Universidade da Beira Interior
Departamento de Informática

Nº 76 - 2012

MobileTicket - Sistema Ticketing para Android

Elaborado por:

Rúben A. A. Gonçalves

Orientador:

Prof. Doutor Paul Andrew Crocker

AGRADECIMENTOS

Agradeço em especial à Mafalda Duarte, minha namorada pelo apoio que me deu durante a realização de todo o projeto.

Gostaria de agradecer também à minha família e amigos por todo o apoio dado, assim como às duas instituições de ensino que me ajudaram, a Universidade da Beira Interior e a Universidade de São Paulo.

Conteúdo

Agradecimentos **i**

Conteúdo **iii**

Lista de Figuras **v**

Acronimos **vii**

1 INTRODUÇÃO **1**
1.1 Introdução 1
1.2 Motivação 2
1.3 Objectivos 2
1.4 Estrutura do Relatório 3

2 Estudo do Mercado **5**
2.1 Dispositivos Móveis e Smartphones 5
2.2 Android 5
2.2.1 Ascenção na Europa 7
2.2.2 Android nos Estados Unidos da América 8
2.2.3 Versões 9
2.3 Aplicações Relacionadas 10

3 TRABALHO DESENVOLVIDO **11**
3.1 Tecnologias e ferramentas 11
3.2 Instalação e configuração 13
3.3 Bibliotecas externas 14
3.4 Descrição do sistema - MobileTicket 15
3.5 Desenvolvimento 16

3.5.1 Projeção 16
3.5.2 Base de dados 17
3.5.3 Servidor 29
3.5.4 Aplicação Móvel 39

4 CONCLUSÕES E TRABALHO FUTURO **57**
4.1 Trabalho futuro 57

Bibliografia **59**

Lista de Figuras

2.1 Ascenção do sistema operativo Android . . . 6
2.2 Estatisticas do Android na Europa . . . 7
2.3 Estatisticas do Android nos E.U.A. . . . 8
2.4 Android distribuições . . . 9
2.5 Mobile Ticket App . . . 10

3.1 MobileTicket . . . 15
3.2 MobileTicket - Funcionamento do sistema . . . 16
3.3 Modelo de Entidade e Relacionamento . . . 18
3.4 Descrição da tabela customer . . . 18
3.5 Descrição da tabela seller . . . 19
3.6 Descrição da tabela location . . . 19
3.7 Descrição da tabela optional field . . . 19
3.8 Descrição da tabela ticket . . . 19
3.9 Web método login . . . 29
3.10 Web método login . . . 32
3.11 Web método getTicketsList . . . 34
3.12 Web método getTicketUpdatedByCustomer . . . 36
3.13 Web método sendMMS . . . 38
3.14 Estrutura de um projeto Android . . . 39
3.15 MobileTicket splash screen . . . 42
3.16 MobileTicket splash screen . . . 42
3.17 MobileTicket lista de bilhetes . . . 46
3.18 MobileTicket lista de bilhetes . . . 47
3.19 MobileTicket sub menu ordenar . . . 49
3.20 MobileTicket informação do bilhete . . . 52
3.21 MobileTicket informação do bilhete . . . 53
3.22 MobileTicket informação do bilhete . . . 54

3.23 MobileTicket informação do bilhete 55

Lista de Acrónimos

ADT - *Android Development Tools*

API - *Application Programming Interface*

APK - *Application Package*

AVD - *Android Virtual Device*

AVDM - *Android Virtual Device Manager*

DAO - *Data Access Object*

GUI - *Graphical User Interface*

IDE - *Integrated Development Environment*

OHA - *Open Handset Aliance*

SDK - *Software Development Kit*

SGBD - *Sistema de Gestão de Base de Dados*

SQL - *Structured Query Language*

Capítulo 1

INTRODUÇÃO

1.1 Introdução

Nos últimos anos, a internet transformou-se num verdadeiro mercado virtual, é cada vez mais fácil e seguro adquirir algum produto ou serviço através da internet. Ao crescimento da internet e das compras online regista-se também um aumento do uso de dispositivos móveis e do sistema operativo Android, por isso foi estudada a possibilidade de conceção de um sistema de bilhetes eletrónicos móveis, algo ainda pouco explorado no mundo móvel.

A ideia passa por desenvolver um sistema completamente abstrato do vendedor, ou seja, um sistema para todo o tipo de bilhetes, desde para peças de teatro, até bilhetes para transportes. Com um sistema no dispositivo móvel, o utilizador ganha uma facilidade na portabilidade do bilhete, bem como segurança, pois o bilhete torna-se algo digital, mais difícil de perder, juntado ainda a estas vantagens seria completamente desnecessário a impressão dos bilhetes, poupando assim o utilizador final de custo de impressão e o ambiente com um gasto desnecessário de papel.

1.2 Motivação

No âmbito da disciplina de Projeto, foi proposto a realização de um sistema ticketing para o sistema operativo móvel Android. A escolha deste projeto, deveu-se ao interesse de realizar um sistema ticketing abstrato na área móvel, de modo a desenvolver aptidões na programação móvel, aprofundar conhecimentos nas áreas de base de dados e programação Java.

1.3 Objectivos

Este projeto tem como objetivo desenvolver um sistema ticketing abstrato para o sistema operativo móvel Android. Este pretende ser independente do vendedor do bilhete, armazenando os bilhetes numa base de dados, para posteriormente os clientes poderem usufruir dos bilhetes comodamente.

Outro dos objetivos do projeto é explorar o sistema operativo Android, tendo em conta que é uma plataforma afirmada e em constante crescente no mercado. Pretende-se também aprofundar alguns dos conhecimentos adquiridos ao longo do curso, bem como o desenvolvimento da capacidade de tomada de decisões e resolução de eventuais problemas.

1.4 Estrutura do Relatório

Este relatório está dividido em três capítulos. Segue-se um breve resumo de cada um deles:

Capítulo 1 - Este capítulo, é constituído pela *Introdução*, onde é feita uma pequena noção introdotória sobre o sistema desenvolvido. É apresentada a *Motivação* que levou à escolha deste projecto e também os seus *Objectivos*.

Capítulo 2 - Este capítulo é constituído por um pequeno estudo sobre o Android e seus concorrentes, suas distribuições e também sobre algumas aplicações semelhantes com a proposta.

Capítulo 3 - No capítulo três é relatado de forma pormenorizada o desenvolvimento de todo o sistema desenvolvido. Serão também identificadas todas as ferramentas essenciais utilizadas.

Capítulo 4 - No último capítulo serão tiradas algumas conclusões e apresentadas as dificuldades encontradas no desenvolvimento do projecto. Será ainda perspectivado o trabalho futuro relativo ao sistema.

Capítulo 2

Estudo do Mercado

2.1 Dispositivos Móveis e Smartphones

O mercado dos dispositivos móveis é um mercado que cresce de dia para dia, segundo a Gartner as vendas mundiais de dispositivos móveis totalizaram 440,5 milhões de unidades no terceiro trimestre de 2011, um aumento de 5,6 por cento em relação ao mesmo período do ano passado. Já vendas de smartphones para consumidores finais atingiram 115 milhões de unidades no terceiro trimestre de 2011, um aumento de 42 por cento em relação ao terceiro trimestre de 2010.

Com um mercado em forte crescimento e com tanto ainda para explorar, é sempre uma mais valia desenvolver para dispositivos móveis.

2.2 Android

O Android é um sistema operativo para dispositivos móveis, baseado em Linux. Foi inicialmente desenvolvido pelo Google e posteriormente pela Open Handset Alliance, mas a Google é a responsável pela gestão do produto e pela engenharia de processos.

Desde o seu lançamento oficial que tem vindo a conquistar uma grande quota de mercado, tornando-se muito atrativo para o desenvolvimento de aplicaçõese, hoje é muito notável o crescimento do android e a sua consolidação como preferência dos consumidores.

