UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

MATHEUS PACHECO ABEGG

Desenvolvimento de um Protótipo de Solução Colaborativa para o Controle de Listas de Compras

Trabalho de Graduação

Prof. Dr. Leandro Krug Wives
Orientador

Porto Alegre, dezembro de 2014

“You, me, or nobody is gonna hit as hard as life.
But it ain’t about how hard you hit.
It’s about how hard you can get hit and keep moving forward.
How much you can take and keep moving forward.
That’s how winning is done!”
— ROCKY BALBOA

AGRADECIMENTOS

Agradeço a todas as pessoas que me estimularam o bastante para realizar este trabalho:

Aos meus pais e a toda minha família, que me apoiou, cobrou, deu toda a estrutura que precisei para poder estudar durante todos esses anos e, principalmente, serviu de inspiração e modelo para que eu pudesse chegar onde cheguei hoje.

A Carol, que tantas vezes evitou que eu fizesse coisas menos importantes em detrimento deste trabalho.

A Thais, que me ajudou a perceber a importância de concluir esta etapa da minha vida.

Agradeço em especial ao professor Leandro Krug Wives, meu orientador, sem o qual este trabalho não teria acontecido.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

HTTP	Hypertext Transfer Protocol
REST	Representational State Transfer
API	Application Programming Interface
JSON	JavaScript Object Notation
TDD	Test-Driven Development
GPS	Global Positioning System
URI	Uniform Resource Identifier

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

Atualmente, os dispositivos móveis estão muito presentes no cotidiano das pessoas, facilitando as mais diferentes tarefas. Uma necessidade comum a muitas pessoas é o gerenciamento de listas de compras, em especial, de maneira coletiva. Nesse sentido, este trabalho apresenta o desenvolvimento de um protótipo de aplicativo Android que visa tornar mais fácil o gerenciamento compartilhado de listas de compras. A solução é composta por dois elementos: um servidor web baseado em REST, que armazena os dados e atua como ponto de integração entre clientes, e um aplicativo Android que se comunica com o servidor através do protocolo HTTP. Ambos seguem boas práticas de construção de software e foram projetados de maneira a facilitar sua evolução. Assim, os processos de desenvolvimento do servidor REST e do cliente Android são descritos e são apresentadas algumas de suas principais características.

Palavras-chave: Android, REST, aplicativo móvel, serviço web.

Development of a Prototype for a Collaborative Solution for Shopping Lists Management

ABSTRACT

Mobile devices are very present on people's day to day life, making several tasks easier. An usual need most people have is the management of shopping lists, specially in a collaborative way. In that sense, this work presents the development of a prototype of Android application that aims to ease the shared management of shopping lists. The solution consists of two elements: a web server based on REST, which stores the data and works as an integration point between clients, and an Android application, which communicates with the server through the HTTP protocol. Both were developed with good software building practices in mind and were projected to allow for easy evolution. Thereby, the development process of the REST server and Android client are described and some of their main characteristics are presented.

Keywords: android, REST, mobile application, web service.

1 INTRODUÇÃO

O uso de dispositivos móveis está cada vez mais presente na vida cotidiana das pessoas. Segundo dados do IDC (2014), a venda de smartphones atingiram o volume recorde de vendas de 35,6 milhões de unidades vendidas no Brasil em 2013, um crescimento de 123% em relação ao ano anterior. Com aparelhos cada vez mais potentes, tarefas do dia-a-dia vêm sendo facilitadas. Navegação por GPS, envio de mensagens, e integração com redes sociais são apenas alguns exemplos das atividades que se tornam mais fáceis graças aos smartphones.

Um dos problemas comuns na vida de muitas pessoas é o gerenciamento de listas de compras. É bastante comum que seja utilizado um simples pedaço de papel onde são anotados os itens que precisam ser comprados. Esta técnica é bastante eficiente mas apresenta algumas limitações: uma lista só pode estar disponível em um único lugar em um determinado momento, o que gera um conflito no momento em que deseja-se que mais de uma pessoa gerencie esta lista. Além disso, quando uma lista é utilizada por mais de uma pessoa (e.g., uma lista de compras de uma família), todos precisam ter acesso físico ao pedaço de papel.

1.1 Objetivo

Este trabalho tem o objetivo de projetar e descrever o processo de desenvolvimento de um protótipo de aplicação que visa facilitar o gerenciamento colaborativo de listas de compras com a utilização de smartphones. Apesar de existirem algumas soluções no mercado, nenhuma delas atende a todos os requisitos propostos por este trabalho: capacidade de gerência colaborativa de listas e código aberto da solução. O requisito de colaboração foi obtido a partir de uma necessidade relatada por usuários.

Além de criar uma proposta simples de solução para o problema, este trabalho propõe uma fundação que permita que os resultados aqui obtidos sejam facilmente evoluídos. Para isso, além do código aberto, são utilizadas boas práticas de construção de software bastante difundidas no mercado.

1.2 Estrutura do texto

Este trabalho está organizado em cinco capítulos que abordam diferentes temas relevantes ao processo. As subseções a seguir apresentam os capítulos presentes no texto.

Capítulo 1 Apresenta o contexto em que este trabalho foi desenvolvido além da motivação que levou à sua execução. São apresentados seus objetivos e, por fim, a maneira como o texto está organizado.

Capítulo 2 Apresenta trabalhos que apresentam a maneira como funciona o processo de desenvolvimento para a plataforma móvel escolhida para este trabalho, trabalhos relacionados ao estilo arquitetural REST além de soluções proprietárias existentes que visam resolver o problema proposto.

Capítulo 3 Baseado nos estudos de trabalhos relacionados, este capítulo apresenta uma visão geral sobre o desenvolvimento de aplicativos para a plataforma Android e sobre o estilo arquitetural REST.

Capítulo 4 Neste capítulo é apresentado o processo de desenvolvimento da solução proposta. Estão presentes detalhes de como o servidor foi desenvolvido e testado e detalhes do desenvolvimento da aplicação Android.

Capítulo 5 Por fim, são apresentadas as conclusões e resultados obtidos além de possíveis avanços para a solução.

2 TRABALHOS RELACIONADOS

Neste capítulo serão apresentados alguns trabalhos que descrevem o processo de desenvolvimento de aplicativos para a plataforma Android, além de alguns dos principais produtos com propósito similar ao da ferramenta proposta nesse trabalho. Também são apresentados trabalhos implementados utilizando a abordagem REST, que é seguida neste trabalho por apresentar caraterísticas que tornam simples o desenvolvimento de aplicações WEB, conforme apresentado no Capítulo 3. A partir do estudo das soluções existentes, foram definidas quais as funcionalidades seriam implementadas para a solução proposta.

2.1 O processo de desenvolvimento para a plataforma Android

Nesta sessão serão apresentados trabalhos que, de alguma maneira, ilustram o processo de desenvolvimento de aplicativos para a plataforma Android. Será apresentada uma rápida descrição do assunto principal de cada um deles, além dos principais pontos abordados em relação à plataforma.

2.1.1 Check In Poa

Em Check In Poa: um aplicativo Android para turistas em Porto Alegre, é apresentada a "[...] elaboração e [...] desenvolvimento de um aplicativo Android no formato de um jogo direcionado a turistas em viagem a Porto Alegre ou moradores que desejem conhecer melhor a cidade em que vivem."(MACALÃO, 2013, p. 9).

O processo de desenvolvimento do aplicativo é descrito em detalhes. Primeiro são introduzidos dois ambientes integrados de desenvolvimento, Eclipse e Android Studio, juntamente com seu processo de instalação, além de suas principais características. Em seguida, a plataforma Android é explicada, começando pelo histórico, logo após a arquitetura do ecossistema Android é apresentada. Finalmente, a autora demonstra conceitos básicos sobre o desenvolvimento de aplicativos para dispositivos Android.

2.1.2 SACI

Em "SACI: Sistema de Apoio à Coleta de Informações", Souza(2014) explica o processo de criação de um sistema de coleta e gerenciamento de dados com foco em auditoria de pontos de venda.

Parte do sistema consiste de um cliente para Android. Souza(2014) apresenta um visão geral da arquitetura da plataforma e justifica algumas das escolhas feitas durante a implementação do sistema. São, também, mostrados alguns detalhes de componentes do sistema, como persistência e serviços baseados em localização. Outro ponto interessante é que princípios de design fornecidos pelo *Google* são mencionados, apesar de não haver um detalhamento de tais princípios.

2.1.3 Candy Castle

Candy Castle: Um Jogo Sério para Pacientes com Diabetes (JULLIEN, 2013) é um jogo que tem como objetivo estimular pacientes com diabetes a levarem uma vida mais ativa e consciente em relação a seu nível de glicose no sangue.

O autor descreve em detalhes pontos importantes da implementação do jogo. O primeiro passo da implementação foi a exibição de um mapa na tela do dispositivo do jogador, para isso foi necessário adicionar algumas bibliotecas e requisitar permissões do usuário para que o aplicativo tivesse acesso à *internet*. A exibição do mapa baseia-se na localização atual do usuário. Jullien (2013) diz, também, que existem duas maneiras de obter a localização de um dispositivo, e ambas as maneiras necessitam de autorização do usuário.

Em seguida é descrito o processo de inserção de marcadores no mapa. É oferecida uma explicação detalhada de como proceder para criar esse tipo de marcador. São descritos classes e métodos necessários, desafios encontrados e a soluções para os mesmos.

O terceiro elemento de implementação apresentado por Jullien (2013) é referente ao armazenamento de dados. Messe sentido, Jullien apresenta conceitos como *Intents* e *SharedPreferences*, que, em conjunto, são utilizados para que informação possa ser comunicada entre diversos pontos do aplicativo. Além disso, o processo de armazenamento de dados do aplicativo é descrito e as classes e bibliotecas utilizadas para tal fim têm seu funcionamento descrito.

Após explicar o funcionamento do fluxo de dados no sistema, o autor expõe características referentes ao processamento de dados e comunicação com o servidor *web* que serve dados para o dispositivo móvel. Devido a imposições da plataforma Android, chamadas de rede devem nunca devem ser feitas na *thread* principal (GOOGLE, 2014b), e, visando uma melhor experiência de usuário, processamentos que demoram muito para serem executados, devem ser rodadas em *threads* separadas.

A comunicação do aplicativo com o servidor acontece através do protocolo HTTP. Nessa parte do texto são expostos detalhes de como é feita a comunicação do aplicativo

móvel com o servidor, os passos de *login* e envio de *scores* para o servidor são descritos. Além disso, são mencionadas as autorizações que o usuário precisa fornecer ao aplicativo para que tais comunicações ocorram.

