INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA SUL-RIO-GRANDENSE - CAMPUS PASSO FUNDO
CURSO DE TECNOLOGIA EM SISTEMAS PARA INTERNET

FERNANDO ZONATTO

MY MONEY: UMA SOLUÇÃO PARA O CONTROLE FINANCEIRO PESSOAL EM PLATAFORMAS MÓVEIS

Jorge Luís Boeira Bavaresco

PASSO FUNDO
2015

FERNANDO ZONATTO

MY MONEY: UMA SOLUÇÃO PARA O CONTROLE FINANCEIRO PESSOAL EM PLATAFORMAS MÓVEIS

Monografia apresentada ao Curso de Tecnologia em Sistemas para Internet do Instituto Federal Sul-rio-grandense, Campus Passo Fundo, como requisito parcial para a obtenção do título de Tecnólogo em Sistemas para Internet.

Orientador: Jorge Luís Boeira Bavaresco

PASSO FUNDO

2015

FERNANDO ZONATTO

MY MONEY: UMA SOLUÇÃO PARA O CONTROLE FINANCEIRO PESSOAL EM PLATAFORMAS MÓVEIS

Trabalho de Conclusão de Curso aprovado em ____/____/____ como requisito parcial para a obtenção do título de Tecnólogo em Sistemas para Internet

Banca Examinadora:

_____

Jorge Luís Boeira Bavaresco

_____

José Antônio Oliveira de Figueiredo

_____

Josué Toebe

_____

Coordenação do Curso

PASSO FUNDO

2015

AGRADECIMENTOS

Primeiramente agradeço aos meus pais, Solange Almeida da Silva e José Carlos Zonatto, pelo apoio e pelas condições proporcionadas, que fizeram com que eu chegasse até aqui.

Agradeço ao professor Jorge Luís Boeira Bavaresco pelo acompanhamento prestado e pelos ensinamentos e experiências compartilhadas, que auxiliaram na conclusão deste projeto.

Agradeço aos meus colegas pela convivência que tivemos, passando por momentos de descontração, dificuldades, trocas de experiência e conhecimentos que nos engrandeceram e contribuiriam com nossas trajetórias durante o curso.

Por fim, agradeço demais professores e funcionários do IFSUL por proporcionarem as condições necessárias para o ensino e o aprendizado de qualidade que nos é ofertado nesta instituição.

“It's a long way to the top if you wanna rock 'n' roll”

Bon Scott

RESUMO

O aumento da utilização de aplicativos móveis nos dias de hoje, traz consigo a oportunidade de se pesquisar as mais variadas formas de desenvolvimento para as plataformas móveis. O presente estudo teve como objetivo a análise e o desenvolvimento de um aplicativo móvel que auxilie os seus usuários no controle de suas finanças. Para implementação deste aplicativo, foi criada uma arquitetura distribuída que possui dois componentes, sendo o primeiro um aplicativo móvel e o segundo um servidor web para persistência de dados remotamente. O aplicativo foi desenvolvido utilizando-se linguagens como HTML, CSS e JavaScript, e *frameworks* como o Foundation for Apps, AngularJS e PhoneGap. O servidor web, que funciona como um backup dos dados, foi desenvolvido com a plataforma Java EE, e com o auxílio da API JPA, de containers EJB e de um RESTful *web service*. O resultado deste estudo foi o desenvolvimento do aplicativo com as funcionalidades básicas de uma aplicação para controle financeiro e com a estrutura mencionada anteriormente.

Palavras-chave: Aplicativo, Servidor, Java, AngularJS, Web service.

ABSTRACT

The increased use of mobile applications today, brings with it the opportunity to research the several forms of development for mobile platforms. This study aims at the analysis and development of a mobile application that helps its users to control their finances. To implement this application, a distributed architecture that will have two components will be created, the first being a mobile application and the second a web server for remote data persistence. The application will be developed using languages such as HTML, CSS and JavaScript, and frameworks such as Foundation for Apps, Angularjs and PhoneGap. The web server, which will act as a backup of data will be developed with the Java EE platform, and with the help of the JPA API, EJB containers and a RESTful web service. The result of this study was the application development with the basic features of an application for financial control and the structure mentioned above.

Keywords: App, Server, Java, AngularJS, Web service.

LISTA DE FIGURAS

Figura 1: Versões do Android e seus apelidos 15
Figura 2: Arquitetura do iOS 16
Figura 3: Especificações de hardware do Windows Phone. 18
Figura 4: Arquitetura da plataforma Windows Phone. 19
Figura 5: Encapsulamento de uma aplicação PhoneGap. 21
Figura 6: Arquitetura do PhoneGap 22
Figura 8: Um simples RESTful web *service* de um livro. 26
Figura 9: Representação JSON de uma ordem de compras. 27
Figura 10: Sincronização de dados entre uma entidade e uma tabela. 30
Figura 12: Diagrama de Casos de uso 38
Figura 13: Diagrama de Atividades Efetuar Login. 49
Figura 14: Diagrama de Atividades de Sincronização de Dados. 50
Figura 15: Diagrama de Classes 52
Figura 16: Projeto do web service no NetBeans 54
Figura 17: Bibliotecas utilizadas no projeto 55
Figura 18: Banco de dados do servidor 56
Figura 19: Código do arquivo de persistência de dados (*persistence.xml*) 56
Figura 20: Código do arquivo *glassfish_resources.xml* 57
Figura 21: Resultado do comando *foundation-apps new MyMoney* no terminal 58
Figura 22: Estrutura do projeto do aplicativo 59
Figura 23: Resultado do comando *foundation-apps watch* 59
Figura 24: Aplicativo criado, sendo executado no navegador 60
Figura 25: Diagrama de componentes da arquitetura do sistema 61
Figura 26: Classes da camada de Modelo. 62
Figura 27: Classe Usuario 63
Figura 28: Classe OperacaoID 65
Figura 29: Classe Operacao 66
Figura 30: Tabela Operacao no banco de dados do servidor 67
Figura 31: Classes da camada DAO 67
Figura 32: Classe UsuarioDAO 68
Figura 33: Classe CORSFilter 70
Figura 34: Classes da camada Serviço 70

Figura 35: Classe UsuarioREST 71
Figura 36: Acesso ao serviço pelo navegador 72
Figura 37: Arquivo index.html do aplicativo 73
Figura 38: *View* do formulário de inserção de operações 74
Figura 39: Tela de criação de operações financeiras 75
Figura 40: Tela principal, *login* e criação de novo usuário 76
Figura 41: Arquivo routes.js 77
Figura 42: Definição dos controles da aplicação no arquivo *controller.js* 77
Figura 43: Controle Usuário 78
Figura 44: Arquivo *db.js* 80
Figura 45: Criação de uma *ObjectStore* 81
Figura 46: *ObjectStores* do aplicativo My Money 82
Figura 47: Função de sincronização de dados 83
Figura 48: Requisições de sincronia com o servidor 84

LISTA DE ABREVIATURAS E SIGLAS

API: Application Programming Interface.
BSD: Berkeley Software Distribution.
CSS: Cascading Style Sheets.
DDL: Data Definition Language.
DML: Data Manipulation Language.
GPS: Global Positioning System.
HTML: Hypertext Markup Language.
HTTP: Hypertext Transfer Protocol.
JPA: Java Persistence API.
JPG: Joint Photographic Experts Group.
JSON: JavaScript Object Notation.
JVM: Java Virtual Machine.
OHA: Open Handset Alliance.
ORM: Object-relational Mapping.
PDF: Portable Document Format.
REST: Representational State transfer.
SGBD: Sistema de Gerenciamento de Banco de Dados.
SOAP: Simple Object Access Protocol.
SQL: Structured Query Language.
URI: Uniform Resource Identifier.
URL: Uniform Resource Locator.
WWDC: Worldwide Developer Conference.
XML: eXtensible Markup Language.

SUMÁRIO

1 INTRODUÇÃO 11
1.1 OBJETIVOS 11
1.1.1 Objetivo Geral 11
1.1.2 Objetivos Específicos 12
2 REFERENCIAL TEÓRICO 13
2.1 PLATAFORMAS DE DISPOSITIVOS MÓVEIS 13
2.1.1 Android 13
2.1.2 iOS 15
2.1.3 Windows Phone 17
2.2 PHONEGAP 19
2.2.1 Funcionamento 21
2.3 AngularJS 23
2.4 JAVA 23
2.5 RESTful Web Services 24
2.5.1 JAX-RS 25
2.6 JSON 26
2.7 PERSISTÊNCIA DE DADOS 28
2.7.1 JPA 29
2.7.2 PostgreSQL 31
2.8 EJB 32
3 METODOLOGIA 34
3.1 ESTUDO DE CASO 34
3.1.1 Contexto Atual 34
3.1.2 Aplicativos Existentes 35
4 MODELAGEM DO SISTEMA 37
4.1 REQUISITOS FUNCIONAIS E NÃO FUNCIONAIS 37
4.1.1 Requisitos Funcionais 37
4.1.2 Requisitos Não Funcionais 37
4.2 DIAGRAMA DE CASOS DE USO 38
4.3 DESCRIÇÃO DOS CASOS DE USO 39
4.3.1 Caso de Uso Efetuar Login 39
4.3.2 Caso de Uso Criar Usuário 41

4.3.3 Caso de Uso Visualizar Saldo 42
4.3.4 Caso de Uso Manter Contas 43
4.3.5 Caso de Uso manter Categorias 45
4.3.6 Caso de Uso Manter Operações Financeiras 46
4.3.7 Caso de Uso Sincronizar Dados 47
4.4 DIAGRAMA DE ATIVIDADES 49
4.4.1 Diagrama de Atividades Efetuar *Login* 49
4.4.2 Diagrama de Atividades de Sincronização de Dados 50
4.5 DIAGRAMA DE CLASSES 51
5 DESENVOLVIMENTO 54
5.1 PREPARAÇÃO DO AMBIENTE DE DESENVOLVIMENTO 54
5.1.1 Servidor 54
5.1.2 Aplicativo Móvel 58
5.2 ARQUITETURA DO SISTEMA 60
5.3 O SERVIDOR WEB 62
5.3.1 A Camada Modelo 62
5.3.2 A Camada DAO 67
5.3.3 A Camada de Serviço 69
5.4 O APLICATIVO MÓVEL 72
5.4.1 As Views 73
5.4.2 Os Controladores 76
5.4.3 A persistência local de dados 79
5.4.4 A sincronização de dados com o servidor 82
CONSIDERAÇÕES FINAIS 85
REFERÊNCIAS 87

1 INTRODUÇÃO

A utilização de dispositivos móveis, como *smartphones* e *tablets,* tem aumentado cada vez mais nos dias de hoje. Estes dispositivos possuem aplicativos que são desenvolvidos com o intuito de facilitarem inúmeras tarefas executadas por seus usuários nas suas atividades diárias. Alguns destes aplicativos conseguem executar digitalmente e de forma automatizada algumas tarefas que antes eram executadas de forma manual, provendo um maior desempenho e uma maior praticidade na realização destas tarefas, além de garantir a confiabilidade e segurança das informações que podem estar sendo utilizadas nestas tarefas.

O controle financeiro de uma pessoa é parte importante de sua vida nos dias de hoje. O controle das contas por meio de papel e caneta ou planilhas eletrônicas já se tornou muito defasado. As pessoas necessitam realizar este controle de maneira rápida, fácil, e se possível, em tempo real.

A partir destas informações, o presente trabalho propõe o desenvolvimento de um aplicativo de controle financeiro pessoal que consiga atender estes requisitos da melhor maneira possível. O aplicativo será desenvolvido utilizando tecnologias que facilitem que ele se torne multiplataforma e possuirá um sistema de sincronização de dados com um servidor remoto, onde serão armazenadas todas as informações dos usuários que o utilizam.

Para isto foram estudadas as tecnologias que serão utilizadas em seu desenvolvimento e foi feita uma modelagem da aplicação, baseada em um estudo de caso realizado e nos requisitos definidos. Além disso são apresentadas informações sobre o desenvolvimento e por fim as considerações finais a respeito deste estudo.

1.1 OBJETIVOS

Nesta seção serão apresentados os objetivos deste trabalho de conclusão.

1.1.1 Objetivo Geral

Desenvolver um aplicativo multiplataforma para dispositivos móveis, que auxilie o usuário em seu controle financeiro.

1.1.2 Objetivos Específicos

- Modelar um aplicativo móvel de controle financeiro baseando-se em um estudo de caso.
- Desenvolver um aplicativo móvel multiplataforma para controle financeiro, utilizando conceitos de programação para dispositivos móveis e suas respectivas tecnologias.
- Fazer com que este aplicativo sincronize seus dados com uma base de dados remota.

2 REFERENCIAL TEÓRICO

Nesta seção será apresentado um embasamento teórico para o aplicativo a ser desenvolvido. Serão abordados assuntos sobre as principais plataformas de dispositivos móveis dos dias de hoje. Também serão abordados conceitos sobre o Phonegap e o AngularJS, *frameworks* utilizados para o desenvolvimento do aplicativo. Será tratado também sobre o Java e as tecnologias utilizadas no servidor, como RESTful *webservice* e JSON, e por fim serão abordados conceitos sobre as tecnologias de persistência de dados envolvidas no projeto.

2.1 PLATAFORMAS DE DISPOSITIVOS MÓVEIS

Nesta seção serão apresentadas as três principais plataformas para desenvolvimento de aplicativos móveis existentes nos dias de hoje. São elas: Android, iOS e Windows Phone.

2.1.1 Android

O Android surgiu como uma resposta do Google ao mercado de *smartphones.* Ele é uma “plataforma de desenvolvimento para aplicativos móveis, baseada em um sistema operacional Linux, com diversas aplicações já instaladas e, ainda, um ambiente de desenvolvimento bastante poderoso, ousado e flexível” (LECHETA, 2013, p.22).

Seu desenvolvimento teve início em 2003 pela empresa Android Inc que em 2005 foi comprada pelo Google, que hoje lidera o desenvolvimento da plataforma (MONTEIRO, 2012). Mas não é só o Google que está por trás do Android. Em 2007 foi criada a Open Handset Alliance (OHA), que é um grupo de grandes empresas dos ramos de *software*, *hardware* e telecomunicações. Fazem parte deste grupo empresas como a Motorola, LG, Samsumg, Sony e muitas outras (LECHETA, 2013).

A Open Handset Alliance nasceu com o objetivo de criar uma plataforma para dispositivos móveis que fosse de código aberto e livre e que também fosse moderna e flexível para o desenvolvimento de aplicações corporativas, sempre visando o a satisfação dos usuários com o produto final. Os fabricantes de celulares, os usuários

comuns e principalmente os desenvolvedores de aplicativos se beneficiaram com esta união (LECHETA, 2013).

O resultado desta união foi o lançamento da primeira versão do Android em 2007, ainda em fase beta. Uma plataforma que conta com um sistema operacional baseado em Linux, uma rica interface gráfica, GPS, aplicações já instaladas e um ambiente de desenvolvimento poderoso, inovador e flexível. Além de utilizar a poderosa linguagem Java e todos os seus recursos para o desenvolvimento das aplicações (LECHETA, 2013).

Por ele ser de código aberto e possuir uma licença bastante flexível, os fabricantes podem fazer as alterações que julgarem necessárias no código-fonte para customizarem seus produtos, e os usuários e desenvolvedores podem apontar e corrigir falhas existentes na plataforma (LECHETA, 2013).

O sistema operacional do Android foi baseado no *kernel* 2.6 do Linux e é o responsável pelo gerenciamento de processos, memória e *threads*, além da segurança de arquivos e pastas. Cada aplicativo iniciado dispara um novo processo no sistema operacional e vários processos podem ser executados simultaneamente, cabendo ao *kernel* o controle de memória entre os processos (LECHETA, 2013).

A segurança do Android é baseada na segurança do Linux. Sendo que cada aplicativo é executado em um processo e cada processo possui uma *thread* dedicada. Para cada aplicativo instalado no aparelho é criado um usuário no sistema operacional, de forma com que nenhum outro usuário seja capaz de acessar esta aplicação (LECHETA, 2013).

Pelo motivo de suas aplicações serem escritas em Java, o Android possui uma máquina virtual chamada de Dalvik. Ela é uma espécie de máquina virtual Java (JVM) otimizada para dispositivos móveis (LECHETA, 2013).

Uma curiosidade a respeito do Android, é que cada versão da plataforma recebe um apelido inspirado em um doce, como pode-se visualizar na Figura 1.

Figura 1: Versões do Android e seus apelidos

Fonte: http://oficinadanet.com.br

Hoje em dia, o Android é a plataforma mais utilizada em dispositivos móveis no mundo inteiro, sendo responsável por 78% dos smartphones vendidos no primeiro semestre de 2015 (International Data Corporation, 2015).

2.1.2 iOS

O iOS é o sistema operacional utilizado pelos dispositivos móveis da Apple.

> Inicialmente o iOS foi desenvolvido para o iPhone, mas logo se tornou também um sistema operacional voltado não apenas para esse dispositivo, mas sim para todos os dispositivos móveis que a Apple estava lançando, entre eles os iPods, iPads e mais recentemente a Apple TV (MILANI, 2012, p.14).

Desenvolvido pela própria Apple, o iOS tem como característica a restrição de sua execução somente ao hardware construído pela própria empresa. Sendo assim, somente os dispositivos fabricados pela Apple podem executar com sucesso o iOS (MILANI, 2012). Famosa por construir bons conjuntos de hardware e aliando-os a um sistema criado por eles mesmos, a Apple consegue garantir um bom desempenho e uma boa estabilidade em seus dispositivos móveis (LECHETA, 2012).

A primeira versão do iOS foi lançada junto ao primeiro iPhone em 2007, durante a WWDC (Apple World Wide Developer Conference), evento para desenvolvedores realizado pela Apple. No ano de 2008 foi lançado o kit para desenvolvimento de aplicativos (SDK) e a App Store, loja virtual que distribui as aplicações pelo mundo (LECHETA, 2012).

Desde seu primeiro ano, o iPhone impressionou o mundo e atraiu atenções de usuários, desenvolvedores e grandes empresas. Muito disso por meio da sua facilidade de uso, sua tela *multitouch* e a integração de seus recursos e aplicações. Mas a grande revolução no mercado se deu devido a criação da App Store, uma grande ideia de Steve Jobs que permitiu que os aplicativos fossem distribuídos e instalados facilmente nos aparelhos, e além disso surgiu com uma proposta de remuneração aos desenvolvedores das aplicações (LECHETA, 2012).

A linguagem de programação utilizada para o desenvolvimento de aplicativos para o iOS é a Objective-C. Ela foi criada no início dos anos 80 e é baseada no C padrão. Ela é uma linguagem dinâmica, que possibilita a reutilização do código, a transmissão de mensagens entre determinadas estruturas e a criação de classes em tempo de execução (LECHETA, 2012).

A arquitetura do iOS é baseada em quatro camadas, como pode-se visualizar na Figura 2:

Figura 2: Arquitetura do iOS

Fonte: MILANI, 2012.

A primeira camada é a qual o desenvolvedor interage e chama-se Cocoa Touch. O Cocoa Touh é um *framework* que foi criado para facilitar ao desenvolvedor, o acesso aos mais diversos recursos dos aparelhos, inclusive em outras camadas (MILANI, 2012). Ele possui uma boa integração “ao ambiente de desenvolvimento Xcode e facilita a utilização de recursos de multimídia, networking, persistência, animações e muitos outros, escondendo a complexidade desses assuntos do desenvolvedor” (LECHETA, 2012, p.22).

A segunda camada chama-se Media e é responsável pelo gerenciamento de áudio, vídeo e animações, além das tecnologias para criação de aplicações gráficas (MILANI, 2012).

A terceira camada, chamada de Core Services, disponibiliza ao programador os serviços de manipulação de arquivos e acesso ao SQLite, entre outros (MILANI, 2012).

Por fim, a camada chamada de Core OS é o núcleo do sistema operacional, sendo responsável pelo gerenciamento de *sockets*, certificados e energia, além de gerenciar a parte de segurança e de comunicação do sistema com o mundo externo (MILANI, 2012).

O ambiente de desenvolvimento oficial utilizado para a criação de aplicativos para o iOS é o Xcode, desenvolvido pela própria Apple também.

O iOS apresentou um *market share*[1] de 18,3% no primeiro semestre de 2015 (International Data Corporation, 2015). Sendo assim, foi a segunda plataforma para smartphones mais vendida nesse período.

2.1.3 Windows Phone

A plataforma Windows Phone como a conhecemos hoje em dia, surgiu em fevereiro de 2010, como nome de “Windows Phone 7 Series”. Ela foi criada pela Microsoft como uma resposta ao mercado de *smartphones* após o lançamento do iPhone pela Apple em 2007 e do Android pouco tempo depois (MÔNACO e DO CARMO, 2012).

Mesmo antes do lançamento do iPhone e da popularização dos *smartphones,* a Microsoft já desenvolvia um sistema operacional voltado para dispositivos móveis, era o chamado Windows Mobile. Ele teve sua primeira versão lançada no ano 2000 e atingiu seu auge na versão 6.0, lançada em 2007, e até chegou a ser um nome de destaque na época. Mas enfrentando problemas como o de aparelhos com telas e *hardwares* muito diferentes, o sistema nunca caiu no gosto dos desenvolvedores, que nunca tinham a certeza de que os aplicativos desenvolvidos funcionariam em todos os dispositivos desta plataforma (MÔNACO e DO CARMO, 2012).