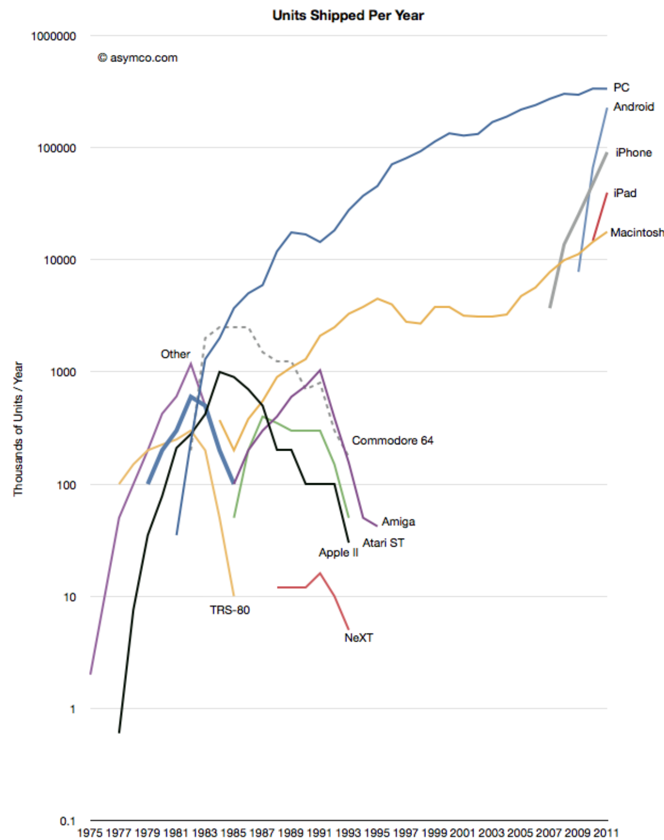


Figura 2.1: *Ascenção do sistema operativo Android*

O gráfico feito pela Asymco, que avalia "a ascensão e queda da computação pessoal", de 1975 a 2011. O Android OS está muito perto de quebrar a histórica barreira de vendas dos computadores pessoais e consequentemente liderar o seguimento computacional ainda em 2012.

2.2.1 Ascenção na Europa

Num estudo revelado pela renomada comScore, o mercado europeu demonstra ter adotado o sistema operativo móvel do Google, o Android, como o seu preferido. Tanto que a taxa de crescimento dele é de 16,1 por cento em um ano (julho 2010/2011). Neste mesmo período o iOS, sistema operativo móvel da Apple, obteve um crescimento de apenas 1,2 por cento.

Top Smartphone Platforms in EU5 by Share of Smartphone Users*
3 Month Average Ending July 2011 vs. July 2010
Total EU5 (DE, FR, IT, ES and UK) Mobile Subscribers, Age 13+
Source: comScore MobiLens

Smartphone Platform	Share (%) of EU5 Smartphone Users		
	Jul-10	Jul-11	Point Change
Total Smartphone Users	*100.00%*	*100.0%*	*0.0*
Symbian	53.9%	37.8%	-16.1
Google	6.0%	22.3%	16.2
Apple	19.0%	20.3%	1.2
RIM	8.0%	9.4%	1.5
Microsoft	11.5%	6.7%	-4.8

Figura 2.2: *Estatisticas do Android na Europa*

2.2.2 Android nos Estados Unidos da América

Segundo os dados mais recentes para o mercado norte-americano, o Android já arrecada uma quota de mercado que chega aos 54 por cento respeitantes aos smartphones. Com estes valores, o Android suplanta em mais do dobro o iOS da Apple, visto que este último conseguiu apenas uns respeitosos 28 por cento

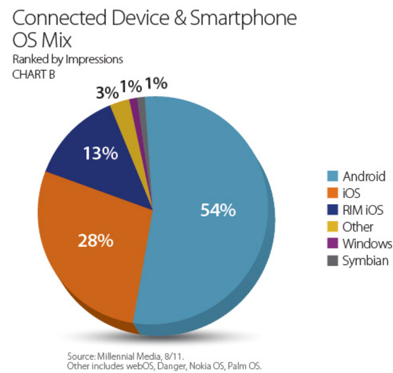
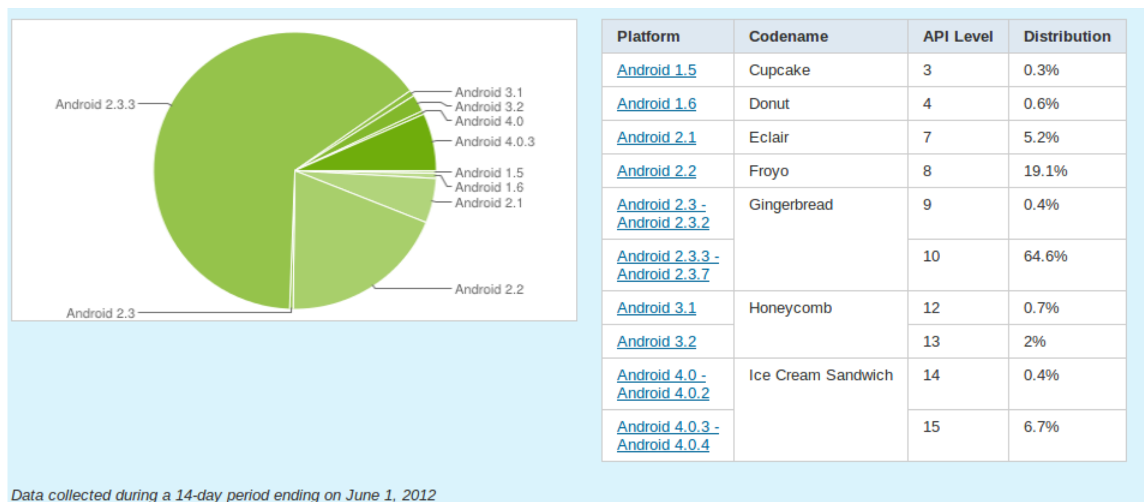


Figura 2.3: *Estatisticas do Android nos E.U.A.*

2.2.3 Versões

Com várias versões já lançadas, era necessário escolher uma versão base para o desenvolvimento da aplicação MobileTicket.



Platform	Codename	API Level	Distribution
Android 1.5	Cupcake	3	0.3%
Android 1.6	Donut	4	0.6%
Android 2.1	Eclair	7	5.2%
Android 2.2	Froyo	8	19.1%
Android 2.3 - Android 2.3.2	Gingerbread	9	0.4%
Android 2.3.3 - Android 2.3.7		10	64.6%
Android 3.1	Honeycomb	12	0.7%
Android 3.2		13	2%
Android 4.0 - Android 4.0.2	Ice Cream Sandwich	14	0.4%
Android 4.0.3 - Android 4.0.4		15	6.7%

Figura 2.4: *Android distribuições*

Embora sendo a versão 2.3 a mais utilizada, foi esconhida para o desenvolvimento da aplicação a versão 2.2, devido ao fato de ainda haver muitos dispositivos com essa mesma versão.

2.3 Aplicações Relacionadas

Aplicações ticketing para Android existem algumas disponiveis, mas são aplicações dedicadas a uma só empresa, o que limita muito o utilizador, pois caso pretenda ter dois bilhetes de empresas diferentes é obrigado a ter duas aplicações distintas.

Exemplo de aplicações:

- Heathrow mobile app
- IRCTC Book tickets online

Ambas as aplicações mencionadas acima, são dedicadas a uma só empresa, e têm sempre ligadas à aplicação um módulo de aquisição de bilhetes.

Uma aplicação norte americana nominada Mobile Ticket App, surge em boa posição relativamente aos seus concorrentes. Ela permite comprar e pesquisar bilhetes de eventos das mais variadas empresas mas apenas nos Estados Unidos da América.

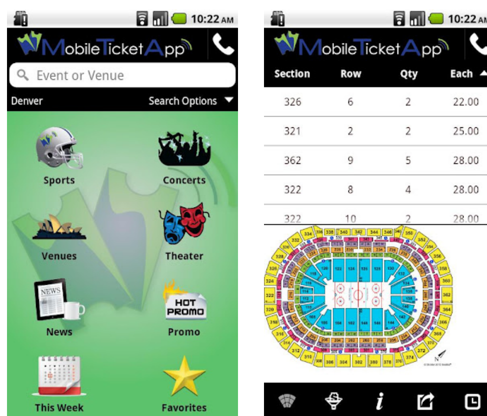


Figura 2.5: *Mobile Ticket App*

A ideia é desenvolver uma aplicação completamente independente do vendedor e do pais, e ter somente o módulo de listagem dos bilhetes adquiridos, via online.

Capítulo 3

TRABALHO DESENVOLVIDO

3.1 Tecnologias e ferramentas

Tecnologias

As tecnologias utilizadas no projeto foram:

- Java - Linguagem de programação utilizada para a programação do servidor e da aplicação móvel.
- MySQL - Sistema gestor de base de dados utilizado para guardar a informação relativa aos bilhetes digitais.

