INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

ISEL | DEETC

ÁREA DEPARTAMENTAMENTAL DE ENGENHARIA DE
ELECTRÓNICA E TELECOMUNICAÇÕES E DE COMPUTADORES

**Mestrado em Engenharia Informática e Multimédia**

## Engenharia de Software

# Relatório Final

**Ano Lectivo:**
**2021/2022**

*Aluno:*
45414 - Arman FREITAS

*Professor:*
Eng. Luis MORGADO

06/02/2022

# Contents



# List of Figures

# 1 Introdução

O presente trabalho tem como intuito um maior entendimento sobre a área de engenharia de *software*. Para tal, foi desenvolvida uma aplicação *web*, através das práticas dadas nas aulas.

A aplicação desenvolvida consiste num *website* de *booking* de voos, onde é possível consultar voos para vastos destinos.

De forma a seguir todas as boas práticas para a maior facilidade na construção da aplicação teve se em conta o seguinte processo de desenvolvimento:

- Especificação de Requisitos.
- Arquitetura Lógica.
- Arquitetura Detalhada.

Devido aos requisitos se manterem constantes durante todo o processo desenvolvimento, foi adotada esta estrutura, possibilitando uma boa divisão das partes, diminuindo a complexidade em cada delas.

# 2 Análise de requisitos

A análise de requisitos é responsável por recolher dados indispensáveis para o desenvolvimento de um sistema. A ideia é começar com uma visão geral do problema e, chegar a requisitos concretos do sistema.

A **visão** geral consiste no documento de visão, onde se descreve o sistema em termos gerais, o contexto operacional, as partes envolvidas, as características do sistema, etc. Este documento define, de forma geral, o problema assim como a solução.

Um **glossário** foi também desenvolvido com o propósito de prevenir falhas de comunicação. Habitualmente são utilizados glossários entre cliente-fornecedor para que ambos se consigam entender na especificação de requisitos ou até mesmo em falhas que possam eventualmente acontecer. Este documento permite estabelecer uma linguagem comum entre o cliente e o fornecedor.

Com estes pilares bem estabelecidos (visão e glossário), é possível especificar os requisitos. Esta especificação é realizada através de **casos de utilização** (outras metodologias tais como o *Agile* podem utilizar outros aspetos de especificação de requisitos tais como a *user stories*) e **especificação suplementar**.

Os **casos de utilização** representam os requisitos funcionais, que são elaborados na perspetiva dos utilizadores e, a **especificação suplementar** requisitos não funcionais, especificação suplementar de funcionalidades.

O presente trabalho inclui os seguintes casos de utilização:

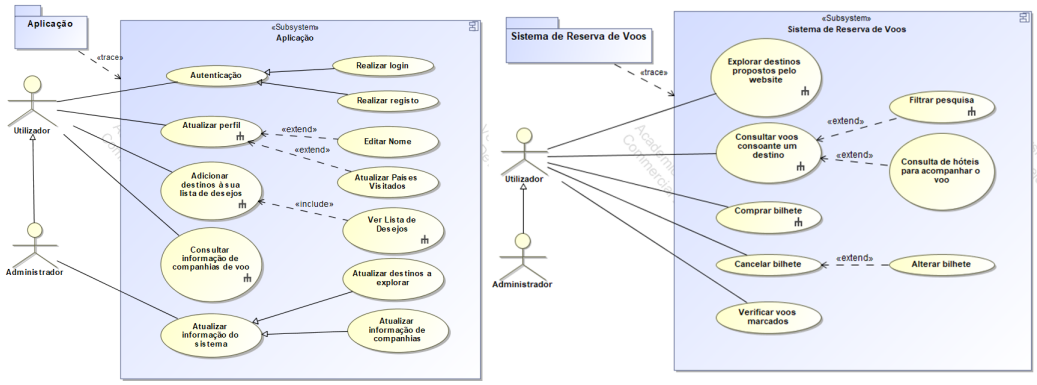


Figure 1: Diagrama de casos de utilização

De forma a manter coesão entre as partes foram utilizadas relações entre casos, tais como:

- Relação de extensão.
- Relação de inclusão.
- Relação de generalização.

A relação de generalização pretende especializar um caso de utilização abstrato.

A relação de extensão (*extend*) é opcional e, estende o comportamento do caso de utilização, adicionando mais passos.

Já a relação de inclusão (*include*) procura completar um caso de utilização, incluindo outro.

Por fim, analisando os casos de utilização detalhadamente, foi desenvolvida a específicação complementar. Nesta, constam os requisitos não funcionais tais como especificação de restrições em campos específicos do sistema para ter em conta durante o processo de construção.

