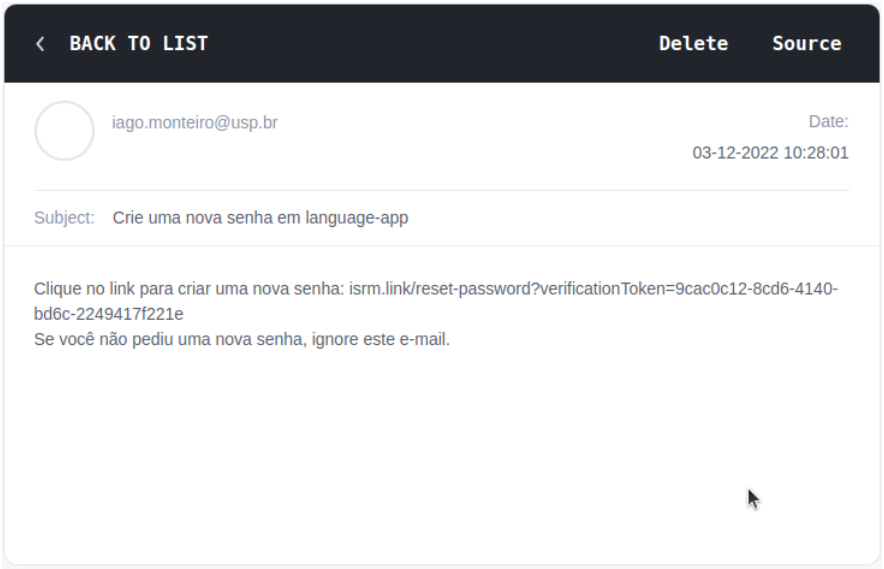


Figura 15: E-mail enviado ao usuário com a *url* para se realizar a troca de senha

3.8 Troca de senha

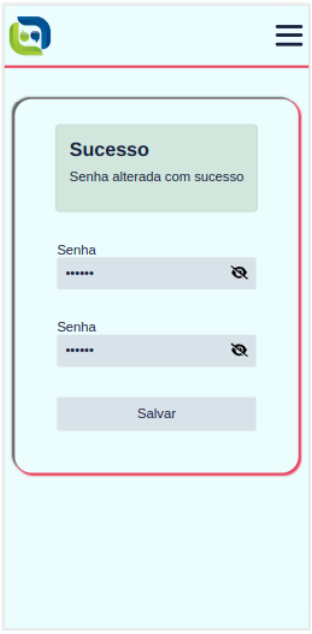


Figura 16: Nova senha é válida e a senha do usuário foi redefinida com sucesso

Na tela de troca de senha, o usuário insere uma nova senha e sua confirmação. Uma requisição é enviada para o *endpoint* PUT /users/passwords, com as senhas e o token.

No *backend*, verifica-se se o token ainda está válido. Assim, se o usuário levar mais de 3h para trocar de senha após a requisição, ele terá que reiniciar o processo. Verifica-se,

também, se a senha e a sua confirmação são idênticas.

Se todas as informações conferirem, a senha informada é criptografada e a entidade de usuário é atualizada no banco de dados. Além disso, atualiza-se o token de requisição de troca de senha para que sua data de expiração seja *Date().now()*. Assim, esse token estará inválido daqui para frente, e o usuário não poderá reutilizá-lo para trocar de senha novamente. Por último, a *tokenVersion* é incrementada (o usuário sofre um *sign out*), o que implica o passo de comunicação assíncrona com a aplicação de domínio informando a nova versão do token, como descrito nas seções anteriores.

Abaixo, um diagrama de sequência de todo o processo de troca de senha, desde sua requisição até a troca da senha.

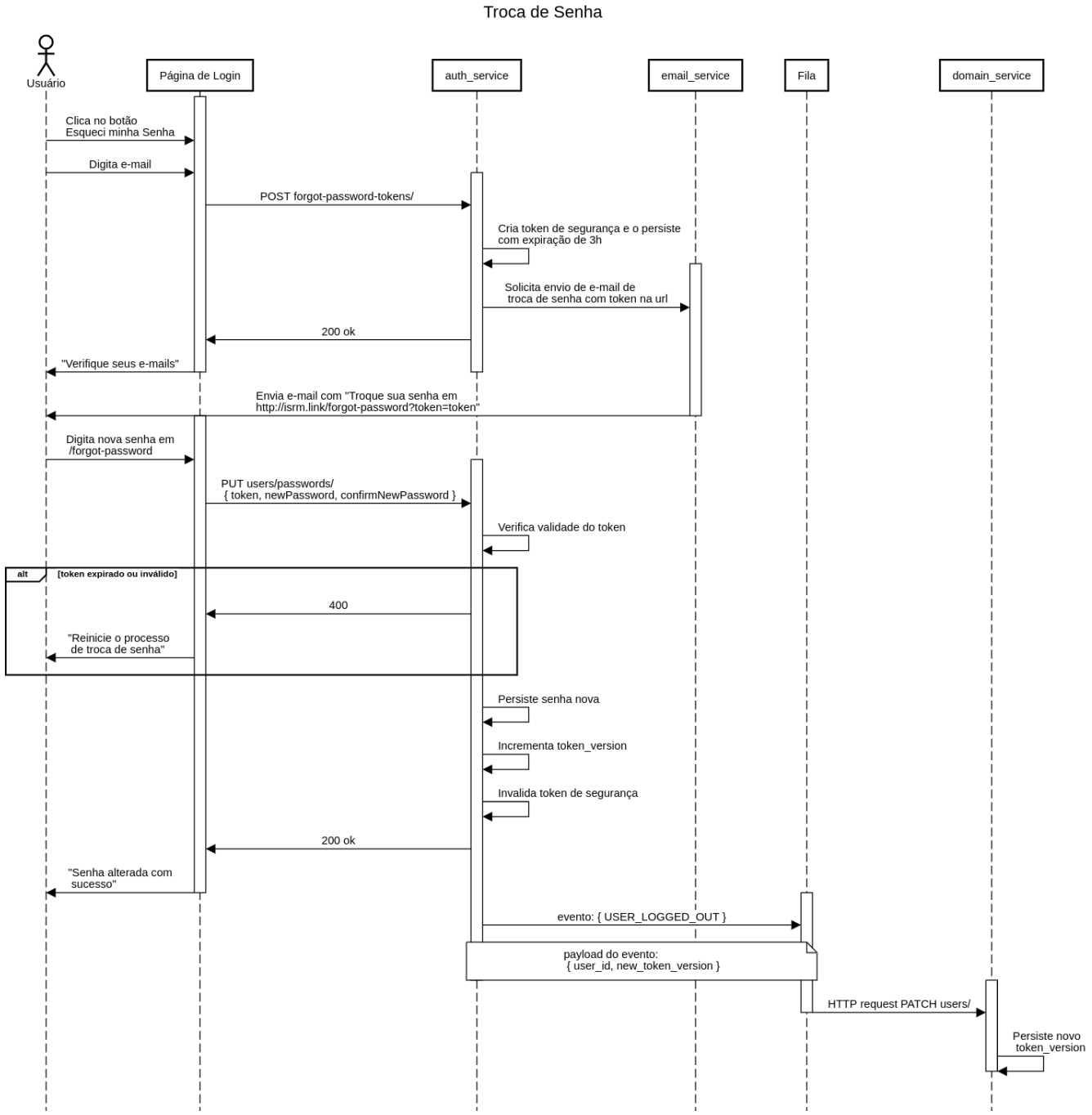


Figura 17: Diagrama de Sequência do fluxo de requisição de nova senha e de troca de senha

3.9 Troca de imagem de perfil

Com esta funcionalidade, é possível trocar a imagem genérica de perfil que é designada a todos os usuários no momento de criação de conta.

Basta acessar a página de perfil pela barra de navegação e clicar no ícone de editar a foto. Então, no modal, fazer upload de uma nova imagem a partir dos arquivos do

computador.

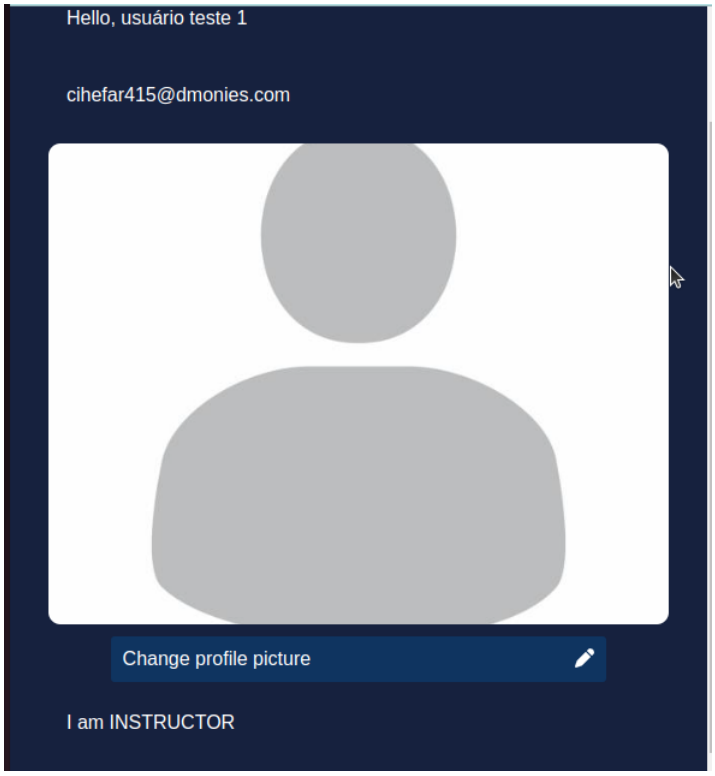


Figura 18: Tela de perfil do usuário antes da primeira troca de imagem de perfil

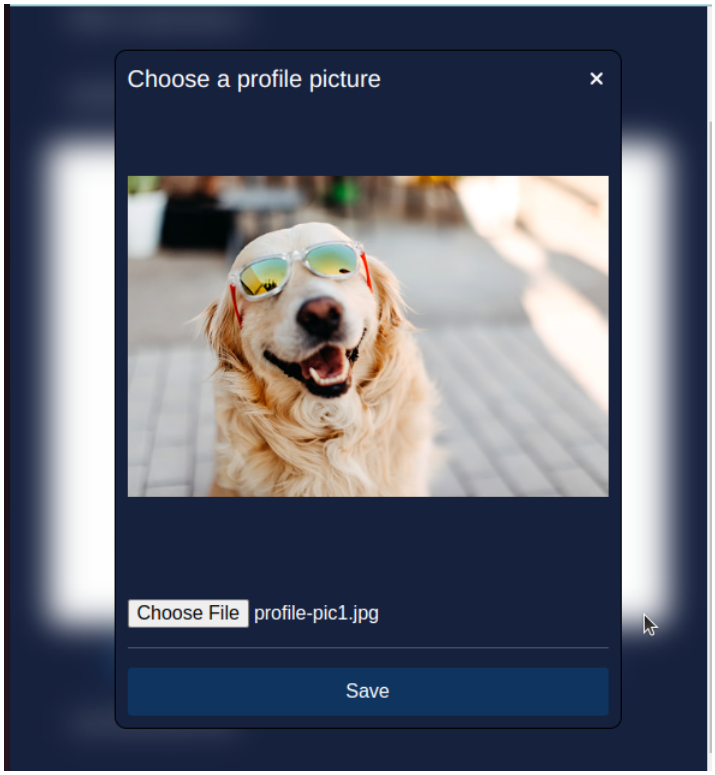


Figura 19: Usuário escolhe uma nova imagem de perfil no *modal*

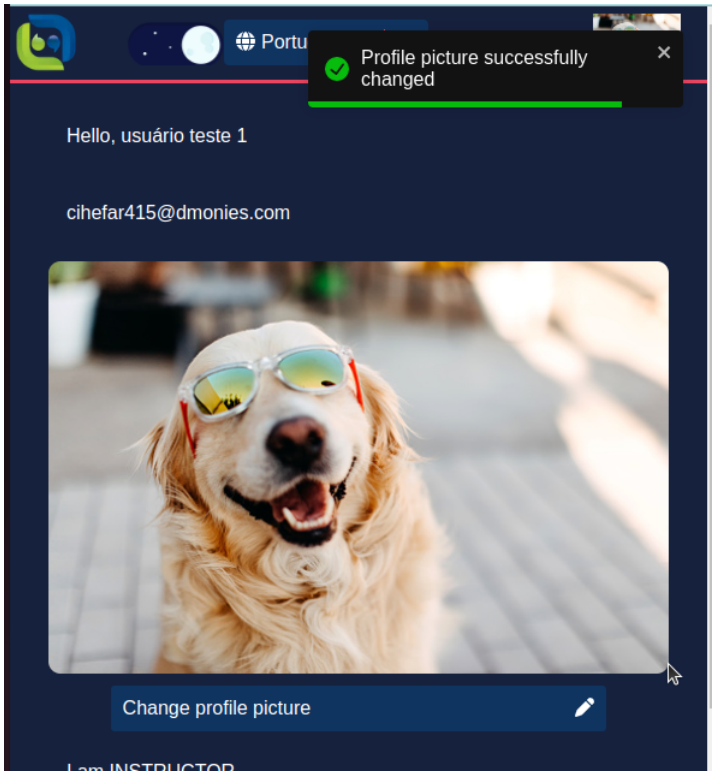


Figura 20: Nova imagem de perfil é salva com sucesso

A imagem é armazenada no serviço de *storage* da AWS, o S3. Para que o navegador sempre exibisse a última foto de perfil, as imagens são armazenadas tendo como nome o id do usuário seguido de um *timestamp* do momento em que a troca de imagem acontece. Sem esse timestamp, todas as imagens de perfil de um mesmo usuário teriam o mesmo nome (e, portanto, a mesma *url*). Isso seria um problema porque o navegador não atualizaria a a imagem renderizada na tela após uma troca, já que a sua *url* não teria mudado. Utilizar uma string aleatória como nome da imagem também seria uma estratégia válida, mas foi utilizado o timestamp do momento da troca.

