UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
DEPARTAMENTO ACADÊMICO DE INFORMÁTICA
CURSO DE ESPECIALIZAÇÃO EM TECNOLOGIA JAVA

JAKSON DOS SANTOS BRITZKE

CONTROLE FINANCEIRO WISECASH

MONOGRAFIA DE ESPECIALIZAÇÃO

PATO BRANCO
2017

JAKSON DOS SANTOS BRITZKE

CONTROLE FINANCEIRO WISECASH

Monografia de especialização apresentada na disciplina de Metodologia da Pesquisa, do Curso de Especialização em Tecnologia Java, do Departamento Acadêmico de Informática, da Universidade Tecnológica Federal do Paraná, Câmpus Pato Branco, como requisito parcial para obtenção do título de Especialista.

Orientador: Prof. Vinicius Pegorini

PATO BRANCO
2017

MINISTÉRIO DA EDUCAÇÃO
Universidade Tecnológica Federal do Paraná
Câmpus Pato Branco
Departamento Acadêmico de Informática
Curso de Especialização em Tecnologia Java

UTFPR

TERMO DE APROVAÇÃO

CONTROLE FINANCEIRO WISECASH

por

JAKSON DOS SANTOS BRITZKE

Este trabalho de conclusão de curso foi apresentado em 11 de outubro de 2017, como requisito parcial para a obtenção do título de Especialista em Tecnologia Java. Após a apresentação o candidato foi arguido pela banca examinadora composta pelos professores Vinicius Pegorini (Orientador), Ives Rene V. Pola e João Gilherme B. Pichetti membros da banca. Em seguida foi realizada a deliberação pela banca examinadora que considerou o trabalho aprovado.

Vinicius Pegorini
Prof. Orientador (UTFPR)

Ives Rene V. Pola
Banca (UTFPR)

João Guilherme B. Pichetti
Banca (UTFPR)

Robison Cris Brito
Coordenador da IV Especialização
em Tecnologia Java

A Folha de Aprovação assinada encontra-se na Coordenação do Curso.

AGRADECIMENTOS

Primeiramente agradeço a Deus, segundo lugar meus familiares pelo apoio durante a esta jornada.

Aos professores, em especial ao meu orientador Vinicius Pegorini por toda atenção, auxílios e por contribuir sempre com sugestões e opiniões favorecendo na minha tomada de decisão.

Agradeço a todos que direta ou indiretamente me auxiliaram na construção deste trabalho.

RESUMO

BRITZKE, Jakson dos Santos. Controle financeiro Wisecash. 2017. 55 f. Monografia (Trabalho de especialização) – Departamento Acadêmico de Informática, Universidade Tecnológica Federal do Paraná, Câmpus Pato Branco. Pato Branco, 2017.

O Controle financeiro Wisecash é uma aplicação *Web* que foi que desenvolvida inteiramente baseada na necessidade em que as pessoas possuem em controlar sua vida financeira, geralmente por falta de tempo ou por não terem disciplina em se organizar financeiramente. Pensando nesta necessidade e também considerando que atualmente a maioria das pessoas utilizam a *Web* para realizar compras ou pagamentos, uma aplicação nesse formato viabilizará um melhor controle das finanças. O desenvolvimento utilizou tecnologias *Java 8*, *Spring*, *AngularJS*, *BootStrap* e *PostGreSQL*. O sistema permite ao usuário realizar os lançamentos de gastos ou ganhos (parcelados ou não), gerenciar os lançamentos, realizar transferências entre contas, definição de metas, facilitando a consulta de dados com facilidade.

Palavras-chave: Controle Financeiro. Aplicação Web. AngularJS. REST.

ABSTRACT

BRITZKE, Jakson dos Santos. Wisecash financial control. 2017. 55 f. Monograph (Specialization work) - Academic Department of Informatics, Federal Technological University of Paraná, Campus Pato Branco. Pato Branco, 2017.

Wisecash Financial Control is a Web application that was developed entirely based on people needs in controlling their financial lives, usually due to lack the of time or the discipline to self-organize financially. Considering that many people use the Web to shop and to perform payments, a financial application will improve the control of their finances. The development of the web application was made using Java 8, Spring Framework, AngularJS, BootStrap and PostGreSQL technologies. The system allows the users to control their expenses or gains, to make transfers bank transfers, set goals and view reports about their financial data.

Keywords: Financial Control. Web Application. AngularJS. REST.

Lista de Figuras

LISTA DE QUADROS

LISTAGENS DE CÓDIGOS

Lista de Siglas

API	*Application Programming Interface*
CSS	*Cascading Style Sheets*
HTML	*HyperText Markup Language*
IDE	*Integrated Development Environment*
JPA	*Java Persistence API*
JVM	*Java Virtual Machine*
SYSML	*Systems Modeling Language*
UML	*Unified Modeling Language*
XML	*Extensible Markup Language*
CRUD	*Create, Read, Update, Delete*
JSON	*JavaScript Object Notation*
RIA	*Rich Internet Application*
HTTP	*Hypertext Transfer Protocol*
MVC	*Model View Controller*
SPA	*Single Page Application*
AJAX	*Asynchronous Javascript and XML*

SUMÁRIO

1 INTRODUÇÃO

Atualmente é comum que as pessoas utilizem algum tipo de controle para suas finanças, seja ele baseado em anotações em papel, uma planilha simples ou realizado por meio de um aplicativo. Por meio do planejamento é possível conhecer, em detalhes, os ganhos, além de aprender a poupar, gastar adequadamente e controlar as finanças para atingir os objetivos pretendidos (JUNIOR; GARCIA, 2005). As alternativas facilitam a gestão do controle das despesas frente às receitas, porém dependendo do seu formato demandam de um processo dispendioso e, muitas vezes, trazem retorno limitado, dificultando, e até impossibilitando uma visão mais ampla da real situação financeira.

A gestão financeira pessoal é baseada em vários lançamentos diários, que em certo ponto podem se tornar complexos de serem controladas. Sendo assim torna-se indispensável um sistema computacional capaz de atender essa demanda, com intuito de disponibilizar informações confiáveis e úteis aos seus utilizadores, segundo Macedo Júnior (2007, p. 26): “é o processo de gerenciar seu dinheiro com o objetivo de atingir a satisfação pessoal”.

Uma aplicação *web* que apresente uma interface simplista, permitindo que qualquer pessoa com o mínimo de conhecimento na área de tecnologia possa utilizar, tem por finalidade suprir essas necessidades. O controle financeiro pessoal pode ser considerado simples, mas pode tornar-se complexo e desestruturado com o passar do tempo se não adequadamente gerenciado.

Em um panorama geral, é possível notar a dificuldade que as pessoas têm em controlar sua vida financeira, seja ela por não terem tempo para se organizar ou pela falta de uma ferramenta que seja de fácil manuseio, entre outros. Essa dificuldade pode resultar em pessoas que acabam gastando muito mais do que ganham, simplesmente por não controlarem os seus gastos.

Considerando que as aplicações *web* têm alcançado um grande número de usuários devido à grande facilidade de acesso à Internet seja ela a partir de computadores ou *smartphones,* é possível notar que em um futuro bem próximo a maiorias das aplicações serão desenvolvidas para *web*. Assim, os usuários poderão acessar os sistemas de qualquer local, basta possuir acesso à Internet.

A aplicação desenvolvida como resultado deste trabalho atenderá às características e às necessidades apontadas permitindo que os usuários consigam controlar seus gastos e ganhos e planejar de acordo com os resultados que o sistema apresentará. Os detalhamentos e

as informação apresentadas de maneira organizada podem ajudar as pessoas a tomarem as decisões mais acertadas em termos de gastos a partir de suas receitas.

1.1 OBJETIVOS

Desenvolver uma aplicação web que visa suprir a necessidade dos usuários realizarem um controle mais efetivo das suas finanças pessoais.

1.1.2- OBJETIVOS ESPECÍFICOS

Para atingir as metas pretendidas neste projeto, a aplicação *Web* deverá ser capaz de gerenciar as seguintes ações:

- Gerenciar os lançamentos de Gastos e Ganhos;
- Gerenciar lançamentos Parcelados;
- Gerenciar Transferência entre contas;
- Gerenciar Metas mensais e anuais;
- O usuário será capaz de ter uma visão geral de suas movimentações financeiras;
- Extrair relatórios detalhados da situação financeira do usuário;

1.2 JUSTIFICATIVA

Em muitos casos, a dificuldade ou até mesmo a crise financeira se dá pela falta de um controle sobre as despesas e receitas, ausência de um controle sobre as contas no banco, carência de uma ferramenta que permita o usuário visualizar suas futuras contas a pagar\receber, ou até mesmo verificar qual tipo de lançamento tem tomado uma maior parte de sua renda. Muitas vezes são feitas anotações em papel no momento da despesa, mas depois essas informações acabam se perdendo, impossibilitando um controle de todos os gastos.

A opção pelo desenvolvimento de uma aplicação *web* vem diretamente ligado ao fato do usuário acessar o sistema de qualquer local, afinal, basta estar conectado à Internet que o usuário poderá controlar todos os seus lançamentos de forma inteligível e prática.

Considerando a facilidade no acesso à Internet, outro ponto que é de extrema relevância é o uso dos *smartphones*. Baseado nesse segundo ponto é importante salientar que o desenvolvimento da aplicação *web* apresentada neste trabalho estará adepto ao uso em smarpthones devido ao *design responsivo*, que nada mais é que uma técnica de estruturação *HTML* e *CSS* que se adequam ao navegador utilizado pelo usuário.

Portanto, o sistema proposto auxiliará no controle financeiro pessoal, afinal a gestão financeira aparenta ser algo simples, porém em muitos casos é causadora de grandes transtornos.

1.2 ESTRUTURA DO TRABALHO

Este texto está organizado em capítulos, onde primeiramente é apresentado a ideia da aplicação desenvolvida incluindo o objetivo geral, especifico e a justificativa.

No Capítulo 2 é apresentado o referencial teórico que apresenta a proposta conceitual da aplicação desenvolvida. Neste capítulo estão descritas as aplicações com interfaces ricas, denominadas *RIAs*, e um conceito primordial referente às *Single-Page-Applications.*

O Capítulo 3 relaciona as ferramentas e tecnologias utilizadas no desenvolvimento deste trabalho, onde é feita uma breve descrição sobre cada ferramenta.

No Capítulo 4 demonstra a aplicação desenvolvida, contendo a modelagem, a funcionalidade do sistema juntamente com algumas imagens e alguns códigos fontes de relevância.

No Capítulo 5 é mostrada a conclusão juntamente com as considerações finais.

2 REFERENCIAL TEÓRICO

Este capítulo tem por objetivo apresentar o referencial teórico do trabalho, que concentra-se no desenvolvimento de uma aplicação *web*, a qual futuramente será integrado a um aplicativo financeiro já disponível na Google Play.