# 3 Projeto de arquitetura da aplicação

Consta no desenvolvimento, operação e manutenção de *software* de maneira sistemática e quantificável.

Algumas das caraterísticas e métricas para o desenvolvimento de *software* são:

**Complexidade:** Complexidade é o grau de dificuldade de previsão das propriedades de um sistema dadas as propriedades das partes individuais. A mesma está relacionada com a informação que é necessária para a caracterização de um sistema.

Um sistema é tanto mais complexo quando mais informação seja necessária para a sua descrição.

**Entropia:** É a medida do grau de dispersão relativa que existe num sistema fechado num dado instante de tempo.

## 3.1 Métricas de Arquitetura

**Coesão:** Nível coerência funcional de um subsistema/módulo (até que ponto esse módulo realiza uma única função):

- Um nível de coesão baixo leva a que, em caso de necessidade de alteração de um subsistema, o número de módulos afectados seja elevado.
- Se o nível de coesão for elevado, o número de módulos afectados será minimizado.
- Um módulo com um nível de coesão baixo é mais complexo, logo mais difícil de conceber e de testar.

**Acoplamento:** Grau de interdependência entre subsistemas. Com um nível de acoplamento baixo obtém-se:

- Maior facilidade de desenvolvimento, instalação, manutenção e expansão.

- Melhor escalabilidade, devido à possibilidade de distribuição e replicação de módulos.
- Maior tolerância de falhas, uma vez que a falha de um subsistema tem um impacto restrito.

**Simplicidade:** Nível de facilidade de compreensão e comunicação da arquitectura implementada

**Adaptabilidade:** Nível de facilidade de alteração da arquitectura para incorporação de novos requisitos ou de alterações nos requisitos previamente definidos

## 3.2 Princípios

**Abstração:** Processo de descrição de conhecimento a diferentes níveis de detalhe e tipos de representação. É uma ferramenta base para lidar com a complexidade

**Modularidade:** Composta pela **decomposição** do sistema em partes coesas, o **encapsulamento** de dos detalhes internos do sistema e, as **interfaces** de comunicação com o exterior.

**Fatorização:** Processo de redução de redundância através de **herança** e **delegação**.

## 3.3 Modelo de Domínio

O modelo de domínio, faz parte da arquitetura lógica da aplicação. Aqui, são descritos todos os objetos necessários para suportar o comportamento e os dados do sistema, num ponto de vista completamente lógico. Isto é, apesar de ser um diagrama UML de classes, não procura especificar detalhadamente a arquitetura, mas sim representar o conhecimento que o arquiteto tem sobre o domínio da aplicação.

O modelo de domínio da aplicação é o seguinte (fig. 2):

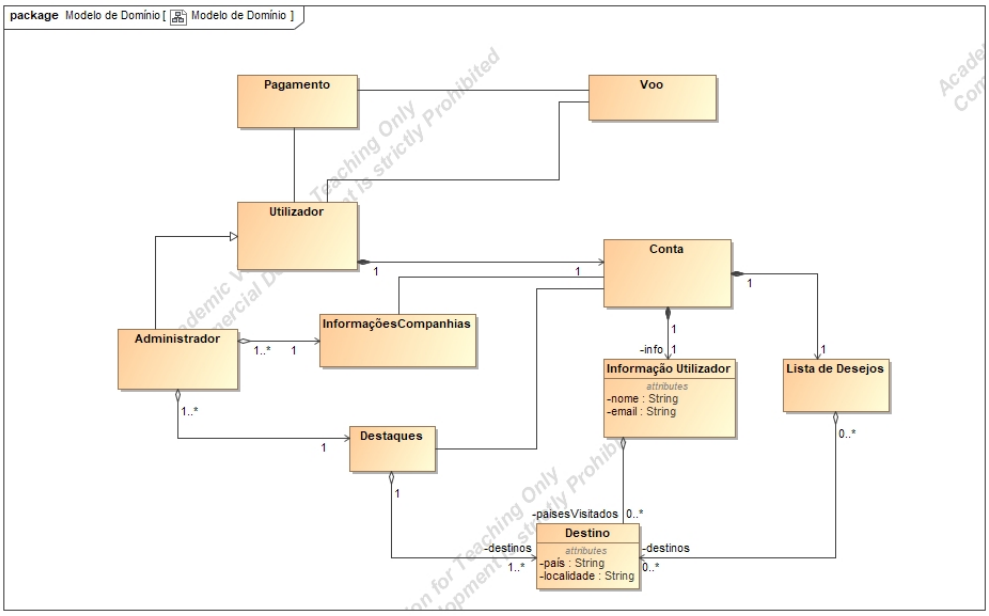


Figure 2: Modelo de Domínio

## 3.4 Padrão *Model View Controller*

São definidas várias camadas que reduzem o acoplamento entre níveis lógicos e físicos. É definida uma arquitetura lógica de 3 camadas:

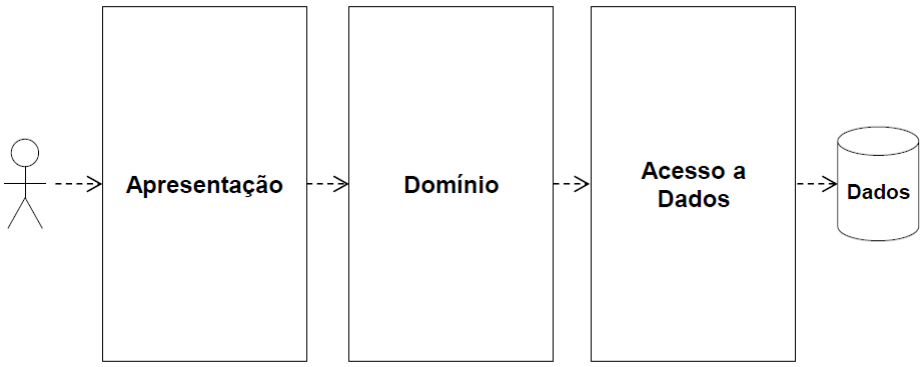


Figure 3: Arquitetura de 3 camadas

O padrão MVC (*Model View Controller*) procura separar o modelo de domínio do controlo de interação e da apresentação. Os papeis do Modelo, da Vista e do Controlador são podem ser descritos com ajuda da seguinte figura:

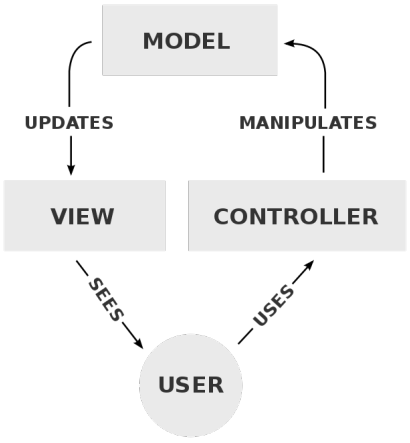


Figure 4: Padrão MVC

O **Modelo** está encarregue de gerir os dados do domínio da aplicação. As **Vistas** gerem a informação que é demonstrada ao utilizador e, o **Controlador** interpreta os *inputs* do utilizador.

Sendo assim, do ponto de vista de um utilizador, o mesmo faz *requests* ao controlador, que por sua vez manipula o modelo, que afeta diretamente as vistas que o utilizador vê.

## 3.5 Realização dos casos de utilização

Com um modelo de domínio desenvolvido e, um padrão determinado, é necessário começar a detalhar os casos de utilização estabelecidos na Análise de Requisitos.

Para tal, são utilizados diagramas de interação e temporização que, conseguem descrever detalhadamente cada caso de utilização. Estes diagramas permitem ao arquiteto da aplicação prever o fluxo de controlo que acontece na aplicação. São estes diagramas que, devem fazer uso do conhecimento do modelo de domínio de forma a dar origem a diagramas de classes mais específicos.

## 3.6 Redução de Acoplamento

Uma forma de reduzir o acomplamento em sistemas, é o uso de uma camada de fachada. Isto permite que todo o apenas haja uma interface de comunicação entre camadas e, não seja tudo ligado diretamente.

A fachada disponibiliza uma interface uniforme apartir de um conjunto de interfaces internas de um sistema.

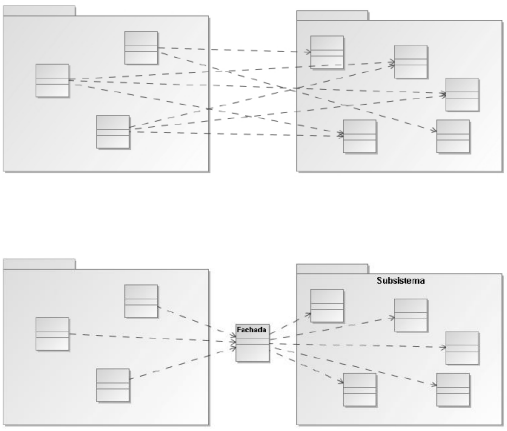


Figure 5: Uso de uma fachada

No caso da presente aplicação, é utilizada uma fachada para traduzir os dados vindos da apresentação para a camada de domínio. Como a fachada possuí dependências não se insere diretamente na camada de domínio.