Detalhes da comunicação do aplicativo com o Facebook são fornecidos pelo autor. Para isso é utilizada uma biblioteca que oferece uma API que torna tudo bastante mais fácil. São apresentados detalhes da implementação como nomes de métodos chamados, parâmetros passados e funcionamento das *theads* envolvidas no processo.

Por último, o autor explica o funcionamento de outros dois botões existentes no aplicativo, o primeiro permite que o jogador acesse sua localização e entre valores de taxa de glicose no sangue e o segundo que oferece informações sobre o progresso do jogador.

2.1.4 Gerenciamento de Sistemas Turísticos

Marcon, em Um Sistema Android para Gerenciamento de Roteiros Turísticos, "[...] apresenta a concepção e o desenvolvimento de um aplicativo Android para prover o gerenciamento de roteiros turísticos a partir de uma abordagem distribuída com o armazenamento de dados na nuvem com a plataforma Google App Engine."(MARCON, 2013).

O autor, entre outras coisas, descreve como fez a escolha pela plataforma Android e apresenta uma breve descrição do processo utilizado para o desenvolvimento do aplicativo. Em seguida explica como implementou a aquisição de dados utilizando APIs do Google Places, responsável pela busca de estabelecimentos, Twitter e Google Maps, de onde foram extraídas informações referentes a rotas.

Vale ressaltar que o autor apresenta detalhes de como foi implementado o gerenciamento de locais favoritos e viagens. Inclusive trechos de código fonte estão presentes no texto visando facilitar o entendimento do leitor.

Por último é apresentada uma seção que descreve o funcionamento do aplicativo. Esta seção contém imagens e explicações das telas do sistema, dentre elas, a tela que apresenta uma rota ao usuário, que conta também com um trecho de código que esclarece seu funcionamento. Além disso, na descrição do aplicativo é apresentado o sistema de notificações, internacionalização, além de integração com outros serviços.

2.1.5 Music XML

Em Protótipo de um conversor MusicXML para MIDI usando Android, Kerber (2012) apresenta o processo de criação do protótipo que visa a converter música representada em formato MusicXML (MAKEMUSIC, 2014) para o formato MIDI, de maneira que possa ser reproduzida utilizando os recursos presentes em dispositivos Android.

A autora apresenta o escopo que será abordado do protótipo proposto e apresenta a plataforma MOTODEV Studio, que oferece um ambiente de desenvolvimento completo para aplicativos Android. Esse pacote não é mais oferecido pela Motorola, que recomenda a instalação manual dos componentes necessários para o desenvolvimento (KERBER,

2012).

O foco principal desse trabalho é apresentar detalhes do *parser* de MusicXML portanto a interface "[. . .] foi mantida simples e funcional."(KERBER, 2012). São apresentados diversos aspectos da definição MusicXML, detalhes da implementação da conversão da representação visual para notas musicais. Não são apresentados detalhes de como a arquitetura da plataforma escolhida impactou no desenvolvimento do protótipo.

2.1.6 Modelando Um Cliente Voip Inteligente Para A Plataforma Android

Uzejka(2011) apresenta, em "Modelando um Cliente VOIP Inteligentepara a Plataforma Android", uma descrição detalhada do funcionamento da plataforma Android e uma avaliação de como as características de tal plataforma pode influenciar o desenvolvimento do trabalho proposto. Um dos principais motivos apresentados para a escolha da plataforma utilizada, Android, é o fato de elementos do sistema poderem ser facilmente substituídos por aplicativos externos. No caso desse trabalho, o discador padrão é substituído pelo protótipo apresentado.

São apresentados alguns trechos de código que ilustram a maneira como o desenvolvedor pode utilizar as funcionalidades providas pela plataforma Android. Um exemplo disso é a possibilidade de descoberta do número de telefone do dispositivo em que o aplicativo está sendo executado. Isso é feito através de uma chamada de serviço do sistema, o que evita que o desenvolvedor possa causar danos aos dispositivos.

2.2 Trabalhos relacionados ao protocolo REST

Aqui serão apresentados trabalhos que têm em sua base serviços *web* baseados no estilo arquitetural REST (*REpresentational State Tranfer*) , um estilo arquitetural para o desenvolvimento de aplicações Web apresentado por Fiedling (2000) em sua tese de doutorado. Assim como na seção anterior, será feita uma pequena apresentação sobre o assunto central de cada trabalho e, em seguida, as principais características da arquitetura apresentadas por cada um.

2.2.1 Estendendo o Rest-Unit

Feller (2010) apresenta um *framework* que visa a geração automática de *drivers* de teste para serviços *web* baseados em REST a partir de modelos especificados a segundo o padrão U2TP (UML 2.0 Test Profile).

São apresentados diversos elementos apresentados pela arquitetura REST. Dentre eles os mais relevantes são o conceito de comunicação sem estado - "cada requisição do cliente para o servidor deve conter toda a informação necessária para seu entendimento"(FELLER, 2010) - e algumas das boas práticas que devem ser levadas em consideração ao construir um serviço REST. A autora apresenta também alguns conceitos referentes a teste de soft-

ware.

2.2.2 PyRester

Em "PyRester: Uma abordagem baseada em modelos U2TP para geração de código de teste unitário para RESTful Web Services", Rosa (2011) propõe uma extensão e adaptação do trabalho de Feller (2010). PyRester consiste em adicionar suporte à geração de código de teste para a linguagem Python, além de suporte a *payloads* no formato JSON.

Em relação às características da arquitetura REST, são apresentados conceitos básicos e importantes. Dado que esse trabalho se trata de uma arquitetura com alguns princípios bastante simples e bem definidos, não há muitas diferenças em relação ao que é apresentado em Feller(2010).

Alguns dos pontos que devem ser citados são, de novo, a comunicação sem estado, que torna a implementação no servidor bastante mais simples. As boas práticas que devem ser seguidas durante o desenvolvimento de serviços REST são citadas.

2.3 Aplicativos Semelhantes

Nesta seção serão apresentados alguns dos aplicativos mais bem avaliados na Play Store - loja de aplicativos da Google, utilizada pela plataforma Android - que têm como principal objetivo o gerenciamento de listas de compras. Estão apresentadas aqui apenas funcionalidades descritas pelos desenvolvedores de cada uma do aplicativos em sua página na Play Store.

Dentre a coleção de funcionalidade, foram selecionadas aquelas consideradas mais importantes. Os aplicativos comparados são:

- Lista de Compras;
- Shopping List;
- Meu Carrinho;
- Out of Milk;

Como pode ser observado na Tabela 2.1, nenhuma das soluções existentes possui código aberto. Além disso, a possibilidade de edição colaborativa de listas só é citada por um dos desenvolvedores e, em testes realizados durante a execução deste trabalho, tal funcionalidade é bastante limitada.

Tabela 2.1: Comparação das funcionalidades presentes nos aplicativos semelhantes

Funcionalidades/Aplicativo	Lista de Compras	Shopping List	Meu Carrinho	Out of Milk
Múltiplas Listas de compras	**Sim**	**Sim**	**Sim**	**Sim**
Compartilhamento de listas	**Sim**	**Sim**	**Sim**	**Sim**
Digitação facilitada de itens	**Sim**	**Sim**	Não	**Sim**
Categorização de Itens	Não	**Sim**	Não	Não
Registro de Preço	Não	Não	**Sim**	**Sim**
Leitura de Código de Barras	Não	Não	**Sim**	**Sim**
Suporte a múltipla plataformas	Não	Não	**Sim**	Não
Edição colaborativa de Listas	Não	Não	**Sim**	Não
Controle de Usuários	**Sim**	**Sim**	**Sim**	**Sim**
Código Aberto	Não	Não	Não	Não

3 PLATAFORMA ANDROID E ESTILO REST

Baseado no estudo de trabalhos relacionados, este capítulo apresenta uma visão geral da plataforma Android e a anatomia básica de suas aplicações. Além disso, é feito um panorama sobre o estilo arquitetural REST em conjunto com as práticas recomendadas para a aplicação de tal estilo.

3.1 Plataforma Android

Nesta seção são apresentados os principais elementos da arquitetura da plataforma Android além dos conceitos mais comuns envolvidos no desenvolvimento de aplicações para a plataforma.

3.1.1 Arquitetura

Lecheta (2010 apud MACALÃO 2013) apresenta a arquitetura da plataforma Android como dividida em camadas, as quais são responsáveis por manter o controle de seus respectivos processos. A Figura 3.1 apresenta uma visão geral de como as camadas estão divididas.

Tutorialspoint (2014) apresenta uma descrição a respeito de cada camada que compõe a arquitetura da plataforma Android:

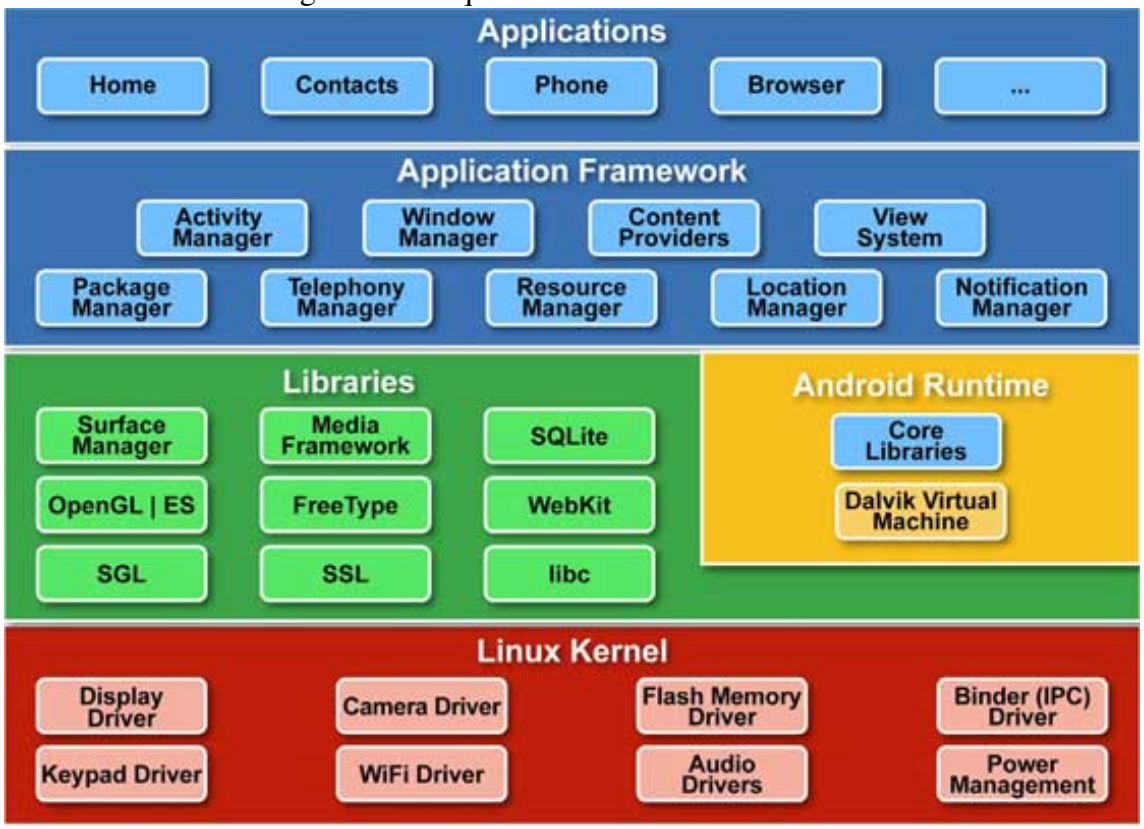
Kernel Linux

Na base do sistema operacional encontra-se o Kernel do Linux em sua versão 2.6 com apenas algumas modificações. Esta camada oferece recursos básicos como gerenciamento de processos, gerenciamento de memória, gerenciamento de dispositivos como câmera, tela e outras coisas. Além disso, o Kernel é responsável por lidar com interfaces de rede e outros dispositivos.