2.1 RICH INTERNET APPLICATIONS (RIAs)

As RIAs são aplicações web que oferecem a sensibilidade, recursos e funcionalidades "ricas", que se aproximam das aplicações de *desktop*. As primeiras aplicações para a internet suportavam apenas uma interface gráfica de usuário de *Hipertext Markup Language (HTML)* básico; e embora oferecessem funções simples, as aplicações não tinham a aparência nem proporcionavam a sensação de uma aplicação *desktop*. As conexões relativamente lentas da internet em que as aplicações se baseavam conduziram ao termo "*World Wide Wait'*, ou seja, "espera" (DEITEL, Paul; DEITEL, Harvey, 2009)**.**

Para as aplicações tradicionais a alternativa encontrada por muitos desenvolvedores era uma ferramenta *back-end* que se encarregava da estrutura de *model view controller* (MVC), e as páginas do sistema eram construídas conforme demanda, necessitando de vários acessos ao servidor, em consequência a velocidade de resposta era baixa.

Ao longo dos anos o *HTML* e o *JavaScript* evoluíram muito, solucionando algumas limitações e problemas existentes. A evolução, entretanto, veio por meio do surgimento do *Asynchronous JavaScript and XML (AJAX),* conjunto de técnicas para desenvolvimento *Web* focadas em *JavaScript* dotadas de recursos que proporcionam uma interação mais rica e rápida para o usuário.

Estas aplicações geralmente possuem clientes que realizam atividades relacionadas à interface, enquanto o servidor de aplicação processa e armazena dados e simplifica as atualizações de dados enviados para o cliente (LAWTON, 2008). Essa simplificação ocorre porque apenas partes da tela que sofrem alterações são atualizadas. Essa forma de interação aumenta a rapidez de resposta da aplicação às ações do cliente porque o usuário não precisa aguardar que dados sejam movidos entre o cliente e o servidor (LAWTON, 2008).

2.1.1- SINGLE PAGE APPLICATION

Single Page Application (SPA), basicamente significa codificar menos no *server-side* e mais no *cliente-side.* Em outras palavras podemos dizer que boa parte da aplicação esta desenvolvida no lado cliente, e as aplicações neste formato fazem uma transição entre os *templates* carregados (KLAUS, Allan, 2016).

Uma aplicação SPA trabalha sem atualizar a página e sem que o usuário viaje de uma página para outra, isso fornece um ganho relevante na experiência do usuário e ainda ganha-se em performance, pois delegamos lógica no lado cliente e isso acaba ajudando, diminuindo o tempo de requisição no *server-side* (KLAUS, Allan, 2016).

Em uma aplicação *web* tradicional, o cliente (navegador) inicia a comunicação com o servidor solicitando uma página e o servidor processa e envia o HTML com as informações para o cliente. Em interações subsequentes com a página se o usuário navega para um outro *link* ou envia um formulário com dados, uma nova solicitação é enviada para o servidor e o fluxo começa novamente. O servidor processa o pedido e envia uma nova página ao navegador em resposta a ação solicita pelo cliente (SAXENA, Rahul, 2014). Processo demonstrado na Figura 1.

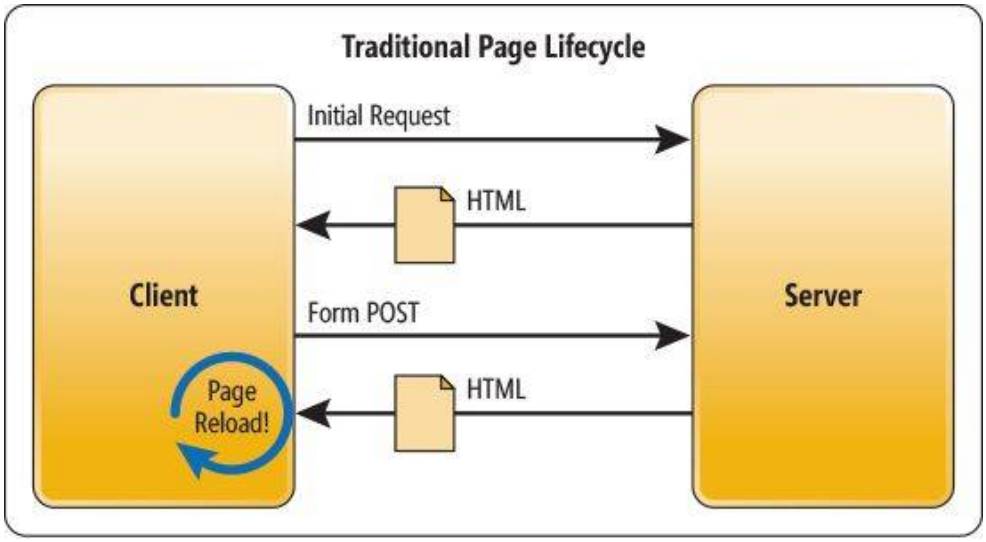


Figura 1 – Ciclo de vida de uma aplicação Web tradicional
Fonte: Saxena (2014)

No ciclo de vida de uma *Single-Page-Application,* a página inteira é carregada no navegador após a primeira solicitação, mas as interações posteriores ocorrem usando solicitações em *Ajax.* Isso significa que o navegador deve atualizar apenas a parte da página que mudou, não havendo a necessidade de recarregar a página inteira. Essa abordagem reduz

o tempo gasto para a aplicação responder as ações dos usuários (SAXENA, Rahul, 2014), a Figura 2 representa este ciclo.

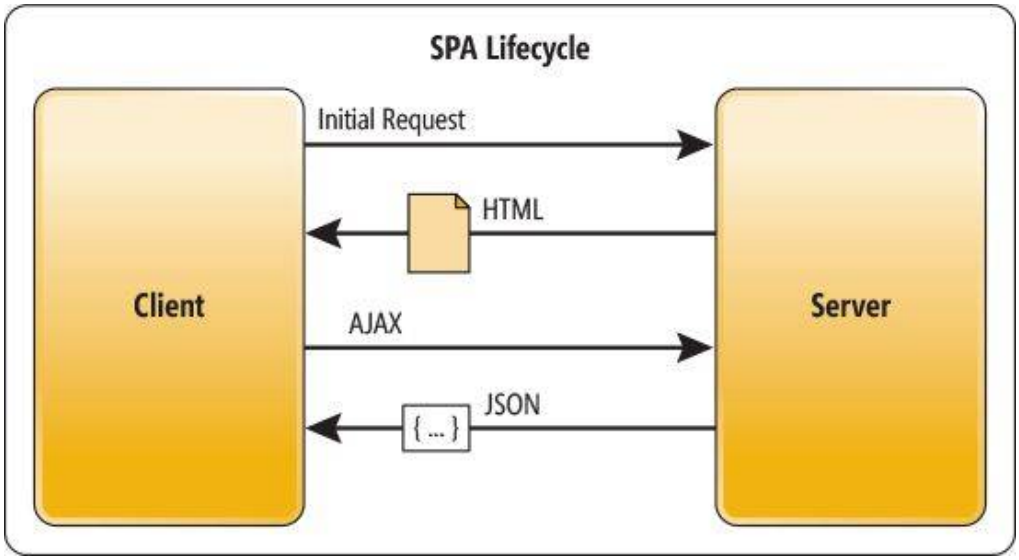


Figura 2 - Ciclo de vida de uma aplicação Single-Page-Application
Fonte: Saxena (2014)

A Figura 3, mostra uma pesquisa sobre a busca pelo termo referente as *Single-Page-Application* no *Google Trends*, demonstrando o nível de popularidade nos últimos 2 anos, ficando comprovado o grande sobre este assunto. O valor apresentado na pesquisa vai de uma escala de 0 a 100, onde o valor de 100 é o pico de popularidade de um termo, 50 corresponde à metade da popularidade e da mesma forma uma pontuação de 0 significa que o termo teve menos de 1% de popularidade.

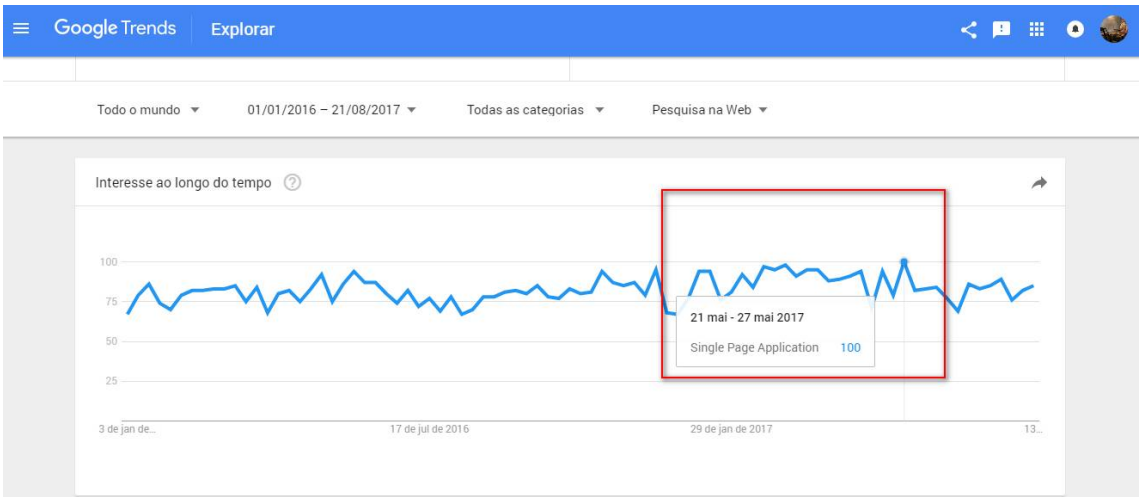


Figura 3 - Popularidade do Termo Single Page Application
Fonte: Google Trends

3 FERRAMENTAS, TECNOLOGIAS E METÓDOS

Este capítulo apresenta as ferramentas e tecnologias utilizadas no desenvolvimento deste trabalho como linguagens de programação, *frameworks* utilizados e também a definição dos métodos cobrindo desde a definição dos requisitos até a conclusão final do projeto.

3.1- FERRAMENTAS E TECNOLOGIAS

O Quadro 1 apresenta as ferramentas e tecnologias utilizadas na modelagem e no desenvolvimento da aplicação *web*.

Ferramenta / Tecnologia	Versão	Disponível em	Aplicação
Java™ SE	8u131	http://www.oracle.com/technetwork/java/javase/downloads/	Linguagem para desenvolvimento da aplicação
Visual Paradigm	13.2	https://www.visual-paradigm.com/download/	Modelagem do sistema
Bootstrap	3.3.7	http://getbootstrap.com	*Framework* de estilizações de páginas por meio de CSS.
IntelliJ IDEA	2017.2.1	https://www.jetbrains.com/idea/download/	IDE para desenvolvimento da aplicação
Spring Boot	1.5.4	https://projects.spring.io/spring-boot/	Processo de configuração e publicação da aplicação.
Spring Data JPA	1.5.4	https://projects.spring.io/spring-data-jpa/	Persistência dos dados
Spring Security	1.5.4	https://projects.spring.io/spring-security/	Framework de Segurança
AngularJS	1.6.5	https://angularjs.org/	Biblioteca JavaScript utilizada no desenvolvimento da interface.
PostgreSQL	9.6	https://www.postgresql.org/download	Armazenagem de dados
JasperSoft Studio Community Edition	6.4.0	http://community.jaspersoft.com/download	Desenvolvimento de Relatórios

Quadro 1 – Tecnologias e ferramentas utilizadas na modelagem e na implementação do Sistema

3.1.2- JAVA SE

O Java SE é uma ferramenta de desenvolvimento para a plataforma Java, possui o ambiente necessário para criação e execução de aplicações Java, onde a mesma já possui a

máquina virtual Java (JVM), compilador e as APIs necessárias juntamente com outros utilitárias oferecendo uma melhor funcionalidade.