## 3.7 Arquitetura de mecanismos

A arquitetura de mecanismos, pretende especificar os mecanismos da aplicação definidos nos casos de utilização. É aqui que se deve olhar para os diagramas de interação criados anteriormente. Muitas das classes criadas podem ser retiradas diretamente da realização dos casos de utilização assim como alguns dos seus métodos. A ideia é detalhar os mecanismos necessários para tornar reais os casos de utilização.

## 3.8 Arquitetura geral da solução

Com os mecanismos construídos, pretende-se abstrair um pouco mais, ficando assim com uma visão geral da solução proposta. A arquitetura geral da solução é constituída por subsistemas que, por sua vez, alojam mecanismos. De seguida pode-se observar a arquitetura geral da solução para a aplicação a desenvolver:

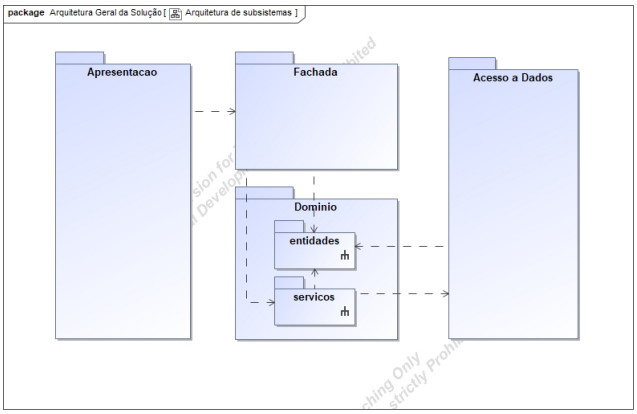


Figure 6: Diagrama de pacotes - Arquitetura geral da solução

# 4 Implementação do protótipo de teste

Os mecanismos foram definidos e a arquitetura geral também, resta testar as classes presentes no domínio. O protótipo de testes deve testar a camada de domínio. Esta é uma camada sem quaisquer dependências e, por isso pode ser testada utilizando apenas uma linguagem de programação pura, neste caso o JAVA.

Com ajuda das classes presentes nos mecanismos é possível traduzir os diagramas de classes da camada de domínio em código propriamente dito. Antes disso, é apenas necessário o desenho da aplicação de teste. Esta, deve conter uma classe por cada caso de utilização a testar de domínio. É também necessária uma classe base que, correrá os diferentes testes todos juntos.

O diagrama de classe do protótipo de teste encontra-se na seguinte figura (fig. 7):

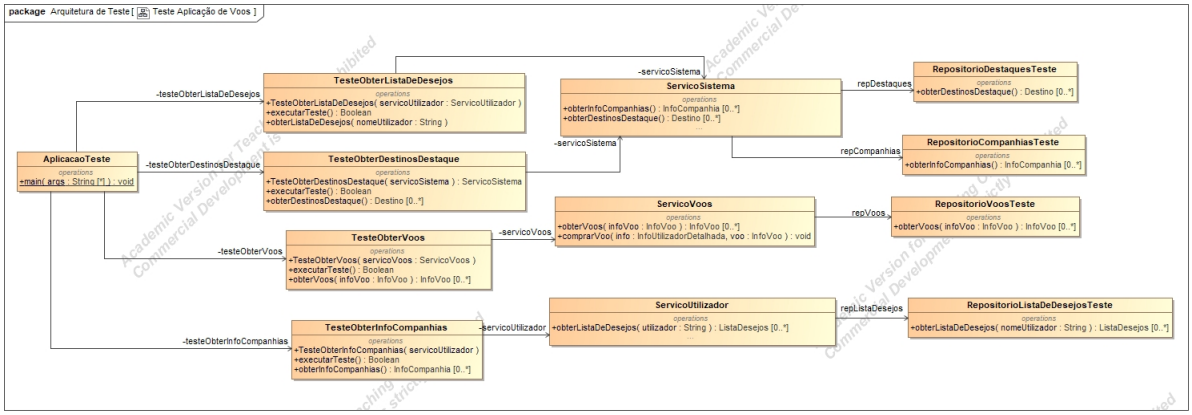


Figure 7: Diagrama UML - Protótipo de teste

No caso desta aplicação de teste, foram testadas quatro funcionalidades de domínio. É de notar que o os repositórios presentes não vão diretamente a uma base dados, retornam apenas objetos diretamente. Isto pois, a aplicação de teste apenas pretende testar o domínio e, o acesso a dados não pertence ao domínio.