Neste mesmo ano de 2007 a Apple então lança o iPhone, que ao aliar um sistema operacional com uma boa usabilidade à uma vasta gama de aplicativos que o complementam, consegue uma revolução no mercado de smartphones. No ano

---

[1] Grau de participação de uma empresa no mercado em termos das vendas de um determinado produto; fração do mercado controlada por ela.

seguinte o Google lança o Android, que mesmo após um início conturbado, vira um grande sucesso de vendas. Nessa época, a Microsoft que já estava pronta para lançar o Windows Mobile 7, decide descontinuá-lo, lançando-o como a versão 6.5 e concentrando todos seus esforços na revolução completa de seu sistema. Surge assim em 2010, o Windows Phone 7, nova aposta da Microsoft para o mercado de dispositivos móveis (MÔNACO e DO CARMO, 2012).

> O objetivo da Microsoft com o Windows Phone é oferecer a plataforma para os dispositivos, e não o hardware onde a plataforma será executada. Com isso é possível garantir uma maior competitividade de mercado através da existência de aparelhos de diversos fabricantes diferentes e com especificações diferentes. (MÔNACO e DO CARMO, 2012, p.6).

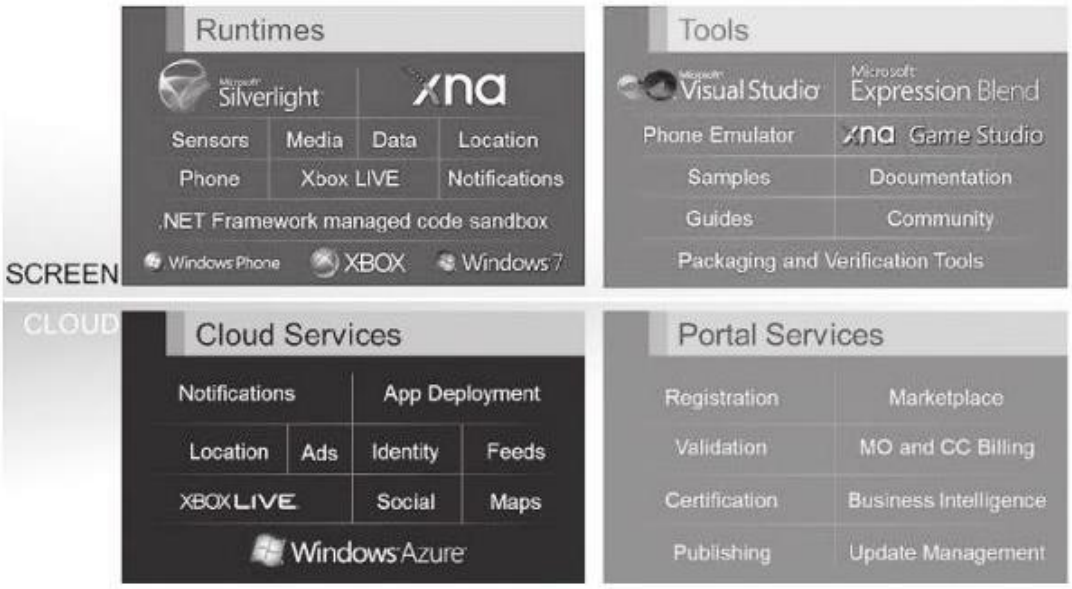
Para facilitar o trabalho dos desenvolvedores e garantir a maior satisfação do usuário, a Microsoft exige dos fabricantes do *hardware* algumas especificações mínimas para os aparelhos, conforme pode ser visto na Figura 3.

Figura 3: Especificações de hardware do Windows Phone.

Fonte: MÔNACO e DO CARMO, 2012.

A arquitetura do Windows Phone é construída com base em quatro componentes principais. Os quais são apresentados na Figura 4:

Figura 4: Arquitetura da plataforma Windows Phone.



Fonte: MÔNACO e DO CARMO, 2012.

Dentro do Runtime estão o Silverlight e o XNA Framework, onde todo o desenvolvimento é feito em código gerenciado e por meio de um conceito chamado de *sandbox,* onde o código da aplicação se comunica por meio de APIs específicas com as funções nativas do dispositivo (MÔNACO e DO CARMO, 2012).

As Tools são as ferramentas propriamente ditas para o desenvolvimento na plataforma. Tendo como destaque o Visual Studio voltado para a codificação e o Expression Blend com o foco em criação de animações e elementos visuais (MÔNACO e DO CARMO, 2012).

Os Cloud Services se referem a integração das aplicações com recursos e serviços na nuvem e o Portal Services é como se fosse uma central de desenvolvedores, onde são submetidas e certificadas as aplicações e onde os usuários podem localizá-las e baixa-las (MÔNACO e DO CARMO, 2012).

O Windows Phone é atualmente a terceira plataforma mais utilizada em smartphones atualmente. Possuindo um *market share* de 2,7% no primeiro semestre de 2015 (International Data Corporation, 2015).

2.2 PHONEGAP

O “PhoneGap é um *framework open source* para construção de aplicações nativas multiplataforma utilizando-se padrões de tecnologias *web* como *HyperText*

Markup Language (HTML), *Cascading Stylesheets* (CSS) e *JavaScript*" (WARGO, 2012, p.3, tradução nossa). Ele foi criado para simplificar a maneira de como são desenvolvidos aplicativos móveis e sua utilização cresce significativamente com o tempo.

Como open source, o PhoneGap está sob a licença Apache 2.0. Isto significa que ele é aberto e utiliza tecnologias abertas, além de ser totalmente grátis. Sendo assim ele pode ser usado para o desenvolvimento de qualquer tipo de aplicativo móvel, seja ele grátis, comercial ou open source. O PhoneGap foi criado e é mantido pela Adobe.

As aplicações desenvolvidas com o PhoneGap são consideradas híbridas. Sendo assim elas não são puramente *web* e nem nativas. Em geral as partes responsáveis pela interface gráfica do usuário, assim como a lógica da aplicação e a comunicação com um servidor são baseadas nas linguagens HTML, CSS e JavaScript. Já as partes em que ocorrem a comunicação e o controle do dispositivo são baseadas na linguagem nativa que a plataforma apresenta (GHATOL e PATEL, 2012).

O PhoneGap é capaz de criar uma ponte que faz com que o JavaScript se comunique com a linguagem nativa do dispositivo, fazendo com que a API do JavaScript tenha acesso a funcionalidades específicas do dispositivo, como câmera, GPS e outras mais (GHATOL e PATEL, 2012).

O PhoneGap é somente um *framework*, ele não inclui nenhuma IDE ou ambiente de desenvolvimento em seu pacote.

Atualmente ele suporta as seguintes plataformas de sistemas operacionais:

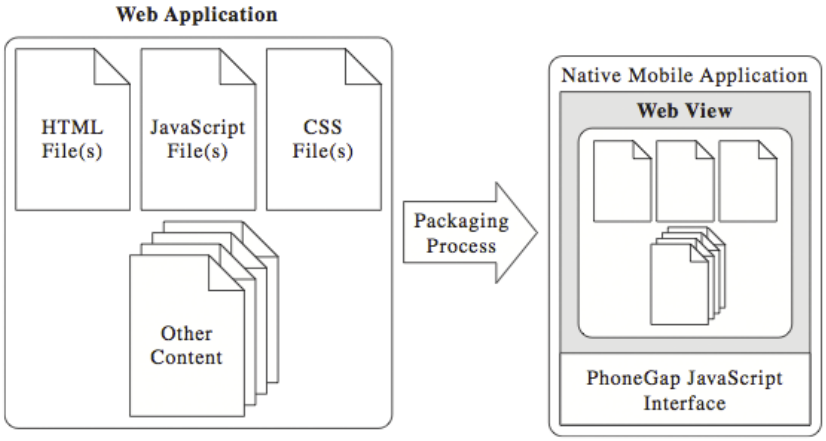
- Amazon FireOS
- Android
- BlackBerry
- Firefox OS
- iOS (iPhones e iPads)
- Ubuntu
- Windows Phone 8
- Tizen

Novas plataformas serão adicionadas assim que elas se tornarem populares entre os usuários e desenvolvedores de dispositivos móveis.

2.2.1 Funcionamento

O PhoneGap permite que um desenvolvedor construa um aplicativo para dispositivos móveis por meio da utilização de tecnologias web como HTML, CSS e JavaScript. Sendo assim, o desenvolvedor cria uma aplicação web para dispositivos móveis, e o PhoneGap, através de um conjunto de ferramentas que ele disponibiliza, encapsula esta aplicação dentro de uma aplicação nativa a plataforma móvel escolhida para o desenvolvimento (WARGO, 2012). Este processo pode ser visto na figura 5.

Figura 5: Encapsulamento de uma aplicação PhoneGap.



Fonte: WARGO, 2012

Dentro desta aplicação nativa, existirá uma Web View, que é um componente de uma aplicação nativa responsável pela renderização de conteúdo web dentro de uma tela do dispositivo (WARGO, 2012).

Esta Web View ocupará todo o espaço da tela do dispositivo e quando a aplicação for iniciada, ela se encarregará de carregar a página inicial da aplicação web para dentro da Web View, tornado possível a interação do usuário com a aplicação web previamente criada. Isso possibilita que o usuário, através de links ou códigos JavaScript, tenha acesso a outros conteúdos que foram encapsulados nesta aplicação ou consiga acesso à internet e obtenha conteúdo de servidores remotos (WARGO, 2012).

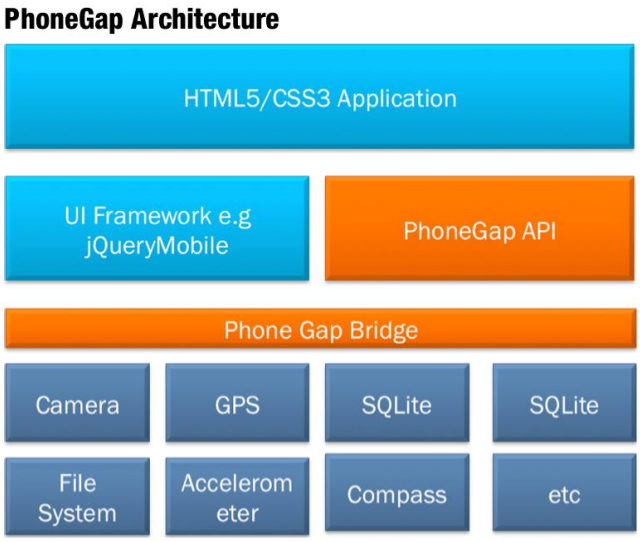
A aplicação web encapsulada dentro da aplicação nativa funciona exatamente como qualquer outra aplicação web. Seu layout é construído baseado em fontes, cores, espaçamentos e outas propriedades adicionadas ao HTML ou implementadas

por CSS. Ela pode acessar outras páginas HTML sejam elas locais ou de um servidor web. E é utilizada a linguagem JavaScript para a construção da lógica da aplicação, para a implementação de dinamismo no conteúdo das páginas HTML, para o envio ou recebimento de conteúdo de servidores externos e para outras funcionalidades (WARGO, 2012).

Aplicações web que rodam diretamente em navegadores não possuem a capacidade de acessar componentes nativos de um dispositivo como câmera ou acelerômetro. Grande parte dos aplicativos criados hoje em dia necessitam deste acesso. O PhoneGap entra em ação nesse momento, proporcionando que a aplicação web contida dentro da aplicação PhoneGap tenha acesso a estes componentes nativos que fogem do contexto web (WARGO, 2012).

Esta funcionalidade é possível graças a um conjunto de APIs e bibliotecas JavaScript do PhoneGap e de um componente nativo que ele possui que trabalha em segundo plano e faz a comunicação com o dispositivo (GHATOL e PATEL, 2012). Na figura 6 podemos visualizar a arquitetura do Phonegap.

Figura 6: Arquitetura do PhoneGap



Fonte: GHATOL e PATEL, 2012

De acordo com as características apresentadas é possível concluir que o PhoneGap é um excelente framework para se desenvolver um aplicativo móvel que pode se tornar multiplataforma.

2.3 AngularJS

O AngularJS é um *framework* estrutural para construção de aplicações web dinâmicas. Com ele é possível de se utilizar o HTML como linguagem de seu *template* e ele permite que a sintaxe do HTML seja estendida para demonstrar seus componentes de maneira clara e sucinta. Com sua ligação de dados e injeção de dependência é possível que se elimine grande parte do código que seria escrito. Todo seu processamento ocorre no navegador do cliente, permitindo que se use qualquer tecnologia no lado do servidor (ANGULARJS, 2015).

O Angular consegue simplificar o desenvolvimento de uma aplicação por conta de seu grande nível de abstração. Mas esta abstração tem um custo, o Angular não é recomendado para o desenvolvimento de qualquer aplicação web, ele foi pensado para a construção de aplicações CRUD. Mas como as aplicações CRUD representam grande parte das aplicações web, ele é um *framework* que pode ser bastante utilizado (ANGULARJS, 2015).

O framework utiliza o HTML como linguagem dos *templates* e o JavaScript como linguagem de controle destes *templates*.

2.4 JAVA

Em 1991 a Sun Microsystems financiou um projeto de pesquisa que resultou em uma linguagem baseada em C++. Seu criador, James Gosling, chamou-a de Oak, em homenagem a uma árvore de carvalho que podia ser vista da janela de seu escritório, só que mais tarde descobriu-se que já existia uma linguagem de programação com este nome. Foi então, que ao visitar uma cafeteria local, uma equipe da Sun sugeriu o nome Java, em referência ao nome da cidade de origem de um tipo de café importado, e este nome foi o escolhido (DEITEL e DEITEL, 2010).

O projeto de pesquisa passava por algumas dificuldades, mas em 1993, com a explosão de popularidade da Web, a Sun viu potencial em se utilizar o Java para adicionar conteúdo dinâmico, como interatividade e animações, às páginas web. E isto foi o que deu uma nova vida ao projeto (DEITEL e DEITEL, 2010).

Em 1995 o Java foi anunciado formalmente pela Sun em uma conferência do setor. Inicialmente ele chamou a atenção da comunidade de negócios por causa do enorme interesse na Web. Hoje em dia ele é utilizado para o desenvolvimento de

aplicativos corporativos, para o aprimoramento de servidores Web, para o desenvolvimento de aplicativos para o consumo popular entre muitos outros propósitos (DEITEL e DEITEL, 2010).

> Programas Java consistem em partes chamadas classes. As classes incluem partes chamadas métodos que realizam tarefas e retornam informações quando as tarefas são concluídas. Você pode criar cada parte necessária para formar seus programas Java. Entretanto, a maioria dos programadores tira proveito das ricas coleções de classes existentes nas bibliotecas de classe Java, que também são conhecidas como Java APIs (Application Programing Interfaces). (DEITEL e DEITEL, 2010, p. 7).

O Java consegue ser bastante versátil, por isto, consegue também ser multiplataforma. Pensando nisso e reconhecendo as diferentes necessidades dos desenvolvedores, a Sun criou diferentes especificações para o Java, baseando-se nos diferentes tipos de plataformas: desktop (Standard), corporativa (Enterprise) e móvel (Micro Edition) (SAMPAIO, 2011).

Com a versão Standard (Java SE) é possível construir-se a maioria das aplicações. Já que ela possui APIs que resolvem os mais diversos problemas de programação, desde redes até operações remotas. A versão Enterprise (Java EE) possui APIs que trabalham com containers (servidores) que hospedam aplicações corporativas. E a versão Micro (Java ME) fornece especificações e APIs para a criação de aplicações para plataformas embarcadas ou móveis (SAMPAIO, 2011).

2.5 RESTful Web Services

Após o surgimento do conceito de Web 2.0, da popularização dos dispositivos móveis com suas aplicações que agregam dados e da revolução na maneira como desenvolvemos aplicações web com HTML5 e JavaScript, um novo tipo de web *service* ganhou popularidade: o RESTful web *service*. Levando em conta esses acontecimentos, grandes empresas como Google, eBay, Amazon e Yahoo!, deixaram de utilizar web *services* do tipo SOAP para passarem a utilizar os RESTful web *services* (GONCALVES, 2013).

Representational State Transfer (REST) é uma arquitetura baseada no protocolo HTTP e nas URIs. Ela tenta transferir o funcionamento básico da web em geral para dentro de um web *service*. Neste modelo, cada URL representa um objeto e é possível interagir com este objeto utilizando-se os métodos HTTP GET (para

recuperar um conteúdo), DELETE (para excluir um conteúdo), POST (para criar um conteúdo) ou PUT (para atualizar um conteúdo) (GONCALVES, 2013).

Enquanto os web *services* SOAP foram feitos para que se fosse possível utilizar vários tipos de protocolos de transferência, os web *services* RESTful foram criados especificamente para o protocolo HTTP, utilizando assim o máximo de recursos deste protocolo. A arquitetura REST trata cada pedaço de informação como um recurso e o endereça por meio de URIs, os links padrões da web. Seguindo esses princípios e possuindo interface e protocolo bem definidos, as aplicações RESTful tendem a ser simples, leves e de alta performance (GONCALVES, 2013).

Os recursos são o centro da arquitetura REST. Eles são o conteúdo com o qual o cliente deseja interagir ou referenciar, ou são alguma informação relevante para ser referenciada por um *hyperlink* (um produto, um resultado de uma busca, uma tabela com conteúdo...). Estes recursos podem estar armazenados em um banco de dados, em um arquivo ou algum outro lugar em que possam ser endereçados por uma URI. As URIs (Uniform Resource Identifier) são identificadores únicos constituídos por um nome e um endereço estruturado que indica onde que um recurso pode ser encontrado, ex.: http://www.apress.com/java (GONCALVES, 2013).

Quando um cliente acessa um recurso, ele nunca interage de maneira direta como mesmo. Ele, na verdade, interage com uma representação deste recurso que permanece no servidor. Esta representação pode estar em formato de texto, JSON, XML, PDF, JPG ou algum outro. Um mesmo recurso pode possuir mais de um tipo de representação (GONCALVES, 2013).

O protocolo HTTP funciona baseado nas requisições e respostas trocadas entre um cliente e um servidor. Sendo assim, na arquitetura REST, quando um cliente deseja acessar um recurso do servidor, ele envia uma requisição HTTP para o servidor por meio de umas URI, que por sua vez responde enviando uma representação do recurso acessado (GONCALVES, 2013).

2.5.1 JAX-RS

O JAX-RS é a API para construção de RESTful web *services* e Java. A primeira versão lançada em 2008 definia uma lista de APIs que auxiliavam na utilização da arquitetura REST, mas somente no lado do servidor. Junto com o Java

EE 7 foi lançado o JAX-RS 2.0 que também define as APIs do lado do cliente (GONCALVES, 2013).

Por meio dele é possível se criar um RESTful web *service* de maneira elegante e com apenas algumas anotações como é apresentado na Figura 8 (GONCALVES, 2013).

Figura 8: Um simples RESTful web *service* de um livro.

```
@Path("/book")
public class BookRestService {

    @GET
    @Produces("text/plain")
    public String getBookTitle() {
        return "H2G2";
    }
}
```

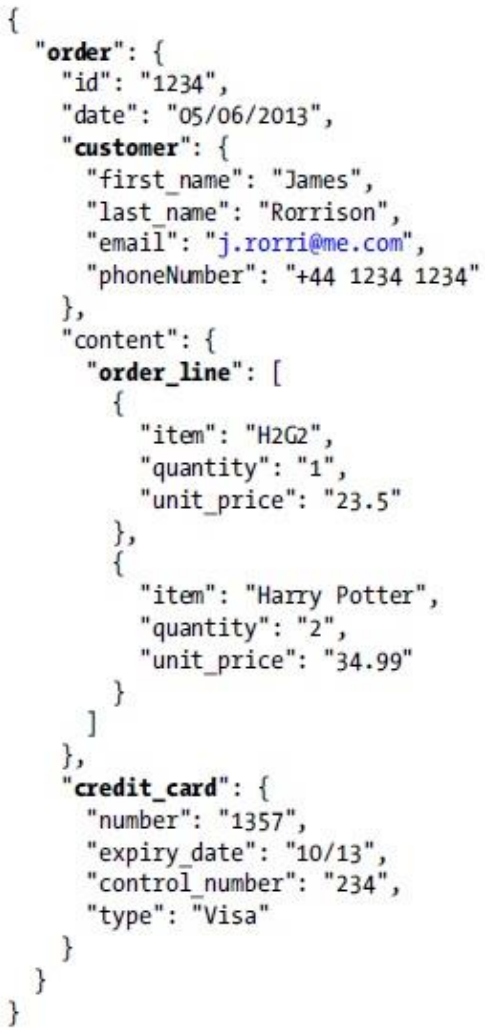
Fonte: GONCALVES, 2013.

BookRestService é uma classe Java com a anotação @Path, que indica que o recurso será hospedado na URI como caminho “/book”. O método getBookTitle() está marcado para processar requisições do tipo GET (por meio da anotação @GET) e retornar um texto (identificado pela anotação @Produces(“text/plain”)). Para acessar este recurso, o cliente necessitaria de um cliente HTTP, como um browser, e deveria acessar o link http://www.meuservidor/book. O retorno seria o conteúdo “H2G2” em formato de texto (GONCALVES, 2013).

2.6 JSON

JavaScript Object Notation (JSON) é um tipo de formatação leve utilizada para troca de dados. Ele é fácil de ser lido e escrito por seres humanos, e fácil de ser interpretado e gerado pelas máquinas. É muito utilizado para a serialização e transmissão de dados estruturados por meio de uma conexão de rede entre um servidor e uma aplicação web. Ele serve como uma alternativa ao XML e pode ser consumido diretamente pelo código JavaScript de uma aplicação web (GONCALVES, 2013).