Como neste desenvolvimento necessita destas funcionalidades citadas foi utilizado a versão 8 do Java, juntamente com outros frameworks que serão citados logo em seguida. Esta versão está disponível no site da Oracle, empresa que gerencia a linguagem atualmente.

A Figura 4 exibe a tela sobre a versão do Java.

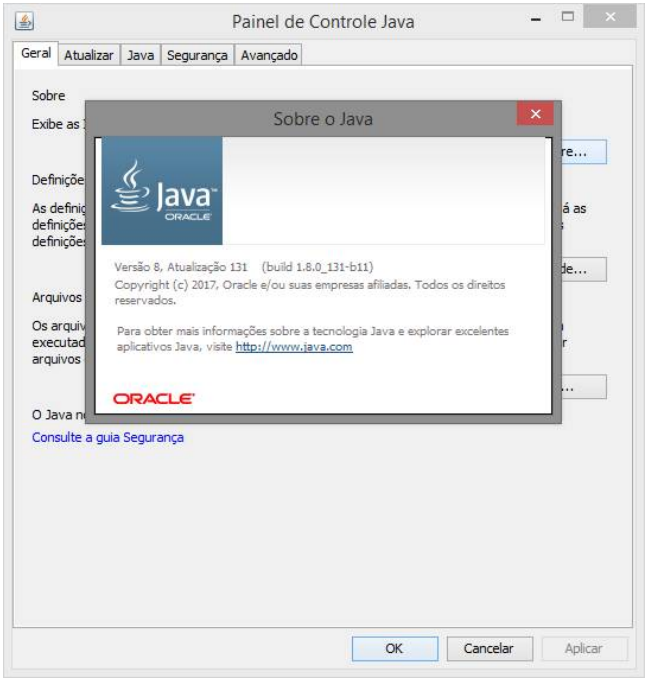


Figura 4 - Java SE

3.1.3 - VISUAL PARADIGM

Visual Paradigm for UML (*Unified Modeling Language*) é uma ferramenta de modelagem para diagramas UML. Essa ferramenta fornece suporte para gerenciamento de casos de uso, classes, diagrama de requisitos SysML (*Systems Modeling Language*) e projeto de banco de dados com diagrama de entidades e relacionamentos organizando os fluxos de trabalho, além de gerar documentação (VISUAL PARADIGM, 2013).

Na Figura 5 pode ser observada a tela inicial do software, que para este trabalho foi utilizada a versão não comercial.

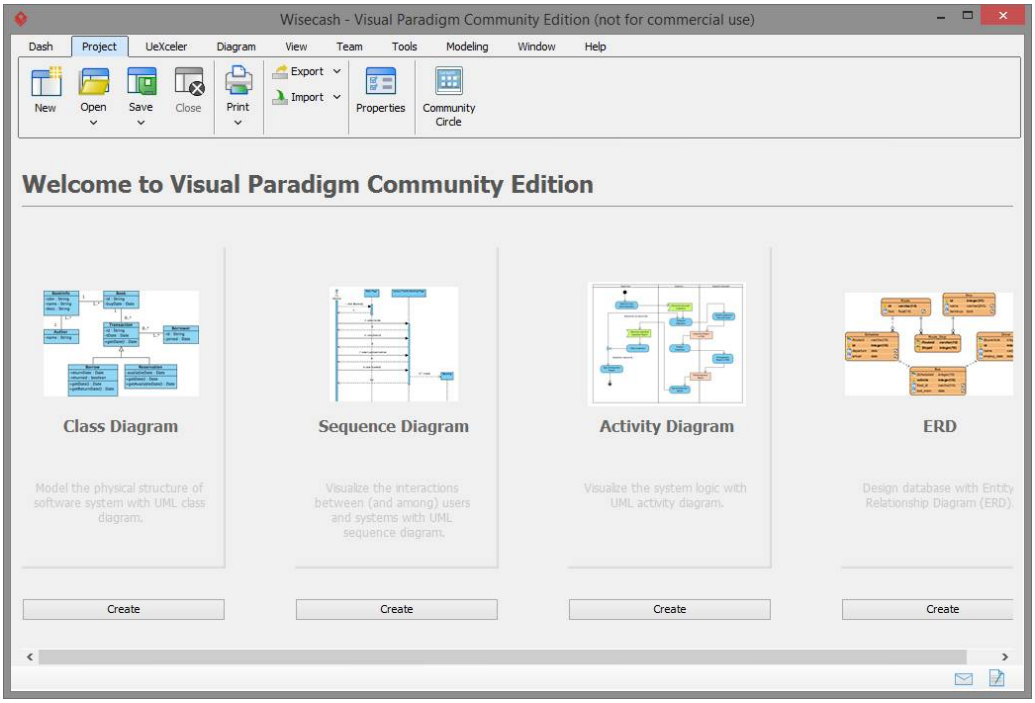


Figura 5 - Visual Paradigm UML Community Edition

3.1.4 - BOOTSTRAP

Bootstrap é uma coleção de vários elementos e funções personalizáveis para projetos *web*, previamente elas são empacotadas em uma única ferramenta, porém para o desenvolvimento de aplicações é possível realizar a escolha dos elementos desejados.

Esses elementos nada mais são do que elementos personalizáveis em uma combinação de *HTML, CSS e JAVASCRIPT*. Dentro desses benefícios com respeito ao *framework* entra o conceito de responsividade, onde ao utilizar determinados elementos e classes a aplicação torna-se totalmente passível a responsividade, permitindo assim que usuários possam acessar a aplicação de um smartphone com um alto grau de usabilidade.

Além desta ferramenta, neste projeto foi utilizado algumas diretivas específicas para o desenvolvimento da interface que também fazem parte da biblioteca *Bootstrap*, porém específicas para o *AngularJs*. Os componentes utilizados foram:

- *DatePicker Popup* – Que são os calendários trazendo uma maior usabilidade para o usuário final;
- *Progressbar* – São as barras de progresso também buscando usabilidade e facilidade no entendimento;

- *UI Select* – São os campos de escolha ou múltiplas escolhas, buscando também facilidade no uso da aplicação.

No geral o *Bootstrap* é um conjunto de ferramentas de código aberto ou seja, ele é gratuito e pode ser baixado em http://getbootstrap.com e https://angular-ui.github.io/bootstrap/ que são as diretivas específicas para uso juntamente com o *AngularJS*.

3.1.5 - INTELLIJ IDEA

O *IntelliJ* IDEA é um ambiente de desenvolvimento integrado Java para o desenvolvimento de software a qual foi criada pela *JetBrains*, uma empresa de desenvolvimento líder em criação de *softwares* inteligentes sempre buscando elevar a produtividade.

A primeira versão do *Intellij* foi lançada em 2001, e em 2010 recebeu relevantes resultados comparando-se com as demais tecnologias (*Eclipse, NetBeans* e *JDeveloper*). A partir deste resultado em 2014 a Google anunciou a primeira versão do *Android Studio*, uma IDE de código aberto específica para desenvolvimento de aplicativos Android, muito utilizada atualmente.

Atualmente, este é o IDE muito utilizada na maioria dos desenvolvedores *Java*, possuindo diversos recursos e plug-ins, além de uma interface bastante funcional conforme mostra a Figura 6.

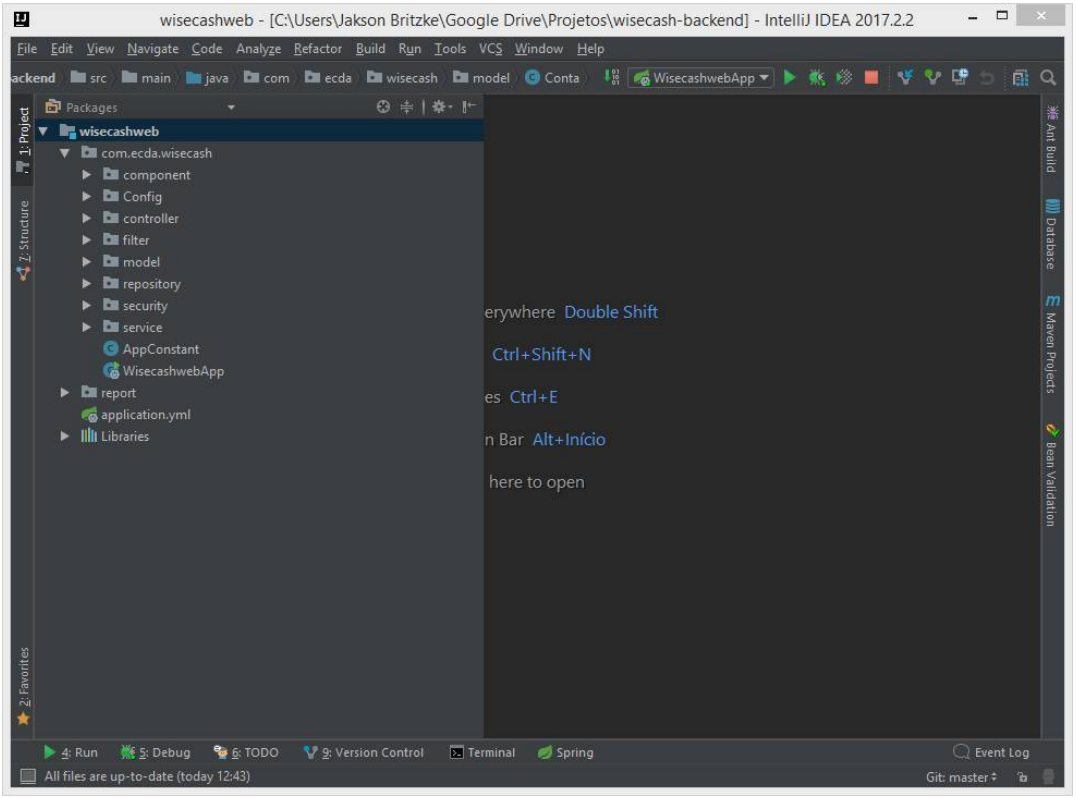


Figura 6 - IntelliJ IDEA

3.1.6 - SPRING BOOT

O *Spring boot* é um *framework*, onde seu principal intuito é reaproveitar tecnologias, ou seja, o objetivo não é trazer novas soluções a problemas que já foram resolvidos, mas sim reaproveitar as tecnologias e aumentar a produtividade do desenvolvedor.

Este *framework* é baseado em alguns princípios:

- Prover uma experiência de início de projeto rápida e direta;
- Apresentar uma visão sobre como devemos configurar os projetos *Spring*, sempre de modo flexível e suficiente para que possa ser facilmente substituída conforme os requisitos do projeto;
- Fornece vários requisitos pré-configurados como por exemplo, acesso a base de dados, métricas, servidor de aplicações embarcados, isso facilita o desenvolvimento de aplicações;

Nesse formato é minimizado ao máximo a necessidade de arquivos XML.

