UNIVERSIDADE AUTÓNOMA DE LISBOA

LUÍS DE CAMÕES

Fast Ticket

Relatório de Projeto para a obtenção do grau de Licenciado

Engenharia Informática

Autores: Bernardo Gomes Rocha e José Afonso H. Oliveira

Orientador: Daniel Silvestre

Número do/a candidato/a: 20160632 e 20160605

(Página em Branco)

1 Dedicatória

A realização deste projeto contou com importantes apoios e incentivos sem os quais não se teria tornado uma realidade e aos quais estaremos eternamente gratos.

Ao Professor Doutor Daniel Silvestre, pela sua orientação, total apoio, disponibilidade, pelo saber que transmitiu, pelas opiniões e críticas, total colaboração no solucionar de dúvidas e problemas que foram surgindo ao longo da realização deste trabalho e por todas as palavras de incentivo.

A todos os docentes que ao longo do nosso percurso académico nos transmitiram os seus conhecimentos, sendo a sua maioria aplicados neste projeto.

Ao Sr. José António G. Oliveira, pelo grande apoio que deu na criação da caixa de madeira utilizada no leitor de cartões, desde a disponibilização do material utilizado, ao tempo disponibilizado para dar um apoio.

2 Resumo

Uma das grandes preocupações dos países desenvolvidos na atualidade é o nível da pegada ecológica que estes deixam. Ao longo de vários anos os países têm tentado baixar a suas pegadas, implementando medidas que melhorem esses valores, e uma das medidas geralmente implementadas é a redução do número de veículos nas grandes cidades através da promoção dos transportes públicos. Com o aumento da utilização destes meios de transporte gera-se outro problema, que é a grande afluência aos transportes públicos, que leva ao aparecimento de filas bastante grandes para a aquisição de novos bilhetes, especialmente em épocas de grande movimento turístico, isto pode levar ao aumento do número de queixas nos atendimentos de bilheteiras e deixa clientes insatisfeitos. O mesmo acontece noutros locais que sejam bastante frequentados. Neste projeto, propõem-se algumas medidas que permitem uma aquisição e utilização prática e rápidas dos bilhetes evitando assim esses problemas.

Palavras-chave: Comodidade; Rapidez; Eficácia; Flexibilidade.

3 Abstract

One of the major concerns of developed countries today is the level of ecological footprint they leave behind. Over several years, countries have been trying to lower their footprints by implementing measures that improve these values. So, one of the measures generally implemented is to reduce the number of vehicles in major cities by promoting public transport. With the increasing use of these means of transport appears another problem, the great influx of public transport, which leads to the appearance of large queues for the acquisition of new tickets, especially in times of tourist influx, this can lead to an increase in the number of complaints in the ticket line and leaves customers dissatisfied. The same happens in other places that are quite frequented. In this project, some measures are proposed that allow a practical and quick use for the acquisition of tickets, thus avoiding these problems.

Keywords: Convenience; Speed; Efficiency; Flexibility.

4 Índice

5 Lista de Fotografias/Ilustrações

6 Introdução

O projeto Fast Tickets apareceu como resposta a uma necessidade encontrada no mercado. Hoje em dia quando pretendemos utilizar os transportes públicos de uma forma ocasional, temos a necessidade de adquirir um cartão que tem um valor acrescentado ao preço do bilhete, para turistas ou pessoas que vão estar temporariamente num dado local, é um custo adicional que achamos que pode ser evitado. Para além disso, utilizadores que queiram fazer uma viagem, pré-carregando os cartões, que requeira alguns transbordos entre várias operadoras, vão precisar de ter vários cartões, uma para cada operadora.

Uma alternativa a não precisar de comprar cartões extras é comprar o bilhete após utilizarem a viagem, o que é uma situação que não é muito viável, porque apesar de se poupar no número de cartões que se compra, temos de dispensar tempo para ir para as filas para adquirir o próximo bilhete.

O nosso projeto visa complementar o sistema de transportes atual, implementando uma aplicação onde o utilizador pode comprar, gerir e usufruir dos seus bilhetes comodamente á distância de um clique. Este sistema permite ao utilizador pré-comprar os bilhetes e escolher no momento da validação qual dos bilhetes pretende utilizar para o percurso que vai fazer.

7 Planeamento

Inicialmente para criarmos um bom planeamento do projeto, começou-se por criar um plano no MS Project que contem todas as tarefas que são necessárias fazer. Definiu-se, quem fica responsável por uma dada tarefa de modo a que tudo fosse feito dentro dos prazos. Esse project tornou tudo mais fácil, pois permitiu uma boa organização entre os intervenientes porque permitiu que não houvesse dúvidas de quem fazia o quê, e pudéssemos trabalhar ao mesmo tempo em tarefas que se interligavam, e assim terminar tudo nos prazos estabelecidos. Em termos de horas disponibilizadas pelos intervenientes, previu-se que cada um deles disponibiliza-se em média 251 horas para este projeto, como podemos na Figura 1.

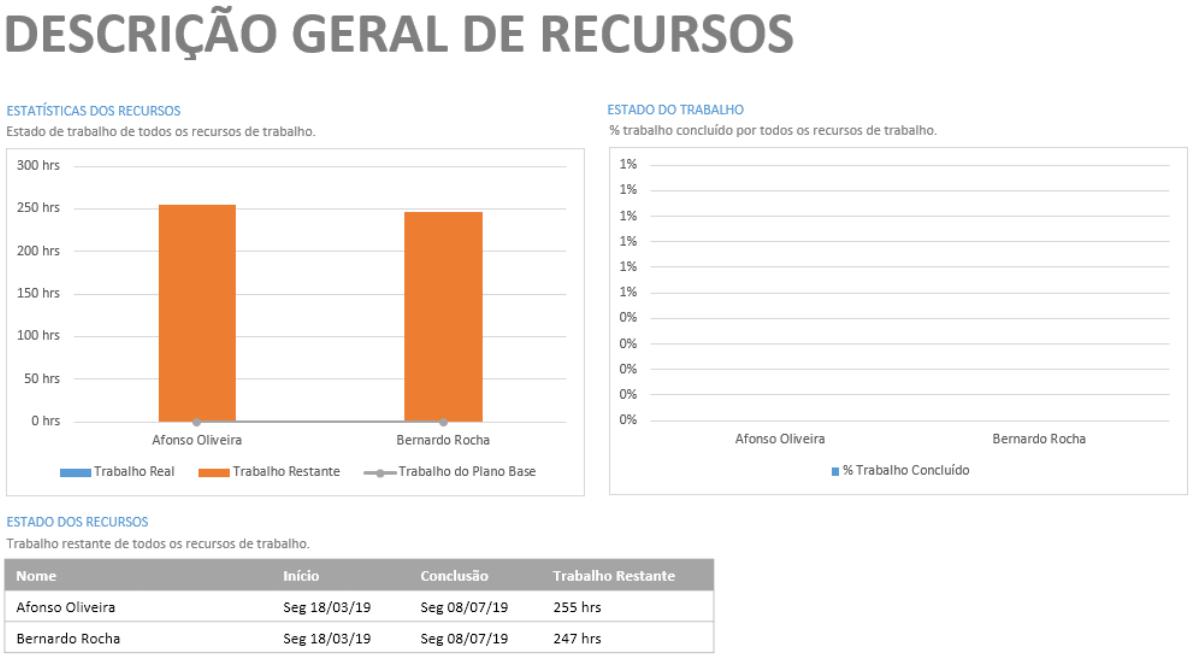


Figura 1 - Descrição geral de recursos

Começou-se o projeto pelo planeamento, onde identificámos que existem 4 entidades, sendo elas: Servidor de API, a Aplicação para Android, o Website e o Leitor de cartões como podemos ver na Figura 2.

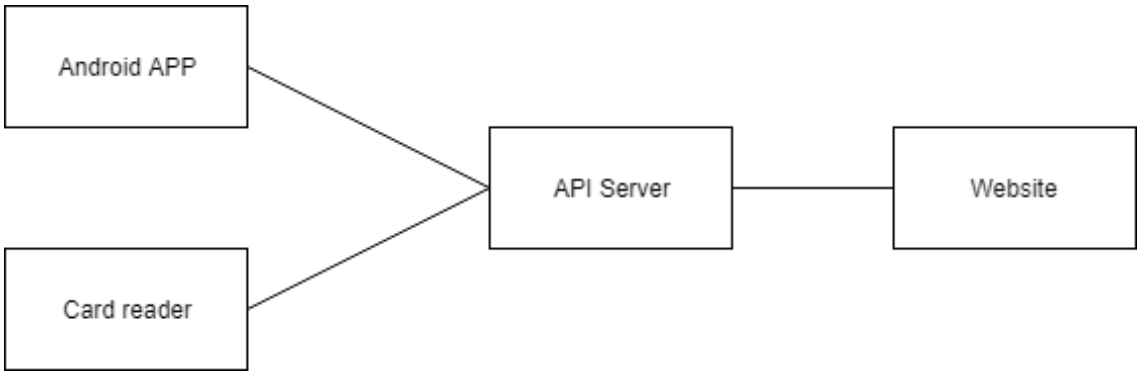


Figura 2 - Diagrama do projeto

7.1 Servidor de API

O servidor de API está responsável por consultar a informação na base de dados e facultar esse dado a aplicações externas. Esta API é composta por duas interfaces, a interface que é utilizada no leitor de cartões e a interface que é utilizada pela Aplicação Android. Na interface das máquinas pretende-se que estas tenham apenas permissão para:

- Validar a entrada de um bilhete;
- Validar a saída de um bilhete.

Na interface que é utilizada para a aplicação Android pretende-se que disponha das seguintes funcionalidades:

- Consultar a lista de empresas registadas;
- Consultar a lista de tipos de serviços disponibilizados;
- Consultar a lista de pacotes que uma dada empresa tem para oferecer;
- Consultar a lista de bilhetes do utilizador autenticado;
- Adquirir novos bilhetes.

Ambas as interfaces têm sistema de autenticação, mas que são distintos um do outro. No caso da aplicação utilizada pelos utilizadores, estes devem de se autenticar utilizando o email e a senha utilizadas durante a fase de registo. Após a autenticação ser efetuada, é utilizada o método de autenticação por token, que consiste na disponibilização de uma sequência de caracteres única que garante o acesso a aplicação durante o tempo estipulado como validade, assim cada vez que o utilizador pretende aceder á API não precisas estar a enviar constantemente os seus dados, bastando apenas enviar a token de sessão.

No caso dos terminais de validação, estes utilizam um método diferente, este utiliza o método de API Key, cada máquina tem associada a si uma chave única e uma chave secreta que servem como método de autenticação da mesma.

7.2 Website

O objetivo do website é disponibilizar interfaces ao utilizador de modo a que este possa interagir com os vários sistemas através do seu computador. Desde comprar até gerir os seus próprios bilhetes. Para isso o site dispõe de uma loja com todos os bilhetes e vários filtros para tornar a pesquisa fácil e rápida para o utilizador. Em termos de pagamento é feito de uma forma fácil, rápida e segura. Este também dispõe de um espaço para os utilizadores poderem visualizar quais são os bilhetes que já adquiriu.

7.3 Leitor de cartões

Para melhorar a funcionalidade dos leitores de cartões e para os tornar mais cómodos aos utilizadores, definiu-se 2 interfaces de input de bilhetes por parte do utilizador. Esses dois métodos de input são: a validação do bilhete por código QR e a validação do bilhete por NFC. Ambas as interfaces têm as suas vantagens e desvantagens, o NFC é uma operação mais rápida e mais prática, mas tem a desvantagens de alguns dispositivos, especialmente os mais antigos, não suportarem este método. O caso do código QR é um método que funciona em praticamente todos os dispositivos, mas é mais demorado porque requer o posicionamento do código de modo a que o leitor o possa ler. Estes dois métodos foram escolhidos e utilizados de modo a que ambos se completassem para melhorar a experiência do utilizador da melhor maneira possível.

Para além destes dois métodos de input pelo utilizador pretende-se que este tenha um feedback (output) da operação. Esse feedback é dado por LED’s, por um LCD que informa o resultado da operação em forma textual, e um aviso sonoro que informa se o bilhete foi validado ou não. Este aviso sonoro também é útil a nível de acessibilidades, pois permite que pessoas invisuais possam utilizar as máquinas e obter um feedback idêntico às restantes. O esquema utilizado para o leitor de cartões pode ser visto na Figura 3.

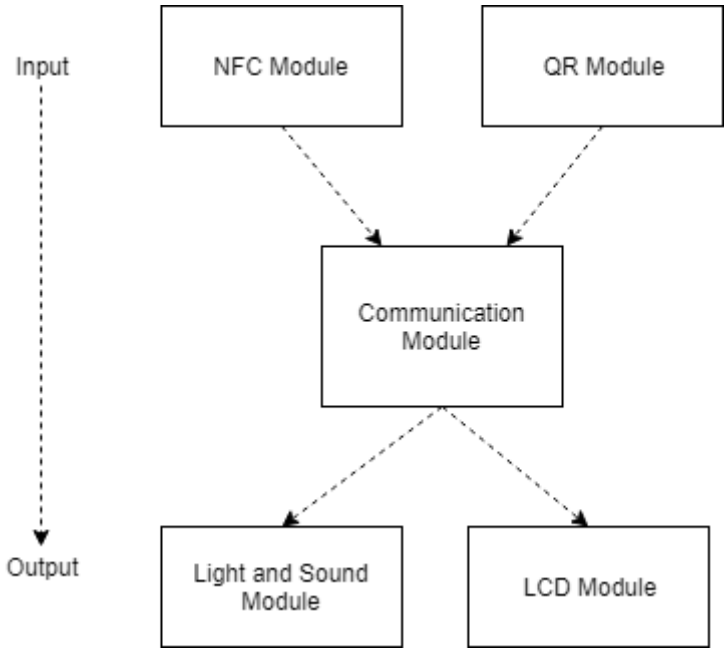


Figura 3 - Esquema utilizado no planeamento do Leitor de cartões

7.4 Aplicação para Android

Pretende-se que a aplicação para Android permita ao utilizador comprar os bilhetes, gerir os seus bilhetes e por usufruir dos seus bilhetes. Ao efetuar a compra de um bilhete, pretende-se que o utilizador indique para que categoria pretende o bilhete, por exemplo, se é um bilhete de transportes terrestre, se é um bilhete de aviação, teatro, entre outros.

Após a escolha da categoria a que este pertence, pretende-se mostrar todas as empresas que oferecem produtos dentro dessas categorias, e a respetiva oferta de cada uma. Ao comprar um pacote/oferta o utilizador receberá o seu bilhete na sua conta pessoal para poder usufruir deste.

Para validar um bilhete pretende-se que também seja simples e rápido, bastando aceder á sua conta pessoal, escolher um dos bilhetes já adquiridos e escolher como o pretende validar, se por NFC ou QR Code.

Figura 4 - Exemplo de código QR

Figura 5 - Tecnologia NFC

8 Desenvolvimento

8.1 Servidor de API

O servidor de API foi desenvolvido com a finalidade de disponibilizar a informação ao site, á aplicação Android e á máquina de validação. Isto permite que todos os serviços tenham a mesma base e assim o sincronismo de informação torna-se um processo mais fácil e rápido, porque os serviços apenas têm de fazer pedidos http ao servidor e este dar-lhes á respostas para que possam ser apresentadas aos utilizadores. Durante o desenvolvimento desta API foram consultadas várias documentações que estão referenciadas na web grafia nos pontos IV, V, VI, VII e VIII.

O servidor de API é onde está localizada a base de dados que está dividida em 9 tabelas, Django user, Company, Ticket machine, Package type, Package, Ticket, Config value, Config key, Config data como podemos ver o esquema completo na Figura 6.

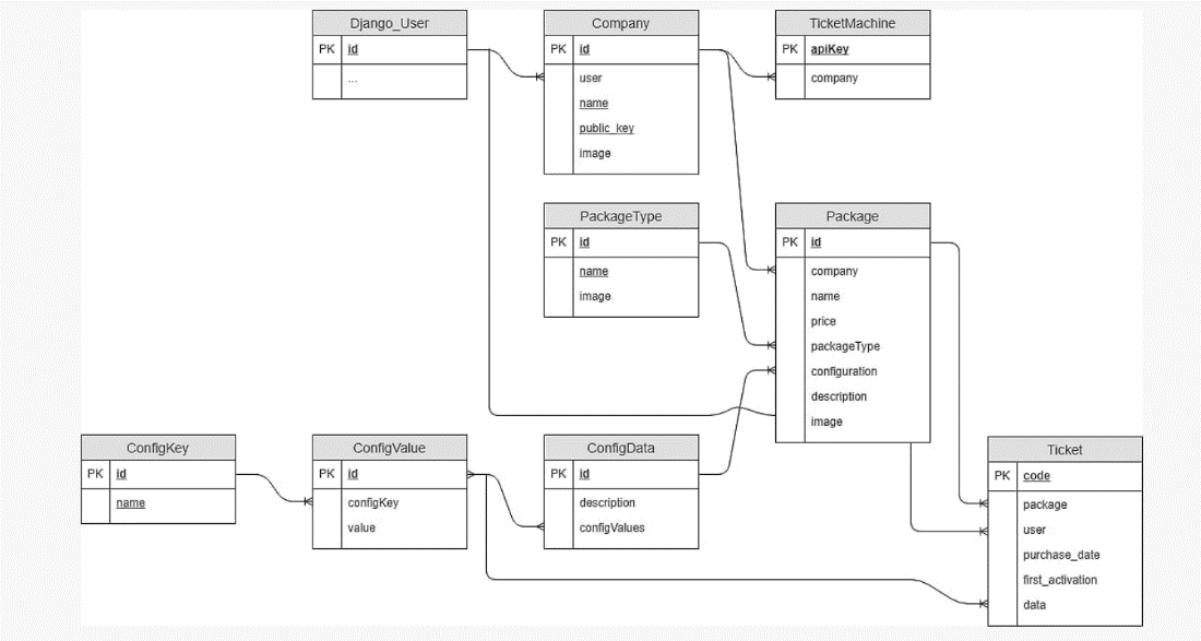


Figura 6 - Esquema da base de dados

Django user é uma tabela criada pelo Django que guarda todos os utilizadores registados no projeto, bem como os dados associados com este. Company guarda todas as empresas registadas para depois as enviar para os vários clientes. Ticket machine é a tabela criada para armazenar a informação relativa às máquinas que validam os bilhetes, é também nesta tabela que se faz a associação entre a máquina e a empresa para o qual esta máquina trabalha. Package type que guarda as categorias dos serviços/pacotes. Package que guarda os pacotes inseridos por cada uma das empresas. Ticket que guarda as informações acerca de todos os bilhetes, desde

quem o comprou, quando o comprou, quando foi ativado pela primeira vez, que pacote/serviço foi comprado. No caso da tabela ConfigData armazenam as configurações dos pacotes/serviços. A tabela ConfigValue e ConfigKey armazenam os valores e as chaves das possíveis configurações dos pacotes/serviços ou bilhetes. Todas estas tabelas são criadas no ficheiro models.py da machine_api como podemos ver na Figura 7.

```
  1  import binascii
  2  import os
  3
  4  from django.db import models
  5
  6  from django.contrib.auth.models import User
  7  from rest_framework_api_key.models import APIKey
  8
  9  class Company(models.Model):
 39
 40  class TicketMachine(models.Model):
 51
 52  class ConfigKey(models.Model):
 63
 64  class ConfigValue(models.Model):
 77
 78  class ConfigData(models.Model):
 91
 92  class PackageType(models.Model):
105
106  class Packages(models.Model):
141
142  class Ticket(models.Model):
```

Figura 7 - Ficheiro models.py localiza na machine api

Em termos da base dados, esta passou por vários estados de otimização antes de chegar ao resultado final. Começou-se por perceber quantas entidades estão presentes para serem criadas as tabelas, de seguida identificou-se quais seriam as relações entre ambas as tabelas, de modo a que a consulta fosse o mais rápida possível.

O servidor API esta dividido em duas interfaces, a interface das máquinas representada na Figura 8 e a interface dos utilizadores representado na Figura 9. A interface das máquinas tem como foco a máquina de validação de bilhetes e a interface do utilizador contém tudo o que se refere aos utilizadores.

Figura 8 - Pasta da API para máquinas

Figura 9 - Pasta da API para utilizadores

O ficheiro views.py da interface das máquinas que está representado na Figura 10 contém apenas duas funções, o *checkinTicket* que vai verificar se um dado bilhete pode ser validado e se puder valida-o e o *checkoutTicket* que vai fazer o checkout de um dado bilhete.

```
@api_view(['POST'])
@permission_classes((HasAPIKey,))
def checkInTicket(request):

@api_view(['POST'])
@permission_classes((HasAPIKey,))
def checkOutTicket(request):
```

Figura 10 - Ficheiro views.py da interface das máquinas

Para o machine API foi criada uma função a ApiUtil como podemos ver na Figura 11 que contem as funções básicas que vão ser necessárias.

```
# Create the APIUtil
apiUtil = APIUtil(request)
```

Figura 11 - Construtor da classe ApiIUtil utilizada nas views

Essas funções como podemos ver com mais detalhe na Figura 12 são retornar um token que está a ser utilizado, obter a machine em uso ou obter um dado bilhete.

```python
from rest_framework_api_key.models import APIKey

from machine_api.models import TicketMachine, Ticket, Packages

DEFAULT_ERROR_KEY = 'error'
DEFAULT_DATA_KEY = 'data'

class APIUtil:

    def __init__(self, request):
        """ Create an instance of the API Utils """
        self.request = request

    def getAPIToken(self):
        """ Get the API Token from the request """
        return self.request.META.get('HTTP_API_TOKEN')

    def getMachine(self):
        """ Get the API Token from the request """
        token = self.getAPIToken()

        # Get the api key
        apiKey = get_object_or_404(APIKey, token=token)

        # Get the machine
        machine = get_object_or_404(TicketMachine, apiKey=apiKey)

        return machine

    def getTicket(self):
        """ Get the ticket from the user request """

        # Get the code for the ticket
        code = self.request.data.get('code')

        # Check if the code is invalid
        if code is None:
            raise Http404

        # Get the machine
        machine = self.getMachine()

        # Get the ticket
        ticket = get_object_or_404(Ticket, code=code, package__company=machine.company)

        return ticket
```

Figura 12 - Funções básicas da ApiUtil

A User API é a parte do servidor que esta responsável por tudo o que é referente aos utilizadores e por isso já têm um número de funções significativamente maior.

Figura 13 - Ficheiro views.py do user_api

Na Figura 13 podemos visualizar que todas as funções têm o *api_view*, *authentication_classes* e o *permission_classes*. O *api view* é o modo como a informação vai ser enviada se é por GET ou POST, a *authentication_classes* é a forma como o cliente deve-se authenticar, se é por sessão ou por token e o *permission_classes* serve para indicar quem tem permissão para aceder a esta página, neste caso o *IsAutenticated* quer dizer que este utilizador tem de estar autenticado e não pode aceder como anónimo.

As várias funções como o *aboutUser* que retorna as informações do utilizador que está atualmente autenticado. O *listCompanies* consultar as empresas existentes. O *listTickets* lista os tickets existentes para um dado utilizador. O *listpackegetype* lista as categorias que podemos encontrar. O *listpackagesforcompany* lista os vários pacotes de bilhetes que cada empresa tem para oferecer. E por fim o *peformpayment* que vai ser a função chamada cada vez que o

utilizador quer efetuar uma compra e tem como função fazer o processamento da compra do bilhete.

Na função *about_user* que podemos ver na Figura 14 existe uma função que é a *serializer* que tem como função delinear quais são os campos necessários de retornar.

```
def aboutUser(request):
    serializer = UserSerializer(request.user)
    return JsonResponse(serializer.data)
```

Figura 14 - Função serializer criada para o about user

```
from django.contrib.auth.models import User, Group
from rest_framework import serializers

from django.contrib.auth.models import User

from machine_api.models import PackageType, Company, Packages, Ticket, ConfigKey, ConfigValue, ConfigData
from django.conf import settings
import socket

class UserSerializer(serializers.ModelSerializer):
    class Meta:
        model = User
        fields = ('url', 'username', 'email', 'groups')

class GroupSerializer(serializers.ModelSerializer):
    class Meta:
        model = Group
        fields = ('url', 'name')

class CompanySerializer(serializers.ModelSerializer):
    class Meta:
        model = Company
        fields = ('name', 'public_key', 'image')

class ConfigKeySerializer(serializers.ModelSerializer):
    """ Serialize the config key information """
    class Meta:
        model = ConfigKey
        fields = ('name',)

class ConfigValueSerializer(serializers.ModelSerializer):

    name = serializers.CharField(source='configKey.name')

    """ Serialize the ticket information """
    class Meta:
        model = ConfigValue
        fields = ('name', 'value')

class PackageTypeSerializer(serializers.ModelSerializer):
    """ Serialize the package type information """
    class Meta:
        model = PackageType
```

Figura 15 - Funções contidas no serializer

Com estas funções que vemos na Figura 15 podemos escolher quais são os campos que queremos que o cliente tenha acesso e assim não precisamos de ter campos presentes que este não precisa de ter acesso.

8.2 Website

O website pretende criar uma interface para o utilizador interagir com o sistema que seja fácil e rápida de usar. Para isso como framework foi utilizado o Django que é uma framework de alto-nível em python, muito fácil de usar e que resolve com as suas funcionalidades muitos problemas do web development para que assim possamos abstrair de partes mais básicas de desenvolvimento e focar nas partes mais importantes como a estratégia de negócio. Durante o desenvolvimento do website foi consultada a documentação do Django que pode ser vista na web grafia IV. Inicialmente começamos por escolher e desenvolver a cara do nosso website, ou seja, o template para isso utilizamos HTML e CSS para que esteticamente o template ficasse como queríamos e que conseguisse conter todas as funcionalidades que queríamos para o website.