3.10 Obter usuário

Uma maneira alternativa de lidar com tokens de autenticação que é muito comum na indústria é inserir no token os dados completos do usuário, como nome, imagem de perfil, papel e e-mail. Porém, nesta aplicação, como foi mencionado anteriormente, o token contém apenas a versão da autenticação e o id do usuário. Foi tomada essa decisão para possibilitar que os dados do usuário sejam atualizados sem a necessidade de refazer a autenticação.

Tomemos como exemplo o caso de uso descrito acima, em que o usuário troca sua

imagem de perfil. Se o link para essa imagem estivesse contido no token de autenticação que foi obtido no momento de *login*, o token passaria a estar desatualizado a partir da troca da imagem.

O que se faz nesta aplicação é ter um *endpoint* HTTP GET /users/ na API de autenticação que, a partir do id do usuário contido no token de autenticação, consulta o usuário no banco de dados e retorna suas informações relevantes. Dessa maneira, após uma troca de imagem de perfil, ou atualização de qualquer outro dado, basta recorrer novamente a esse *endpoint*, e teremos as informações do usuário atualizadas no navegador.

3.11 Consulta autenticada à API de domínio

O principal propósito da autenticação é que consultas à API de domínio respondam apenas a usuários devidamente autenticados e autorizados. Para garantir esse propósito, os *endpoints* da API de domínio executam, antes do código do endpoint em si, um código intermediário, chamado *middleware*. Um middleware é um tratamento realizado à requisição antes dela ser respondida. Neste caso, esse tratamento realiza os seguintes passos, com o objetivo de garantir a autenticação e autorização do usuário:

1. Verifica se a requisição tem um *header* HTTP chamado *authorization*
2. Verifica se o valor desse *header* é a string *Bearer* seguida de um espaço seguida de outra string. Essa outra string é tida como o token de autenticação.
3. Realiza leitura do token utilizando o segredo (mencionado em 3.1) garantindo que não houve quebra de integridade.
4. Lê o token para garantir que ele contém id de usuário e versão de token.
5. Obtém o usuário correspondente do banco de dados. Como esse código é executado tanto na API de domínio quanto na API de autenticação (nesta última, para fornecer os dados do usuário para a página, ver 3.10), o id do usuário contido no token deve ser o mesmo nos bancos de dados das duas APIs. Para garantir isso, o evento que a API de autenticação envia para a fila quando um usuário novo é criado contém seu id.
6. Verifica se a versão de token do usuário é igual à versão informada no token. Isso garante que tokens emitidos antes de eventos de *sign out*, como clicar no botão

"Sair"ou trocar a senha estejam inválidos, pois esses eventos terão incrementado a versão do token.

7. Se todas essas validações forem bem-sucedidas, um objeto *user* é adicionado à requisição, e será utilizado por diversos *endpoints* da API de domínio. Muitos *endpoints* precisam saber o e-mail do usuário ou se trata-se de um instrutor ou um estudante, por exemplo.

Abaixo, o código que realiza o que acaba de ser descrito, em Typescript. Note que, caso uma das validações dê errado, um erro é lançado, e a requisição de destino original da API de domínio não é ativada.

```
async (req, headers) => {

    if (!headers.authorization) throw new MissingTokenError();

    const [header, token] = headers.authorization.split(' ');
    if (header !== 'Bearer') throw new MalformedTokenError();

    let tokenPayload;
    try {
      tokenPayload = await tokenService.verify(token);
    } catch(e) {
      console.log("token verification error:",e)
      throw new CouldNotVerifyTokenError();
    }

    if(!Object.keys(tokenPayload).includes('id') ||
      !Object.keys(tokenPayload).includes('tokenVersion') ||
        isNaN(Number(tokenPayload.tokenVersion)))
          throw new InsufficientTokenError();

    const userDTO = await userRepository.getUserById(tokenPayload.id);

    if (!userDTO) throw new UserNotFoundError();

    if (userDTO.tokenVersion !== tokenPayload.tokenVersion) throw new
       Forbidden();

    req.user = {
      id: userDTO.id,
      tokenVersion: userDTO.tokenVersion,
      email: userDTO.email,
```

```
31        name: userDTO.name,
32        role: userDTO.role,
33        image: userDTO.image
34      };
35  }
```

Listing 3.1: Middleware que adiciona user à requisição a partir de um token de autenticação válido

4 FUNCIONALIDADES DA APLICAÇÃO

Nesta seção, são descritas as funcionalidades da aplicação, tudo o que tange o aprendizado de idioma estrangeiro.

4.1 Associar estudante a instrutor

Um estudante pode, em algum momento, se associar a um instrutor. Esse passo não é obrigatório para que o estudante possa realizar as atividades disponíveis na plataforma, mas oferece ao estudante a possibilidade de ter suas realizações de atividades corrigidas por um instrutor. O instrutor pode listar as atividades que seus estudantes realizaram e lhes fornecer feedback.

O fluxo começa com a tela de realizar convite de associação, em que o instrutor insere o e-mail do estudante ao qual gostaria de se associar.

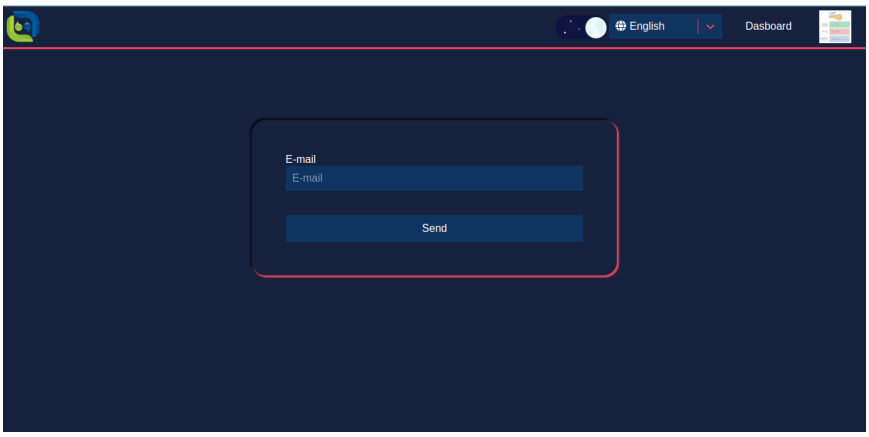


Figura 21: Tela em que o instrutor insere o e-mail do estudante ao qual gostaria de se associar

O estudante recebe, então, um e-mail o convidando a se associar ao instrutor, com um link para uma página com detalhes do instrutor e um botão para aceitar ou recusar o convite de associação.

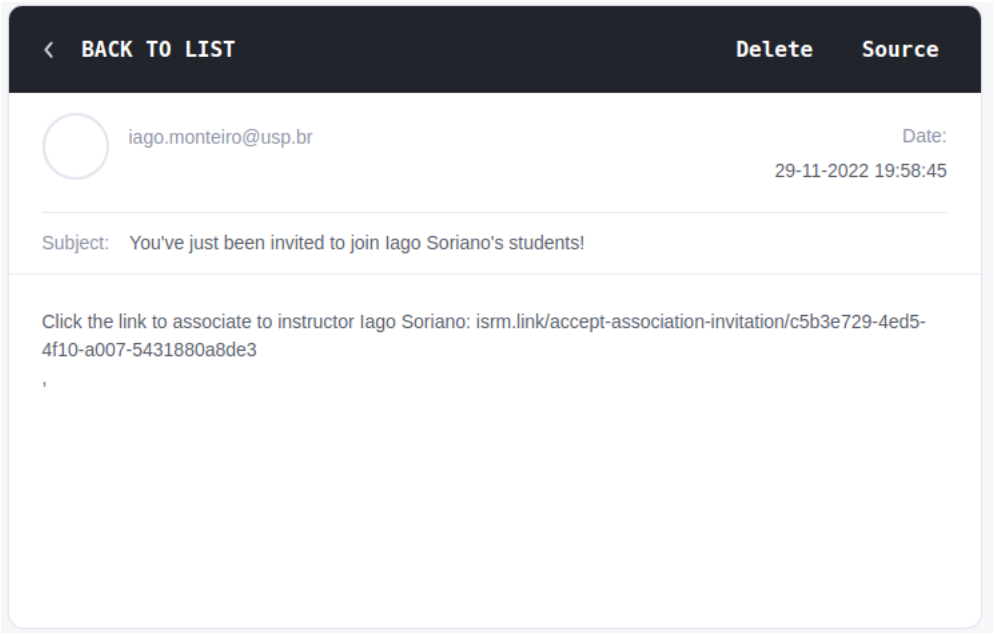


Figura 22: E-mail informando o estudante do convite e contendo a *url* para aceitar a associação

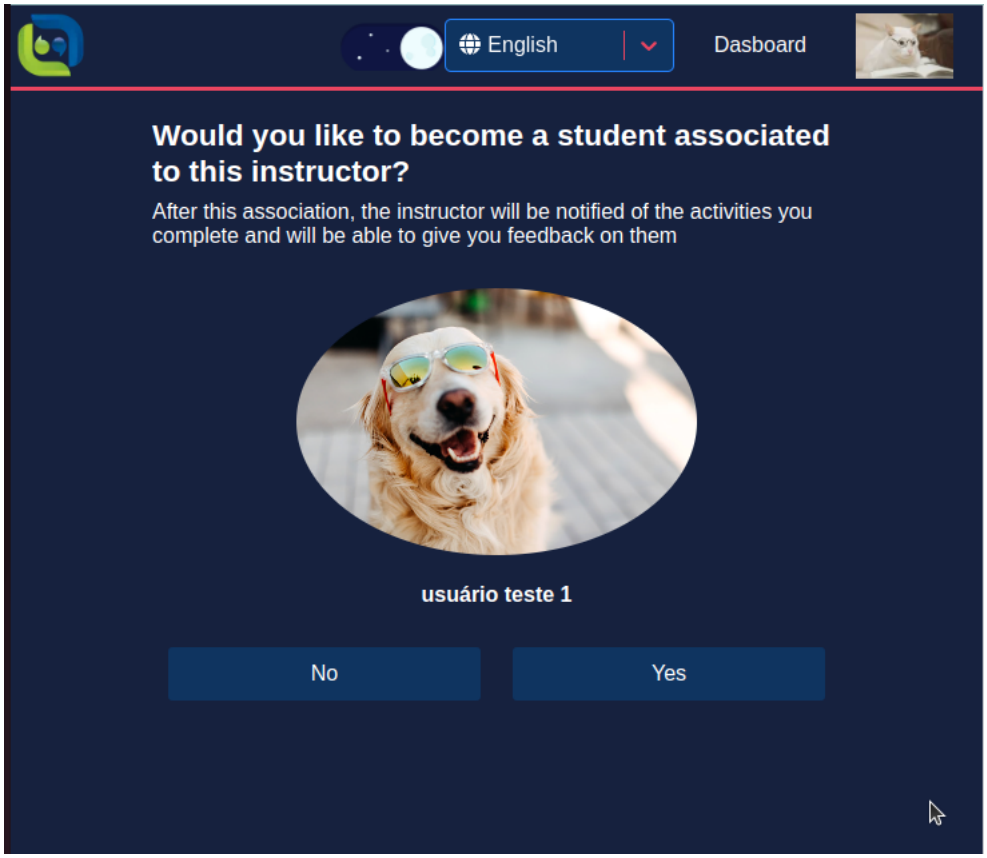


Figura 23: Tela em que o estudante aceita a associação ao instrutor

4.2 Criar nova Atividade

Esta funcionalidade é utilizada apenas por usuários instrutores. Atividades são compostas de três partes:

1. **Detalhes e Descrição**. Esta seção contém título da atividade, tópicos a que a atividade pertence, seu nível CEFR (CEFR,) e uma descrição opcional.

2. **Conteúdo**. Conteúdo linguístico que o estudante irá consumir ao realizar a atividade. Pode ter formato de vídeo ou de texto (em HTML).

3. **Instruções**. Perguntas a respeito do conteúdo. Podem ser perguntas dissertativas ou de múltipla escolha. Ao criar uma instrução do tipo dissertativa, o instrutor deve fornecer uma sugestão de resposta. Já no caso de perguntas de alternativas, o instrutor indica qual (ou quais) alternativas são as corretas.

Em telas grandes (acima de 780px de largura), o usuário pode observar uma pré-visualização da atividade em tempo-real, enquanto preenche as informações que vão compor a atividade. Já em telas pequenas, apenas os campos de inserção de dados são visíveis.

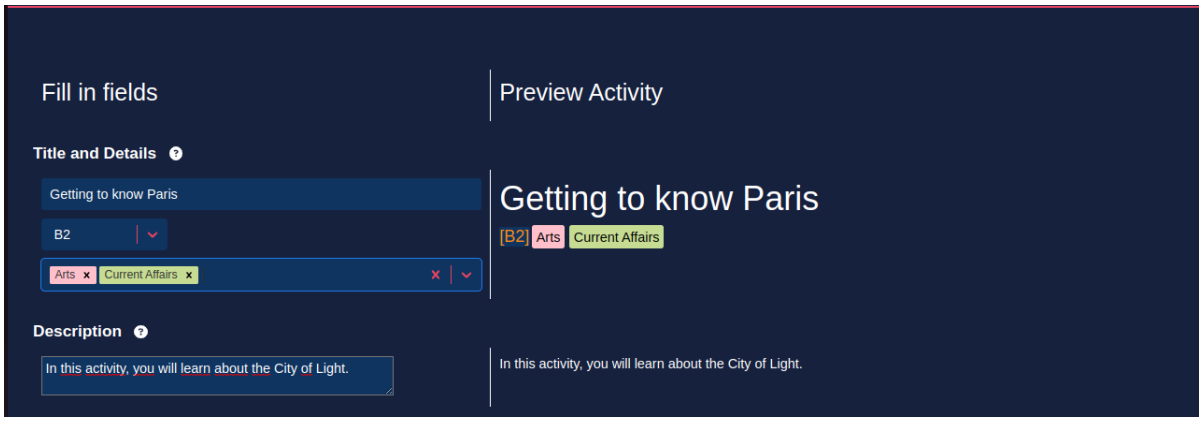


Figura 24: Seção de Detalhes e Descrição com pré-visualização

Para criar atividades de tipo texto, o usuário tem disponível um editor WYSIWYG (What You See Is What You Get), que é um *widget* de uma página web que permite que o usuário produza código HTML sem ter conhecimentos de HTML, utilizando apenas um editor visual. Nesta aplicação, foi utilizado o editor TinyMCE (TINY...,).