Ferramentas

As ferramentas utilizadas no projeto foram:

- Eclipse - O Eclipse é um IDE[1] open source e foi utilizado para desenvolver a aplicação móvel.
- NetBeans - O NetBeans é um IDE e foi utilizado para o desenvolvimento do servidor de suporte à aplicação móvel.
- Mysql Administrator - É uma ferramenta que pertence ás ferramentas GUI[2] do mysql e foi utilizada para administrar a base de dados, os utilizadores e as suas respetivas permissões.
- Mysql Query browser - É uma ferramenta que pertence ás ferramentas GUI do mysql e foi utilizada para criar a base de dados e realizar todos os scripts SQL[3].
- Android SDK[4] - É um conjunto de ferramentas de apoio ao desenvolvimento de software para a plataforma Android. O SDK possui o AVD[5], que permite simular um dispositivo móvel, com todas as versões existentes desta plataforma até ao momento.
- GlassFish - É um servidor de aplicação open source, para a plataforma Java EE.

[1] Integrated Development Environment
[2] Graphical User Interface
[3] Structured Query Language
[4] Software Development Kit
[5] Android Virtual Device

3.2 Instalação e configuração

Em baixo encontra-se descrito detalhadamente como instalar e configurar todas as ferramentas inerentes ao projeto. O sistema operativo utilizado para o desenvolvimento do projeto foi o Linux Ubuntu.

Instalar o Android SDK

1. Download do Android SDK em http://developer.android.com/index.html, e prosseguir com a sua instalação.
2. Iniciar o Android SDK.
    (a) Instalar as versões do Android disponíveis especialmente acima da 2.1 inclusive.
    (b) Iniciar o AVDM[6] em tools->Manage AVDs.
    (c) Criar um novo AVD com a versão 2.2 com o google maps.

Instalar o Eclipse

1. Download do Eclipse em http://www.eclipse.org/downloads/packages/eclipse-classic-372/indigosr2
2. Instalar o plug-in ADT[7] para o Eclipse. Ajuda->Instalar novo software, e procurar por ADT
3. Indicar o caminho do SDK em windows->preferencias->android->SDK Localization.

[6] Android Virtual Device Manager
[7] Android Development Tools

Instalar o NetBeans e o GlassFish

Fazer o download em http://netbeans.org/downloads/start.html e prosseguir com a instalação.

Instalar o MySQL Community Server

Fazer o download em http://www.mysql.com/downloads/mysql e prosseguir com a instalação.

Instalar o Mysql GUI

Fazer o download em http://www.mysql.com/downloads/workbench e prosseguir com a instalação.

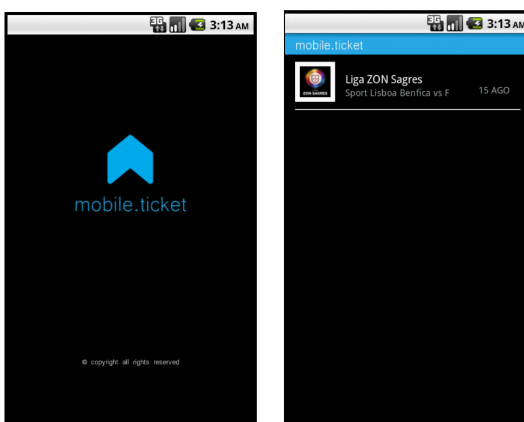
3.3 Bibliotecas externas

As bibliotecas externas utilizadas para o projeto foram:

- kSOAP2 - É a biblioteca que fornece a API para a conexão a um Web Service desenvolvido em Java.
- MySQL connector - É a biblioteca que possiblita a conexão entre uma aplicação e o Mysql SGBD.

3.4 Descrição do sistema - MobileTicket

Ao comprar um bilhete via online, toda a sua informação será enviada para uma base de dados, que posteriormente irá dar origem a um bilhete digital. Quando o cliente aceder à aplicação MobileTicket através de um dispositivo móvel, este irá descarregar automaticamente todos os bilhetes do respetivo cliente para futuramente ser usufruídos.



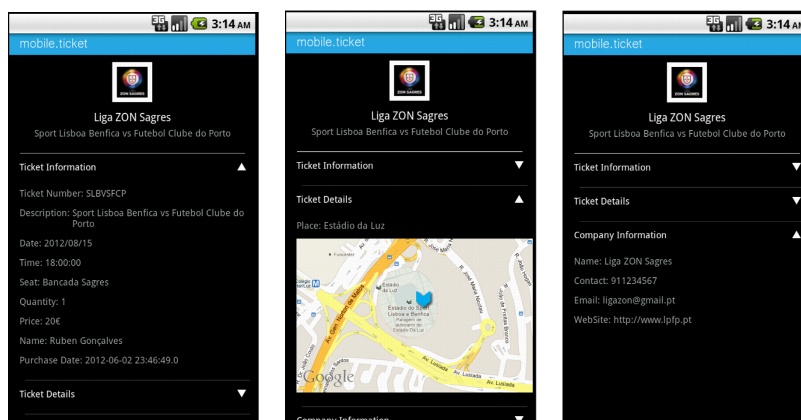


Figura 3.1: *MobileTicket*

3.5 Desenvolvimento

3.5.1 Projeção

Após estabelecida a ideia de realizar um sistema ticketing, o primeiro passo para o começo do seu desenvolvimento foi estruturar quais as ferramentas e tecnologias a serem usadas e estabelecer uma arquitetura para o sistema. Após tudo estabelecido, houve a necessidade de um estudo prévio e uma adaptação ao Android, bem como a integração deste com Web services.

O sistema está dividido essensialmente em três partes distintas:

- Base de dados
- Servidor
- Aplicação móvel

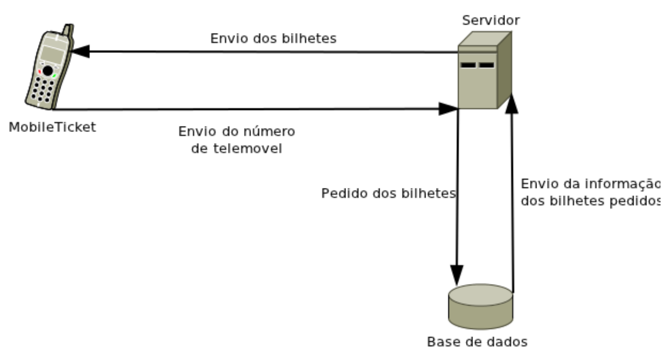


Figura 3.2: *MobileTicket - Funcionamento do sistema*

Sempre que a aplicação MobileTicket é iniciada, é enviado para o servidor, o número de telemóvel do dispositivo para este poder pedir à base de dados caso exista os bilhetes do cliente, e posteriormente enviar para o cliente os bilhetes pedidos.

3.5.2 Base de dados

A base de dados foi desenvolvida em Mysql, tendo como objetivo guardar a informação dos bilhetes, dos vendedores e dos clientes.
Era importante desenvolver uma base de dados, segura e com mecanismos que aumente a performance, pois é nela que contém toda a informação da aplicação, sendo assim foram criados procedimentos armazenados para toda a iteração com a base de dados.
Procedimentos armazenados são compilados do lado do servidor, por isso não é necessário passar pela rede grandes queries o que poderia tirar performance à aplicação.
Uma das principais vantagens dos procedimentos armazenados, é a possibilidade de usar transações, o que significa ou ele executa todas as operações com sucesso, ou não executará nenhuma, essa ação é chamada de ação atómica.

Modelo de Entidade e Relacionamento

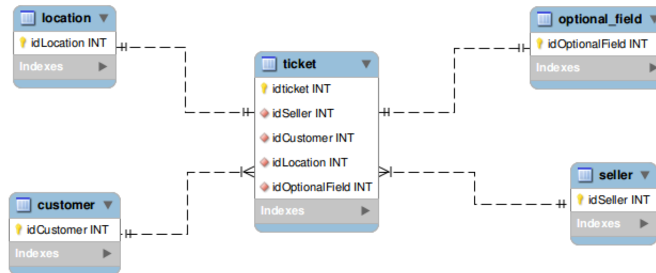


Figura 3.3: *Modelo de Entidade e Relacionamento*

Como é visível na imagem acima, toda a informação é ligada na tabela *ticket*.
O relacionamento entre as tabela *ticket* com a tabela *location* e da tabela *ticket* com a tabela *optional field* é de um para um, para assegurar que cada bilhete terá somente uma localização e tambem somente uma descrição dos campos opcionais.
Já os relacionamentos entre a tabela *ticket* com a tabela *customer* e da tabela *ticket* com a tabela *seller* é de muitos para um, fazendo com que cada vendedor e cliente tenham mais de um bilhete associados a eles.