Sua primeira liberação foi em Outubro de 2002 onde seus criadores sempre se preocuparam em que ele fosse o mais leve e simples possível. Atualmente ele é uma camada

dentro do ecossistema *Spring*. Esse *framework* encontra-se disponibilizado no site da *Spring* https://projects.spring.io/spring-boot/.

3.1.7 - SPRING DATA JPA

O *Spring Data JPA* é um projeto que tem por objetivo, simplificar o acesso a tecnologias de armazenamento de dados. Sejam elas relacionais (*MySQL, Postgresql, etc.)* ou não (*MongoDB, Redis, etc.).* Neste projeto é utilizado um banco relacional *Postgresql* o qual será incluído mais informações no decorrer do trabalho.

O *Spring-Data* possui vários projetos integrados a ele como *Spring Data Commons*, *Spring Data Rest*, *Spring Data for Apache* e vários outros. O *framework* agiliza o desenvolvimento padronizando algumas funcionalidades, isso faz com que seja mais fácil a implementação de algumas funcionalidades básicas, um exemplo claro disso é a implementação padrão dos métodos *save, delete, findOne* entre outros.

3.1.8 - SPRING SECURITY

Baseado na sua nomenclatura já denuncia o que é o projeto *Spring Security,* é um projeto que trata da segurança das aplicações, tendo um suporte excelente para autenticação e autorização. Com poucas configurações o *Spring security,* já fornece uma autenticação via banco de dados ou mesmo por memória.

Um fato importante desse ferramental é a autorização, ou seja, por meio das permissões que são atribuídas pode-se proteger as requisições *web (*como telas do sistema por exemplo), a simples invocação de um método e até a instância de um objeto.

Outro ponto importante ao utilizar o básico do *Spring security* é que ele garante que a aplicação já está protegida contra alguns ataques como *Session fixation,* que basicamente é um ataque onde o invasor obtém a ID de sessão, conseguindo assim conectar-se a aplicação e o *Cross Site Request Forgery* que é um tipo de ataque no qual comandos não autorizados são transmitidos através de um utilizador em que o website confia (*Links, Scripts, tags)* etc.

3.1.9 - ANGULARJS

O *AngularJS* é um *framework Java Script open-source*, o qual é mantido pela Google, que auxilia na execução das aplicações chamadas *single-page-applications*, nomenclatura esta que será tratada em um novo tópico por ser uma prática relevante nos dias atuais. A filosofia deste *framework* parte de que uma programação declarativa é muito mais importante que uma programação imperativa quando se trata de desenvolvimento *web*.

A programação declarativa parte do pressuposto que você deve dizer ao computador "o que" precisa ser feito, cabendo ao computador decidir qual a melhor solução para uma determinada solicitação, já a programação Imperativa é ao contrário, ou seja, é necessário informar ao computador "como" as instruções devem ser executadas.

O angular ensina uma nova sintaxe ao navegador através de construções chamadas de diretivas, na prática nada mais é do que novas tags ou atributos de elementos para o *HTML*, *o HTML* é excelente para declaração de conteúdos estáticos porém quando necessitamos que um determinado conteúdo seja dinâmico isso torna-se restrito, este é um dos pontos que o *AngularJS* se sobressaí muito bem, pois ele permite expandir o vocabulário *HTML* na aplicação.

O *framework* trabalha com um conceito onde as alterações feitas em qualquer camada reflete na outra, ou seja, toda vez que altera-se algum dado na camada *model*, as alterações se reflete na apresentação desse dado na camada *view*, e vice-versa. O *AngularJS* é disponibilizado gratuitamente no endereço https://angularjs.org/ onde possui uma ótima documentação.

3.1.10 - POSTGRESQL

PostgreSQL é um banco de dados objeto relacional chamado de código aberto, sua estruturação é baseada em forma de tabelas (linhas e colunas) e suporte total a chaves estrangeiras. Ele é multiplataforma, ou seja, pode ser executado na maioria dos sistemas operacionais incluindo Linux, Mac OS e Windows.

Algumas características do PostgreSQL são:

- Permite Sub-Consultas;
- *Joins;*
- Gatilhos (*Triggers);*

- Tipos definidos pelo usuário;
- Funções armazenadas (*Stored Procedures*), que podem ser escritas em várias linguagens de programação (PL/PgSQL, Perl, Python, Ruby, e outras);
- Controle de concorrência multi-versão(MVCC), nesse mecanismo, os processos de leitura não bloqueiam processos de escrita e vice-versa, reduzindo drasticamente a contenção entre transações concorrentes e paralisação parcial ou completa (*deadlock*);
- Integridade Referencial;
- Backup on-line;
- Esquemas(*Schemas*);
- Áreas de armazenamento (*Tables Spaces*);
- *Commit* em duas fases entre outros recursos;

3.1.10.1 - GERENCIADOR DE BANCO - DBEAVER

Nesse trabalho exceto a criação do banco de dados, optei pela utilização de uma ferramenta externa para administração do banco de dados chamada DBeaver, devido a facilidade de uso e leveza ao manipular as informações, basicamente ele é uma ferramenta de banco de dados universal também de código aberto e multiplataforma e pode ser encontrado no endereço https://dbeaver.jkiss.org/download/. Na Figura 7 é apresentado o layout da ferramenta.

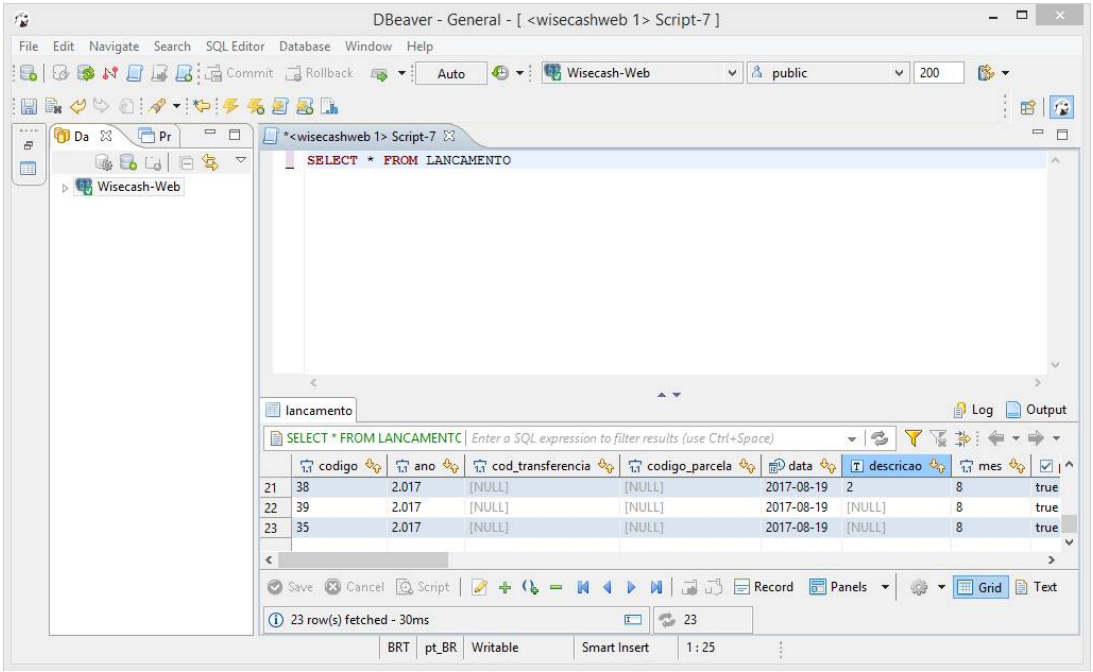


Figura 7 - DBeaver - Ferramenta de Administração de Banco de Dados

O PostgreSQL também possui sua própria ferramenta de administração, chamada pgAdmin, e pode ser encontrada em https://www.pgadmin.org/download/

3.1.11 - JASPERSOFT STUDIO COMMUNITY EDITION

O *Jaspersoft Studio*, é um gerador de relatório baseado na IDE de desenvolvimento Eclipse, que permite a criação de layouts sofisticados contendo gráficos, imagens, sub-relatórios, tabelas de referência cruzada e muito mais. Os dados podem ser acessados de várias formas, *JDBC, TableModels, Java Beans, XML, Hibernate, CSV*. Para este desenvolvimento foi utilizado o modo *Java beans*, onde o software realiza a leitura dos atributos da classe (*.class*).

Essa ferramenta permite a publicação de relatórios em vários formatos por exemplo: *PDF, RTF, XML, XLS, CSV, HTML, XHTML,* texto, *DOCX* ou *Open Office*. Na Figura 8 é representada a ferramenta utilizada, a mesma pode ser encontrada em https://community.jaspersoft.com/download

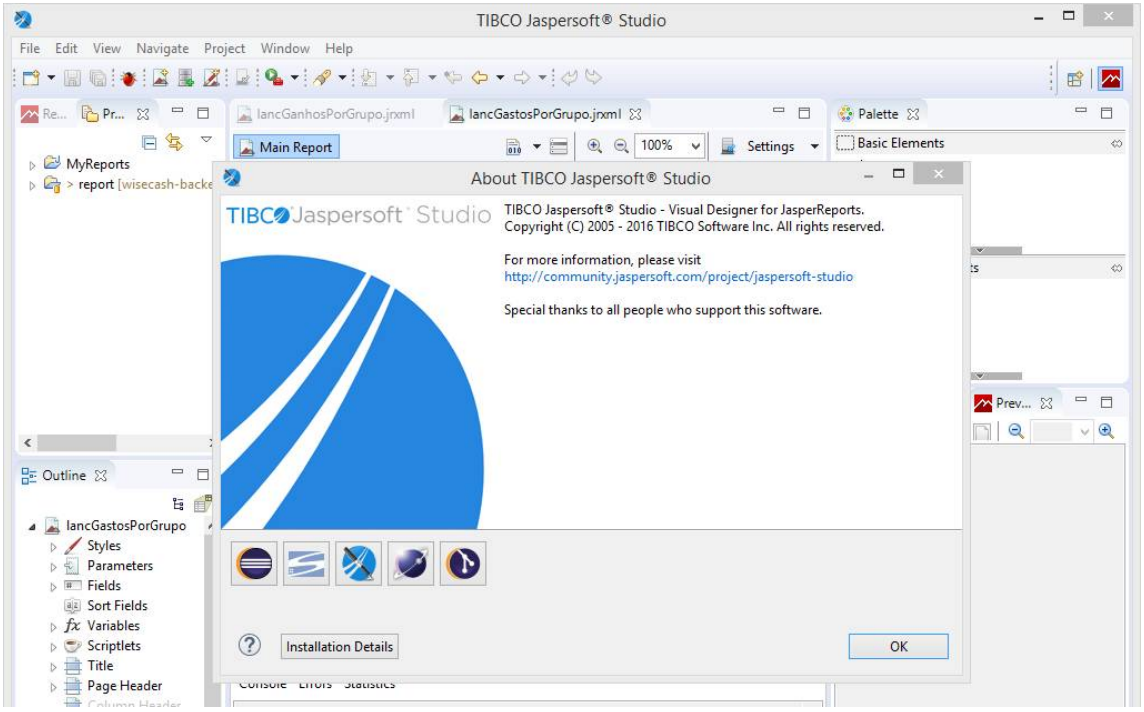


Figura 8 – Jaspersoft Studio – Gerador de Relatórios

3.2 – MÉTODO

A fim de concretizar o desenvolvimento deste trabalho alguns métodos foram utilizados, a fim de clarificar com mais detalhes os passos seguidos durante o desenvolvimento. Abaixo serão descritas as etapas utilizadas:

I. Definição do escopo do sistema a ser desenvolvido, baseado no entendimento da necessidade dos usuários, analisando as dificuldades enfrentadas no dia a dia com respeito ao controle financeiro pessoal e também considerado as funcionalidades de um *app* para Android já existente;
II. Definição das tecnologias utilizadas, ou seja, os recursos necessários para o desenvolvimento do sistema, seguido da preparação do ambiente para o desenvolvimento;
III. Análise, projeto e construção de UML, ou seja, a definição dos casos de uso, definição das classes e seus devidos atributos, relacionamentos;
IV. Implementação das funcionalidades.
V. Realização de Testes

3.2.1 – ESCOPO DO SISTEMA

A aplicação Web descrita no decorrer deste trabalho visa fornecer uma ferramenta que possibilite os usuários a realizarem o seu controle financeiro pessoal. A aplicação permite que o usuário possa controlar suas contas, por exemplo Conta Poupança, Conta Corrente, Carteira etc. Além das contas é necessário a definição de grupos que basicamente é o que define a separação dos lançamentos. Quando cita-se “Grupos”, é uma opção que fornece o usuário a possibilidade de nomear os seus gastos ou ganhos, por exemplo, Comida, Lanches, Salário, Estudos etc.