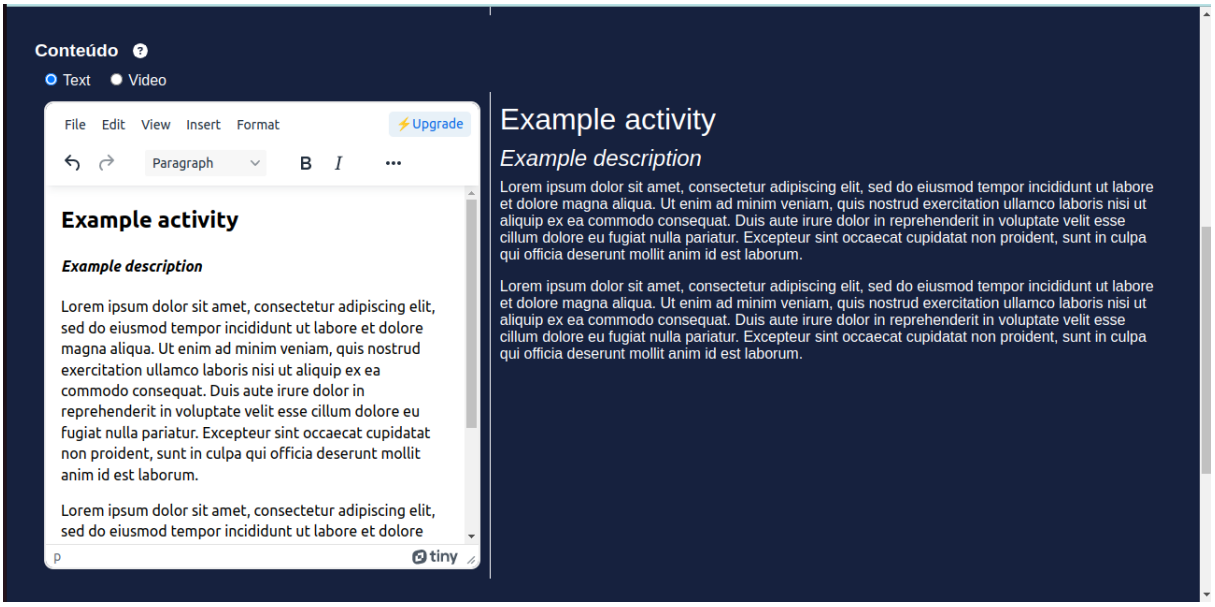


Figura 25: Inserção de um texto como conteúdo da atividade

Para a criação de atividades em vídeo, o usuário insere o id do vídeo do *YouTube*, além das marcações, em segundos, do início e do final da parte do vídeo relevante para a sua atividade pedagógica.

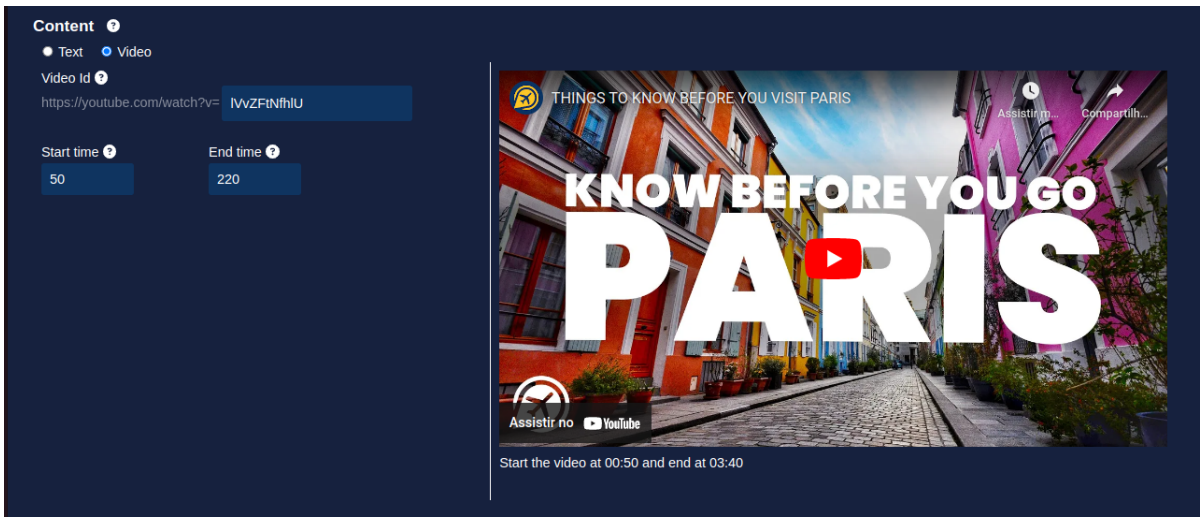


Figura 26: Inserção de um vídeo do YouTube como conteúdo da atividade

Na seção de instruções, o botão "Nova Instrução"abre um *modal* na tela em que pode-se inserir o texto da instrução, uma sugestão de resposta no caso de perguntas dissertativas, e as alternativas, no caso de perguntas de múltipla escolha.

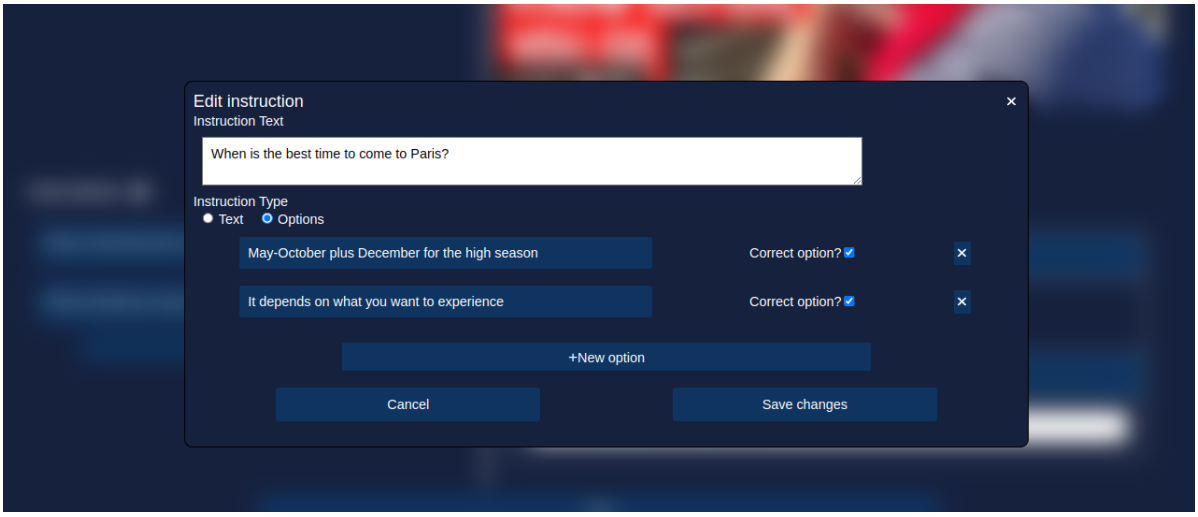


Figura 27: Inserção de nova instrução que tem duas alternativas corretas

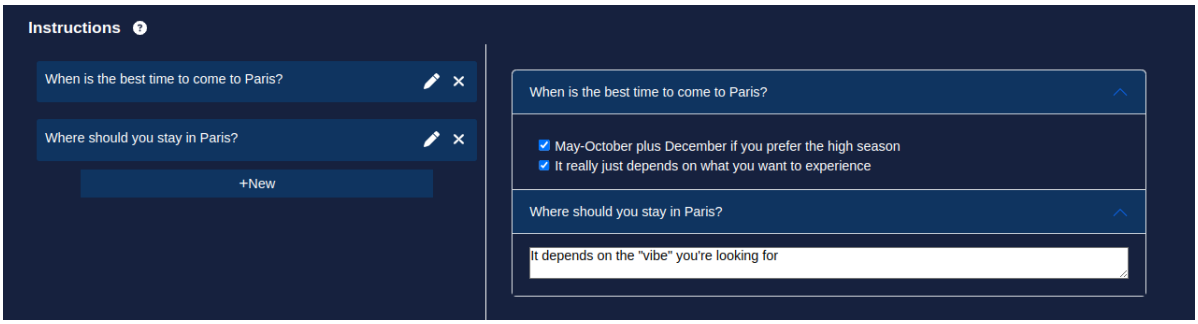


Figura 28: Visualização de todas as instruções já inseridas, com suas respostas indicadas

4.3 Listar atividades

Esta funcionalidade é utilizada tanto por estudantes procurando uma atividade nova para realizar quanto por instrutores que desejam procurar suas próprias atividades ou de outros instrutores.

A tela mostra *cards* com informações das atividades, como título, temas, nível CEFR, descrição e tipo de conteúdo. A busca conta também com filtros para cada um desses valores.

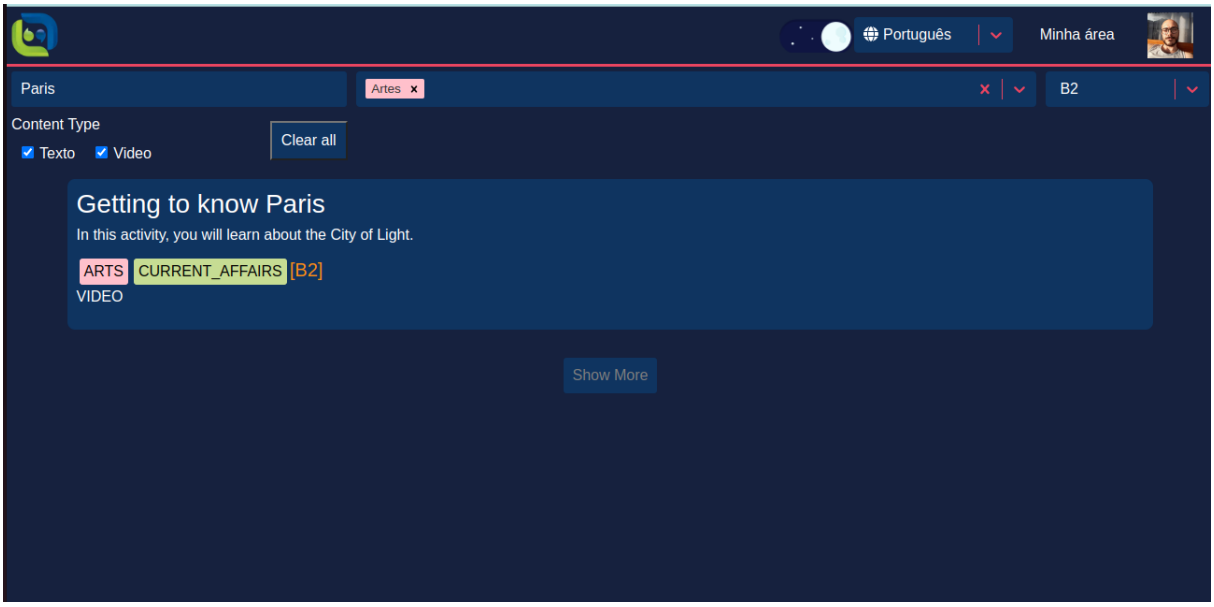
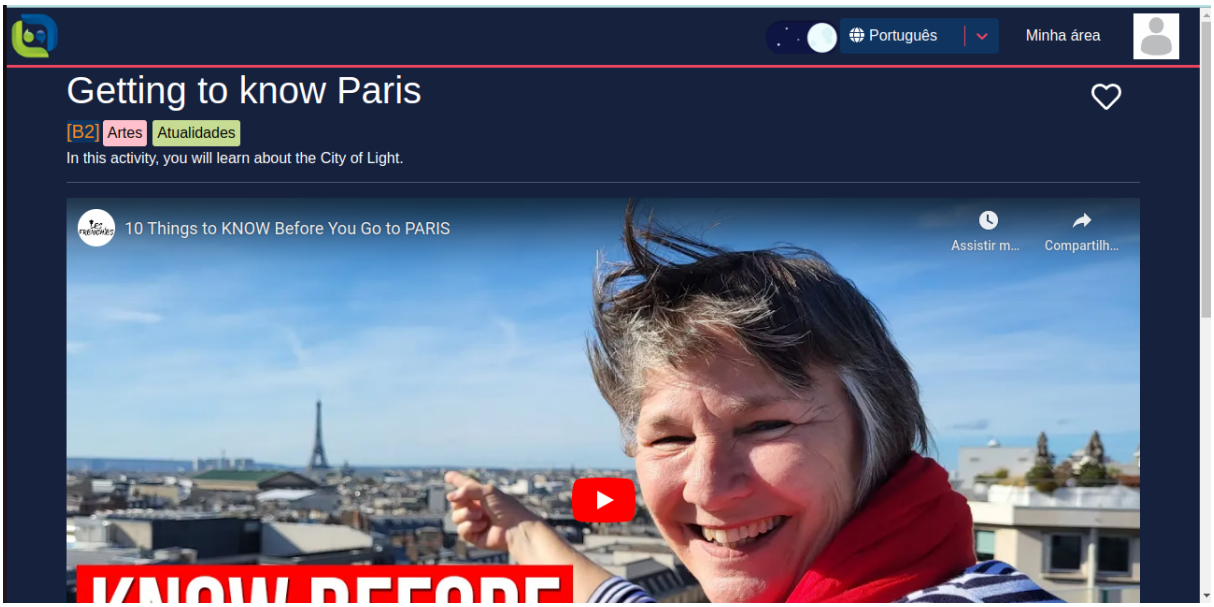
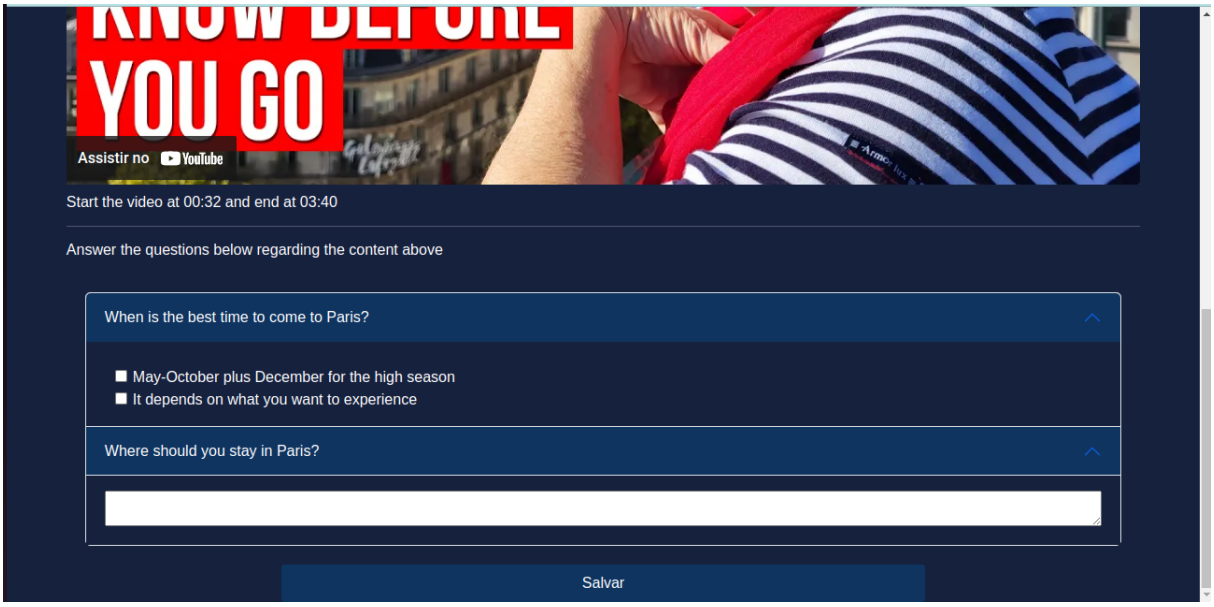