Começamos por dividir tudo o que e estático no site para uma pasta *static* que contém tudo o que pertence a CSS, imagens, JavaScript e fontes como podemos ver na Figura 16. Esta divisão é bastante útil, pois permite uma flexibilidade no site e uma boa organização, caso haja necessidade de fazer uma alteração apenas precisa-se de mudar num único sítio.

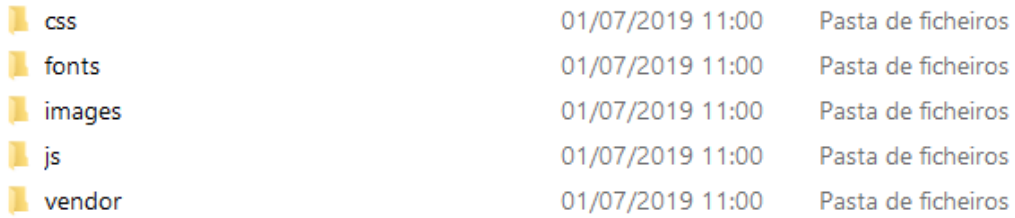


Figura 16 - Pastas estáticas do website

Para facilitar o desenvolvimento dividimos em vários templates base que podem ser visualizados na Figura 17 que serviam todas as páginas do website como o header e o footer.

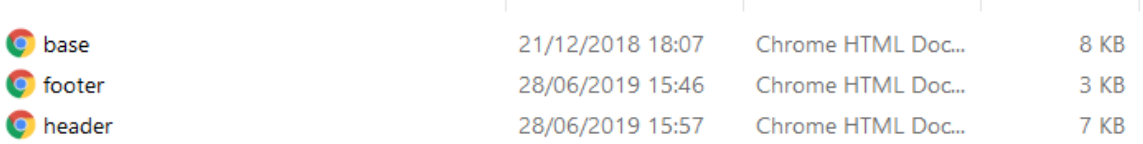


Figura 17 - Templates base

E depois basta utilizar os placeholders do Django, que podemos encontrar na Figura 18, que nos permite reutilizarmos o código, evitando a repetição.

```
<!-- Header -->
{% include "emarket/base/header.html" %}

<!-- Content -->
{% block content %}{% endblock %}

<!-- Footer -->
{% include "emarket/base/footer.html" %}
```

Figura 18 - Placeholders do Django

O template começa por uma página inicial atrativa e que mostra um pouco do que o nosso site contém através de imagens interativas e botões com ligações para as várias páginas que este dispõe. De seguida temos a página da loja em que o utilizador começa por escolher qual é o tipo de bilhete que pretende (transportes, museus, teatro, cinema etc.) em seguida seleciona a empresa que contém o bilhete pretendido e por fim escolher o seu bilhete, claro que caso o utilizador não pretenda percorrer este caminho e queira chegar mais rápido ao bilhete existem filtros que podem ser utilizados. Depois de ser efetuada alguma compra, esses bilhetes ficam registados na zona de gestão de bilhetes, que pode ser encontrada na área do cliente. Esta página foi criada para que os utilizadores pudessem ter uma gestão dos bilhetes já adquiridos e dos que ainda precisam de comprar. Por fim temos a página do sobre que fala um bocadinho da missão da empresa e das suas finalidades.

Para que o website fosse dinâmico todas as *queries* foram feitas nas views e colocadas em variáveis que depois são passadas para o template. Na Figura 19 podemos visualizar o ficheiro *views.py* do website que esta dividido por funções e cada função está destinada a uma página do site assim podemos passar simplesmente as variáveis necessárias ao que vai ser apresentado na página.

```python
class SignUp(generic.CreateView):
    form_class = UserCreationForm
    success_url = reverse_lazy('login')
    template_name = 'signup.html'

def index(request):
    empresas = Company.objects.all()
    type = PackageType.objects.all()
    return render(request, 'emarket/emarket/index.html', {'type': type, 'empresas': empresas})

def loja(request):
    empresas = Company.objects.all()
    type = PackageType.objects.all()
    company = Company.objects.all()
    return render(request, 'emarket/emarket/tipos_de_bilhetes.html', {'type': type, 'company': company,  'empresas':empresas})

def gestao_bilhetes(request):
    empresas = Company.objects.all()

    if request.user.is_authenticated:

        bilhetes = Ticket.objects.filter(user = request.user)
        return render(request, 'emarket/emarket/gestao_bilhetes.html', {'bilhetes': bilhetes, 'empresas':empresas})


    else:
        return redirect('/website/accounts/login/')

def loja_empresas(request):
    empresas = Company.objects.all()
    type = PackageType.objects.all()
    company = Company.objects.all()
    company_type = request.GET.get("Type", None)
    print(company_type)
    if company_type is None:
        company_display = Company.objects.all()
    else:
        company_display = Company.objects.prefetch_related('packages_set').filter(packages__packageType__name= request.GET.get("Type")).distinct()

    return render(request, 'emarket/emarket/empresas.html', {'type': type, 'company': company, 'company_display':company_display, 'empresas':empresas})

def sobre(request):
    empresas = Company.objects.all()
    return render(request, 'emarket/emarket/about.html', {'empresas':empresas})

def bilhetes(request):
    empresas = Company.objects.all()
    type = PackageType.objects.all()
    company = Company.objects.all()
    todos_bilhetes = request.GET.get("company", None)
    if todos_bilhetes is None:
        bilhete = Packages.objects.all()
```

Figura 19 - Código utilizado nas Views do website

Nos templates são feitos loops for como podemos ver na Figura 20 para percorrer todas as instâncias e apresentá-las visualmente. Esta forma dinâmica de apresentar o produto torna o código muito mais simples pois os produtos não têm de ser apresentados um a um, mas sim todos de uma vez o que os ajuda a reciclar código e torná-lo mais limpo.

```
{% for types in type %}
<li class="p-t-4">
    <a href="loja_empresas?Type={{types.name}}" class="s-text13 active1">
        {{ types.name }}
    </a>
</li>
{% endfor %}
<br>
<h4 class="m-text14 p-b-7">
    Empresas
</h4>
{% for companys in company %}
<li class="p-t-4">
    <a href="bilhetes?company={{companys.name}}"  class="s-text13 active1">
        {{ companys.name }}
    </a>
</li>
{% endfor %}
```

Figura 20 - Código utilizado no template com ciclos for

No website para cada página foi criado um URL que pode ser consultado na Figura 21.

```
from django.contrib import admin
from django.urls import include, path
from django.views.generic.base import TemplateView


from . import views

urlpatterns = [
    path('', views.index),
    path('accounts/', include('django.contrib.auth.urls')),
    path('signup/', views.SignUp.as_view(), name='signup'),
    path('home', TemplateView.as_view(template_name='home.html'), name='home'),
    path('loja', views.loja),
    path('loja_empresas', views.loja_empresas),
    path('sobre', views.sobre),
    path('bilhetes', views.bilhetes),
    path('gestao_bilhetes', views.gestao_bilhetes),
    path('bilhetes/<int:id>/', views.product_detail),

]
```

Figura 21 - Ficheiro URL.py do website

Em termos de segurança o site dispõe de um sistema de autenticação do usuário do próprio Django, este sistema manipular contas de usuários, tipos de usuários, permissões que cada um dispõe e as sessões baseadas em cookies.

Para as empresas que queiram ver os seus bilhetes apresentados no nosso website cada uma é lhes atribuído um usuário de empresa que lhes permite aceder a zona onde poderão dizer quais os bilhetes que pretendem vender e os preços. E depois serão apresentados de forma dinâmica no site.

O website utiliza o server api como base para lhe fornecer todas as informações que precisa e assim só existe a preocupação de como elas serão apresentadas aos utilizadores, a api é partilhada por todos os serviços do nosso projeto. E isso permite que existe um sincronismo entre os diferentes serviços.

Em termos visuais o site ficou com o aspeto que podemos ver na Figura 22, Figura 23, Figura 24 e Figura 25.

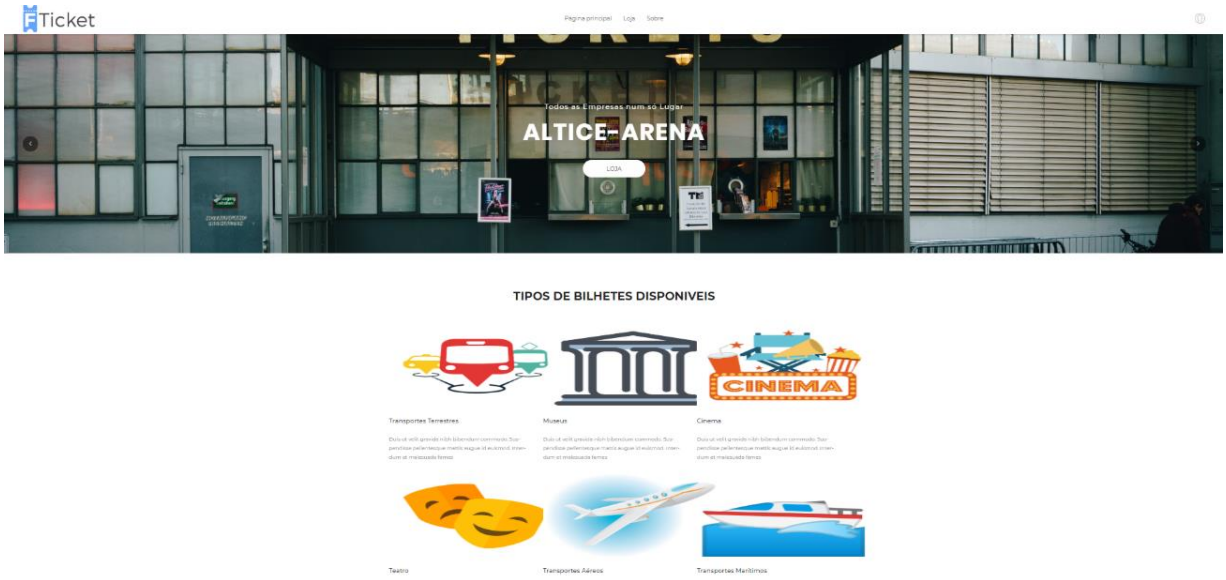


Figura 22 - Página inicial do website

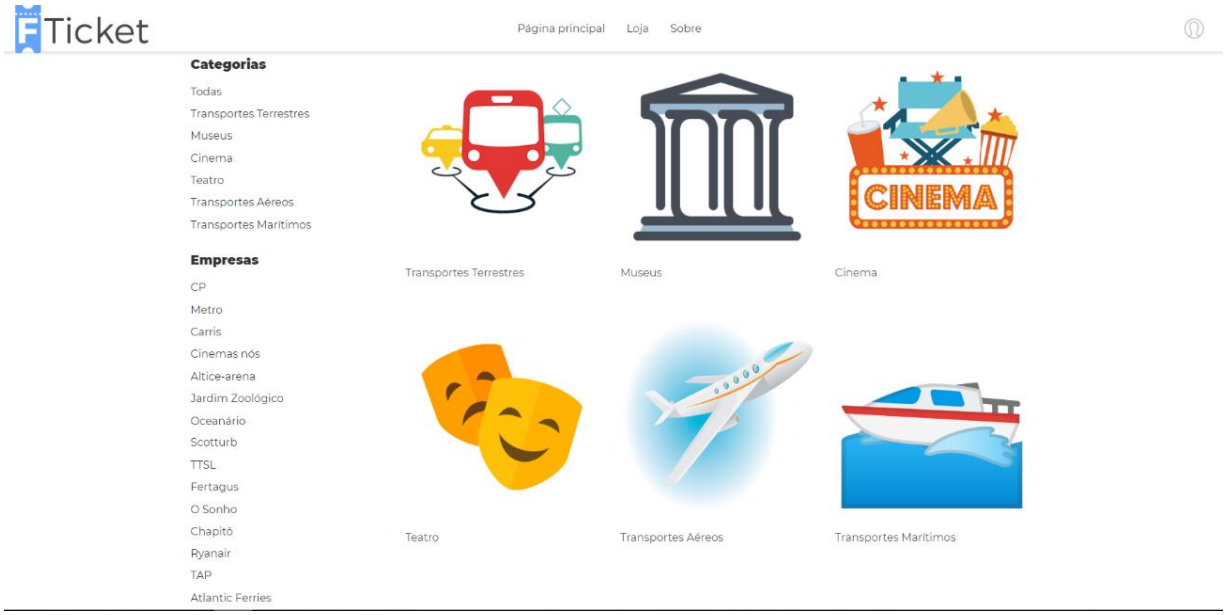


Figura 23 - Página da loja do website

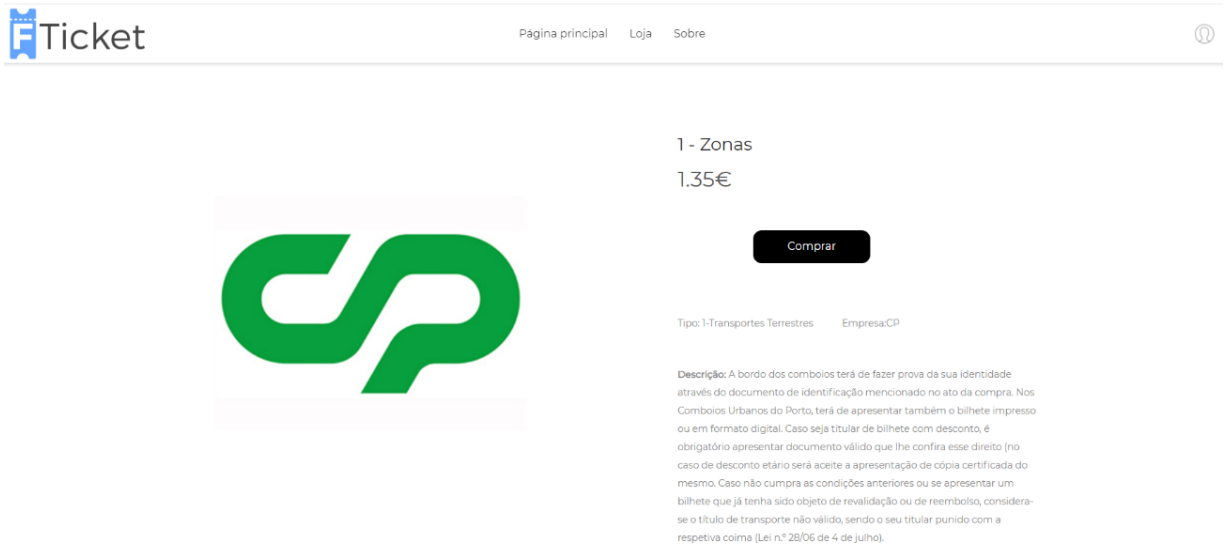


Figura 24 - Página do detalhe do bilhete do website

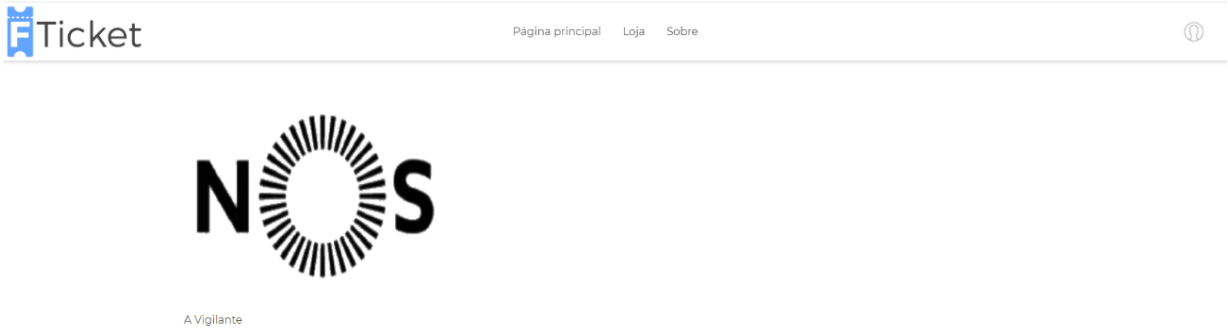


Figura 25 - Página da gestão de bilhetes do website

8.3 Leitor de cartões

O leitor de cartões pretende recriar uma porta de controlo de acessos utilizada nos transportes e consiste num Raspberry PI que corre um software desenvolvido em Python, que lê códigos QR que contêm a informações acerca de bilhetes e que leia mensagens NDEF com informações acerca do bilhete que são enviadas através do sistema NFC de um dispositivo Android. Durante o desenvolvimento foi utilizado como apoio a documentação do *Raspberry PI* que está presente na web grafia no ponto IX.

Em termos de plano a nível de hardware, foi utilizado no leitor de cartões pode ser visto na Figura 26 e Figura 27. Em termos de ligações do PN532 utilizou-se o manual deste para identificar qual seria a ordem de ligação dos cabos, que pode ser consultado na web grafia I.

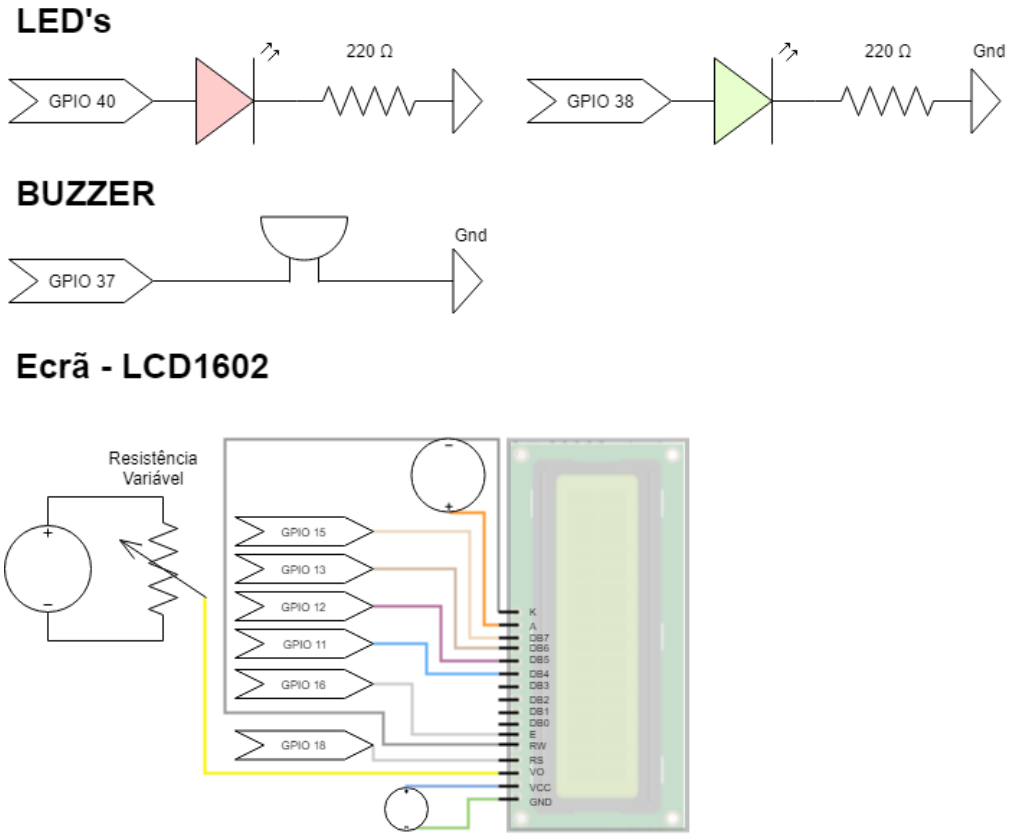


Figura 26 - Constituição do hardware - Parte 1

Figura 27 - Constituição do hardware - Parte 2

Os LEDs e o LCD estão responsáveis por disponibilizar um feedback visual aos utilizadores do estado do seu bilhete, se este for verde significa que o bilhete foi validado com sucesso e caso seja vermelho significa que não foi validado. Em ambas as situações o LCD dá o feedback da ação, sendo o positivo uma mensagem a informar que o bilhete foi validado com sucesso, e em caso de negativo, informa qual foi a situação que o levou a recusar esse bilhete.

O Buzzer está responsável por fornecer o feedback auditivo aos utilizadores, de modo a ser inclusivo também a invisuais. Caso a operação seja afirmativa (o bilhete tenha sido validado) o som será um beep, caso a ação tenha sido negativa, o som serão 3 beeps intervalados.

O PN532 e a câmara estão responsáveis por ler os bilhetes do utilizador, o PN532 é utilizado para ler a informação recebida por mensagem NFC e a câmara está responsável por

ler os códigos QR. Neste leitor de cartões estão presentes vários módulos que podem ser vistos na Figura 28.

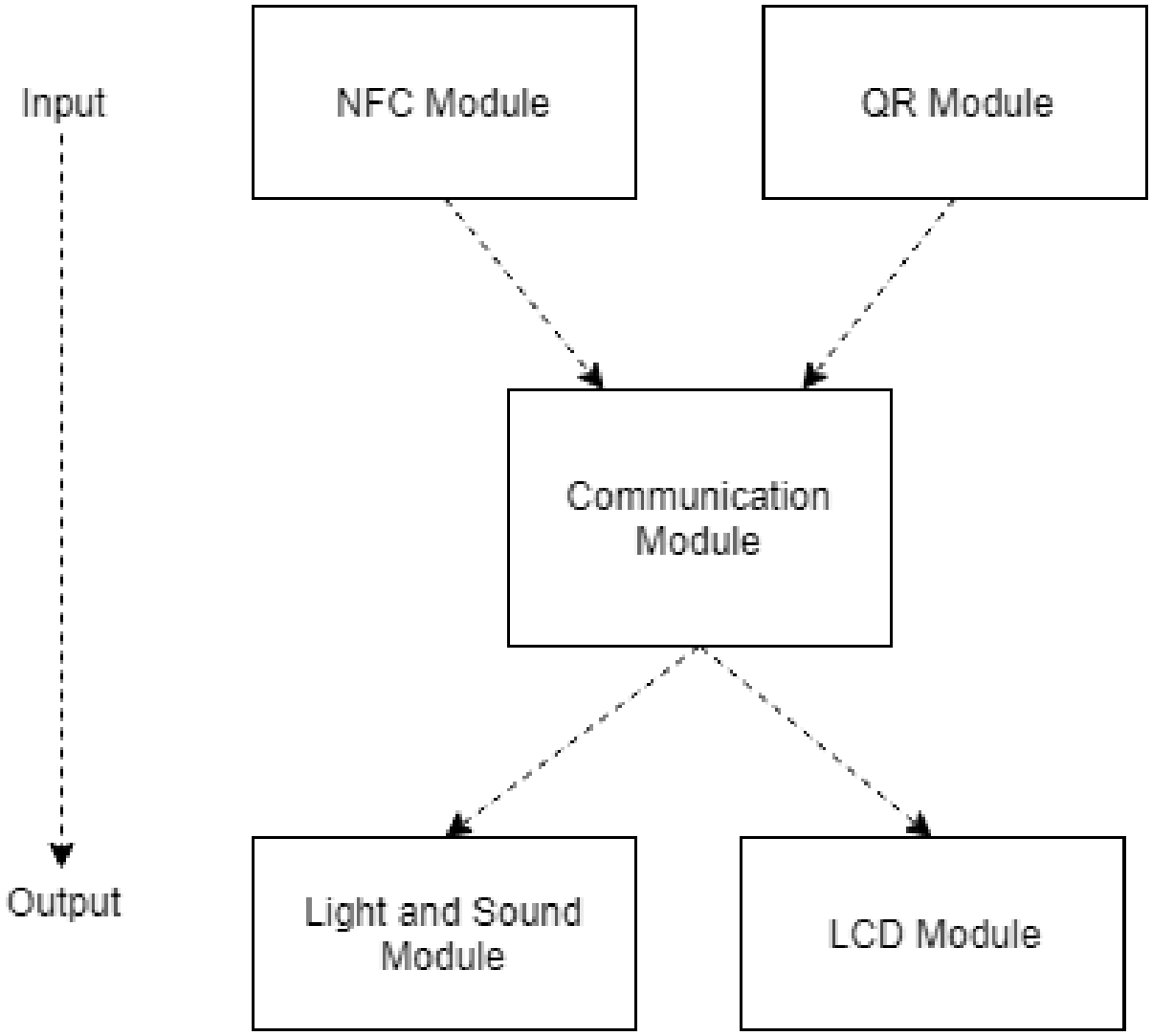
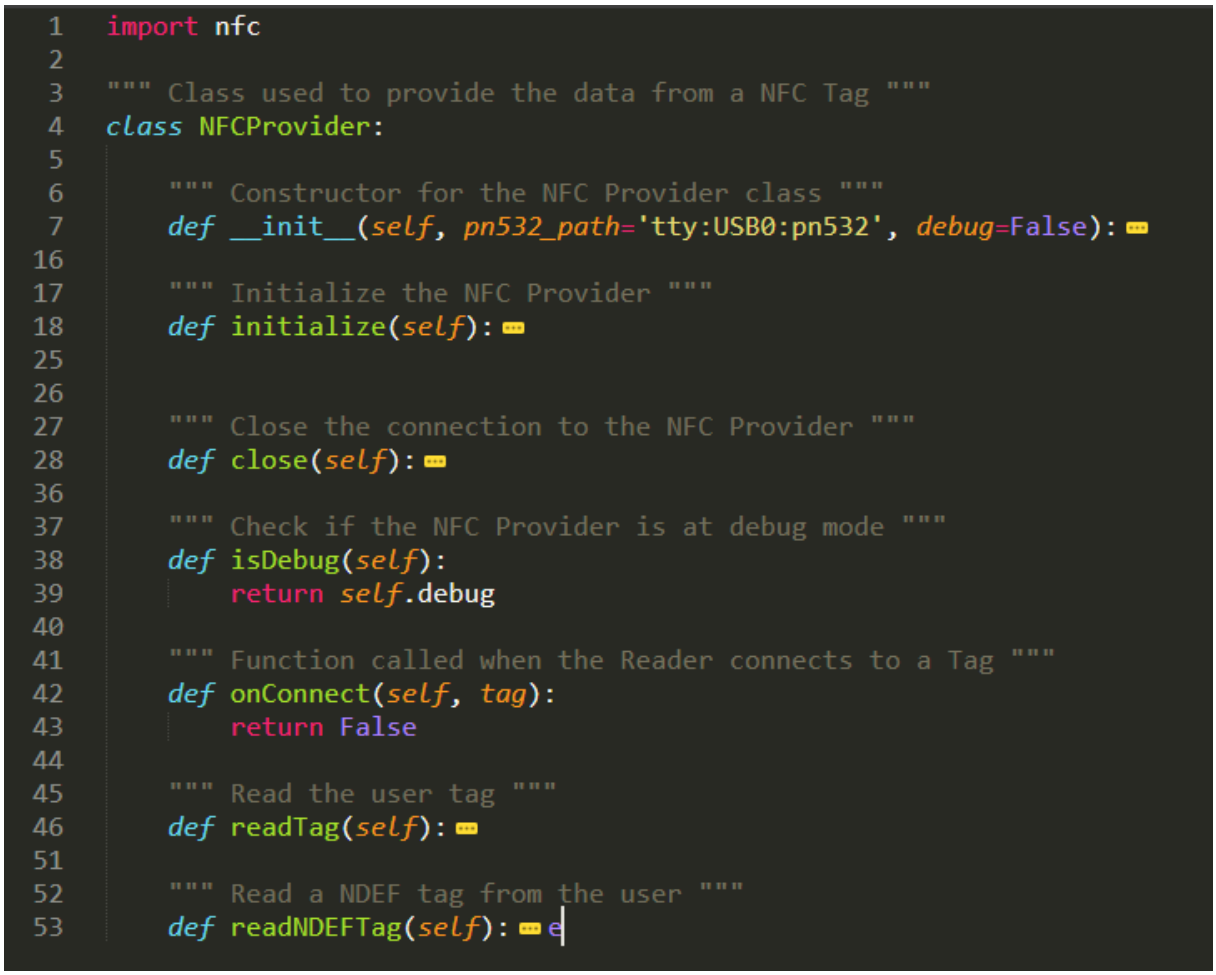