A partir desses lançamentos a aplicação permite que o usuário tenha uma visão objetiva com respeito as suas finanças, no *Dashboard* inicial o usuário visualiza o saldo das contas, um extrato mensal dos lançamentos do mês e também um gráfico anual mostrando o valor ganho e gasto de todos os meses do ano corrente.

É comum as pessoas realizarem compras parceladas, lançamentos estes que contém um número de parcelas fixas. Para atender essa necessidade a aplicação conta com a geração destas parcelas, onde o usuário informa o valor fixo e a aplicação realiza a geração das parcelas para os meses seguintes.

Considerando esse conjunto de dados que permite o usuário saber "*Para onde vai o seu dinheiro*" de uma forma simples e prática, existe a funcionalidade de cadastramento de metas, onde é informado o "Valor que deseja gastar", e visualmente o usuário vai acompanhando o valor da sua meta, quanto já gastou e quanto ainda pode gastar. Essa meta pode ser definida em dois formatos, a primeira delas é a mensal, ou seja, o sistema irá considerar os gastos do mês corrente abatendo ao valor que deseja gastar, e a meta anual onde define-se o valor que deseja gastar anualmente, sendo assim é demonstrado sistematicamente quanto o usuário poderá gastar por mês.

4 RESULTADO

4.1 MODELAGEM DO SISTEMA

Uma premissa fundamental para o desenvolvimento do sistema são os requisitos funcionais e não funcionais. Os requisitos funcionais têm a finalidade de agregar valor ao usuário ou facilitar o trabalho que ele desenvolve. Já os não funcionais são relacionados ao ambiente onde o sistema está inserido (requisitos operacionais, requisitos de segurança, desempenho entre outros) (MARETTI, André, 2014). Neste desenvolvimento foram definidos alguns destes requisitos conforme listados no Quadro 2 e Quadro 3.

Identificação	Nome	Descrição
01	Realizar cadastro de usuário	Possibilitar que os usuários se cadastrem na aplicação.
02	Realiza cadastro de Contas	Permitir que o usuário cadastre suas contas.
03	Realiza cadastro de Grupos	Permitir que o usuário cadastre seus grupos definindo quais são ganhos e quais são gastos.
04	Realiza Lançamentos	Permite que o usuário lance todos os seus ganhos e gastos.
05	Realiza Lançamentos recorrentes.	Possibilita o usuário lançar gastos ou ganhos recorrentes.
06	Define metas	Permitir o usuário definir metas de gasto, onde elas podem ser mensais ou anuais.
07	Realiza transferência entre contas	Possibilita o usuário realizar transferência entre contas.

Quadro 2 – Requisitos funcionais

Identificação	Nome	Descrição
01	Autenticação de usuário	Visando garantir segurança para a aplicação é utilizado um método de autenticação via token para cada requisição solicitada.
02	Primeiro Acesso	Após o usuário realizar o seu cadastro um e-mail é enviado com um link de confirmação da conta.
03	Loading nas requisições	Deverá possuir um Loading afim de informar ao usuário que alguma tarefa está sendo executada.
04	Responsividade da Aplicação	A aplicação deverá se adequar aos padrões de responsividade, permitindo ao usuário acessar o sistema a partir de dispositivos menores.
05	Tradução do Sistema	Permitir que o usuário possa alterar a linguagem (Português, Inglês e Espanhol).

Quadro 3 - Requisitos não Funcionais

Com base nos requisitos funcionais e não funcionais citados, foi desenvolvido o diagrama de caso de uso, conforme a Figura 9. No diagrama pode ser visto o ator que é o usuário da aplicação, e quais ações ele irá executar, os CRUDs (*Create*, *Read*, *Update* e *Delete*).

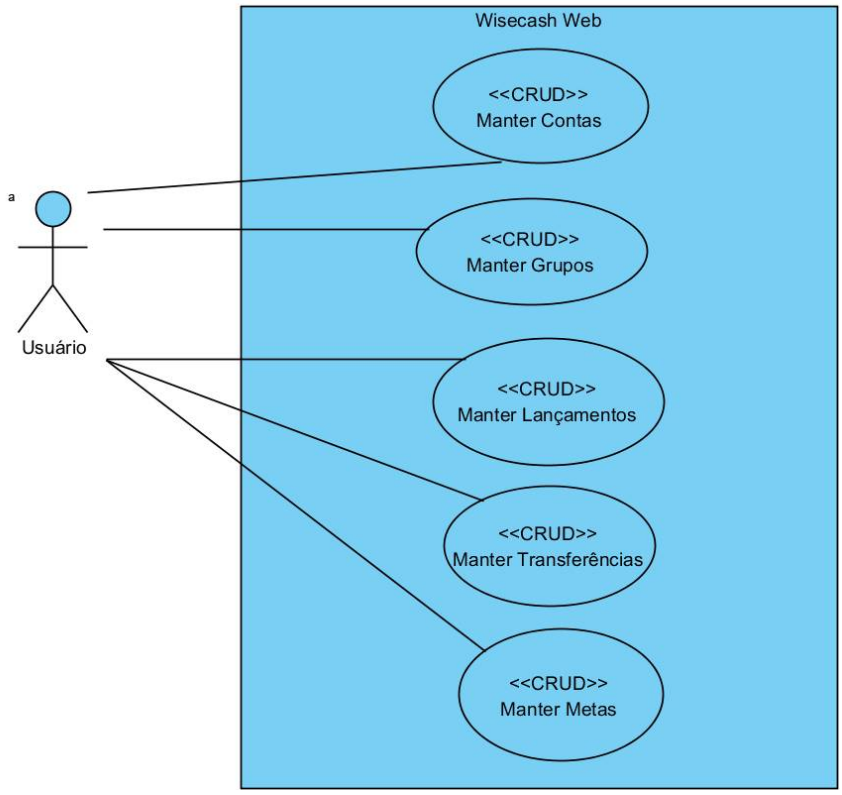


Figura 9 - Diagrama de Caso de Uso

Além do diagrama de caso de uso mostrado na Figura 9, na Figura 10 é apresentado o diagrama de classes, no qual a medida que a aplicação foi necessitando de novos atributos ou até mesmo novas classes as mesmas foram incrementadas. A classe conta possui os atributos código e descrição onde o código é chave primária, descrição é usada para identificação da conta e o usuário para realizar a separação das contas por usuário, esta mesma estrutura aplica-se para as demais classes.

Nas classes de lançamento, metas e transferências além dos atributos principais elas também possuem alguns diferenciais que se aplicam a lógica da aplicação. Na classe lançamento foi criado outros atributos necessários, por exemplo, data, valor, pago, tipo do lançamento, grupo, conta, mês, ano e usuário. Para definição do tipo do lançamento foi criado um enum, afim de definir se o lançamento é “D” para despesa, ou “R” para Receita, a mesma ideia foi utilizada para definição das metas, onde “M” para metas mensais e “A” para anuais. Outro ponto a ser destacado é o campo de mês e ano, onde foi optado por gravá-los separadamente afim de facilitar na criação de consultas personalizadas.

Além das classes demonstradas na Figura abaixo, foram criadas algumas classes que não são persistidas no banco de dados, classes estas que auxiliaram na construção de consultas

personalizadas retornando os dados formatados, estas classes foram utilizadas principalmente para desenvolvimento dos gráficos, geração dos parcelamentos. A Figura 10 representa o diagrama de classes, com suas devidas associações e as multiplicidades