Figura 29: Visualização das atividades disponíveis e *inputs* dos filtros

4.4 Realizar Atividade

Nesta tela, um estudante pode realizar uma atividade. A tela é dividida em três seções. A primeira conta com os detalhes da atividade; a segunda, com o conteúdo; e a terceira, com as instruções a serem respondidas. Uma vez respondidas as perguntas, o estudante clica no botão "Salvar", e as suas respostas são enviadas para a API de domínio e persistidas.





4.5 Listar Atividades Realizadas

Nesta tela, estudantes podem escolher uma realização de atividade. Pode haver várias realizações de atividade para uma mesma atividade, pois o estudante pode escolher realizar a mesma atividade mais de uma vez. No canto superior direito dos *cards* de cada atividade realizada, há um ícone que indica se a atividade já recebeu feedback (um v) ou não (uma ampulheta)

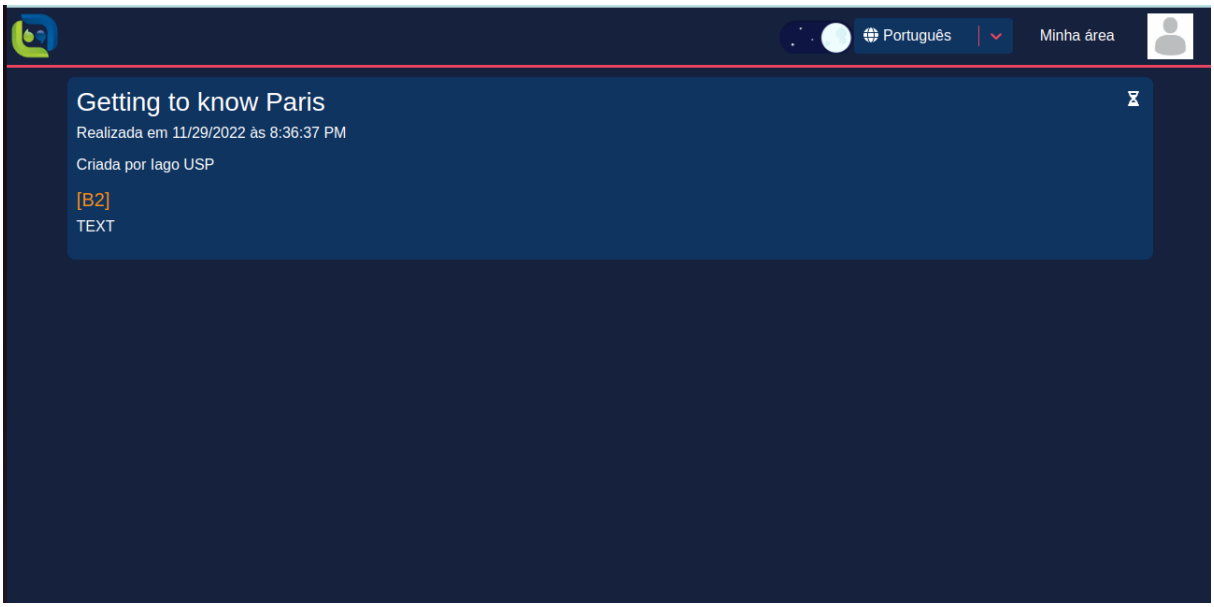


Figura 30: Visualização das atividades atividades realizadas pelo estudante

Instrutores utilizam esta tela para escolher uma realização de atividade de um de seus

estudantes e lhe fornecer feedback.

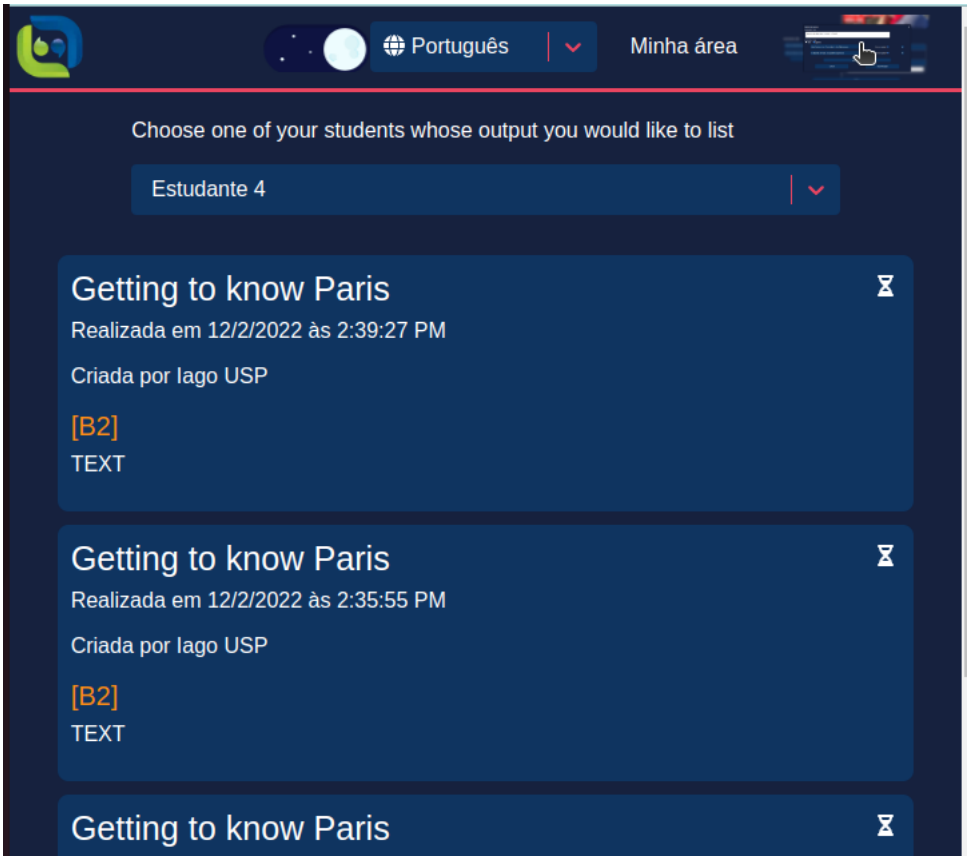


Figura 31: *dropdown* permite que instrutor selecione de qual dentre seus estudantes gostaria de listar as atividades realizadas

4.6 Fornecer e Visualizar Feedback a Atividade Realizada

Ao clicar em um dos cards descritos na listagem de realização de atividade, a aplicação nos redireciona para a página de visualização de realização de atividade. Esta página é utilizada por estudantes para visualizar o feedback dado a cada uma das perguntas que respondeu, e por instrutores para fornecer o feedback.

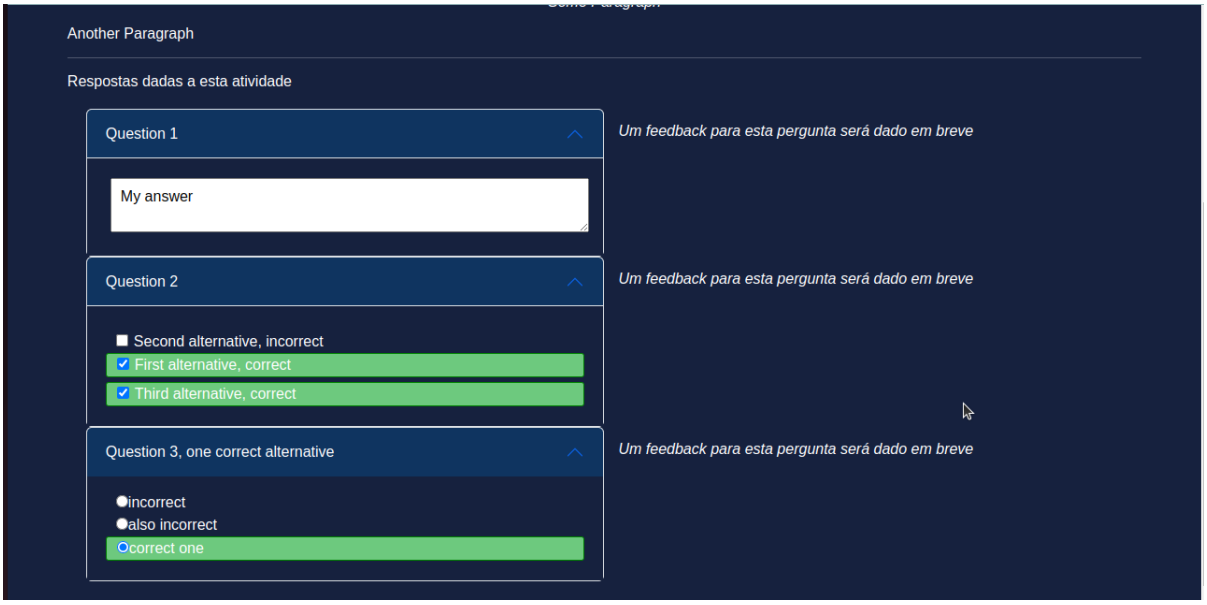


Figura 32: Estudante vê que ainda não foi dado feedback para a sua produção

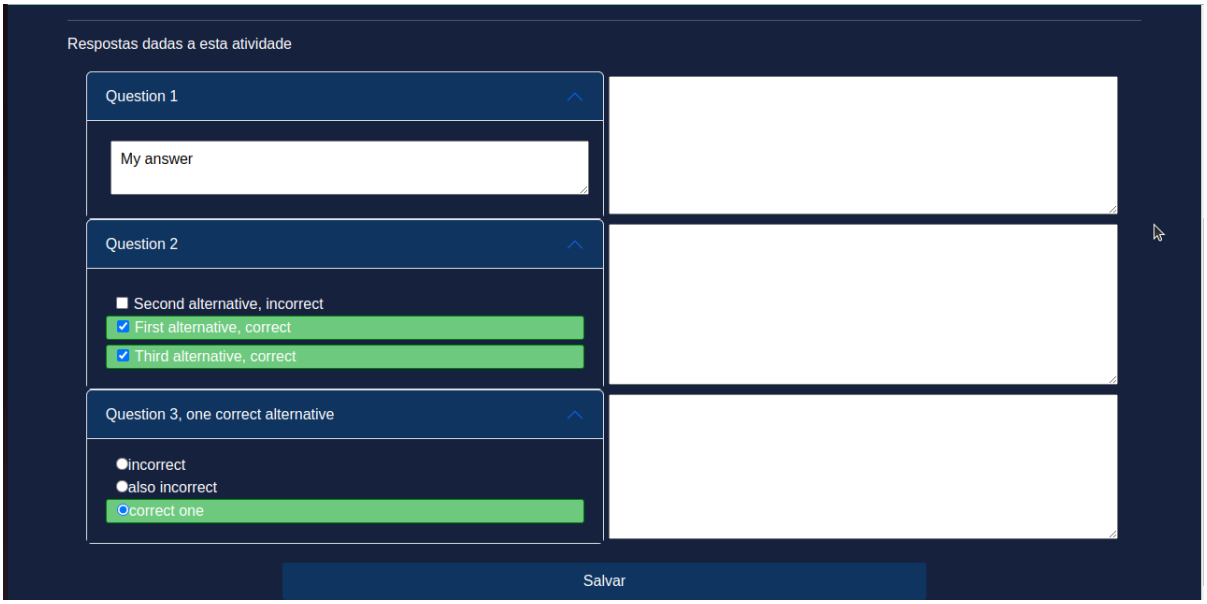


Figura 33: Instrutor vê a produção do estudante e pode escrever um feedback para cada resposta