Tabelas

Tabela customer

Column Name	Data Type	NOT NULL	AUTO INC	Flags	Default Value
idCustomer	INTEGER	☑	☑	UNSIGNED	NULL
phoneNumber	INTEGER	☑	☐	UNSIGNED	NULL
registrationDate	DATETIME	☑	☐		NULL

Figura 3.4: *Descrição da tabela customer*

Tabela seller

Column Name	Data Type	NOT NULL	AUTO INC	Flags	Default Value
idSeller	INTEGER	✓	✓	UNSIGNED	NULL
company	VARCHAR(255)	✓			NULL
logo	VARCHAR(255)	✓			NULL
contact	VARCHAR(255)				'null'
email	VARCHAR(255)				'null'
webSite	VARCHAR(255)				'null'

Figura 3.5: *Descrição da tabela seller*

Tabela location

Column Name	Data Type	NOT NULL	AUTO INC	Flags	Default Value
idLocation	INTEGER	✓	✓	UNSIGNED	NULL
place	VARCHAR(255)	✓			NULL
latitude	VARCHAR(255)	✓			NULL
longitude	VARCHAR(255)	✓			NULL

Figura 3.6: *Descrição da tabela location*

Tabela optional field

Column Name	Data Type	NOT NULL	AUTO INC	Flags	Default Value
idOptionalField	INTEGER	✓	✓	UNSIGNED	NULL
quantity	INTEGER	✓		UNSIGNED	1
price	VARCHAR(255)				'null'
seat	VARCHAR(255)				'null'
customerName	VARCHAR(255)				'null'

Figura 3.7: *Descrição da tabela optional field*

Tabela ticket

Column Name	Data Type	NOT NULL	AUTO INC	Flags	Default Value
idTicket	INTEGER	✓	✓	UNSIGNED	NULL
idSeller	INTEGER	✓		UNSIGNED	NULL
idCustomer	INTEGER	✓		UNSIGNED	NULL
idLocation	INTEGER	✓		UNSIGNED	NULL
idOptionalField	INTEGER	✓		UNSIGNED	NULL
ticketNumber	VARCHAR(255)	✓			NULL
description	VARCHAR(255)	✓			NULL
date	VARCHAR(255)	✓			NULL
time	VARCHAR(255)				'null'
shortDate	VARCHAR(255)	✓			NULL
registrationDate	TIMESTAMP	✓			CURRENT_TIMESTAMP

Figura 3.8: *Descrição da tabela ticket*

Procedimentos Armazenados

- O procedimento *getTicketsByCustomerId* seleciona os bilhetes que o cliente passado por parametro possui.
  Caso não exista nenhum bilhete o procedimento returnará *null*.

```
CREATE PROCEDURE ‘getTicketsByCustomerId‘(in id int)
BEGIN

select t.idTicket,s.logo,s.company,t.description,t.ticketNumber,t.shortDate,
t.date,t.time, op.seat,op.quantity,op.price,op.customerName,
l.place,l.latitude,l.longitude,
s.contact,s.email,s.webSite,t.registrationDate
from ticket t
inner join seller s on t.idSeller = s.idSeller
inner join customer c on c.idCustomer=t.idCustomer
inner join location l on l.idLocation=t.idLocation
inner join optional_field op on op.idOptionalField=t.idOptionalField
where c.phoneNumber=id order by t.date,t.time;

END$$
```

- O procedimento *getTicketUpdatedByCustomer* tem a função de procurar todos os novos bilhetes adquiridos do cliente passado por parametro. É passado também por parametro o id do ultimo bilhete que o cliente adquiriu.
  Caso não exista nenhum novo bilhete o procedimento returnará *null*.

```
CREATE PROCEDURE `getTicketUpdatedByCustomer`(in id int, in last int)
BEGIN

DECLARE maximo int;
SELECT max(idTicket) INTO maximo FROM ticket t inner join customer c
on c.idCustomer=t.idCustomer where t.idTicket=last;
```

No inicio do procedimento é declarada uma variável do tipo int e armazenado nela o ultimo bilhete adquirido pelo cliente passado por parametro.
Logo depois é realizada a procura de todos os novos bilhetes.

```
select t.idTicket,s.logo,s.company,t.description,t.ticketNumber,
t.shortDate,t.date,t.time,
op.seat,op.quantity,op.price,op.customerName,
l.place,l.latitude,l.longitude,
s.contact,s.email,s.webSite,t.registrationDate
from ticket t
inner join seller s on t.idSeller = s.idSeller
inner join customer c on c.idCustomer=t.idCustomer
inner join location l on l.idLocation=t.idLocation
inner join optional_field op on op.idOptionalField=t.idOptionalField
where c.phoneNumber=id and t.idTicket between last+1 and maximo order by
t.date,t.time;

END$$
```

- O procedimento *insertCustomer* tem como fim inserir na tabela *customer* um novo cliente.

```
CREATE PROCEDURE ‘insertCustomer‘(in phoneNumber varchar(255))
BEGIN

insert into customer values(null,phoneNumber,CURRENT_TIMESTAMP);

END$$
```

- O procedimento *insertLocation* tem como fim inserir na tabela *location* um nova localização.

```
CREATE PROCEDURE ‘insertLocation‘(in place varchar(255),
latitude varchar(255),longitude varchar(255))
BEGIN

insert into location values(null,place,latitude,longitude);

END$$
```

- O procedimento *insertOptionalField* tem como fim inserir na tabela *optional field* novos campos opcionais.

```
CREATE PROCEDURE ‘insertOptionalField‘(
in quantity integer, in price varchar(255),
in seat varchar(255),in customerName varchar(255))
BEGIN

insert into optional_field values(null,quantity,price,seat,customerName);

END$$
```

- O procedimento *insertSeller* tem como fim inserir na tabela *seller* um novo vendedor.

```
CREATE PROCEDURE ‘insertSeller‘(in company varchar(255),
in logo varchar(255),in contact varchar(255),
in email varchar(255),in webSite varchar(255))
BEGIN

insert into seller values(null,company,logo,contact,email,
webSite,CURRENT_TIMESTAMP);

END$$
```

- O procedimento *selectSingleCustomer* é responsável por procurar o cliente passado por parametro.

```
CREATE PROCEDURE ‘selectSingleCustomer‘(in number varchar(255))
BEGIN

select phoneNumber from customer where phoneNumber = number ;

END$$
```

- O procedimento *validateCustomer* procura o cliente passado por parametro. Caso o cliente procurado não exista ele irá inserir o cliente procurado na tabela *customer*, invocando o procedimento armazenado *insertCustomer*, caso contrário o procedimento returnará o cliente procurado.

```
CREATE PROCEDURE ‘validateCustomer‘(in customerPhone varchar(255))
BEGIN

SELECT phoneNumber INTO @customerPhoneNumber FROM customer
where phoneNumber=customerPhone;

if ISNULL(@customerPhoneNumber) then

  call insertCustomer(customerPhone);

  call selectSingleCustomer(customerPhone);

else

  select @customerPhoneNumber as phoneNumber;

end if;

END$$
```

Transações

- O procedimento *deleteTicket* é responsável por apagar um dado bilhete da base de dados.

```
CREATE PROCEDURE ‘deleteTicket‘()
BEGIN

DECLARE i INT Default 0 ;
DECLARE ticketcount INTEGER;
DECLARE idO INTEGER;
DECLARE idL INTEGER;
DECLARE idT INTEGER;
DECLARE error SMALLINT DEFAULT 0;
DECLARE CONTINUE HANDLER FOR SQLEXCEPTION SET error = 1;

START TRANSACTION;
```

O primeiro passo do procedimento é contabilizar e guardar na variável *ticketCount* quantos bilhetes existem na base de dados com a data expirada, após isso será iniciado um ciclo com *ticketCount* iterações.

```
SET ticketCount = (SELECT count(*) FROM ticket t
where date<CURRENT_DATE );
select ticketCount,i;
IF error = 1 THEN
        ROLLBACK;
end if;
WHILE i < ticketCount  DO
```

Por cada iteração, ele irá eliminar das tabelas *location* e *optional field*, os registos correspondentes ao bilhete, e somente após estas duas remoções o bilhete será removido na totalidade.

```
SET idL = (SELECT idLocation FROM ticket t
```

```
where date<CURRENT_DATE limit 1 );
IF error = 1 THEN
         ROLLBACK;
end if;
SET idO = (SELECT idOptionalField FROM ticket t
where date<CURRENT_DATE limit 1 );
IF error = 1 THEN
         ROLLBACK;
end if;
SET idT = (SELECT idTicket FROM ticket t
where date<CURRENT_DATE limit 1 );
IF error = 1 THEN
         ROLLBACK;
end if;
delete from ticket where idTicket=idT;
IF error = 1 THEN
         ROLLBACK;
end if;
delete from location where idLocation=idL;
IF error = 1 THEN
         ROLLBACK;
end if;
delete from optional_field where idOptionalField=idO;
IF error = 1 THEN
         ROLLBACK;
end if;
set i=i+1;
end while;
COMMIT;

END$$
```

Caso algum passo do procedimento não seja comprido, o bilhete será removido visto tratar-se de uma transação.