# 5 Implementação do protótipo aplicacional

Por fim, em termos de desenvolvimento de aplicação, é finalmente o criado o protótipo aplicacional. Foi definido do documento de visão, que a aplicação era na *Web*. No entanto, devido à arquitetura ter sido construída sem qualquer acoplamento com a linguagem de programação, caso fosse necessário fazer a aplicação para outro dispositivo, seria bastante possível e, sem muita complexidade em termos de arquitetura.

Para este protótipo foram realizados os seguintes casos de utilização:

- Explorar destinos propostos pelo *website*.
- Consultar voos consoante um destino.
- Ver Lista de desejos.
- Consultar informação de companhias de voo.

O protótipo aplicacional foi concebido com o seguinte *Tech Stack*:

- *Framework* JAVA *Spring*.
- Base de dados relacional *Postgres*.
- *ReactJS*.

O diagrama de implantação que o sistema implementa é o seguinte (fig. 8)

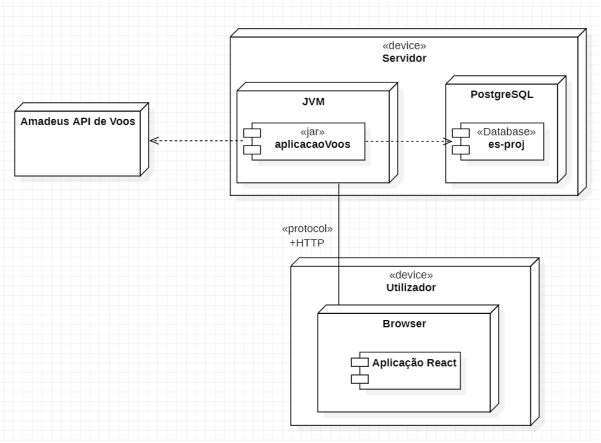


Figure 8: Diagrama de Implantação

Para a devida tradução dos diagramas de classes em código propriamente dito, foi utilizado o diagrama de detalhe de partes e mecanismos para o Domínio, Acesso a Dados e controladores de Apresentação, assim como o diagrama de mecanismo de apresentação para detalhar as vistas presentes na apresentação. Seguem-se as os dois diagramas, respetivamente:

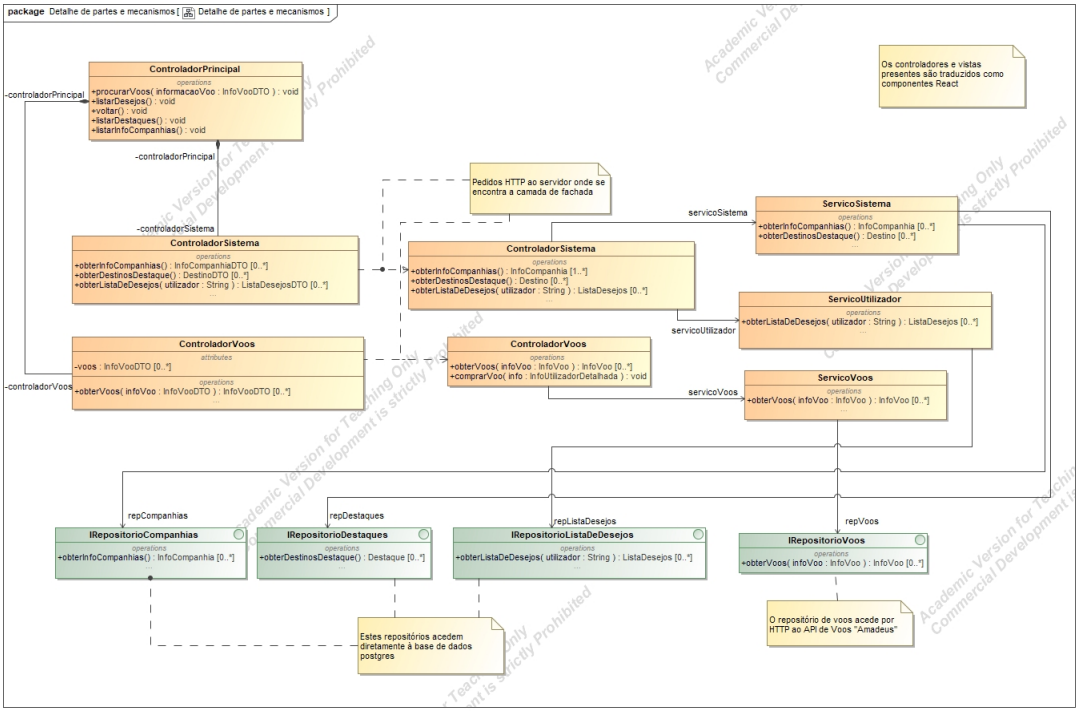


Figure 9: Diagrama UML - Detalhe de partes e mecanismos

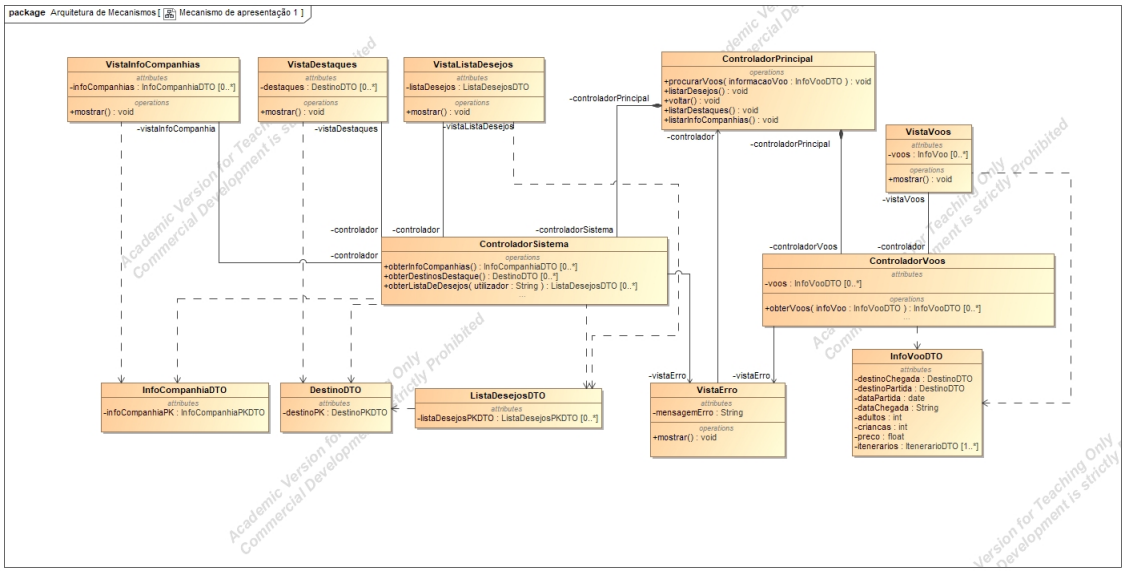


Figure 10: Diagrama UML - Mecanismo de apresentação

## 5.1 Domínio e Acesso a Dados

A camada de fachada, o domínio e, o acesso a dados encontra-se todo presente na plataforma JAVA e, é considerado o *backend* do sistema.

A razão pela qual não ter sido escolhido algo como o *NodeJS* para esta parte do trabalho, tem a ver com o facto de, não só de se estar mais habituado à linguagem de programação JAVA, mas também pois o JAVA, permite uma maior robustez devido a possuir tipos de variável, o que, claramente se traduz em menos ambiguidade e entropia no desenvolvimento.

A *Framework Spring*, disponibiliza um forma fácil de expor *endpoints* para camada de apresentação. Estes *endpoints* são disponibilizados pela camada de fachada que, acede diretamente aos serviços presentes no domínio que, por sua vez comunicam com a base de dados ou APIs externas.

A *Framework* disponibiliza também uma forma mais fácil de conexão com bases de dados, não tendo de estar a transferir o *drivers* para o programa comunicar com a base de dados específica. Apenas é necessário adicionar uma dependência no ficheiro "*pom.xml*" e, os acessos nos ficheiros de configuração.

Assim, o *JAVA Spring*, facilitou em grande escala o trabalho que se teria a criar esta parte da aplicação.

## 5.2 API de Voos - *Amadeus*

Esta, foi uma API utilizada para a recolha de voos presentes em determinadas datas. De todas as APIs encontradas, a *Amadeus*, é a mais completa e fácil de utilizar, tendo em conta a linguagem de programação JAVA.

Alguns repositórios (apenas o repositório de voos) podem comunicar diretamente com APIs externas. Através desta API é possível, pesquisa e compra de voos, com elevado detalhe na informação recebida. No entanto, para efeitos de tradução de arquitetura e redução de complexidade, todos os dados vindos API externa foram convertidos em objetos de domínio que, perdem alguma da informação vinda do exterior, de forma a eliminar dados não relevantes ao domínio do problema.