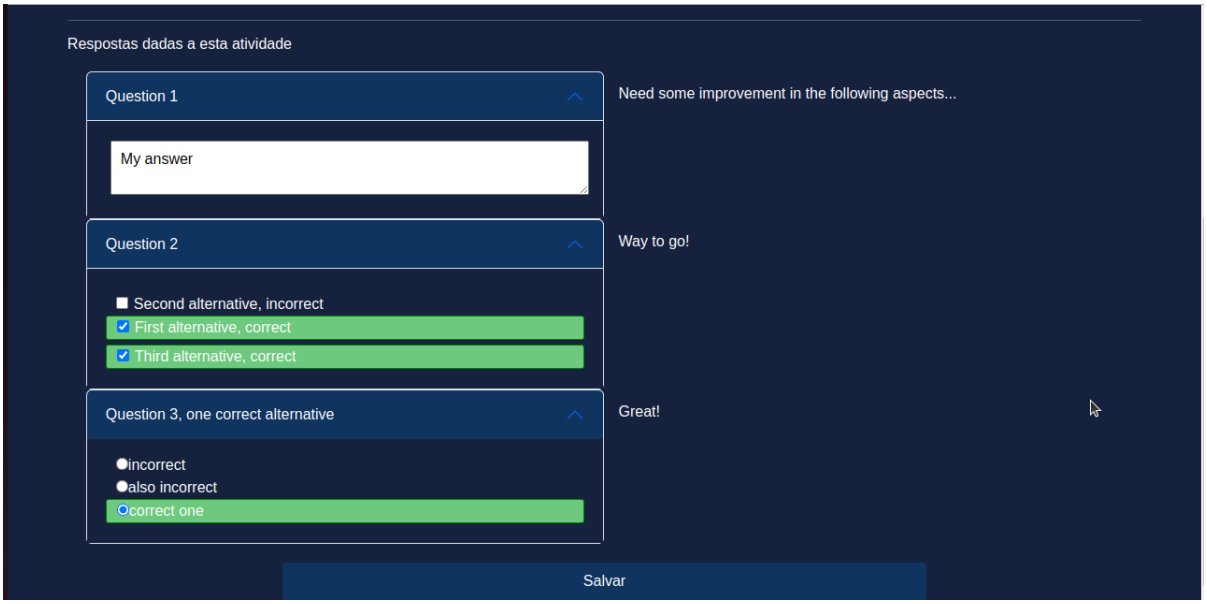


Figura 34: Estudante pode visualizar feedback que recebeu

5 CONCEITOS E TÉCNICAS

Neste capítulo, estão descritas alguns conceitos de engenharia de software e algumas tecnologias utilizadas neste trabalho, selecionadas com o critério de serem interessantes e relevantes para o estado atual da indústria de desenvolvimento web.

5.1 Experiência de Desenvolvimento

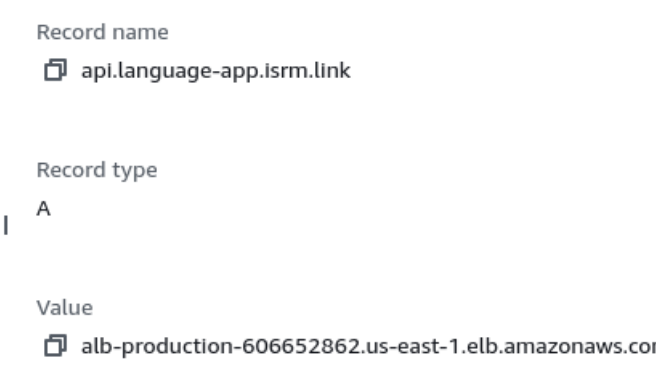
5.1.1 Ambientes de Produção e Pré-Produção

Em aplicações modernas, em que a entrega rápida e seguras de novas funcionalidades é uma requisito importante para a experiência de desenvolvimento, é comum ter mais de um ambiente em que as aplicações são executadas.

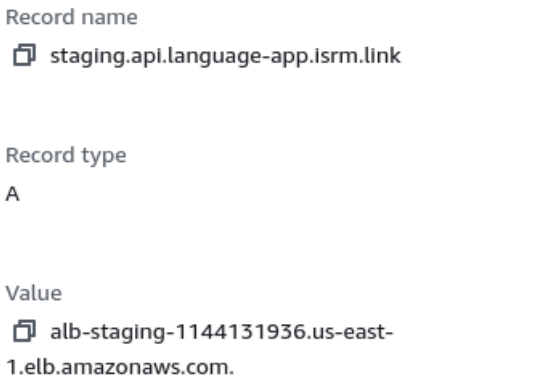
Um ambiente é o de **Produção**, em que se executa os programas com os quais os usuários interagem, e onde se persistem os dados desses usuários. Outro ambiente é o de **Pré-Produção**, acessado apenas pelo time de desenvolvimento, e serve como ambiente de testes da aplicação como um todo antes de disponibilizá-la aos usuários. O objetivo de se ter múltiplos ambientes de entrega de código é garantir que a versão do código e da infraestrutura que chega ao ambiente de Produção é a mais estável e correta possível: uma versão da aplicação que fora amplamente testada e cujo bom funcionamento conta com alto nível de confiança do time de desenvolvimento.

Abaixo, o registro de DNS que direciona o nome de domínio *staging.api.language-app.isrm.link* para o Balanceador de Carga do ambiente de pré-produção, chamado de *staging*, e o nome de domínio *api.language-app.isrm.link* sendo direcionado para o ambiente de produção.

No *frontend*, há dois sites: um de domínio staging.language-app.isrm.link, e outro de nome language-app.isrm.link. Aquele prefixado por *staging* é o *frontend* do ambiente de pré-produção.



(a) Registro de DNS que aponta para o ambiente de Produção



(b) Registro de DNS que aponta para o ambiente de Staging

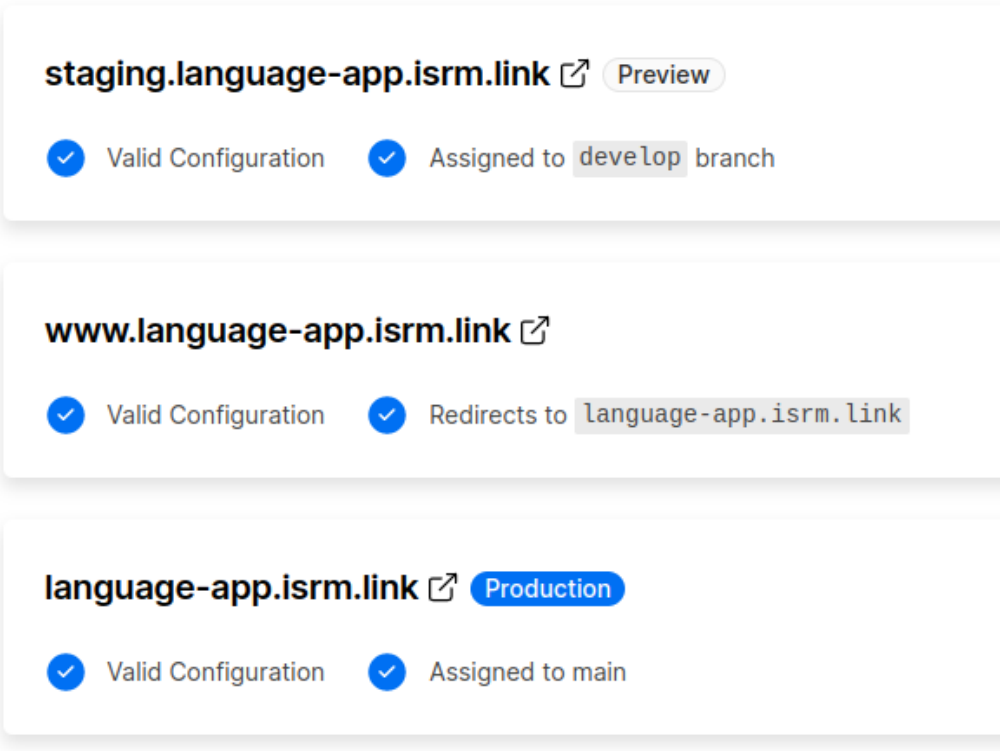


Figura 36: Site da Vercel, empresa que faz *hosting* de aplicações NextJS, em que vê-se os dois domínios: cada um associado a uma ramificação diferente do repositório remoto de código. Para mais detalhes, ver seção sobre CICD

5.1.2 Bastion Host

Um Bastion Host é um servidor acessível apenas para os desenvolvedores da aplicação, e pode ser comunicar diretamente com as instâncias de bancos de dados e com as aplicações. (BASTION...,)

Neste trabalho, foi criada uma instância EC2, uma máquina virtual para propósito geral e um dos serviços mais antigos da AWS, dentro da VPC em que o banco de dados

e as aplicações se encontram. Essa máquina virtual conta com um grupo de segurança, abstração da AWS análoga a um *firewall*, cujas regras permitem acesso via SSH pelos IPs especificados no código de configuração da infraestrutura (ver 5.3 sobre IaaC). Os desenvolvedores da aplicação podem permitir seus IPs para acessar o *bastion host* por SSH e, com isso, acessar, ainda por SSH, as instâncias dos bancos de dados. O script *bastion-ssh.sh* acessa o bastion host por SSH, utilizando seu nome de domínio, fornecido pelo Terraform.

Esse recurso foi amplamente usado durante o desenvolvimento da aplicação, para que fosse possível ver e modificar diretamente os dados presentes nos bancos de dados. Durante o desenvolvimento, alguns *bash scripts* foram criados e mantidos na área de trabalho do bastion host para auxiliar o acesso ao banco de dados, além do código da aplicação.

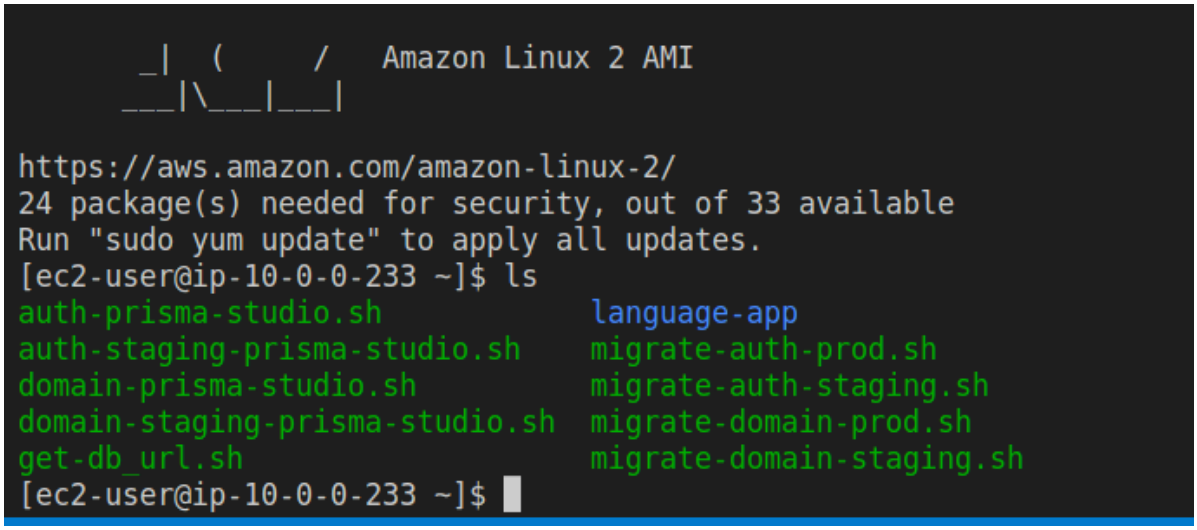


Figura 37: *scripts* na área de trabalho do *bastion host*

Há quatro scripts que terminam em *prisma-studio*, sendo um para cada aplicação para cada ambiente. Eles entram na pasta da aplicação em questão e atualizam o valor da variável de ambiente *DATABASE_URL* para a *url* do banco de dados que aquela aplicação utiliza, e executam o comando *npx prisma studio*, que é um comando pertencente à CLI do *Prisma* (PRISMA,), um *Object Relational Mapper* utilizado pelo código *backend* deste projeto para se comunicar com o banco de dados. Este comando executa um servidor no *bastion host* que retorna uma interface com a qual se interage diretamente com os dados do banco de dados em questão.

```
[ec2-user@ip-10-0-0-233 ~]$ cat auth-prisma-studio.sh
#!/bin/bash

cd language-app/packages/auth-web-api

export DATABASE_URL=postgres://postgres:supersecretpassword@main.cc0
qmkg5rwaj.us-east-1.rds.amazonaws.com:5432/auth-web-api

npx prisma studio
[ec2-user@ip-10-0-0-233 ~]$
```

Figura 38: *script* auth-prisma-studio.sh, que abre a interface do Prisma para se interagir com o banco de dados de produção da aplicação de autenticação através do navegador

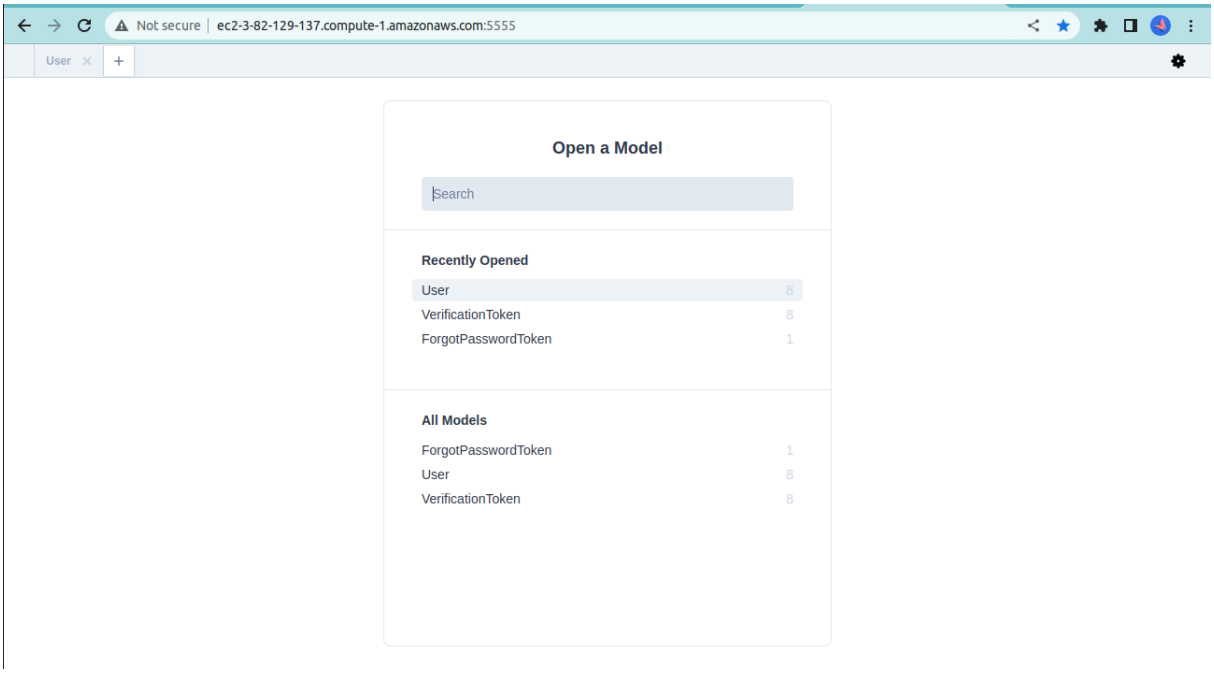


Figura 39: Interface do Prisma para se interagir com o banco de dados de produção da aplicação de autenticação