- O procedimento *insertTicket* insere um novo bilhete da base de dados. O procedimento irá adicionar primeiramente uma nova localização e novos campos opcionais, só após essas duas ações, o bilhete será inserido na base de dados. Visto tratar-se de uma transação, o bilhete não será inserido caso algum passo do procedimento não seja cumprido.

```
CREATE PROCEDURE ‘insertTicket‘(
in place varchar(255),latitude varchar(255),
longitude varchar(255), in quantity integer,
in price varchar(255),in seat varchar(255),
in customerName varchar(255), in idSeller int, in idCustomer
int,in ticketNumber varchar(255),in description varchar(255),
in date varchar(255), in time varchar(255), in shortDate varchar(255))
BEGIN

DECLARE lastLocation INTEGER;
DECLARE lastOptionalField INTEGER;
DECLARE error SMALLINT DEFAULT 0;
DECLARE CONTINUE HANDLER FOR SQLEXCEPTION SET error = 1;

START TRANSACTION;

call insertLocation(place,latitude,longitude);
IF error = 1 THEN
        ROLLBACK;
end if;
     SET lastLocation = (SELECT DISTINCT MAX(idLocation) FROM location
IF error = 1 THEN
          ROLLBACK;
end if;
call insertOptionalField(quantity,price,seat,customerName);
IF error = 1 THEN
   ROLLBACK;
end if;
     SET lastOptionalField = (SELECT DISTINCT MAX(idOptionalField)
     FROM optional_field );
```

```
IF error = 1 THEN
          ROLLBACK;
end if;
insert into ticket values(null,idSeller,idCustomer,lastLocation,
lastOptionalField,ticketNumber,description,date,time,
shortDate,CURRENT_TIMESTAMP);
IF error = 1 THEN
          ROLLBACK;
end if;
COMMIT;
END$$
```

Eventos

- O evento *deleteAllExpiredTickets* é responsável por todos os dias apagar todos os bilhetes com a data expirada.
  Todos os dias na hora da criação do evento, será chamado o procedimento *deleteTicket*.

```
CREATE EVENT deleteAllExpiredTickets
ON SCHEDULE EVERY 1 DAY
DO call deleteTicket()
```

3.5.3 Servidor

O servidor foi desenvolvido para dar suporte à aplicação móvel, e servir de intermediário entre a aplicação móvel e a base de dados.
No desenvolvimento do servidor foi utilizada a DAO[8] como pattern, para permitir separar as regras de negócio das regras de acesso à base de dados.

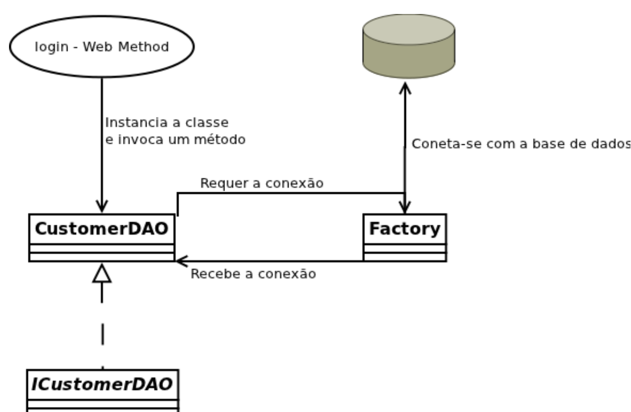


Figura 3.9: *Web método login*

Como é visível no diagrama acima, sempre que é requisitado um web método, ele não irá aceder à base de dados diretamente.
A class DAO, que implementa o interface DAO, irá instanciar a class *Factory* e esta por sua vez retornará uma conexão, com esta conexão a class DAO, poderá aceder à base de dados e assim servir o web método.
A class *Factory*, é completamente independente do SGBD[9].

[8] Data Access Object
[9] Sistema Gestor de Base de Dados

Por razões de segurança e reaproveitamento foram guardados num ficheiro *.properties* todos os dados para a realizar a conexão com a base de dados.
Sempre que for requerida uma nova conexão, o ficheiro será lido para retirar os dados para realizar a conexão.

Exemplo da leitura dos dados para a conexão.

```
String serverUsername;
String serverPassword;

Properties propertiesFile = new Properties();

try {
   propertiesFile.load(new FileInputStream("./serverConfig.properties"));

} catch (IOException ex) {
System.out.println("File not found");
}

serverUsername = propertiesFile.getProperty("username");
serverPassword = propertiesFile.getProperty("password");
```

Desenvolver um serviço web é necessário respeitar algumas normas para a criação de web métodos e parametros web.

Para a declaração do web método é necessário primeiro de tudo escrever *@WebMethod(operationName = "nome")*, e para declarar parametros web é necessário escrever *@WebParam(name = "nome")*
Exemplo de uma declaração de um web método.

```
@WebMethod(operationName = "login")
    public int login(@WebParam(name = "username") String username,
            @WebParam(name = "password") String password,
            @WebParam(name = "phoneNumber") int phoneNumber) {
```

Foram desenvolvidos quatro web métodos, para servir a aplicação móvel.

- O web método *login* recebe três parametros, os primeiros dois para autenticar o pedido, e o terceiro para interagir com a base de dados.

public abstract int webservicemobileticket.WebServiceMobileTicket.login(java.lang.String,java.lang.String,int)
login (, ,)

Figura 3.10: *Web método login*

```
Customer c = new Customer();
c.setPhoneNumber(phoneNumber);
CustomerDAO cDAO = new CustomerDAO();
return cDAO.checkCustomer(c);
```

É instânciado um objeto do tipo *Customer* e é atribuido o parametro *phoneNumber* ao atributo do objeto.

De seguida instânciado um objeto do tipo *CustomerDAO* e é invocado o seu método *checkCustomer* e é retornado o seu resultado.

Ao instanciar a classe *CustomerDAO* é invocado automaticamente o seu cosntrutor, e automaticamente ele coneta-se com a base de dados, através da instância da class *Factory*.

```
Connection conn;
public CustomerDAO() {
        try {
            Factory f = new Factory();
            this.conn = f.getConn();
        } catch (SQLException ex) {
        }
}
```

Ao invocar o método *checkCustomer*, este íra chamar o procedimento *validateCustomer* e devolver o seu resultado.

```
@Override
   public int checkCustomer(Customer c) {
        ResultSet rs;
        CallableStatement calstat;
        try {
            calstat = conn.prepareCall("{
     call validateCustomer('" + c.getPhoneNumber() + "')}");
            calstat.execute();
            rs = calstat.getResultSet();
            while (rs.next()) {
                c.setPhoneNumber(rs.getInt("phoneNumber"));
                System.out.println("RS:" + rs.getString("phoneNumber"))
            }
            rs.close();
            calstat.close();
            conn.close();
        } catch (SQLException ex) {
            return 0;
        } catch (Exception e) {
            return 0;
        }
        return c.getPhoneNumber();
   }
```

- O web método *getTicketsList* recebe três parametros, os primeiros dois para autenticar o pedido, e o terceiro para interagir com a base de dados.

public abstract java.util.List webservicemobileticket.WebServiceMobileTicket.getTicketsList(java.lang.String,java.lang.String,java.lang.String)
getTicketsList (, ,)

Figura 3.11: *Web método getTicketsList*

```
TicketDAO t = new TicketDAO();
List<Ticket> list = t.getTicketList(phoneNumber);
return list;
```

Neste método é instânciado um objeto do tipo *TicketDAO* e é invocado o seu método *getTicketList* e é retornado uma lista do tipo *ticket*.

