Escola Superior de Tecnologia e Gestão

Instituto Politécnico de Viana do Castelo

CTeSP de TPSI - Tecnologias e Programação de Sistemas de Informação

Projeto de Sistemas de Informação

Especificação do projeto

Plataforma "Data Storing"

2024/2025

32187 – Lucas de Linhares

32188 – Guilherme Sousa

Sumário

1. Âmbito / Enquadramento do projeto

O presente projeto insere-se na disciplina de Projeto de Sistemas de Informação e tem como objetivo o desenvolvimento de uma plataforma que permita aos utilizadores carregar imagens de peças de vestuário e categorizá-las. A partir dessas informações, o sistema será capaz de sugerir combinações de outfits de acordo com as condições meteorológicas, proporcionando uma experiência personalizada ao utilizador.

A plataforma destina-se a pessoas interessadas em otimizar a escolha de roupas conforme o clima e as suas preferências pessoais. Utiliza algoritmos de machine learning que, com base em dados fornecidos pelos utilizadores, sugerem combinações adequadas para diferentes estações e temperaturas. Para isso, as imagens carregadas pelos utilizadores são utilizadas para treinar o modelo de machine learning, permitindo ao sistema melhorar continuamente as sugestões.

2. Objetivos

O principal objetivo deste projeto foi planear e desenvolver uma aplicação que permita aos utilizadores gerir e inserir imagens de peças de roupa, bem como classificá-las de forma detalhada, atribuindo informações como tipo, cor e adequação a diferentes condições climáticas.

O âmbito do trabalho incluiu a implementação de uma solução multiplataforma, abrangendo aplicações para Web, Windows, Linux e Android, desenvolvidas de forma integrada em todas as suas camadas de serviços.

Adicionalmente, o dataset gerado pelos utilizadores foi concebido para servir como base de treino para futuros modelos de machine learning, com o objetivo de sugerir outfits personalizados e adequados a cada utilizador.

Desde o início, o projeto teve um grande foco em performance, procurando garantir uma experiência fluida e responsiva em todas as plataformas. Para tal, cada tecnologia e solução adotada foi cuidadosamente selecionada com base em benchmarks e estudos realizados, priorizando eficiência, qualidade de código e segurança.

3. Análise de Requisitos

Requisitos funcionais

Nº	Descrição	Perfil	Prioridade
RF1	Conseguir efetuar login	Administrador Operador	Alta
RF2	Inserir informação (imagens e a sua categorização)	Administrador Operador	Alta
RF3	Pedir acesso para poder se registar na plataforma	Operador	Alta
RF4	Conseguir usar o modelo treinado	Administrador Operador	Alta
RF5	Conseguir iniciar o processo de treinamento do modelo	Administrador	Alta
RF6	Conseguir gerir operadores (aceitar/remover operadores e editar as suas informações)	Administrador	Alta

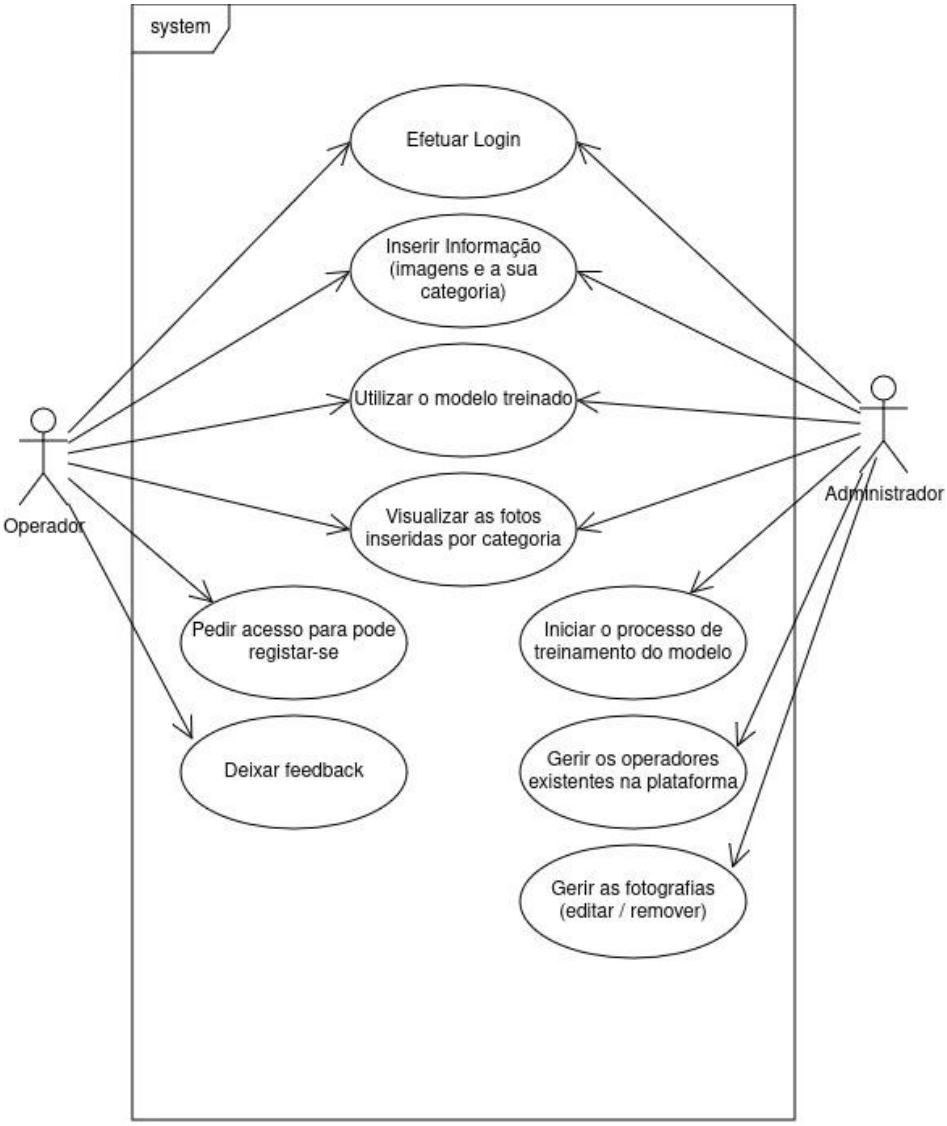
RF7	Conseguir visualizar as fotos inseridas por categoria	Administrador Operador	Média
RF8	Conseguir gerir uma dada fotografia (editar as suas informações ou removê-la)	Administrador	Média
RF9	Conseguir deixar feedback/reportar bugs	Operador	Baixa