Figura 28 - Diagrama da comunicação do Leitor de cartões

O módulo do NFC está responsável pela comunicação entre o hardware PN532 e o software da máquina. Este módulo está presente no ficheiro NFCModule.py. Esta classe tem as funções essenciais para o sistema, como a inicialização da placa, término da mesma e a leitura de uma tag. Esta classe pode ser vista na Figura 29.



```python
import nfc

""" Class used to provide the data from a NFC Tag """
class NFCProvider:

    """ Constructor for the NFC Provider class """
    def __init__(self, pn532_path='tty:USB0:pn532', debug=False):...

    """ Initialize the NFC Provider """
    def initialize(self):...


    """ Close the connection to the NFC Provider """
    def close(self):...

    """ Check if the NFC Provider is at debug mode """
    def isDebug(self):
        return self.debug

    """ Function called when the Reader connects to a Tag """
    def onConnect(self, tag):
        return False

    """ Read the user tag """
    def readTag(self):...

    """ Read a NDEF tag from the user """
    def readNDEFTag(self):...e
```

Figura 29 - Classe NFCProvider presente no NFCModule.py

O módulo QR consiste numa classe utilizada para processamento das imagens vindas da câmara, de modo a ser identificado um código QR numa dada fotografia, permitindo também ler mais do que um código de cada vez. Esta classe pode ser vista na Figura 30.

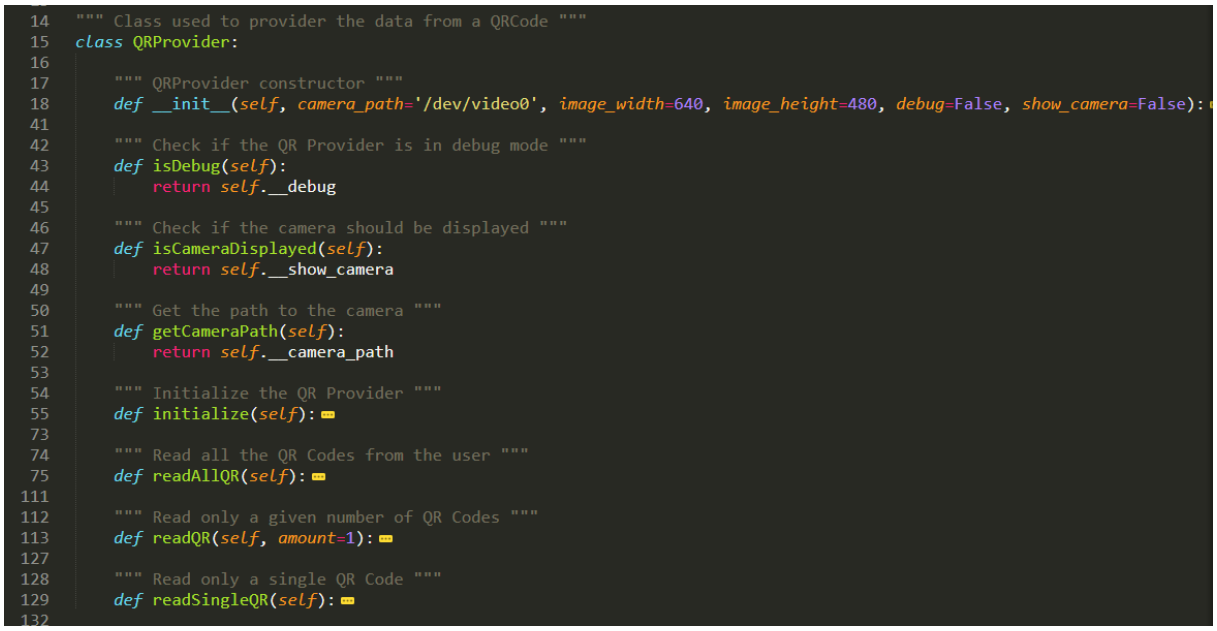


Figura 30 - Classe QRProvider presente no QRModule.py

O módulo de comunicação é o que está responsável por comunicar com o servidor da API, para validar um dado bilhete. Este módulo por cada pedido que faz ao servidor de API para validar um bilhete, também envia os dados de autenticação da máquina em cabeçalho do pedido HTTP. Esta classe pode ser vista na Figura 31.

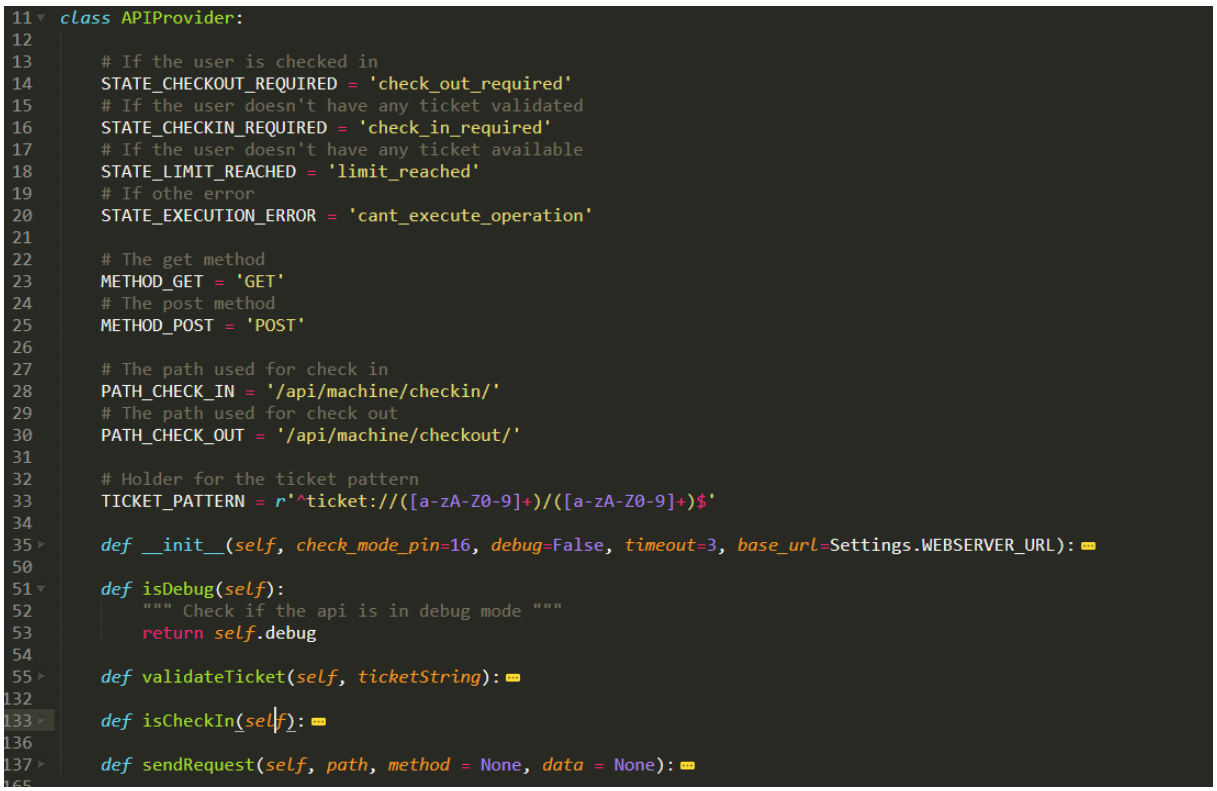


Figura 31 - Classe APIProvider presente no CommunicationModule.py

O módulo iluminação é o módulo que está responsável por controlar os LED's e o Buzzer. Esta classe pode ser vista na Figura 32.

```
class AccessProvider:

    """ GPIO for the afirmative status """
    GPIO_LED_GREEN = 21
    """ GPIO for the negative status """
    GPIO_LED_RED = 20
    """ GPIO for the buzze """
    GPIO_BUZZER = 26
    """ The beep duration """
    BUZZER_BEEP_DURATION = 0.1
    """ The beep interval """
    BUZZER_BEEP_INTERVAL = 0.05

    def __init__(self, lcd_provider, gpio_led_green=GPIO_LED_GREEN, gpio_led_red=GPIO_LED_RED, gpio_buzzer=GPIO_BUZZER, debug

    def isDebug(self): ...

    def initialize(self): ...

    def cleanup(self): ...

    def turnOn(self, pinNumber): ...

    def turnOff(self, pinNumber): ...

    def lightGreen(self, timeout=1): ...

    def lightRed(self, timeout=1): ...

    def playSound(self): ...

    def playAfirmativeSound(self): ...

    def playNegativeSound(self): ...

    def playAfirmative(self, message=Translations.TICKET_VALIDATION_SUCCESS): ...

    def playNegative(self, message=Translations.TICKET_VALIDATION_FAILED): ... )
```

Figura 32 - Classe AccessProvider presente no AccessModule.py

O módulo do LCD está responsável pela comunicação entre o hardware do LCD e o software da máquina. Neste módulo pode-se registar uma mensagem a ser mostrada ao utilizador e esta fica no visor durante um dado período de tempo que é definido pelo programador. Esta classe pode ser vista na Figura 33.

```
from classes.LCDModule.Adafruit_LCDModule import Adafruit_CharLCD

from threading import Thread
from time import sleep

class LCDProvider(Thread):
    """ LCD Provider """

    def __init__(self, on_idle=None, thread_name='LCDProvider', message_format='[{0}] {1}', debug=False): ...

    def isDebug(self): ...

    def log(self, message): ...

    def terminate(self): ...

    def registerMessage(self, message): ...

    def run(self): ...

class LCDMessage:

    def __init__(self, messages, duration=2): ...

    def getMessages(self): ...

    def getDuration(self): ... n
```

Figura 33 - Classe LCDProvider presente no LCDModule.py

Todas as mensagens trocadas com as máquinas utilizam uma tag NDEF ou QR code do tipo URI Record com o seguinte formato:

ticket://<código empresa>/<código bilhete>

Isto permite que as máquinas facilmente façam a validação dos bilhetes, de uma forma rápida e prática. O código da empresa é utilizado como uma filtragem para reduzir o número de pedidos ao servidor. Caso este código não coincida com o da empresa a que esta máquina está vinculada esta recusa a sua operação sem ter de fazer qualquer pedido á API. Caso este bilhete faça parte da empresa em questão, então o pedido é enviado para o servidor da API que responde se o bilhete é válido e foi validado ou reporta o feedback á máquina que por sua vez é apresentado ao utilizador.

Figura 34 - Exemplo de código QR utilizado na comunicação

Após a montagem do hardware e do desenvolvimento do software, começou-se a criação de uma caixa para os equipamentos. Começou-se por cria uma representação da caixa em software de 3D.

Figura 35 - Modelo da caixa vista de frente

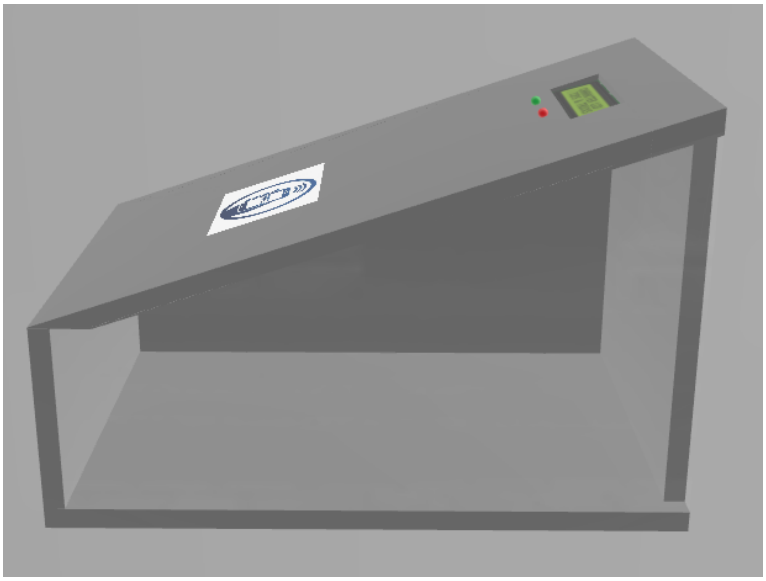


Figura 36 - Modelo da caixa vista de lado

Após chegar-se a um protótipo funcional, criou-se o plano das peças a serem fabricadas que pode ser visto na Figura 37.

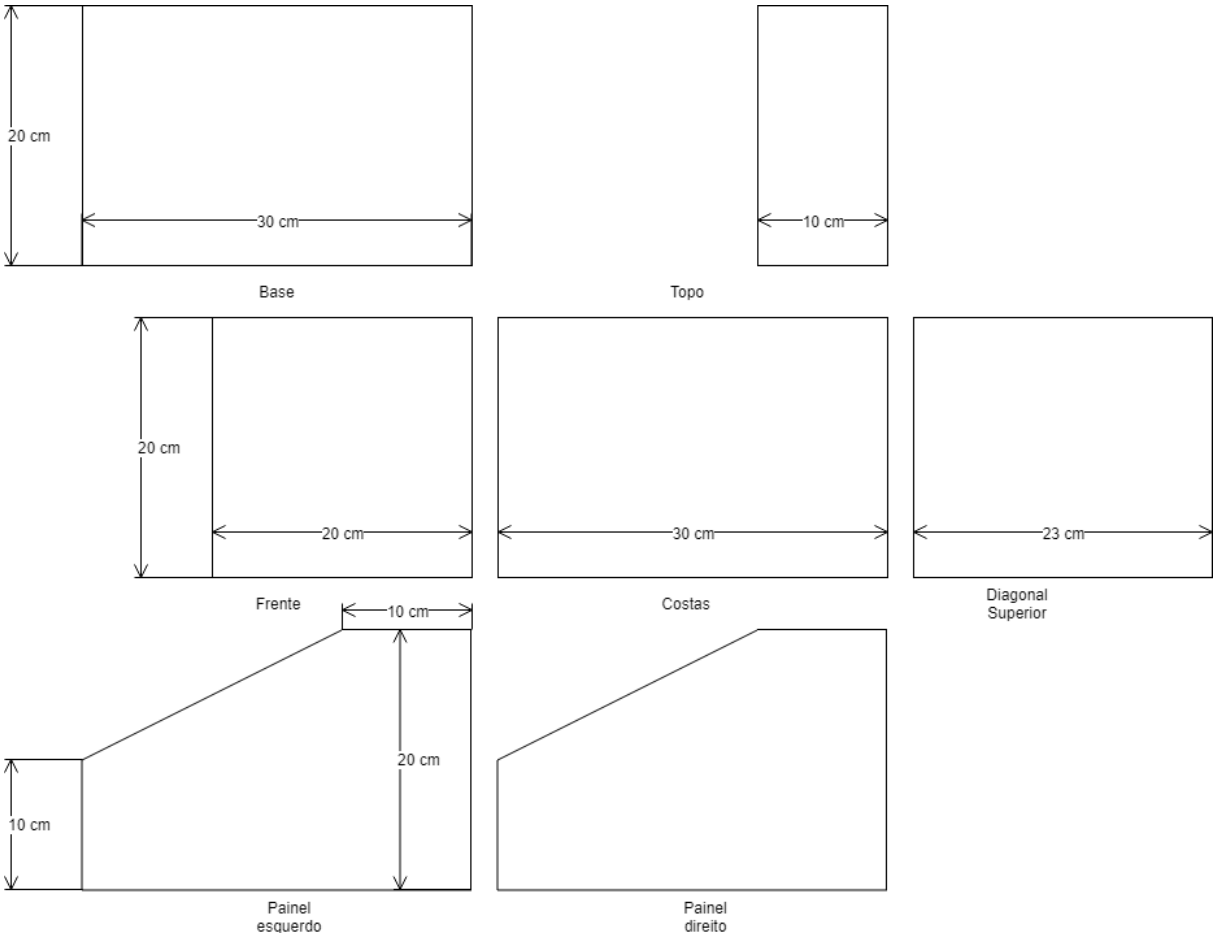


Figura 37 - Plano das peças a serem fabricadas

Definiu-se que os materiais a serem utilizados são, para a construção da caixa foi utilizado MDF e para os materiais transparentes, tal como a lateral direita seria o acrílico. Utilizando então os planos da Figura 37 cortou-se as madeiras nos formatos predefinidos e finalmente aplicou-se um pouco de lixa de modo a retirar pequenas imperfeiçoes que esta pode-se ter.

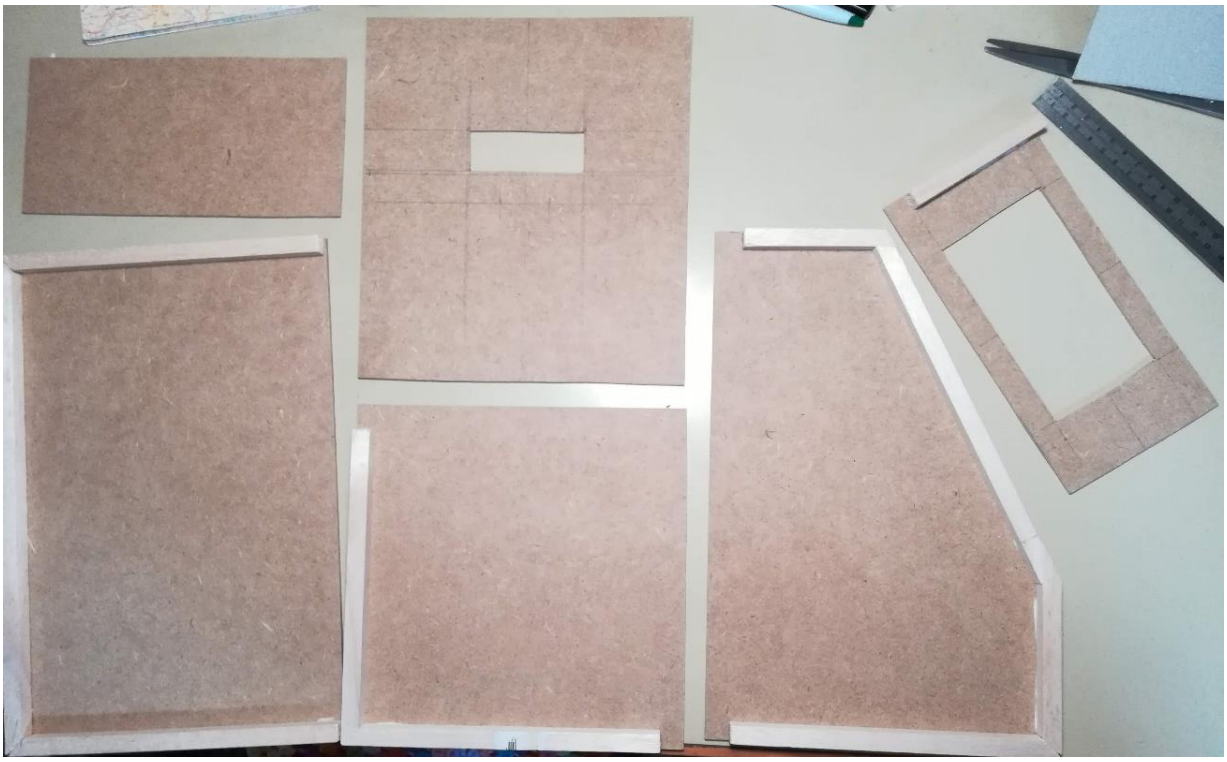


Figura 38 - Peças individualizadas da caixa

Após o recorte destas peças, começou-se a fase de montagem da caixa. Optou-se por não montar a caixa de uma vez, mas deixar a parte superior de fora para se poder testar se os dispositivos cabiam como planeado, ficando assim como mostra na Figura 39 e Figura 40.

Figura 39 - Caixa parcialmente montada

Figura 40 - Caixa parcialmente montada com os dispositivos no interior

Após a confirmação que estava tudo como planeado, então finalmente completou-se a estrutura da caixa ficando como mostra a Figura 41.

Figura 41 - Caixa estruturada

Após terminar-se a estruturação da caixa, passou-se á parte de decoração da mesma. Começou-se por aplicar massa nas arestas, de modo a suavizar os cantos para ficarem mais arredondados. De seguida lixou-se a caixa para retirar os excessos que possam ter ficado e forrou-se a mesma, ficando como vemos na Figura 42 e Figura 43.

Figura 42 - Caixa decorada vista de lado

Figura 43 - Caixa decorada vista de frente

Após a caixa estar finalizada só resta fazer a migração dos dispositivos para o seu interior. Para manter as breadboards no sítio foi utilizado fita cola de dupla face, de modo a que seja fácil de retirar caso haja necessidade.

Finalmente, depois de estar tudo preparado, só foi preciso testar os módulos para certificar que continuam a funcionar após a migração.

8.4 Aplicação Android

Começou-se por criar um diagrama UML que representa as várias funcionalidades da aplicação, como registar utilizador, iniciar sessão, comprar bilhetes e listar bilhetes já adquiridos. Este diagrama pode ser visto na Figura 44.

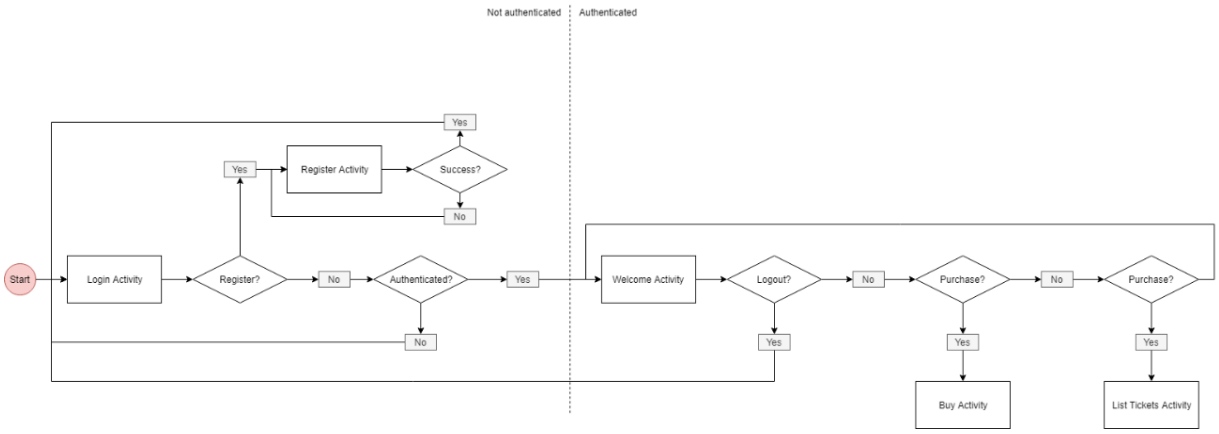


Figura 44 - Diagrama UML da aplicação Android

Após a criação desse diagrama definiu-se qual é a estrutura que será utilizada para essa funcionalidade na Figura 45.