Bibliotecas

Sobre o Kernel do Linux está um conjunto de bibliotecas como o banco de dados SQLite, que é bastante útil para o armazenamento e compartilhamento de dados de aplicações, bibliotecas para a reprodução e gravação de áudio e SSL, responsável por segurança de dados transmitidos via rede.

Figura 3.1: Arquitetura da Plataforma Android



Fonte: (TUTORIALSPOINT, 2014)

Android Runtime

No mesmo nível em que se encontram as bibliotecas, está o *Android Runtime*, a camada de execução Android. Esta seção oferece a Máquina Virtual Dalvik (DVM), que é uma espécie de Máquina Virtual Java adaptada e otimizada especialmente para o sistema Android, além de uma série de bibliotecas centrais que permitem que desenvolvedores implementem aplicações Android utilizando a linguagem Java.

A DVM utiliza as funcionalidades providas pela Kernel Linux e permite que cada aplicativo Android seja executado em seu próprio processo, que é, por sua vez, uma instancia da DVM.

Framework de Aplicação

Esta camada oferece uma série de serviços, como por exemplo o controle de telefonia e localização, que podem ser acessados por desenvolvedores na forma de classes Java.

Aplicações

Neste nível encontram-se todas as aplicações que podem ser instaladas pelo usuário. Aplicativos são sempre instalados neste nível.

3.1.2 Anatomia de Uma Aplicação Android

Uma aplicação Android é composta por diversos componentes, esta seção apresenta uma visão geral de cada um destes componentes. Os mais comuns são Atividades (*Activities*) e Serviços (*Services*). Atividades são representações visuais das aplicações Android, são compostas por Telas e Fragmentos. Uma única aplicação pode ter várias Atividades, normalmente, uma atividade é responsável por gerenciar uma única funcionalidade dentro da aplicação. Serviços executam tarefas em segundo plano, ou seja, sem que haja uma interface direta com o usuário. São capazes de se comunicar com outras aplicações e são geralmente utilizados para tarefas que precisam ser executadas independentemente da interação com o usuário, como por exemplo, receber notificações para que possam ser passadas ao usuário (VOGEL, 2014).

Além de Serviços e Atividades, existem outros dois componentes que têm seu maior foco na comunicação de uma aplicação com outras aplicações instaladas no mesmo dispositivo, Receptores de Transmissão (*Broadcast Receiver*) e Provedores de Conteúdo (*Content Providers*). Receptores de Transmissão podem registrar-se para escutar eventos do sistema, desta maneira, o sistema avisa ao receptor sobre o acontecimento de determinado evento. É possível, por exemplo, registrar um receptor para que o mesmo seja notificado sempre que uma ligação for recebida. Provedores de Conteúdo oferecem formas de comunicação com dados de aplicativos. Um exemplo de Provedor de Conteúdo que existe por padrão na plataforma Android é o conjunto de palavras não existentes no dicionário padrão as quais o usuário deseja armazenar (VOGEL, 2014).

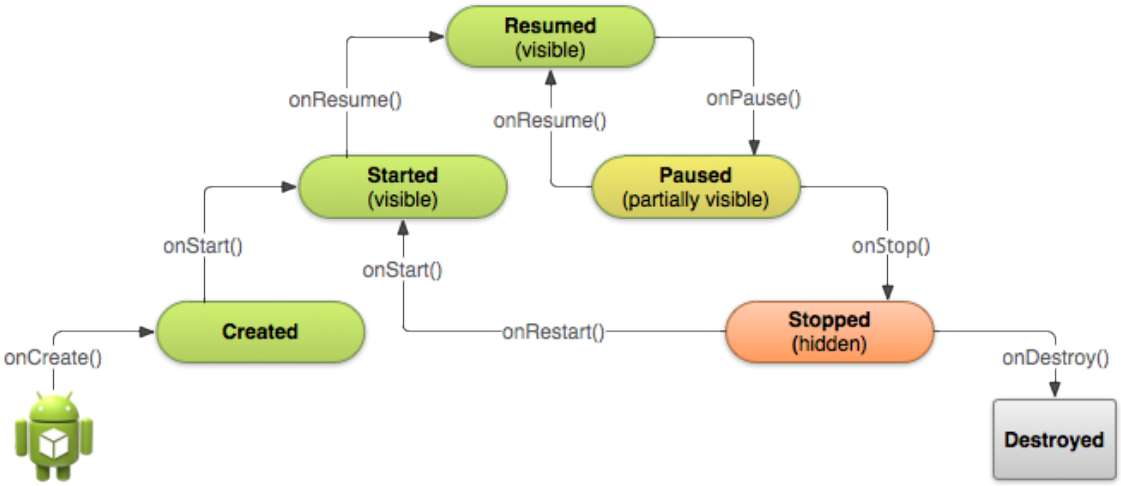
Além dos componentes que gerenciam o comportamentos e acesso a dados da aplicação, existem componentes de interação com o usuário e gerenciamento de telas do aplicativo, Atividades e Fragmentos. Uma atividade, como mencionado anteriormente é uma representação visual de uma aplicação, elas podem conter sua própria interface gráfica, entretanto, é recomendado que a construção destas interfaces seja feita através de Fragmentos (GOOGLE, 2014d). Fragmentos representam comportamentos ou porções de interface de usuário de uma atividade. É possível combinar diversos fragmentos para a criação de interfaces com múltiplas seções além de ser possível que o mesmo fragmento seja utilizado e mais de uma área da aplicação. Fragmentos possuem, apesar de terem seu ciclo de vida atrelado à Atividade a qual pertencem, possuem seu próprio ciclo de vida (VOGEL, 2014).

3.1.2.1 O Ciclo de Vida de Uma Atividade

Em (GOOGLE, 2014e) é apresentado o ciclo de vida de uma Atividade. É importante que os desenvolvedores entendam como funciona este ciclo para que possam planejar adequadamente o comportamento de suas aplicações evitando erros quando, por exemplo, seu aplicativo estiver apenas parcialmente visível, ou estiver em segundo plano.

Existem diversos estados em que uma Atividade pode se encontrar, entretanto, apensar

Figura 3.2: Ciclo de vida de uma Atividade



Fonte: (GOOGLE, 2014e)

três podem ser estáticos, ou seja, Uma atividade só existe, em um perídio longo de tempo, em um destes três estados:

Resumed

Neste estado, a atividade está em primeiro plano, e é o estado em que o usuário pode interagir com ela. Este estado também pode ser referido como 'rodando'.

Paused

Quando a atividade está parcialmente coberta por outra atividade, seja porque a atividade em primeiro plano apresenta algum tipo de transparências, ou porque não cobre toda a tela. Este estado não permite interação do usuário.

Stopped

Uma atividade é considerada parada quando está completamente não visível para o usuário, ou seja, está em segundo plano. Quando uma atividade se encontra neste estado, toda sua informação de estado é mantida, como variáveis, por exemplo, entretanto, ela não pode executar nenhum tipo de código.

Os outros estados em que as atividades podem se encontrar são transitivos, ou seja, o sistema passa por eles rapidamente.

A Figura 3.2 apresenta uma ilustração simplificada do ciclo de vida e uma atividade e as diferentes transições de estado que podem ocorrer.

O desenvolvedor tem o poder de controlar como sua aplicação deve se controlar durante as transições entre os estados através de métodos que podem ser sobre escritos. Mais detalhes sobre como controlar a ciclo de vida de uma Atividade podem ser encontrados em (GOOGLE, 2014f).

3.2 Estilo Arquitetural REST

Em sua tese de doutorado Roy Fiedling(2000) apresenta uma generalização dos princípios arquiteturais que compõe a Web e sugere um estilo arquitetural. Ele descreveu como sistemas distribuídos operam, como recursos interagem entre si e a importância de identificadores únicos. Além disso, falou sobre o uso de um conjunto limitado de operações com significado uniforme que permite a criação de ums estrutura capaz de suportar qualquer tipo de aplicação. Fiedling (2000) refere-se a este estilo arquitetural como *REpresentational State Transfer* (Transferência de Estado Representacional), ou simplesmente REST (WEBBER; PARASTATIDIS; ROBINSON, 2010).

Este estilo arquitetural é modelado em termos de uma série de restrições que são descritos por Fiedling em seu trabalho (FIELDING, 2000):

Cliente-Servidor

Separando responsabilidades de interface de usuário das responsabilidades de armazenamento de dados obtém-se uma melhora na portabilidade da interface do usuário através de várias plataformas além de escalabilidade, devido à simplificação dos componentes do servidor. Talvez para a Web, o mais importante seja que esta separação permite que componentes evoluam independentemente.

Comunicação *Stateless*

Cada requisição feita de um cliente ao servidor deve conter toda a informação necessária para que o servidor possa compreender a requisição, sem que se possa fazer proveito de qualquer informação contextual presente no servidor. Toda informação referente a sessão é, portanto, mantida no cliente. Esta restrição induz às propriedades de *visibilidade* - sistemas de monitoramento não precisam olhar além dos dados de uma única requisição para saber a natureza desta requisição -, *confiabilidade* - é mais fácil recuperar-se de falhas parciais - e *escalabilidade* - a não necessidade de armazenar informações referentes a múltiplas requisições permite ao servidor liberar recursos mais rapidamente, além de simplificar sua implementação.