Figura 9: Representação JSON de uma ordem de compras.



```
{
  "order": {
    "id": "1234",
    "date": "05/06/2013",
    "customer": {
      "first_name": "James",
      "last_name": "Rorrison",
      "email": "j.rorri@me.com",
      "phoneNumber": "+44 1234 1234"
    },
    "content": {
      "order_line": [
        {
          "item": "H2G2",
          "quantity": "1",
          "unit_price": "23.5"
        },
        {
          "item": "Harry Potter",
          "quantity": "2",
          "unit_price": "34.99"
        }
      ]
    },
    "credit_card": {
      "number": "1357",
      "expiry_date": "10/13",
      "control_number": "234",
      "type": "Visa"
    }
  }
}
```

Fonte: GONCALVES, 2013.

Os objetos JSON podem ser serializados dentro da própria estrutura do JSON, o que resulta em uma estrutura muito menos complexa do que a do XML. Quando o valor de uma variável é colocado entre chaves, isto indica que este valor é um objeto. Dentro dos objetos podem ser declaradas inúmeras propriedades. Para isso utiliza-se o par “nome”: “valor” separando-os por virgula. Para acessar uma propriedade é necessário fazer referência ao objeto e ao nome da propriedade (GONCALVES, 2013).

O JSON é uma notação independente e baseada em texto que utiliza uma série de convenções para representar estruturas de dados simples. Várias linguagens de programação já implementaram APIs para tratar documentos em JSON (GONCALVES, 2013).

O JSON pode representar quatro tipos primitivos de dados (números, *strings*, *booleans* e *null*) e dois tipos estruturados (*arrays* e objetos). Segue abaixo a lista de terminologias do JSON segundo Goncalves (2013):

- Números: Representados da mesma maneira que em Java;
- *Strings*: Sequência de caracteres Unicode colocadas entre aspas duplas;
- Valor: Pode estar em um destes formatos; *String*, número, *boolean*, objeto ou *array*;
- *Array*: É uma série de elementos ordenados. Os colchetes demarcam seu início e seu fim. Seus valores são separados por vírgula e podem ser objetos;
- Objetos: Possuem a mesma definição dos objetos do Java. As chaves demarcam seu início e seu fim. Os pares de nome e valor são separados por vírgula e podem representar atributos em Java.

2.7 PERSISTÊNCIA DE DADOS

A maioria das aplicações necessitam de dados persistentes. Se um sistema não preserva seus dados quando for desligado ele se torna pouco prático para o uso. Quando se fala em persistência de dados em um sistema de informação, normalmente trata-se do armazenamento dos dados em um banco de dados utilizando a linguagem SQL (BAUER e KING, 2007).

Bancos de dados relacionais são uma abordagem extremamente flexível e robusta para o gerenciamento de dados. Eles conseguem garantir e proteger a integridade dos dados armazenados e, além disso, ainda possuem outras características que os fazem ser muito utilizados hoje em dia (BAUER e KING, 2007).

Segundo BAUER e KING (2007), os dados tendem a durarem mais tempo do que as próprias aplicações que os utilizam. Estes dados ainda podem ser utilizados por mais de uma aplicação. É neste contexto que a tecnologia relacional se torna muito importante, pois ela é capaz de prover uma maneira de se compartilhar dados entre aplicações diferentes e mantê-los armazenados independentemente das aplicações que os utilizam.

Os bancos de dados relacionais utilizam a linguagem SQL. Esta linguagem se divide em dois tipos: a DDL (*Data Definition Language*) é utilizada para a criação, alteração e exclusão de tabelas e constantes. Quando um esquema de um banco de dados já está pronto, é utilizada a DML (*Data Manipulation Language*), que serve para a manipulação dos dados envolvidos (BAUER e KING, 2007).

Os próximos subcapítulos abordarão as tecnologias de persistência de dados utilizadas no desenvolvimento do aplicativo proposto.

2.7.1 JPA

Todas as aplicações desenvolvidas nos dias de hoje possuem um componente muito importante, os seus dados. A maioria dos dados manipulados pelas aplicações são armazenados em bancos de dados, recuperados e a partir daí analisados (GONCALVES, 2013).

Os bancos de dados são muito importantes, pois armazenam os dados e servem de ponto central entre as aplicações que os utilizam. A maioria dos bancos utilizados hoje em dia são do tipo relacional, onde os dados são armazenados em tabelas, compostas por linhas e colunas. Os dados são identificados por chaves primárias e se relacionam com outras tabelas através de chaves estrangeiras (GONCALVES, 2013).

O problema que surge, é que uma linguagem orientada a objeto como o Java, não segue os mesmos padrões de um banco de dados relacional. No Java, o que é manipulado são os objetos, que por sua vez são instâncias das classes. Existem classes concretas, classes abstratas, *interfaces,* métodos, atributos e outras características totalmente diferentes das existentes em um banco de dados relacional (GONCALVES, 2013).

Os objetos podem até armazenar seus estados, mas somente enquanto a máquina virtual do Java (JVM) estiver rodando. Se a JVM parar ou o *garbage collector* limpar o conteúdo da memória, os objetos desaparecerão junto com seus estados. Sendo assim, alguns objetos precisam ser persistentes no funcionamento de uma aplicação, ou seja, precisam permanecer armazenados em algum lugar permanente para que sejam reaproveitados depois (GONCALVES, 2013).

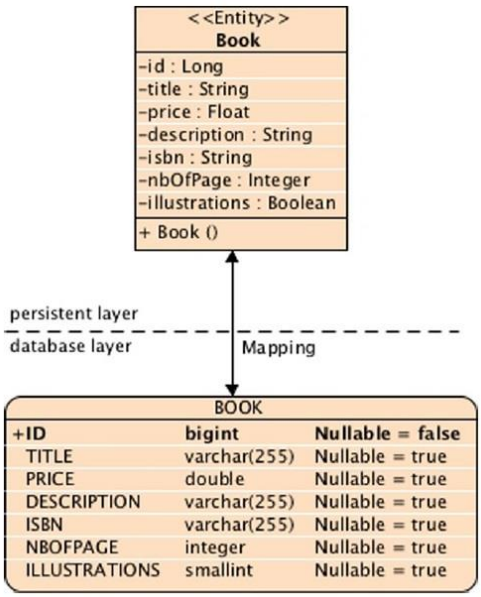
Para resolver este problema surgiu o conceito do *object-relational mapping* (ORM), ou mapeamento objeto-relacional, que é responsável por unir os mundos

dos bancos de dados e dos objetos. Por meio deste conceito é possível que banco de dados relacionais sejam acessados por ferramentas ou *frameworks* que fazem com que os dados relacionais sejam vistos como dados orientados a objeto e vice-versa. A Java Persistence API (JPA) é a principal tecnologia utilizada para atingir este objetivo (GONCALVES, 2013).

Quando é tratado o assunto de mapeamento de objetos para um banco de dados relacional, o termo mais utilizado é “entidade”. Para que seja entendido o básico a respeito da JPA, deve-se ao mínimo saber-se do que se trata uma entidade. Enquanto os objetos são instancias que só vivem na memória enquanto a JVM está rodando, as entidades são objetos que além de viverem na memória são persistidos em um banco de dados. Estas entidades possuem todas as características pertencentes a um objeto e uma vez mapeadas podem ser alteradas, removidas e recuperadas pela JPA (GONCALVES, 2013).

Em resumo, a JPA é uma tecnologia que pode criar uma espécie de correspondência entre objetos e tabelas. As classes, objetos e atributos podem ser mapeados para banco de dados relacionais que possuem tabelas, linhas e colunas. Este mapeamento pode ser visto na Figura 10.

Figura 10: Sincronização de dados entre uma entidade e uma tabela.



Fonte: GONCALVES, 2012.

2.7.2 PostgreSQL

PostgreSQL é um sistema de gerenciamento de banco de dados (SGBD) objeto-relacional desenvolvido a partir do ano de 1977. Começou como um projeto nomeado de Ingres na Universidade da Califórnia em Berkeley. Em 1986 uma equipe liderada por Michael Stonebraker continuou o desenvolvimento do Ingres e criou um banco de dados objeto-relacional chamado de Postgres (WORSLEY e DRAKE, 2002). O projeto contou com o apoio de vários órgãos, entre eles o Army Research Office (ARO) e a National Science Foundation (NSF) (MILANI, 2008).

A primeira versão do projeto ficou pronta em 1987 e foi apresentada em diversos meios. Em 1989 foi lançada a primeira versão estável que contava com atualizações quase que anuais para correção de erros. Em 1991 seu código foi adquirido pela empresa Illustra Information Technologies, que hoje pertence a IBM, e utilizado como SGBD de um grande projeto científico (MILANI, 2008).

Em 1994 ocorreu a primeira grande mudança no Postgres. A linguagem PostQUEL foi trocada pela linguagem SQL pelos desenvolvedores Andrew Yu e Jolly Chen. A partir daí ele foi chamado de Postgre95. Mas em 1996 com o nome já defasado e após algumas melhorias ele passou a ser conhecido como PostgreSQL, nome que leva até hoje (MILANI, 2008).

O PostgreSQL é *open source* e utiliza a licença BSD, a qual possui inúmeras restrições a menos que a licença GNU, o que torna o código muito mais acessível para diversos tipos de utilizações, inclusive para fins comerciais (MILANI, 2008).

Ele pode ser considerado um dos SGBDs mais avançados da atualidade. Segundo Worsley e Drake (2002) e Milani (2008), o PostgreSQL possui um conjunto de características e recursos fundamentais:

- SGBD relacional com suporte a ACID (transações): é um SGBD completo capaz de trabalhar com rotinas e regras complexas. Suporta operações ACID (Atomicidade, Consistência, Isolamento e Durabilidade).
- Replicação: oferece recursos para realizar a replicação de bancos de dados entre servidores.
- Extensibilidade: suporta operações, funções, métodos e tipos de dados definidos pelos usuários.
- *Cluster*: é possível ser configurado em um *cluster* para que trabalhe de forma distribuída.

- *Multithreads*: gerencia várias conexões ao banco de dados de uma única vez.
- Segurança SSL e criptografada: possui suporte nativo a SSL, o que permite a criação de conexões seguras para trafego de informações sigilosas.

2.8 EJB

A camada de persistência, onde está empregada a JPA é a que se encarrega de persistir os objetos do Java em um banco de dados relacional como o PostgreSQL. Mas as entidades criadas nesta camada não foram feitas para executarem tarefas complexas como a interação com outros componentes (outras unidades de persistência, serviços externos, etc.). Sendo assim, na camada de persistência não é recomendado que se aplique a lógica de negócios do sistema. Para que se implemente então a lógica de negócios é necessária a criação de uma camada lógica, e no Java EE 7 esta camada pode ser implementada utilizando-se os Enterprise Java Beans (EJBs) (GONCALVES, 2013).

Os EJBs são componentes que atuam do lado do servidor e são responsáveis pelo encapsulamento da lógica de negócios, das transações e da segurança de uma aplicação. Eles ainda possuem um conjunto de ferramentas para mensagens, acesso remoto, web services, injeção de dependência, clico de vida, dentre outros. Além de se integrarem facilmente a outras tecnologias do Java SE ou do Java EE, como, JDBC, JPA, JTA, JavaMail, JNDI, entre outras (GONCALVES, 2013).

O modelo de programação dos EJBs combina facilidade e robustez, sendo possível utilizá-lo por meio de uma simples anotação no código que será implantado dentro de um *container*. Um *container* EJB é um ambiente de execução que provê serviços como gerenciamento de transação, controle de concorrência, e autorização de segurança. Isto permite então que o desenvolvedor preocupe-se inteiramente com a lógica de negócios da aplicação, enquanto o EJB toma conta das tarefas mais complexas (GONCALVES, 2013).

Existem três tipos de EJBs:

- *Stateless*: Não guardam estado e não são capazes de armazenar dados entre as chamadas de métodos. Qualquer instância pode ser usada por

qualquer cliente. Usado para tarefas que podem ser concluídas com uma chamada única de um método (GONCALVES, 2013).

- *Stateful*: É capaz de guardar estado e permite que diversas chamadas de métodos sejam feitas e mantidas para um único usuário. É utilizado para tarefas que necessitem de várias etapas (GONCALVES, 2013).
- *Singleton*: A sessão é compartilhada entre os clientes e suporta concorrência de acesso. O *container* garante que apenas uma instância seja executada para toda aplicação (GONCALVES, 2013).

3 METODOLOGIA

O aplicativo será desenvolvido com base em um estudo de caso apresentado no próximo capítulo. Este estudo de caso faz uma breve análise das formas utilizadas para controle pessoal de finanças em um contexto atual e apresenta uma análise sobre os aplicativos de controle financeiro existentes atualmente.

Pesquisas bibliográficas também serão utilizadas para a obtenção de conhecimento a respeito da modelagem e das tecnologias que serão utilizadas no desenvolvimento deste aplicativo.

3.1 ESTUDO DE CASO

O foco do trabalho proposto tem como objetivo o desenvolvimento de um aplicativo de controle financeiro pessoal. Ele será desenvolvido para dispositivos móveis e será multiplataforma. Esta seção apresenta o estudo de caso realizado, abordando o contexto atual que leva as pessoas a necessitarem de um aplicativo desses e analisando alguns aplicativos já existentes nessa área.

3.1.1 Contexto Atual

No mundo de hoje, os dispositivos móveis como *smartphones* e *tablets* atendem as mais variáveis demandas de seus usuários. Todas as tarefas que em algum período de tempo atrás eram feitas através de papel e caneta, hoje podem ser executadas de maneira digital através destes dispositivos. Além do aumento da praticidade, ainda tem-se um ganho de desempenho, tempo e confiabilidade.

A tecnologia também facilitou o ato da compra e venda, aumentando assim o número de transações financeiras de uma pessoa durante sua vida. Uma pessoa pode gastar e receber dinheiro das mais variadas maneiras e muitas vezes por dia. Muitas destas pessoas possuem uma vida muito atribulada, devendo-se assim, otimizar o tempo gasto com o controle destas finanças.

Antes do surgimento dos aplicativos de controle financeiro, as pessoas geriam suas finanças através de cadernetas escritas a mão, ou digitalmente, através de planilhas eletrônicas em seus computadores.

Essas planilhas funcionam basicamente através do lançamento de gastos e despesas de uma pessoa durante um período de tempo. E usando-se estes valores lançados é feito um cálculo onde verifica-se o saldo final atingido por ela. Podem também ser lançadas as datas para o pagamento de contas para que a pessoa não se esqueça de seus compromissos.

No caso das planilhas feitas à mão, os cálculos também deveriam ser feitos manualmente, no máximo com o auxílio de uma calculadora, o que acarretaria em um gasto de tempo muito grande para uma tarefa essencial para as pessoas hoje em dia. Além disso os dados poderiam sofrer com falhas de integridade e poderiam ser perdidos muito facilmente.

Nas planilhas eletrônicas o cálculo das receitas e despesas é automatizado através das fórmulas programadas no software, mas as planilhas devem ser criadas por alguém que possua um bom conhecimento da ferramenta. Causando assim problemas de padronização e usabilidade das planilhas. Ainda assim há a desvantagem delas só poderem ser executadas em um mesmo dispositivo, como um Computador Pessoal ou um *Notebook*, e se elas forem salvas somente localmente neste dispositivo, há chance da perda destes dados é muito grande.

Aliando a tecnologia e praticidade que os dispositivos móveis trazem hoje em dia, e a necessidade de uma pessoa possuir um controle sobre suas economias, surge a ideia do desenvolvimento de um aplicativo que fará este controle.

3.1.2 Aplicativos Existentes

Nos últimos anos surgiram vários aplicativos de controle financeiro no mercado de dispositivos móveis. Aqui será feita uma análise superficial de seus recursos e funcionalidades. Serão usados como base os três aplicativos mais baixados até o momento da escrita deste trabalho, na Google Play Store, loja de aplicativos da plataforma Android.

Os aplicativos são: Gastos Diários[2], desenvolvido por Michel Carvajal, com mais de 1 milhão de *downloads*; Gerenciador Financeiro[3], desenvolvido por

[2] Disponível em: <https://goo.gl/eKHUSY> Acesso em abril de 2014.

[3] Disponível em: <https://goo.gl/tM67f> Acesso em abril de 2014.

MobillsLabs, com mais de 500 mil *downloads*; Minhas Economias[4], desenvolvido por Minhas Economias, com mais de 100 mil *downloads*.

Os aplicativos Minhas Economias e Gerenciador Financeiro possuem especificados explicitamente a função de sincronização de dados com um servidor externo na internet, o que garante uma maior integridade e confiabilidade nos dados salvos. Para que isto ocorra, é necessário que exista um cadastro do usuário que utilizará o aplicativo. Sendo esta, a primeira ação necessária de ser feita após a instalação do mesmo. O aplicativo Gastos Diários garante no máximo a proteção dos dados através de uma senha, mas os dados são salvos localmente no dispositivo, aumentando a chance da perda destes dados.

A partir daí, os recursos e funcionalidades dos aplicativos assemelham-se muito uns aos outros. Na tela inicial é sempre mostrado o saldo atual do período de tempo que o usuário selecionou e mais algumas informações.

O ponto principal dos aplicativos é o cadastro de receitas e despesas, presente de forma muito semelhante em todos os aplicativos analisados. Nele basicamente devem ser cadastrados o valor da ação, sua categoria, sua descrição e a data da ação.

O cadastro de categorias de receitas e despesas também pode ser feito em alguns dos aplicativos. Nesse caso o usuário poderá cadastrar uma categoria que só será exibida no aplicativo que ele estiver usando.

Em alguns dos aplicativos também é possível se escolher a conta à que pertence a receita ou despesa. A conta se refere à origem ou o destino do dinheiro utilizado na receita ou despesa. Ela pode ser carteira, cartão de crédito, conta corrente, entre outros, dependendo do aplicativo. Também há a possibilidade de cadastro destas contas em alguns aplicativos.

Após o cadastro das receitas e despesas, o aplicativo trata de fazer os cálculos necessários para exibir o saldo que a pessoa possui no determinado período de tempo que ela selecionou.

[4] Disponível em: <https://goo.gl/hviRR> Acesso em abril de 2014.

4 MODELAGEM DO SISTEMA

Com base nos requisitos levantados e nos aplicativos analisados no estudo de caso foi feita a modelagem do aplicativo que será desenvolvido. Este terá como objetivo controlar as receitas e despesas de uma pessoa, auxiliando em seu gerenciamento financeiro de maneira ágil e segura.

4.1 REQUISITOS FUNCIONAIS E NÃO FUNCIONAIS

Requisitos funcionais definem diretamente uma função que o sistema, ou parte dele deverá executar. Requisitos não funcionais são os que estão relacionados ao uso da aplicação como desempenho, usabilidade e tecnologias envolvidas.

Baseado no estudo de caso foram levantados os requisitos funcionais e não funcionais que serão atendidos pelo aplicativo.

4.1.1 Requisitos Funcionais

- Manter usuários;
- Manter categorias;
- Manter contas;
- Manter operações financeiras (receitas e despesas);
- Calcular as receitas e despesas e gerar o saldo;
- Usar um sistema de *login* e senha para identificar o usuário;
- O usuário só poderá estar logado em um único dispositivo por vez;
- Exibir visualização de saldo.

4.1.2 Requisitos Não Funcionais

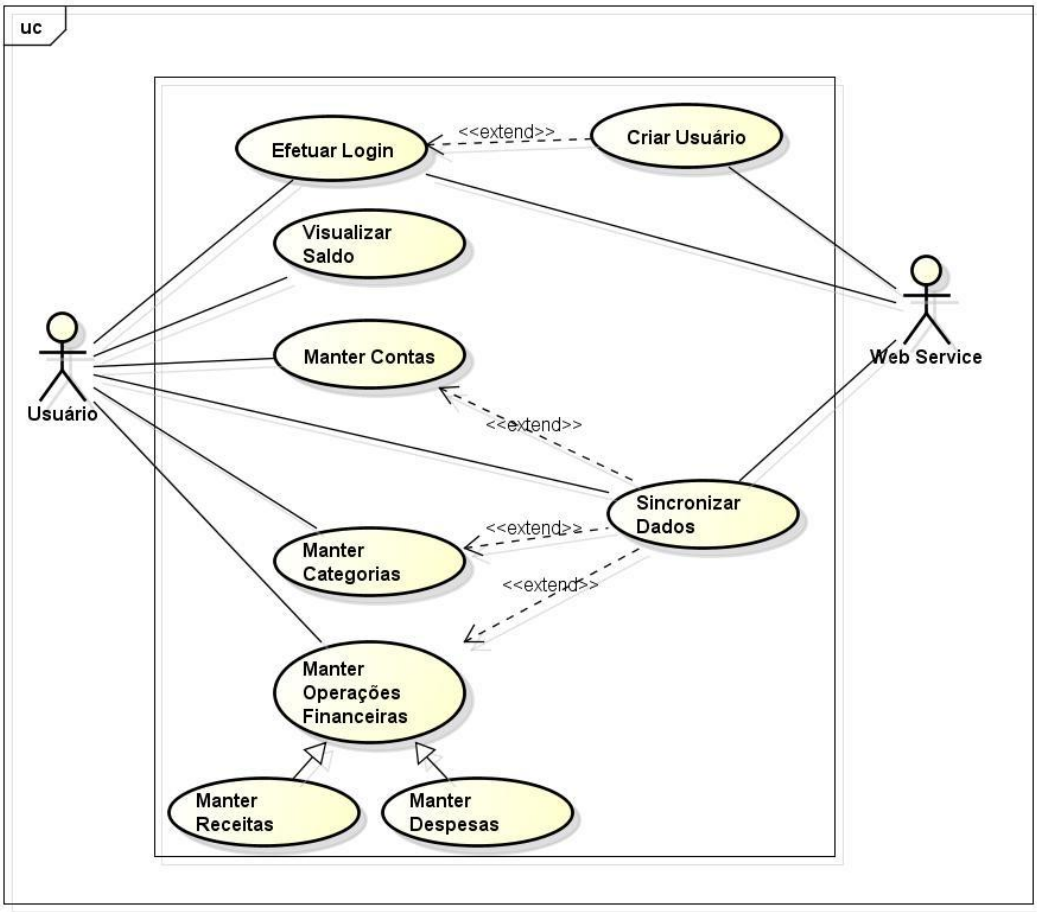
- O aplicativo deve ser multiplataforma;
- O aplicativo deve salvar os dados em um banco de dados local no dispositivo e sincronizá-los com um banco de dados externo;
- A interface do aplicativo deve ser simples e intuitiva.