Ao instanciar a classe *TicketDAO* é invocado automaticamente o seu cosntrutor, e automaticamente ele coneta-se com a base de dados, através da instância da class *Factory*.

Ao invocar o método *getTicketList*, este íra chamar o procedimento *getTicketsByCustomerId* e devolver uma lista do tipo *ticket*, com todos os registos obtidos no procedimento armazenado.

```
 @Override
    public List<Ticket> getTicketList(String phoneNumber) {
        ResultSet rs;
        CallableStatement calstat;
        List<Ticket> list = new ArrayList<Ticket>();
        try {
            calstat = conn.prepareCall("{
      call getTicketsByCustomerId(" + phoneNumber + ")}");
            calstat.execute();
            rs = calstat.getResultSet();
            while (rs.next()) {
                Ticket ticket = new Ticket();
                ticket.setIdTicket(rs.getInt("idTicket"));
```

```
                ticket.setSellerCompany(rs.getString("company"));
                ticket.setSellerLogo(rs.getString("logo"));
                ticket.setDescription(rs.getString("description"));
                ticket.setShortDate(rs.getString("shortDate"));
                ticket.setDate(rs.getString("date"));
                ticket.setTime(rs.getString("time"));
                ticket.setPlace(rs.getString("place"));
                ticket.setContact(rs.getString("contact"));
                ticket.setCustomerName(rs.getString("customerName"));
                ticket.setEmail(rs.getString("email"));
                ticket.setPrice(rs.getString("price"));
                ticket.setQuantity(rs.getInt("quantity"));
                ticket.setSeat(rs.getString("seat"));
                ticket.setTicketNumber(rs.getString("ticketNumber"));
                ticket.setWebSite(rs.getString("webSite"));
                ticket.setLatitude(rs.getDouble("latitude"));
                ticket.setLongitude(rs.getDouble("longitude"));
                ticket.setPurchaseDate(rs.getString("registrationDate")
                list.add(ticket);
            }
            rs.close();
            calstat.close();
            conn.close();
        } catch (SQLException ex) {
        }
        return list;
    }
```

- O web método *getTicketUpdatedByCustomer* recebe quatro parametros, os primeiros dois para autenticar o pedido, o terceiro para identificar o utilizador e o último para indicar qual o ultimo bilhete adquirido pelo cliente.

public abstract java.util.List webservicemobileticket.WebServiceMobileTicket.getUpdatedTicketList(java.lang.String,java.lang.String,java.lang.String,int)
getUpdatedTicketList (, , ,)

Figura 3.12: *Web método getTicketUpdatedByCustomer*

```
TicketDAO t = new TicketDAO();
List<Ticket> list = t.getUpdatedTicketList(phoneNumber, last);
return list;
```

Neste método é instânciado um objeto do tipo *TicketDAO* e é invocado o seu método *getUpdatedTicketList* e é retornado uma lista do tipo *ticket*.

Ao instanciar a classe *TicketDAO* é invocado automaticamente o seu cosntrutor, que automaticamente coneta-se com a base de dados, através da instância da class *Factory*.

Ao invocar o método *getUpdatedTicketList*, este íra chamar o procedimento *getTicketUpdatedByCustomer* e devolver uma lista do tipo *ticket*, com todos os registos obtidos no procedimento armazenado.

```
 @Override
    public List<Ticket> getUpdatedTicketList(String phoneNumber, int last) {
        ResultSet rs;
        CallableStatement calstat;
        List<Ticket> list = new ArrayList<Ticket>();
        try {
            calstat = conn.prepareCall("{
    call getTicketUpdatedByCustomer(" + phoneNumber +
    "," + last + ")}");
            calstat.execute();
            rs = calstat.getResultSet();
```

```
        while (rs.next()) {
            Ticket ticket = new Ticket();
            ticket.setIdTicket(rs.getInt("idTicket"));
            ticket.setSellerCompany(rs.getString("company"));
            ticket.setSellerLogo(rs.getString("logo"));
            ticket.setDescription(rs.getString("description"));
            ticket.setShortDate(rs.getString("shortDate"));
            ticket.setDate(rs.getString("date"));
            ticket.setTime(rs.getString("time"));
            ticket.setPlace(rs.getString("place"));
            ticket.setContact(rs.getString("contact"));
            ticket.setCustomerName(rs.getString("customerName"));
            ticket.setEmail(rs.getString("email"));
            ticket.setPrice(rs.getString("price"));
            ticket.setQuantity(rs.getInt("quantity"));
            ticket.setSeat(rs.getString("seat"));
            ticket.setTicketNumber(rs.getString("ticketNumber"));
            ticket.setWebSite(rs.getString("webSite"));
            ticket.setLatitude(rs.getDouble("latitude"));
            ticket.setLongitude(rs.getDouble("longitude"));
            ticket.setPurchaseDate(rs.getString("registrationDate")
            list.add(ticket);
        }
        rs.close();
        calstat.close();
        conn.close();
    } catch (SQLException ex) {
    }
    return list;
}
```

- O web método *sendSMS* recebe cinco parametros, os primeiros dois para autenticar o pedido, o terceiro para especificar o número de origem, o quarto para especificar o número de destino e o último para especificar o texto a ser enviado.

public abstract java.lang.String webservicemobileticket.WebServiceMobileTicket.sendSMS(java.lang.String,java.lang.String,java.lang.String,java.lang.String,java.lang.String) throws webservicemobileticket.MalformedURLException_Exception

sendSMS (, , , ,)

Figura 3.13: *Web método sendMMS*

O método *sendSMS* é responsável por enviar uma sms para um número destino especificado. Foi utilizada a API[10] da empresa lusosms. O objetivo deste método é para ser usado para utilizadores, que não possuiem nenhum dispositivo móvel Android, assim é enviado o bilhte mas via sms para o set telemóvel.

```
String inputLine = "";
        try {
            String url = "http://www.lusosms.com/enviar_sms_get.php?username="
                    + "&password=" + password
                    + "&origem=" + callingFrom
                    + "&destino=" + callingTo
                    + "&mensagem=" + message.replace(" ", "+");
            URL sms = new URL(url);
            URLConnection conn = sms.openConnection();
            BufferedReader in = new BufferedReader(new InputStreamReader(conn.g
            while ((inputLine = in.readLine()) != null) {
                System.out.println(inputLine);
            }
            in.close();
        } catch (Exception e) {
        }
        return inputLine;
```

[10] Application Programming Interface

3.5.4 Aplicação Móvel

Para começar a desenvolver a aplicação móvel foi antes necessário estudar o sistema operativo Android, como ele funciona, as restrições impostas pela Google, e finalmente idealizar a aplicação.

Projeto Android

Os projetos Android são compilados para um ficheiro APK[11] que futuramente será instalado no dispositivo móvel. Os ficheiros APK contêm o código fonte da aplicação, bem como os ficheiros de recurso. Alguns desses ficheiros de recurso são gerados por defeito enquanto outros devem ser criados caso seja necessário.

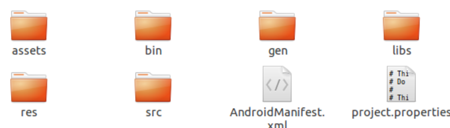


Figura 3.14: *Estrutura de um projeto Android*

Na imagem acima são visíveis, as diretorias e os ficheiros para um projeto Android.

- O diretório *src* contém toda a raiz do ficheiro Activity, bem como todos os outros ficheiros relativos ao código fonte(.java)
- O diretório *bin* contém o output da compilação, é aqui que será armazenado o ficheiro APK.
- O diretório *gen* contém os ficheiros java gerados pelo ADT[12], que são os ficheiros R.java, as classes e os interfaces

[11] Android Package
[12] Android Development Tools

- O diretório *assets* está inicialmente vazio. Neste diretório é possivel armazenar ficheiros de apoio à aplicação como imagens, fonts, etc. Todos os ficheiros armazenados no diretório serão compilados para o ficheiro APK. Este diretório é como um apêndice da aplicação, é possivel navegar nele e ler os ficheiros que este contém, mas após a compilação não é mais possivel modificar o seu conteúdo.
- O diretório *res* contém os recursos da aplicação, como os layouts, menus e string values.
- O diretório *libs* contém as bibliotecas privadas.
- O ficheiro *AndroidManifest.xml* é o ficheiro de controlo que descreve a natureza da aplicação e cada um dos seus componetes, por exemplo:
  - Quais Permissões que a aplicação tem acesso.
  - Quais Bibliotecas externas necessárias.
  - Quais as características que o dispositivo necessita.
  - Quais os níveis da API que são suportados
- O ficheiro *project.properties* contém as propriedades do projeto.