Requisitos não funcionais

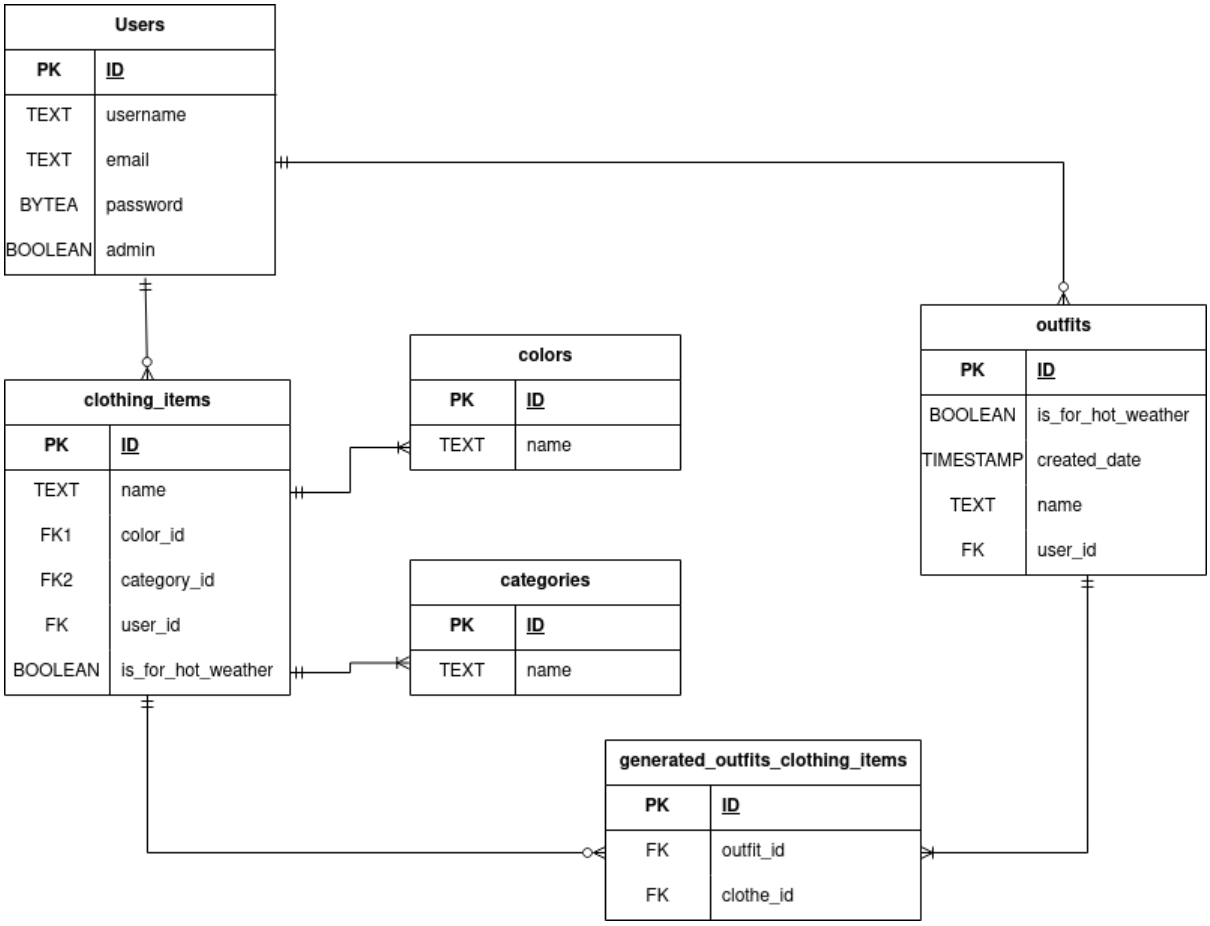
Nº	Descrição	Prioridade
RNF1	Ter o software disponível em vários idiomas (português, inglês)	Baixa
RNF2	Ter o software disponível em todas as principais plataformas (web, mobile e desktop)	Média
RNF3	A plataforma tem de ter uma boa performance	Alta
RNF4	A plataforma tem de ser segura	Alta
RNF5	A plataforma deve estar disponível 24 horas por dia, 7 dias por semana.	Alta
RNF6	A plataforma deve ter um design responsivo	Alta

4. Design e modelação da solução

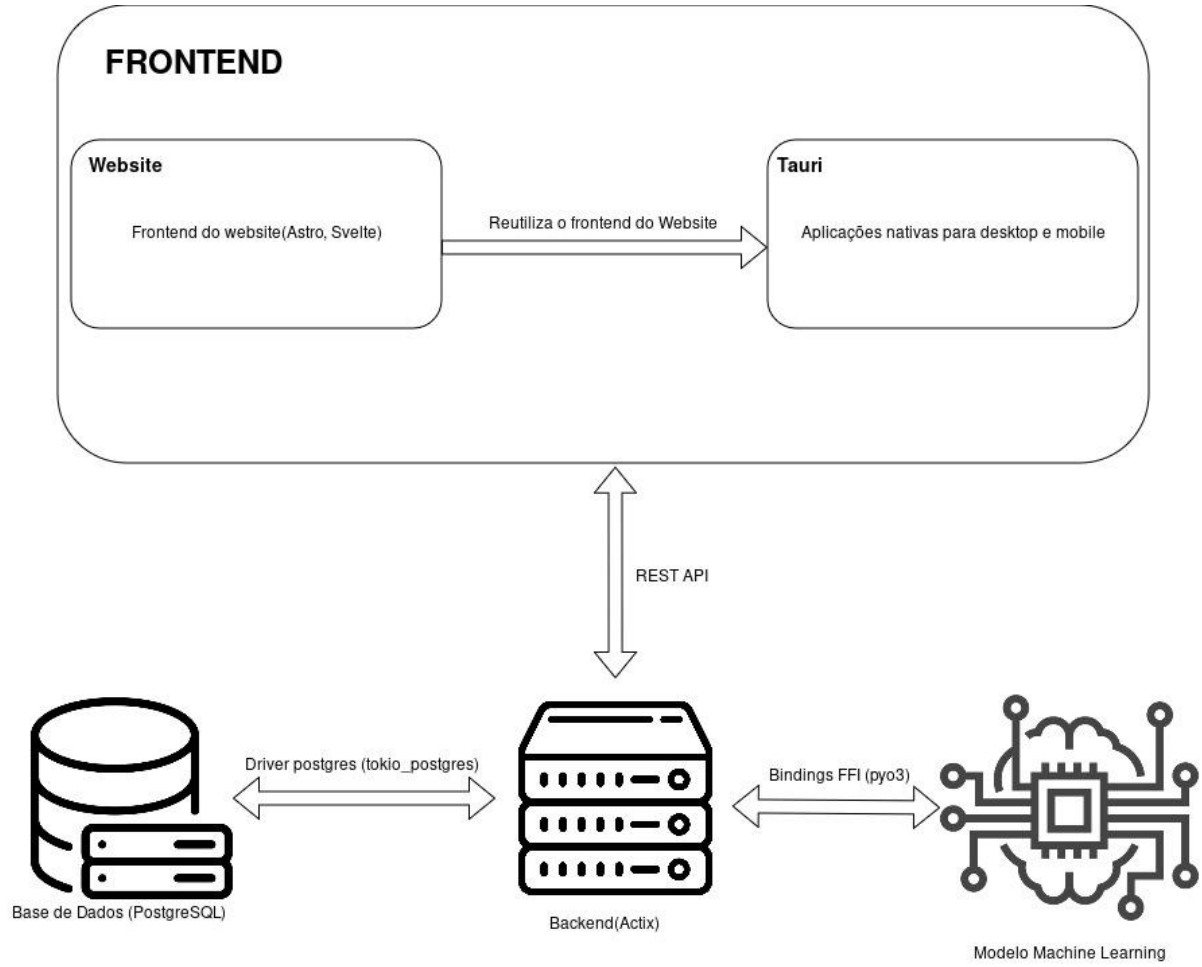
4.1Modelo de casos de uso



4.2Modelo de dados (modelo relacional)



4.3 Arquitetura geral da solução



A base de dados comunica e o modelo de machine leaning comunicam apenas com o backend, e o backend comunica via REST API com o frontend.