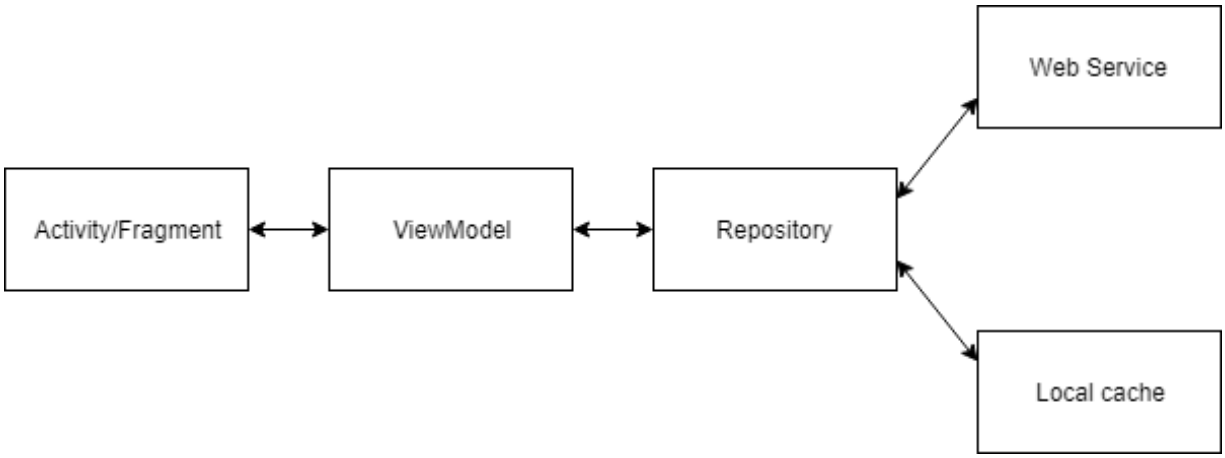


Figura 45 - Arquiteturas da aplicação

Durante o desenvolvimento desta aplicação foram utilizadas como fonte de consulta de informação os endereços presentes na web grafia II e III.A Activity/Fragment representam as Views que estão responsáveis por apresentar os dados ao utilizador. O View Model é o intermediário entre as Views o Repository, este tem a responsabilidade de trabalhar e armazenar temporariamente os dados entre Views, pois quando o estado de um dispositivo Android muda, por exemplo, ao rodar o ecrã, é gerado uma nova View. Para evitar a perda de informação que estava a ser mostrada na View anterior utiliza-se a View Model para transmitir esses dados entre os vários estados. O Web Service é o módulo que está responsável por ir buscar os dados

ao servidor web. A local cache é um sistema que faz cache dos dados de modo a que eles possam ser reutilizados sem ter de repetir pedidos ao servidor. O Repository é a entidade que está responsável por fazer uma gestão dos dados, esta gestão consiste em ir buscar os dados ao Web Service caso estes não existam em cache ou precisem de ser atualizados. Caso estes valores já existam em cache e sejam atualizados, então o Repository faz o pedido a cache por essa informação.

Após esta estrutura feita, começou-se a criar o Repository. Tendo em conta que esta vai ser um dos módulos principais da APP esta vai esta pode ser utilizada em várias Threads em simultâneo, que pode gerar problemas de concorrência á informação. Para evitar este problema, utilizou-se o código que está na Figura 46 para evitar esses problemas.

```
48      /**
49       * Holder for the lock instance.
50       */
51      @NonNull
52      private static final Object LOCK = new Object();
53      /**
54       * Holder for the ticket repository.
55       */
56      @Nullable
57      private static volatile FTicketRepository sTicketRepository;
58
59      /**
60       * Get the instance of the {@link FTicketRepository}.
61       *
62       * @param context The context to get the repository.
63       *
64       * @return The repository instance.
65       */
66      @NonNull
67      public static FTicketRepository getInstance(@NonNull Context context) {
68
69          // Check if the repository is already created
70          if (sTicketRepository == null) {
71              synchronized (LOCK) {
72                  if (sTicketRepository == null) {
73
74                      AppExecutors executor = AppExecutors.getInstance(context);
75                      FTicketDatabase database = FTicketDatabase.getInstance(context);
76                      FTicketAPIProvider apiProvider = FTicketAPIProvider.getInstance(context);
77
78                      sTicketRepository = new FTicketRepository(executor, database, apiProvider);
79                  }
80              }
81          }
82
83          return sTicketRepository;
84      }
85
```

Figura 46 - Contornar os problemas de concorrência

Como podemos ver a variável sTicketRepository é criada através de um sincronismo para evitar a concorrência a variável.

Para a base de dados utilizada como cache no Repository fui utilizado a biblioteca Room que gera uma base de dados SQL a partir de classes Java. Nesta base de dados o esquema foi

baseado na do servidor de API, mas num formato mais simplificado. Esta apenas utiliza 4 entidades, sendo elas a Company, que armazena as informações acerca das companhias existentes, os PackageType que representam os vários tipos de pacotes (se é entretenimento, se é transportes, etc.), a Package que presentam os vários pacotes que os operadores fornecem, e finalmente os Ticket que representam os bilhetes que estão registados em nome do utilizador atual. Na Figura 47 podemos ver este esquema em formato UML.

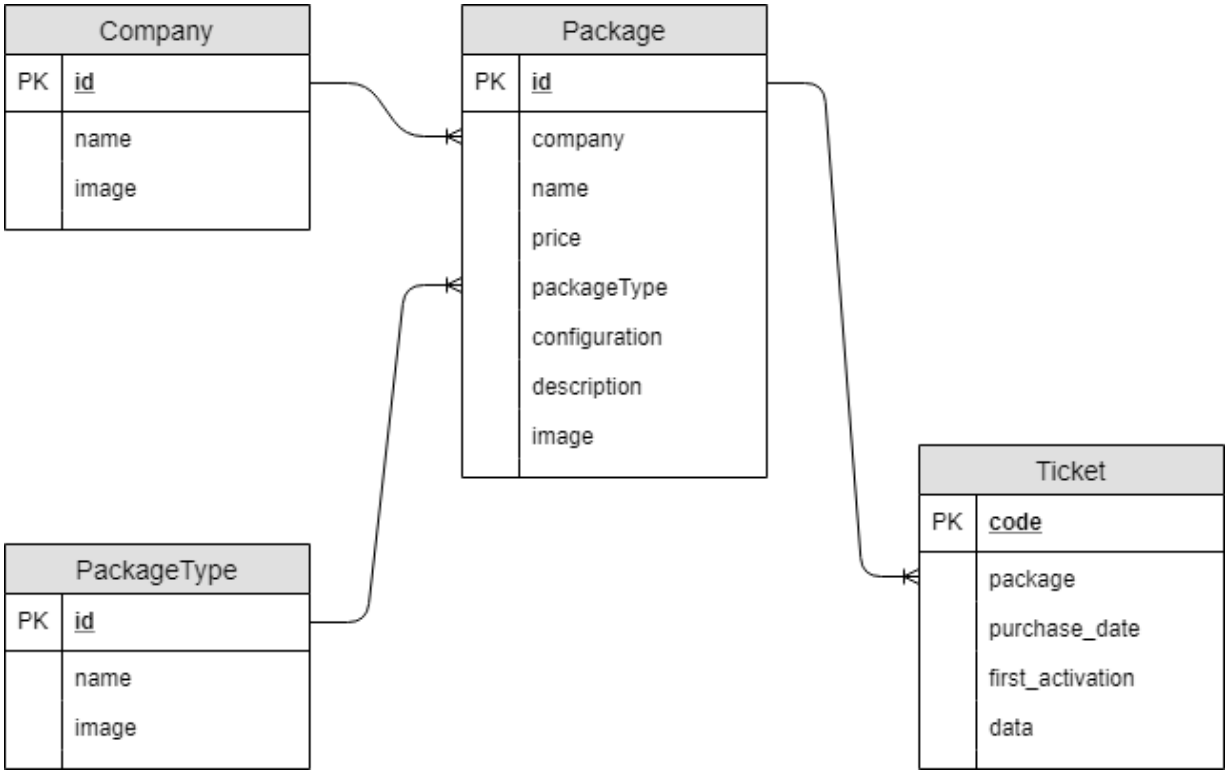


Figura 47 - Esquema da base de dados utilizado na aplicação Android

Em termos de organização do código, todas as classes foram devidamente comentadas, foram utilizados nomes para as variáveis e para as classes que representam as suas funcionalidades de modo a facilitar a sua interpretação. Exemplos podem ser vistos na Figura 43, Figura 44, Figura 45 e Figura 46, tal como no código fonte do projeto.

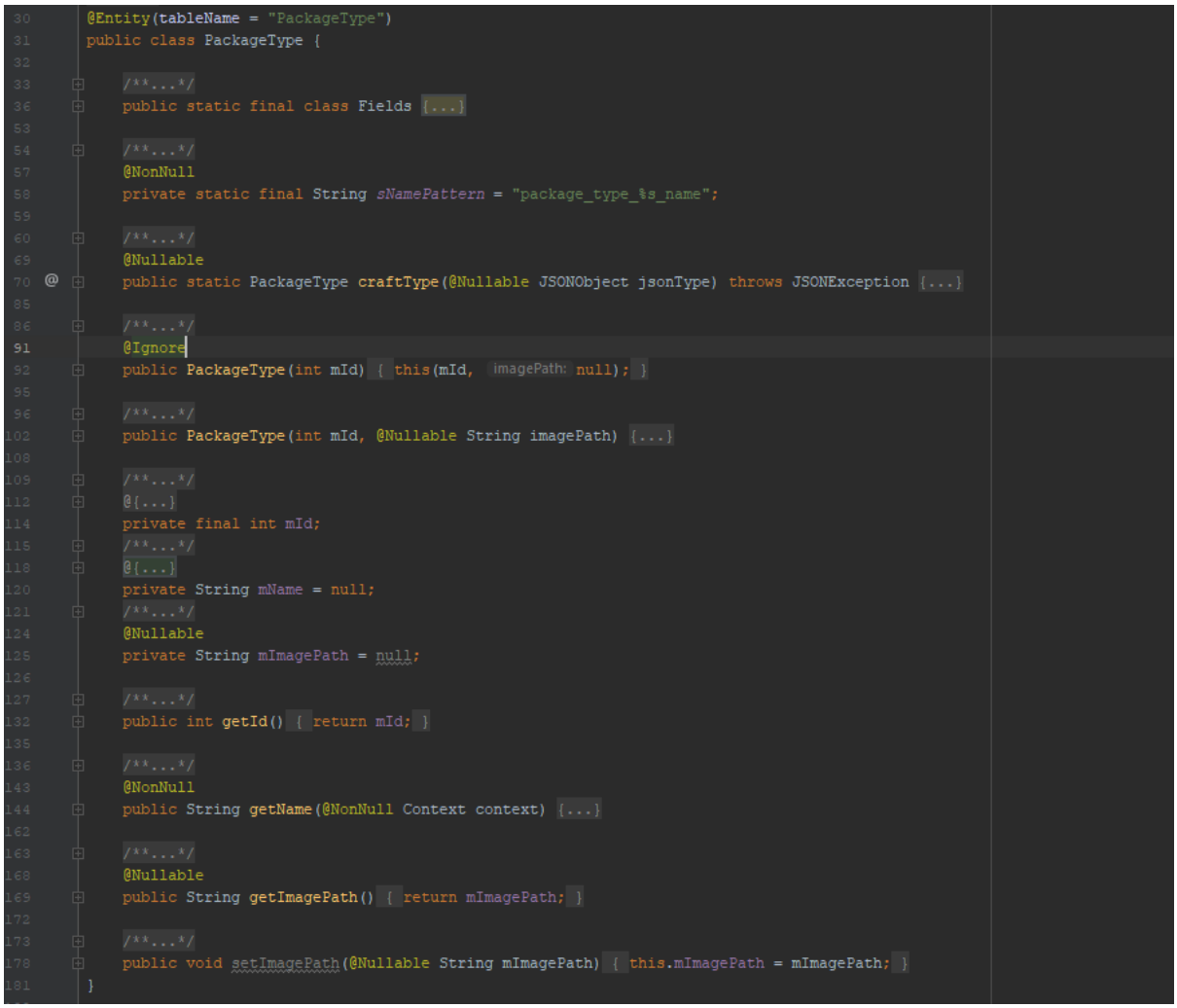
Para cada um dos modelos foi criado um Data Access Object (DAO) para aceder á informação desse modelo. Em cada um dos Data Access Objects foi implementado as funções necessárias para os dados modelos. No exemplo da Figura 48 esta função vai obter um pacote com um dado id. Para se definir qual a query a ser utilizada pelo SGBD para obtenção dos dados, utiliza-se a anotação *Query* com o placeholder :<nome> que vai ser substituído pelo argumento com o dado nome. Neste exemplo, o argumente é o code que representa o Código do pacote a consultar.

```java
/**
 * Get the {@link Package} by it's unique identifier.
 *
 * @return The {@link Package} with the given id, or {@code null} if not found.
 */
@NonNull
@Query("SELECT * FROM Package p WHERE p.id = :code LIMIT 1")
public LiveData<Package> getPackage(int code);
```

Figura 48 - Função DAO para obter uma Package com um dado id

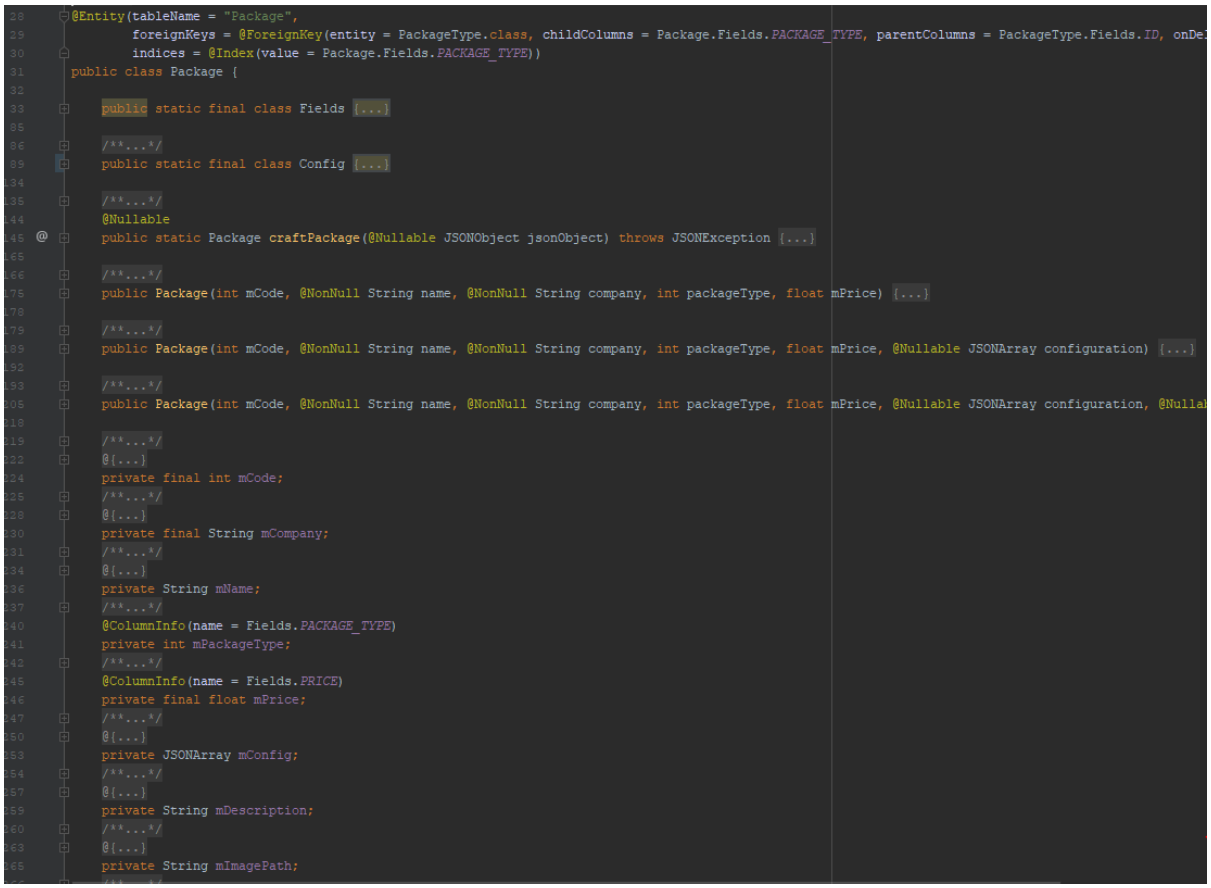
```java
@Entity(tableName = "Companies")
public class Company {

    public class Fields {...}

    /**...*/
    @Nullable
    public static Company craftCompany(@Nullable JSONObject companyData) throws JSONException {...}

    /**...*/
    public Company(@NonNull String mId, @NonNull String name) { this(mId, name, image: null); }

    /**...*/
    public Company(@NonNull String mId, @NonNull String name, @Nullable String image) {...}

    /**...*/
    @PrimaryKey
    @ColumnInfo(name = Fields.ID)
    @NonNull
    private final String mId;
    /**...*/
    @ColumnInfo(name = Fields.NAME)
    @Nullable
    private final String mName;
    /**...*/
    @ColumnInfo(name = Fields.IMAGE)
    @Nullable
    private String mImagePath;

    /**...*/
    @NonNull
    public String getId() { return mId; }

    /**...*/
    @NonNull
    public String getName() { return mName; }


    /**...*/
    public void setImagePath(@Nullable String mImagePath) { this.mImagePath = mImagePath; }

    /**...*/
    @Nullable
    public String getImagePath() { return mImagePath; }

}
```

Figura 49 - Classe do modelo Company



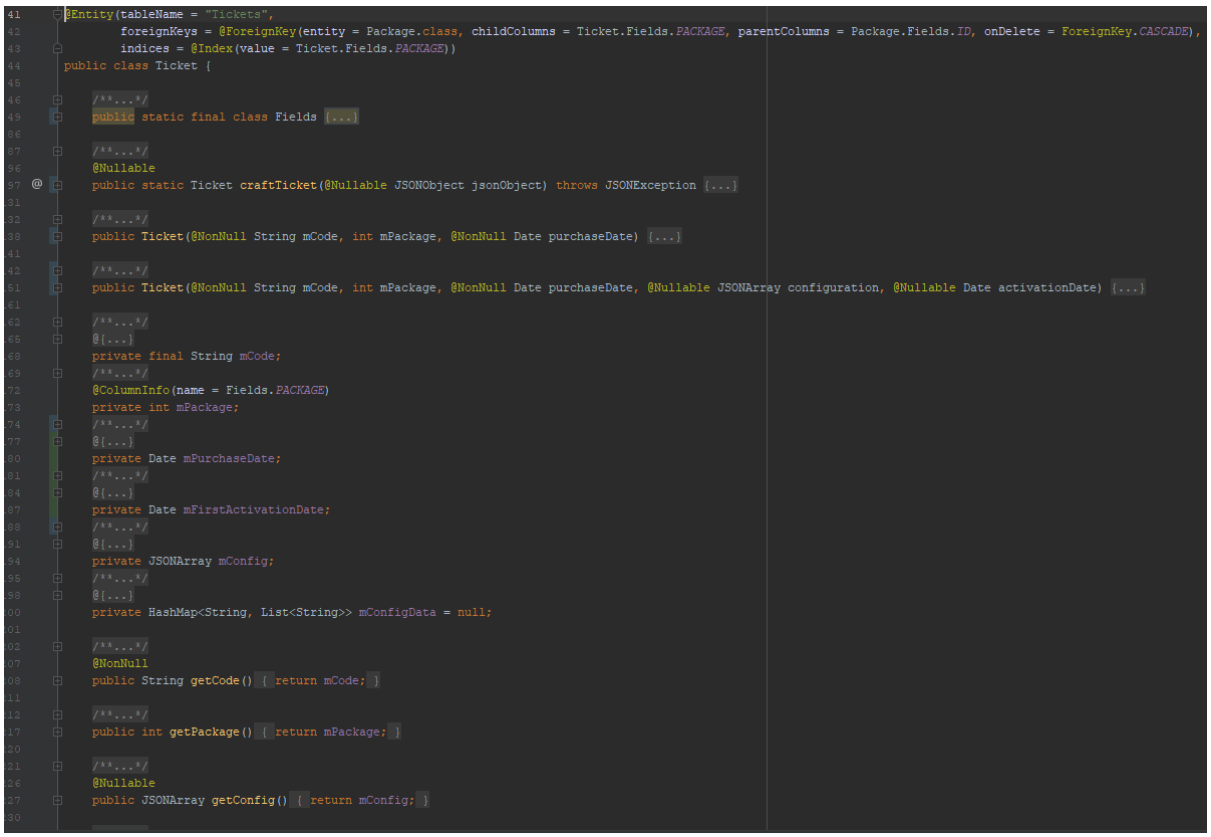
```
@Entity(tableName = "PackageType")
public class PackageType {

    /**...*/
    public static final class Fields {...}

    /**...*/
    @NonNull
    private static final String sNamePattern = "package_type_%s_name";

    /**...*/
    @Nullable
    public static PackageType craftType(@Nullable JSONObject jsonType) throws JSONException {...}

    /**...*/
    @Ignore
    public PackageType(int mId) { this(mId,  imagePath: null); }

    /**...*/
    public PackageType(int mId, @Nullable String imagePath) {...}

    /**...*/
    @{...}
    private final int mId;
    /**...*/
    @{...}
    private String mName = null;
    /**...*/
    @Nullable
    private String mImagePath = null;

    /**...*/
    public int getId() { return mId; }

    /**...*/
    @NonNull
    public String getName(@NonNull Context context) {...}

    /**...*/
    @Nullable
    public String getImagePath() { return mImagePath; }

    /**...*/
    public void setImagePath(@Nullable String mImagePath) { this.mImagePath = mImagePath; }
}
```

Figura 50 - Classe to modelo PackageType



```
@Entity(tableName = "Package",
        foreignKeys = @ForeignKey(entity = PackageType.class, childColumns = Package.Fields.PACKAGE_TYPE, parentColumns = PackageType.Fields.ID, onDe
        indices = @Index(value = Package.Fields.PACKAGE_TYPE))
public class Package {

    public static final class Fields {...}

    /**...*/
    public static final class Config {...}

    /**...*/
    @Nullable
    public static Package craftPackage(@Nullable JSONObject jsonObject) throws JSONException {...}

    /**...*/
    public Package(int mCode, @NonNull String name, @NonNull String company, int packageType, float mPrice) {...}

    /**...*/
    public Package(int mCode, @NonNull String name, @NonNull String company, int packageType, float mPrice, @Nullable JSONArray configuration) {...}

    /**...*/
    public Package(int mCode, @NonNull String name, @NonNull String company, int packageType, float mPrice, @Nullable JSONArray configuration, @Nullab

    /**...*/
    @{...}
    private final int mCode;
    /**...*/
    @{...}
    private final String mCompany;
    /**...*/
    @{...}
    private String mName;
    /**...*/
    @ColumnInfo(name = Fields.PACKAGE_TYPE)
    private int mPackageType;
    /**...*/
    @ColumnInfo(name = Fields.PRICE)
    private final float mPrice;
    /**...*/
    @{...}
    private JSONArray mConfig;
    /**...*/
    @{...}
    private String mDescription;
    /**...*/
    @{...}
    private String mImagePath;
```

Figura 51 - Classe do modelo Package



```
@Entity(tableName = "Tickets",
        foreignKeys = @ForeignKey(entity = Package.class, childColumns = Ticket.Fields.PACKAGE, parentColumns = Package.Fields.ID, onDelete = ForeignKey.CASCADE),
        indices = @Index(value = Ticket.Fields.PACKAGE))
public class Ticket {

    /**...*/
    public static final class Fields {...}

    /**...*/
    @Nullable
    public static Ticket craftTicket(@Nullable JSONObject jsonObject) throws JSONException {...}

    /**...*/
    public Ticket(@NonNull String mCode, int mPackage, @NonNull Date purchaseDate) {...}

    /**...*/
    public Ticket(@NonNull String mCode, int mPackage, @NonNull Date purchaseDate, @Nullable JSONArray configuration, @Nullable Date activationDate) {...}

    /**...*/
    @{...}
    private final String mCode;
    /**...*/
    @ColumnInfo(name = Fields.PACKAGE)
    private int mPackage;
    /**...*/
    @{...}
    private Date mPurchaseDate;
    /**...*/
    @{...}
    private Date mFirstActivationDate;
    /**...*/
    @{...}
    private JSONArray mConfig;
    /**...*/
    @{...}
    private HashMap<String, List<String>> mConfigData = null;

    /**...*/
    @NonNull
    public String getCode() { return mCode; }

    /**...*/
    public int getPackage() { return mPackage; }

    /**...*/
    @Nullable
    public JSONArray getConfig() { return mConfig; }
```

Figura 52 - Classe do modelo Ticket

Estas base de dados é gerida por uma classe denominada FTicketDatabase. Esta contém todas a funções necessárias para aceder ao DAO de cada modelo.

```java
/**
 * Get the access module for the {@link PackageType} table.
 *
 * @return The {@link pt.forever.school.fticket.api.database.access.PackageTypeAccess} instance.
 */
@NonNull
public abstract PackageTypeAccess packageTypeAccess();

/**
 * Get the access module for the {@link pt.forever.school.fticket.api.database.models.Company} table.
 *
 * @return The {@link pt.forever.school.fticket.api.database.access.CompanyAccess} instance.
 */
@NonNull
public abstract CompanyAccess companyAccess();

/**
 * Get the access module for the {@link pt.forever.school.fticket.api.database.models.Package} table.
 *
 * @return The {@link pt.forever.school.fticket.api.database.access.PackageAccess} instance.
 */
@NonNull
public abstract PackageAccess packageAccess();

/**
 * Get the access module for the {@link pt.forever.school.fticket.api.database.models.Ticket} table.
 *
 * @return The {@link pt.forever.school.fticket.api.database.access.TicketAccess} instance.
 */
@NonNull
public abstract TicketAccess ticketAccess();
```

Figura 53 - Métodos para aceder ao DAO de cada modelo

Para obter-se a informação através da internet, criou-se a classe FTicketAPIProvider que obtém os dados diretamente do servidor.