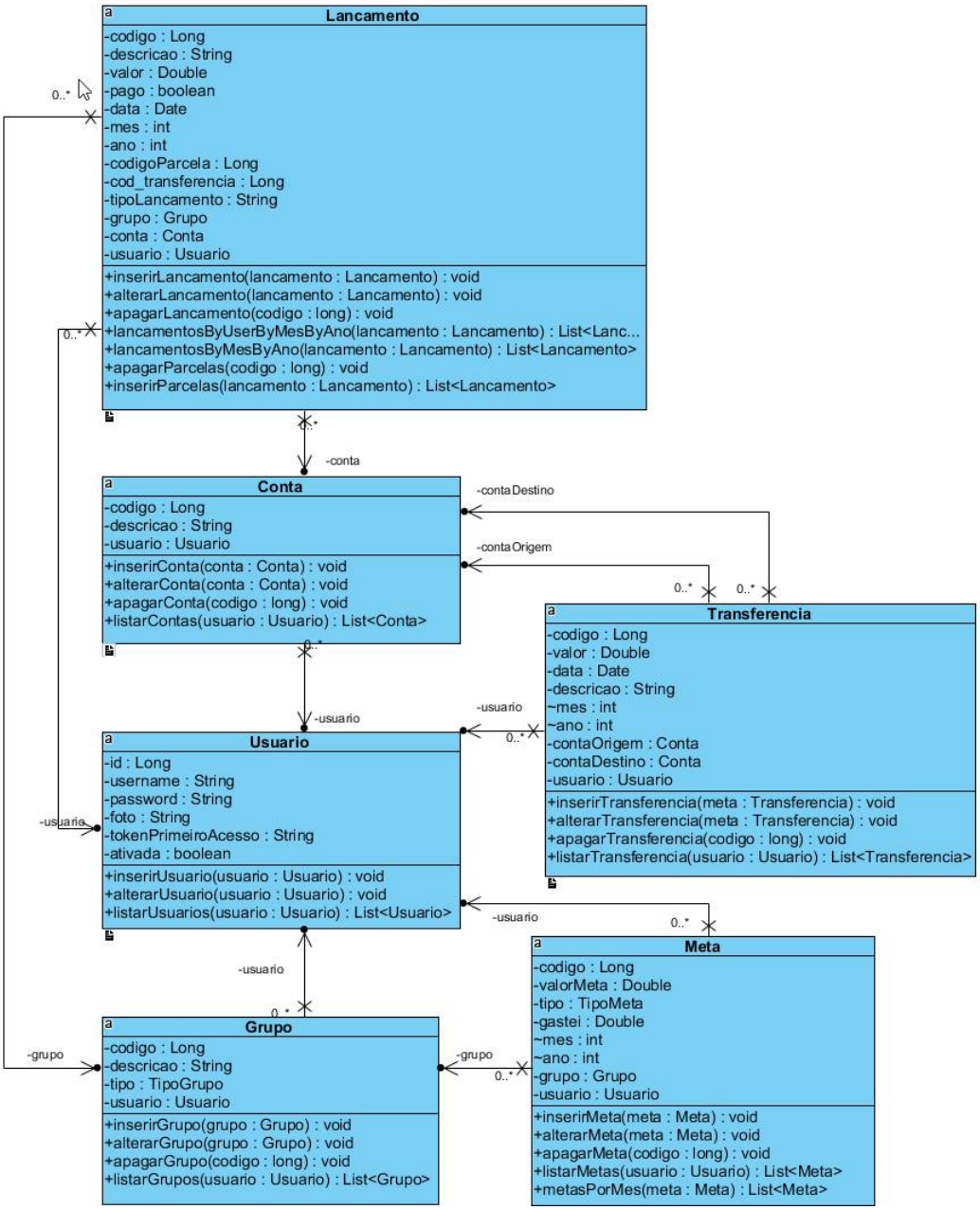


Figura 10 - Diagrama de Classes

Em relação ao banco de dados, na entidade *usuário*, serão armazenados os dados do usuário, login, senha, se a conta está ativa etc. Na entidade *contas* ficará a definição das contas, exemplo conta poupança, conta corrente etc. Já a entidade *grupos* armazena informações dos separadores, onde define-se se o mesmo é um “Ganho” ou “Gasto”. Na

entidade *lançamento* refere-se aos lançamentos que serão apontados pelo usuário (ganho ou gasto, conta, grupo, valor, data, etc). Na classe de *transferência* armazena-se as ações que o usuário realiza quando movimenta valores entre contas e pôr fim a entidade *metas* responsável por armazenar as metas definidas pelo usuário (tipo, valor, grupo, usuário). Na Figura 11 é exibido o diagrama de entidade e relacionamento.

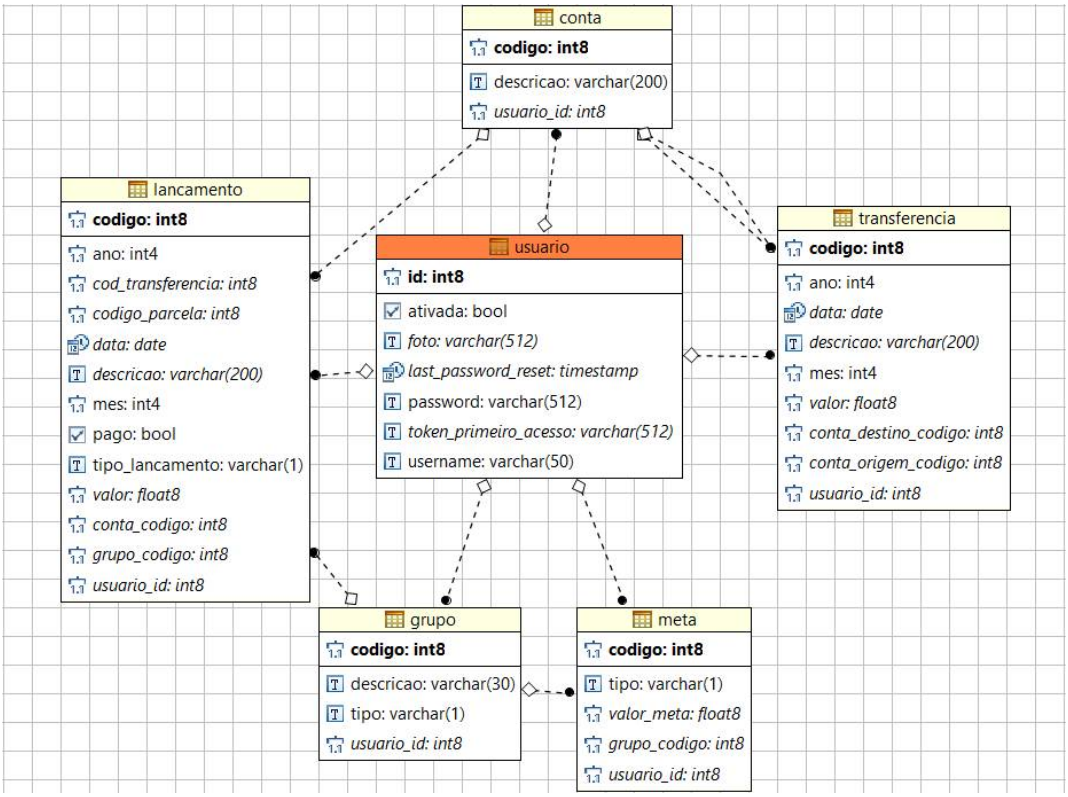


Figura 11 – Diagrama de Entidade Relacionamento

Durante o desenvolvimento do sistema foram realizados testes unitários, corrigindo e melhorando as funcionalidades de acordo com os requisitos propostos.

4.2 APRESENTAÇÃO DO SISTEMA

A aplicação descrita no decorrer deste trabalho é destinada a usuários pessoas físicas que desejam realizar controle de suas finanças. Esse controle envolve os lançamentos de entradas e saídas e gerenciamento das contas bancárias e grupos os quais são os principais responsáveis pela organização e demonstração dos resultados.

Os lançamentos englobam despesas recorrentes, ou seja, se o usuário realizou uma conta parcelada, o sistema apresenta as parcelas com o vencimento e respectivo detalhamento.

Esses dados são importantes para que o usuário possa extrair indicadores e manter-se atualizado de tudo o que está ou estará pendente.

Nesta aplicação *web*, além dos controles já citados foi criada a opção de transferência entre contas. Um exemplo desse caso é quando o usuário se desloca até o banco para sacar dinheiro da sua conta, este dinheiro pode ser utilizado para realizar pagamento de outras contas, ou seja, se o usuário não estiver atento e não tiver uma ferramenta como esta que possa lhe auxiliar nessas movimentações, a chance de ocorrer divergências financeiras é muito grande.

Além disso, os controles propostos permitirão que o usuário possa extrair indicadores, sejam eles mensais, anuais ou até mesmo diários. Existirão filtros capazes de proporcionar e sanar eventuais dúvidas com relação aos lançamentos registrados no sistema.

A Figura 12 apresenta a tela de *login* na aplicação, na qual o usuário informa seu endereço de e-mail e senha, onde o sistema processa e gera um *token* de autenticação permitindo ou não o *login* na aplicação.

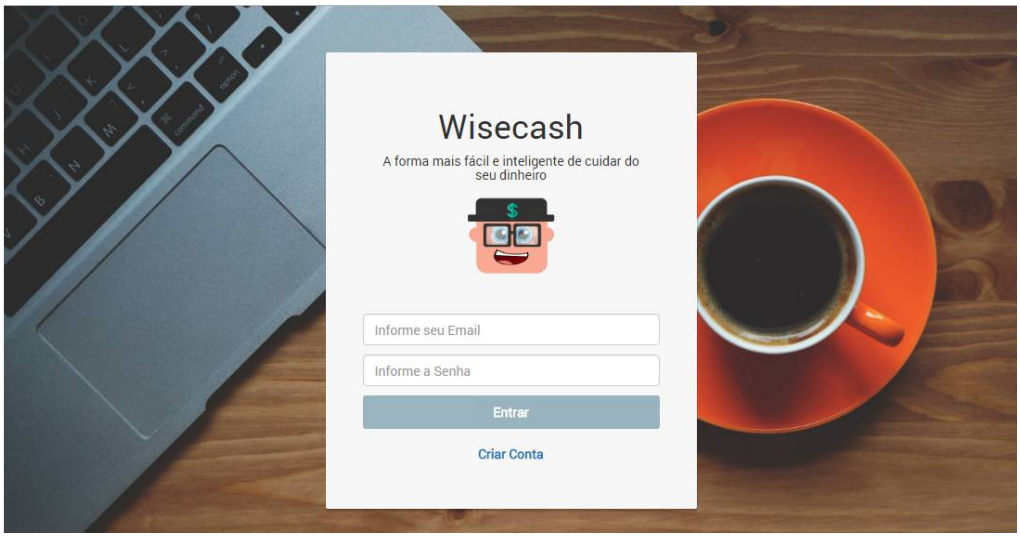


Figura 12 - Tela de Login

Caso o usuário não possua uma conta cadastrada na aplicação, basta o usuário clicar na opção “Criar Conta”, a qual será adicionada para a tela de cadastro de usuário, conforme mostra a Figura 13.

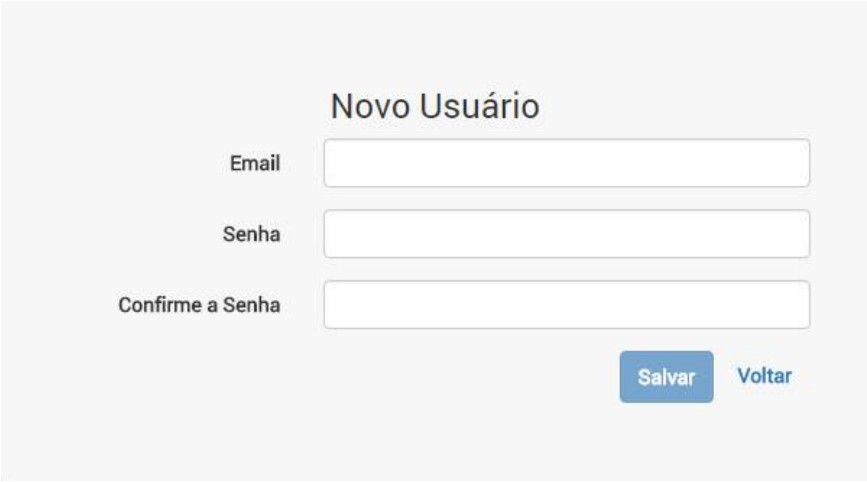


Figura 13 - Cadastro de Usuário

A aplicação conta com as validações de senha, funcionalidade afim de garantir que o usuário realmente saberá sua senha, validação de e-mails válidos e campos obrigatórios assim mantendo o botão desabilitado enquanto as informações não estejam corretamente informadas, essas funcionalidades podem ser observadas na Figura 14.