4.2 DIAGRAMA DE CASOS DE USO

O diagrama de casos de uso tem o objetivo de documentar o que o sistema faz do ponto de vista do usuário. Dentre todos os diagramas da UML, ele é o que descreve os principais processos (casos de uso) do sistema e sua interação com os usuários (atores) do mesmo.

O diagrama a seguir representa os requisitos levantados com base no estudo de caso realizado.

Figura 12: Diagrama de Casos de uso



Fonte: Do autor

4.3 DESCRIÇÃO DOS CASOS DE USO

4.3.1 Caso de Uso Efetuar Login

Nome do Caso de Uso	**Efetuar *Login***
Caso de Uso Geral	
Ator Principal	Usuário
Atores Secundários	WebService
Resumo	Este caso de uso descreve o processo de *login* de um usuário no aplicativo. O usuário só pode estar logado em um dispositivo por vez e este controle é feito baseado na tabela de dispositivos pertencentes ao usuário.
Pré-Condições	Conexão com a internet
Pós-Condições	*Login* efetuado ou não
Fluxo principal	
Ações do Ator	**Ações do Sistema**
1. Informar seu e-mail e senha	
	2. Consultar o banco de dados remoto por meio do Web Service.
	3. Se o usuário existir e o dispositivo pelo qual ele está acessando o aplicativo estiver cadastrado no banco de dados remoto com status 1 (ativo), é autorizado o acesso ao aplicativo. Caso o usuário ou senha estiverem errados é emitida uma mensagem de erro. Caso o dispositivo não esteja cadastrado ou esteja inativo (status 0), chamar Fluxo Alternativo II ou o Fluxo Alternativo III.
Restrições/Validações/Regras de Negócio	1. O usuário precisa possuir um cadastro no banco de dados remoto.
	2. O cadastro do usuário não pode estar ativo em outro dispositivo.
Fluxo Alternativo I – Usuário ainda não cadastrado	
Ações do Ator	**Ações do Sistema**
	1. Executar Caso de Uso Criar Usuário
Fluxo Alternativo II – Dispositivo ainda não cadastrado	

Ações do Ator	Ações do Sistema
	1. Registrar o dispositivo no banco de dados remoto com o status 1 (ativo)
	2. Alterar o status dos outros dispositivos para 0 (inativo)
Restrições/Validações/Regras de Negócio	1. Importar os dados do usuário logado para o banco de dados local
Fluxo alternativo III – Usuário ativo em outro celular	
	1. Perguntar ao usuário se deseja que seu cadastro seja desativado do outro dispositivo.
2. Usuário responde sim ou não.	
	3. Se a resposta for positiva, o cadastro do usuário é desativado do outro dispositivo (o outro dispositivo cadastrado no banco de dados remoto tem seu status alterado para 0 (inativo)), de forma com que ele só possa ser acessado pelo dispositivo atual. Além disso, os dados contidos no banco de dados remoto são importados para o banco de dados local do dispositivo através do Web Service. Se for negativa, o usuário não efetua login neste dispositivo e só pode acessar o aplicativo por meio do outro dispositivo em que está ativo.
Restrições/Validações/Regras de Negócio	1. Importar os dados do usuário logado para o banco de dados local
Estrutura de Dados	
1. E-mail do usuário	
2. Senha do usuário	
3. Identificador do dispositivo	

4.3.2 Caso de Uso Criar Usuário

Nome do Caso de Uso	Criar Usuário
Caso de Uso Geral	
Ator Principal	Usuário
Atores Secundários	WebService
Resumo	Este caso de uso descreve o processo de criação de um novo usuário na aplicação.
Pré-Condições	Conexão com a internet
Pós-Condições	Usuário criado
Fluxo principal	
Ações do Ator	**Ações do Sistema**
1. Solicitar a criação de um novo usuário.	
	2. Exibir uma tela de preenchimento de dados de cadastro.
3. Informar os dados necessários para o cadastro.	
	4. Acessar o Web Service e registar o usuário no banco de dados remoto.
Restrições/Validações/Regras de Negócio	1. Não pode haver um usuário cadastrado no banco de dados remoto com o mesmo e-mail
	2. O nome do usuário, e-mail e senha são campos obrigatórios
	3. O login do usuário deve ser um e-mail no formato válido
	4. A senha deve possuir no mínimo seis caracteres
	5. Ao registrar o novo usuário deve ser gravado também o dispositivo na tabela Dispositivos e esse dispositivo deve receber o status 1 (ativo)
Estrutura de Dados	
1. Nome do usuário	
2. E-mail do usuário (Utilizado como login)	
3. Senha do usuário	
4. Identificador do dispositivo	

4.3.3 Caso de Uso Visualizar Saldo

Nome do Caso de Uso	Visualizar Saldo
Caso de Uso Geral	
Ator Principal	Usuário
Atores Secundários	
Resumo	Este caso de uso descreve o processo de visualização de saldo pelo usuário do aplicativo.
Pré-Condições	Estar autenticado no aplicativo
Pós-Condições	Saldo visível ao usuário
Fluxo principal	
Ações do Ator	**Ações do Sistema**
1. Solicitar a visualização do saldo.	
	2. Selecionar do banco de dados local as operações e a partir delas fazer os cálculos necessários e exibir o saldo do usuário na tela.
Restrições/Validações/Regras de Negócio	1. O usuário já deve ter cadastrado alguma operação financeira de receita ou despesa.
	2. O usuário pode escolher um determinado período de tempo, ou uma conta ou categoria específica para a exibição do saldo.
	3. Somente selecionar as operações com o status de sincronização diferente de 3 (excluído)
Fluxo Alternativo I – Usuário ainda não cadastrou operação financeira	
Ações do Ator	**Ações do Sistema**
	1. Exibe o saldo de R$0,00 e uma mensagem de que ainda não foram cadastradas operações financeiras

4.3.4 Caso de Uso Manter Contas

Nome do Caso de Uso	Manter Contas
Caso de Uso Geral	
Ator Principal	Usuário
Atores Secundários	
Resumo	Este caso de uso descreve as possíveis atividades de manutenção do cadastro de contas (carteira, cartão de crédito, conta corrente), ou seja, permite visualizar, cadastrar, alterar ou excluir contas na aplicação.
Pré-Condições	Estar autenticado no aplicativo
Pós-Condições	Conta visualizada, criada, alterada ou excluída
Fluxo principal	
Ações do Ator	**Ações do Sistema**
1. Solicitar a criação de uma nova conta.	
	2. Exibir a tela com o formulário de criação de uma conta.
3. Informar o nome da conta.	
	4. Chamar a validação de regras de negócio e registrar a conta no banco de dados local.
Restrições/Validações/Regras de Negócio	1. Não haver uma conta cadastrada com o mesmo nome por este usuário
	2. O nome da conta deve ter no mínimo dois caracteres
	3. A conta deve ser registrada no banco de dados local com o status de sincronização igual a 1 (novo).
Fluxo Alternativo I – Alterar Conta	
Ações do Ator	**Ações do Sistema**
1. Selecionar a opção de Visualização de Contas	
	2. Exibir as contas que o usuário tem cadastradas que possuírem o status de sincronização

	diferente de 3 (excluído).
3. Selecionar a conta que deseja ser alterada e realiza as alterações desejadas.	
	4. Chamar a validação de regras de negócio e registrar a alteração da conta no banco de dados local.
Restrições/Validações/Regras de Negócio	1. Não haver uma conta cadastrada com o mesmo nome por este usuário
	2. O nome da conta deve ter no mínimo dois caracteres
	3. A conta deve ter seu status de sincronização alterado para 2 (alterado)
Fluxo Alternativo II – Excluir Conta	
Ações do Ator	**Ações do Sistema**
1. Selecionar a opção de Visualização de Contas	
	2. Exibir as contas que o usuário tem cadastradas que possuírem o status de sincronização diferente de 3 (excluído).
3. Selecionar a conta que deseja excluir.	
	4. Alterar o status de sincronização da conta para 3 (excluído).
Fluxo Alternativo III – Sincronizar Dados com o Banco de Dados Remoto	
Ações do Ator	**Ações do Sistema**
1. Se o ator selecionou a opção de sincronização de dados.	
	2. Executar o Caso de Uso Sincronizar Dados.
Estrutura de Dados	
1. Nome da conta	

4.3.5 Caso de Uso manter Categorias

Nome do Caso de Uso	Manter Categorias
Caso de Uso Geral	
Ator Principal	Usuário
Atores Secundários	
Resumo	Este caso de uso descreve as possíveis atividades de manutenção do cadastro de categorias de uma operação financeira, ou seja, permite visualizar, cadastrar, alterar ou excluir categorias na aplicação.
Pré-Condições	Estar autenticado no aplicativo
Pós-Condições	Categoria visualizada, criada, alterada ou excluída
Fluxo principal	
Ações do Ator	**Ações do Sistema**
Idênticas as do caso de uso Manter Contas, exceto pelo fato de ser o cadastro de categorias.	Idênticas as do caso de uso Manter Contas, exceto pelo fato de ser o cadastro de categorias.
Restrições/Validações/Regras de Negócio	Idênticas as do caso de uso Manter Contas, exceto pelo fato de ser o cadastro de categorias.
Fluxo Alternativo I – Alterar Categoria	
Ações do Ator	**Ações do Sistema**
Idênticas as do caso de uso Manter Contas, exceto pelo fato de ser a alteração de categorias.	Idênticas as do caso de uso Manter Contas, exceto pelo fato de ser a alteração de categorias.
Fluxo Alternativo II – Excluir Conta	
Ações do Ator	**Ações do Sistema**
Idênticas as do caso de uso Manter Contas, exceto pelo fato de ser a exclusão de categorias.	Idênticas as do caso de uso Manter Contas, exceto pelo fato de ser a exclusão de categorias.
Fluxo Alternativo III – Sincronizar Dados com o Banco de Dados Remoto	
Ações do Ator	**Ações do Sistema**
Idênticas as do caso de uso Manter Contas, exceto pelo fato de ser a sincronização de categorias.	Idênticas as do caso de uso Manter Contas, exceto pelo fato de ser a sincronização de categorias.
Estrutura de Dados	

1. Nome da categoria

4.3.6 Caso de Uso Manter Operações Financeiras

Nome do Caso de Uso	Manter Operações Financeiras
Caso de Uso Geral	
Ator Principal	Usuário
Atores Secundários	
Resumo	Este caso de uso descreve as possíveis atividades de manutenção do cadastro de operações financeiras (receitas e despesas) ou seja, permite visualizar, cadastrar ou excluir operações financeiras na aplicação.
Pré-Condições	Estar autenticado no aplicativo
Pós-Condições	Operação financeira visualizada, criada, ou excluída
Fluxo principal	
Ações do Ator	**Ações do Sistema**
1. Solicitar a criação de uma nova operação financeira.	
	2. Exibir a tela com o formulário de criação de uma nova operação financeira.
3. Informar o valor da operação, selecionar que tipo de operação ela é (receita ou despesa) e selecionar categoria e conta a qual ela pertence.	
	4. Chamar a validação de regras de negócio e registrar a operação no banco de dados local.
Restrições/Validações/Regras de Negócio	1. Os campos valor, tipo, categoria, conta e data são obrigatórios
	2. O valor deve ser um número positivo
	3. A operação deve ser registrada no banco de dados local com o status de sincronização igual a 1 (novo).

Fluxo Alternativo I – Excluir Operação Financeira	
Ações do Ator	**Ações do Sistema**
1. Selecionar a opção de Visualização de Operações Financeiras	
	2. Exibir as contas que o usuário tem cadastradas que possuírem o status de sincronização diferente de 3 (excluído).
3. Selecionar a operação que deseja excluir.	
	4. Alterar o status de sincronização da operação financeira para 3 (excluído)
Fluxo Alternativo II – Sincronizar Dados com o Banco de Dados Remoto	
Ações do Ator	**Ações do Sistema**
1. Se o ator selecionou a opção de sincronização de dados.	
	2. Executar o Caso de Uso Sincronizar Dados.
Estrutura de Dados	
1. Valor da operação	
2. Tipo da operação (receita ou despesa)	
3. Categoria da operação (cadastradas pelo usuário)	
4. Conta a que pertence a operação (cadastradas pelo usuário)	
5. Descrição da operação	
6. Data da operação	

4.3.7 Caso de Uso Sincronizar Dados

Nome do Caso de Uso	Sincronizar Dados
Caso de Uso Geral	
Ator Principal	Usuário
Atores Secundários	Web Service
Resumo	Este caso de uso descreve a operação de sincronização de dados entre o banco de dados local e o banco de dados remoto. Se um dado for criado, alterado ou excluído localmente, ele será atualizado no banco de dados

	remoto.
Pré-Condições	Estar autenticado no aplicativo e conectado à internet
Pós-Condições	Dados sincronizados
Fluxo principal	
Ações do Ator	**Ações do Sistema**
1. Solicitar sincronização de dados.	
	2. Verificar no banco de dados local, os registros (contas, categorias ou operações) que possuírem o status de sincronização diferente de 4 (sincronizado)
	3. Se o status de sincronização do registro for igual a 1 (novo), deve ser criado um novo registro com as mesmas informações no banco de dados remoto através do Web Service. Se o status de sincronização do registro for igual a 2 (alterado), o registro que possuir o mesmo ID no banco de dados remoto deve ser alterado. Se o status de sincronização do registro for igual a 3 (excluído), o registro de mesmo ID deve ser excluído do banco de dados remoto e do banco de dados local.
	4. Após a sincronização com o banco de dados remoto ser concluída, os registros que foram sincronizados têm seus status de sincronização alterados para 4 (sincronizado).
Restrições/Validações/Regras de Negócio	1. O dispositivo deve estar conectado à internet para fazer a comunicação
Fluxo Alternativo I – Sem conexão com a internet	
Ações do Ator	**Ações do Sistema**
	1. Solicitar conexão com a internet.

4.4 DIAGRAMA DE ATIVIDADES

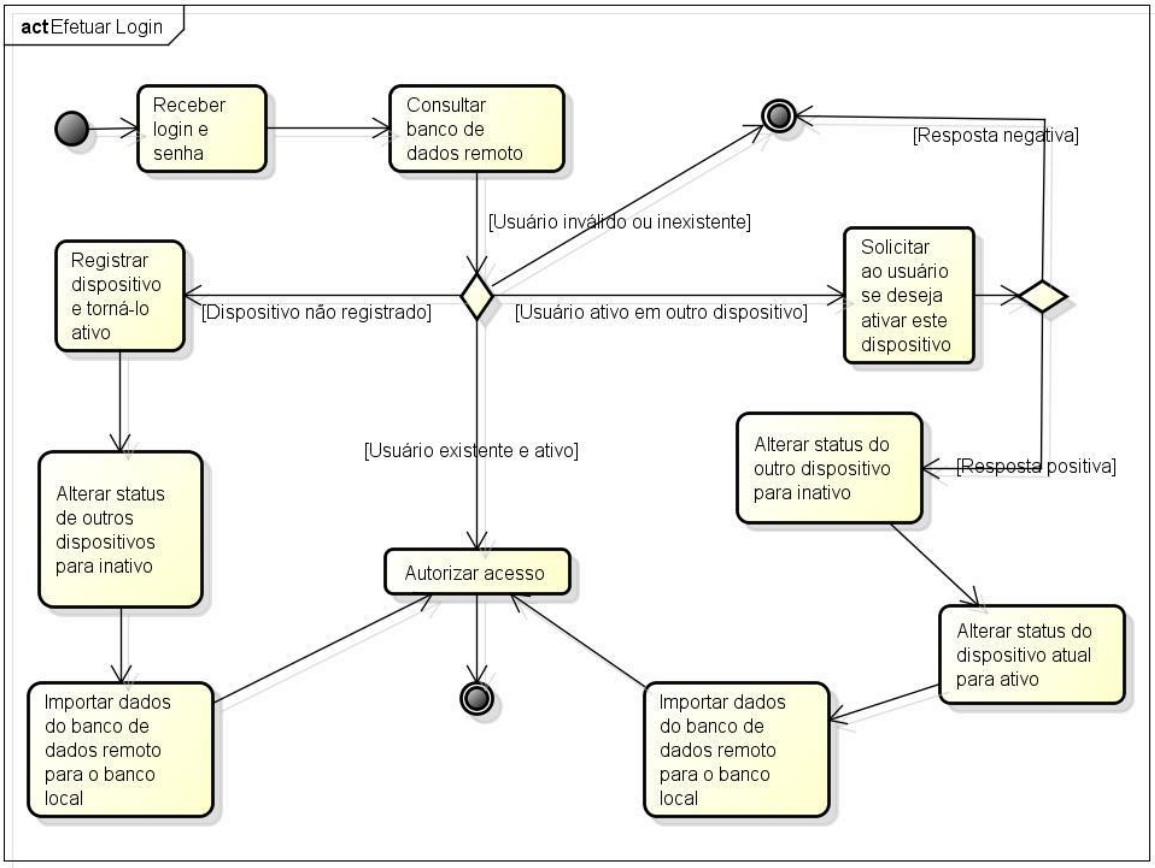
O diagrama de atividades apresenta muitas semelhanças com os antigos fluxogramas utilizados para determinar o fluxo de controle de um algoritmo. Seu objetivo é demonstrar o fluxo das atividades de um sistema, partindo de um ponto inicial até chegar a um ponto final, apresentando as ações intermediárias a estes pontos.

A seguir serão apresentados diagramas de algumas atividades consideradas importantes em relação à aplicação a ser desenvolvida.

4.4.1 Diagrama de Atividades Efetuar *Login*

Este diagrama de atividades tem como objetivo demonstrar os fluxos possíveis existentes no momento em que um usuário tenta efetuar *login* no aplicativo.

Figura 13: Diagrama de Atividades Efetuar Login.



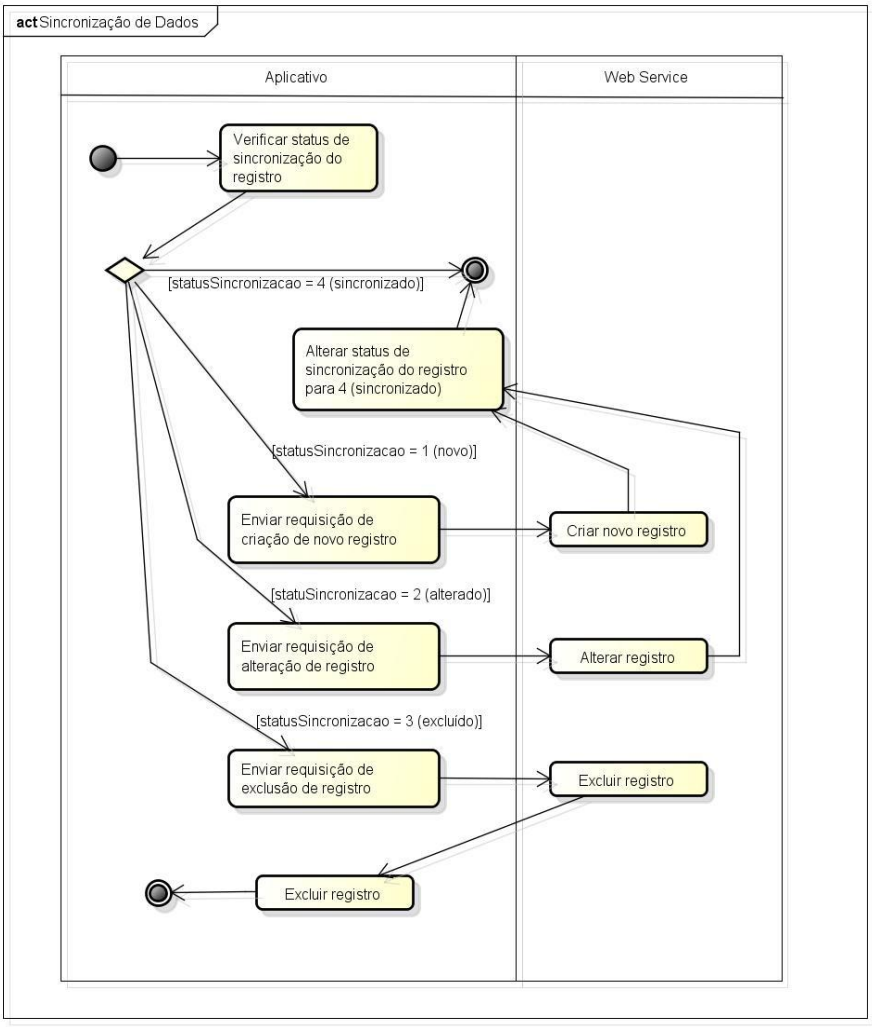
Fonte: Do autor

O aplicativo a ser desenvolvido possui a restrição de que o usuário somente possa estar ativo em um dispositivo móvel por vez. Esta restrição serve para que somente o dispositivo que estiver ativo possa sincronizar dados com o banco de dados remoto, de maneira que ao logar-se em outro dispositivo, este seja obrigado a importar todos os dados referentes ao usuário que estiverem gravados no banco de dados remoto. Sendo assim o diagrama apresentado na Figura 13 demonstra as ações que serão tomadas pelo sistema ao se deparar com este tipo de restrição.