Para este trabalho escolhemos o driver tokio_postgres pois é mais leve do que as alternativas.

Escolhemos também fazer a comunicação do modelo de machine learning com o backend via bindings FFI, pois é a maneira com mais performance, para esta tarefa usamos a biblioteca pyo3 que facilita a criação dos bindings.

4.4 Tecnologias envolvidas

Aqui estão as tecnologias utilizadas para o desenvolvimento deste projeto:

- *Frontend*
  - *Astro – Metaframework moderna*
  - *Svelte – UI framework moderna*

- *BunJS – Runtime de JS rápido e moderno*
- *TypeScript – Utilizado para reduzir erros de runtime*
- *Tauri – Desenvolvimento das aplicações para cada plataforma*

- *Backend*
  - *Rust – Linguagem rápida, segura e moderna*
  - *Actix – Framework com ótima performance*
  - *PostgreSQL (tokio-postgres)*
- *Modelo Machine Learning*
  - *Python – Linguagem mais utilizada para AI*

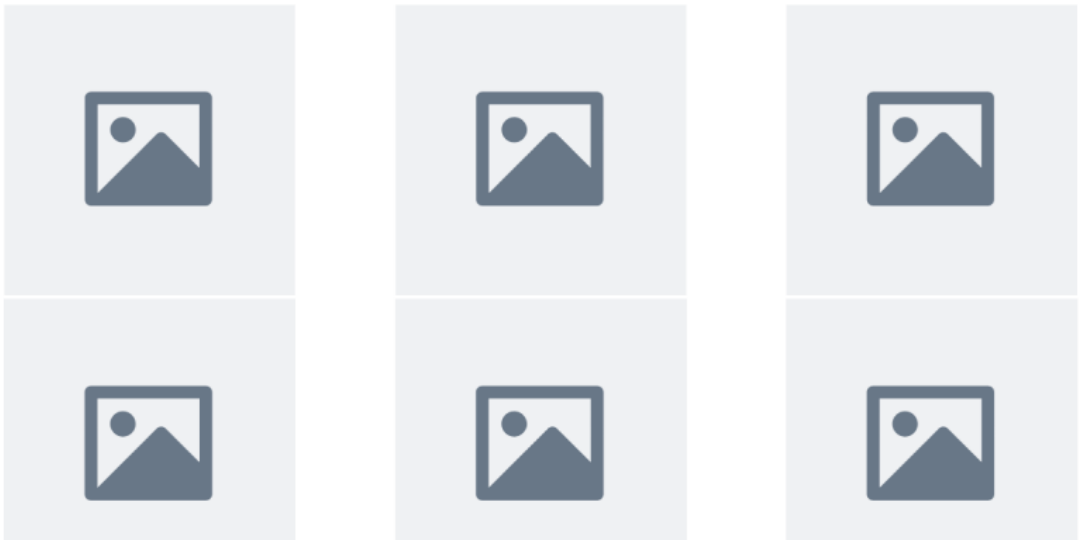
Ferramentas utilizadas:

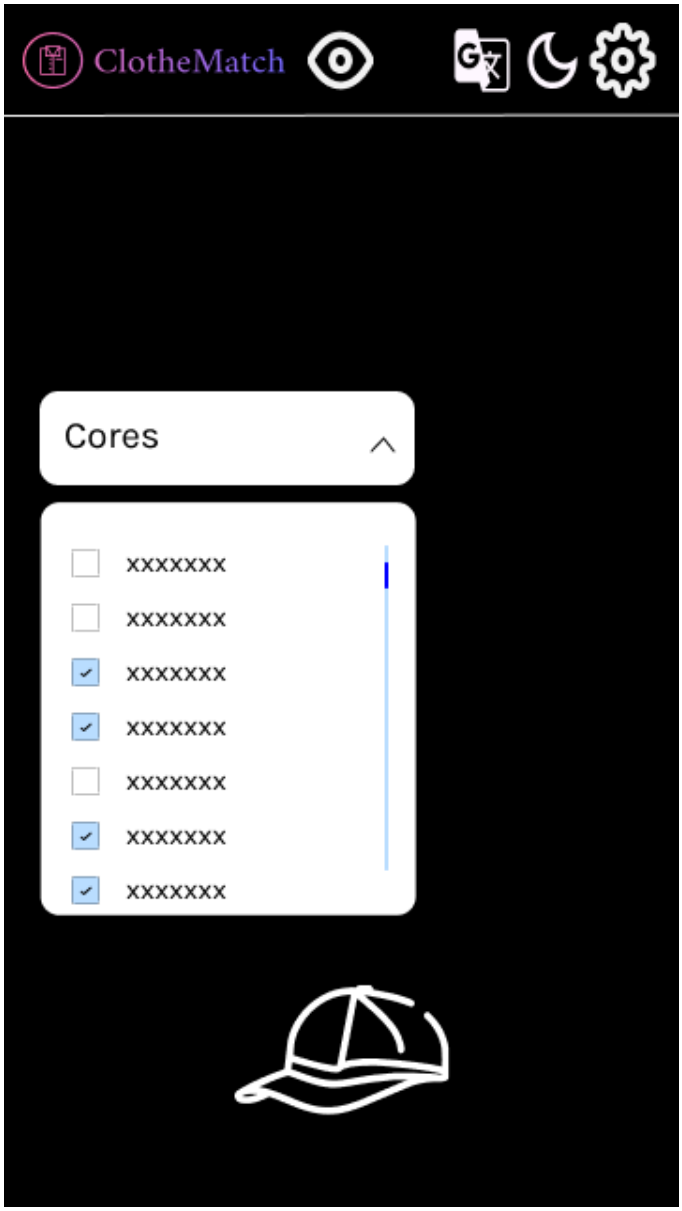
- *Postman -> Utilizado para testar o API*
- *Visual Studio Code -> IDE utilizado com extensões para as tecnologias usadas*
- *DataGrip -> Gerir a base de dados e consola da base de dados*

5. Mockups


ClotheMatch


ClotheMatch
Clothes




ClotheMatch
Cores
xxxxxxx
xxxxxxx
xxxxxxx
xxxxxxx
xxxxxxx
xxxxxxx
xxxxxxx

ClotheMatch

Clothes

Temperature

Default

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

Colors

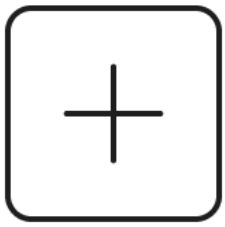
- [ ] xxxxxxx
- [ ] xxxxxxx
- [x] xxxxxxx
- [x] xxxxxxx
- [ ] xxxxxxx
- [x] xxxxxxx
- [x] xxxxxxx