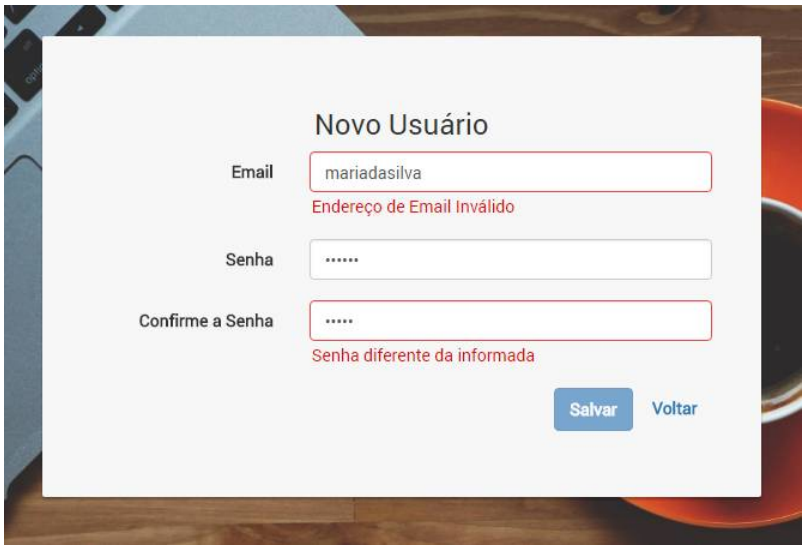


Figura 14 - Validação de Campos

Ao cadastrar um novo usuário, o sistema faz a validação verificando se o endereço de e-mail informado já existe, caso sim o sistema retorna uma mensagem, impossibilitando o usuário informar um endereço que já existe. A Figura 15 apresenta um cadastro realizado com sucesso.

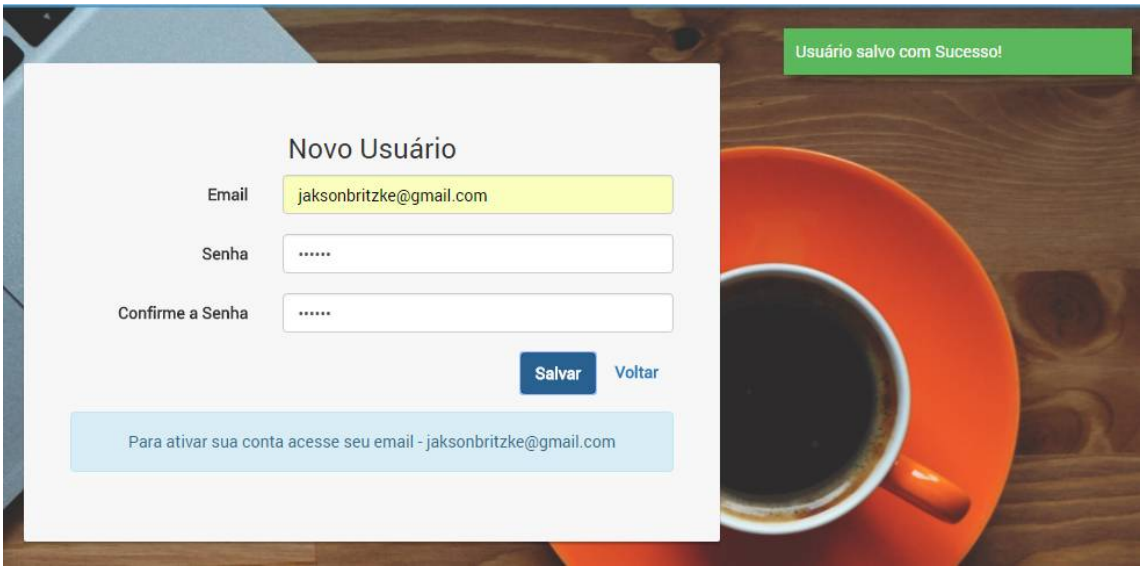


Figura 15 - Cadastrando Usuário

Como pode ser visto na Figura 15, o sistema encaminha um e-mail de confirmação para o usuário, no qual é criado um link contendo um *token* que ativa a conta do usuário quando clicado sobre o botão "Ativar Conta". Se o usuário não ativar a conta ao tentar logar o sistema apresenta uma mensagem "*Acesse seu e-mail para Ativar sua conta*". Na Figura 16 é mostrado o formato do e-mail recebido, bastado apenas o usuário clicar em "Ativar Conta".

Figura 16 - Confirmação da Conta

Caso o usuário tenha sucesso no login, será exibida a tela inicial da aplicação. Nessa tela (*Dashboard)* é mostrado um resumo geral da vida financeira do usuário, listando quanto o usuário "Tinha" que se refere ao resultado dos meses anteriores, os "Ganhos" e "Gastos"

referentes ao mês corrente e o Saldo considerando todos os lançamentos, e por fim, um extrato de cada conta existente no sistema, como pode ser visualizado na Figura 17.

Figura 17 - Tela Inicial - Dashboard

Referente ao extrato das contas, exibido na Figura 17, como o usuário pode ter “N” contas, no sistema foi programado para que mostre apenas as 3 primeiras, afim de fornecer melhor usabilidade ao usuário, porém ele poderá clicar no botão “Ver Todas” e visualizar as demais. Esta mesma funcionalidade similarmente se aplica ao botão “Tinha”, “Ganhos do Mês” e “Gastos do Mês”, onde ao clicar são mostrados os lançamentos.

Ainda na tela Inicial são apresentados dois gráficos, um demonstrando “Para onde vai meu dinheiro”, onde ao passar o mouse ou tocar sobre o grupo é mostrado quantos por cento o grupo representa de gastos durante o mês corrente e outro gráfico favorece a visão anual, ou seja, tudo o que foi gasto e ganho durante o ano corrente de forma visual. Na Figura 18 é mostrado a relação entre Saldo e Contas.

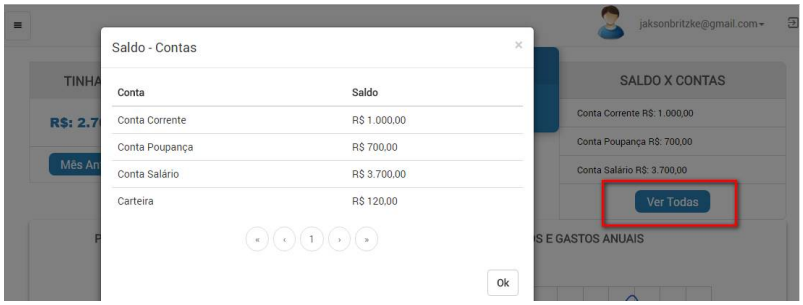


Figura 18 - Saldo x Contas

Na esquerda da Figura 19 é mostrado um menu com as funcionalidades que o sistema contempla, também um botão no topo permitindo o usuário esconder ou mostrar o menu da

esquerda facilitando a visualização das informações. E na parte superior mostra a foto do usuário, onde ao clicar sobre a foto é possível alterar a foto, onde a mesma é salva em um diretório da aplicação e renderizada na tela. Tais informações são observadas na Figura 19.

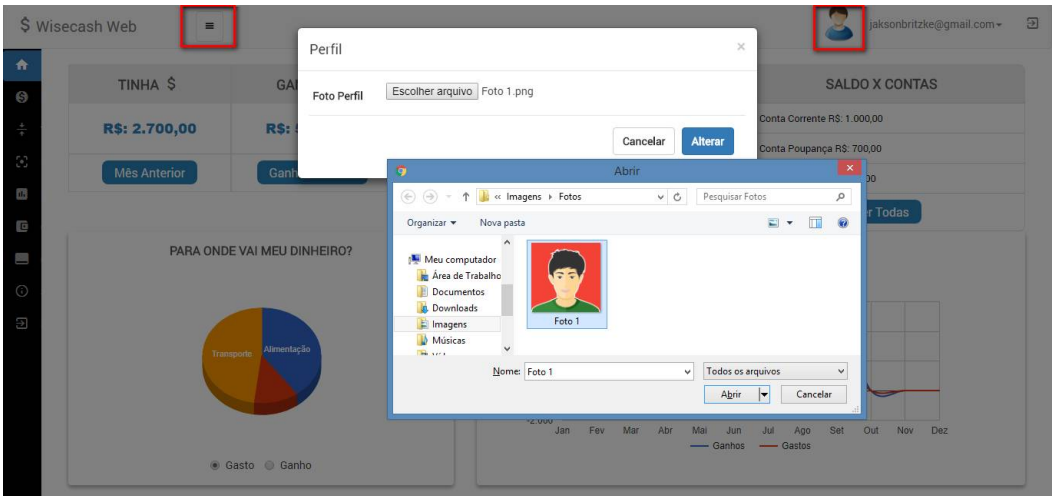


Figura 19 - Menu Esquerda x Foto Perfil

Para demonstração dos dados e extração destes indicadores inicialmente é necessário a inclusão de alguns cadastros básicos, sendo eles, cadastro de “Contas” e “Cadastro de Grupos”.

Quando o usuário clica no menu da esquerda em “Grupos” é mostrado a listagem dos grupos cadastrados, nesta tela o usuário realiza inclusão, edição, exclusão e também visualização das demais páginas, caso a lista possua mais de 7 registros. Visando facilitar a busca por um grupo específico foi implementado filtro no cabeçalho do grid, conforme Figura 20.

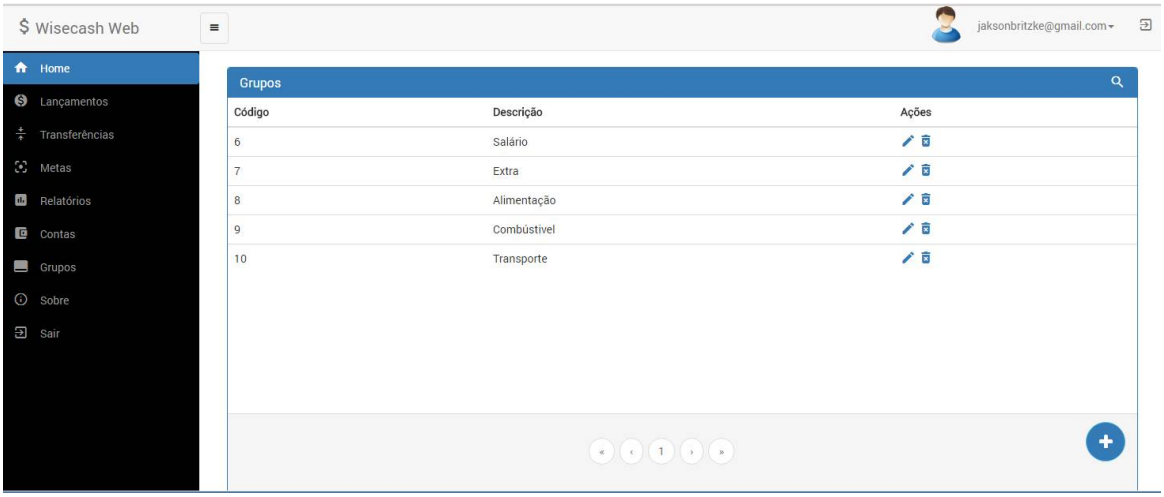


Figura 20 - Listagem de Grupos

Ao clicar sobre o botão de (+) o sistema abrirá o cadastro em uma tela modal, permitindo o usuário informar uma descrição para o “Grupo” e também o tipo de grupo (Ganhos ou Gastos), em todas as telas foram realizados os tratamentos de campos obrigatórios, obrigando o usuário informar uma descrição para habilitar o botão “Salvar”, como apresentado na Figura 21.