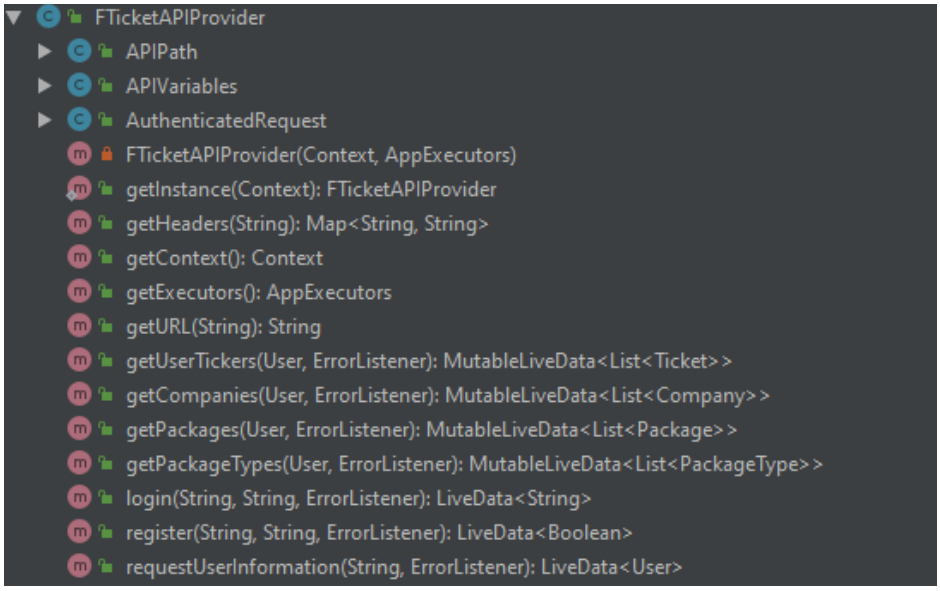


Figura 54 - Funções da classe FTicketAPIProvider

E como interface de comunicação com o resto da aplicação e tem a função de verificar se é necessária atualização dos dados que tem em cache, temos o módulo FTicketRepository.

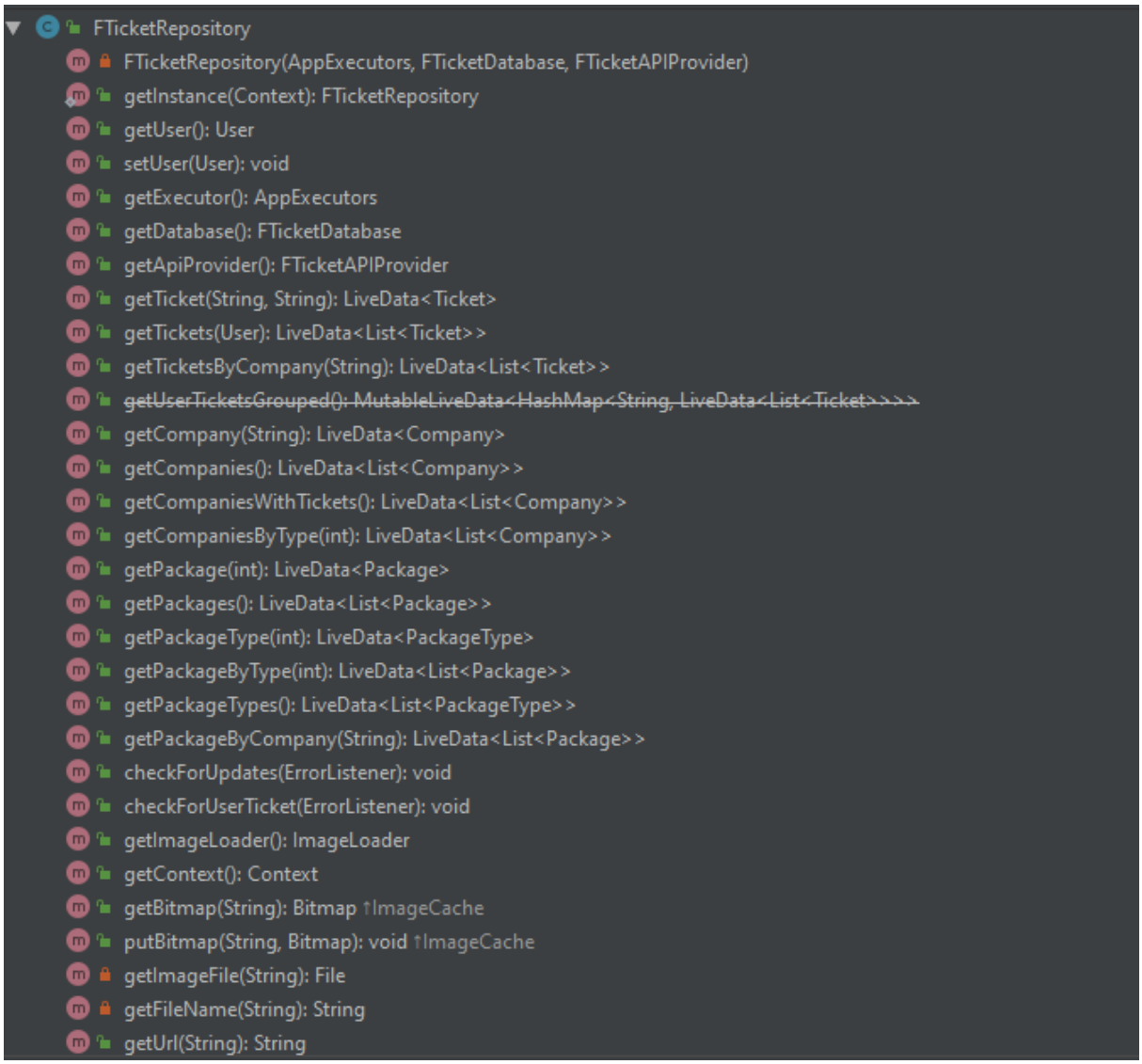


Figura 55 - Funções presentes na classe FTicketRepository

Do lado do utilizador, utilizou-se um visual simples, mas acolhedor. Quando um utilizado está prestes a iniciar sessão, este vê uma página idêntica á da Figura 56. Caso os dados introduzidos, estejam inválidos, a mensagem de erro será mostrada nesse campo como mostra a Figura 57.

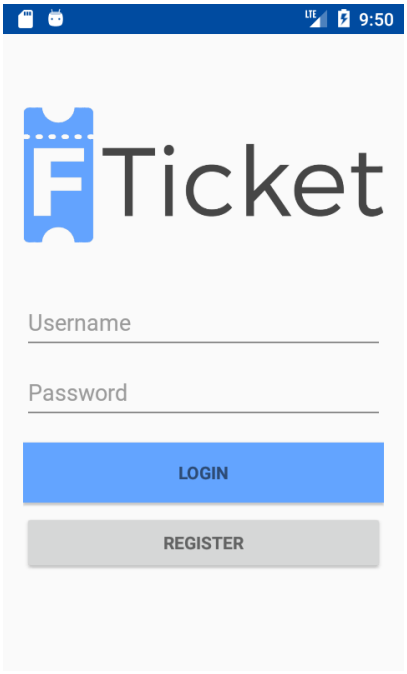


Figura 56 - Ecrã de início de sessão

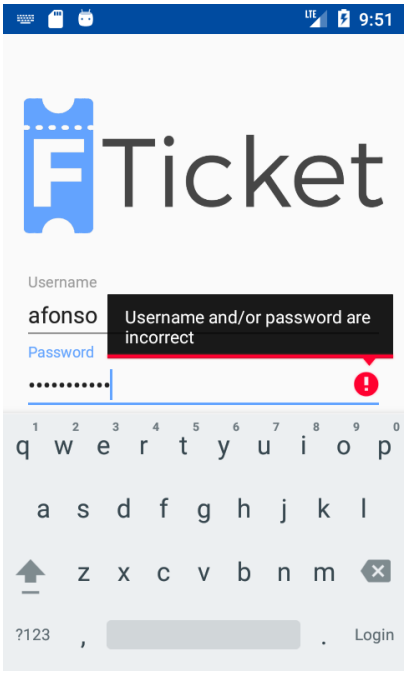


Figura 57 - Ecrã de início de sessão com nome de utilizador ou senha inválida

Caso o utilizado ainda não tenha uma conta criada, este pode-se registar sendo levado para uma página como mostra a Figura 58.

Figura 58 - Ecrã de registar utilizador

Ao iniciar sessão é mostrado a página principal, onde é dado as boas vindas ao utilizado e este pode escolher entre as várias opções do sistema, tal como vemos na Figura 59.

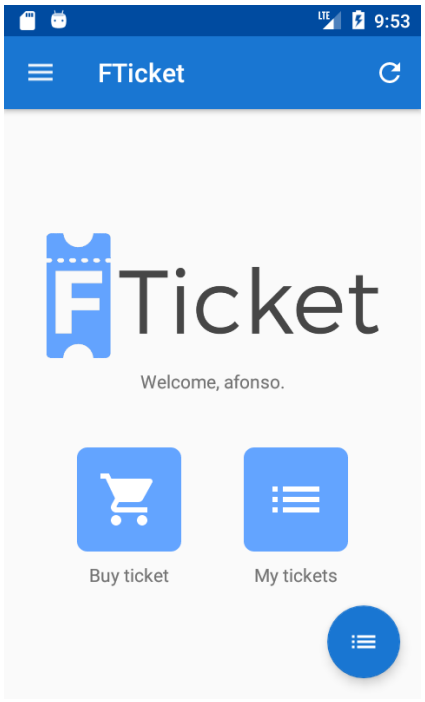


Figura 59 - Página principal da aplicação

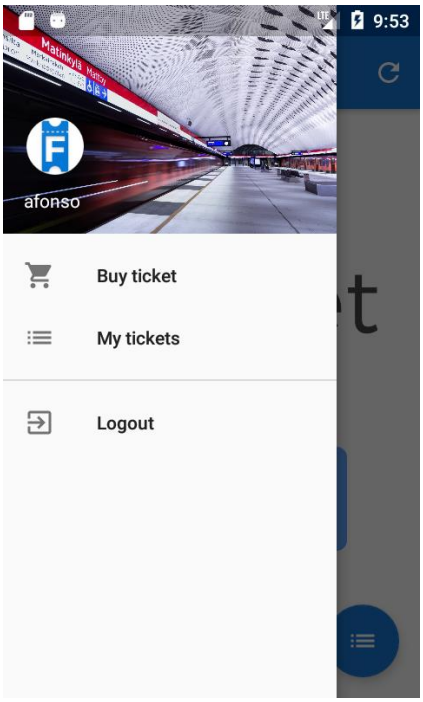


Figura 60 - Painel de navegação da aplicação

Na Figura 60 podemos ver o painel de navegação, que permite navegar entres os vários pontos da aplicação, e o botão para terminar sessão, caso o utilizador o pretenda.

Caso o utilizador escolha a opção de comprar bilhete, ser-lhe-á mostrado uma lista de tipos de bilhetes que existem (Figura 61), sendo um exemplo de alguns deles, os Transportes Terrestre, os Museus, o Cinema, o Teatro, os Transportes Aéreos, os Transportes Marítimos, entre outros. Ao escolher um tipo de transporte, a aplicação mostra todas as empresas que têm serviços desse tipo (Figura 62).

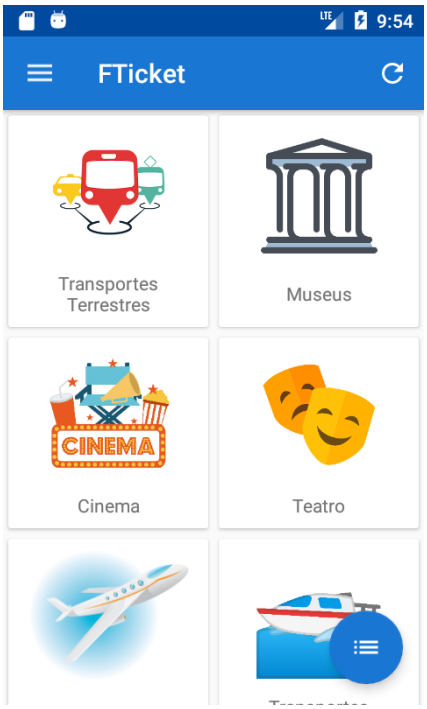


Figura 61 - Listagem de tipos de bilhetes

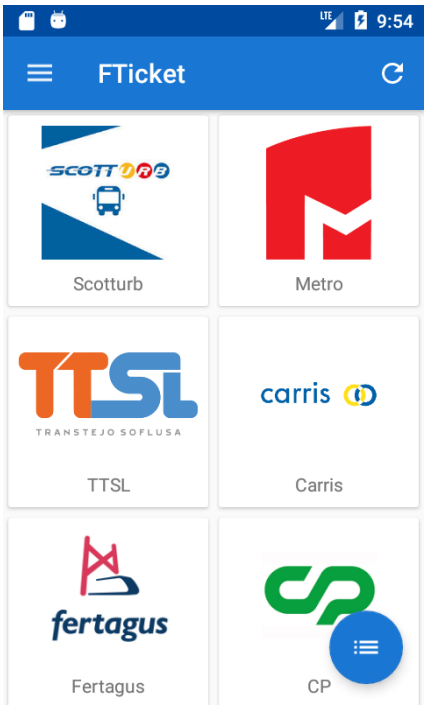


Figura 62 - Listagem de empresas do tipo T. Terrestre

Após escolhida uma empresa, são listados todos os serviços desta (Figura 63). O utilizador pode consultar informações detalhadas sobre este serviço clicando sobre ele, sendo reencaminhado para página de detalhes (Figura 64).

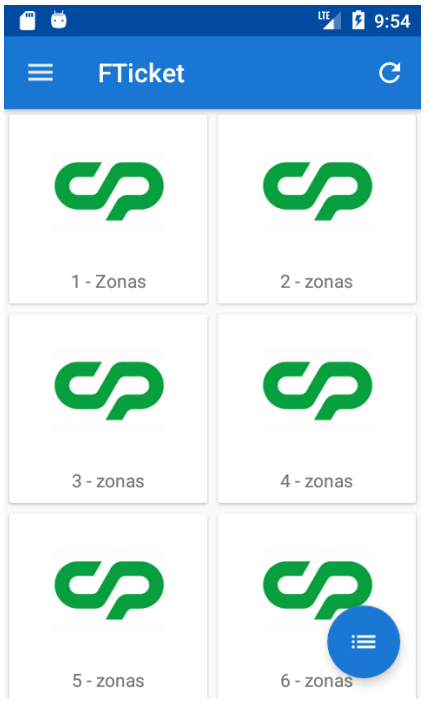


Figura 63 - Listagem de tipos dos serviços da CP

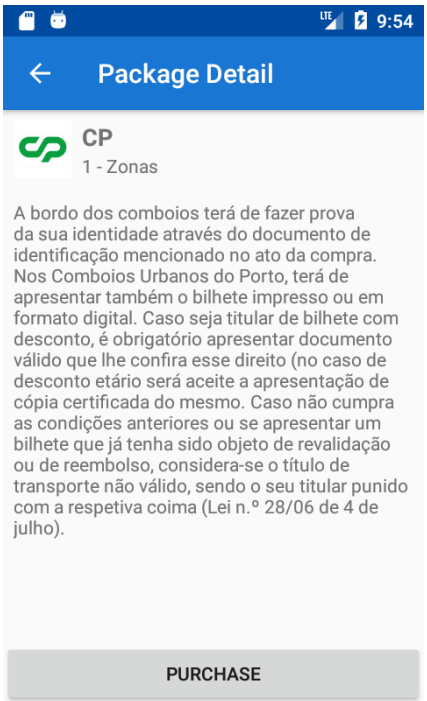


Figura 64 - Informação acerca do serviço 1 zona CP

Nesta página de detalhes o utilizador pode clicar sobre o botão de comprar, sendo reencaminhado para o PayPal para efetuar o seu pagamento (Figura 65).

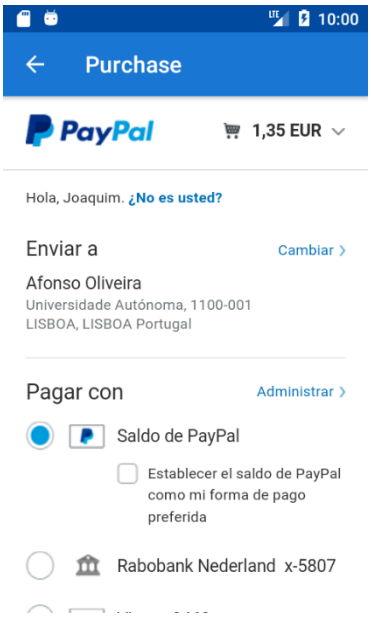


Figura 65 - Página de pagamento do pacote/serviço

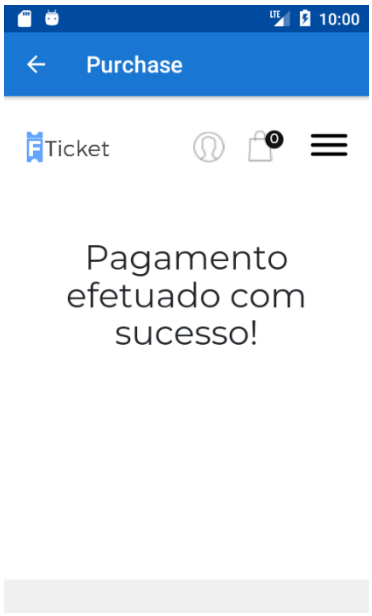


Figura 66 - Página de conclusão de pagamento

Após esse pagamento, o utilizador será reencaminhado para a página de pagamento efetuado com sucesso (Figura 66).

Após o utilizador ter adquirido o bilhete, este passa a estar disponível no espaço dos "meus bilhetes" (Figura 66).

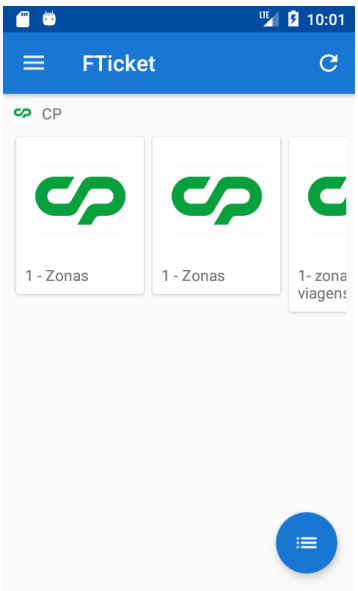


Figura 67 - Ecrã os meus bilhetes

9 Conclusão

Com este projeto podemos evitar muitos dos problemas referidos anteriormente, sobre as grandes afluências às bilheteiras, especialmente em épocas turísticas, implementando o sistema Fast Ticket que permite aos utilizadores comprarem os seus bilhetes diretamente do seu PC, smartphone ou tablet.

Este projeto desafiou-nos de várias maneiras, desde o amadurecimento dos nossos conhecimentos, pois requereu uma grande pesquisa da nossa parte, devido ao facto de um grande número dos módulos utilizados saírem dos conhecimentos dados em contexto académico, o que levou á aquisição de novos conhecimentos que complementam os adquiridos neste contexto, mas acima de tudo, um dos pontos importantes foi a nossa organização como grupo para cumprirmos o que foi estipulado, dentro dos prazos previstos como podemos ver na Figura 68.

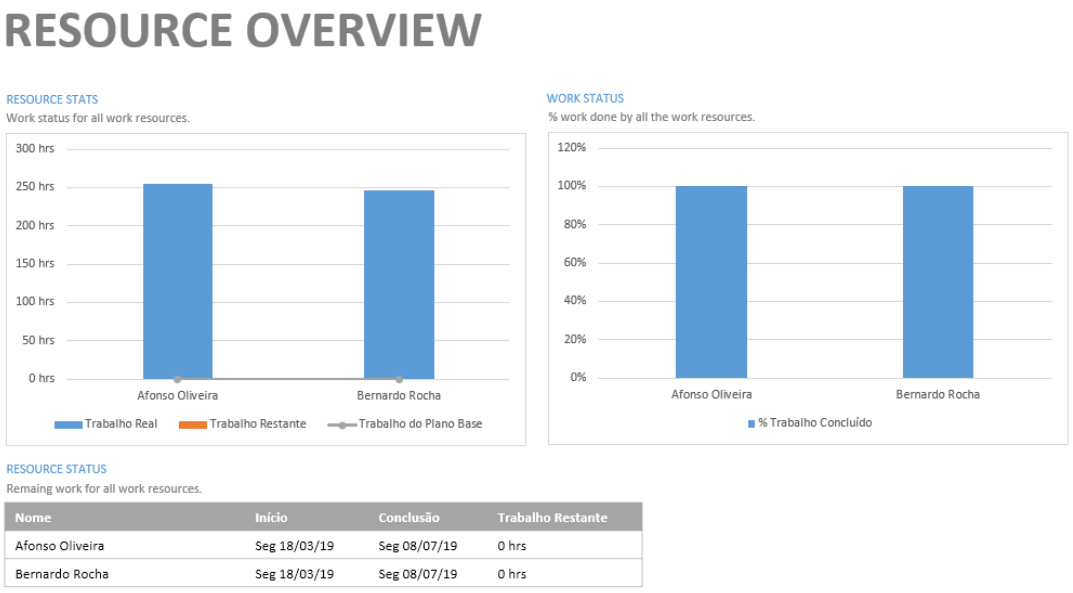


Figura 68 - Horas dos recursos completa

10 Trabalho futuro

Apesar deste projeto estar terminado surgiram diversas ideias que saiam muito do âmbito principal do nosso trabalho inicial, mas que são ideias válidas para desenvolver em trabalho futuro. A primeira ideia que surgiu foi podermos implementar um sistema que ao entrarmos num determinado transporte irá verificar o percurso que o utilizador está a fazer e apenas no fim cobra o valor da viagem realizada, ou seja, o utilizador não tinha de comprar nenhum bilhete em específico pois o sistema no fim da viagem irá verificar qual seria o bilhete mais justo a aplicar dada a viagem efetuada pelo utilizador. A segunda ideia seria podermos implementar a capacidade do sistema, através de um percurso dado pelo utilizador, calcular a rota mais rápida e com menor custo associado, apresentando de imediato ao utilizador todos os bilhetes que tem de adquirir. Todas estas ideias poderão ser implementadas no nosso projeto e com isso torná-lo mais completo e autónomo.