Já os quatro scripts iniciados em *migrate* realizam a migração do banco de dados em questão. O script atualiza o código do *bastion host* para a última versão da ramificação de desenvolvimento do repositório remoto, para obter as últimas atualizações do *schema* do banco de dados, e executa o comando *db:migrate:prod*, definido no *package.json* (arquivo de configuração NodeJS) de cada aplicação. Este comando simplesmente invoca o comando CLI do Prisma que atualiza o banco de dados com o *schema* descrito no arquivo de *schema* do projeto.

```
[ec2-user@ip-10-0-0-233 ~]$ cat migrate-auth-staging.sh
cd language-app/packages/auth-web-api

git pull origin develop

export DATABASE_URL=postgres://postgres:supersecretpassword@main.cc0
qmkg5rwaj.us-east-1.rds.amazonaws.com:5432/auth-web-api-staging

npm run db:migrate:prod
[ec2-user@ip-10-0-0-233 ~]$
```

Figura 40: Conteúdo do *script* que realiza a migração do banco de dados da aplicação de autenticação no ambiente de *staging*

5.1.3 Pipelines de entrega de código: CICD

CICD significa *Continuous Integration, Continuous Delivery*, e trata-se de uma configuração de recursos computacionais em nuvem que permitem a rápida integração de novo código à ramificação principal do repositório (*branch main* do *Github*) garantindo-se que este códigonovo siga devidamente as convenções estabelecidas pelo time, e a posterior entrega do código novo ao usuário, no ambiente de produção.

O pipeline de Entrega Contínua (CICD) adotado neste projeto é composto dos seguintes passos:

1. O controle de versão utiliza os conceitos do Git Lab Workflow (GIT...,). Uma nova funcionalidade, ou conserto de erro, ou refatoração, daqui em diante chamado de *entregável*, é implementado em uma ramificação que parte da branch develop (chamada *branch Feature*) e é nomeada de maneira adequada, como *feature/sales-page* ou *bugfix/homepage*.

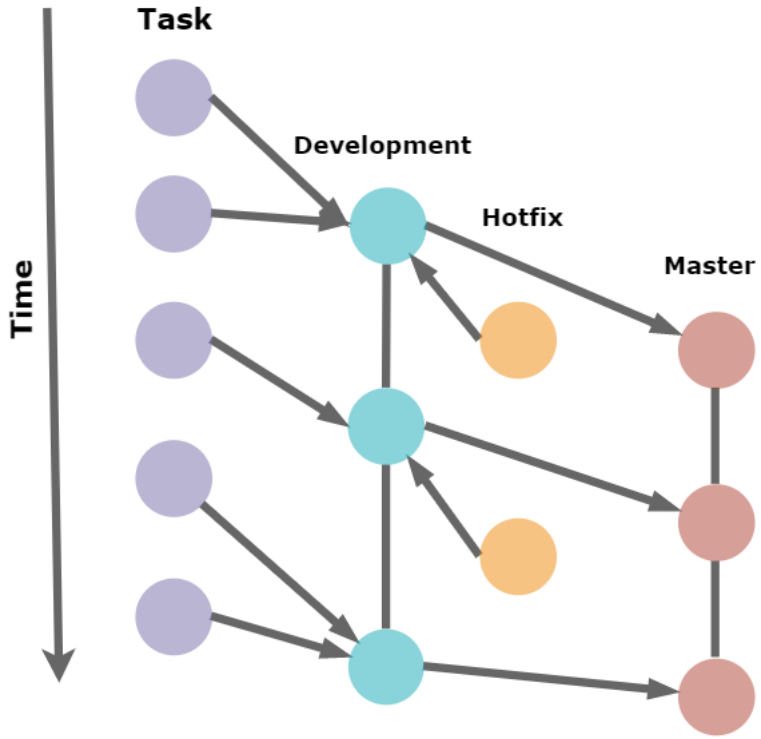


Figura 41: Representação visual do Gitlab Workflow (THE...,)

2. Cada vez que a branch Feature recebe uma nova versão (um novo *commit*), uma série de processos automáticos são desencadeados, como checagem de convenções de estilo de código (*linting*, com ESLint (ESLINT,)) e de formatação (com Prettier (PRETTIER,)), e execução dos testes automatizados. Neste projeto, foram usadas as *Github Actions* (GITHUB...,) para esses processos.

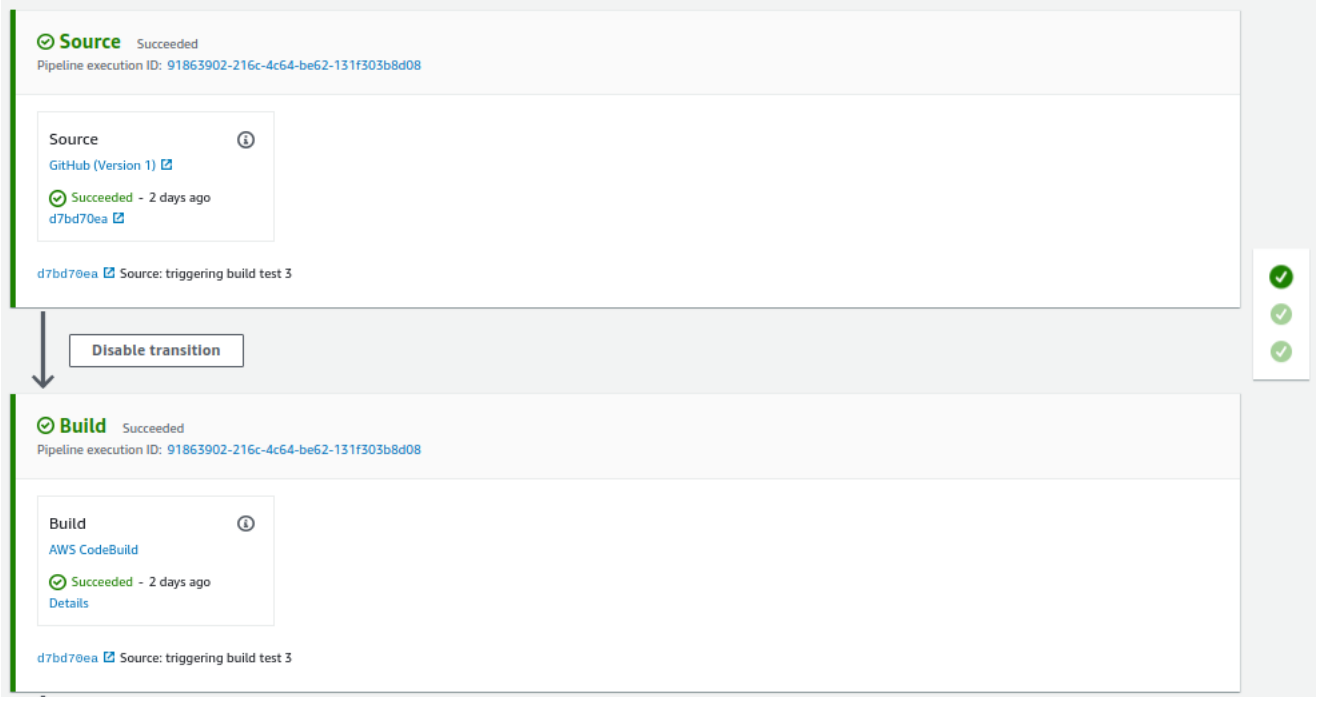
3. Assim que o entregável está pronto, o dono da branch Feature abre um *Pull Request*, ou um pedido para integrar seu código ao código principal. Esse pedido é avaliado pelos colegas. Essa fase de avaliação é chamada de *Code Review* e é iterativa: o entregável pode sofrer alterações até ser aceito à base de código principal. Isso garante que todo o time de desenvolvimento está ciente do código que está sendo entregue e ajuda a manter os padrões adotados previamente intactos. O dono do projeto, então, aprova a integração do entregável. Neste projeto, como fora desenvolvido por apenas uma pessoa, absteve-se desse passo de revisão. Uma vez terminado o processo de revisão, a branch Feature é integrada à branch develop e o pipeline da AWS é ativado.

4. O serviço da AWS CodePipeline "escuta"por novos *commits* na branch develop e inicia seus processos quando isso acontece. Aqui, há quatro pipelines: um para o

ambiente de pré-produção e outro para o ambiente de produção para cada um dos dois serviços. O do ambiente de pré-produção escuta por commits na develop e realiza dois passos:

(a) Constrói a imagem Docker a partir do Dockerfile existente na pasta de cada aplicação e atualiza a imagem que consta como a mais atual no repositório de imagens da AWS, o ECR.

(b) Notifica o ECS para baixar uma nova imagem e executar um novo container do serviço em questão.



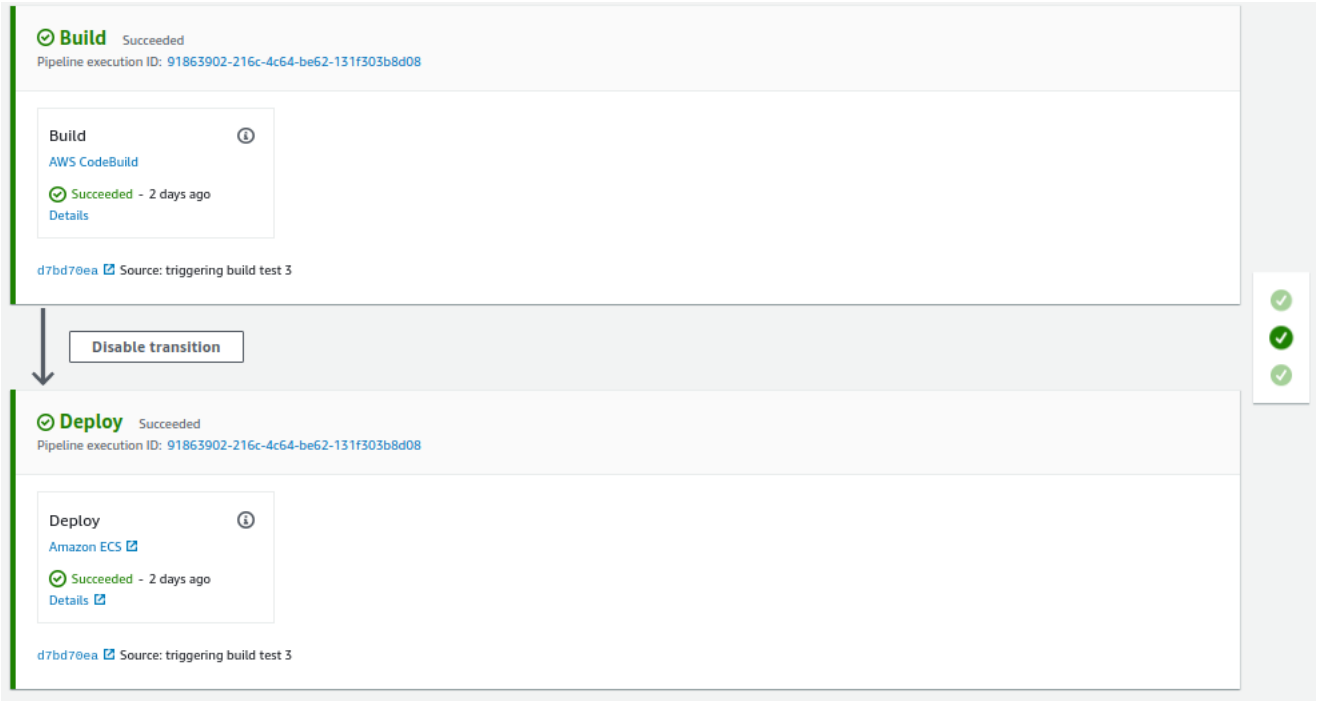


Figura 42: Pipeline de entrega para o ambiende de *staging*

5. Uma vez que as aplicações estão no ambiente de pré-produção, elas são testadas por um time de QA (*Quality Assurance*) e testes *end-to-end* são executados, manual ou automaticamente. Uma vez terminado esse processo, o gerente do projeto integra a branch *develop* à branch *main*. Essa integração ativa o pipeline de produção e é, também, o CD do CICD: entrega contínua, pois o código novo está sendo entregue ao usuário, no ambiente de produção, depois de ter sido integrado à base de código. O pipeline de produção realiza apenas o análogo do passo 4b para o ECS do ambiente de produção, pois a imagem já foi construída em 4a.