ClotheMatch
Carregar imagens aqui
Arraste aqui

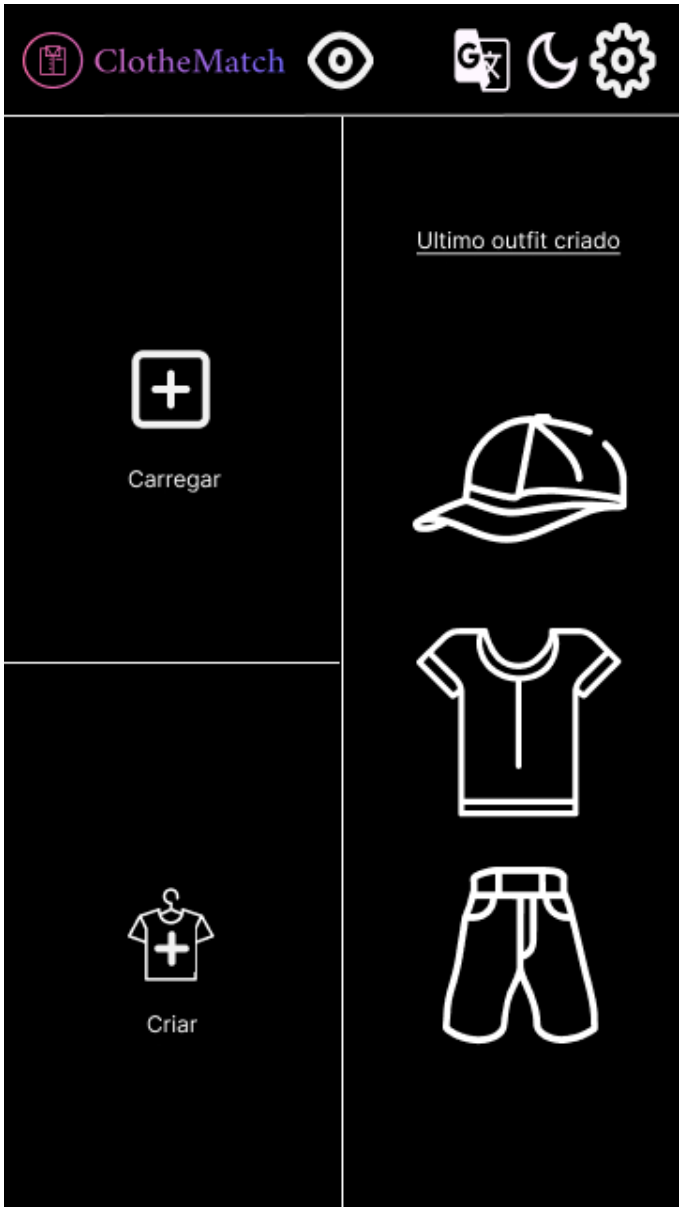
ClotheMatch

Clothes

Upload Images Here



Drag and drop


ClotheMatch
Ultimo outfit criado
Carregar
Criar

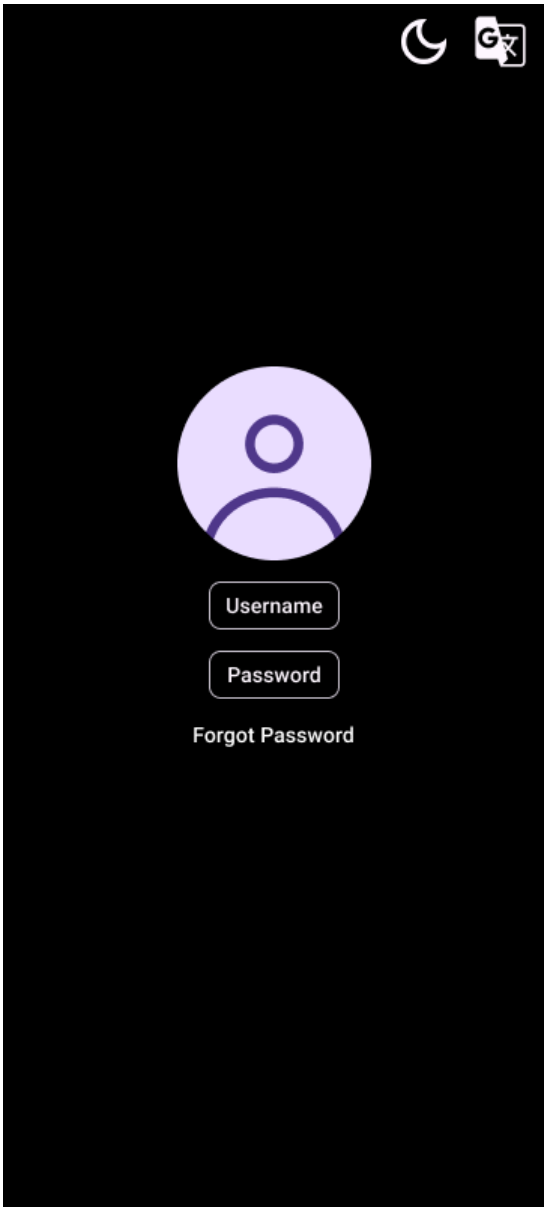
ClotheMatch

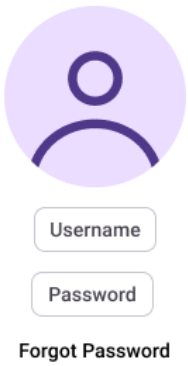
Clothes

Upload

Generate

Last Generated Outfit


Username
Password
Forgot Password



6. Cronograma

O desenvolvimento do frontend ficou maioritariamente a cargo do Guilherme, enquanto o backend foi principalmente responsabilidade do Lucas. No entanto, colaborámos de forma constante ao longo de todo o processo, ajudando-nos mutuamente sempre que necessário. Assim, as tarefas foram bem distribuídas e executadas de forma eficiente, garantindo a coesão do projeto.

19/09 -> Inicio relatório (requisitos, objetivos e ideia inicial das tenologias usadas)
26/09 -> Continuação do desenvolvimento do relatório (modelo relacional, arquitetura e casos de uso)
03/10 -> Inicialização do desenvolvimento do projeto
10/10 até o final -> Continuação do desenvolvimento do projeto e do relatório

7. Implementação

7.1 Estudo de dependências

Neste projeto, estabelecemos como prioridades performance, qualidade de código e segurança. Por isso, dedicámos uma parte significativa do tempo a investigar e selecionar as melhores soluções tecnológicas disponíveis para cada necessidade, com o objetivo de minimizar as dependências externas e maximizar a eficiência da nossa aplicação. Abaixo, detalhamos as principais dependências escolhidas e o racional por trás dessas escolhas:

- **Astro**: Um metaframework moderno e extremamente rápido, que suporta diferentes estratégias de renderização (como SSR, SSG e SPA), permitindo-nos otimizar o frontend de forma eficiente para cada cenário.
- **Svelte**: Uma framework frontend rápida e moderna, focada em compilar componentes para JavaScript puro, o que resulta em menor tamanho de ficheiros e maior performance.
- **TailwindCSS**: Uma solução de CSS inteligente que gera apenas o CSS necessário, reduzindo significativamente o tamanho final do stylesheet e promovendo a reutilização eficiente de estilos.
- **DaisyUI**: Biblioteca de componentes de UI baseada em TailwindCSS, altamente otimizada, que nos permitiu construir rapidamente uma interface moderna e responsiva.
- **sonic_rs**: Um parser JSON escrito em Rust, extremamente rápido, que utiliza **SIMD** para alcançar um desempenho superior, essencial para processamento eficiente de dados.
- **Tauri**: Framework para aplicações desktop e móveis, escrita em Rust, que nos permitiu criar uma aplicação **extremamente leve** (menos de 1 MB no desktop e 5 MB no Android). Tauri utiliza os recursos nativos de cada plataforma, garantindo ótima performance e baixo consumo de recursos.
- **argon2-kdf**: Biblioteca Rust para o algoritmo de hashing de passwords argon2id, considerada a implementação mais rápida (segundo os nossos benchmarks). Esta biblioteca utiliza o código original submetido no Password Hashing Competition (PHC) e cria bindings para Rust.
- **Mimalloc**: Um alocador de memória alternativo, que oferece performance superior em comparação com o alocador padrão do sistema.
- **bun**: Runtime e package manager para JavaScript, reconhecido como o mais rápido atualmente, o que nos permitiu otimizar os nossos scripts e dependências JS.
- **uv**: Um package manager rápido e moderno para Python, utilizado para gerir as dependências necessárias.
- **tokio_postgres**: Driver leve e rápido para PostgreSQL, que utiliza menos abstrações e é focado em desempenho para interações com a base de dados.
- **Rust**: Uma linguagem de programação moderna de baixo nível, que oferece performance no mesmo nível de C e segurança de memória integrada, permitindo-nos escrever código rápido, robusto e seguro.
- **Tokio**: Um runtime assíncrono para Rust, conhecido pela sua eficiência. Sempre que possível, utilizámos paralelismo e concorrência para otimizar o desempenho do sistema.