11 Web grafia

I. “PN532 User Manual, Rev. 02”, https://www.nxp.com/docs/en/user-guide/141520.pdf, (Acedido em 06/07/2019)

II. “Java 8 Documentation”, https://docs.oracle.com/javase/8/docs/api/, (Acedido em 06/07/2019)

III. “Android Documentation”, https://developer.android.com/docs, (Acedido em 06/07/2019)

IV. “Django Documentation”, https://docs.djangoproject.com/en/2.2/, (Acedido em 06/07/2019)

V. “Django Payments Documentation”, https://django-payments.readthedocs.io/en/latest/, (Acedido em 06/07/2019)

VI. “Django Rest Framework Documentation”, https://www.django-rest-framework.org/, (Acedido em 06/07/2019)

VII. “Django Rest Framework API Key Documentation”, https://github.com/manosim/django-rest-framework-api-key, (Acedido em 06/07/2019)

VIII. “PayPal Documentation”, https://developer.paypal.com/docs/, (Acedido em 06/07/2019)

IX. “Raspberry Pi Documentation”, https://www.raspberrypi.org/documentation/, (Acedido em 06/07/2019)

12 Anexo 01 – Manual de utilização - Android

12.1 Registar utilizador

1. Abrir a aplicação FTicket;
2. Clicar em registar;

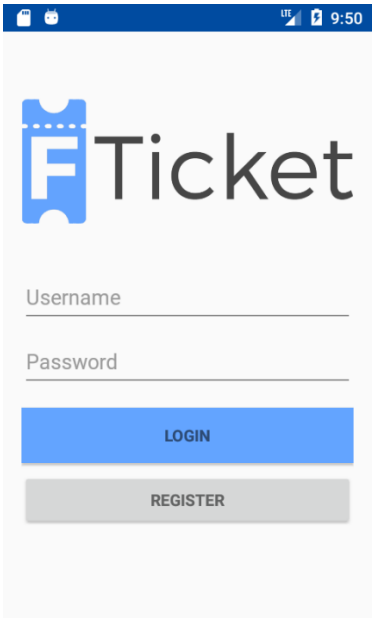


Figura 69 - Ecrã de Login da aplicação Android

3. Preencher o formulário, com os dados pedidos;

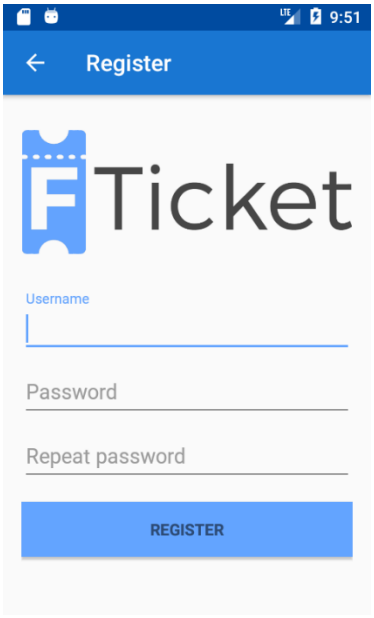


Figura 70 - Ecrã de registo do utilizador

4. Clicar em registar.

12.2 Iniciar sessão

1. Abrir a aplicação FTicket;
2. Preencher o formulário com o seu nome de utilizador;

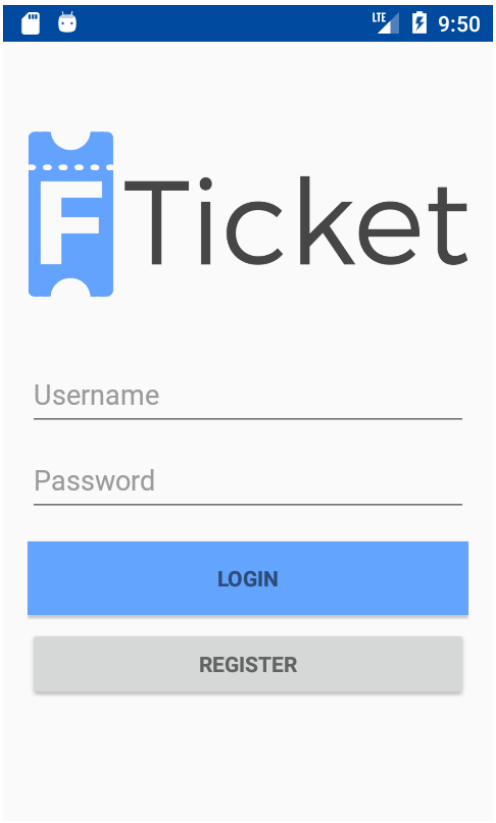


Figura 71 - Ecrã de Login da aplicação Android

3. Preencher o formulário com a sua password;
4. Clicar em Iniciar sessão.

12.3 Comprar bilhetes

1. Iniciar sessão na aplicação;
2. Clicar em comprar bilhete;

Figura 72 - Ecrã principal da aplicação Android

3. Escolher o tipo de bilhete que pretende;

X. Nota: Como exemplo foi utilizado os transportes terrestres.

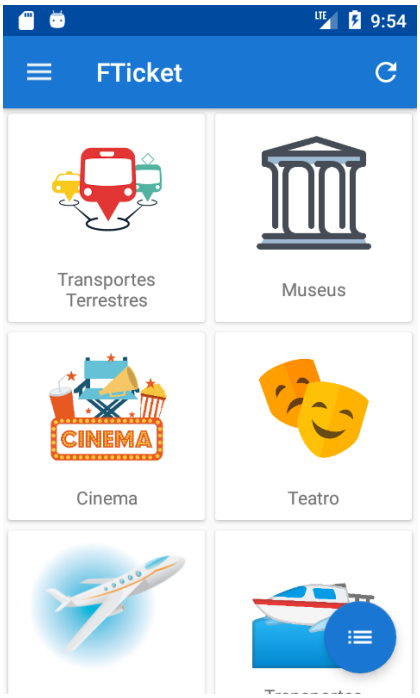


Figura 73 - Ecrã de comprar bilhete aplicação Android

4. Escolher a empresa na qual se pretende adquirir o bilhete;

   Nota: Como exemplo foi utilizado a empresa CP

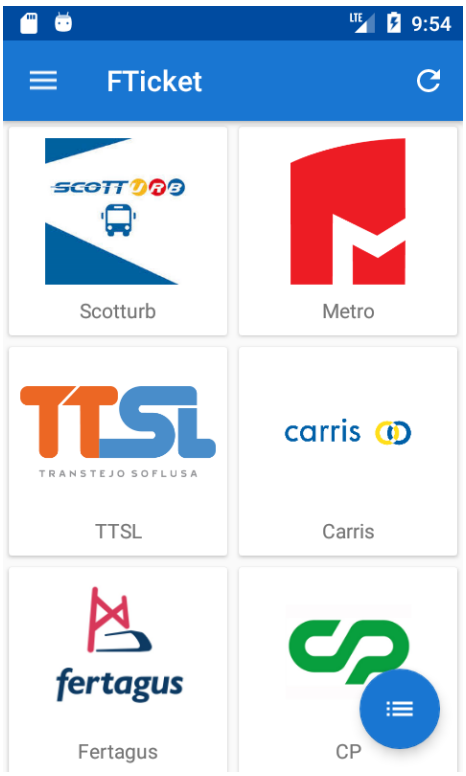


Figura 74 - Ecrã lista de empresas do tipo transporte terrestres

5. Escolher qual dos pacotes se pretende comprar;

   Nota: Como exemplo foi utilizado o pacote "1 – Zonas"

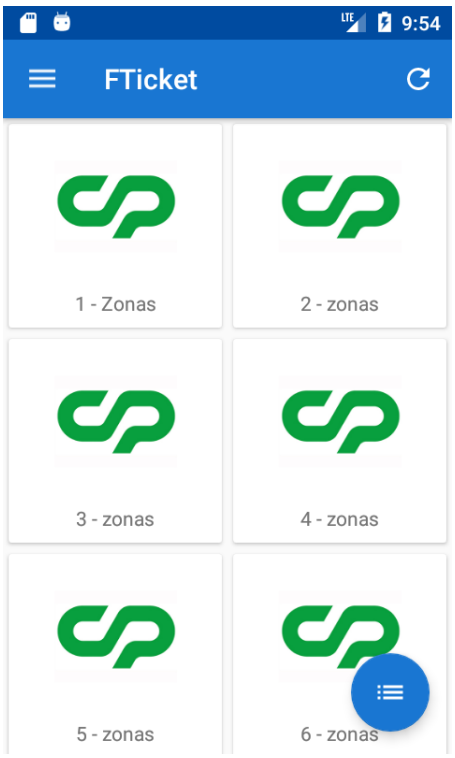


Figura 75 - Ecrã lista de pacotes da empresa CP

6. Clicar na opção comprar;

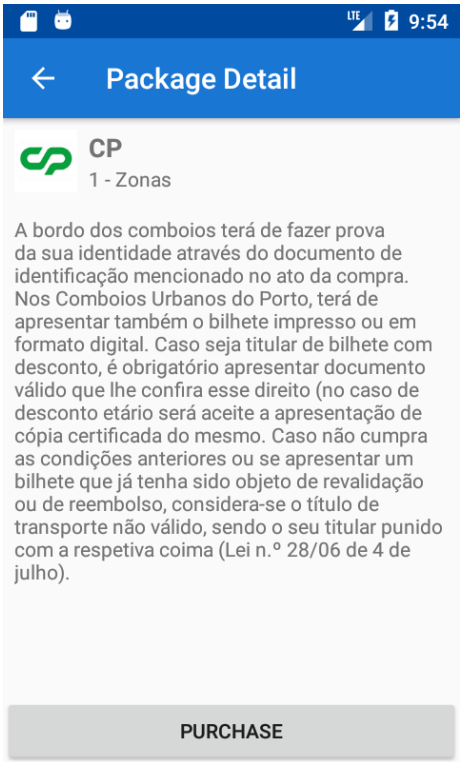


Figura 76 - Ecrã de validação na aplicação Android

7. Efetuar o pagamento;

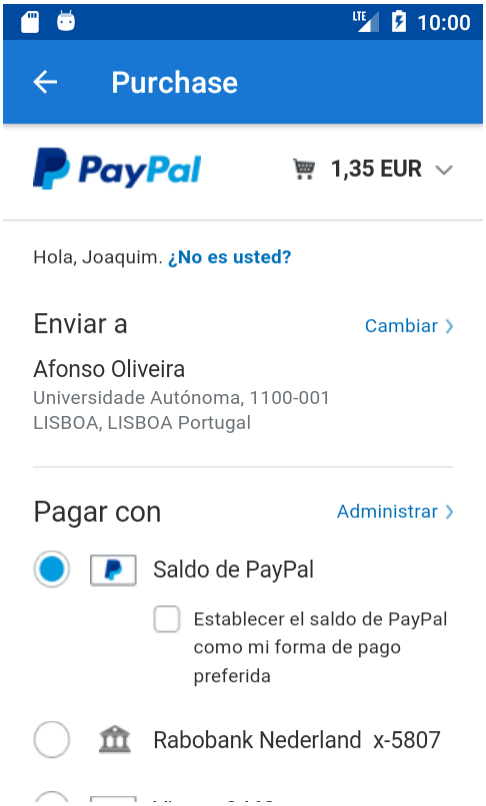


Figura 77 - Ecrã de pagamento PayPal

8. Fechar a janela, após o pagamento estar efetuado.

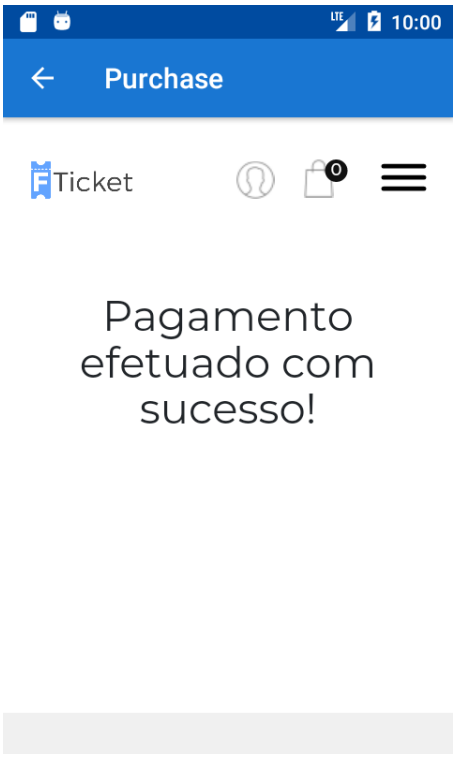


Figura 78 - Ecrã de pagamento completo

12.4 Utilizar um bilhete

1. Iniciar sessão na aplicação;
2. Clicar em meus bilhetes;

Figura 79 - Ecrã principal da aplicação Android

3. Escolher dos bilhetes disponíveis qual pretende;

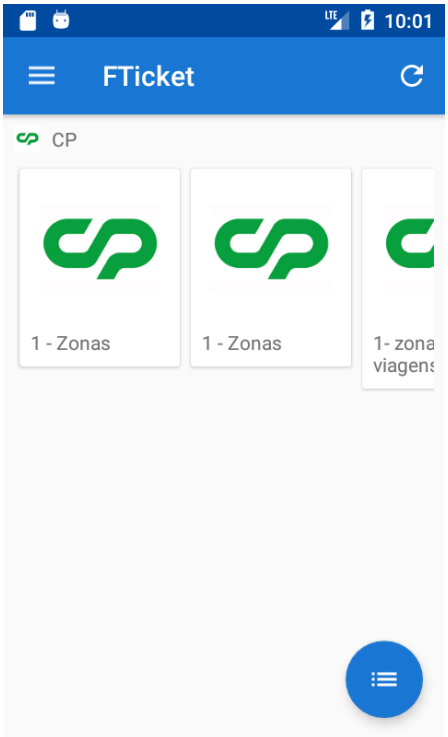


Figura 80 - Ecrã listar bilhetes

4. Escolher a opção validar;

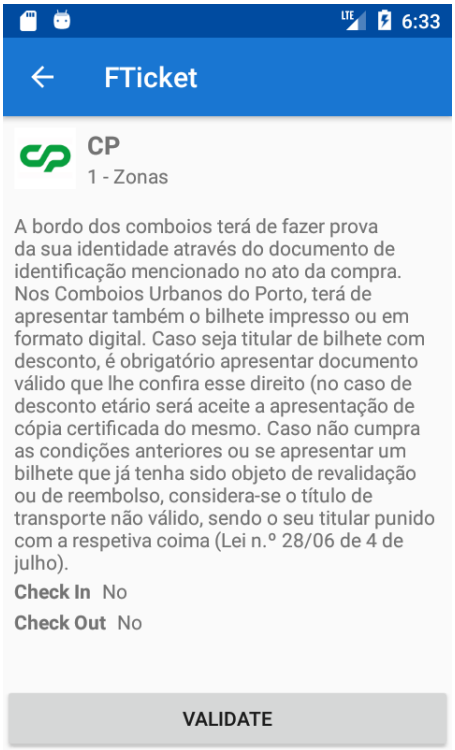


Figura 81 - Ecrã de informação sobre o bilhete

5. Escolher o método de validação e validar;

Figura 82 - Ecrã de validação de bilhete

13 Anexo 02 – Manual de utilização - Website

O cabeçalho do website contem 3 botões “pagina inicial” que nos leva até a página principal do site, “loja” que nos leva diretamente para a loja onde podemos comprar os bilhetes, “sobre” que nos reencaminha para a página da empresa onde está a missão e os objetivos da empresa.

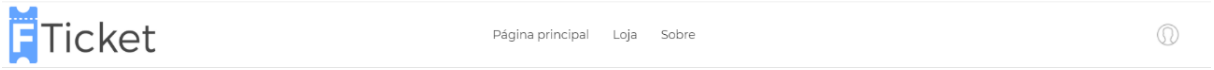


Figura 83 - Cabeçalho do website

Para além disto o cabeçalho ainda contém o painel de gestão de conta que se encontra á direita.

Figura 84 - Ícone de gestão de conta

Ao carregarmos neste ícone aparece um popup de login.

Figura 85 - Popup de login

Quando não existe sessão iniciada esta é o aspeto do popup de login que contém o botão de login e caso não esteja registado tem o botão de registo.

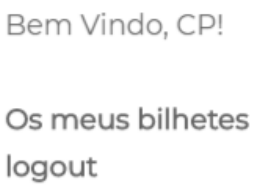


Figura 86 - Popup de login com sessão iniciada

Caso já tenha o login feito este é o aspeto do popup, contem um botão para ir diretamente aos bilhetes adquiridos pela pessoa que tem sessão iniciada e o botão para fazer o logout.

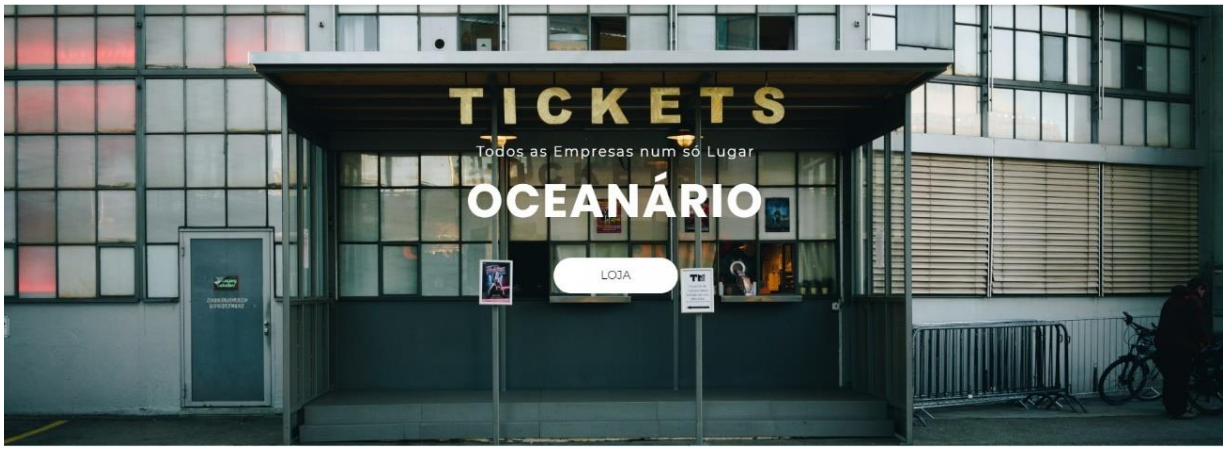


Figura 87 - Slide empresas

Ainda na página inicial temos um slide que vai passando todas as empresas que estão registadas e ao clicar no botão loja vai diretamente para a loja.

Figura 88 - Tipos de bilhetes registados

Um bocadinho mais abaixo temos exposto todas as categorias registadas no website ao clicar em qualquer uma vai diretamente para a loja.

Na pagina da loja contamos com uma lista de filtros que se encontra a esquerda para ser mais rápido chegar aos bilhetes pretendidos e a direita é onde vão ser apresentados tanto as categorias como as empresas e como os bilhetes que depois vão ser selecionados caso os utilizadores não queiram passar por todos estes passos basta selecionarem.

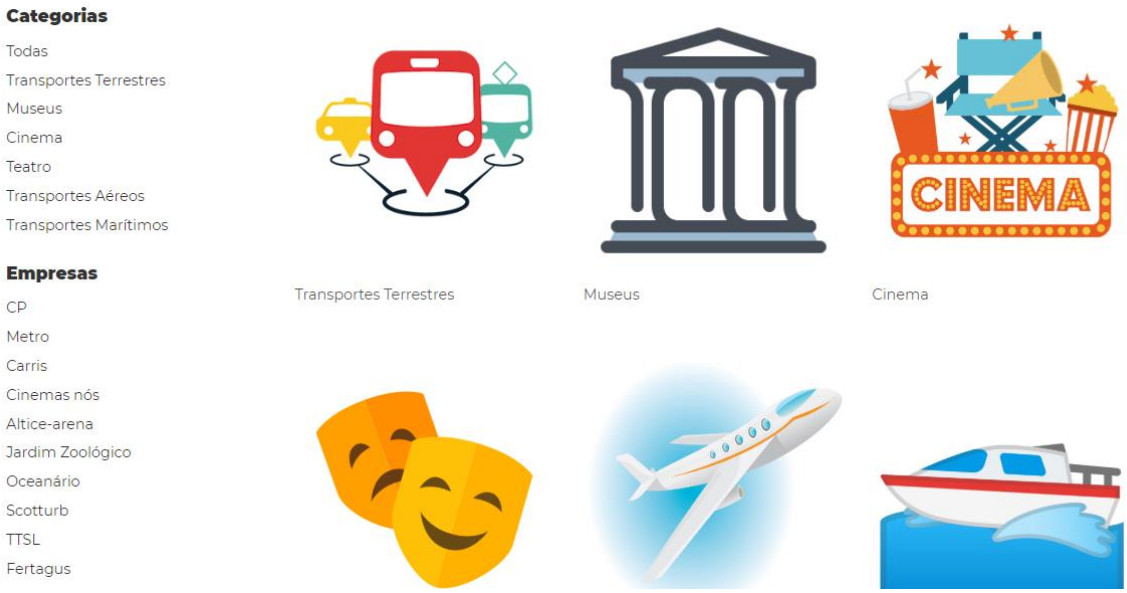


Figura 89 - Página da loja do website

E por último temos a página do detalhe do bilhete onde temos o nome, o preço, a descrição e o botão onde podemos comprar o bilhete apresentado. Caso cliquemos no botão comprar vai-nos abrir a página do *PayPal* que já contém todos os detalhes do produto que estamos a tentar comprar basta só entrar na conta do *PayPal* e pagar.

Figura 90 - Página do detalhe do produto

Caso o pagamento seja efetuado com sucesso aparece esta página que podemos ver na Figura 91.

Figura 91 - Pagamento efetuado com sucesso

.