4.4.2 Diagrama de Atividades de Sincronização de Dados

Este diagrama de atividades tem como objetivo demonstrar os fluxos possíveis existentes no momento em que o sistema executará o processo de sincronização de dados.

Figura 14: Diagrama de Atividades de Sincronização de Dados.



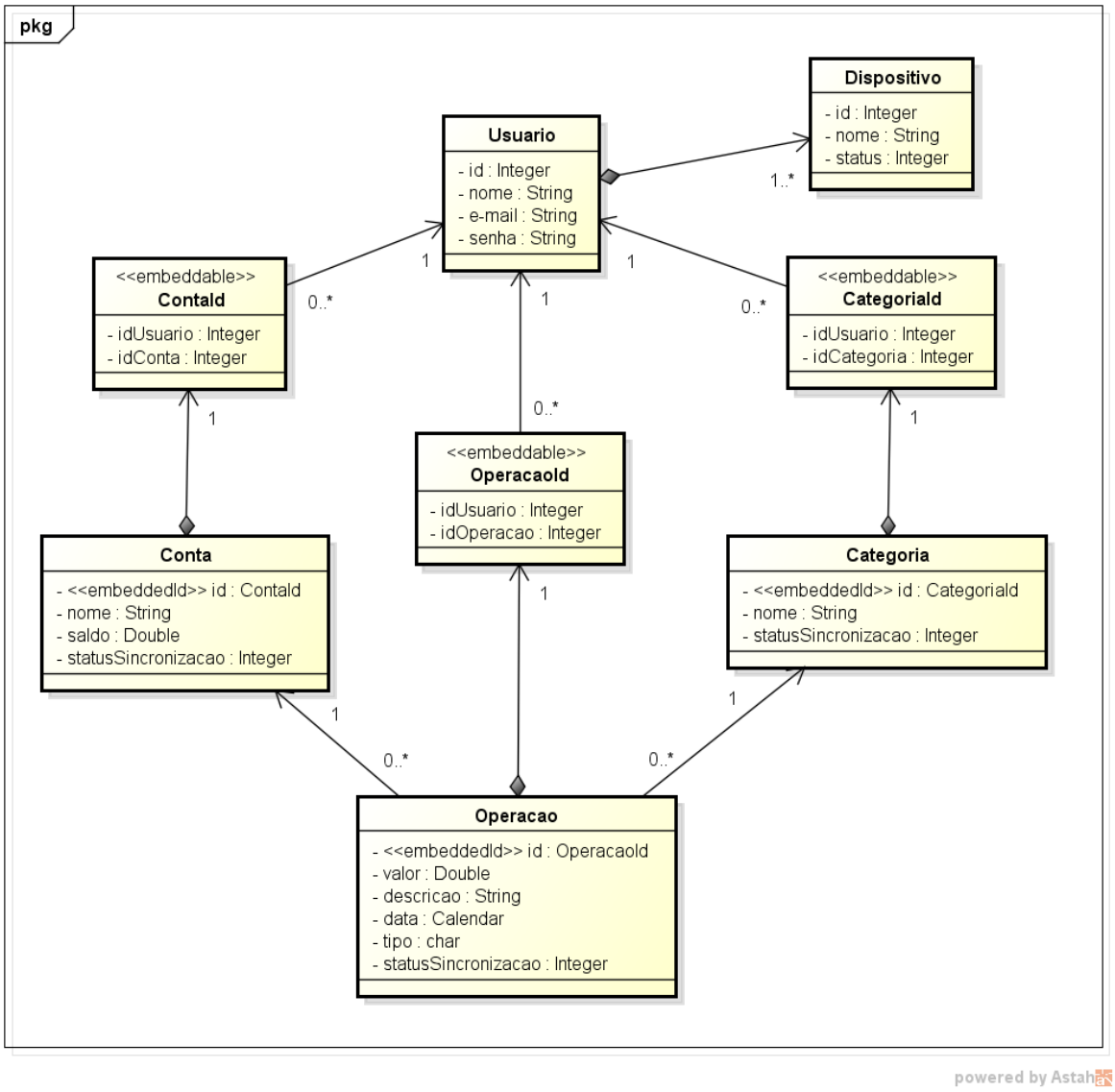
Fonte: Do autor

O processo de sincronização de dados do aplicativo acontece de maneira unilateral, quando houver uma solicitação de sincronização, os dados armazenados localmente no dispositivo serão enviados ao banco de dados remoto por meio do *web service*. Este processo se dá baseado em um campo chamado “statusSincronizacao” que está presente nas tabelas “Contas”, “Categorias” e “Operacoes”, sendo estas as tabelas passíveis a sincronização de seus dados. No momento em que for solicitada a sincronização, o aplicativo verificará em seu banco de dados local o valor armazenado no campo “statusSincronizacao”, e a partir dele executará as ações necessárias para sincronizar estes dados com o banco remoto como demonstra o diagrama da Figura 14.

4.5 DIAGRAMA DE CLASSES

O diagrama de classes visa demonstrar como é a organização das classes dentro do sistema. No caso da figura 15 é apresentado um diagrama com foco na persistência de dados. Nele se encontram as classes, seus atributos, relacionamentos e multiplicidades.

Figura 15: Diagrama de Classes



Fonte: Do autor

A classe “Usuário” possui os atributos referentes ao usuário do aplicativo, como seu e-mail, que será utilizado como *login,* e sua senha.

A classe “Dispositivo” possui seus atributos referentes aos dispositivos com que o usuário acessou o aplicativo. O atributo “nome” será o identificador enviado pelo dispositivo no momento em que for feito o *login* ou a criação de um novo usuário. O atributo “status” é o que verificará se o dispositivo é ativo ou inativo. Sendo o valor 1 para ativo e 0 para inativo.

As classes “ContaId”, “OperacaoId” e “CategoriaId” serão responsáveis pela criação das chaves compostas que servirão como ID único das classes “Conta”, “Operacao” e “Categoria” respectivamente.

A classe “Conta” se refere as contas que poderão ser criadas pelo usuário, como carteira, cartão de crédito e afins. O atributo “saldo” receberá o valor 0 no momento em que for criada a conta e será alterado a cada operação registrada referente a esta conta.

A classe “Categoria” se refere as categorias de operações que podem ser criadas pelo usuário, como alimentação, combustível, padaria e afins.

O atributo “statusSincronizacao”, presente nas classes “Conta”, “Categoria” e “Operação” está ligado ao processo de sincronização de dados como foi explicado anteriormente. Ele pode receber os valores 1, 2, 3 ou 4. No momento em que for criado um novo registro em uma dessas classes, seu valor será igual a 1. Quando um registro for alterado, seu valor será igual a 2. Quando um registro for excluído, seu valor será igual a 3. E quando um registro for sincronizado com o banco de dados remoto, seu valor será igual a 4.

5 DESENVOLVIMENTO

Nesta sessão serão apresentadas informações sobre o desenvolvimento do sistema, como o detalhamento da arquitetura construída, a preparação do ambiente de desenvolvimento e alguns detalhes sobre o desenvolvimento de cada componente e camada do sistema.

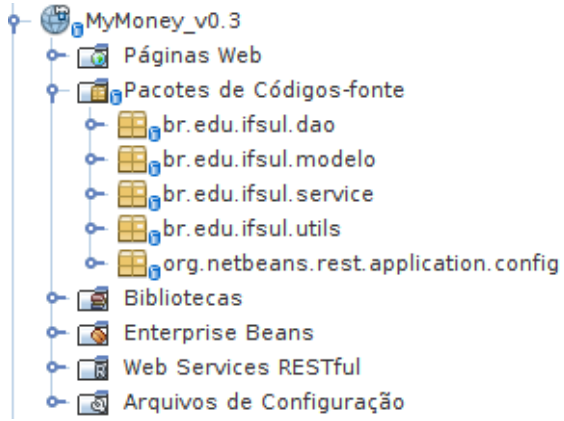
5.1 PREPARAÇÃO DO AMBIENTE DE DESENVOLVIMENTO

O My Money foi projetado para ser desenvolvido a partir da utilização de uma arquitetura constituída por um aplicativo móvel e um servidor web, isso poderá ser visto nas próximas sessões.

5.1.1 Servidor

O sistema começou a ser desenvolvido a partir da camada do servidor web, e como IDE para a programação desta camada foi utilizado o NetBeans 8.0.2. O SGBD utilizado para a persistência de dados no servidor foi o PostgreSQL versão 9.4, com auxílio da ferramenta PGAdmin III. No NetBeans foi criado um projeto Java Web e foram selecionados o GlassFish Server 4.1 como servidor e o Java EE 7 Web como versão do Java EE. O projeto ficou com a estrutura apresentada na Figura 16.

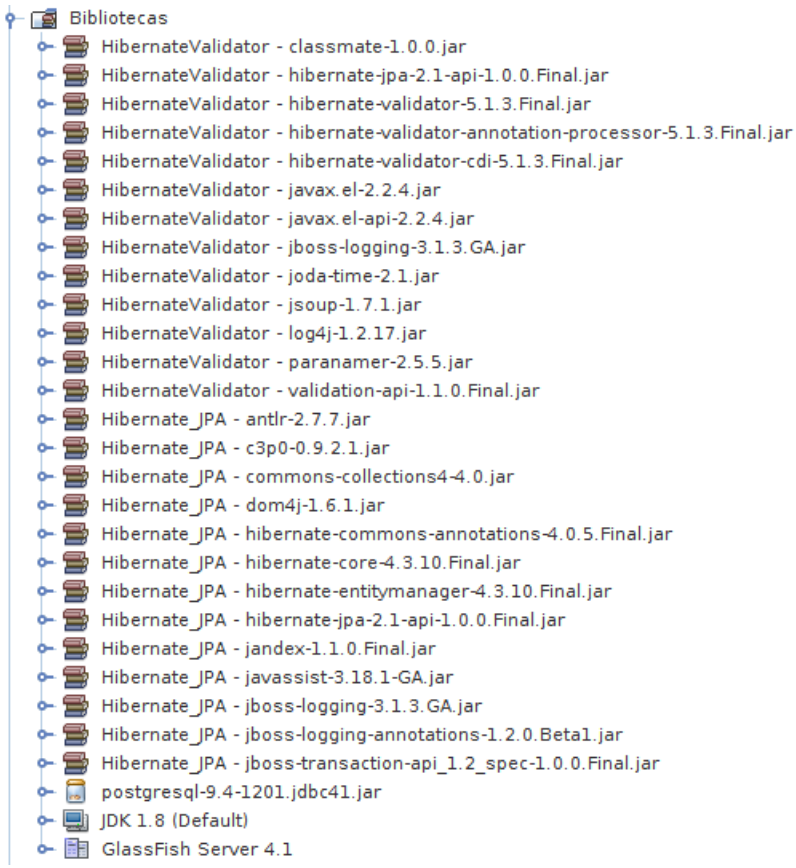
Figura 16: Projeto do web service no NetBeans



Fonte: Do autor

Para este projeto foram importadas as bibliotecas do Hibernate versão 4.3.10 que implementa a versão 2.1 da JPA, do Hibernate Validator versão 5.1.3 que implementa a especificação da *Bean Validation API* para as validações de dados e o driver JDBC do PostgreSQL versão 9.4 responsável pela conexão com o banco de dados. Elas podem ser vistas na Figura 17.

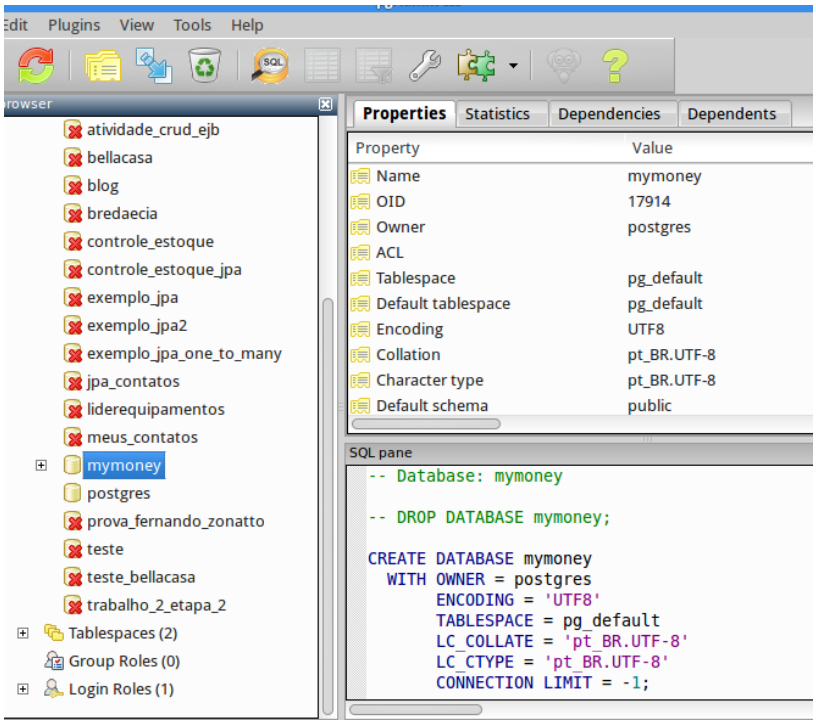
Figura 17: Bibliotecas utilizadas no projeto



Fonte: Do autor

O banco de dados foi criado utilizando-se a ferramenta PgAdmin III como pode ser visto na Figura 18.

Figura 18: Banco de dados do servidor



Fonte: Do autor

No projeto foi criado o arquivo de persistência de dados, que é responsável pela configuração da aplicação para que ela possa utilizar o recurso JDBC do servidor GlassFish e delegar a ele o gerenciamento transacional dos objetos. O arquivo persistence.xml é exigido pela especificação da JPA, pois ele conecta o provedor de JPA ao banco de dados. Este arquivo pode ser visto na Figura 19.

Figura 19: Código do arquivo de persistência de dados (*persistence.xml*)

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="2.1"
             xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
            http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
  <persistence-unit name="MyMoneyPU" transaction-type="JTA">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <jta-data-source>jdbc/MyMoney</jta-data-source>
    <exclude-unlisted-classes>false</exclude-unlisted-classes>
    <properties>
      <property name="hibernate.hbm2ddl.auto" value="update"/>
      <property name="hibernate.dialect" value="org.hibernate.dialect.PostgreSQLDialect"/>
      <property name="hibernate.transaction.jta.platform"
                value="org.hibernate.service.jta.platform.internal.SunOneJtaPlatform"/>
      <property name="hibernate.classloading.use_current_tccl_as_parent" value="false"/>
    </properties>
  </persistence-unit>
</persistence>
```

Fonte: Do autor

O elemento <*persistence-unit*> define o nome da persistência e o tipo de transação, que neste caso é feito por meio da JTA. O elemento <*provider*> define que o provedor da persistência é o Hibernate e o elemento <*jta-data-source*> aponta o nome da fonte de dados. A propriedade <*property name="hibernate.hbm2ddl.auto" value="update"/*> indica que as tabelas do banco de dados serão atualizadas automaticamente assim que o mesmo for criado. A propriedade <*property name="hibernate.dialect" value="org.hibernate.dialect.PostgreSQLDialect"/*> define o dialeto do PostgreSQL, gerando assim SQL otimizado para este SGBD e a propriedade seguinte define a plataforma de transação do Hibernate.

Junto ao arquivo de persistência foi criado o arquivo *glassfish_resources.xml* e nele estão algumas propriedades referentes ao pool de conexões do GlassFish. O arquivo pode ser visto na Figura 20.

Figura 20: Código do arquivo *glassfish_resources.xml*

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE resources PUBLIC "-//GlassFish.org//DTD GlassFish Application Server 3.1
Resource Definitions//EN" "http://glassfish.org/dtds/glassfish-resources_1_5.dtd">
<resources>
    <jdbc-connection-pool allow-non-component-callers="false" associate-with-thread="false"
                          connection-creation-retry-attempts="0"
                          connection-creation-retry-interval-in-seconds="10"
                          connection-leak-reclaim="false" connection-leak-timeout-in-seconds="0"
                          connection-validation-method="auto-commit"
                          datasource-classname="org.postgresql.ds.PGSimpleDataSource"
                          fail-all-connections="false"
                          idle-timeout-in-seconds="300" is-connection-validation-required="false"
                          is-isolation-level-guaranteed="true" lazy-connection-association="false"
                          lazy-connection-enlistment="false" match-connections="false"
                          max-connection-usage-count="0"
                          max-pool-size="32" max-wait-time-in-millis="60000"
                          name="moneyPool" non-transactional-connections="false"
                          pool-resize-quantity="2" res-type="javax.sql.DataSource"
                          statement-timeout-in-seconds="-1"
                          steady-pool-size="8" validate-atmost-once-period-in-seconds="0"
                          wrap-jdbc-objects="false">
        <property name="serverName" value="localhost"/>
        <property name="portNumber" value="5432"/>
        <property name="databaseName" value="mymoney"/>
        <property name="User" value="postgres"/>
        <property name="Password" value="123456"/>
        <property name="URL" value="jdbc:postgresql://localhost:5432/mymoney"/>
        <property name="driverClass" value="org.postgresql.Driver"/>
    </jdbc-connection-pool>
    <jdbc-resource enabled="true" jndi-name="jdbc/MyMoney" object-type="user" pool-name="moneyPool"/>
</resources>
```

Fonte: Do autor

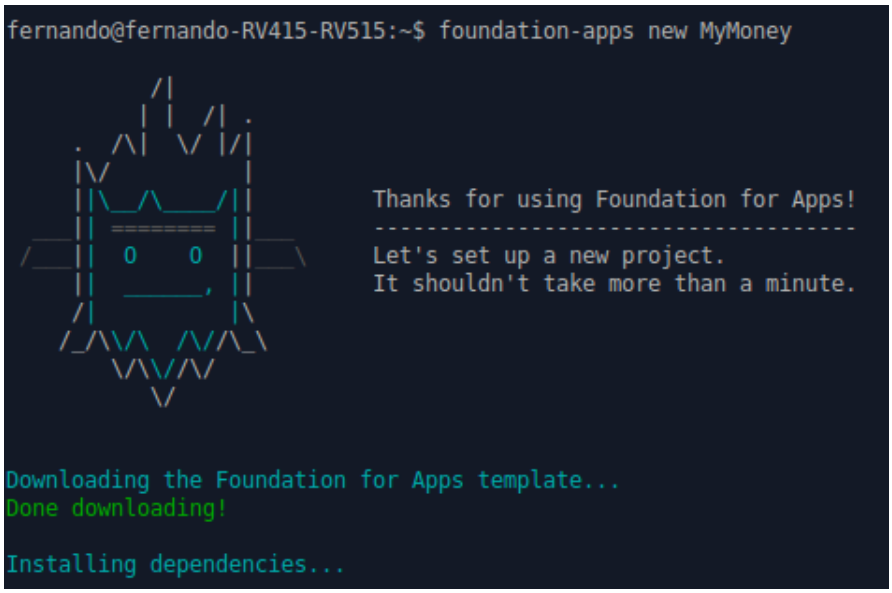
O atributo *name* do elemento <*jdbc-connection-pool*> define o nome do recurso JDBC que o servidor GlassFish utilizará. As propriedades *serverName*, *portNumber*, *dataBaseName*, *User*, *Password*, URL e *driverClass* são referentes a

conexão como banco de dados, e no elemento *<jdbc-resource>* são definidos os nomes do recurso e do *pool* de conexões a serem utilizados.

5.1.2 Aplicativo Móvel

Após a preparação do ambiente para a criação do lado do servidor, foi feita a preparação para o desenvolvimento do aplicativo móvel. O projeto do aplicativo foi criado por meio do *framework* Foundation for Apps, direcionado ao desenvolvimento de aplicativos móveis baseados em aplicações web. Toda instalação e criação de aplicativos por meio deste *framework* pode ser feita via terminal de linha de comando. Para instalá-lo é necessário que seja executado o comando *npm install -g foundation-cli bower gulp* e para a criação de um aplicativo é necessária a execução do comando *foundation-apps new nomeDoProjeto* dentro da pasta onde deseja-se criar o projeto. O resultado da execução do comando pode ser visto na Figura 21.

Figura 21: Resultado do comando *foundation-apps new MyMoney* no terminal



Fonte: Do autor

Após a execução deste comando é criada uma pasta com a estrutura do projeto, como pode ser visto na Figura 22.