Ao optar por estas dependências, conseguimos criar um sistema eficiente, seguro e escalável, mantendo o código limpo e sem sobrecarregar o projeto com dependências desnecessárias. Este processo de seleção meticulosa garantiu que cada tecnologia escolhida se alinhasse com os nossos objetivos e padrões de qualidade.

7.2 Estudo de algoritmos de password hashing

Durante o desenvolvimento deste projeto, realizámos um estudo aprofundado para avaliar diferentes algoritmos de password hashing e as suas implementações em Rust. O objetivo foi selecionar a solução mais eficiente, segura e adaptada às necessidades da plataforma. Além disso,

realizámos uma comparação direta com as implementações padrão de PHP, como PASSWORD_BCRYPT e PASSWORD_ARGON2ID, para obter um panorama claro do desempenho relativo.

Os resultados do estudo estão disponíveis no repositório GitHub: https://github.com/lucascompython/argon2-bench-rust.

Bibliotecas Avaliadas

As bibliotecas testadas foram:

- **argon2**: Implementação em Rust.
- **rust-argon2**: Implementação em Rust.
- **argon2-kdf**: Bindings para a implementação original em C.
- **scrypt**: Implementação em Rust do algoritmo Scrypt.
- **bcrypt**: Implementação em Rust do algoritmo Bcrypt.

Parâmetros Utilizados no Benchmark

Argon2 (variant=Argon2id):

- m=65536
- t=4
- p=4
- password="password"
- salt=randomly generated with rand::rngs::OsRng
- salt_length=16
- hash_length=32
- version=13

Scrypt:

- log_n=16
- r=8
- p=1

Bcrypt:

- cost=10

Os testes foram realizados utilizando a biblioteca **criterion** para benchmarking em Rust.

Resultados

Os benchmarks foram executados num sistema com as seguintes especificações: **AMD Ryzen 9 7950X (32) @ 5.88 GHz**.

Resultados em Rust:

- **argon2**:
  - Hash: **71.344 ms** a **71.638 ms**
  - Verificação: **70.104 ms** a **70.368 ms**
- **rust-argon2**:
  - Hash: **128.73 ms** a **129.14 ms**
  - Verificação: **133.18 ms** a **133.57 ms**
- **argon2-kdf**:
  - Hash: **23.596 ms** a **23.693 ms**
  - Verificação: **23.525 ms** a **23.667 ms**
- **scrypt**:
  - Hash: **89.648 ms** a **89.896 ms**
  - Verificação: **90.840 ms** a **91.185 ms**
- **bcrypt**:
  - Hash: **37.686 ms** a **37.739 ms**
  - Verificação: **37.660 ms** a **37.733 ms**

Resultados em PHP (para comparação):

- **Bcrypt:**
  - Hash: **34.89 ms**
  - Verificação: **34.30 ms**
- **Argon2id:**
  - Hash: **36.84 ms**
  - Verificação: **37.31 ms**

Conclusões

A biblioteca **argon2-kdf** destacou-se como a implementação mais rápida, sendo significativamente mais eficiente do que as alternativas em Rust e PHP para o algoritmo **Argon2id**. A escolha dessa biblioteca, baseada nos nossos benchmarks, foi motivada pela sua excelente performance e pelo facto de utilizar o código original submetido no **Password Hashing Competition (PHC)**.

Além disso, a comparação com PHP demonstrou que a nossa solução não só mantém níveis elevados de segurança, como também atinge tempos de execução competitivos, mesmo utilizando uma linguagem de baixo nível como Rust. Este estudo permitiu-nos tomar uma decisão fundamentada para implementar um sistema de hashing de passwords seguro e otimizado, enquanto contribuímos para a comunidade open source com benchmarks úteis.

7.3 Estudo de primitivas de sincronização

Durante o desenvolvimento do projeto, conduzimos um estudo detalhado para avaliar diferentes primitivas de sincronização em Rust, com o objetivo de identificar a solução mais eficiente e adequada para o nosso caso de uso. Comparamos as primitivas da biblioteca padrão de Rust (std) com as primitivas fornecidas pela biblioteca parking_lot, que é frequentemente destacada por oferecer melhor performance em cenários específicos.

Os benchmarks completos deste estudo estão disponíveis no repositório GitHub: https://github.com/lucascompython/parking_lot_vs_std.

Resultados dos Benchmarks

Os testes foram realizados para várias situações, incluindo operações single-thread, multi-thread e cenários com contenção, abrangendo mutexes, read-write locks e barriers. Abaixo estão os principais resultados:

Primitivas da Biblioteca parking_lot

- **Mutex single-thread**: 23.603 ns – 23.708 ns
- **Mutex multi-thread (4 threads)**: 90.583 µs – 92.480 µs
- **RWLock single-thread (read)**: 23.666 ns – 23.819 ns
- **RWLock single-thread (write)**: 23.572 ns – 23.682 ns
- **RWLock multi-thread (read, 4 threads)**: 90.042 µs – 91.736 µs
- **RWLock multi-thread (write, 4 threads)**: 91.235 µs – 92.893 µs
- **Mutex contended (4 threads)**: 79.287 µs – 81.167 µs
- **RWLock contended (read, 4 threads)**: 79.370 µs – 81.072 µs
- **RWLock contended (write, 4 threads)**: 79.654 µs – 81.266 µs
- **Barrier**: 91.781 µs – 93.468 µs

Primitivas da Biblioteca std

- **Mutex single-thread**: 23.523 ns – 23.677 ns
- **Mutex multi-thread (4 threads)**: 91.450 µs – 93.441 µs
- **RWLock single-thread (read)**: 29.475 ns – 30.103 ns
- **RWLock single-thread (write)**: 23.597 ns – 23.739 ns
- **RWLock multi-thread (read, 4 threads)**: 90.814 µs – 92.477 µs
- **RWLock multi-thread (write, 4 threads)**: 92.022 µs – 93.694 µs
- **Mutex contended (4 threads)**: 80.041 µs – 81.986 µs
- **RWLock contended (read, 4 threads)**: 81.301 µs – 83.227 µs

- **RWLock contended (write, 4 threads)**: 80.308 µs – 81.958 µs
- **Barrier**: 92.871 µs – 94.742 µs

Conclusões

Embora as primitivas da biblioteca parking_lot apresentem ligeiras vantagens em cenários específicos, os resultados gerais mostraram que as primitivas da biblioteca padrão de Rust são igualmente competitivas, especialmente após as recentes atualizações que melhoraram sua performance. Considerando a simplicidade, estabilidade e a integração nativa da biblioteca padrão com o ecossistema de Rust, optámos por utilizá-la no nosso projeto.