14 Anexo 03 – Manual de instalação – Website

1. Arrancar com a máquina "**Projeto – WebServer**" e aguardar até esta pedir para iniciar sessão;

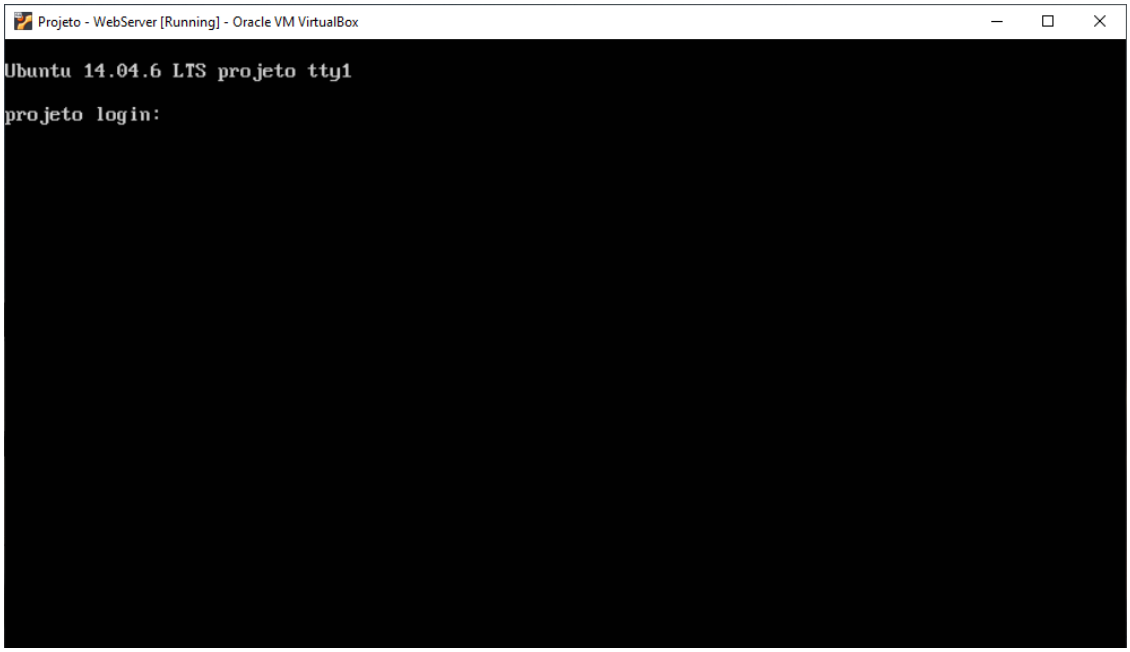


Figura 92 - Ubuntu ecrã de login

2. Iniciar sessão com o nome de utilizador "**projeto**" e senha "**qwerty**";

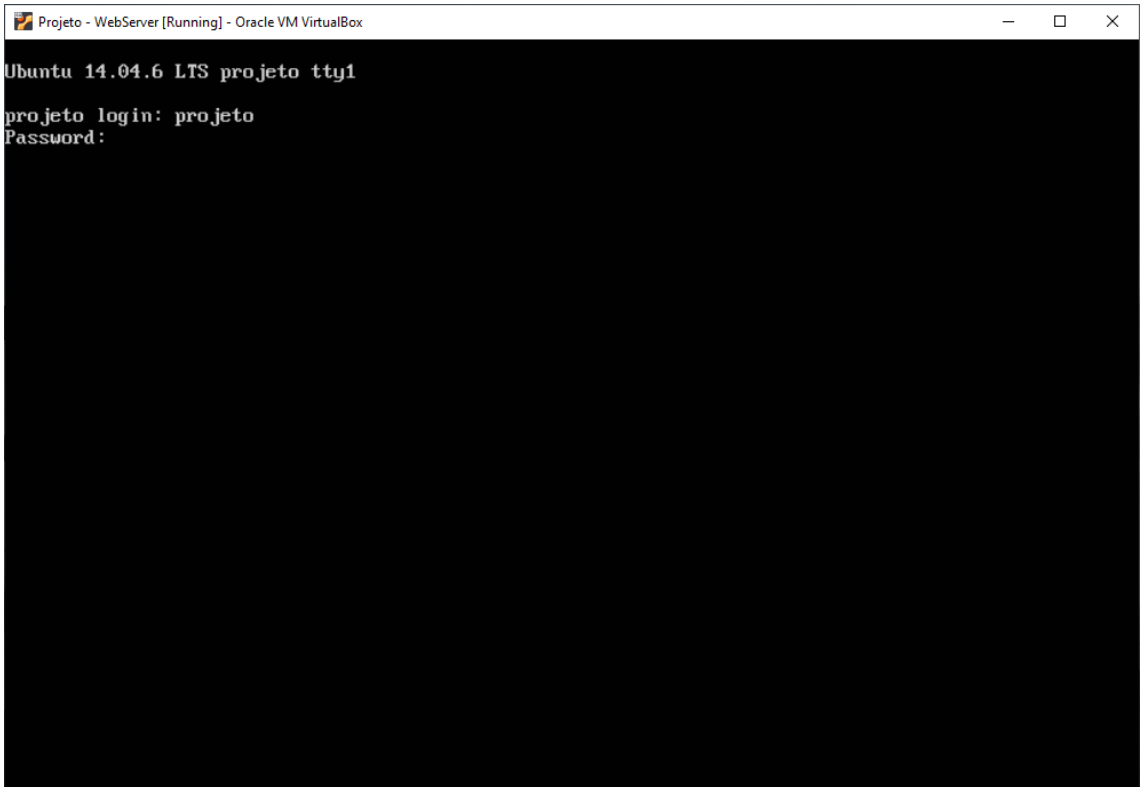


Figura 93 - Ubuntu ecrã de login com senha

3. Iniciar o web service, com o comando "**sudo ./start.sh**" e colocar a senha "**qwerty**".

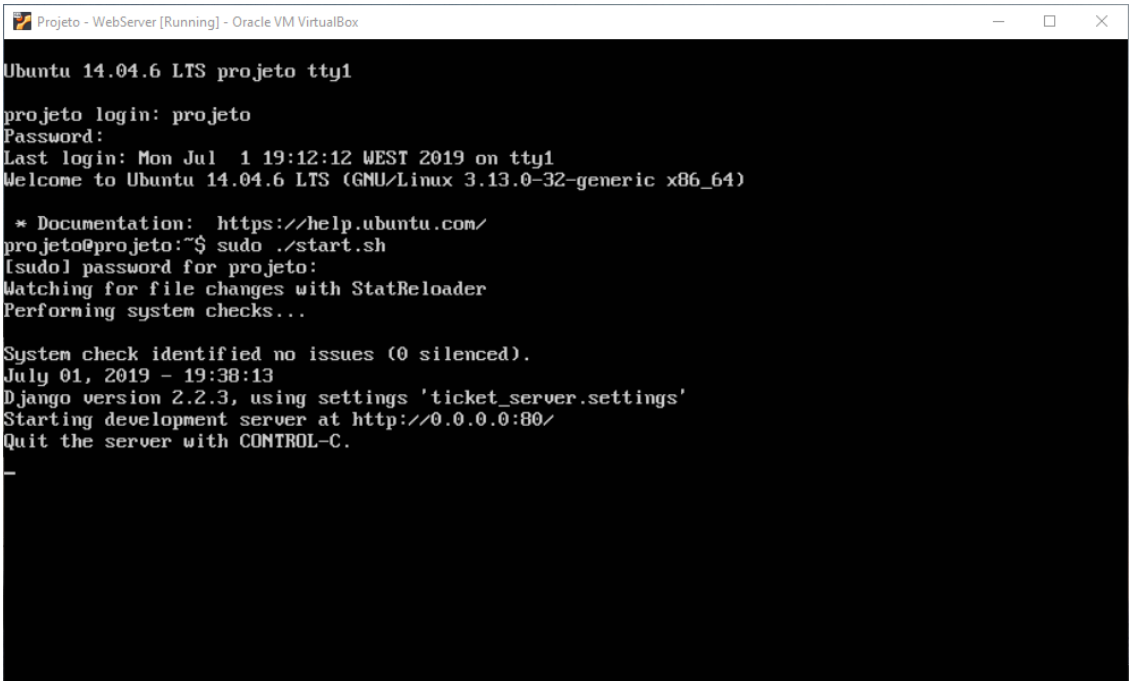


Figura 94 - Ubuntu ecrã com servidor iniciado

4. Arrancar com a máquina "**Projeto – Android**" e aguardar até esta estar ligada;

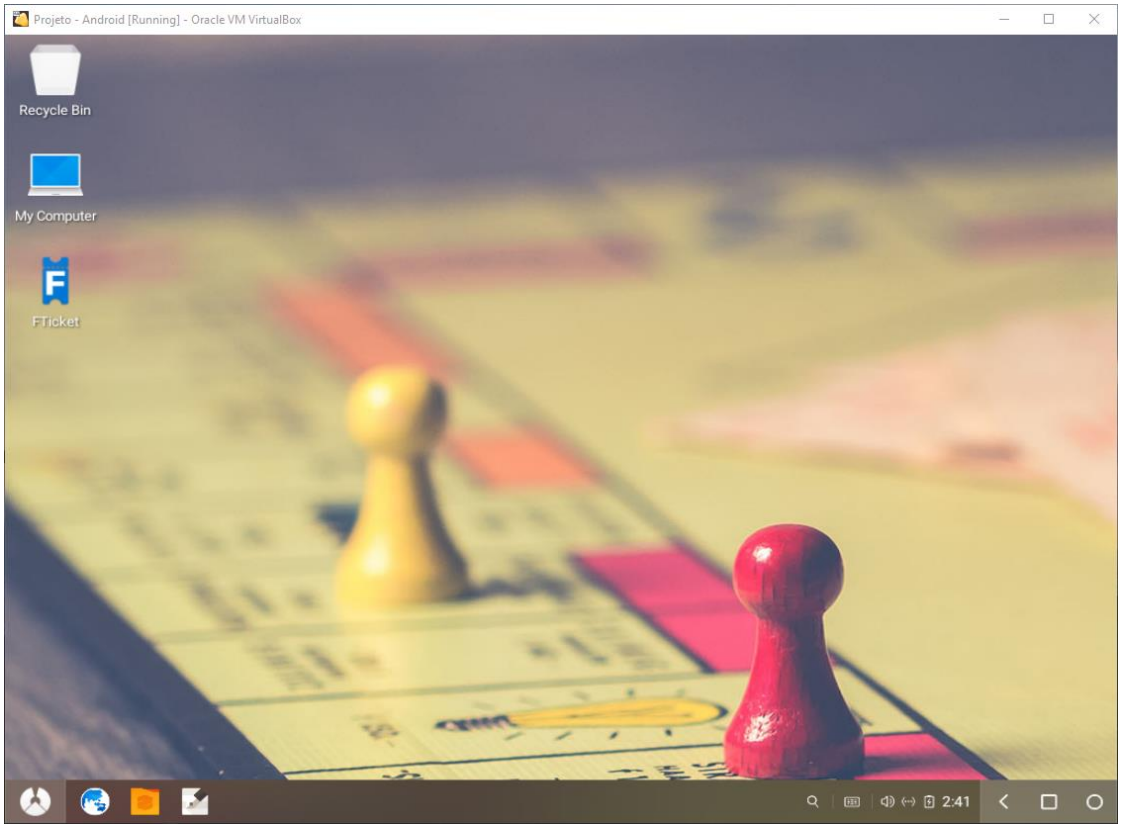


Figura 95 - Ambiente de trabalho do Android

5. Caso o cursor não esteja visível na máquina virtual, desative a integração do rato (Mouse integration);

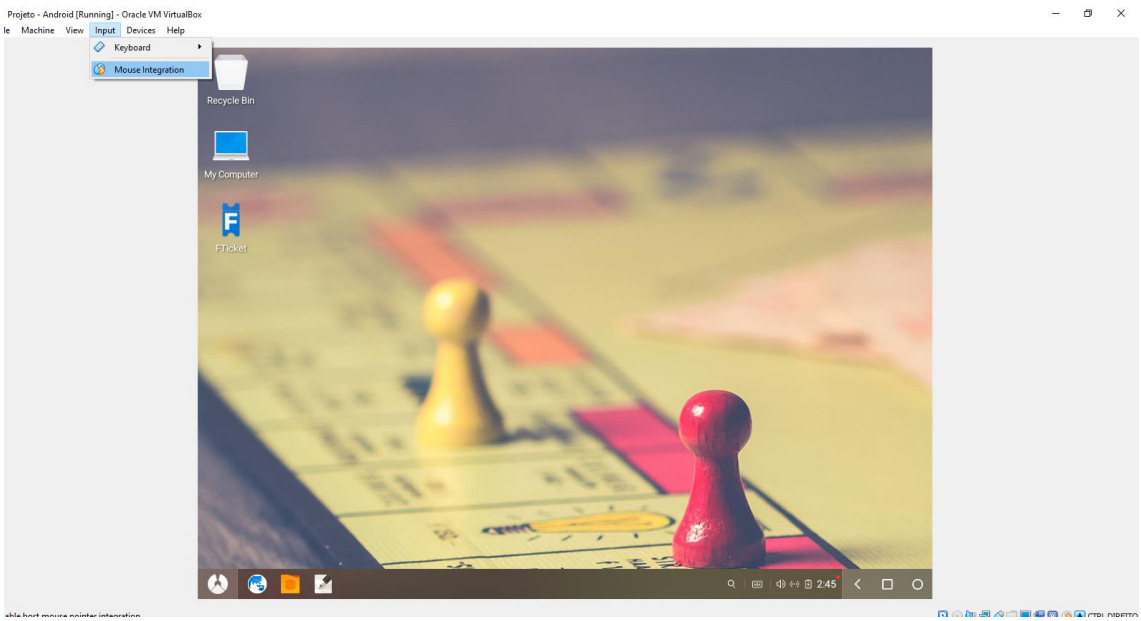


Figura 96 - Desativar integração do rato na virtual box

6. Fazer duplo clique sobre a aplicação "**FTicket**" que está no ambiente de trabalho;

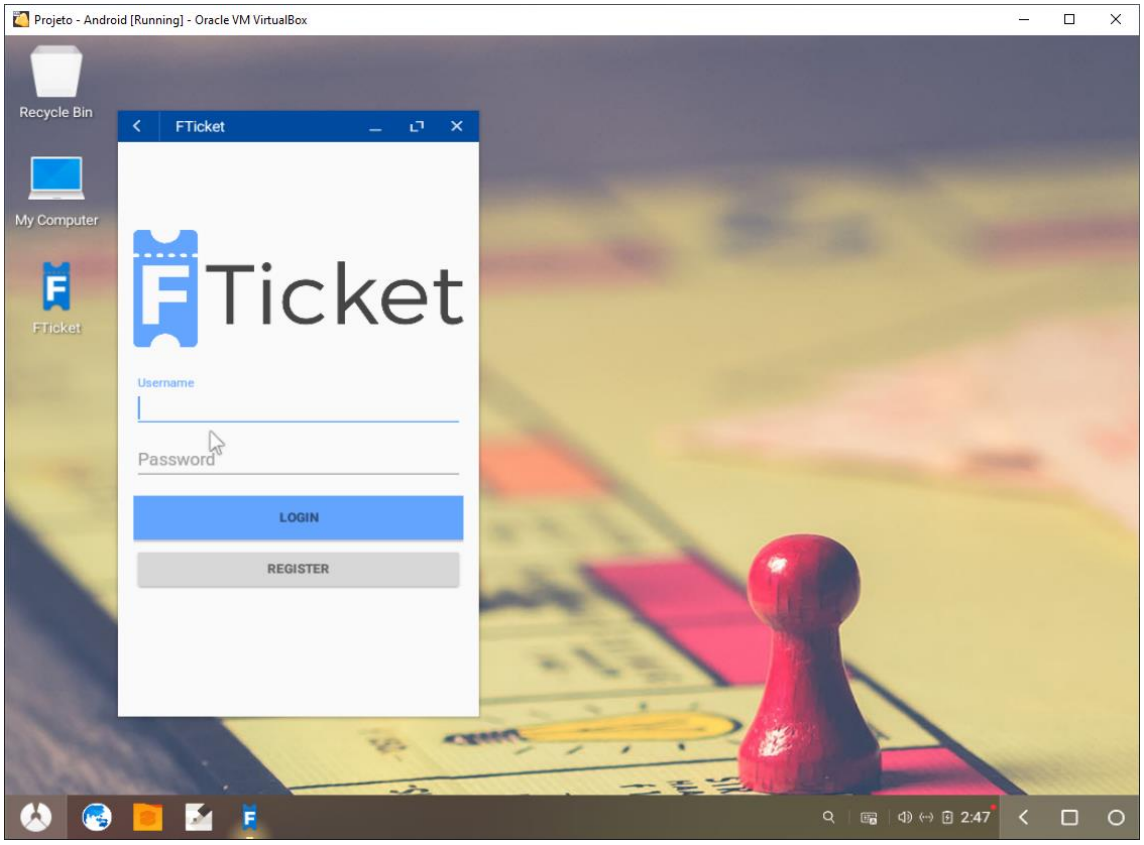


Figura 97 - Aplicação iniciada no Android

15 Anexo 04 – Manual do programador –Android

Para esta aplicação disponibilizou-se o Javadoc's para consulta a informação acerca das várias funções. Este acompanha o relatório com o nome "*FTicket – JavaDoc Android*". Os pacotes estão distribuídos em dois grandes grupo, como podemos ver na Figura 98, o grupo *pt.forever.school.fticket.api* e o *pt.forever.school.fticket.ui*. No pacote *pt.forever.school.fticket.api* estão as classes utilizadas na gestão dos dados da comunicação com o web server, desde a classes utilizadas para troca de informação, até as classes de armazenamento dos dados em cache. Como tal, o módulo de autenticação também está incluído nesta secção.

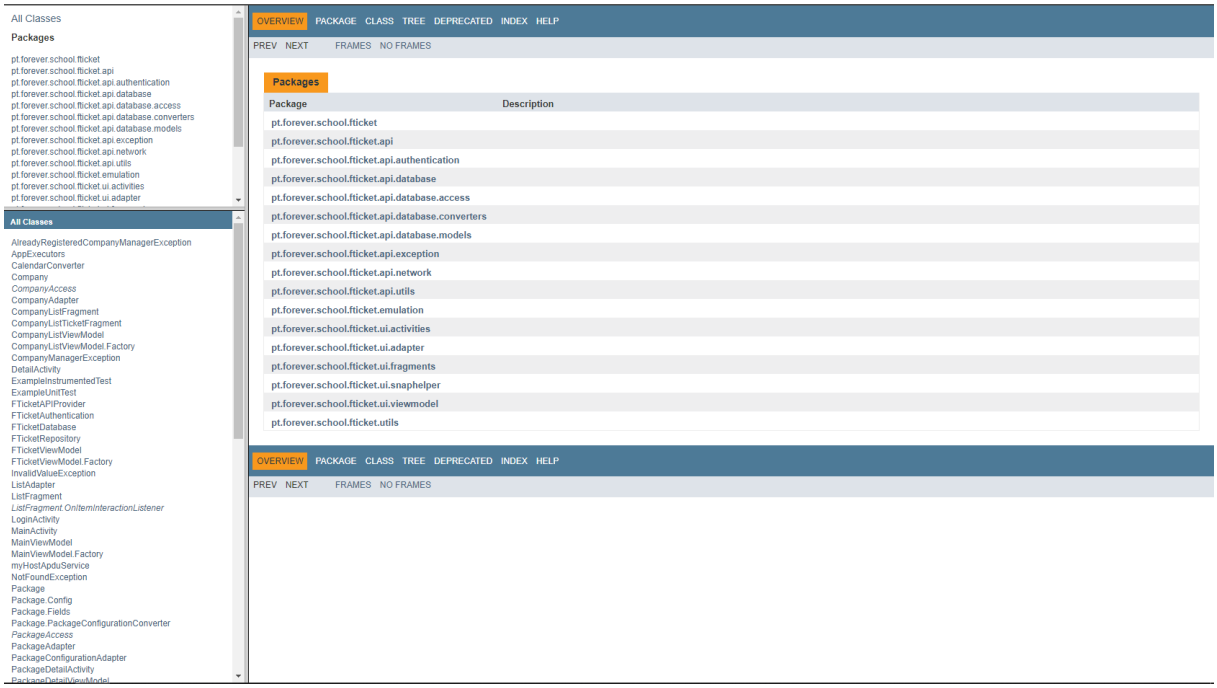


Figura 98 - Lista de pacotes presentes na aplicação Android

No pacote de autenticação, estão disponíveis as classes *User* e a classe *FTicketAuthentication.* A classe *User* contem toda a informação necessária para a identificação de um utilizador, perante a API do web server. Esta informação pode ser vista na Figura 99. Esta informação consiste no nome de utilizador, a token de autenticação, entre outros. Esta classe implementa a classe *Parcelable* do Android, para poder ser armazenada e ser acedida a posteriori.

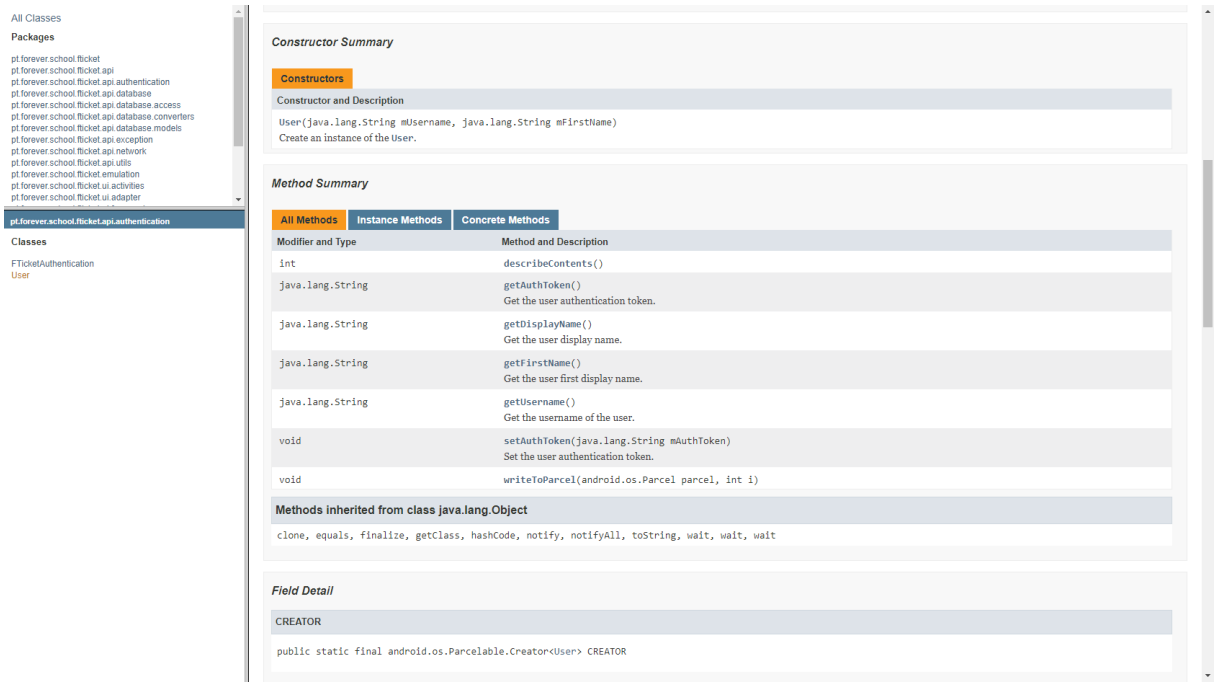


Figura 99 - Classe que contem a informação do utilizador autenticado

A outra classe pertencente a este pacote é a classe de autenticação, é a *FTicketAuthentication*, que está responsável por efetuar os pedidos de autenticação ao web server, como podemos ver na Figura 100.

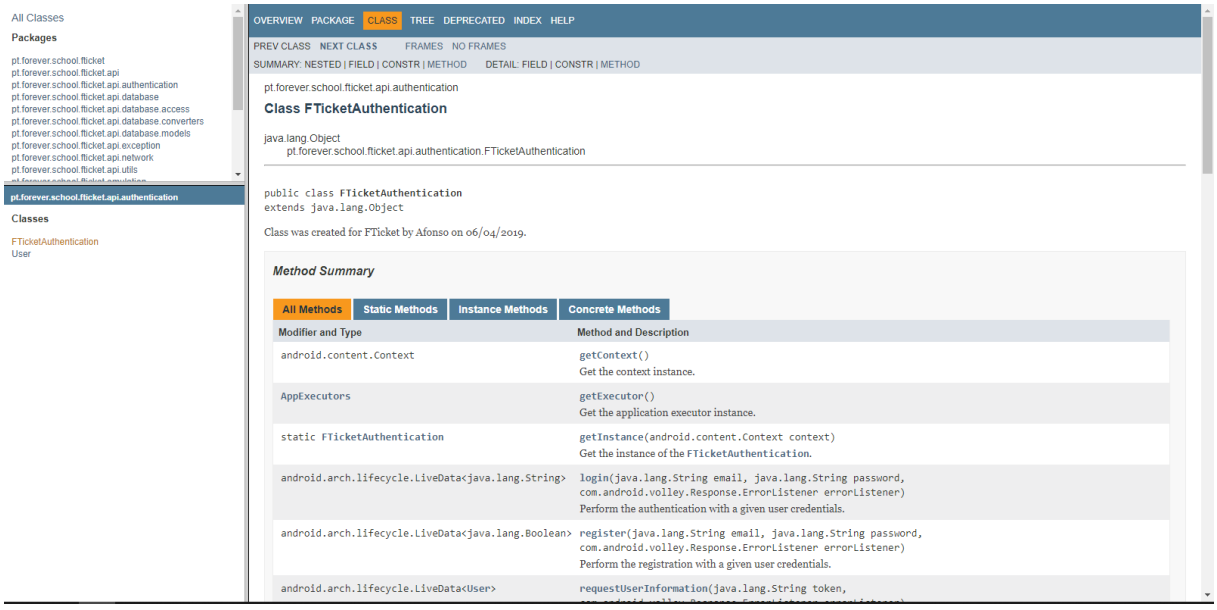


Figura 100 - Classe utilizada como interface de autenticação com o web server

A classe *FTicketAPIProvider* serve de interface entre a aplicação Android e o web server, sendo esta responsável por aceder aos caminhos devidos para obter a informação pretendida. A documentação desta classe encontra-se na Figura 101. Dentro desta classe (uma

nested class) temos outra classe denominada *APIPath* que contem os vários caminhos utilizados para obter as informações do web server.

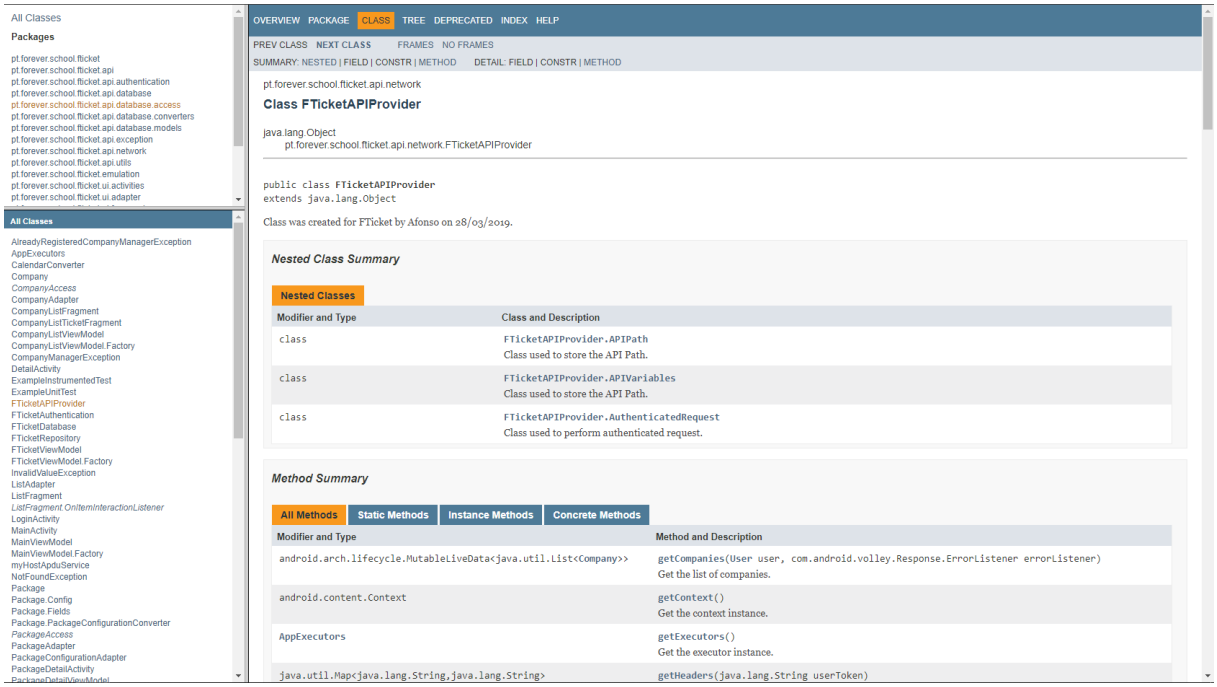


Figura 101 - Documentação da classe FTicketAPIProvider

A classe *FTicketDatabase* é a classe que server de interface com a base de dados. Esta está responsável por consultar a base de dados e retornar os dados á aplicação sem que este tenha de se preocupar como deve transformar a informação no formato adequado. A documentação desta classe, pode ser vista na Figura 102.

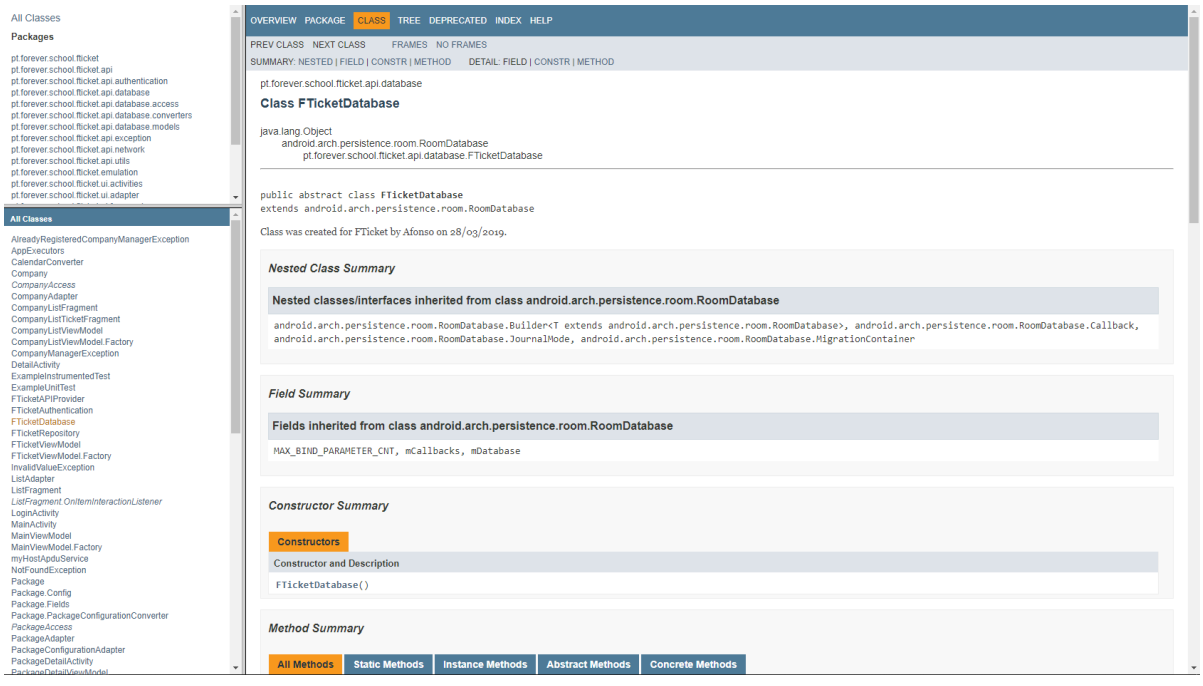


Figura 102 - Documentação da classe FTicketDatabase

Esta classe comunica com outras interfaces que fazem a gestão das tabelas. Essas interfaces estão presentes no pacote *pt.forever.school.fticket.api.database.access* e são *CompanyAccess*, *PackageAccess*, *PackageTypeAccess* e *TicketAccess*, em que a documentação pode ser vista na Figura 103, Figura 104, Figura 105 e Figura 106.