Figura 22: Estrutura do projeto do aplicativo

Fonte: Do autor

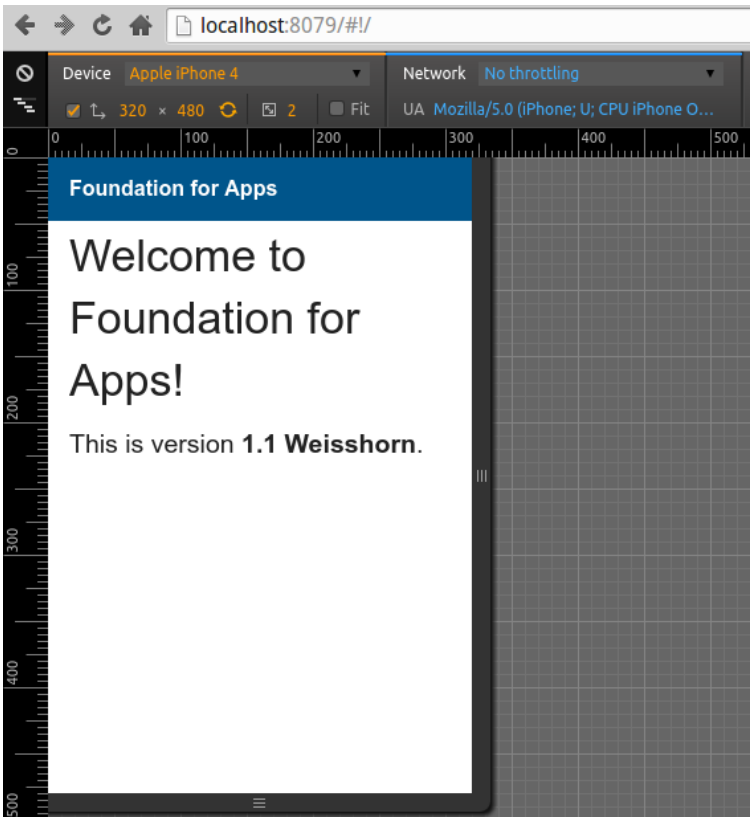
Para a compilação e execução do projeto é necessário que se execute o comando *foundation-apps watch* dentro da pasta do projeto. O resultado da execução deste comando pode ser visto na Figura 23 e a execução do aplicativo no navegador pode ser vista na Figura 24.

Figura 23: Resultado do comando *foundation-apps watch*

```
fernando@fernando-RV415-RV515:~$ cd MyMoney/
fernando@fernando-RV415-RV515:~/MyMoney$ foundation-apps watch

> foundation-apps-template@1.1.0 start /home/fernando/MyMoney
> gulp

[23:33:37] Using gulpfile ~/MyMoney/gulpfile.js
[23:33:37] Starting 'build'...
[23:33:37] Starting 'clean'...
[23:33:37] Finished 'clean' after 11 ms
[23:33:37] Starting 'copy'...
[23:33:37] Starting 'copy:foundation'...
[23:33:39] Finished 'copy:foundation' after 1.49 s
[23:33:39] Starting 'sass'...
[23:33:41] Starting 'uglify:foundation'...
[23:33:41] Starting 'uglify:app'...
[23:33:41] Finished 'copy' after 4 s
[23:33:41] Finished 'uglify:app' after 352 ms
[23:33:42] Finished 'uglify:foundation' after 1.15 s
[23:33:42] Starting 'uglify'...
[23:33:42] Finished 'uglify' after 38 µs
[23:33:43] Finished 'sass' after 3.95 s
[23:33:43] Starting 'copy:templates'...
[23:33:43] Finished 'copy:templates' after 70 ms
[23:33:43] Finished 'build' after 5.56 s
[23:33:43] Starting 'server'...
[23:33:44] Webserver started at http://localhost:8079
[23:33:44] Finished 'server' after 1.21 s
[23:33:44] Starting 'default'...
[23:33:44] Finished 'default' after 55 ms
```

Fonte: Do autor

Figura 24: Aplicativo criado, sendo executado no navegador



Fonte: Do autor

Para a programação do aplicativo móvel foi utilizado o editor Sublime Text 3 e para a visualização do aplicativo em execução foi utilizado o navegador Google Chrome com o *Device mode* ativado.

5.2 ARQUITETURA DO SISTEMA

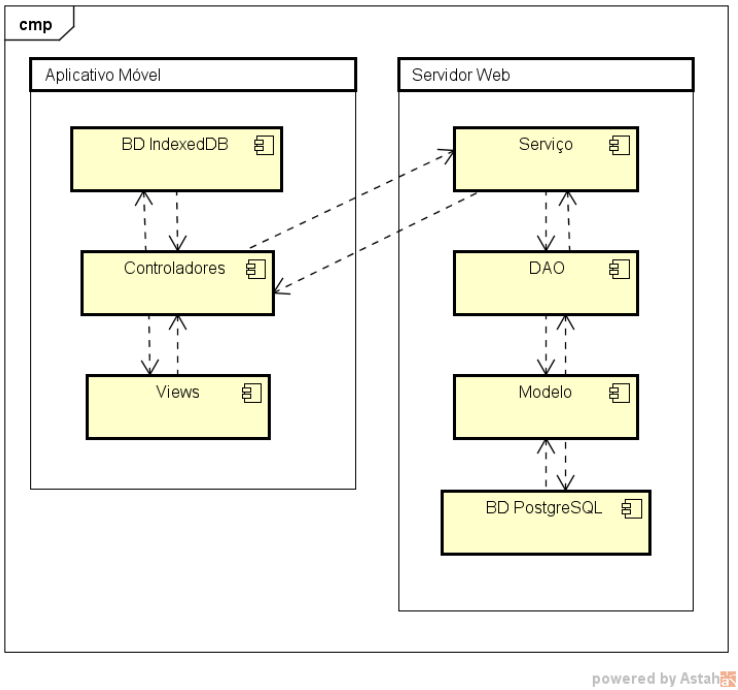
O sistema possui uma arquitetura distribuída, baseada em dois componentes. O primeiro, uma aplicação para o dispositivo móvel, e o segundo, uma aplicação para a persistência remota dos dados. O aplicativo móvel processa e salva seus dados localmente no dispositivo, enquanto a aplicação no servidor remoto tem como objetivo principal a criação de usuários do aplicativo e a sincronização de dados.

As duas aplicações possuem subcamadas. O servidor web está divido em três camadas: modelo, DAO e serviço. A camada modelo é responsável pelo mapeamento objeto-relacional e pela persistência de dados por meio da JPA, nela se encontram as entidades que serão mapeadas e se tornarão tabelas no banco de dados do servidor. A camada DAO é a responsável pela lógica de negócios da

aplicação no lado do servidor. Nela encontram-se os EJBs que fazem a ligação entra a camada de serviço e a camada de modelo e que possuem as operações básicas de um CRUD além de algumas operações extras que se fazem necessárias. A última camada do servidor web é a de serviço, onde estão os RESTful *web services* que serão acessados pelo aplicativo móvel via URIs e que por meio delas executarão as operações da camada DAO.

Já o aplicativo móvel que foi desenvolvido por meio da utilização do *framework* Foundation for Apps que implementa o AngularJS, não possui uma divisão tão clara quanto a do servidor. Mesmo assim pode-se considerar que ele também possui três camadas importantes para o seu funcionamento, sendo elas: *views*, controladores e persistência de dados. As *views* podem ser consideradas as telas do aplicativo, onde o usuário interagirá com o sistema por meio de botões, formulários e outros meios. Elas são compostas por arquivos HTML, CSS e imagens que quando unidos montam a interface visual do aplicativo. Os controladores são responsáveis pela lógica de negócios do aplicativo e pela integração entre os *models* e as *views*. A camada de persistência de dados é onde foi criado o módulo responsável pela utilização do banco de dados IndexedDB no aplicativo. Na Figura 25 é mostrada a arquitetura do sistema por meio de um diagrama de componentes.

Figura 25: Diagrama de componentes da arquitetura do sistema



Fonte: Do autor

5.3 O SERVIDOR WEB

Nesta sessão serão apresentadas algumas informações sobre o desenvolvimento do servidor web que compõe o My Money, destacando-se as camadas que foram apresentadas na sessão anterior.

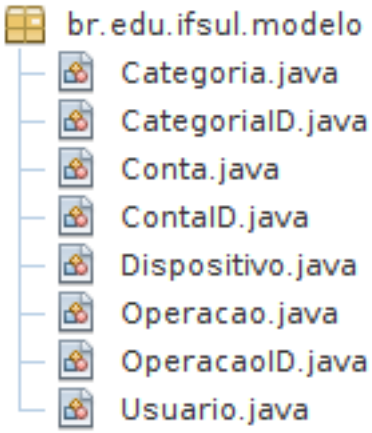
5.3.1 A Camada Modelo

A camada de Modelo é a responsável pelo mapeamento objeto-relacional dos dados que são enviados ao servidor. Esse mapeamento cria a correspondência entre os objetos e as tabelas do banco de dados, e seu trabalho é facilitado pela utilização da JPA. Sendo assim, nessa camada, são criadas as classes que de acordo com as anotações que possuírem se tornarão tabelas no banco de dados do servidor.

Estas classes que serão persistidas necessitam estar no padrão *JavaBeans*, ou seja, precisam implementar a classe *Serializable*, seus atributos devem estar encapsulados, precisam de um construtor sem parâmetros e todos seus atributos devem possuir os métodos *getters* e *setters*.

No caso do sistema My Money, a camada Modelo possui as classes Usuario, Dispositivo, Conta, ContaID, Categoria, CategoriaID, Operacao e OperacaoID como pode ser visto na Figura 26.

Figura 26: Classes da camada de Modelo.



Fonte: Do autor

A classe Usuario possui todos os atributos que um usuário do sistema necessitará para fazer login e utilizar o sistema, e além disso ela é a base para que os usuários possam sincronizar seus dados com o servidor web, indicando a qual usuário pertencem os dados sincronizados. Na Figura 27 é exibido parte do código da classe Usuario com os atributos e anotações necessários que fazem com que ela seja persistida no banco de dados.

Figura 27: Classe Usuario

```
 * @author Fernando Zonatto <zonattofernando@gmail.com>
 */
@Entity
@Table(name = "USUARIO")
public class Usuario implements Serializable {
    @Id
    @Column(name = "ID")
    @SequenceGenerator(name = "SEQ", sequenceName = "SEQ_USUARIO", allocationSize = 1)
    @GeneratedValue(generator = "SEQ", strategy = GenerationType.SEQUENCE)
    private Integer id;
    @Column(name = "NOME", length = 50, nullable = false)
    @Length(min = 3, max = 50, message = "O nome deve possuir entre {min} e {max} caracteres")
    @NotEmpty(message = "O nome deve ser preesnchido")
    private String nome;
    @Column(name = "EMAIL", length = 50, nullable = false, unique = true)
    @Email(message = "E-mail inválido")
    @NotEmpty(message = "O e-mail deve ser preenchido")
    private String email;
    @Column(name = "SENHA", length = 25, nullable = false)
    @Length(min = 6, max = 25, message = "A senha deve possuir entre {min} e {max} caracteres")
    @NotEmpty(message = "A senha deve ser preenchida")
    private String senha;
    @OneToMany(mappedBy = "usuario", cascade = {CascadeType.ALL},
            orphanRemoval = true, fetch = FetchType.EAGER)
    private List<Dispositivo> dispositivos = new ArrayList<>();

    //Construtor, Getters e Setters
```

Fonte: Do autor

A anotação *@Entity* determina que esta classe deva ser reconhecida como uma entidade e então se torne uma tabela no banco de dados. A anotação *@Table(name = "USUARIO")* é utilizada para definir o nome da tabela no banco. Com a anotação *@Id* define-se qual atributo será utilizado como chave primaria desta tabela no banco de dados. Como os usuários utilizadores do sistema serão criados diretamente no banco de dados do servidor web, é necessário que este identificador seja gerado automaticamente pelo provedor de persistência de dados. Isto é possível graças a anotação *@GeneratedValue,* onde foi escolhida a estratégia de geração como *GenerationType.SEQUENCE.* A sequência de geração foi criada pela anotação *@SequenceGenerator*, onde seu nome foi definido.

As colunas das tabelas foram personalizadas por meio das anotações *@Column*, as quais recebem como parâmetro o nome da coluna, o tamanho e se elas podem ou não serem nulas.

As *anotações @Length, @Email e @NotEmpty* pertencem a API do Hibernate Validator e indicam algumas restrições de integridade que os atributos possuem. Caso estas restrições não sejam respeitadas, a API emite mensagens de erro de acordo com as validações.

A anotação *@OneToMany* representa a relação de um para muitos existente entre as classes Usuario e Dispositivo, sendo que um usuário pode possuir muitos dispositivos e um dispositivo pertence a um único usuário. Esta anotação demonstra então que a classe Usuario possuirá uma lista de objetos da classe Dispositivo e esta lista será mapeada pelo atributo “usuário” existente dentro da classe Dispositivo. A opção *CascadeType.ALL* define que quando uma operação relacionada ao banco de dados for chamada na entidade Usuario ela será passada para a entidade Dispositivo também. O parâmetro *orphanRemoval = true* define que quando um usuário for removido, os seus dispositivos serão removidos também. A opção *fetch = FetchType.EAGER* indica que quando um objeto Usuario for instanciado, todos os seus dispositivos serão instanciados também.

As classes Conta, Categoria e Operação receberão os dados que o usuário deseja sincronizar e que já estão armazenados no seu dispositivo, sendo assim, eles já possuem uma chave primária que foi definida pelo banco de dados local do aplicativo. Para que não haja conflitos entre estes identificadores foi implementado um sistema de chaves primárias compostas. Elas são baseadas no identificador único dos usuários armazenados no banco de dados do servidor e nos próprios identificadores que elas já possuíam no banco de dados do dispositivo. Na JPA, este sistema de chaves é implementado por meio do uso das anotações *@Embeddable* e *@EmbeddedId*. Na Figura 28 pode ser vista a classe OperacaoID, responsável pela criação da chave primária composta que serve como identificador único da classe Operacao.

Figura 28: Classe OperacaoID

```
 * @author Fernando Zonatto <zonattofernando@gmail.com>
 */
@Embeddable
public class OperacaoID implements Serializable {
    @ManyToOne
    @JoinColumn(name = "ID_USUARIO", referencedColumnName = "ID", nullable = false)
    private Usuario usuario;
    @Column(name = "ID_OPERACAO", nullable = false)
    private Integer idOperacao;

    //Construtor, Getters e Setters
```

Fonte: Do autor

Nesta classe podemos ver a anotação *@Embeddable* que indica que esta classe poderá ser incorporada por outra. O atributo usuario possui as anotações *@ManyToOne* e *@JoinColumn(name = "ID_USUARIO", referencedColumnName = "ID", nullable = false)*, que representam o relacionamento de muitos para um entre as classes OperacaoID e Usuario. Sendo que a primeira anotação define o relacionamento e a segunda define o nome da coluna que será chave estrangeira na tabela Operacao e a qual coluna ela faz referência na tabela Usuario. O atributo idOperacao é o identificador que veio do banco de dados do dispositivo móvel.

Na Figura 29 é exibida a classe Operacao, que será persistida no banco de dados e possuirá chave primaria composta pelo ID do usuário e pelo ID vindo do dispositivo.

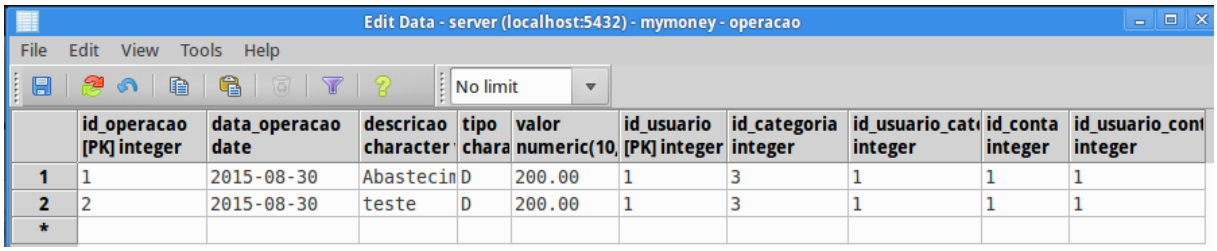
Figura 29: Classe Operacao

```
@Entity
@Table(name = "OPERACAO")
public class Operacao implements Serializable {
    @EmbeddedId
    private OperacaoID operacaoID;
    @Column(name = "VALOR", columnDefinition = "DECIMAL(10,2)", nullable = false)
    private Double valor;
    @Column(name = "DESCRICAO", length = 250, nullable = false)
    @Length(max = 250, message = "A descrição não pode possuir mais de 250 caracteres")
    @NotEmpty(message = "A descrição deve ser enviada")
    private String descricao;
    @Column(name = "DATA_OPERACAO", nullable = false)
    @Temporal(TemporalType.DATE)
    private Calendar data;
    @Column(name = "TIPO", nullable = false)
    private String tipo;
    @ManyToOne
    @JoinColumns(
        value = {
            @JoinColumn(name = "ID_USUARIO_CONTA", referencedColumnName = "ID_USUARIO",
                    nullable = false),
            @JoinColumn(name = "ID_CONTA", referencedColumnName = "ID_CONTA",
                    nullable = false)
        }
    )
    private Conta conta;
    @ManyToOne
    @JoinColumns(
        value = {
            @JoinColumn(name = "ID_USUARIO_CATEGORIA",
                    referencedColumnName = "ID_USUARIO", nullable = false),
            @JoinColumn(name = "ID_CATEGORIA", referencedColumnName = "ID_CATEGORIA",
                    nullable = false)
        }
    )
    private Categoria categoria;

    //Contrutor, Getters e Setters
```

Fonte: Do autor

Nesta classe o atributo operacaoID que é do tipo OperacaoID recebe a anotação @EmbeddedId. Isto significa que esta classe incorporará a classe OperacaoID e utilizará os atributos desta outra classe como identificador único, criando assim a chave primária composta que se faz necessária neste caso. Além disso, a classe Operacao possui os atributos conta e categoria que fazem referência a qual conta e categoria uma operação pertence. O relacionamento é definido pela anotação *@ManyToOne* e a anotação *@JoinColumns* define que essas chaves estrangeiras farão referência as chaves primarias compostas das tabelas Conta e Categoria. Na Figura 30 pode-se verificar como que ficou a estrutura física da tabela Operacao no banco de dados do servidor.

Figura 30: Tabela Operacao no banco de dados do servidor



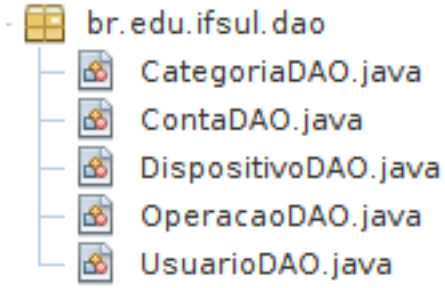
	id_operacao [PK] integer	data_operacao date	descricao character	tipo chara	valor numeric(10,	id_usuario [PK] integer	id_categoria integer	id_usuario_cat integer	id_conta integer	id_usuario_cont integer
1	1	2015-08-30	Abastecim	D	200.00	1	3	1	1	1
2	2	2015-08-30	teste	D	200.00	1	3	1	1	1
*										

Fonte: Do autor

5.3.2 A Camada DAO

No My Money, a camada DAO (Data Access Object) é a responsável por implementar a lógica funcional do servidor web da aplicação. É ela que permite o acesso aos dados persistidos no banco do servidor. Para a criação desta camada foram utilizados os EJBs, que garantem a separação da camada de persistência da camada de serviço e ainda adicionam gerenciamento de transações e segurança. Na Figura 31 pode-se observar as classes que fazem parte da camada DAO.

Figura 31: Classes da camada DAO



Fonte: Do autor

Todas as classes que fazem parte desta camada são *beans* de sessão que não guardam estado e que por natureza são transacionais. A principal função destas classes é executar os métodos de criação, leitura, atualização e exclusão (CRUD) nos dados persistidos por meio das classes da camada de Modelo. Essas classes então estarão todas empacotadas e distribuídas no servidor de aplicações GlassFish, permitindo assim que seus métodos CRUD sejam acessados pela camada de serviço do servidor por meio dos EJBs.

Para que as transações sejam executadas e as operações CRUD funcionem, o *bean* de sessão acessa o banco de dados por meio da fonte de dados

(jdbc/myMoney), a qual foi criada no servidor GlassFish e que está ligada ao banco de dados mymoney do PostgreSQL.

Na Figura 32 é exibido parte do código da classe UsuarioDAO.

Figura 32: Classe UsuarioDAO

```
 * @author Fernando Zonatto <zonattofernando@gmail.com>
 */
@Stateless
public class UsuarioDAO implements Serializable {
    @PersistenceContext(name = "MyMoneyPU")
    private EntityManager em;

    public UsuarioDAO() {
    }

    public void inserir(Usuario objeto) throws Exception {
        em.persist(objeto);
    }

    public void editar(Usuario objeto) throws Exception {
        em.merge(objeto);
    }

    public void deletar(Integer id) throws Exception {
        Usuario u = em.find(Usuario.class, id);
        u = em.merge(u);
        em.remove(u);
    }

    public Usuario buscarPorId(Integer id) throws Exception {
        return (Usuario) em.find(Usuario.class, id);
    }

    public List<Usuario> listarTodos() {
        return em.createQuery("SELECT u FROM Usuario u ORDER BY u.id").getResultList();
    }

    // Getters, setters e outros métodos
```

Fonte: Do autor

Nesta classe pode-se perceber a existência da anotação *@Stateless*, que indica que a classe é um *bean* de sessão que é incapaz de guardar estado e que qualquer cliente poderá acessar qualquer instância dos objetos desta classe. O gerenciamento destas instancias é executado automaticamente pelo *container* EJB. A anotação *@PersistenceContext* tem o objetivo de indicar ao EJB qual unidade de persistência que será utilizada. Neste caso será a persistência MyMoneyPU, sobre a qual foi explicada na sessão de preparação do ambiente de desenvolvimento. Nesta classe também é instanciado um objeto *EntityManager*, o qual é injetado diretamente no *bean* de sessão e por meio dele o EJB realiza suas transações. Por fim, a classe

possui os métodos que executam as operações CRUD por meio dos métodos nativos do *EntityManager*.