Este estudo foi valioso para garantir que a nossa aplicação alcançasse um bom equilíbrio entre desempenho, simplicidade e manutenção. Além disso, os benchmarks realizados contribuíram para a documentação de performance dessas primitivas, uma vez que poucos recursos atualizados estavam disponíveis.

7.4 Estudo de otimizações de compilação

Como parte do desenvolvimento deste projeto, dedicámos uma parte significativa do nosso tempo ao estudo de técnicas de otimização tanto em tempo de compilação (compile time) quanto em tempo de execução (run time). O objetivo era garantir que o desempenho da aplicação fosse maximizado sem comprometer a manutenibilidade do código.

7.4.1 Otimizações a compile time

Para reduzir o tempo de compilação, melhorar a eficiência do processo de desenvolvimento, melhorar o tamanho da app e a sua performance, implementámos as seguintes estratégias:

1. Minimização de Dependências: Escolhemos as bibliotecas essenciais, evitando dependências desnecessárias, o que resultou em compilações mais rápidas e binários mais leves. Por exemplo, usamos argon2-kdf em vez de bibliotecas mais pesadas para hashing de passwords, e sonic_rs como parser de JSON por ser otimizado para uso de SIMD.

2. Configuração de Perfis de Compilação: Depois de investigar diversos recursos, fizemos alterações nas configurações do projeto (Cargo.toml), do compilador (config.toml), build flags e variáveis de ambiente nos projetos em rust (app e backend):
    - Cargo.toml:

```
[profile.release]
codegen-units = 1 # Permite que o LLVM execute melhor otimização.
lto = true # Permite otimizações de link-time.
panic = "abort" # Maior desempenho e menos código ao desativar manipuladores de pânico.
strip = true # Garante que os símbolos de depuração sejam removidos.
opt-level = "z" # Otimizar para tamanho. APENAS PARA A APP POIS PODE REDUZIR A PERFORMANCE
```

- config.toml na app:

```
[unstable]
build-std = [
  "std",
  "panic_abort",
] # Compilar a biblioteca std a partir da fonte e abortar em caso pânico
build-std-features = [
  "optimize_for_size",
  "panic_immediate_abort",
] # Otimizar o tamanho e entrar em pânico e abortar imediatamente
trim-paths = true # Remover o caminho para o diretório do projeto das informações de depuração
```

- config.toml no backend:

```
[build]
rustflags = [
  "-C",
  "target-cpu=native",
] # Otimizar para o CPU em que a compilação está a ser executada o que pode ou não afetar SIMD dependendo do CPU
```

- Variáveis de ambiente na compilação da app:

```
"-Zlocation-detail=none" # Remover informação sobre localização do ambiente em que o projeto foi compilado
"-Zfmt-debug=none" # Remover formatação de debugs
```

3. Compressão de binário: Utilizamos opcionalmente o UPX para comprimir o binário da app o que reduz o seu tamanho por cerca de 20%. A configuração é a seguinte: `upx --ultra-brute caminho_do_binario`
4. Compressão de recursos no frontend: Utilizamos um plugin para Astro, Astro Compress para comprimir o projeto. Adicionámos nas integrações em conjunto com a integração de Svelte e TailwindCSS:

```
// @ts-check
import { defineConfig } from "astro/config";

import svelte from "@astrojs/svelte";

import tailwind from "@astrojs/tailwind";

// https://astro.build/config
export default defineConfig({
integrations: [svelte(), tailwind(), (await import("@playform/compress")).default()],
});
```

7.4.2 Otimizações a run time

Durante a execução da aplicação, implementámos várias técnicas para garantir uma performance eficiente e responsiva, com especial foco em paralelismo e utilização otimizada dos recursos do sistema:

1. Utilização de Algoritmos Otimizados:
   Fizemos benchmarks para escolher algoritmos de hashing de passwords, decidindo-nos pelo argon2-kdf, que apresentou o melhor desempenho em tempo de execução. Além disso, usamos o alocador de memória mimalloc, que se mostrou mais rápido e eficiente em cenários de alocação intensiva.
2. Paralelismo e Concorrência:
   Usámos o runtime Tokio para operações assíncronas e paralelas no backend, permitindo a execução eficiente de tarefas I/O-bound e CPU-bound. Sempre que possível, adotámos abordagens concorrentes, como processamento de múltiplos uploads simultaneamente.
3. Análise e Minimização de Alocações Dinâmicas:
   Através de ferramentas como *cargo-flamegraph*, identificámos e eliminámos alocações desnecessárias, reduzindo o overhead de runtime e melhorando a performance geral.
4. Benchmarking Contínuo:
   Para monitorizar e comparar a performance em tempo de execução, utilizámos crates como *criterion* para realizar benchmarks detalhados e identificar pontos de melhoria.

7.5 Criação dos diversos subprojetos e build system

Dividimos o projeto nas seguintes partes: app, frontend, backend, model.

Decidimos criar um build system personalizado pois cada subprojeto tem o seu próprio build system com diversas opções/configurações de compilação. Ficaria muito extenso e chato estar a decorar uma série de comandos enormes para cada subprojeto.

Então em python criarmos este build system que consiste de um ficheiro make.py por subprojeto e um ficheiro make.py na raiz do projeto. O ficheiro make.py na raiz do projeto chama os outros de cada subprojeto que queremos compilar, ou então podemos simplesmente usar o make.py de cada subprojeto. Todos eles são implementam uma *dataclass Args* que define os argumentos/modos de compilação de cada projeto. Por exemplo o a dataclass do make.py da app:

```
@dataclass
class Args:
        dev: bool
        release: bool
```

```
        mobile: bool
        clean: bool
        upx: bool
        nightly: bool
        native: bool
        build_frontend: bool
        run: bool
        smallest: bool
        keys: bool
```

Todos estes argumentos estão importados pelo make.py geral e/ou usados como argumentos de CLI.

Exemplos de comandos:

```
# app
./make.py --help
# resultado
usage: make.py [-h] [-d | -r] [-m] [-bf] [-c] [-u] [-R] [-n] [--native] [-s] [-k]

App build script

options:
-h, --help show this help message and exit
-d, --dev Run in development mode
-r, --release Build in release mode
-m, --mobile Build the mobile version (requires Android NDK and SDK)
-bf, --build-frontend
Also build the frontend on release
-c, --clean Clean the target directory
-u, --upx Compress the binary with UPX
-R, --run Run the release application
-n, --nightly Build with nightly compiler for further optimizations
--native Build with native CPU target (not recommended for distribution)
-s, --smallest Build with optimizations for size (enables release, nightly and upx)
-k, --keys Create keystore and upload keys
# compilar app no modo mais otimizado para mobile
./make.py -m –s
# início da compilação
```