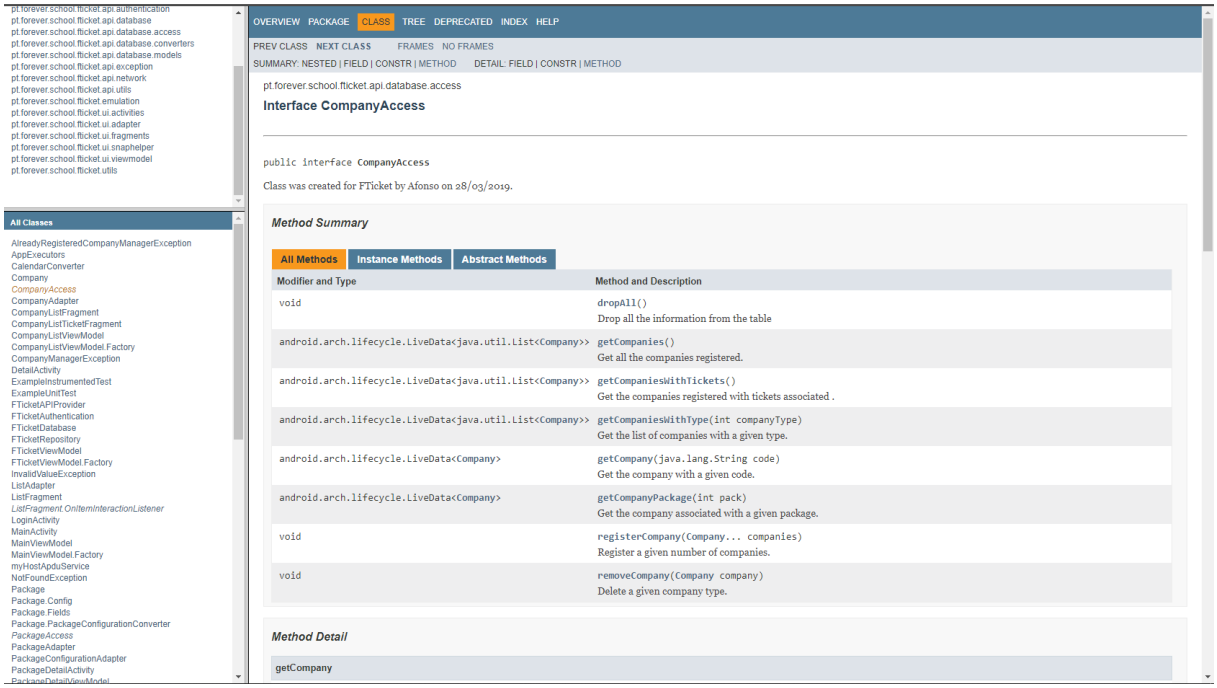


Figura 103 - Documentação da interface da tabela Company

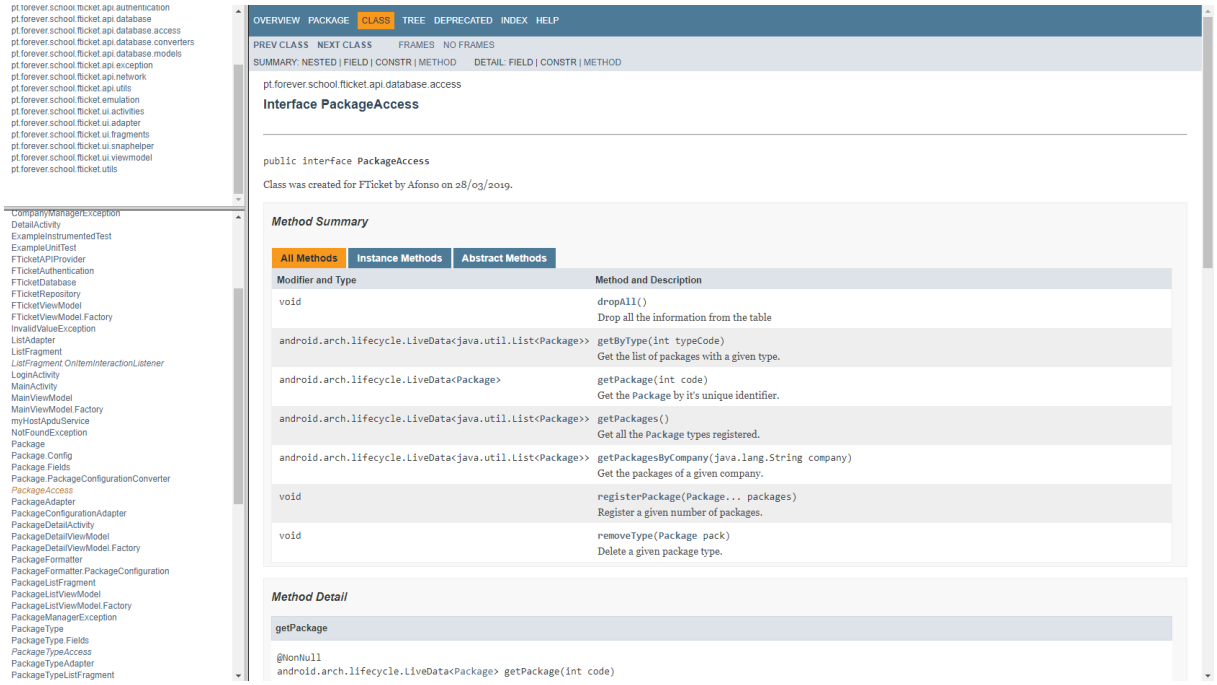


Figura 104 - Documentação da interface da tabela Package

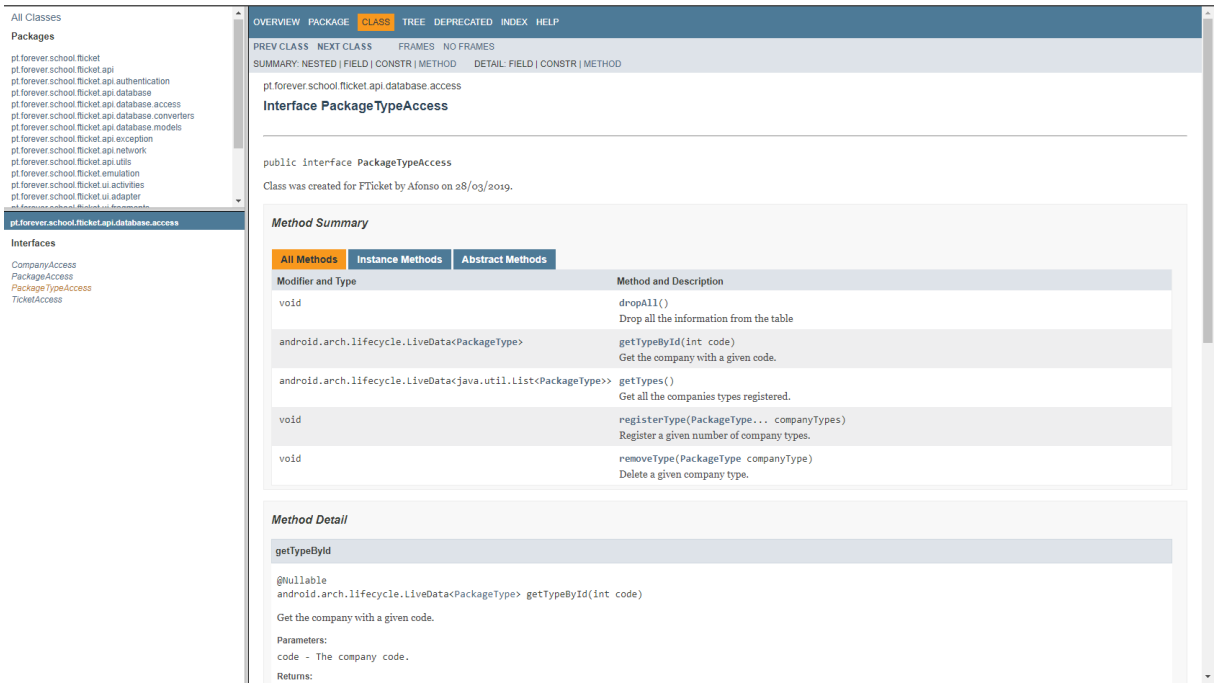


Figura 105 - Documentação da interface da tabela PackageType

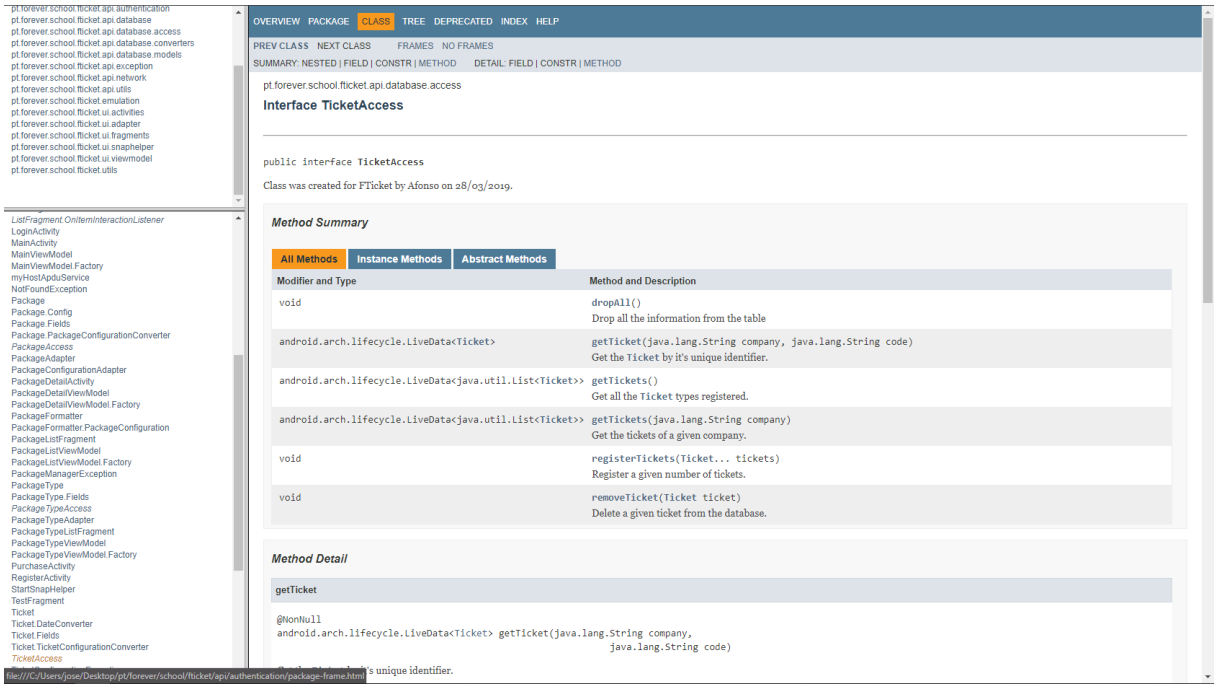


Figura 106 - Documentação da interface da tabela Ticket

A classe *FTicketRepository* é a classe que serve de interface de dados. Esta interface faz a gestão entre os dados que estão em cache e os dados que são transferidos do web server de uma forma transparente, isto permite maior velocidade de resposta, porque se estes dados já estiverem presentes em cache, é escusado estar a fazer um pedido a API do web server que é mais demorada. A documentação pode ser vista na Figura 107.

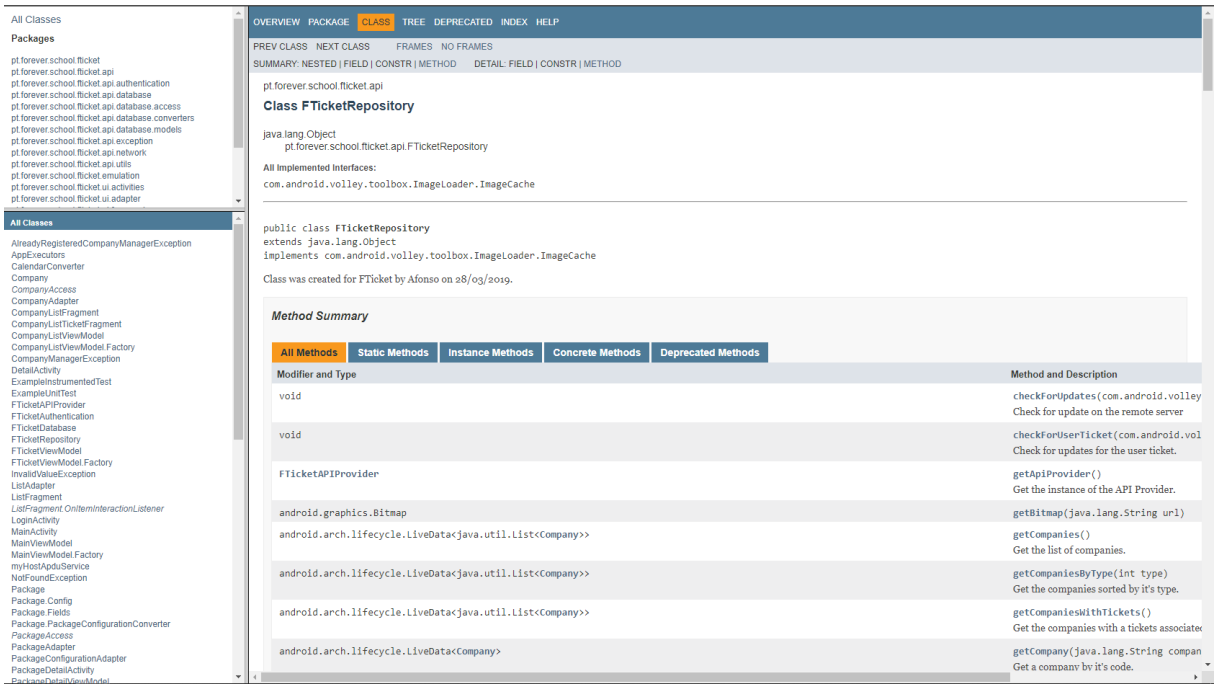


Figura 107 - Documentação da classe FTicketRepository

No pacote *pt.forever.school.fticket.ui* estão as classes utilizadas na representação visual da informação. Esta pacote está divido em pacotes mais pequenos, em que cada um deles guarda uma parte distinta das classes como podemos ver na Figura 108. Estes pacotes são *activities* onde estão as classes das várias atividades da aplicação, como o ecrã de início de sessão, de registo de utilizador, entre outros. O módulo *pt.forever.school.fticket.ui.adapter* que serve para popular as listas com informações de companhias, pacotes, entre outros, *pt.forever.school.fticket.ui.fragments* são pequenas partes visuais que podem ser reutilizadas em várias atividades distintas, *pt.forever.school.fticket.ui.snaphelper* é um pacote auxiliar para as listas *e pt.forever.school.fticket.ui.viewmodel* que serve para preservar a informação ao longo de mudanças de estado do telemóvel, tal como a rotação de ecrã, interrupção da aplicação para atender uma chamada, entre outras causas que podem ser consultadas na classe ViewModel do Android.

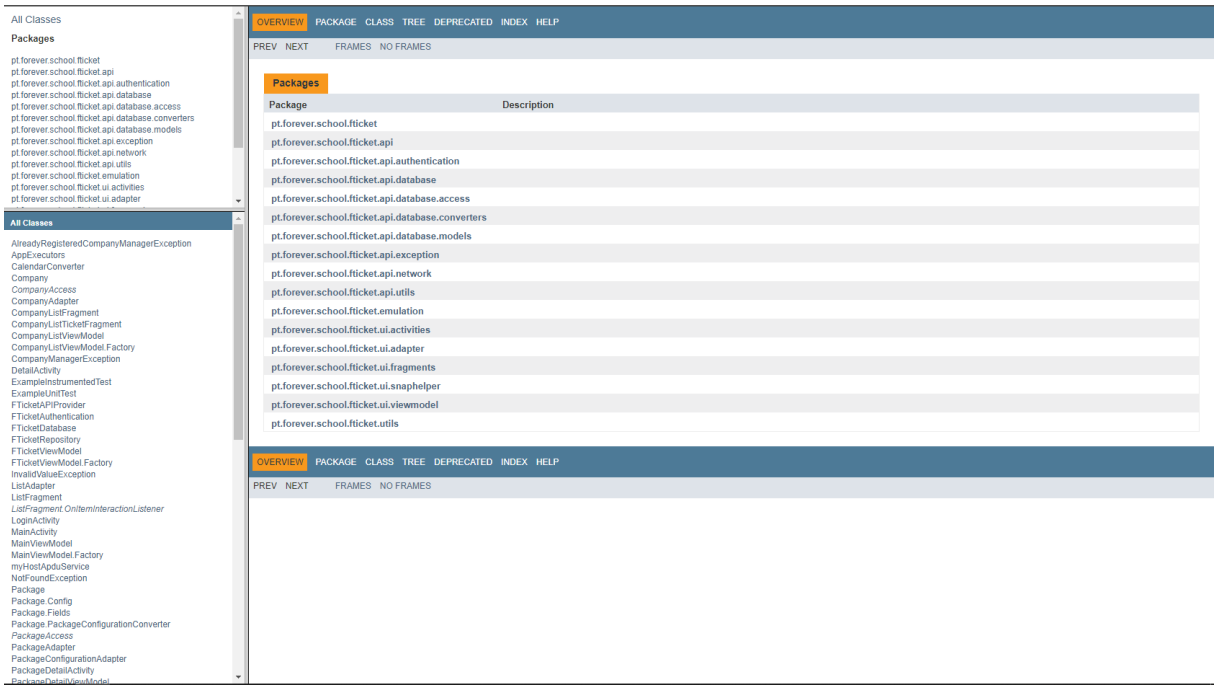


Figura 108 - Listagem de pacotes na documentação

A classe *FTicketViewModel* é a classe que mantém a informação entre as várias sessões da aplicação é a classe que armazena a ligação á interface do *FTicketRepository*.

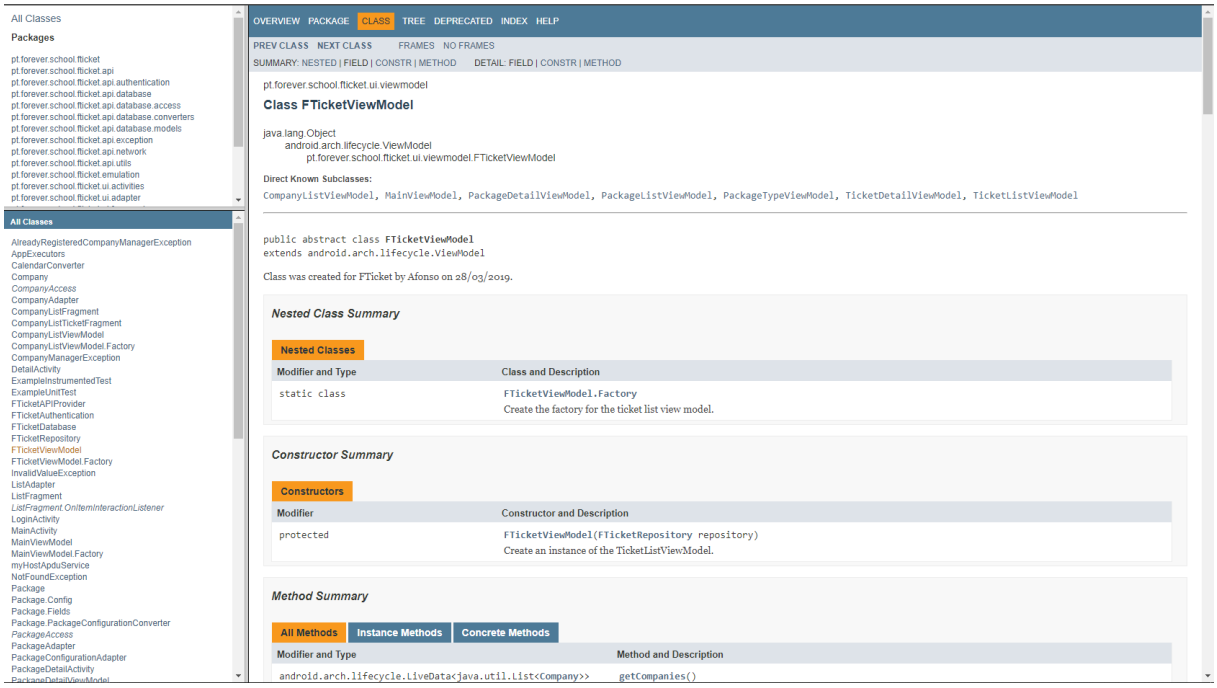


Figura 109 - Documentação da classe FTicketViewModel

16 Anexo 05 – Manual do programador –Web API

A Web API é composta por 2 interfaces, a interface dos utilizados e a interface das máquinas. A interface do utilizador contem os caminhos que podem ser vistos na Figura 110 e na Figura 111.

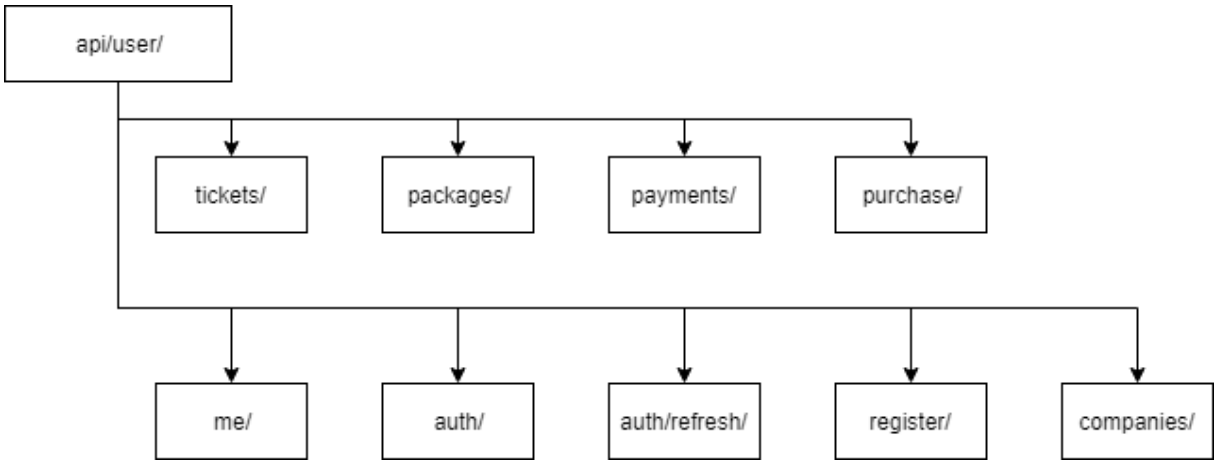


Figura 110 - Caminhos da user api

Caminho	Métodos http	Requer autenticação	Descrição
/api/user/tickets/	GET	Sim	Devolve todos os bilhetes associados ao utilizador atual.
/api/user/packages/	GET	Sim	Devolve todos os pacotes registados na base de dados.
/api/user/payments/	GET	Sim	Processa um pagamento que já tenha sido pago.
/api/user/purchase/	GET, POST	Sim	Efetua uma compra de um dado pacote.
/api/user/me/	GET	Sim	Devolve informações sobre o utilizador autenticado.

/api/user/auth/	POST	Não	Efetua operações de autenticação de um utilizador.
/api/user/auth/refresh/	POST	Não	Refresca um token do utilizador, quando esta está expirada.
/api/user/register/	POST	Não	Registar um novo utilizador no sistema.
/api/user/companies/	GET	Sim	Devolve todas companhias registados na base de dados.

Figura 111 - Caminhos da API do utilizador

A interface das máquinas contem os caminhos que podem ser vistos na Figura 112 e Figura 113. As máquinas utilizam um sistema de autenticação baseado em token e chave secreta. No *Header* do pedido http deve ser enviado 2 valores que correspondem a autenticação da mesma, sendo estes *Api-Token* e *Api-Secret-Key*. Estes dois valores têm de coincidir com os valores que se foram registados pelo funcionário que instalou a máquina na dada empresa, para esta poder ter acesso aos dados a que foi estipulado.

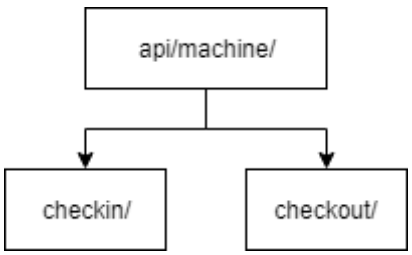


Figura 112 - Caminhos da machine api

Caminho	Métodos http	Requer autenticação	Descrição
/api/machine/checkin/	POST	Sim	Executa a operação de entrada num dado bilhete.
/api/machine/checkout/	POST	Sim	Executa a operação de saída num dado bilhete.

Figura 113 - Caminhos da API das máquinas