## 5.3 Base de Dados

A base de dados relacional guarda todos os estados necessários. Foi utilizada uma base de dados *Postgres*, com o seguinte modelo Entidade Associação (fig. 11), de forma a suportar as entidades presentes no domínio:

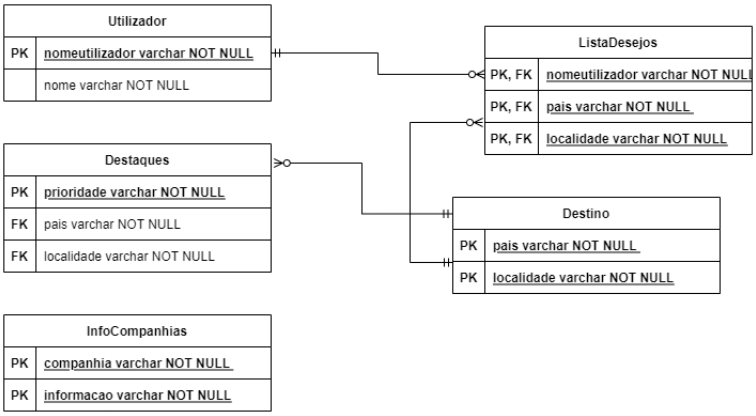


Figure 11: Modelo Entidade associação - Base de Dados *Postgres*

Uma base de dados relacional tem as suas vantagens. Primeiramente a sua **simplicidade**, como se pode observar pelo modelo Entidade Associação. Para manusear com ela é necessário apenas *queries SQL* sem grande complexidade.

A capacidade de **acesso a dados** é também um ponto forte que a diferencia de base de dados com modelos hierárquicos, por exemplo. Não é necessário navegar uma árvore para chegar aos dados. Apenas se faz *fetch* de informações de tabelas ou vistas diretamente.

A **flexibilidade** permite que, a base de dados seja de fácil expansão, facilitando a entrada de mais dados.

Todas estas razões levaram ao uso de uma base de dados relacional ao invés de uma *No-SQL*, por exemplo, presente em plataformas como a *Firebase* da *Google*. O uso do *Postgres*, invés do *MySQL*, *MongoDB*, entre outras aplicações, deve-se apenas à experiência que já se possuía no *Postgres*, estando a ser utilizado noutras unidades curriculares.

## 5.4 Apresentação

O *ReactJS* é onde se situa a camada de apresentação e, possuí todos os controladores e vistas. O uso desta biblioteca de *Javascript*, desenvolvida pela empresa *Meta* (antigo *Facebook*), deve-se a facilidade que o *ReactJS* dá na construção de aplicações na *web*. Para criação de interfaces de utilização complexas, aconselha-se a utilização de bibliotecas que facilitam este processo.

O *ReactJS*, não só ajuda no desenvolvimento da **interface de utilizador**, como tem também muita **performance**.

Os controladores e vistas foram desenvolvidos como sendo componentes *React*. Isto para que, os controladores consigam um rápido acesso ao *input* dos utilizadores e, as vistas tenham acesso direto à informação que vai é mostrada ao utilizador.

A aplicação ficou com o seguinte aspeto (fig. 12):