Cache

A restrição de *cache* é adicionada com o objetivo de melhorar a performance da rede. Esta restrição determina que os dados contidos em uma resposta do servidor a uma requisição devem ser marcados como *cacheable* ou *non-cacheable*. Se uma resposta for *cacheable*, isto significa que o cliente pode reutilizar os dados contidos nesta resposta para requisições equivalentes, em outro momento.

A grande vantagem desta restrição é possibilitar ao cliente potencialmente eliminar parcial ou totalmente algumas requisições, o que melhora a eficiência do sistema, escalabilidade além de uma melhor performance percebida pelo usuário.

Interface Uniforme

A principal característica que diferencia REST de outros estilos é sua ênfase em interfaces uniformes entre componentes. O princípio de Engenharia de Software da generalidade aplicado às interfaces dos componentes simplifica a arquitetura do sistema como um todo além de oferecer melhor visibilidade das interações.

Sistema em Camadas

Sistemas em camadas permitem que sua arquitetura seja construída de maneira hierárquica restringindo o comportamento de componentes de maneira que cada componente não possa "ver"além da camada com a qual interage. Esta restrição adiciona uma barreira que diminui a complexidade geral do sistema e promove independência entre camadas. Camadas podem ser usadas para encapsular sistemas legados e proteger sistemas novos de clientes legados, simplificando componentes por mover funcionalidades que não são usadas com frequência para um intermediário comum.

Código sob demanda

REST permite que funcionalidade dos clientes seja estendida baixando e executando código no formato de *applets* e scripts. Isto simplifica a implementação de clientes ao reduzir o número de funcionalidades que precisam ser implementadas. Esta restrição, apesar de aumentar a extensibilidade do sistema, diminui sua visibilidade, portanto é uma restrição opcional na arquitetura.

REST é uma abstração de elementos arquiteturais contidos em sistemas de *hypermedia* distribuídos. REST é focado nos papéis dos componentes, suas interações e sua interpretação de dados significativos, detalhes de implementação e sintaxe de protocolos são ignorados de modo a permitir tal foco. REST engloba as principais restrições em torno dos componentes, conectores e dados que definem a base da arquitetura da Web e, assim, definem também a essência de seu comportamento como uma aplicação baseada em rede (FIELDING, 2000).

3.2.1 *RESTFul Web Services*

Um *RESTful Web Service* é um serviço Web simples, construído com base no estilo REST, que utiliza padrões bem definidos como HTTP e URI e que pode ser definido como uma coleção de recursos com três aspectos bem definidos (SILVA, 2011):

- Uma URI básica para o serviço (e.g. https://servicoweb.com/recursos);
- Um tipo de mídia de Internet (MIME) para os dados suportados pelo serviço (e.g. XML, JSON);
- Um conjunto de operações suportadas pelo serviço utilizando os verbos que compõem os métodos HTTP.

Por não ser um padrão e sim um estilo arquitetural, não existe uma maneira "oficial"de se construir serviços REST, e sim um conjunto de boas práticas que são usadas para guiar o desenvolvimento. Algumas delas são (RICHARDSON; RUBY, 2007):

Endereçabilidade

Um Serviço é endereçável se expõe aspectos relevantes de seu conjunto de dados através de *recursos*. Cada recurso deve possuir seu próprio URI único e um mesmo URI nunca deve representar mais de um recurso.

Interface Uniforme

Toda interação entre clientes e recursos deve se dar através de métodos HTTP básicos. Cada recurso deve expor alguns ou todos estes métodos e um mesmo método deve ter o mesmo comportamento em todos os recursos que o suportam. Os métodos HTTP e seu comportamento esperado são os seguintes:

GET Uma requisição GET solicita informações sobre um recurso. A informação é retornada como um conjunto de cabeçalhos e uma representação. O cliente nunca envia nenhum tipo de representação junto de uma requisição GET.

HEAD Igual a uma requisição GET, exceto pelo fato de que apenas os cabeçalhos são retornados, ou seja, a representação é omitida.

PUT Uma requisição PUT é uma asserção a respeito do estado de um recurso. O cliente geralmente envia uma representação junto de uma requisição PUT e o servidor tenta criar ou modificar o recurso para que seu estado esteja de acordo com a representação enviada. Uma requisição PUT pode ser feita sem nenhuma representação e neste caso apenas verifica a existência ou não do recurso.

DELETE O método HTTP DELETE é utilizado para indicar ao servidor que um recurso não deve mais existir. Nenhuma representação é enviada junto de uma requisição DELETE.

POST Uma requisição POST é utilizada na tentativa de criar-se um novo recurso a partir de um recurso existente. A representação enviada junto a requisição descreve o estado inicial do recurso. Assim como PUT, requisições POST não necessariamente precisam enviar uma representação.

OPTIONS Raramente utilizada, esta requisição é uma tentativa de descobrir quais operações são suportadas por um recurso. Serviços modernos fornecem tal informação antecipadamente.

Representatividade através de códigos de status HTTP Operações sobre recursos devem retornar um código de status adequando. A Tabela 3.1 apresenta os códigos

Tabela 3.1: Códigos de status HTTP mais comuns em REST

Código	Nome	Descrição
200	*OK*	Requisição bem sucedida.
201	*Created*	A requisição foi bem sucedida e um novo recurso foi criado.
400	*Bad Request*	A não pode ser processada pelo servidor por estar malformada.
404	*Not Found*	O servidor não pode encontrar o recurso identificado pelo URI da requisição.
500	*Internal Server Error*	Algo inesperado ocorreu no servidor que o impediu de completar a requisição

mais comuns e seus significados. Mais informações e detalhes sobre o uso correto de códigos de status podem ser encontrados em (W3, 1999).

4 O PROCESSO DE DESENVOLVIMENTO

Este capítulo apresenta o processo de desenvolvimento do protótipo proposto para o aplicativo de controle de listas de compras. O nome escolhido para o aplicativo é Batatas.

4.1 Metodologia de Desenvolvimento

Baseado no estudo de aplicativos de controle de lista de compras existentes e em suas principais funcionalidades, foi escolhido um conjunto mínimo de necessidades que deveriam ser implementadas para resolver o problema descrito abaixo.

> "Quando vou ao mercado faço uma lista das coisas que preciso comprar. Muitas vezes minha esposa lembra de coisas que deveriam estar na lista mas foram esquecidas. Hoje, não existe uma maneira simples de ela poder adicionar tais itens à lista que levo ao mercado.
>
> Gostaria que houvesse uma maneira de minha esposa adicionar itens a uma lista comum e que essas modificações ficassem disponíveis a mim assim que feitas, sem a necessidade de que eu replique as mesmas modificações em minha lista."

Por se tratar de um protótipo, algumas funcionalidades foram preteridas com o objetivo de simplificar a implementação e manter o foco no problema em questão. Um exemplo de funcionalidade que não foi implementada durante a construção deste protótipo é o controle de usuários. Todos os usuários têm acesso a todas as listas existentes, isso não é um problema visto que o aplicativo não está disponível publicamente portanto o acesso aos dados existentes é limitado.

Levando-se em conta tais limitações, foram implementadas as seguintes funcionalidades:

Criação de listas de compras.

Os usuários podem criar listas que serão identificadas por seus nomes.

Adição de itens a uma lista de compras existente.

Dado que um usuário criou uma lista de compras, ele pode adicionar itens a ela. Itens em uma lista são caracterizados por seu nome e pela quantidade que deve ser comprada.

Sinalização de compra de um item.

Usuários são capazes de indicar que um determinado item de uma lista foi comprado.

Sinalização de não compra de um item.

Analogamente à marcação de um item como comprado, o usuário pode sinalizar que itens não foram comprados. Essa sinalização e a sinalização de compra de um item são mutuamente excludentes.

Remoção de uma lista de compras.

Usuários são capazes de selecionar uma ou múltiplas listas de compras e removê-las do sistema. Isso permite que erros possam ser corrigidos.

Eliminação de itens de uma lista de compras existente.

Analogamente à remoção de listas, usuários podem remover um ou vários itens de uma lista.

Como se pode perceber, a capacidade de edição de listas e itens não foi implementada. Essa escolha foi feita baseada no fato de que erros podem ser corrigidos, mesmo que não da melhor maneira possível.

Para o desenvolvimento de fato do protótipo, optou-se pelo uso de uma adaptação de metodologias ágeis que pregam o fluxo contínuo de trabalho, o limite de trabalho em andamento além da entrega frequente de valor ao cliente. Tal abordagem permite que tudo o que é desenvolvido possa ser rapidamente apresentado ao cliente, que por sua vez, pode fazer críticas e sugerir melhorias o mais cedo possível, de maneira a evitar desperdícios (HIBBS; JEWETT; SULLIVAN, 2009). Definidas as funcionalidades necessárias e a metodologia adotada para o desenvolvimento, foi feita a priorização do desenvolvimento, ou seja, quais funcionalidades seriam implementadas primeiro. A ordem escolhida foi a seguinte:

1. *Criação de listas de compras;*
2. *Adição de itens a uma lista de compras existente;*
3. *Remoção de itens de uma lista de compras existente;*
4. *Remoção de uma lista de compras;*
5. *Sinalização de compra de um item;*

6. *Sinalização de não compra de um item.*

Para cada uma das funcionalidades foi implementado o funcionamento do servidor REST além do cliente Android, permitindo a apresentação de resultados de forma transparente além da execução de melhorias incrementais no aplicativo.

4.2 Desenvolvimento do Servidor

O desenvolvimento de servidor REST foi feito utilizando a linguagem Ruby. A escolha da linguagem foi feita baseada nas ferramentas existentes, em sua sintaxe bastante sucinta além da familiaridade do autor deste trabalho com a linguagem.

Em conjunto com a linguagem, foi utilizado o *micro-framewok* Sinatra para a implementação do controle de requisições HTTP. Assim como a escolha da linguagem para a implementação, a escolha do *framework* baseou-se na simplicidade com que é possível controlar requisições, assim como na familiaridade do autor com tal ferramenta.

As seções a seguir apresentam detalhes da implementação.

4.2.1 Estrutura do Projeto

A Figura 4.1 apresenta a estrutura de pastas e arquivos que compõem o servidor REST implementado. Um dos componentes mais importantes é o arquivo *server.rb*, que representa o ponto de entrada do aplicativo. Nele são feitas referências a arquivos e bibliotecas que precisam ser carregados durante a execução da aplicação além de algumas configurações. Um exemplo de configuração que pode ser visto na Figura 4.2 é que o tipo de conteúdo de todas as requisições feitas para o servidor, assim como suas respostas será no formato *application/json*.