Conceção da Aplicação

No inicio do desenvolvimento da aplicação foi decidido criar um ficheiro xml na diretoria *res*, atualizável, ou seja, criava-se um ficheiro no momento da primeira inicialização da aplicação com todos os bilhetes descarregados e após isso só se conectava ao servidor quando o cliente deseja-se atualizar a lista de bilhetes, isso iria evitar um uso frequente da internet e dos serviços do servidor, visto que os bilhetes eram lidos diretamente do ficheiro xml atualizável. Mas tal ideia não foi possível conceber, porque não é possível modificar nenhum ficheiro dos dos diretórios *res* e *assets* após a compilação da aplicação.
Após essa descoberta, ficou então decidido fazer uma conexão ao servidor sempre que a aplicação é iniciada e não apenas na primeira vez para descarregar os bilhetes, após isso é feito somente conexões ao servidor quando o cliente desejar atualizar a lista de bilhetes, isso irá tornar a aplicação mais dependente da internet e possivelmente com menos performance devido ás frequentes conexões ao servidor.

Para a realização do projeto foi decidido criar uma classe Activity para cada layout existente, visto que a classe Activity define o ecrã que irá interagir com o utilizador. Ficou establecido também criar uma classe de auxílio nominada de *UtilityClass*, tendo esta todas as funções para auxiliar o desenvolvimento da aplicação.

Ao iniciar a aplicação é iniciado um splash screen.

Figura 3.15: *MobileTicket splash screen*

Após a conclusão do splash screen é iniciada uma nova activity.

Esta Activity é responsável por montar o layout de visualização dos bilhetes.
No início da Activity é feita uma tentativa de conexão ao servidor para fazer o download da lista de bilhetes do portador do telemóvel em uso.
Caso a conexão não seja sucedida, pelas mais diversas razões como alguma falha durante a conexão ao servidor, ou a internet encontre-se inacessível, não será feito o download da lista de bilhetes e o aparecerá um símbolo no canto superior direito do ecrã indicando que alguma falha ocorreu durante o processo de download da lista de bilhetes.

Figura 3.16: *MobileTicket splash screen*

```
if(!login()){

//Declaração dos objectos para o layout

ImageView imgNotification;

//*************Associar widgets ás variaveis****************

imgNotification = (ImageView) findViewById(R.id.imgNotification);

imgNotification.setVisibility(View.VISIBLE);
}
```

No teste acima exposto é testado o resultado do método *login*, que caso devolva falso, a imagem de notificação será apresentada no ecrã.

```
protected boolean login() {

//Verifica se existe acessoa à internet
if(!UtilityClass.isOnline(this)){

return false;

}else{

request=UtilityClass.getSoapObject(NAME_SPACE, "login");

request.addProperty("username", "mobileTicketServer03#");

request.addProperty("password", "u#55t69%$");

request.addProperty("phoneNumber", getMyPhoneNumber());

try {

response=UtilityClass.invokeMethod(URL, NAME_SPACE, "login",request);
```

```
phoneCustomer=Integer.parseInt(response.getProperty(0).toString());


}catch (Exception e) {

return false;

}

}

return true;
}
```

O método *login*, primeiramente verificará o estado da internet, caso haja internet é feita a conexão ao servidor enviando três parametros.
O parametro username e o parametro password têm como objetivo realizar a autenticação no servidor e o último parametro, o numero de telemóvel, informa o servidor da identidade do dispositivo móvel.

Após a passagem dos parametros é invocado o web método *login*.

Após a invocação do web método *login* com sucesso, será feito o download de todos os bilhetes existentes e será preenchido um *ArrayList* com todos os bilhetes recebidos, e posteriormente serão guardados numa *ListView*, para apresentação.

```
for(i=0;i<response.getPropertyCount();i++){

SoapObject aux = (SoapObject)response.getProperty(i);

Ticket t = new Ticket();
t.setIdTicket(Integer.parseInt(aux.getProperty("idTicket").toString()));
t.setSellerCompany((aux.getProperty("sellerCompany").toString()));
t.setSellerLogo(((aux.getProperty("sellerLogo").toString())));
t.setDescription(((aux.getProperty("description").toString())));
t.setShortDate(((aux.getProperty("shortDate").toString())));
t.setDate(((aux.getProperty("date").toString())));
t.setTime(((aux.getProperty("time").toString())));
t.setPlace(((aux.getProperty("place").toString())));
t.setContact(((aux.getProperty("contact").toString())));
t.setCustomerName(((aux.getProperty("customerName").toString())));
t.setEmail(((aux.getProperty("email").toString())));
t.setPrice(((aux.getProperty("price").toString())));
t.setQuantity(((Integer.parseInt(aux.getProperty("quantity").toString()))));
t.setSeat(((aux.getProperty("seat").toString())));
t.setTicketNumber(((aux.getProperty("ticketNumber").toString())));
t.setWebSite(((aux.getProperty("webSite").toString())));
t.setLatitude((Double.parseDouble(aux.getProperty("latitude").toString())));
t.setLongitude((Double.parseDouble(aux.getProperty("longitude").toString()))
t.setPurchaseDate(((aux.getProperty("purchaseDate").toString())));
ticketList.add(t);
}
```

Em pirmeiro lugar são contados quantos bilhetes foram recebidos, depois é inicializado um cilco *for*, alojando a informação do bilhete num objeto *ticket*, e adicionado a uma uma lista.

```
adapter = new TicketArrayAdapter(getApplicationContext(), R.layout.ticket_list_item

lv=(ListView) this.findViewById(R.id.lvTicket);

lv.setAdapter(adapter);
```

O trecho de código acima referido faz transporte da informação da Lista para o layout da textitListView.

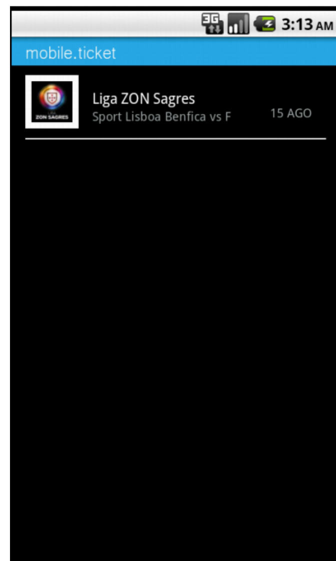


Figura 3.17: *MobileTicket lista de bilhetes*

Nesta activity é ainda possível fazer uso dos menus oferecidos.

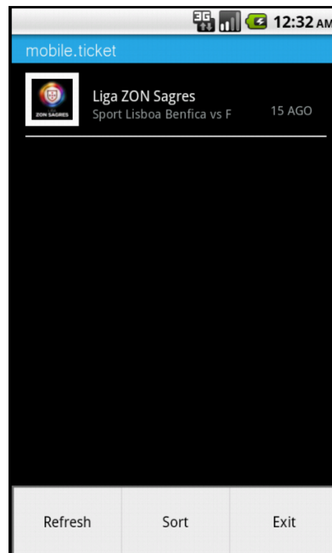


Figura 3.18: *MobileTicket lista de bilhetes*

```
@Override
public boolean onCreateOptionsMenu(Menu menu){

MenuInflater inflater = getMenuInflater();

inflater.inflate(R.menu.ticket_list_menu, menu);

return true;
}
```

Através do método *onOptionsItemSelected(MenuItem item)* é possível selecionar uma das opções do menu.

```
if(item.getItemId() == R.id.mnuRefresh){

getTicketsList("getUpdatedTicketList");

adapter.notifyDataSetChanged();

updated=1;

return true;

}
```

O menu *Refresh*, faz a atualização dos bilhetes, ou seja, faz uma pesquisa na base de dados para verificar se existe algum novo bilhete que não conste na lista de bilhetes.

O menu *Sort*, ordena a lista de bilhetes, é possivel ordenar a lista, pelo nome do bilhete, por data, ou por data de compra.