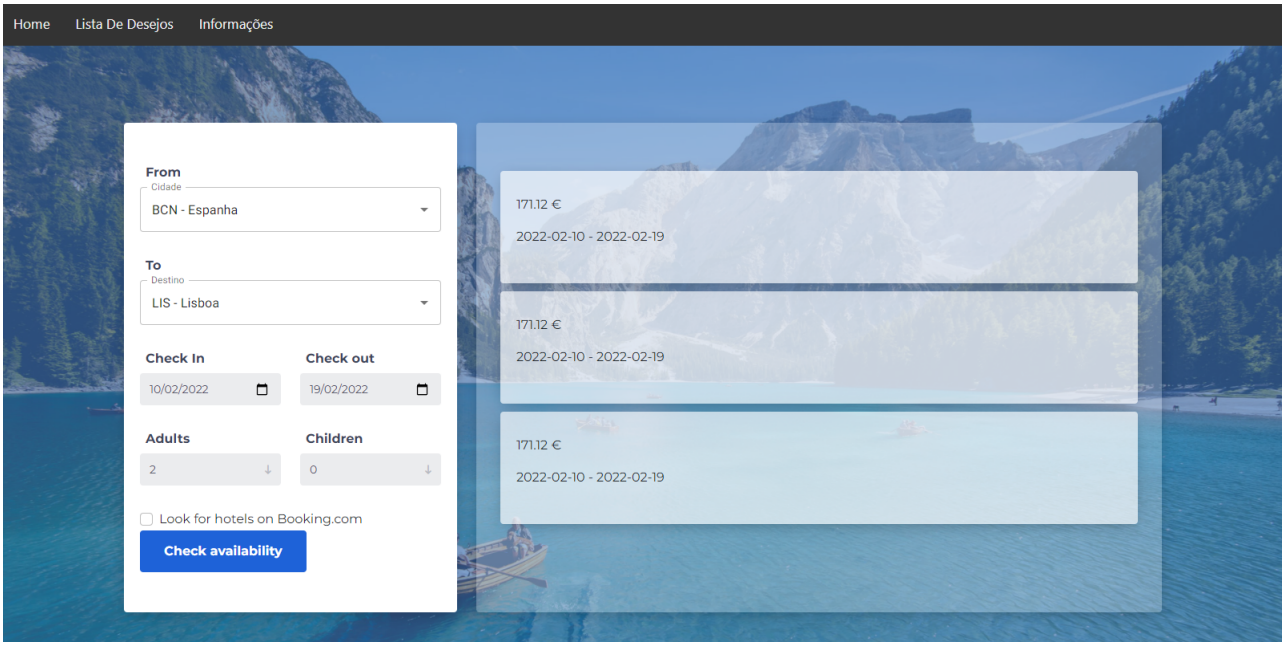


Figure 12: Protótipo aplicacional

# 6 Análise crítica do projeto realizado

Tendo em conta o desenvolvimento do projeto, salienta-se a existência de alguns pontos fortes, assim como pontos fracos. Demarca-se o facto dos casos de utilização não terem sido detalhados o melhor possível inicialmente, o que deu origem a um resultado diferente do final apresentado. Apenas foi detetado o erro nos casos de utilização quando se tinha começado a implementação dos mesmos. No entanto, como ainda se estaria numa parte prematura do projeto, o erro foi retificado e, adicionalmente foi realizada uma revisão à visão do problema e, todos os casos de utilização de forma a que, não fosse possível outro erro relacionado.

O problema deste acontecimento tem a ver com, não estando bem estruturados os casos de utilização, todas as etapas a seguir não iram estar corretas. Nesta sequência, revela-se o facto de quanto mais preliminar o problema, uma maior taxa de erros constará no projeto.

Para prevenir situações como esta, pode-se adotar uma metodologia de desenvolvimento com um dos princípios ágeis, o uso de iterações. Realizando o projeto de forma iterativa, uma equipa é de certa forma obrigada a rever os casos de utilização por cada iteração.

Houve também pontos mais fortes que, permitiram um desenvolvimento do sistema de forma simples e eficaz. A arquitetura não ficou complexa, podendo ser reproduzida em qualquer sistema independentemente das linguagens utilizadas. Isto permitiu uma tradução rápida para o JAVA. Foram impedidas improvisações de ultima hora, apenas se seguiu o diagrama e desenvolveu-se o código como o descrito.

# 7 Conclusão

Em boa verdade foi um projeto imensamente denso. No inicio, foi deveras difícil desviar o olhar das tecnologias a utilizar. Como se constrói um sistema, sem deliberar sobre a tecnologia? A verdade é que as linguagens de programação apenas ajudam a traduzir instruções para os computadores perceberem. Esta unidade curricular deu a entender que, com uma boa arquitetura, as falhas são drasticamente menores. Não estando "'preso" a uma linguagem de programação, é desenhado um sistema muito fácil de entender, com menor complexidade e, por sua vez, menor entropia.

As linguagens são ferramentas certamente importantes nos dias de hoje, mas sem uma arquitetura bem desenhada é sempre impossível a criação de sistemas reais, de grande escala, que garantem confiabilidade aos clientes.

Um aspeto inicial que pareceu estranho no inicio mas, rapidamente se entranhou foi, linguagens como o *Javascript* muitas vezes podem não recomendar o uso de programação orientada a objetos. Como se cria uma arquitetura para um programa escrito funcionalmente? A forma como se realizou no presente trabalho (na parte de *ReactJS*) foi, apenas seguir a arquitetura. Se calhar não seriam desenhadas classes, mas sim componentes que, cumpriam tudo o que estava presente na arquitetura, todavia fora de um paradigma orientado a objetos.

Como principais conceitos tem-se que, a arquitetura deve ser sempre pensada inicialmente como um problema não dependente de uma tecnologia ou plataforma, pois dependências como essas podem vir a causar um desvios na forma de resolver problemas que poderiam ser resolvidos com mais simplicidade. Tem-se também que, a análise de requisitos não deve nunca ser descartada, pelo que é daí que se determina o domínio do problema e tudo o resto. Descartar um pilar como estes leva logo a caminhos de grande entropia que, muitas vezes não levam a lado nenhum.