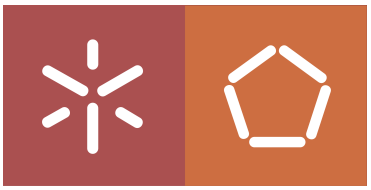

Universidade do Minho
Escola de Engenharia

Ano Letivo 2025/2026

AirTrail: Análise da Arquitetura

Aplicações e Serviços de Computação em Nuvem

Grupo 11

MEI - 1º Ano - 1º Semestre

Trabalho realizado por:

PG61130 - Ana Margarida Pires
PG58805 - Diana Dinis
PG61522 - Francisco Mota
PG59966 - João Silva
PG61551 - Manuel Fernandes

Braga, 17 de janeiro de 2026

Índice

1. Introdução

O presente relatório inicial tem como objetivo a exploração da aplicação *AirTrail* no quesito da sua arquitetura e principais componentes, funcionalidades e APIs e uma breve discussão sobre o esperado para as próximas fases de desenvolvimento.

1.1. Enquadramento

Este relatório pertence ao trabalho prático de Aplicações e Serviços de Computação em Nuvem, dividido em três *checkpoints*, onde este é referente ao primeiro *checkpoint*.

2. AirTrail

AirTrail é uma aplicação que permite aos seus utilizadores rastrear voos e consultar o próprio histórico de viagens.

Pela sua descrição, as funcionalidades da aplicação são as seguintes, presentes na Tabela 1.

Tabela 1: Funcionalidades da aplicação AirTrail

Funcionalidade	Descrição
Mapa Mundial	Ver os voos todos num mapa interativo
Histórico de Voo	Manter o histórico dos voos num lugar apenas
Estatísticas	Obter percepções do histórico
Múltiplos utilizadores	Gerir múltiplos utilizadores, partilha de dados entre estes, com autenticação e integração com OAuth
Importar voos	Importar voos de várias fontes

2.1. Instalação Manual

A instalação manual da aplicação permite conhecer o fluxo inteiro da instalação da aplicação, que acabará por ser uma vantagem quando se for implementar as automatizações em Ansible nas próximas fases.

A aplicação tem como requisitos:

- Git ⟶ Gestor de versões;
- Bun ⟶ Gestor de pacotes de JavaScript;
- Node.js ⟶ Ambiente de JavaScript;
- PostgresSQL ⟶ Motor de base de dados;

Com a documentação fornecida, apenas é necessário instalar os pacotes de requisitos, dependências utilizando já o gestor Bun, instanciar e iniciar uma base de dados Postgres e finalizar com um Build, que a aplicação fica pronta a ser utilizada.

A aplicação destaca-se pela sua forma dinâmica de adicionar voos, permitindo que os utilizadores registem facilmente novas viagens de forma rápida e intuitiva. Também é de notar o *design* criativo de rastrear as viagens pelo mapa do mundo interativo, que torna o uso da aplicação mais apelativo.

2.1.1. Dificuldades

Pela dimensão da aplicação, o tamanho original de memória das máquinas virtuais impossibilita o *build* da aplicação. Para resolver, aumentou-se manualmente o tamanho da memória em `NODE_OPTIONS`.

Como uma instalação local de PostgreSQL, foi necessário os passos extra para a configuração do utilizador AirTrail, a criação da base de dados e mudar o URL presente no ficheiro `.env`.

Apesar de ter sido concebida para ser instalada através de *containers Docker*, simplificando o processo de *deployment* e eliminando problemas de dependências de *software*, não foi utilizada por não se encontrar no *scope* deste trabalho prático, pois serão utilizadas estratégias diferentes de automatização.

2.2. Arquitetura e Principais Componentes

Na Figura 1 está presente um esquema dos principais componentes da aplicação AirTrail:

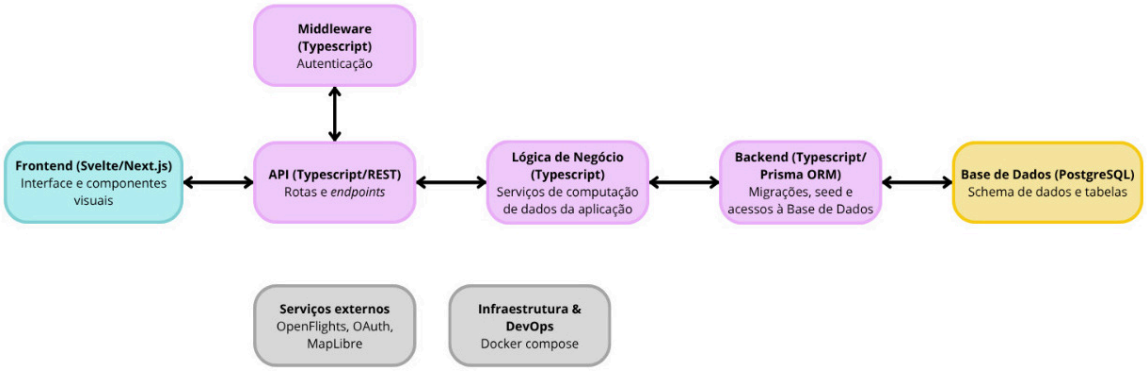

Figura 1: Arquitetura e principais componentes

Fundamentalmente, pode ser resumido em:

- **Frontend:** Interface em Svelte
- **Backend/API:** Implementado em TypeScript com Prisma ORM
- **Base de Dados:** PostgreSQL, gerido num container isolado (airtrail_db)
- **Docker Compose:** Orquestra os containers definidos em **docker/production** e **docker/development**

2.3. APIs

As rotas da API estão localizadas em **src/lib/server/routes/**

Os principais *endpoints* são os seguintes, na Tabela 2:

Tabela 2: Principais *endpoints* da aplicação AirTrail

Categoria	Endpoint	Método	Descrição
Aircraft	/api/aircraft/save/form	POST	Inserção um avião pelo formulário
Airlines	/api/airline/save/form	POST	Criação de companhia aérea por formulário
Airports	/api/airport/save/form	POST	Inserção de aeroportos por formulário
Flights	/api/flight/get/:id	GET	Retorna voo por ID
Flights	/api/flight/list	GET	Lista de voos
Flights	/api/flight/delete	DELETE	Remoção de voo
Flights	/api/flight/save/form	POST	Inserção de um novo voo
User	/api/users/login	POST	Autenticação de utilizadores
User	/api/users/add	POST	Criação de novo utilizador
Statistics	/api/visited-countries	GET	Consulta de países visitados

Estas APIs são invocadas internamente pelo *frontend*, garantindo uma separação entre a interface do utilizador e a lógica de negócio, como já visto na Figura 1.

3. Conclusão

3.1. Potenciais Gargalos de Desempenho e Pontos Únicos de Falha

Um dos principais gargalos a nível de desempenho da aplicação são as componentes *backend*/API e a base de dados, uma vez que o *Backend* executa a lógica de negócio (ex: "consultar histórico" ou "rastrear voo"), consumindo recursos computacionais consideráveis. Para executar essa lógica, o *Backend* necessita de realizar pedidos (*queries*) à base de dados. Num cenário de elevada concorrência, múltiplos utilizadores concorrentes, a base de dados ficaria sobrecarregada com um volume excessivo de pedidos provenientes do servidor de *backend*. Isto resultaria num aumento significativo da latência da aplicação. Consequentemente, os pedidos demorariam mais tempo a ser processados, forçando o utilizador a esperar vários segundos pela resposta. Esta degradação da *performance* afeta diretamente a experiência de utilização, tornando a aplicação lenta. Adicionalmente, este cenário de sobrecarga poderia conduzir ao esgotamento dos recursos do servidor (CPU, memória).

Outro potencial gargalo reside nas operações de I/O de disco, particularmente relevantes quando a aplicação realiza operações de escrita intensivas na base de dados, visto que o acesso ao disco é significativamente mais lento do que o acesso à memória RAM. Dessa forma, em cenários de carga elevada, se a base de dados não conseguir manter os dados mais acedidos em cache, será forçada a realizar operações frequentes de leitura/escrita em disco, aumentando muito a latência das *queries*.

Um ponto único de falha poderá vir a ser a base de dados, tendo em conta que se este componente falha, se não houver replicação desta ou outros mecanismos que minimizem o seu *downtime*, os pedidos provenientes do *backend* ficam por responder, levando a uma falha no serviço generalizado da aplicação.

3.2. Trabalho Futuro

Vai ser utilizada uma arquitetura orientada a serviços (SOA), onde o *frontend* e o *backend*, que vão ficar juntos no que se denominará *WebApp*, e a base de dados terão os seus próprios servidores, em nós diferentes, de modo a prevenir que, se um componente falha, falhe o serviço inteiro, e que não haja competição de recursos entre estes, vista a necessidade de recursos para operações de I/O por parte da base de dados.

A base de dados será o foco principal do desenvolvimento, dada a sua importância e problemas possíveis. Posto isto, será implementada uma versão da arquitetura *Manager-Worker* de modo a mitigar os gargalos de desempenho, dentro de um *pod*.

Na Figura 2 está presente o que se espera implementar nas próximas fases:

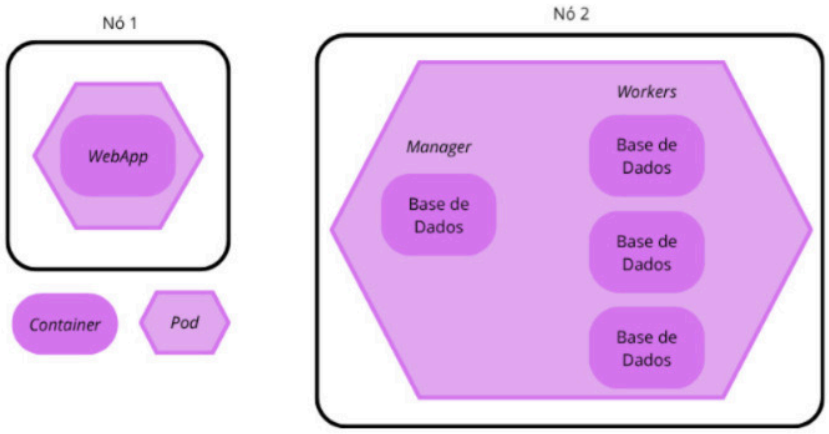

Figura 2: Arquitetura em Kubernetes a ser implementada