Além do ponto de entrada da aplicação, duas pastas são importantes para que a aplicação funcione corretamente. A pasta *models*, que contém a definição dos modelos de domínio da aplicação, e a pasta *routes*, que contém os arquivos responsáveis pelo processamento e resposta das requisições feitas ao servidor.

4.2.2 Implementação dos Modelos

Por se tratar de um domínio bastante reduzido, a modelagem do sistema tornou-se surpreendentemente simples. Existem apenas três elementos que representam os conceitos de domínio da aplicação: *Lists*, *Products* e *Items*.

Lists representa o conceito de uma lista. Esse modelo é composto por uma coleção de *Items* e um nome. Além disso, foram implementados alguns métodos que auxiliam a adição e recuperação de itens, um método que verifica se a lista em questão está vazia ou não e um último método que é responsável por oferecer uma representação da lista no formato JSON. A implementação desse modelo pode ser vista na Figura 4.3.

Figura 4.1: Estrutura de pastas e arquivos do projeto do servidor.

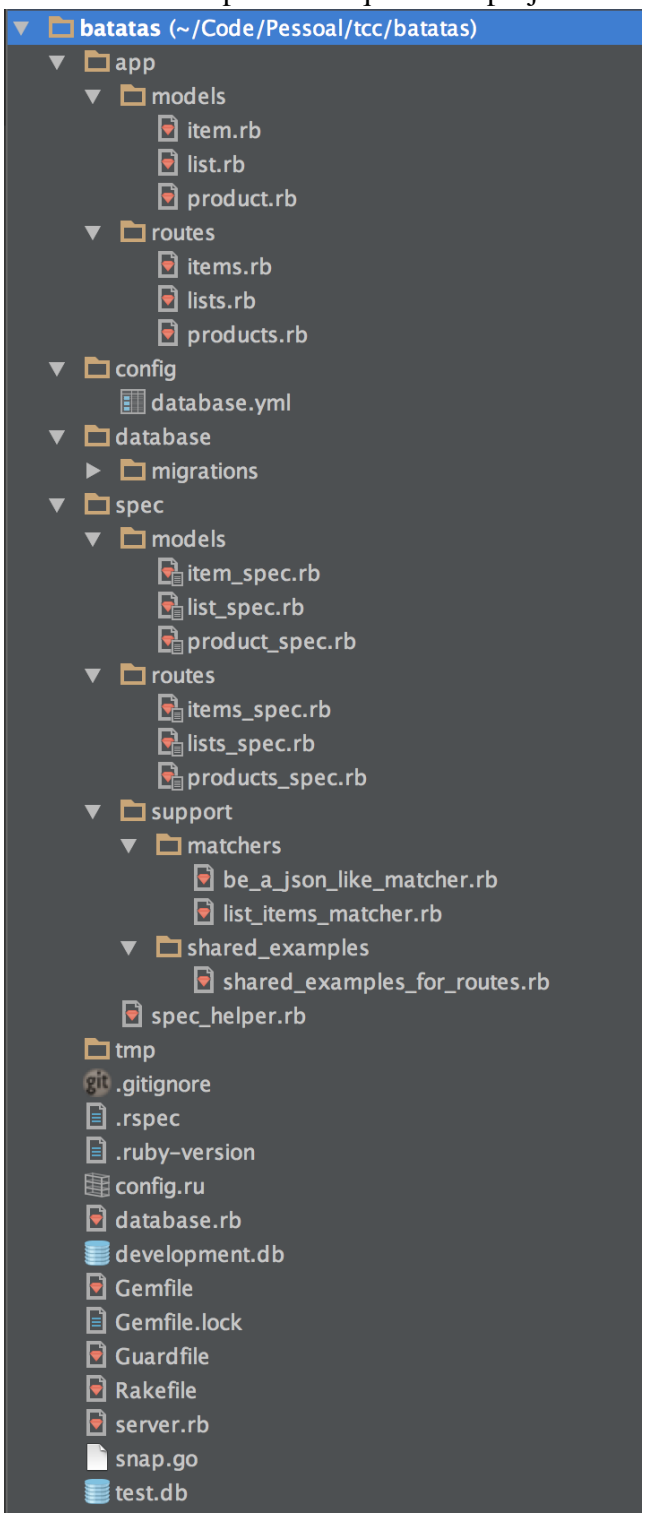


Figura 4.2: Arquivo server.rb. O ponto de entrada da aplicação servidor.

```ruby
require './database'
require 'sinatra/json'
require 'json'

Dir[File.dirname(__FILE__) + '/app/models/**'].each { |model| require model }
Dir[File.dirname(__FILE__) + '/app/routes/**'].each { |route| require route }

helpers do
  def logger
    request.logger
  end
end

before do
  content_type 'application/json'
end
```

Figura 4.3: Arquivo list.rb. Implementação do modelo de Lista

```ruby
class List < Sequel::Model
  one_to_many :items

  def add(items)
    items = [items].flatten
    items.each { |item| add_item(from_json(item)) }
    save
  end

  def to_json
    {
        id: id,
        name: name,
        items: items.map(&:to_json)
    }
  end

  def item(id)
    items.find { |i| i.id == id}
  end

  def empty?
    items.empty?
  end

  private
  def from_json(item)
    options = {product: Product.with_name(item['name']), amount: item['amount']}
    options[:bought] = item['bought'] if item['bought']
    options
  end
```

Figura 4.4: Arquivo product.rb. Implementação do modelo de Produto.

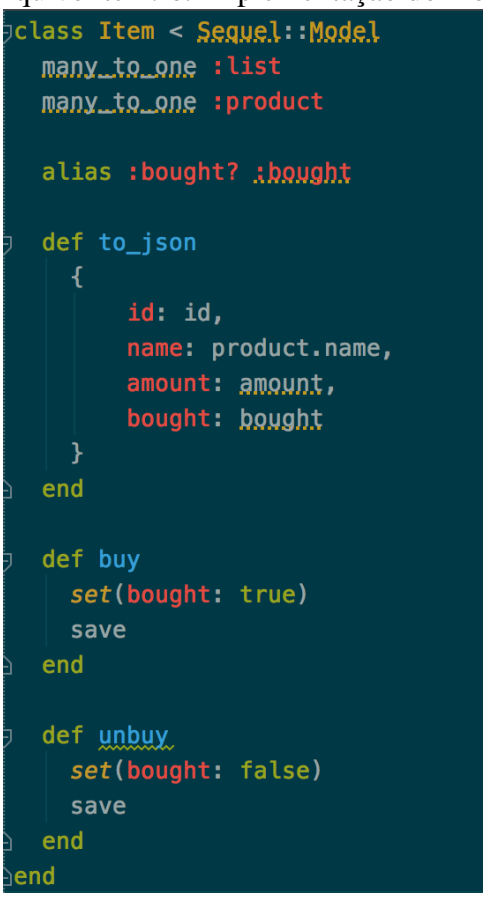
```ruby
class Product < Sequel::Model
  Sequel::Model.plugin :json_serializer

  def self.with_name name
    self.first(name: name) || create(name: name)
  end

  def to_json
    {
        name: name
    }
  end
end
```

Products representa o conceito de um produto. Esse modelo é o mais simples implementado neste trabalho; possui uma única propriedade que é seu nome. Afim de tornar mais fácil a garantia de não duplicidade de produtos, foi implementado um método auxiliar que permite acessar um produto através de seu nome. Caso já exista um produto com o nome especificado é simplesmente retornada uma referência ao produto existente. Caso contrário, um novo produto é criado com o nome em questão. Além disso, essa classe também possui um método que oferece uma representação do produto no formato JSON.

Items é o modelo que estabelece a relação entre uma lista e os produtos contidos nela. Nele são armazenadas informações sobre a qual lista de compras cada produto está associado, além disso também contém a quantidade de itens a ser comprado assim como seu estado atual (comprado/não comprado). Além de um método que fornece a representação do item no formato JSON, essa classe implementa métodos que tornam mais simples a interação com a propriedade que diz se o item já foi comprado ou não. O método *"bought?* é apenas um nome alternativo que retorna o valor da propriedade *booleana bought*. Os métodos *buy* e *unbuy* alteram o estado do item de maneira que os clientes da classe não precisem se preocupar com detalhes internos da mesma. Os detalhes podem ser encontrados na Figura 4.5.

A técnica de mapeamento objeto-relacional (JBOSS, 2014) foi utilizada de maneira a abstrair comandos de interação com o banco de dados tornando a utilização dos modelos no código bastante simples. A biblioteca Sequel foi utilizada para viabilizar esta técnica. A definição da estrutura de tabelas é definida nos arquivos da pasta *migrations*, que contém arquivos que descrevem alterações a serem feitas no esquema do banco de dados. A Figura 4.6 ilustra a primeira migração de esquema, que cria o banco de dados e as primeiras tabelas necessárias para o desenvolvimento.

Figura 4.5: Arquivo item.rb. Implementação do modelo de Item.



```ruby
class Item < Sequel::Model
  many_to_one :list
  many_to_one :product

  alias :bought? :bought

  def to_json
    {
        id: id,
        name: product.name,
        amount: amount,
        bought: bought
    }
  end

  def buy
    set(bought: true)
    save
  end

  def unbuy
    set(bought: false)
    save
  end
end
```

Figura 4.6: Arquivo create_database.rb. Responsável pela criação das primeiras tabelas no banco de dados.

```ruby
require 'sinatra'
require 'sinatra/sequel'

Sequel.migration do
  up do
    create_table :lists do
      primary_key :id
      String :name, null: false
      timestamp :created_at, null: false
    end

    create_table :products do
      primary_key :id
      text :name, null: false
      timestamp :created_at, null: false
    end

    create_table :items do
      primary_key [:list_id, :product_id], name: 'item_pk'
      foreign_key :list_id, null: false
      foreign_key :product_id, null: false
      Fixnum :amount
    end
  end

  down do
    drop_table :lists
    drop_table :products
    drop_table :items
  end
end
```

4.2.3 Testes Automatizados

Como mencionado em 4.1, o desenvolvimento das funcionalidades do servidor foi feito de maneira incremental. Tal abordagem também foi utilizada para o desenvolvimento de cada uma das funcionalidades. Cada pequeno trecho de comportamento esperado foi implementado de maneira que pudesse ser testado antes que o desenvolvimento de um próximo comportamento fosse iniciado.

Foi utilizada a técnica de desenvolvimento guiado por testes (TDD) (BECK, 2001) que permite que pequenas iterações possam ser utilizadas durante o desenvolvimento de um *software*. Além de estimular que o desenvolvedor pense a respeito de como deseja que o sistema se comporte antes de implementá-lo de fato, testes escritos de maneira descritiva garantem que todo o sistema reaja da maneira desejada dadas as situações previstas, e também funcionam como uma espécie de documentação viva do código. O resultado da execução dos testes oferece uma descrição bastante detalhada de todos os comportamentos previstos para o sistema implementado, como pode ser visto na Figura 4.7.