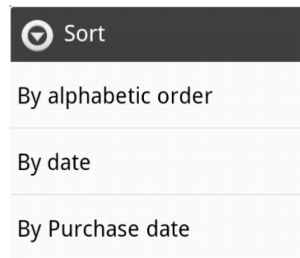


Figura 3.19: *MobileTicket sub menu ordenar*

Exemplo da ordenação por data.

```
if(item.getItemId() == R.id.mnuSortDate){

Comparator<Ticket> comperator = new Comparator<Ticket>() {

@Override
public int compare(Ticket t1, Ticket t2) {

return t1.getDate().compareToIgnoreCase(t2.getDate());

}
};

Collections.sort(ticketList, comperator);

adapter.notifyDataSetChanged();

return true;

}
```

Para realizar a ordenação é usada a Classe *Comparator* e é feito o override do método *compare*, passando-lhe dois objetos *ticket* para poder comparar o atributo *date*, após isso é feito uma atualização ao *adapter*.

O menu *exit*, termina eventualmente com a aplicação.

```
if(item.getItemId() == R.id.mnuExit){

updated=0;

this.finish();

return true;

}
```

Ao invocar o método *finish()*, na realidade, a aplicação não termina, esse método é equivalente a adormecer a aplicação, que irá para uma fila no sistema operativo, com todos os processos adormecidos.

Ao clicar em algum item da lista de bilhetes, é invocado o método *setOnItemClickListener*.

```
lv.setOnItemClickListener(new AdapterView.OnItemClickListener() {
@Override
public void onItemClick( AdapterView<?> parent, View item,int position, long

Ticket t = adapter.getItem( position );

Intent i = new Intent(MobileTicketActivity.this, ViewTicket.class);

i.setFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP);

i.putExtra("Ticket", t);

startActivity(i);

}
});
```

Ao invocar o método acima referido, será inicializada uma nova Activity. Nesta Activity é acrescentada informação, ou seja é iniciada com ela o objeto, que foi clicado na lista de bilhetes, para assim podermos usar a sua informação na nova Activity.

Ao ser iniciada a nova Activity é guardado numa variável do tipo *ticket*, o *itent* vindo da anterior Activity, após isso a Activity carrega o layout que mostra o conteúdo do bilhete clicado na lista de bilhetes

```
Intent i = getIntent();

Ticket ticket = (Ticket)i.getSerializableExtra("Ticket");
```

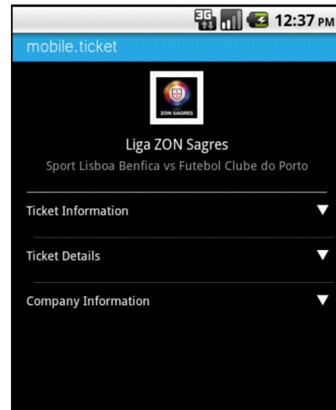


Figura 3.20: *MobileTicket informação do bilhete*

Este layout está dividido em quatro secções:

- A primeira parte do layout apresenta o logotipo da empresa, o nome da empresa que vendeu o bilhete e a descrição do bilhete.

  Em primeiro são declaradas duas variáveis do tipo *TextView* e de seguida são lhe atribuidas os valores presentes no objeto *ticket*.

```
try {

Bitmap bitmap = BitmapFactory.decodeStream(getApplicationContext().
getResources().getAssets().open("images/"+ticket.getSellerLogo()));

ivLogo.setImageBitmap(bitmap);
} catch (IOException e) {
e.printStackTrace();
}

tvCompany.setText(ticket.getSellerCompany());
tvDescriptionH.setText(ticket.getDescription());
```

- A segunda parte do layout apresenta toda a informação do bilhete.

Figura 3.21: *MobileTicket informação do bilhete*

- A terceira apresenta a informação relativamente à localização do local do evento do bilhete.

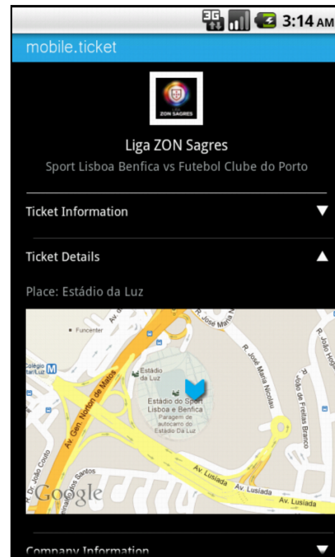


Figura 3.22: *MobileTicket informação do bilhete*

Esta parte do layout é um pouco diferente das restantes, nela está integrada o google maps, para mostrar a localização do evento do bilhete no mapa.
Para fazer a integração do google maps na aplicação é necessário primeiramente inicializar uma variável do tipo *GeoPoint* e outra do tipo *MapView*.

```
private GeoPoint point;
private MapView myMap;
```

Após a inicialização, associa-se o layout *mymap*, à variável *myMap* e define-se algumas definições do mapa, como o zoom a posição.
Na variável declarada do tipo *GeoPoint*, é instânciado um novo objeto e passa-se as coordenadas, a latitude e a longitude.

```
myMap = (MapView) findViewById(R.id.mymap);
myMap.setBuiltInZoomControls(true);
```

```
MapController mc;
mc=myMap.getController();
Double lat = ticket.getLatitude();
Double lng = ticket.getLongitude();
point = new GeoPoint(lat.intValue(),lng.intValue());
mc.setCenter(point);
mc.setZoom(17);
```

- A quarta apresenta a informação do vendedor.

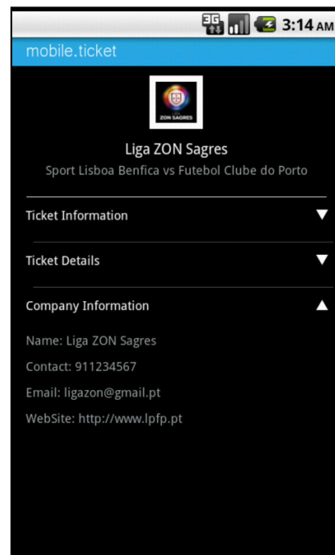
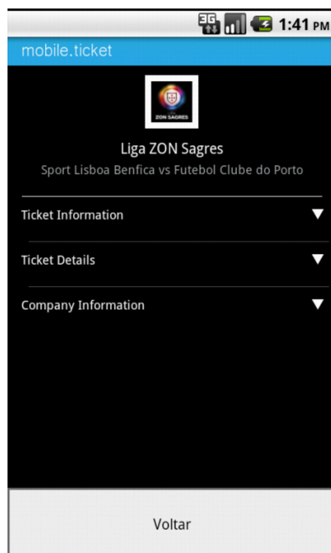


Figura 3.23: *MobileTicket informação do bilhete*

Nesta activity é ainda possível fazer uso dos menu oferecido, para voltar para a lista de bilhetes.



Capítulo 4

CONCLUSÕES E TRABALHO FUTURO

Os objetivos definidos inicialmente para este projeto , foram atingidos com sucesso. Tal como foi idealizado neste projeto, foi implementado um sistema ticketing para a plataforma Android, e assim foi possível melhorar todos os conhecimentos já adquiridos nas áreas de base de dados, programação, programação orientada a objetos, estruturas de dados e engenharia de software e foram ainda adquiridos novos conhecimentos na tecnologia Android.

O desenvolvimento deste projeto tornou-se assim bastante enriquecedor, possibilitando a aprendizagem de novas ferramentas e tecnologias e também o aperfeiçoamento de tecnologias, ferramentas e conceitos, lecionados ao longo do período académico

4.1 Trabalho futuro

Como trabalho futuro predende-se estender mais as funcionalidades do sistema, assim faria com que o sistema ficasse mais completo. Seria interessante implementar uma arquitetura de módulos no sistema, ou seja, haveria um módulo central responsável por ligar todos os módulos, e cada módulo seria desenvolvido de forma completamente independente e abstrata. Após implementar a arquitetura por módulos no sistema, seria interessante desenvolver um módulo que possibilita-se a compra de bilhetes

através do dispositivo móvel, isso faria com que o sistema ficasse muito completo, e faria que o utilizador ficasse mais dependente do sistema mais tempo.

Bibliografia

- http://developer.android.com/resources/dashboard/platform-versions.html
- http://www.marketingimob.com/2012/01/o-crescimento-da-plataforma-android-e-o.html
- http://comlimao.com/2012/03/23/smartphones-crescem-mais-de-100-e-android-lidera-lancamentos/
- http://safarigeek.blogspot.com.br/2012/01/grafico-mostra-impressionante-ascensao.html
- http://www.kerodicas.com/destaques/artigo=53697/
- http://www.e-thesis.inf.br/index.php