5.3.3 A Camada de Serviço

A camada de serviço tem o objetivo de ser a interface pela qual o aplicativo móvel My Money se comunicará com seu servidor web para persistência e sincronização de dados. Esta camada utilizará a tecnologia dos RESTful *web services* para conseguir realizar esta comunicação de uma maneira mais simples e elegante. Com um *web service* do tipo REST é possível que o aplicativo acesse os recursos do servidor e consiga realizar as operações necessárias apenas por meio das URLs e de métodos HTTP. O REST ainda permite que as informações trocadas entre o aplicativo e o servidor estejam na formatação JSON, o que facilita muito esta transação, pois o aplicativo utiliza a linguagem JavaScript, e JSON é a notação escrita dos objetos em desta linguagem. Na linguagem Java, os RESTful *web services* são gerenciados pela API Jax-RS e também são *beans* de sessão que não guardam estado.

Antes de entrar na parte do desenvolvimento das classes desta camada deve ser analisado um ponto muito importante, o aplicativo móvel utilizará AJAX para acessar o *web service* de maneira assíncrona e ele será executado dentro de um container web do dispositivo móvel. Sendo assim, a transação entre o aplicativo e o servidor será bloqueada pela política de *Cross-origin resource sharing* (CORS), ou compartilhamento de recursos de origem cruzada, que exige que o servidor envie alguns dados no cabeçalho de resposta da requisição HTTP para demonstrar que o servidor aceita que seus recursos sejam acessados por uma página externa ou por um domínio diferente. A solução para este problema está exatamente na edição do cabeçalho de resposta das requisições HTTP direcionadas a este web service. A API do Jax-RS permite esta edição e na Figura 33 é mostrada a classe que foi criada para resolver este problema.

Figura 33: Classe CORSFilter

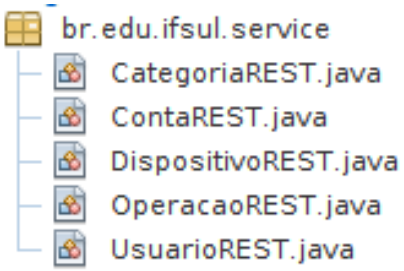
```
 * @author Fernando Zonatto <zonattofernando@gmail.com>
 */
@Provider
public class CORSFilter implements ContainerResponseFilter {

    @Override
    public void filter(final ContainerRequestContext requestContext,
                       final ContainerResponseContext cres) throws IOException {
        cres.getHeaders().add("Access-Control-Allow-Origin", "*");
        cres.getHeaders().add("Access-Control-Allow-Headers", "origin, "
                + "content-type, accept, authorization");
        cres.getHeaders().add("Access-Control-Allow-Credentials", "true");
        cres.getHeaders().add("Access-Control-Allow-Methods", "GET, POST, PUT, "
                + "DELETE, OPTIONS, HEAD");
        cres.getHeaders().add("Access-Control-Max-Age", "1209600");
    }

}
```

Fonte: Do autor

A classe CORSFilter possui uma anotação *@Provider*, que indica que esta classe poderá customizar o *runtime* do Jax-RS, neste caso a customização realizada será a do cabeçalho de resposta das requisições HTTP. É estendida a classe *ContainerResponseFilter*, responsável pelas respostas das requisições HTTP, e na sobrescrita do método *filter* são adicionados os dados necessários ao cabeçalho desta reposta. Assim é possível que o aplicativo acesse o *web service* sem que haja problema com a política CORS.

Na Figura 34 podem-se visualizar as classes que fazem parte da camada de Serviço.

Figura 34: Classes da camada Serviço



Fonte: Do autor

Para demonstrar a criação e o funcionamento de um serviço, a Figura 35 exibe parte do código da classe UsuarioREST, responsável por transformar as

operações da classe UsuarioDAO em recursos que podem ser acessados pelo aplicativo móvel via URL.

Figura 35: Classe UsuarioREST

```
@Stateless
@Path("usuario")
public class UsuarioREST {
    @EJB
    private UsuarioDAO dao;

    public UsuarioREST() {
    }

    @POST
    @Consumes("application/json")
    @Produces("application/json")
    // JSON p/ inserir = {"email":"zonattofernando@gmail.com","nome":"Fernando","senha":'
    public Mensagem insert(Usuario objeto) {
        Mensagem m;
        try {
            dao.inserir(objeto);
            m = new Mensagem(0, "Usuário criado com sucesso!");
            m.setUsuario(objeto);
        } catch (Exception e) {
            m = new Mensagem(1, "Erro ao criar usuário! ", Mensagem.getMensagemErro(e));
            objeto = null;
        }
        return m;
    }

    @GET
    @Produces("application/json")
    public List<Usuario> findAll() {...3 linhas }

    @PUT
    @Consumes({"application/json"})
    @Produces("application/json")
    public Mensagem edit(Usuario objeto) {...8 linhas }

    @DELETE
    @Path("{id}")
    public Mensagem remove(@PathParam("id") Integer id) {...8 linhas }
```

Fonte: Do autor

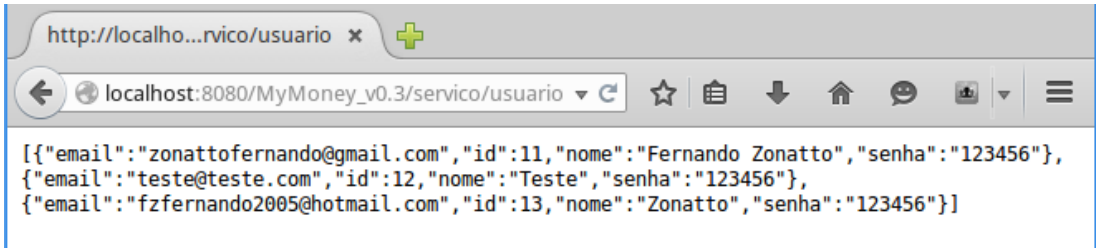
Percebe-se que um serviço do tipo REST também é um *bean* de sessão que não guarda estado, pois nessa classe utiliza-se a anotação *@Stateless.* Este serviço é definido pela utilização da anotação *@Path*, pertencente ao pacote *javax.ws.rs.* Esta anotação recebe como parâmetro o caminho pelo qual o serviço será acessado pela URL. Esta classe possui um atributo do tipo UsuarioDAO que é injetado com referência ao EJB, por meio da anotação *@EJB*. O método *insert* é o responsável pela criação de usuários no servidor, ele possui a anotação *@POST*, que representa que ele será executado quando a URL do serviço for chamada pelo método POST do HTTP. A anotação *@Consumes("application/json")* indica que o objeto do tipo

Usuario que é recebido por parâmetro nesse método deve vir do aplicativo no formato JSON, como pode ser visto no comentário do código. E a anotação *@Produces("application/json")* indica que a resposta do servidor será um objeto do tipo Mensagem também no formato JSON.

Esta classe Mensagem é uma classe auxiliar, criada para exibir as mensagens de resposta que o servidor envia ao aplicativo, ela possui um atributo chamado *status,* que recebe o valor 0 se houver sucesso na requisição ou 1 se houver erro, além da mensagem que será exibida ao usuário e do objeto Usuario que acabou de ser persistido no servidor. A classe UsuarioREST ainda apresenta outros métodos e outras anotações, como as *@PUT, @DELETE e @GET,* que representam os outros métodos de uma requisição HTTP. Além disso a anotação *@Path* é utilizada em alguns métodos, o que indica que URLs específicas são necessárias para acesso aos mesmos. Dentro de cada método desta classe é invocado um objeto UsuarioDAO e chamado o método referente a operação desejada.

Na Figura 36 é mostrado o resultado que o navegador exibe ao se acessar a URL do serviço de usuários. É exibida a lista de usuários do servidor, pois ao ser acessada pelo método GET do HTTP, essa URL invoca o método *findAll* da classe UsuarioREST.

Figura 36: Acesso ao serviço pelo navegador



Fonte: Do autor

5.4 O APLICATIVO MÓVEL

Nesta sessão serão apresentadas algumas informações sobre o desenvolvimento do aplicativo móvel que compõe o sistema My Money, destacando-se alguns pontos importantes que fazem parte dele. Ele foi desenvolvido utilizando-se as linguagens HTML, CSS e JavaScript .

5.4.1 As Views

Como o aplicativo móvel My Money foi desenvolvido utilizando-se o *framework* Foundation for Apps, ele tem toda a sua funcionalidade baseada nos recursos provenientes de outro *framework*, o AngularJS. As *views* de uma aplicação AngularJS são templates HTML que possuem como grande diferencial a utilização de diretivas (tags, classes, css ou comentários) específicas do *framework* para poderem se comunicar com a camada de controle e modelo da aplicação. Na Figura 37 pode ser visto o código do arquivo *index.html,* o qual foi criado automaticamente pelo Foundation for Apps, e é a *view* principal do aplicativo e onde todas as outras *views* serão inseridas.

Figura 37: Arquivo index.html do aplicativo

```
<!doctype html>
<html lang="pt-BR" ng-app="application">
    <head>
        <meta charset="utf-8" />
        <meta name="viewport" content="width=device-width, initial-scale=1.0" />
        <title>My Money</title>
        <link href="./assets/css/app.css" rel="stylesheet" type="text/css">
        <script src="./assets/js/foundation.js"></script>
        <script src="./assets/js/templates.js"></script>
        <script src="./assets/js/routes.js"></script>
        <script src="./assets/js/idbstore.min.js"></script>
        <script src="./assets/js/app.js"></script>
        <script src="./assets/js/controller.js"></script>
        <script src="./assets/js/db.js"></script>
    </head>
    <body>
        <div class="grid-frame vertical">
            <div class="grid-content shrink" style="padding: 0;">
                <ul class="primary condense menu-bar">
                    <li><a><strong>My Money</strong></a></li>
                </ul>
            </div>
            <div ui-view class="grid-content">
            </div>
        </div>
    </body>
</html>
```

Fonte: Do autor

Neste arquivo, o principal a se observar é a diretiva *ng-app,* situada na *tag html,* que indica que esta é uma aplicação AngularJS e que recebe o nome do módulo principal da aplicação. No arquivo também é referenciado o arquivo css que fará a estilização de todas as telas do aplicativo e os arquivos JavaScript necessários para o projeto. Os arquivos foundation.js, templates.js, routes.js e app.js

são criados automaticamente pelo Foundation for Apps, já os outros foram inseridos e possuem algumas funcionalidades importantes do aplicativo que serão explicadas nas próximas sessões. No interior da *tag body* do arquivo estão os códigos responsáveis pela criação do menu superior do aplicativo, que será exibido em todas as telas, mas pode-se destacar a *div* que possui a diretiva *ui-view*, pois é dentro dela que as outras *views* do aplicativo serão exibidas de maneira assíncrona e sem *refresh* na página.

Para exemplificar um pouco mais a fundo o funcionamento das *views*, a Figura 38 apresenta o código da *view formOperacao*, responsável pela inserção de uma operação financeira no banco de dados do aplicativo.

Figura 38: *View* do formulário de inserção de operações

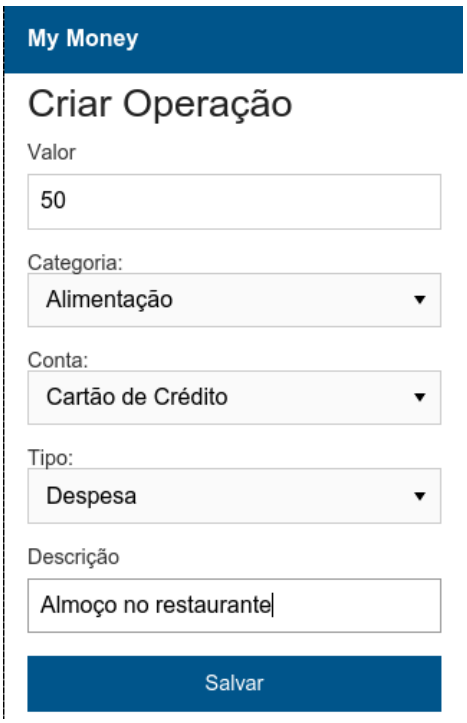
```
<div class="grid-container">
    <h2>Criar Operação</h2>
    <form name="criarOperacaoForm"
          ng-submit="criarOperacaoForm.$valid && inserir(obj)"
          novalidate>
        <label>
            Valor
            <input type="number" ng-model="obj.valor" required>
        </label>
        <label for="categoria">Categoria:
            <select id="categoria"
                ng-options="categoria as categoria.nome for categoria in categorias"
                ng-model="obj.categoria" required>
            </select>
        </label>
        <label for="conta">Conta:
            <select id="conta"
                ng-options="conta as conta.nome for conta in contas"
                ng-model="obj.conta" required>
            </select>
        </label>
        <label for="tipo">Tipo:
            <select id="tipo" ng-model="obj.tipo" required>
                <option value="R">Receita</option>
                <option value="D">Despesa</option>
            </select>
        </label>
        <label>
            Descrição
            <input type="text" ng-model="obj.descricao" required>
        </label>
        <button class="expand button">Salvar</button>
        <span class="alert label" ng-show="erro">{{msg}}</span>
        <span class="success label" ng-show="sucesso">{{msg}}</span>
    </form>
</div>
```

Fonte: Do autor

Nesta *view* podemos observar um formulário que recebeu o nome de *criarOperacaoForm.* Dentro da *tag form* há a diretiva *ng-submit*, que é responsável por executar as operações necessárias no momento em que o formulário for

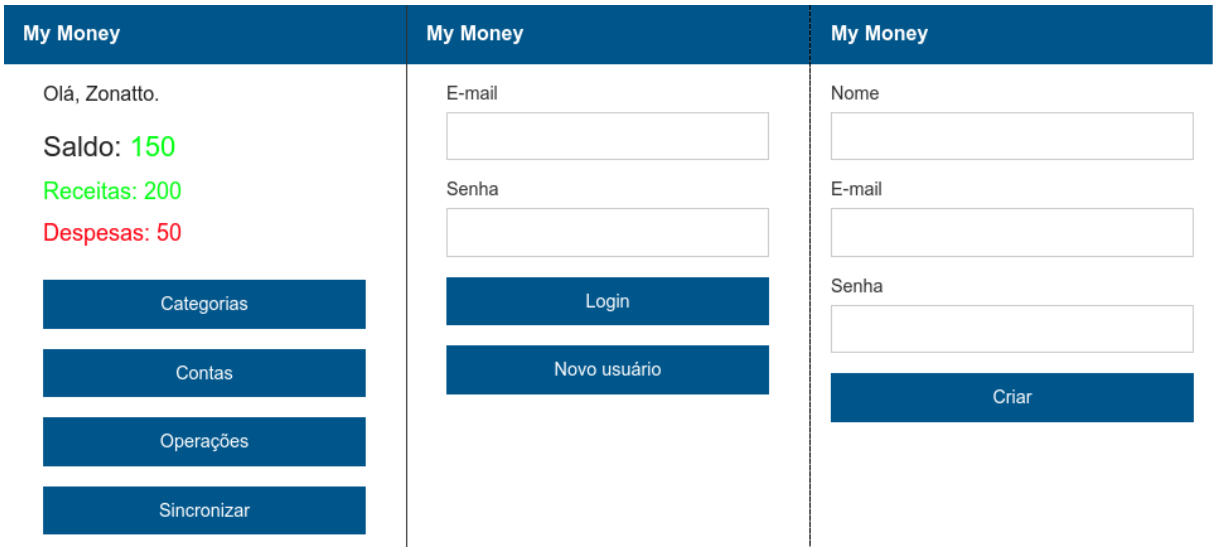
submetido. Neste caso são executadas duas operações, a validação dos campos e a inserção dos valores no banco. A validação dos campos do formulário é chamada por meio da função *$valid*, nativa do AngularJS, e verifica se os campos do formulário *criarOperacaoForm* foram preenchidos, retornando *true* se a resposta for positiva. A inserção fica a cargo da função inserir(), que está implementada dentro do controlador acoplado a esta *view*. O formulário possui dois *selects* que são preenchidos dinamicamente de acordo com as categorias e contas cadastradas no aplicativo. Isto é feito por meio da diretiva *ng-options,* que percorre da mesma maneira que um *foreach* os *arrays* de categorias e contas presentes no controlador desta *view*. A diretiva *ng-model*, presente em todos os campos do formulário, é responsável pela criação do modelo *obj,* que é o objeto em JavaScript que será enviado como parâmetro na função inserir() para ser persistido no banco de dados do aplicativo. Na Figura 39 é exibida a tela de criação de operações, que é o resultado visual da *view formOperacao*. E a Figura 40 mostra algumas outras *views* importantes do aplicativo.

Figura 39: Tela de criação de operações financeiras



Fonte: Do autor

Figura 40: Tela principal, *login* e criação de novo usuário



Fonte: Do autor

5.4.2 Os Controladores

Como já foi discutido anteriormente, os controladores do AngularJS são responsáveis pela lógica de negócios do aplicativo e pela integração entre os *models*, as *views* e outros componentes da aplicação. Eles são totalmente escritos em JavaScript e no caso do aplicativo My Money, se encontram dentro do arquivo *controller.js*, que foi referenciado no *index.html* do aplicativo como pôde ser visto na sessão anterior.

É muito comum no AngularJS, que exista um controlador acoplado a cada *view* criada, mas também é possível que um controlador seja utilizado por mais de uma *view*, ou uma *view* possa utilizar mais de um controle. Os controladores são usualmente acoplados as views por meio da diretiva *ng-controller* que pode ser utilizada dentro de qualquer *tag* HTML de uma *view* do AngularJS. Mas no caso da utilização do Foundation for Apps, um arquivo chamado de *routes.js* é designado para fazer a ligação entre as *views* e controladores, além de algumas outras funções. A Figura 41 exibe o arquivo *routes.js*, lembrando que ele também é referenciado no *index.html* desta aplicação.

Figura 41: Arquivo routes.js

```
var foundationRoutes = [
    {"name":"index","url":"/", "controller": "controleAbrir"},

    {"name":"home","url":"/home","path":"templates/home.html",
        "controller": "controleLogin"},

    {"name":"listar", "url":"/listar","path":"templates/listar.html",
    "animationIn": "slideInRight"},

    {"name":"novoUsuario", "url":"/novo-usuario",
        "path":"templates/novo-usuario.html", "animationIn": "slideInRight",
        "controller": "controleUsuario"},

    {"name":"private", "url":"/private","path":"templates/private/home.html",
        "animationIn": "slideInRight", "controller": "controlePrivate"},

    {"name":"formCategoria", "url":"/private/categoria/form",
        "path":"templates/private/categoria/form.html",
        "animationIn": "slideInRight", "controller": "controleCategoria"},

    {"name":"formConta", "url":"/private/conta/form",
        "path":"templates/private/conta/form.html", "animationIn": "slideInRight",
        "controller": "controleConta"},

    {"name":"formOperacao", "url":"/private/operacao/form",
        "path":"templates/private/operacao/form.html",
        "animationIn": "slideInRight", "controller": "controleOperacao"}
];
```

Fonte: Do autor

Pode-se verificar que este arquivo possui um *array* chamado de *foundationRoutes*, onde se encontram objetos em JavaScript responsáveis cada um por uma *view*. Estes objetos possuem como atributos o nome da *view*, a URL de cada uma, o caminho físico delas e os controladores acoplados a elas, além de ser possível definir efeitos de transição de tela para as *views*.

Um controlador é definido utilizando-se o método *controller* do AngularJS. Dentro deste método é definido o nome do controlador e uma função que será executada por ele, como podemos visualizar na Figura 42.

Figura 42: Definição dos controles da aplicação no arquivo *controller.js*

```
1  (function() {
2      var app = angular.module('application');
3
4      /*Controladores*/
5      app.controller('controleAbrir', AbrirCtrl);
6      app.controller('controleUsuario', UsuarioCtrl);
7      app.controller('controleLogin', LoginCtrl);
8      app.controller('controleCategoria', CategoriaCtrl);
9      app.controller('controleConta', ContaCtrl);
10     app.controller('controleOperacao', OperacaoCtrl);
11     app.controller('controlePrivate', PrivateCtrl);
12
13     /*Implementação dos controladores*/
14
```

Fonte: Do autor

A variável *app* é criada e recebe como valor o módulo principal da aplicação. Logo abaixo são definidos os controladores por meio do método *controller* chamado dentro da variável *app*. São passados como parâmetro deste método o nome do controlador que será utilizado no arquivo *routes.js,* e a função que o controlador executará e que se encontra neste mesmo arquivo *controller.js.*

Para exemplificar a funcionalidade de um controlador, é exibido na Figura 43, o código do *controleUsuario*, responsável pela criação de um usuário da aplicação.