A figura 4.8 apresenta a descrição do comportamento que o servidor deve ter quando uma requisição GET é feita ao recurso */lists* do servidor REST. Fica bastante evidente para qualquer programador que leia os testes de que forma o servidor deve se comportar dada uma situação específica.

Apesar de bastante simples, a implementação do servidor conta com 67 casos de testes automatizados, que, em conjunto, permitem ao desenvolvedor confiar no comportamento do *software*. Esse conjunto de testes permite verificar rapidamente se alterações feitas no código levaram alguma das funcionalidades a deixar de funcionar como esperado.

4.3 Desenvolvimento do Cliente Android

A plataforma Android foi escolhida para a implementação do primeiro cliente do serviço. Nesta seção serão apresentadas algumas das informações mais relevantes do desenvolvimento do aplicativo.

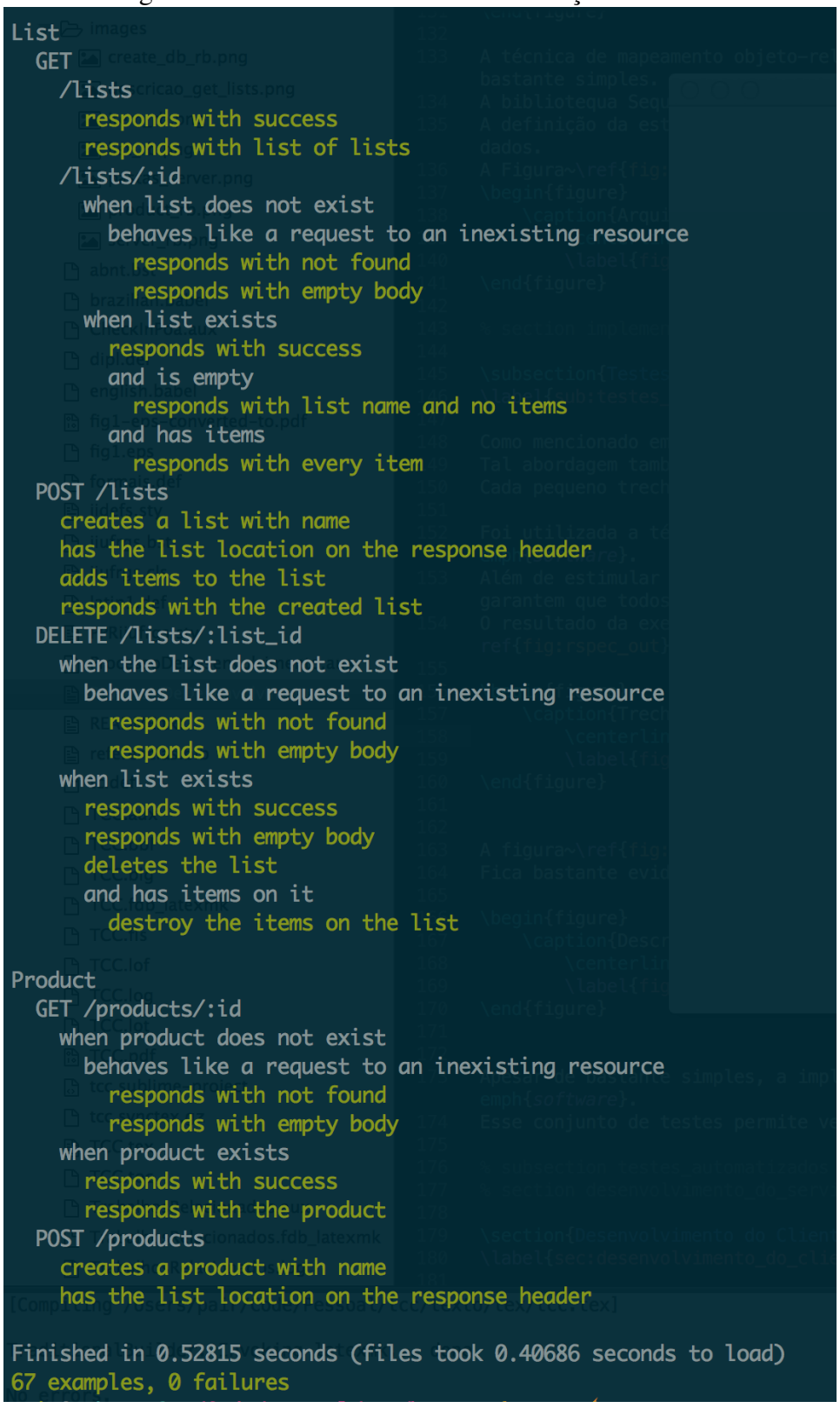
4.3.1 Estrutura de Classes e Pacotes

O código fonte do aplicativo Android se divide em cinco grandes grupos de classes.

No pacote raiz estão duas classes que coordenam a interação e criação dos fragmentos responsáveis pelas demais classes do aplicativo, *ListsOverviewActivity* e *ShoppingListActivity*. Além disso, as classes *Activity* respondem à resposta do usuário quando criando listas de compras e adicionando itens às mesmas.

O pacote *view* contém classes que adicionam componentes customizados à aplicação. Foram implementadas duas adições, uma delas torna mais fácil a criação de campos de texto que utilizam fontes que não estão disponíveis por padrão, estas fontes foram usadas para representar ícones no aplicativo, em especial os ícones que representam se um item

Figura 4.7: Trecho do resultado da execução dos testes

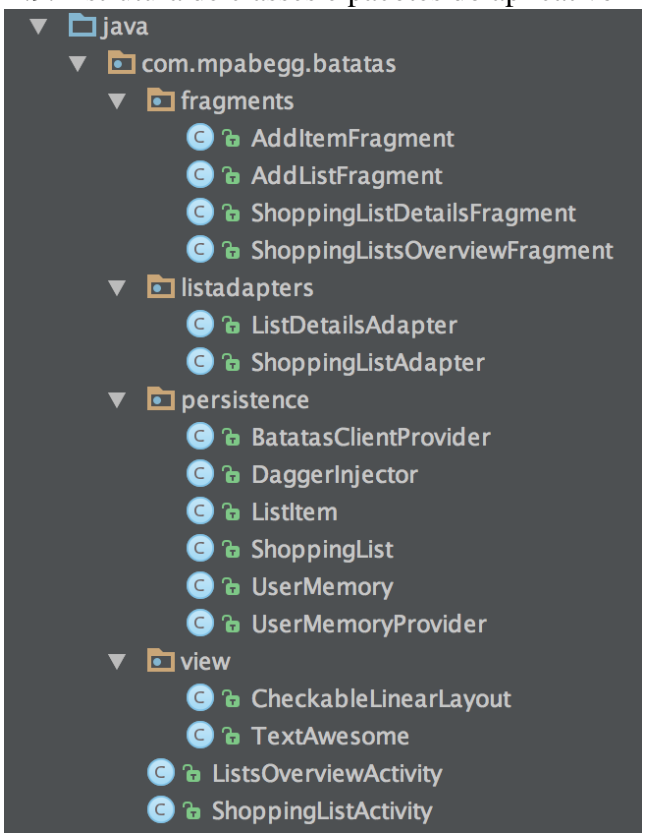


```
List
  GET
    /lists
      responds with success
      responds with list of lists
    /lists/:id
      when list does not exist
        behaves like a request to an inexisting resource
          responds with not found
          responds with empty body
      when list exists
        responds with success
        and is empty
          responds with list name and no items
        and has items
          responds with every item
  POST /lists
    creates a list with name
    has the list location on the response header
    adds items to the list
    responds with the created list
  DELETE /lists/:list_id
    when the list does not exist
      behaves like a request to an inexisting resource
        responds with not found
        responds with empty body
    when list exists
      responds with success
      responds with empty body
      deletes the list
      and has items on it
        destroy the items on the list

Product
  GET /products/:id
    when product does not exist
      behaves like a request to an inexisting resource
        responds with not found
        responds with empty body
    when product exists
      responds with success
      responds with the product
  POST /products
    creates a product with name
    has the list location on the response header

Finished in 0.52815 seconds (files took 0.40686 seconds to load)
67 examples, 0 failures
```

Figura 4.8: Descrição do comportamento de uma requisição através de testes unitários

```ruby
describe 'GET' do
  let (:list) { ... }
  describe '/lists' do
    it 'responds with success' do
      get '/lists'
      expect(last_response.status).to be 200
    end

    it 'responds with list of lists' do
      add_items_to_list

      get '/lists'
      expect(last_response.body).to be_a_json_like "[#{full_list_body}]"
    end
  end
```

Figura 4.9: Estrutura de classes e pacotes do aplicativo Android



já foi comprado ou não. A segunda adição permite que o usuário possa fazer seleções múltiplas em listas, que também não está disponível por padrão para desenvolvedores. A seleção múltipla é importante para que a funcionalidade de deleção de listas e itens se torne mais fácil de executar pelos usuários.

A pacote *listadapters* contém classes que funcionam como pontes entre as listas apresentadas ao usuário e os dados apresentados por estas listas (GOOGLE, 2014g). No sistema são apresentadas duas telas de listas, uma que apresenta todas as listas de compras existentes, e outra que representa os itens contidos nas listas de compras.

As classes que representam porções de comportamento, os fragmentos (GOOGLE, 2014d), estão no pacote *fragments*. Existem quatro fragmentos na aplicação, dois que definem o comportamento das caixas que permitem ao usuário criar novas listas de compras e adicionar itens a uma lista de compras existente. Outros dois fragmentos controlam a maneira como as listas de compras são apresentadas ao usuário. As Figuras 4.10 e 4.11 apresetam, respectivamente, os fragmentos responsáveis por adicinar um ítem a uma lista de compras e por exibir detalhes de uma lista, como por exemplo, os itens contidos na mesma e suas quantidades.

O último pacote de classes existente contém classes que representam os modelos de domínio, ou seja, listas de compras e itens contidos em uma lista. É importante notar que, diferentemente da modelagem do servidor, o aplicativo Android não contém o conceito de produto não pertencente a uma lista. As classes responsáveis pela comunicação com o servidor web também estão contidas no mesmo pacote, *persistence*. Mais detalhes sobre a comunicação do aplicativo móvel com o servidor REST são apresentados na subseção 4.3.2.

4.3.2 Integração com o Servidor

O componente mais importante da aplicação Android é o cliente do serviço REST. Este componente é responsável por oferecer a possibilidade de listas de compras serem compartilhadas entre diversos dispositivos.