Estrutura do Projeto:

```
tree -L 2
.
├── app
│   ├── build.rs
│   ├── capabilities
│   ├── Cargo.lock
│   ├── Cargo.toml
│   ├── gen
│   ├── icons
│   ├── keystore.jks
│   ├── make.py
│   ├── README.md
│   ├── src
│   └── tauri.conf.json
├── backend
│   ├── Cargo.lock
│   ├── Cargo.toml
│   ├── make.py
│   ├── README.md
│   ├── sql
│   └── src
├── frontend
│   ├── astro.config.mjs
│   ├── bun.lockb
│   ├── make.py
│   ├── package.json
│   ├── public
│   ├── README.md
│   ├── src
│   ├── svelte.config.js
│   ├── tailwind.config.mjs
│   └── tsconfig.json
├── make.py
├── model
│   ├── main.py
│   ├── make.py
│   ├── pyproject.toml
│   └── README.md
├── README.md
```

```
├── resources
│   ├── Modelo Casos de Uso PDSDI Lucas Linhares 32187 Guilherme Sousa 32188.drawio
│   ├── Modelo Relacional PDSDI Lucas Linhares 32187 Guilherme Sousa 32188.drawio
│   └── PDSDI.postman_collection.json
└── utils.py
```

7.6 Lógica de Login: Utilização de Sessões e Estudo de JWT

No desenvolvimento da lógica de autenticação para a plataforma, implementámos um sistema baseado em sessões para gerir os estados de login dos utilizadores. Esta abordagem foi cuidadosamente analisada e escolhida em detrimento de outras soluções, como JSON Web Tokens (JWT), tendo em conta os requisitos e características específicas do nosso projeto.

Sessões: A Solução Implementada

Optámos por utilizar sessões armazenadas no servidor para gerir a autenticação. A lógica de funcionamento é a seguinte:

1. Quando um utilizador faz login, o servidor cria uma sessão única associada ao utilizador.
2. Um identificador de sessão (Session ID) é enviado ao cliente sob a forma de um cookie seguro e com a flag HttpOnly, para evitar acessos por scripts maliciosos.
3. As informações da sessão são armazenadas no lado do servidor, numa base de dados ou memória temporária, garantindo a segurança dos dados do utilizador.
4. Sempre que o cliente faz uma nova requisição, o cookie com o Session ID é enviado ao servidor, permitindo a validação do utilizador sem necessidade de voltar a autenticar.

Razões para Escolher Sessões:

- **Segurança Centralizada**: As informações sensíveis (como permissões ou informações de perfil) permanecem no servidor e não são expostas ao cliente, reduzindo os riscos de ataques como manipulação de tokens.
- **Facilidade de Revogação**: Sessões podem ser facilmente revogadas no lado do servidor (por exemplo, no caso de logout ou de sessão expirada), garantindo maior controlo.
- **Simplicidade no Uso e Implementação**: As sessões são uma solução tradicional e amplamente documentada, facilitando a integração e manutenção.

JWT: Uma Alternativa Estudada

Durante a fase de conceção, investigámos a utilização de JWT (JSON Web Tokens) como alternativa. Os JWT são amplamente usados em sistemas modernos de autenticação, especialmente em arquiteturas sem estado (stateless).

Os JWT funcionam assinando digitalmente um token que contém informações do utilizador. Este token é enviado ao cliente e incluído em cada requisição subsequente, sem necessidade de armazenamento no servidor.

Vantagens do JWT que Considerámos:

- **Arquitetura Sem Estado**: Ao contrário de sessões, os JWT não exigem armazenamento no servidor, permitindo maior escalabilidade em sistemas distribuídos.
- **Versatilidade**: Os JWT podem ser usados em sistemas híbridos, como autenticação para API REST e comunicação entre microserviços.
- **Redução da Sobrecarga no Servidor**: Menos carga de trabalho para o servidor, já que não há necessidade de gerir estados de sessão.

Razões para Não Usar JWT neste Projeto:

1. **Segurança**: Em sistemas baseados em JWT, os dados são armazenados no cliente (dentro do token). Apesar de serem assinados, se um token for comprometido, pode ser usado até expirar, e revogar tokens torna-se mais complexo.
2. **Complexidade Desnecessária**: Para o tamanho e escopo da nossa aplicação, o uso de JWT introduziria mais complexidade sem benefícios claros, dado que a gestão de sessões já cumpre adequadamente as nossas necessidades.
3. **Performance e Escalabilidade**: Sessões, em combinação com o nosso setup baseado em PostgreSQL e Tokio, apresentam uma performance muito próxima da utilização de JWT, especialmente para o tamanho atual da nossa base de utilizadores.

Aqui está código de exemplo do nosso backend sobre a sessão:

```
#[post("/login")]
async fn login(
    db: web::Data<Arc<DbClient>>,
    login_data: web::Bytes,
    session: Session,
) -> impl Responder {
    let Json(data): Json<LoginRequest> = Json::from_bytes(login_data).unwrap();

    match db.login_user(&data.email, &data.password).await {
        Ok(user) => {
            session.insert("user_id", user.user_id).unwrap();
            session.insert("is_admin", user.is_admin).unwrap();
            HttpResponse::Ok().finish()
        }
        Err(_) => HttpResponse::Unauthorized().finish(),
    }
}

fn validate_session(session: &Session) -> Result<i32, HttpResponse> {
    let user_id = session.get::<i32>("user_id").unwrap_or(None);
```

```
  match user_id {
    Some(id) => {
      session.renew();
      Ok(id)
    }
    None => Err(HttpResponse::Unauthorized().finish()),
  }
}
#[get("/protected")]
async fn protected(session: Session) -> impl Responder {
  let user_id = match validate_session(&session) {
    Ok(id) => id,
    Err(response) => return response,
  };

  let is_admin = session.get::<bool>("is_admin").unwrap().unwrap();

  if !is_admin {
    return HttpResponse::Forbidden().finish();
  }

  HttpResponse::Ok().body(format!(
    "User logged in with id: {}; And is admin: {}",
    user_id,
  session.get::<bool>("is_admin").unwrap().unwrap()
  ))
}
```

7.7 Sistema de internacionalização

Neste projeto o frontend é traduzido em português e em inglês.

O nosso sistema funciona de uma maneira em que as paginas com as duas línguas são geradas a compile-time, para que não haja necessidade de processar as traduções a run-time. Para não termos de copiar todas as nossas páginas duas vezes (uma para cada língua) no make.py do frontend temos a seguinte função que é sempre executada:

```
def _copy_files() -> None:
"""This function copies all the pages to the /pt folder"""

  Colors.info("Copying files")
  start = perf_counter()

  entries = os.listdir(f"{CWD}/src/pages")
```

```
  for entry in entries:
    if os.path.isfile(f"{CWD}/src/pages/{entry}"):
      copyfile(f"{CWD}/src/pages/{entry}", f"{CWD}/src/pages/pt/{entry}")

  elapsed = perf_counter() - start
  Colors.success(f"Copied files in {elapsed:.2f} seconds")
```

Esta função copia os ficheiros com as traduções em português para a diretoria /pt pois a root / é inglês.

Criamos as seguintes funções de ajuda no frontend para realizar as traduções:

```
// i18n/utils.ts
import { translations, defaultLang } from "./translations";

export function getLangFromUrl(url: URL) {
 const [, lang] = url.pathname.split("/");
 if (lang in translations) return lang as keyof typeof translations;
 return defaultLang;
}

export function useTranslations(lang: keyof typeof translations) {
 return function t(key: keyof (typeof translations)[typeof defaultLang])
 {
   return translations[lang][key] || translations[defaultLang][key];
 };
}

export function useTranslatedPath(lang: keyof typeof translations) {
 return function translatePath(path: string, l: string = lang) {
  return l === defaultLang ? path : `/${l}${path}`;
 };
}
```

A função getLangFromUrl aceita um objeto URL como argumento e retorna a língua atual.