Figura 43: Controle Usuário

```
/*Controle criação de Usuário*/
UsuarioCtrl.$inject = ['$scope', '$stateParams', '$state', '$controller', '$http', '$location', 'bancoSrv'];
function UsuarioCtrl($scope, $stateParams, $state, $controller, $http, $location, bancoSrv) {
    angular.extend(this, $controller('DefaultController',
        {$scope: $scope, $stateParams: $stateParams, $state: $state}));
    /*Função de criação de usuário*/
    $scope.criarUsuario = function (usuario) {
        /*Registra usuário no servidor*/
        $http.post('http://localhost:8080/MyMoney_v0.3/servico/usuario', usuario)
            .success(function(data) {
                if (data.status == 0) {
                    sucessoCallback(data.msg);
                    /*Limpar bd local*/
                    bancoSrv.usuarios.clear();
                    bancoSrv.categorias.clear();
                    bancoSrv.contas.clear();
                    bancoSrv.operacoes.clear();
                    /*Gravar usuário no bd local*/
                    bancoSrv.usuarios.put(data.usuario, onSuccess, onError);

                    /*Registra dispositivo no servidor*/
                    var dispositivo = {nome: "Android", status: 1, usuario: data.usuario};
                    $http.post('http://localhost:8080/MyMoney_v0.3/servico/dispositivo', dispositivo)
                        .success(function(data) {
                            if (data.status == 0) {
                                /*Redireciona para tela principal do aplicativo*/
                                $location.path('/private');
                            } else {
                                erroCallback(data.msg);
                            }
                        })
                        .error(function() { erroCallback("Erro ao criar dispositivo!"); });
                } else { erroCallback(data.msg); }
            })
            .error(function() { erroCallback("Erro ao criar usuário!"); });
    }
```

Fonte: Do autor

Este controlador é um dos mais completos do aplicativo, pois ele tem o objetivo de pegar os dados do formulário de criação de usuários e enviá-los ao servidor para que sejam persistidos no banco de dados remoto, e com a resposta do servidor, armazenar este usuário também no banco de dados local.

Antes da implementação da função *UsuarioCtrl* é necessária a injeção das dependências que serão utilizadas por ela, prática comum no AngularJS. É utilizado então o objeto *$inject*, que recebe como parâmetro os objetos das dependências

necessárias. Podemos destacar entre estas dependências o objeto $scope, que representa o modelo a ser utilizado pela view acoplada a este controlador, ou seja, tudo que for adicionado ao modelo por meio do comando *$scope.funcaoOuVariavel*, poderá ser requisitado na *view* por meio de diretivas ou *data bindings.*

Em seguida a função *UsuarioCtrl* é implementada e recebe por parâmetro as dependências injetadas. Esta função então estende o *DefaultController*, do AngularJS, passo que é recomendado pela documentação do *framework.* Após tudo isso, o comando *$scope.criarUsuario* define a função que será chamada pela *view* e a insere no objeto *$scope*. Dentro desta função então é executada a requisição *ajax* ao servidor por meio do comando *$http.post,* que envia uma requisição do tipo POST ao *web service*, passando por parâmetro a URL do serviço e o objeto *usuario* criado pelo formulário da *view novoUsuario*. O método *.success()* é executado se houver resposta positiva do servidor web. Dentro dele é verificado o valor do atributo *status,* que foi criado como *flag* no servidor, para verificar se houve ou não erro na execução do método invocado. Se o valor dele for igual a 0, houve sucesso, e então é exibida a mensagem de sucesso na tela por meio da função *sucessoCallback(data.msg)*.

Após isso todos os dados salvos no banco de dados local são apagados pelas funções *bancoSrv.nomeDoBanco.clear()* e o usuário inserido no banco de dados do servidor também é inserido no banco de dados local por meio da função *bancoSrv.usuarios.put*(). O dispositivo pelo qual o usuário foi criado também é enviado ao servidor. Neste caso, como o aplicativo está rodando em um navegador web, foi simplesmente criado um objeto *dispositivo* para ser enviado ao servidor com os atributos necessários. Por fim, se o dispositivo foi persistido com sucesso no servidor, o aplicativo é redirecionado para a tela inicial do usuário por meio do comando *$location.path('/private').*

Os outros controladores existentes no arquivo *controller.js* são referentes as funções de *login* no aplicativo, persistência de categorias, contas e operações, e exibição de saldo do usuário.

5.4.3 A persistência local de dados

Os dados criados no aplicativo são persistidos localmente por meio da API de armazenamento de dados em navegadores *web* chamada de IndexedDB. O

IndexedDB trabalha de maneira diferente dos outros bancos de dados de navegador e não se assemelha nem um pouco a qualquer banco de dados relacional. Sua estrutura não é fixa e nem organizada em tabelas, e ele não utiliza nenhuma linguagem de queries para consulta de seus dados. Ele basicamente armazena objetos JavaScript e os identifica por índices, sendo que estes objetos não precisam ter a mesma estrutura e os índices podem ser de tipos variados de dados. Este tipo de armazenamento é chamado de *ObjectStore*.

Uma implementação crua de armazenamento no IndexedDB, partindo-se da estaca zero, pode ser um tanto complicada, demandando muito tempo para estudo da ferramenta e muitas linhas de código para sua implementação. Para facilitar o uso desta tecnologia, no desenvolvimento desta aplicação, foi utilizada uma biblioteca em JavaScript chamada de IDBWrapper[5], que tem como objetivo abstrair e tornar mais fácil o acesso aos recursos do IndexedDB.

Para a implementação desta biblioteca auxiliar foi necessário o *download* do arquivo *idbstore.min.js*, que foi colocado junto ao projeto e referenciado no *index.html* do aplicativo. Também foi necessária a criação de um módulo que implementasse a criação das *ObjectStores* referentes as categorias, contas, operações e ao usuário que serão armazenados. A Figura 44 mostra o código do arquivo *db.js*, onde está implementado este módulo.

Figura 44: Arquivo *db.js*

```
(function(){
    /**
    *  Module Banco de dados
    *
    *  Referente ao banco de dados IndexedDB e a biblioteca IDBWrapper
    */
    var modulo = angular.module('bancoDeDados', []);

    /*Criação da Factory*/
    modulo.factory('bancoSrv', function(){
        return {
            usuarios:   osUsuarios,
            categorias: osCategorias,
            contas:     osContas,
            operacoes:  osOperacoes
        };
    });

    /*Criação das ObjectStores*/
```

Fonte: Do autor

[5] Disponível em: <https://github.com/jensarps/IDBWrapper> Acesso em setembro de 2015.

O módulo *bancoDeDados,* foi criado por meio do método *.module(),* e logo após foi criado o serviço *bancoSrv* que torna possível o acesso a este módulo. Este serviço é uma *factory* que tem como retorno o acesso as ObjectStores que foram criadas neste mesmo arquivo. A Figura 45 exibe o código de criação da *ObjectStore osOperacoes*, responsável pela persistência das operações no banco de dados local.

Figura 45: Criação de uma *ObjectStore*

```
/*Criação da Object Store de operações*/
var osOperacoes = new IDBStore({
    dbVersion: 1,
    storeName: 'operacao',
    keyPath: 'id',
    autoIncrement: true,
    onStoreReady: function(){
        console.log('Store ready!');
    }
});
```

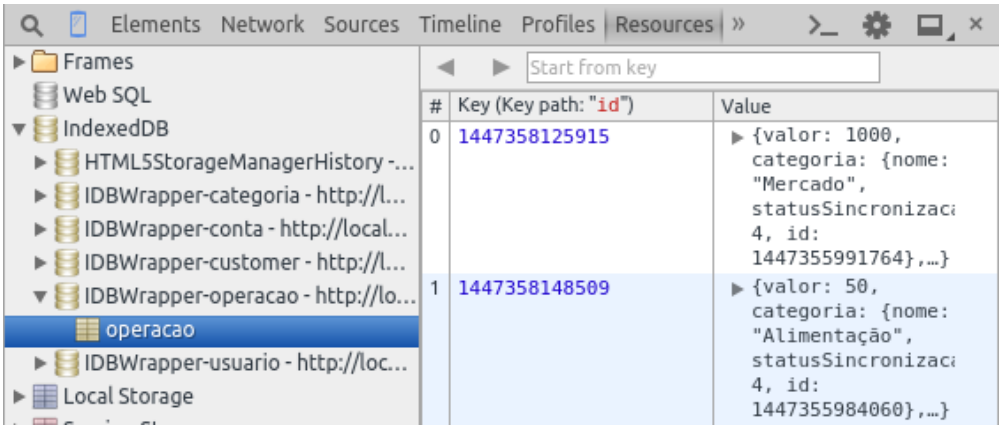
Fonte: Do autor

Pode-se perceber pelo código, que a implementação de uma *ObjectStore* utilizando-se o IDBWrapper é simples. É necessário somente que se instancie um novo objeto IDBStore e se utilizem os atributos necessários. Neste caso foi utilizada uma implementação simples, onde é passado a versão do banco de dados, o nome da *store* e o índice chave, também sendo indicado que ele é incrementado automaticamente pela própria API.

Após uma IDBStore ser criada, será possível acessá-la por meio dos seguintes métodos: *put, get, getAll, remove e clear*. O método *put* é responsável pela inserção e edição de dados de uma *ObjectStore.* O *get* recupera um objeto único de uma *ObjectStore* por meio de seu *keyPath* e o *getAll* recupera todos os dados. O *remove*, exclui um dado por meio de seu *keyPath* e o *clear* exclui todos os dados de uma *ObjectStore*. Para a utilização destes métodos nesta aplicação, o serviço *bancoSrv* foi injetado como dependência nos controladores necessários e os métodos chamados através dele, como pôde ser visto na Figura 43 que exibiu o controle de usuários do aplicativo.

A Figura 46 mostra as *ObjectStores* criadas nesta aplicação e que podem ser vistas na aba *Resources* do Chrome DevTools do navegador Google Chrome.

Figura 46: *ObjectStores* do aplicativo My Money



Fonte: Do autor

5.4.4 A sincronização de dados com o servidor

A sincronização de dados entre o aplicativo móvel e o servidor é um dos pontos fortes da arquitetura do sistema My Money, pois ela permite que os usuários salvem seus dados remotamente e possam acessá-los de um outro dispositivo móvel em uma outra ocasião.

Para efetuar a sincronização destes dados, as informações armazenadas no dispositivo possuem uma *flag* chamada de *statusSincronizacao*, e de acordo com esta *flag* é possível saber se o dado foi inserido, editado ou removido do banco de dados da aplicação local.

A lógica de funcionamento deste procedimento é um tanto quanto simples. Na tela principal do aplicativo existe um botão chamado sincronização, que ao ser acionado, executa uma função no controlador desta tela. Esta função por sua vez tem o objetivo de buscar todos os dados das *ObjectStores* referentes as categorias, contas e operações e verificar qual é o status de sincronização destes objetos. Então, de acordo com este status, são executadas requisições ao servidor que irão inserir, editar ou remover dados do banco de dados remoto. Na Figura 47 pode ser vista a função que executa a sincronização de dados.

Figura 47: Função de sincronização de dados

```
/*Função de sincronização com o servidor*/
$scope.sincronizar = function() {
    /*Busca o usuário do app*/
    bancoSrv.usuarios.getAll(function(data) {
        var usuario = data[0];
        /*Busca categorias do banco de dados local*/
        bancoSrv.categorias.getAll(function(data) {
            data.forEach(function(obj, index) {
                /*Cria objeto categoria que será enviado ao servidor*/
                /* Ex.: {"categoriaID":{
                            "idCategoria":1447355991764,
                            "usuario":{
                                "email":"fzfernando2005@hotmail.com",
                                "id":13,"nome":"Zonatto","senha":"123456"}
                            },
                        "nome":"Mercado"
                        }*/
                var categoria = {};
                categoria.categoriaID = {};
                categoria.categoriaID.idCategoria = obj.id;
                categoria.categoriaID.usuario = usuario;
                categoria.nome = obj.nome;

                if (obj.statusSincronizacao == 1) {...
                } else if (obj.statusSincronizacao == 2) {...
                } else if (obj.statusSincronizacao == 3) {...
                }
            });
        }, onError);

        bancoSrv.contas.getAll(function(data) {...
        }, onError);

        bancoSrv.operacoes.getAll(function(data) {...
        }, onError);
    }, onError);
}
```

Fonte: Do autor

Primeiramente a função sincronizar é adicionada ao objeto *$scope*, fazendo parte então do modelo que será utilizado nesta *view*. A primeira ação da função é buscar o usuário que está cadastrado no banco de dados do aplicativo por meio do método *getAll* e armazenar o objeto recuperado na variável *usuário.* Após isso, o método *getAll* do IDBWrapper também será utilizado para que se busquem todas as categorias, contas e operações registradas no aplicativo. Na imagem, foi exibido como exemplo, parte do código de sincronização de categorias. Após elas serem recuperadas, o *array data*, que é o resultado da busca, será percorrido por um *forEach.* Para cada objeto percorrido pelo *forEach*, é criado um novo objeto JavaScript com os dados recuperados e com a estrutura em que ele possa ser reconhecido pelo *web service*. Após termos os objetos no formato correto, pode-se

observar na Figura 48 o código que executa a comunicação com o servidor de acordo com o status de sincronização de cada objeto recuperado.

Figura 48: Requisições de sincronia com o servidor

```
if (obj.statusSincronizacao == 1) {
    $http.post('http://localhost:8080/MyMoney_v0.3/servico/categoria', categoria)
        .success(function(result) {
            if (result.status == 0) {
                obj.statusSincronizacao = 4;
                bancoSrv.categorias.put(obj, onSuccess, onError);
                console.log("Sincronizado com sucesso!");
            }
        }).error(function(e) {
            console.log("erro "+e);
        });
} else if (obj.statusSincronizacao == 2) {
    $http.put('http://localhost:8080/MyMoney_v0.3/servico/categoria', categoria)
        .success(function(result) {
            if (result.status == 0) {
                obj.statusSincronizacao = 4;
                bancoSrv.categorias.put(obj, onSuccess, onError);
                console.log("Sincronizado com sucesso!");
            }
        }).error(function(e) {
            console.log("erro "+e);
        });
} else if (obj.statusSincronizacao == 3) {
    $http.delete('http://localhost:8080/MyMoney_v0.3/servico/categoria/', categoria)
        .success(function(result) {
            if (result.status == 0) {
                bancoSrv.categorias.remove(obj, onSuccess, onError);
                console.log("Removido cm sucesso!");
            }
        })
        .error(function(e) {
            console.log("erro "+e);
        });
}
```

Fonte: Do autor

Pode-se visualizar na imagem que é criada uma estrutura condicional que verifica o atributo *statusSincronizacao* dos objetos recuperados e a partir daí são feitas as requisições ao servidor web. Se o status for igual a 1, significa que este objeto foi criado no aplicativo, então é feita uma requisição com o método *$http.post* ao web service e este objeto é então persistido no banco de dados remoto. É feita então uma verificação para confirmar se o objeto foi persistido com sucesso no servidor, e então o status de sincronização dele recebe o valor 4 e ele é editado no banco de dados local por meio do método *put* do IDBWrapper. Se o status for igual a 2, a requisição feita ao servidor é do tipo *$http.put*, para que o objeto seja alterado no banco de dados remoto. E se o status for igual a 3, é feita uma requisição do tipo *$http.delete*, que remove este dado do servidor e logo após receber a resposta positiva da operação remota, este dado também é removido do banco de dados local por meio do método *remove* do IDBWrapper.

CONSIDERAÇÕES FINAIS

O presente estudo teve como objetivo o desenvolvimento de um aplicativo móvel para controle financeiro de seus usuários. Este tipo de aplicação é muito utilizado nos dias de hoje, sendo uma boa área para se estudar e aplicar os conhecimentos adquiridos.

A aplicação apresentada neste estudo poderia ter sido desenvolvida de inúmeras maneiras e com os mais variados paradigmas de programação. Mas neste caso optou-se pela criação de uma arquitetura distribuída, baseada no modelo cliente-servidor, de modo que ela trouxesse funcionalidades úteis e interessantes à aplicação.

Para auxiliar a criação desta arquitetura foi feito um grande trabalho de modelagem, que abrangeu diversos diagramas e teve o objetivo de entregar uma visão de como o sistema ficaria, mesmo antes dele começar a ser implementado.

O desenvolvimento desta arquitetura também abrangeu diversas tecnologias. Elas vão desde a utilização da plataforma Java EE no lado do servidor, com o auxílio da API JPA, dos *containers* EJB, da JTA e do provedor de persistência Hibernate, até a utilização de *frameworks* como o Foudation for Apps e o AngularJS, além da API de armazenamento de dados IndexedDB no lado do aplicativo móvel. Esta grande quantidade de tecnologias, que as vezes se diferem muito umas das outras, provocou a necessidade de estudo de seu funcionamento, mesmo com o curto período de tempo para realização do projeto. Em algumas delas, observou-se somente o necessário para que fossem implementadas nesta aplicação. Mas com certeza elas serão estudadas mais a fundo para projetos futuros.

No início do projeto foram encontradas algumas dificuldades, principalmente em relação a criação do recurso e do *pool* de conexões necessários para a utilização da JPA e do EJB no servidor GlassFish. Mas após os problemas serem sanados, essas duas tecnologias auxiliaram muito na produtividade do desenvolvimento do servidor. O aplicativo móvel foi desenvolvido sem que houvessem problemas graves, o maior empecilho foi o tempo gasto com o estudo das novas tecnologias, como o IndexedDB e o AngularJS principalmente, o que acarretou na diminuição do tempo para desenvolvimento do aplicativo.

A versão do aplicativo apresentada neste estudo possui as funcionalidades básicas necessárias em um aplicativo de controle financeiro. Nele é possível que o

usuário mantenha suas operações financeiras, as categorias e contas a que elas pertencem, e além disso, exiba seu saldo na tela. As telas do aplicativo foram construídas da maneira mais simples, utilizando-se do layout padrão que o Foundation for Apps entrega, apenas para que a funcionalidade do aplicativo fosse testada. Como trabalhos futuros, pretende-se melhorar o layout das telas e deixá-lo mais intuitivo e agradável. Além disso, pretende-se implementar as visualizações de saldos e extratos por períodos específicos e até alguns gráficos que consigam fazer com que o usuário tenha maior controle sobre suas finanças.

No quesito da arquitetura, a grande maioria das funcionalidades propostas foram implementadas com sucesso. A criação de usuários diretamente no servidor e a sincronização de dados funcionam exatamente como foram propostas. O sistema de login também está funcional, de maneira que no momento em que o usuário efetua login no sistema, ele tem todos os seus dados baixados para seu dispositivo. A única funcionalidade que não foi implementada, foi a do controle de dispositivos do usuário, que indica por qual dispositivo ele está acessando e impede de que ele use mais de um dispositivo ao mesmo tempo e assim desorganize o sistema de sincronização. Esta funcionalidade também será implementada em trabalhos futuros.

A questão da segurança do RESTful *web service* não foi abordada, sendo assim, qualquer um teria acesso livre ao serviço. Esta é outra questão a ser estudada em trabalhos futuros.

Por fim, não houve tempo suficiente para que se utilizasse o PhoneGap, e se exportasse o aplicativo para ser utilizado fisicamente em um dispositivo. Esta tarefa não foi necessária no momento do desenvolvimento, pois o aplicativo não utiliza nenhum recurso nativo dos dispositivos e seu teste pôde ser feito por meio de um navegador web. Mesmo assim seu funcionamento foi estudado, e futuramente, quando o aplicativo estiver pronto para ser utilizado pelo público, ele será exportado e se tornará um aplicativo multiplataforma, podendo até ser colocado nas lojas virtuais de aplicativos.

REFERÊNCIAS

ANGULARJS. **Developer Guide.** Disponível em: <https://docs.angularjs.org/guide>. Acesso em 24 de outubro de 2015.

BAUER, Christian; KING, Gavin. **Java persistence com hibernate.** Rio de Janeiro, RJ: Ciência Moderna, 2007.

DEITEL, Paul; DEITEL, Harvey. **Java: como programar. 8. ed.** São Paulo: Pearson, 2010.

GHATOL, Rohit; PATEL, Yogesh. **Beginning PhoneGap**. New York: Apress Media, 2012.

GONCALVES, Antonio. **Beginning Java EE 7**. Apress, 2013.

GUEDES, Gilleanes T. A. **UML 2: uma abordagem prática. 2. ed.** São Paulo: Novatec, 2011.

INTERNATIONAL DATA CORPORATION. **Smartphone OS Market Share, Q1 2015**. Disponível em: <http://goo.gl/McnZJy>. Acesso em 06 de junho de 2015.

LECHETA, Ricardo R. **Desenvolvendo para iPhone e iPad**. Novatec Editora, 2012.

LECHETA, Ricardo R. **Google Android-3ª Edição: Aprenda a criar aplicações para dispositivos móveis com o Android SDK**. Novatec Editora, 2013.

MILANI, André. **PostgreSQL-Guia do Programador**. Novatec Editora, 2008.

MILANI, André. **Programando para iPhone e iPad: Aprenda a construir aplicativos para o iOS**. Novatec Editora, 2012.

MONACO, Thiago; DO CARMO, Rodolpho Marques. **Desenvolvendo aplicações para Windows Phone**. Brasport, 2012.

MONTEIRO, João B. **Google Android: Crie aplicações para celulares e tablets.** Casa do Código, 2012.

SAMPAIO, Cleuton. **Java Enterprise Edition 6-Desenvolvendo Aplicações Corporativas**. Brasport, 2011.

WARGO, John M. **PhoneGap Essentials: Building Cross-Platform Mobile Apps**. Addison-Wesley, 2012.

WORSLEY, John; DRAKE, Joshua D. **Practical PostgreSQL**. "O'Reilly Media, Inc.", 2002.