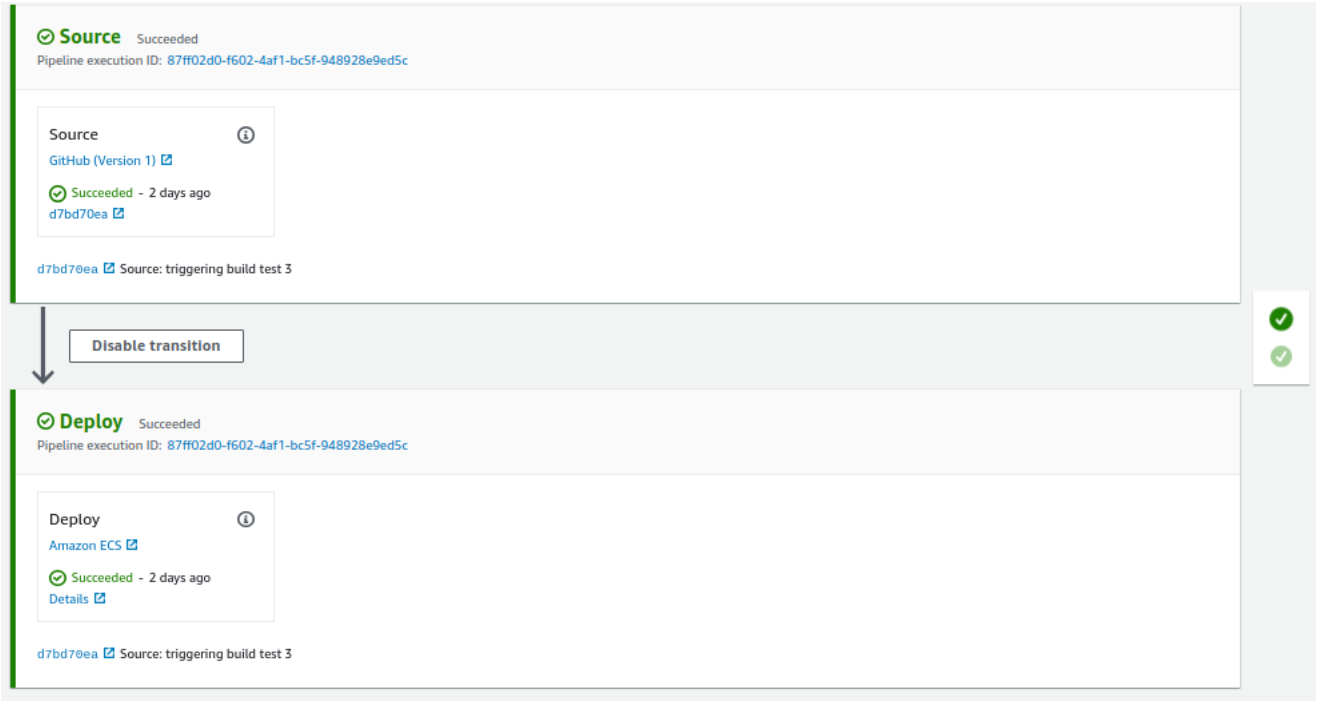


Figura 43: Pipeline de entrega para o ambiende de produção

5.2 Arquitetura Limpa

A Arquitetura Limpae trata de um padrão de design de software, como o MVC e a arquitetura em três camadas, criado por Robert Martin.

Recomenda-se ao leitor explorar os códigos em *packages/auth-web-api* e *packages/web-api*, que são os códigos *backend* desta aplicação, assim como suas dependências em *packages/common-platform*, o código de plataforma, para obter uma compreensão mais profunda de como a Arquitetura Limpa é aplicada neste projeto.

5.2.1 Visão Geral

A Arquitetura Limpa é um conjunto de restrições sobre o design do software. Segundo o autor desta abordagem, mais restrições sobre o programador induzem a código de maior qualidade (MARTIN, 2017).

O diagrama representado abaixo ilustra como as camadas da arquitetura devem se relacionar entre si. As setas pretas indo de camadas mais externas para as mais internas são as mesmas setas dos diagramas UML, e indicam dependência.

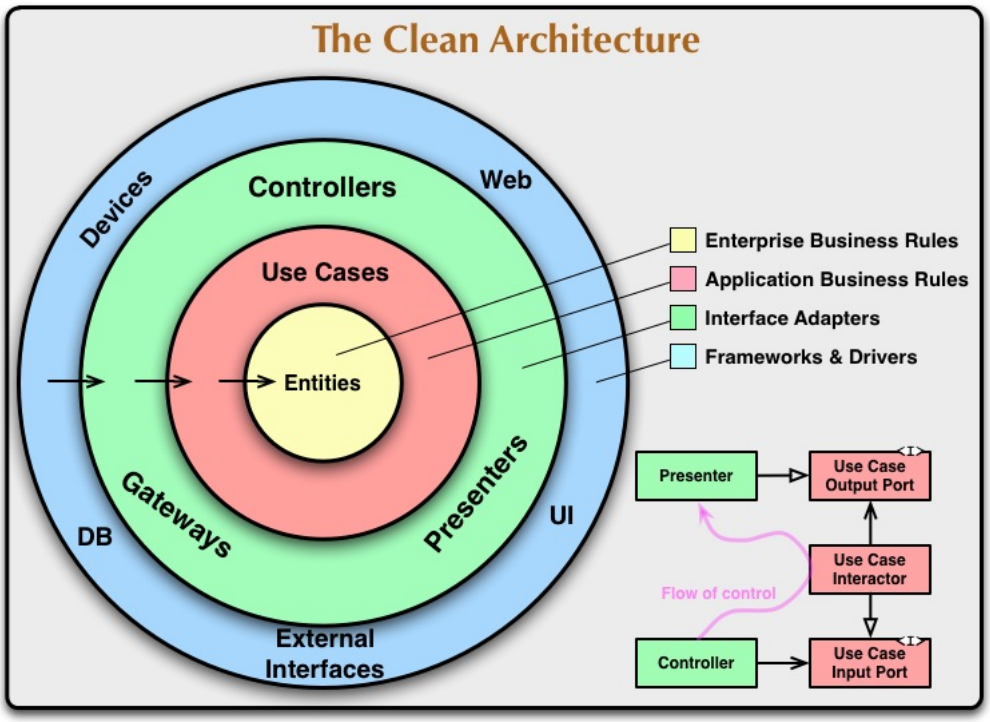


Figura 44: Diagrama original da Arquitetura Limpa (MARTIN, 2012)

Encontram-se na internet diversas maneiras, diversas variações e interpretações para se implementar a Arquitetura Limpa. Nesta seção é descrita a maneira como a implementação acontece no código *backend* desta aplicação, além da teoria original criada por Robert Martin.

5.2.2 Inversão de Dependência

Dentre os conceitos da Arquitetura Limpa, um dos mais importantes é o da Inversão de Dependências. No diagrama acima, esse princípio está representado pelas setas que apontam das camadas exteriores para as camadas interiores. Isso significa que código das camadas externas só pode depender de código de camadas internas. Preferencialmente, apenas do código da camada mais interna imediatamente adjacente. Camadas mais internas nunca devem mencionar (ou depender de) código de camadas mais externas.

A vantagem desta restrição é garantir que o código das camadas mais internas, de alto nível de abstração, onde as regras de negócio são executadas, não será alterado quando novas decisões de baixo nível de abstração, involvendo tecnologias, como bibliotecas HTTP ou de acesso a bancos de dados, são tomadas. Isso garante, também, que mudanças na escolha das tecnologias terá um impacto limitado no software como um todo, não se propagando para todas as classes do sistema, pois as tecnologias específicas restringem

sua presença às camadas mais externas.

5.2.3 Camada de Domínio, ou *Entities*

Na camada de Entities, chamada de domain neste trabalho, estão definidas as classes das entidades do negócio. Essas classes descrevem as propriedades de cada entidade e os métodos que realizam ações sobre essas entidades, além de métodos que realizam validações dessas propriedades. O código desta camada é chamado de uma abstração de alto nível: ele não tem quaisquer dependências externas, como bancos de dados ou APIs externas.

5.2.4 Camada de Casos de Uso, ou *Application*

A camada de aplicação, application ou Use Cases, é onde as entidades definidas e implementadas no *domain* são manipuladas para alterar o estado da aplicação e realizar os casos de uso. Os nomes application e use cases são apropriados, pois o código desta camada faz o que a aplicação deve fazer: implementar os casos de uso.

5.2.5 Camada de Adaptadores de Interface

A camada de adapters contém código que conecta os frameworks às necessidades específicas da aplicação. Por exemplo, os casos de uso necessitam de classes de repositório para persistir e alterar o estado da aplicação. Então, a camada de aplicação define interfaces que atendem às suas necessidades de negócio. O código da camada de adapters importa essas interfaces e as implementa em classes concretas. Essas classes concretas precisam de código de bibliotecas externas para se conectar ao banco de dados. Para se manter "limpa"de dependências externas, a camada de adaptadores vai, por sua vez, definir outras interfaces, que o código da camada de frameworks deve implementar em classes concretas que, elas sim, se comunicam com o banco de dados.

A camada de adaptadores é uma herança da arquitetura de Ports and Adapters, ou Hexagonal Architecture (COCKBURN, 2005). O objetivo dos adaptadores, então, é transformar a interface de uma classe (neste caso, a interface de uma dependência externa que se comunica com o banco de dados) em uma interface que outra classe espera (neste caso, a interface definida pela camada de use cases).

5.2.6 Camada de Frameworks

Todo o código que menciona bibliotecas, *frameworks* e softwares externos, ou seja, as dependências do projeto, encontra-se na pasta frameworks. O objetivo dessa restrição é manter o "coração da aplicação", nas pastas application e domain, puros e facilmente adaptáveis a mudanças de frameworks. Assim, se surgir a necessidade de alterar o banco de dados, o ORM, o framework HTTP, a arquitetura de API, ou qualquer outro detalhe externo às regras de negócio, essas mudanças se concentrarão nesta pasta, nesta camada de *frameworks*, e em mais nenhuma outra região do projeto.

Neste projeto, há três classes de dependências externas: bancos de dados, servidores HTTP e serviços externos.

A pasta de bancos de dados contém código de dependências que se conectam a bancos de dados, como ORMs, ou bancos de dados em memória. Todas seguem a interface definida pela camada de adaptadores.

A camada de servidor HTTP define código de frameworks para servidores HTTP. No ambiente NodeJS, o exemplo mais comum desses frameworks é o ExpressJS (EXPRESS,). Esta camada importa os controladores HTTP definidos na camada de adaptadores e adapta sua sintaxe para a sintaxe específica do Express. Se surgisse a necessidade de expor esta aplicação sob a forma de uma aplicação desktop, com interface de terminal, ao invés de através de um servidor web, uma nova pasta surgiria ali utilizando novas dependências externas, e novos adaptadores surgiriam na pasta adapters.

5.3 Infraestrutura como Código (IaaC)

Aplicações complexas que fazem uso de microsserviços têm diversos componentes computacionais diferentes, como máquinas virtuais, bancos de dados, soluções de persistência de arquivos, redes privadas, *proxies* (todas esses componentes estão presentes na arquitetura deste trabalho), e cada um desses componentes vem acompanhado de um certo número de parâmetros de configuração que o fazem funcionar da maneira desejada e em conjunto com os outros componentes do ambiente/arquitetura.

Uma maneira de configurar (ou provisionar) todos esses componentes é utilizar o console do provedor de nuvem (neste trabalho, a AWS, mas outros exemplos famosos seriam o Google Cloud Platform e a Azure), uma interface de um website em que se inserem os parâmetros de configuração e se escolhem os componentes através de botões e

listas.

Esta é uma estratégia válida para o provisionamento de infraestruturas pequenas, que não envolvem o trabalho de muitos desenvolvedores e que não mudam com frequência. Para todos os outros casos, contudo, essa estratégia torna as tarefas de manutenção e portabilidade da infraestrutura muito difíceis. Por isso, é uma boa prática (e, muitas vezes, uma prática essencial) utilizar infraestrutura como código, ao invés de provisionar infraestrutura utilizando interfaces gráficas em websites.

Infraestrutura como código é uma prática moderna de manutenção de infraestruturas computacionais na nuvem. Quando se utiliza infraestrutura como código, todos os componentes da infraestrutura são representados em um arquivo de configuração, escrito em formatos JSON ou YAML, e todas as suas configurações ficam ali especificadas. Uma grande vantagem dessa prática é a **facilidade de compreensão da infraestrutura**. É mais simples de se compreender quais são os componentes que formam uma infraestrutura na nuvem e como eles se relacionam entre si ao se ler um arquivo de configuração destes, ao invés da alternativa, que seria navegar pela interface do console do provedor de nuvem, procurando cada informação que se deseja.

Havia, para este trabalho, duas principais alternativas no mercado para IaaC. Primeiramente, o CodeDeploy, solução mantida pela própria AWS, e pode ser utilizada apenas para o provisionamento de infraestrutura naquele provedor de nuvem. A outra alternativa, a que foi escolhida aqui, é o Terraform (TERRAFORM,). Trata-se de uma solução de infraestrutura como código em que é possível, com o mesmo código, provisionar infraestrutura para mais de um provedor de nuvem diferente, como AWS, GCP e Azure. Foi escolhido o Terraform devido à sua versatilidade e popularidade no mercado.

O Terraform permite o agrupamento lógico de conjuntos de componentes da infraestrutura em módulos. Na imagem abaixo, vê-se uma parte da estrutura de pastas do projeto, que mostram quais módulos foram escolhidos para organizar a arquitetura deste projeto.

Figura 45

- **bastion** agrupa a configuração do *bastion host*, incluindo seu sistema operacional e suas configurações de firewall (grupos de segurança da AWS).
- **cicd** contém as configurações dos pipelines de CICD, incluindo recursos da AWS como *CodePipeline*, *CodeBuild* e o repositório remoto de imagens de containers, *ECR (Elastic Container Registry)*. Esta pasta inclui também as configurações de autorização para cada recurso (*AWS IAM)*.
- **db** mantém as configurações do banco de dados Postgre na RDS, como as configurações da instância, da máquina virtual e do grupo de segurança, assim como as configurações de rede privada da VPC particulares ao banco de dados.
- **ecs** agrupa as configurações do *cluster ECS*, aquelas pertencentes ao orquestrador de containers como um todo, e não a cada serviço (aplicação) que está em execução. Este módulo tem também configurações de DNS.
- **network** tem todas as configuraçoes da VPC, como suas máscaras de IP e suas sub-redes privadas.
- **s3** provisiona um *bucket S3*, solução de armazenamento de arquivos, que serve à aplicação de autenticação para armazenar imagens de perfil dos usuários, assim como um arquivo .jpg, o primeiro item a ser armazenado no repositório de arquivos, que é a imagem de perfil dos usuários antes destes escolherem trocar de imagem de perfil.