A função useTranslations aceita uma língua e retorna uma função t que efetua as traduções.

A função useTranslatedPath aceita uma língua e retorna uma função translatePath que traduz o endereço para o idioma certo, ou seja, se tivermos em */pt* e quisermos ir para */login* ele vai traduzir para */pt/login*

São usadas da seguinte maneira:

```
// Componente em Svelte que recebe o URL via props do Astro
import {
 getLangFromUrl,
 useTranslatedPath,
 useTranslations,
} from "src/i18n/utils";
// ... Função
```

```
 let { windowLocation }: { windowLocation: URL } = $props();
 const lang = getLangFromUrl(windowLocation);
 const t = useTranslations(lang);
 const translatePath = useTranslatedPath(lang);
// Continuação...

// No template

<p>{t("login.email")}</p>
<a href={translatePath("login/forgot")}>{t("login.forgot.password")}</a>
```

7.8 Tudo o Resto

Não faz sentido detalhar todo o código desenvolvido neste relatório, dado o nível de complexidade e extensão do projeto. Contudo, todo o código-fonte está disponível publicamente no nosso repositório no GitHub:

https://github.com/lucascompython/PDSDI-TP

Aqui pode se explorar em detalhe as várias camadas do sistema, desde a lógica de autenticação, passando pela integração com a base de dados, até ao frontend desenvolvido. O repositório também contém documentação básica e instruções para executar o projeto localmente.

8. Conclusão

Este projeto destacou-se não só pela aplicação prática de conceitos de sistemas de informação, mas também pela nossa autonomia e iniciativa no desenvolvimento de uma solução completa, utilizando tecnologias e abordagens que não foram previamente abordadas no curso. Desde o início, a cadeira teve uma abordagem puramente prática, sem aulas teóricas, o que nos permitiu total liberdade para explorar ferramentas como Astro, Svelte, Tauri, Rust, Python, PostgreSQL e muitas outras tecnologias e abordagens que trouxemos para o projeto.

Durante o desenvolvimento, realizámos investigações aprofundadas sobre tópicos cruciais para a implementação do sistema. Um dos destaques foi a pesquisa relacionada com algoritmos de hashing de passwords em Rust, na qual criámos benchmarks para comparar uma série de algoritmos e implementações, incluindo a implementação padrão do PHP. Após uma análise criteriosa, optámos pelo algoritmo argon2id e utilizámos a biblioteca argon2-kdf em Rust, que modificámos através de um fork para melhor se adequar às nossas necessidades. Além disso, submetemos um pull request para o projeto original, permitindo-nos contribuir para uma ferramenta open source amplamente utilizada, algo que consideramos um marco significativo no nosso trabalho.

Outra área de pesquisa foi a comparação de primitivas de sincronização em Rust, onde analisámos as implementações padrão da biblioteca standard (stdlib) e da biblioteca parking_lot. Identificámos

uma escassez de recursos atualizados sobre o tema, o que nos levou a realizar testes para comparar a performance de ambas as abordagens. Optámos por utilizar as primitivas padrão da stdlib, dado que as atualizações recentes as colocaram ao mesmo nível de performance da biblioteca parking_lot, eliminando a necessidade de uma dependência externa.

Este projeto foi, assim, uma oportunidade única para combinar a prática com investigação independente, permitindo-nos expandir o nosso conhecimento técnico e contribuir ativamente para a comunidade open source. Além de termos desenvolvido uma plataforma funcional que cumpre os objetivos definidos, o processo enriqueceu a nossa formação enquanto futuros profissionais da área de sistemas de informação.

9. Melhorias futuras

Apesar de termos alcançado os principais objetivos do projeto, identificámos várias áreas com potencial para melhorias e expansões futuras. Uma das mais significativas é a implementação de um modelo de machine learning avançado para a geração de outfits. Devido ao tempo limitado, utilizámos apenas um algoritmo simples e baseado em regras para combinar peças de roupa, o que, embora funcional, não explora todo o potencial da plataforma.

A introdução de um modelo de machine learning permitiria personalizar ainda mais as sugestões de outfits, tendo em conta não só as preferências individuais de cada utilizador, mas também padrões de estilo, tendências de moda e contextos climáticos mais complexos. Esse modelo poderia ser treinado com os dados já armazenados na plataforma, utilizando técnicas como recommender systems ou redes neurais, para oferecer combinações mais precisas e adaptadas às necessidades dos utilizadores.

Outras melhorias futuras incluem:

- **Aprimorar a experiência do utilizador (UX)**: Tornar o frontend ainda mais intuitivo e dinâmico, com funcionalidades como filtros avançados para visualização de roupas e uma interface mais responsiva.
- **Expandir os atributos das peças de roupa**: Permitir que os utilizadores adicionem mais informações, como estação do ano, ocasiões específicas ou padrões visuais, para enriquecer a base de dados e melhorar as sugestões.
- **Integração de notificações e recomendações personalizadas**: Enviar sugestões de outfits ou lembretes para adicionar novas peças de roupa, utilizando notificações baseadas em preferências e condições climáticas.
- **Compilar Plataformas**: Nomeadamente plataformas Apple, IOS e Mac OS. Não desenvolvemos para estas plataformas pois as mesmas requerem serem desenvolvidas em dispositivos Apple. Da maneira que o projeto foi desenvolvido eram apenas necessárias umas pequenas alterações no make.py da app para poder alcançar isto.
- **Conteinerização**: Adicionar suporte para compilar e executar por exemplo o backend em Docker. Com isso, conseguiremos garantir uma execução consistente em diferentes

ambientes, melhorar a portabilidade entre plataformas e simplificar o processo de deployment. Além disso, o uso de containers permitirá uma gestão mais eficiente das dependências e facilitará a escalabilidade do sistema.

Essas melhorias futuras têm o potencial de transformar a plataforma numa ferramenta ainda mais poderosa, não só para os utilizadores finais, mas também como um exemplo de aplicação prática no domínio de sistemas de informação e machine learning.

10. Referências

https://crates.io/crates/tokio-postgres

https://pyo3.rs

https://krausest.github.io/js-framework-benchmark/current.html

https://astro.build/

https://tailwindcss.com/docs/optimizing-for-production

https://daisyui.com/

https://github.com/cloudwego/sonic-rs

https://github.com/cloudwego/sonic-rs/tree/main/benchmarks

https://tauri.app/

https://crates.io/crates/argon2-kdf

https://github.com/lucascompython/argon2-bench-rust

https://github.com/microsoft/mimalloc

https://github.com/microsoft/mimalloc/blob/dev/doc/bench-2021/bench-amd5950x-2021-01-30-a.svg

https://bun.sh/

https://blog.probirsarkar.com/hono-js-benchmark-node-js-vs-deno-2-0-vs-bun-which-is-the-fastest-8be6c210f5d8

https://en.wikipedia.org/wiki/Single_instruction,_multiple_data

https://github.com/johnthagen/min-sized-rust

https://nnethercote.github.io/perf-book/

https://upx.github.io/

https://github.com/PlayForm/Compress

https://github.com/flamegraph-rs/flamegraph

https://github.com/bheisler/criterion.rs