Diferentemente de outras soluções apresentadas no Capítulo 2, que utilizam conceitos bastante básicos para gerenciar requisições HTTP, para este trabalho, optou-se por utilizar uma biblioteca que torna a implementação de um cliente para serviços REST bastante simples. O uso de Retrofit (SQUARE, 2014a) permite que upa API REST seja representada através de uma simples *Interface* JAVA.

A Figura 4.12 apresenta tudo o que é necessário para que seja definida uma API REST. O primeiro passo para a criação de um cliente para uma API REST com a biblioteca Retrofit (2014a) é a declaração de uma *interface* Java pública. Ela irá conter todas as definições de métodos disponíveis na API.

Dado que exista uma *interface* pública declarada, resta especificar os recursos presentes na API REST que estarão disponíveis para o cliente sendo implementado. Cada

Figura 4.10: Fragmento responsável pela adição de um ítem a uma lista

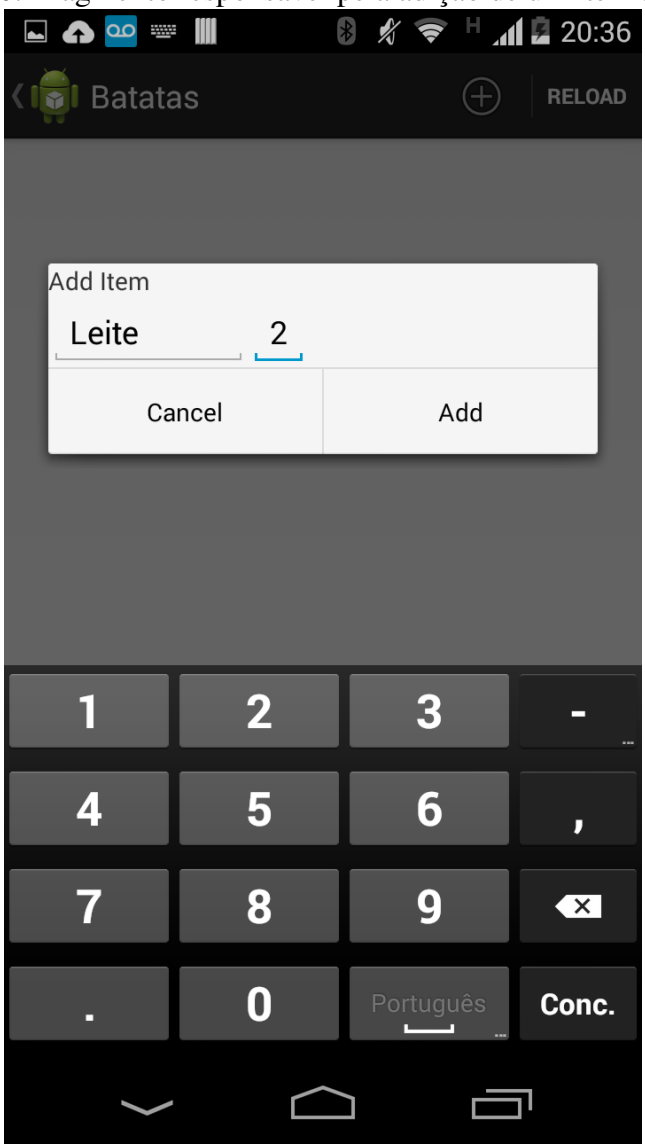


Figura 4.11: Fragmento que exibe detalhes de uma lista de compras

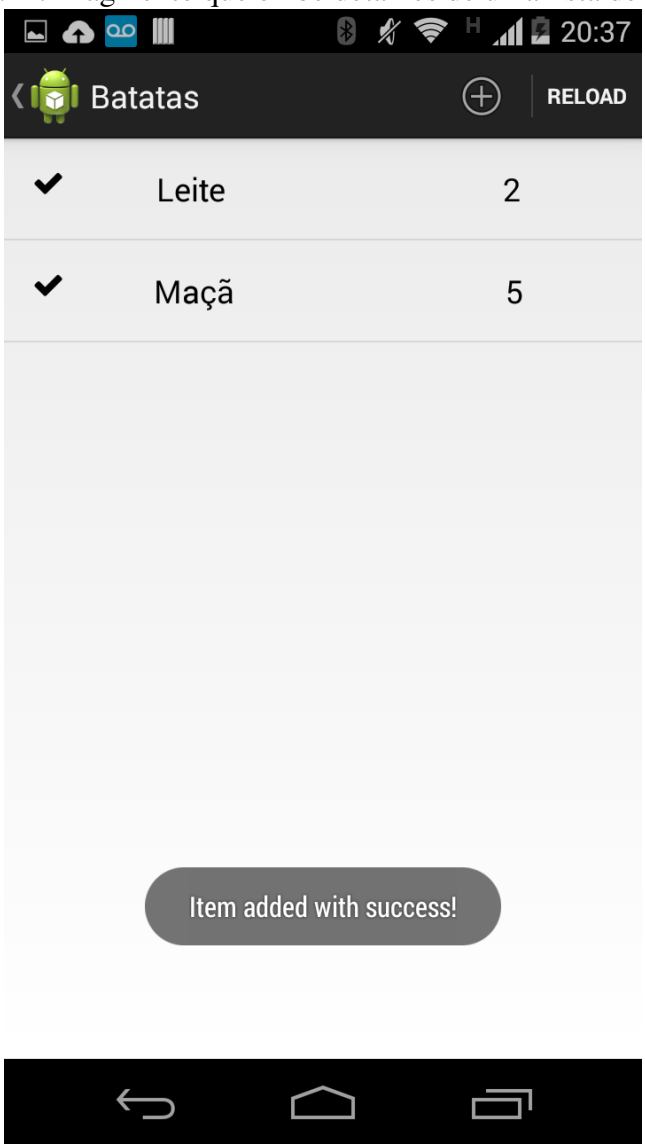


Figura 4.12: Definição do Cliente REST para Android

```
package com.mpabegg.batatas.persistence;

import ...

@Module(injects = {
        UserMemory.class,
        ListDetailsAdapter.class
},
        library = true)
public class BatatasClientProvider {
    private static final String BATATAS_STATING_ENDPOINT = "http://batatas-staging.herokuapp.com";

    public interface BatatasClient {
        @GET("/lists")
        void allLists(Callback<List<ShoppingList>> callback);

        @GET("/lists/{list_id}")
        void getList(@Path("list_id") Long listId, Callback<ShoppingList> callback);

        @POST("/lists")
        void newList(@Body ShoppingList list, Callback<ShoppingList> callback);

        @DELETE("/lists/{list_id}")
        void deleteList(@Path("list_id") Long listId, Callback<Void> callback);

        @POST("/lists/{list_id}/items")
        void addItem(@Path("list_id") Long listId, @Body ListItem items, Callback<ListItem> callback);

        @DELETE("/lists/{list_id}/items/{item_id}")
        void deleteItem(@Path("list_id") Long listId, @Path("item_id") Long itemId, Callback<Void> callback);

        @POST("/lists/{list_id}/items/{item_id}/bought")
        void buyItem(@Path("list_id") Long listId, @Path("item_id") Long itemId, Callback<ListItem> callback);

        @DELETE("/lists/{list_id}/items/{item_id}/bought")
        void unbuyItem(@Path("list_id") Long listId, @Path("item_id") Long itemId, Callback<ListItem> callback);
    }

    @Provides
    BatatasClient provideBatatasStaging() {
        final RestAdapter restAdapter = new RestAdapter.Builder().setClient(new OkClient()).setEndpoint(BATATAS_STATING_ENDPOINT).build();
        return restAdapter.create(BatatasClient.class);
    }
}
```

ponto de acesso da API REST é representado através de um método Java. Esses métodos definem qual método HTTP deve ser executado quando um recurso é acessado, qual o caminho do recurso do recurso que deseja-se acessar além de eventuais parâmetros necessários à chamada REST.

Como pode-se observar na Figura 4.12, a definição de qual método HTTP e qual o caminho relativo do recurso desejado é feita através de anotações (ORACLE, 2014) Java. Cada anotação representa um método HTTP e a String passada como parâmetro à anotação indica o caminho ao qual se deseja acessar. Junto do caminho pode-se indicar a presença de parâmetros necessários à uma requisição HTTP. Quando necessários, parâmetros devem ser representados como simples parâmetros de métodos JAVA, bastando o uso de uma anotação que indica à biblioteca como este parâmetro deve ser utilizado. No caso do método *getList* é preciso que o identificador da lista seja passado como parâmetro, isso é feito com a indicação da existência de tal parâmetro no caminho da requisição e do parâmetro *listId* do método Java, anotado como um parâmetro de caminho, através da anotação *@Path*.

Como já foi mencionado na Seção 2.1.3, requisições de rede não devem ser feitas na *thread* principal em aplicações Android. Para indicar essa imposição à biblioteca Retrofit (2014a), todos os métodos que definimos têm retorno do tipo *void*, ou seja, não retornam nenhum valor, além disso, existe um parâmetro extra que é passado a esses métodos cujo papel é definir qual ação deve ser executada ao final de uma requisição, o *callback*. Este parâmetro deve ser uma instancia de um objeto do tipo Callback, que por sua vez pode

Figura 4.13: Implementação de um Callback

```
batatas.getList(getCurrentListId(), new Callback<ShoppingList>() {
    @Override
    public void success(ShoppingList shoppingList, Response response) {
        listDetailsAdapter = new ListDetailsAdapter(getActivity(), R.layout.layout_list_item_item, shoppingList.allItems());
        setListAdapter(listDetailsAdapter);
        setupListView(getListView(), listDetailsAdapter);
        setListShown(true);
    }

    @Override
    public void failure(RetrofitError error) {
        Toast.makeText(getActivity(),
                "Could not fetch List.", Toast.LENGTH_LONG).show();
    }
});
```

ser parametrizado com o tipo de objeto esperado como retorno da requisição HTTP. A Figura 4.13 apresenta um exemplo de como *callbacks* são implementados.

Além da definição do cliente para a API REST, a Figura 4.12 apresenta detalhes de como uma instância do cliente pode ser obtida. Isto é feito através da classe Builder, fornecida pela biblioteca, que permite que algumas configurações adicionais sejam feitas, como por exemplo, o endereço do servidor onde o serviço está disponível. Mais detalhes sobre podem ser obtidos em (SQUARE, 2014a). As demais anotações presentes são provenientes da biblioteca Dagger (SQUARE, 2014b), entretanto seu funcionamento não faz parte do escopo deste trabalho.