- **server** configura o balanceador de carga de cada servidor e as configurações do container que cada aplicação executa, como suas variáveis de ambiente, e memória e CPU disponíveis para execução.

- **sqs** configura a fila que comunica os dois serviços, assim como o código e a configuração da função Lambda que consome a fila e ativa o serviço de domínio.

Ademais, dentro da pasta *environment*, encontram-se as instanciações dos módulos descritos anteriormente, em que se declara quantos de cada um desses módulos se deseja, e passam-se parâmetros para cada um destes, para que suas configurações estejam completas. Encontram-se também *scripts bash* que auxiliaram no dia-a-dia do desenvolvimento deste projeto, como, por exemplo:

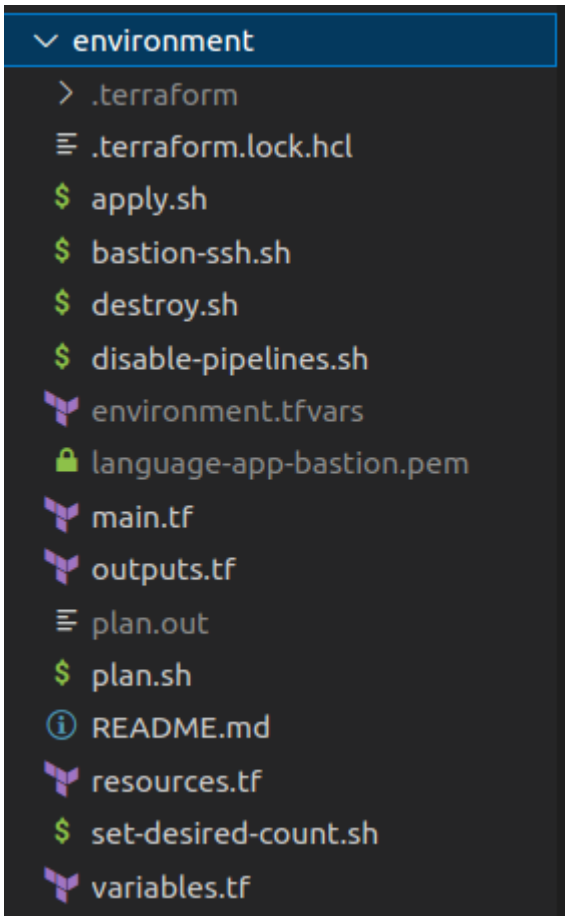


Figura 46: Pasta *environment*, que contém a instanciação dos módulos descritos acima, além de *bash scripts* que auxiliaram o dia-a-dia do desenvolvimento

- *plan.sh*, que utiliza a *CLI (Command Line Interface)* do Terraform para criar um arquivo chamado *plan.out*, que descreve quais mudanças o Terraform deve executar na infraestrutura que está provisionada no momento para que ela se iguale àquela

descrita nos arquivos de coniguração. Esse script era utilizado após se realizar alterações nos arquivos de configuração.

- *apply.sh* aplica as alterações previstas em *plan.out* à infraestrutura real da AWS, utilizando a CLI da AWS instalada na máquina de desenvolvimento.
- *set-desired-count.sh* utiliza a CLI da AWS para reduzir o número de containers ativos de todas as aplicações em todos os ambientes a zero, para poupar custos enquanto as aplicações não estavam sendo utilizadas, assim como para religar os servidores.

6 CONCLUSÃO

Neste capítulo, descreve-se alguns dos aprendizados ao se realizar o projeto e seus próximos passos.

6.1 Aprendizados

Esta seção é escrita em primeira pessoa, pois reflete a experiência do autor.

Este foi o projeto de engenharia de software mais complexo e completo que já realizei. Anteriormente, em experiências com estágio em empresas, os projetos variavam em complexidade, mas eu nunca havia tido uma visão tão completa de um trabalho de engenharia de software. Nessas experiências, frequentemente a base de código já existia, as principais decisões de infraestrutura e de convenções de código já haviam sido tomadas, e as tarefas a serem realizadas como estagiário dificilmente permitiam a abrangência com que trabalhei neste projeto, ao se projetar e implementar um sistema de ponta a ponta, do levantamento de requisitos à infraestrutura, passando pela experiência de usuário e pelas convenções de código.

Esta grande oportunidade de aprendizado trouxe consigo algumas lições importantes, dentre as quais:

1. **Planejar antes de implementar** é uma lição clássica, acredito, em qualquer área da engenharia. Apesar de ser algo amplamente discutido nas disciplinas da graduação, talvez seja necessário passar por algum retrabalho antes de realmente absorver a lição. Pelo menos foi o meu caso. Algum retrabalho foi realizado neste projeto, principalmente devido à baixa qualidade do planejamento, e não tanto à falta de um. O escopo do projeto deve de ser alterado algumas vezes durante o seu curso, por se tratar de um projeto de alto risco. Entende-se por projeto de alto risco aquele em que as tecnologias e técnicas necessárias não são completamente familiares aos desenvolvedores de antemão. O planejamento inicial contava com um escopo

maior do que o que acabou sendo realizado, com mais funcionalidades, justamente porque muitas das tecnologias ainda eram desconhecidas e foi necessário mais tempo do que o inicialmente previsto para aprendê-las, sacrificando, assim, tempo que iria para a implementação de mais funcionalidades. Para tentar mitigar esse problema nos próximos projetos, levarei em conta as próximas duas lições, descritas a seguir.

2. **Realizar o mais simples primeiro** diz respeito a não introduzir no sistema complexidade sem que a complexidade se prove realmente necessária. Afinal, complexidade é um custo: custo de manutenção, pois código mais complexo requer mais esforço para se alterar, e custo de *onboarding* de novos profissionais no time, que têm que compreender as convenções empregadas antes de se tornarem produtivos no projeto. Fiquei com a impressão de que a maneira como implementei Arquitetura Limpa no *backend*, juntamente com o código de plataforma, foi um pouco complexa demais. Tentei seguir a Arquitetura Limpa à risca, como descrita no livro, e o código acabou ficando bastante complexo. Mesmo com toda a complexidade, tive que fazer concessões ao modelo proposto por Robert Martin no seu texto original. Termino esta fase do projeto com a impressão de que deveria ter feito mais concessões ao modelo original para ganhar em simplicidade. Vale notar que pesquisei muitas implementações da Arquitetura Limpa em NodeJS e Typescript enquanto estava concebendo essas convenções, antes do início do TCC, e a maioria das implementações que eu encontrava faziam ainda mais dessas concessões.

3. **Implementar de maneira vertical, e não horizontal (ou realizar provas de conceito primeiro)**. Entendo por uma abordagem de desenvolvimento horizontal, ou por camadas, implementar tudo o que é necessário implementar em cada camada de cada vez, e progredir de uma camada a outra em sucessão. No caso deste projeto, implementar-se-ia primeiramente toda a camada de domínio do *backend*, com suas regras de negócio e testes automatizados, então, a camada de aplicação, que executa as funcionalidades da aplicação, depois toda a interface, e assim sucessivamente. Entendo agora que esta abordagem traz severas desvantagens em um projeto de alto risco: gasta-se muito tempo em camadas em que as tecnologias já são conhecidas, e potencialmente sobra pouco tempo para implementar camadas em que a tecnologia ainda é desconhecida. Uma abordagem mais apropriada seria uma mais vertical, em que se implementa um pouco de cada camada, mas se passa por todas as camadas rapidamente e se entende os riscos de cada uma antes de se expandir horizontalmente. Concretamente, neste projeto, seria o caso de implementar apenas algumas classes da camada de domínio, um caso de uso e um *endpoint* HTTP, e

uma página que usasse esse endpoint, além do mínimo necessário na infraestrutura para ver tudo isso funcionando. Esse "mínimo"é uma **prova de conceito**, e estou convencido de que é a melhor abordagem para para projetos de alto risco.

Essas foram lições holísticas do campo da engenharia, e que dizem respeito ao percurso do autor pela formação como engenheiro da computação na Escola Politécnica. Além delas, muito se aprendeu sobre tecnologia.

A respeito de experiência de desenvolvimento e pipelines de entrega, eu nunca havia provisionado qualquer infraestrutura em ambiente de nuvem, e tinha apenas uma idéia abstrata de como provedores de nuvem pública funcionavam.

Eu tampouco havia projetado uma experiência e uma interface de usuário da complexidade que as deste trabalho se tornaram, e a tarefa de implementar código de cliente (*frontend*) livre de *bugs* e que proporcionem uma experiência fluida para o usuário se mostrou uma empreitada mais difícil do que o esperado, e que ainda não foi alcançada plenamente.

No *backend*, a estrutura de Arquitetura Limpa e código de plataforma foi o resultado de experimentações com esses conceitos iniciadas em fevereiro de 2021 (ano anterior à produção deste trabalho), e foi gratificante chegar a uma versão desse conjunto de convenções que aderissem tão bem à teoria original, apesar de mais trabalho ainda ser necessário na elaboração de convenções para código de testes.

6.2 Próximos passos

Um dos objetivos deste trabalho, como menciona-se na introdução, é a elaboração (e exploração, pelo autor) de convenções de código, infraestrutura e experiência de desenvolvimento profissionais e modernas. Ainda há o que se fazer para se alcançar plenamente esses objetivos. Segue uma breve descrição do que falta, do ponto de vista de tecnologia.

1. **Observabilidade** é o conjunto de técnicas que permite a um time de desenvolvimento compreender e acompanhar o que acontece nas suas aplicações enquanto estas estão em produção. Compreender os erros que os usuários encontram, se os servidores estão sobrecarregados (e o quanto), e o tráfego geral da aplicação. Uma aplicação observável precisa contar com várias tecnologias diferentes, que possibilitem coleção de métricas e de logs e suas visualizações. No futuro, será implementado neste projeto instrumentação (coleção de métricas e logs) utilizando o projeto Open

Source Open Telemetry (OPEN. . . ,), e visualização de logs e métricas utilizando Graphana (GRAPHANA,).

2. **Testes Automatizados** são essenciais para qualquer aplicação profissional. Este projeto, contudo, na sua atual versão (final de 2022), conta com muito poucos. É necessário implementar testes no backend nas camadas de domínio e casos de uso, assim como no código de plataforma. Muitas das convenções suficientes para implementar o código desses testes automatizados já estão definidas no projeto, mas a maturidade e a qualidade dessas decisões só pode ser verificada ampliando-se a cobertura de testes da aplicação. Ademais, o front-end do projeto não conta com testes. Planeja-se utilizar o Cypress (CYPRESS,) para realizar os testes de interface, e ter ao menos um por página.

3. A **Arquitetura do frontend** foi pouco explorada neste projeto. Por arquitetura entende-se design de código: quais classes e funções criar para implementar algum algoritmo, de que maneira modularizar o código, onde colocar cada linha de código. A arquitetura limpa, amplamente implementada no *backend* deste projeto, não tem uma tradução simples e direta para o código do *frontend*, que costuma ser bastante diferente, principalmente por lidar com manipulação de interfaces e com HTML e por ter que seguir as convenções determinadas pelo framework Javascript (neste caso, o ReactJS). Por esste motivo, nenhum arcabouço teórico específico foi empregado ao se tomar decisões de design no front-end. A consequência negativa dessa negligência é um código de *frontend* de difícil manutenção. Deseja-se explorar melhores decisões deste design, iniciando-se pela própria Arquitetura Limpa, ao se procurar maneiras de utilizá-la no *frontend*.

6.3 Links úteis

Vídeo de demonstração da aplicação https://youtu.be/zu0unBdP3oI

Repositório com o código do projeto https://github.com/iago-srm/language-app

REFERÊNCIAS

AWS EC2. ⟨https://aws.amazon.com/pt/ec2/⟩.

AWS ECS. ⟨https://aws.amazon.com/pt/ecs/⟩.

AWS Lambda. ⟨https://aws.amazon.com/pt/lambda/⟩.

AWS RDS. ⟨https://aws.amazon.com/pt/rds/⟩.

AWS S3. ⟨https://aws.amazon.com/pt/s3/⟩.

AWS SQS. ⟨https://aws.amazon.com/pt/sqs/⟩.

AWS VPC. ⟨https://aws.amazon.com/pt/vpc/⟩.

BASTION Host. ⟨https://aws.amazon.com/pt/solutions/implementations/linux-bastion/⟩.

CEFR. ⟨d(https://www.britishcouncil.org.br/quadro-comum-europeu-de-referencia-para-linguas-cefr⟩.

COCKBURN, A. jan. 2005. Disponível em: ⟨https://alistair.cockburn.us/hexagonal-architecture/⟩. Acesso em: 20 oct. 2022.

CYPRESS. ⟨d(https://www.cypress.io/⟩.

DOCKER. ⟨https://www.docker.com/⟩.

ESLINT. ⟨dhttps://eslint.org/⟩.

EXPRESS. ⟨https://www.npmjs.com/package/express⟩.

GIT Lab Workflow. ⟨https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow⟩.

GITHUB Actions. ⟨d(https://github.com/features/actions⟩.

GRAPHANA. ⟨d(https://grafana.com/⟩.

MACORATTI, J. C. jun. 2019. Disponível em: ⟨https://www.macoratti.net/19/06/aspnc_autjwt1.htm⟩. Acesso em: 17 oct. 2022.

MARTIN, R. C. aug. 2012. Disponível em: ⟨https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html⟩. Acesso em: 19 oct. 2022.

MARTIN, R. C. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. 1. ed. California: Pearson, 2017.

OPEN Telemetry. ⟨d(https://opentelemetry.io/⟩.

POSTGRE. ⟨https://www.postgresql.org/⟩.

PRETTIER. ⟨d(https://prettier.io//⟩.

PRISMA. ⟨d(https://www.prisma.io/⟩.

SENDGRID. ⟨d(https://sendgrid.com/⟩.

TERRAFORM. ⟨https://www.terraform.io/⟩.

THE Art of Designing GitlabFlow for a Team Project. ⟨d(https://medium.com/swlh/the-art-of-designing-gitlab-flow-for-a-team-project-76994b9df337⟩.

TINY MCE. ⟨d(https://www.tiny.cloud/⟩.