4.4 Análise dos resultados obtidos

Como pode ser observado na Tabela 4.1, a solução desenvolvida neste trabalho não contém algumas das funcionalidades existentes em outras soluções como, por exemplo, o controle de usuários. Apesar destas limitações, oferece uma contribuição importante no que diz respeito ao uso de smartphones para facilitar o gerenciamento de listas de compras. A solução implementada permite que o gerenciamento colaborativo de listas de compras possa ser feito de maneira simples e direta. Essa caraterísca está presente em apenas um dos aplicativos estudados.

Nenhum dos aplicativos semelhantes existentes é uma solução de código aberto. Embora apresentem um número grande de funcionalidades, outros desenvolvedores não podem utilizar os conhecimentos adquiridos durante o desenvolvimento desses aplicativos. Além do aplicativo Android e servidor resultantes deste trabalho, os detalhes de implementação aqui apresentados e disponíveis no código fonte do servidor permitem que soluções semelhantes possam ser facilmente desenvolvidas com o propósito de resolver outros problemas. Ou seja, as técnias e ferramentas apresentadas neste trabalho podem ser utilizadas para que ainda mais soluções para outros problemas possam ser desenvolvidas.

Tabela 4.1: Comparação deste trabalho com aplicativos semelhantes

Funcionalidades/Aplicativo	Lista de Compras	Shopping List	Meu Carrinho	Out of Milk	Este Trabalho
Múltiplas Listas de compras	**Sim**	**Sim**	**Sim**	**Sim**	**Sim**
Compartilhamento de listas	**Sim**	**Sim**	**Sim**	**Sim**	**Sim**
Digitação facilitada de itens	**Sim**	**Sim**	Não	**Sim**	Não
Categorização de Itens	Não	**Sim**	Não	Não	Não
Registro de Preço	Não	Não	**Sim**	**Sim**	Não
Leitura de Código de Barras	Não	Não	**Sim**	**Sim**	Não
Suporte a múltipla plataformas	Não	Não	**Sim**	Não	Não
Edição colaborativa de Listas	Não	Não	**Sim**	Não	**Sim**
Controle de Usuários	**Sim**	**Sim**	**Sim**	**Sim**	Não
Código Aberto	Não	Não	Não	Não	**Sim**

5 CONCLUSÃO

Este trabalho apresentou a criação de uma fundação que pode ser evoluída com o propósito de resolver um problema presente no cotidiano de várias pessoas. O uso de dispositivos móveis está cada vez mais presente na vida da população e hábitos simples estão sendo substituídos por soluções inteligentes que utilizam o poder crescente destes dispositivos.

Por se tratar de um protótipo, algumas das funcionalidades existentes em outras soluções não estão presentes neste trabalho, entretanto podem ser facilmente implementadas de maneira colaborativa visto que todo o código que implementa o servidor é aberto e está disponível em https://github.com/mpabegg/batatas. A evolução de clientes em diversas plataformas é possível e facilitada devido à interface simples e uniforme oferecida pela API desenvolvida. Basta que o cliente tenha uma conexão com a internet para que a comunicação com o servidor possa ser implementada utilizando-se métodos do protocolo HTTP.

Por fim, mesmo com a presença de algumas limitações, o aplicativo construído pode ser utilizado de maneira satisfatória e resolve o problema proposto, ou seja, o gerenciamento colaborativo de listas de compras. Modificações feitas no estado de listas de compras ficam imediatamente disponíveis no servidor e, por consequência, em todos os clientes implementados. O foco dado para a colaboração entre pessoas durante a implementação da solução preenche uma lacuna existente em outros aplicativos cujo propósito principal é o gerenciamento de listas de compras. Mesmo aplicações comercias não apresentam essa possibilidade, o que evidencia o valor da solução apresentada neste trabalho.

5.1 Trabalhos Futuros

Durante a fase de estudo de escopo deste trabalho ficou claro que várias das funcionalidades presentes em aplicativos semelhantes precisam estar presentes para que o aplicativo implementado possa ser, de fato, utilizado pelo grande público. Dentre estas funcionalidades, destaca-se o controle de usuários e acesso a listas de compras. É possível implementar controle de usuários utilizando serviço existentes, como Google e Facebook,

o que remove complexidade desnecessária do servidor.

Além disso, o código do aplicativo Android ainda não se encontra disponível pois não se encontra em um estado apropriado para isso. Cobertura do código por testes automatizados é fundamental para que possa ser lançado ao público de maneira segura. Isso é um problema pequeno, visto que a natureza REST da API do servidor permite que diversas aplicações clientes possam ser desenvolvidas.

Em relação a suporte multiplataforma, é necessário que sejam criados mais clientes que interagem com o servidor. Duas das plataformas mais importantes são Web e iOS.

A restrição mais relevante presente neste trabalho é a necessidade de que exista um servidor que armazena os dados e oferece acesso à API REST. A resolução deste problema é fundamental para que o resultado aqui obtido possa ser completamente aberto e gratuito. O autor acredita que este é o problema mais interessante a ser estudado, entretanto, não é parte do escopo proposto para este trabalho.

REFERÊNCIAS

BECK, K. **Test Driven Development**: by example. [S.l.]: Addison-Wesley Professional, 2001.

BRASIL, I. **IDC Releases**. Disponível em: <http://br.idclatin.com/releases/news.aspx?id=1613>. Acesso em: novembro 2014.

CAPIGAMI. **Out of Milk**. Disponível em: <https://play.google.com/store/apps/details?id=com.capigami.outofmilk>. Acesso em: junho 2014.

COMMUNITY, R. **Ruby Programming Language**. Disponível em: <https://www.ruby-lang.org/en/>. Acesso em: outubro 2014.

FELLER, N. J. **Estendendo Rest-Unit**: geração baseada em u2tp de drivers e dados de teste para restful web services. 2010. Trabalho Individual — Instituto Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

FIELDING, R. T. **Architectural Styles and the Design of Network-based Software Architectures**. 2000. Tese (Doutorado) — University of California, Irvine.

GOOGLE. **Developer**. Disponível em: <http://developer.android.com/index.html>. Acesso em: outubro 2014.

GOOGLE. **NetworkOnMainThreadException**. Disponível em: <http://developer.android.com/reference/android/os/NetworkOnMainThreadException.html>. Acesso em: outubro 2014.

GOOGLE. **Play Store**. Disponível em: <https://play.google.com/store>. Acesso em: outubro 2014.

GOOGLE. **Fragments**. Disponível em: <https://developer.android.com/guide/components/fragments.html>. Acesso em: novembro 2014.

GOOGLE. **Starting an Activity**. Disponível em: <https://developer.android.com/training/basics/activity-lifecycle/starting.html>. Acesso em: outubro 2014.

GOOGLE. **Managing the Activity Lifecycle**. Disponível em: <https://developer.android.com/training/basics/activity-lifecycle/index.html>. Acesso em: outubro 2014.

GOOGLE. **List Adapters**. Disponível em: <https://developer.android.com/reference/android/widget/ListAdapter.html>. Acesso em: novembro 2014.

HIBBS, C.; JEWETT, S.; SULLIVAN, M. **The Art of Lean Software Development**. [S.l.]: O'Reilly Media, 2009.

JBOSS. **What is Object/Relational Mapping?** Disponível em: <http://hibernate.org/orm/what-is-an-orm/>. Acesso em: novembro 2014.

JULLIEN, M. D. O. **Candy Castle: um jogo sério para pacientes com diabetes**. 2013. Trabalho Individual — Instituto Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

KERBER, A. L. **Protótipo de um conversor MusicXML para MIDI usando Android**. 2012. Trabalho Individual — Instituto Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

KIWI3. **Lista de Compras**. Disponível em: <https://play.google.com/store/apps/details?id=com.shoppinglist>. Acesso em: junho 2014.

LECHETA, R. R. **Google Android**: aprenda a criar aplicacções para dispositivos móveis com o android sdk. [S.l.]: São Paulo: Novatec Editora, 2010.

MACALÃO, P. R. **Check in Poa: um aplicativo android para turistas em porto alegre**. 2013. Trabalho Individual — Instituto Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

MAKEMUSIC. **Music XML**. Disponível em: <http://www.musicxml.com/for-developers/>. Acesso em: outubro 2014.

MARCON, D. M. **Um Sistema Android para Gerenciamento de Roteiros Turísticos**. 2013. Trabalho Individual — Instituto Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

MEUCARRINHO. **Lista de Compras - MeuCarrinho**. Disponível em: <https://play.google.com/store/apps/details?id=com.meucarrinho>. Acesso em: junho 2014.

ORACLE. **Annotations**. Disponível em: <https://docs.oracle.com/javase/tutorial/java/annotations/>. Acesso em: novembro 2014.

PRODUCTIONS, D. **Shopping List**. Disponível em: <https://play.google.com/store/apps/details?id=com.DramaProductions.Einkaufen5>. Acesso em: junho 2014.

RICHARDSON, L.; RUBY, S. **RESTful Web Services**. [S.l.]: O'Reilly Media, 2007.

SILVA, T. R. D. **PyRester**: uma abordagem baseada em modelos u2tp para geração de código de teste unitário para restful web services. 2011. Trabalho Individual — Instituto Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

SINATRA. **Sinatra**. Disponível em: <http://www.sinatrarb.com/>. Acesso em: outubro 2014.

SOUZA, M. V. **SACI: sistema de apoio à coleta de informações**. 2014. Trabalho Individual — Instituto Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

SQUARE, I. **Retrofit**. Disponível em: <http://square.github.io/retrofit/>. Acesso em: novembro 2014.

SQUARE, I. **Dagger**. Disponível em: <http://square.github.io/dagger/>. Acesso em: novembro 2014.

TUTORIALSPOINT. **Android Architecture**. Disponível em: <https://docs.oracle.com/javase/tutorial/java/annotations/>. Acesso em: novembro 2014.

TWITTER. **Documentation**. Disponível em: <https://dev.twitter.com/overview/documentation>. Acesso em: outubro 2014.

UZEJKA, G. D. M. **MODELANDO UM CLIENTE VOIP INTELIGENTE PARA A PLATAFORMA ANDROID**. 2011. Trabalho Individual — Instituto Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

VOGEL, L. **Android Development - Tutorial**. Disponível em: <http://www.vogella.com/tutorials/Android/article.html>. Acesso em: novembro 2014.

W3. **HTTP 1.1**. Disponível em: <http://www.w3.org/Protocols/rfc2616/rfc2616.html>. Acesso em: novembro 2014.

WEBBER, J.; PARASTATIDIS, S.; ROBINSON, I. **REST in Practice**: hypermedia and systems architecture. [S.l.]: O'Reilly Media, 2010.