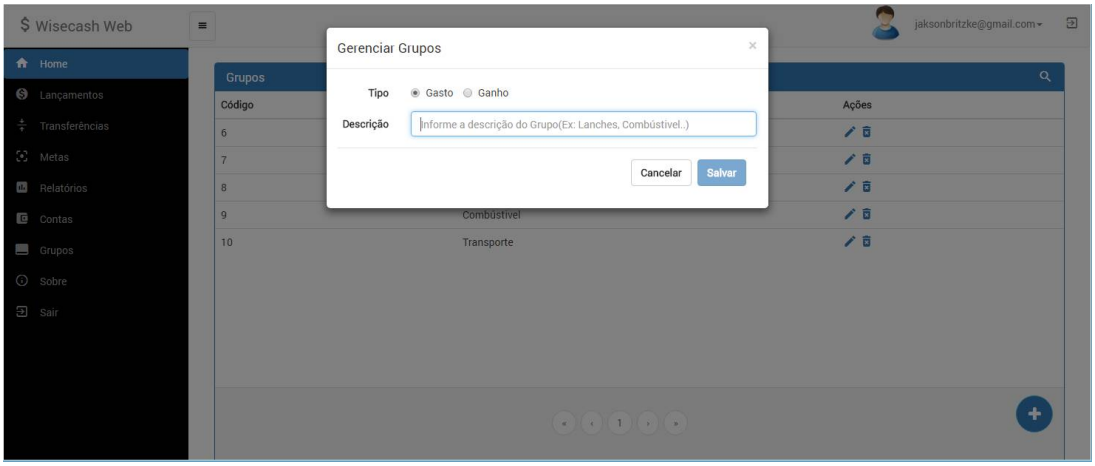


Figura 21 - Cadastro de Grupos

O Cadastro de “Contas” foi desenvolvido utilizando o mesmo conceito de usabilidade, layout e validações. Inicialmente é exibida a listagem de contas e também os atalhos para que o usuário realize as ações de inclusão, edição e exclusão.

Referente ao cadastro de lançamentos, ele basicamente funciona da mesma maneira, diferenciando apenas os filtros que o usuário pode realizar. Essa tela é exibida na Figura 22.

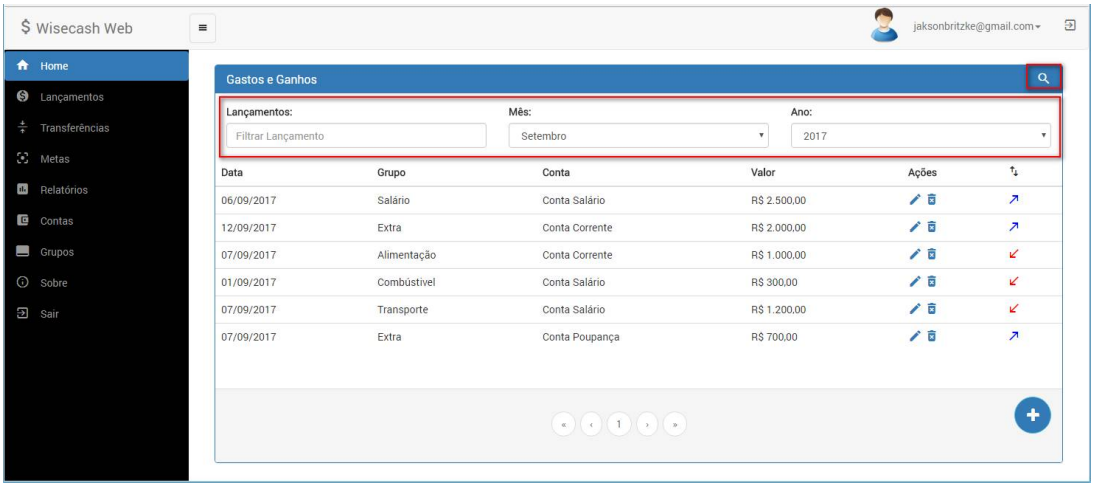


Figura 22 - Listagem de Lançamentos

Na tela de lançamento de um “Gasto” ou “Ganho”, são apresentados os campos de Valor, Data, Tipo do Lançamento, Conta, Descrição e uma opção afim de definir se o lançamento foi Pago ou Recebido. Automaticamente quando o usuário seleciona o tipo “Ganho” somente mostrará os grupos definidos como ganho e a *checkBox* que define se o lançamento é quitado é alterada para “Recebido”, o mesmo ocorre ao marcar o tipo “Gasto”, mostra apenas os grupos de gasto e altera-se a *checkBox* para “Pago”, facilitando a seleção e fornecendo melhor usabilidade ao usuário.

Nos campos de Grupo e Contas foi implementado um campo do tipo *auto-complete* visando facilitar a usabilidade, caso o usuário possua vários registros, onde conforme o usuário digitar parte da descrição, são filtrados os valores, facilitando a seleção. A tela de lançamento é exibida na Figura 23.

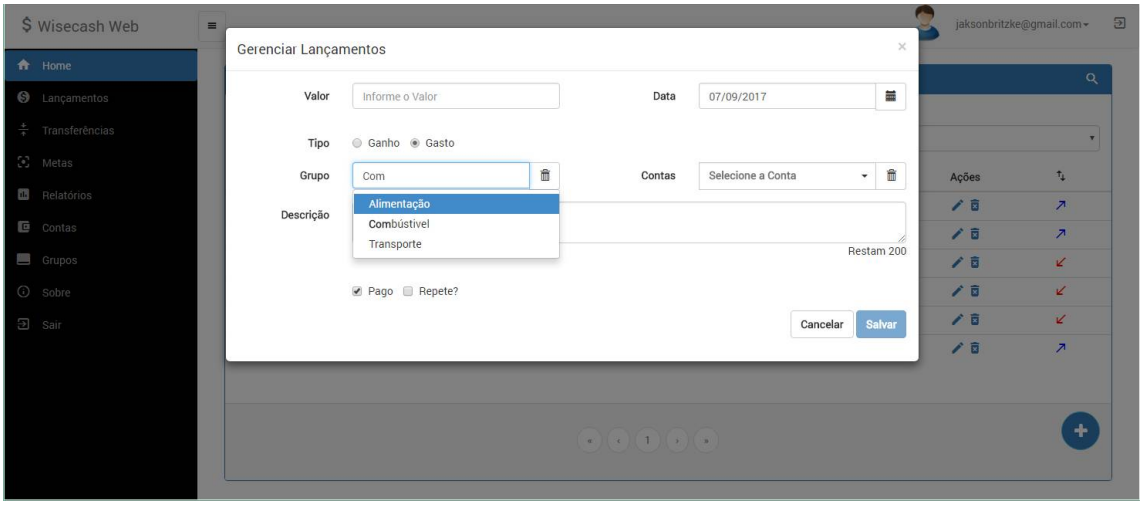


Figura 23 - Cadastro de Lançamentos

Em todos os cadastros foram implementadas as notificações de inclusão, edição e exclusão e também as validações de formulário conforme citado anteriormente. A Figura 24 apresenta um exemplo de notificação.

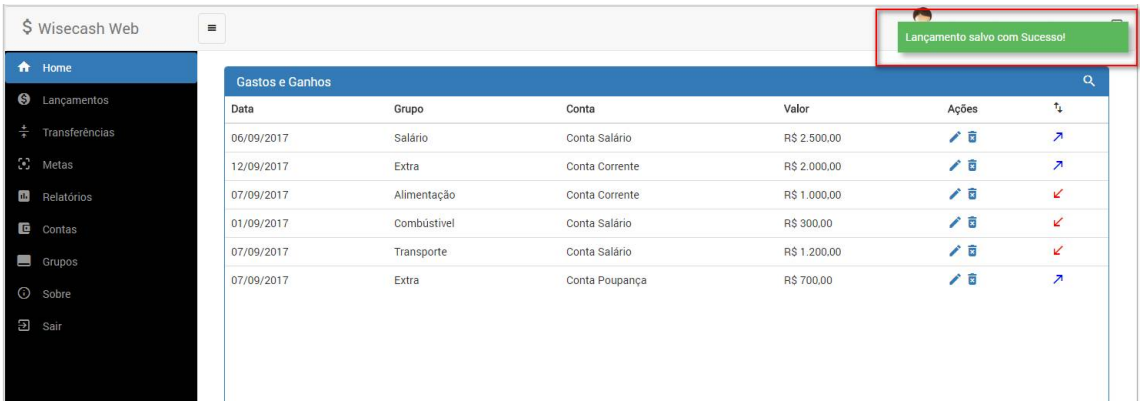


Figura 24 - Exemplo de Notificação

Para a exclusão foi implementada uma nova tela modal, permitindo que o usuário confirme a ação a ser realizada, como apresentado na Figura 25.

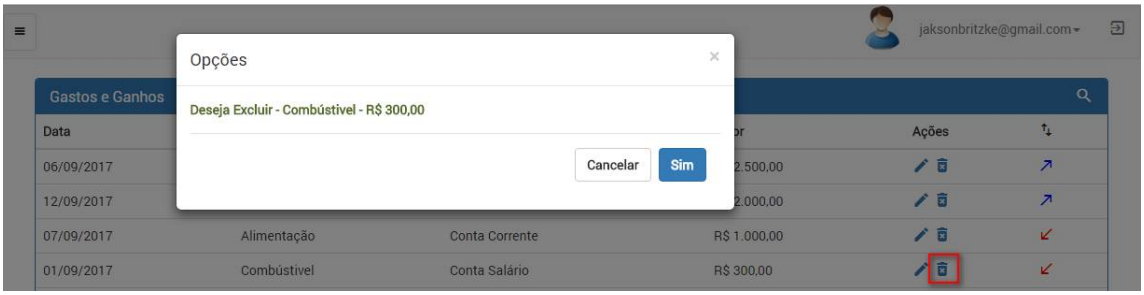


Figura 25 - Confirmação Exclusão

O usuário poderá incluir lançamentos recorrentes, para esta funcionalidade foi implementada uma opção na tela de lançamentos chamada “Repetir”. Na Figura 26 é mostrado o campo no qual o usuário clica para ser gerado os parcelamentos.

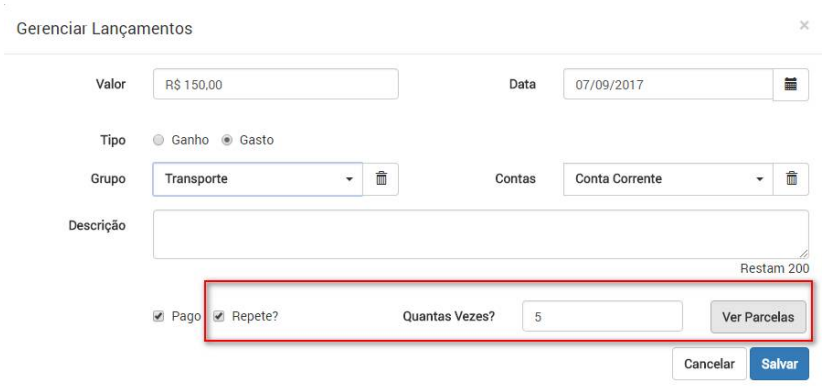


Figura 26 - Gerar Parcelas

Após o usuário clicar em “Ver Parcelas” é enviada uma requisição para o servidor contendo as informações do lançamento, no qual o servidor retorna um *json* com os parcelamentos gerados. Na Figura 27 é exibido como o usuário visualiza essa tela.

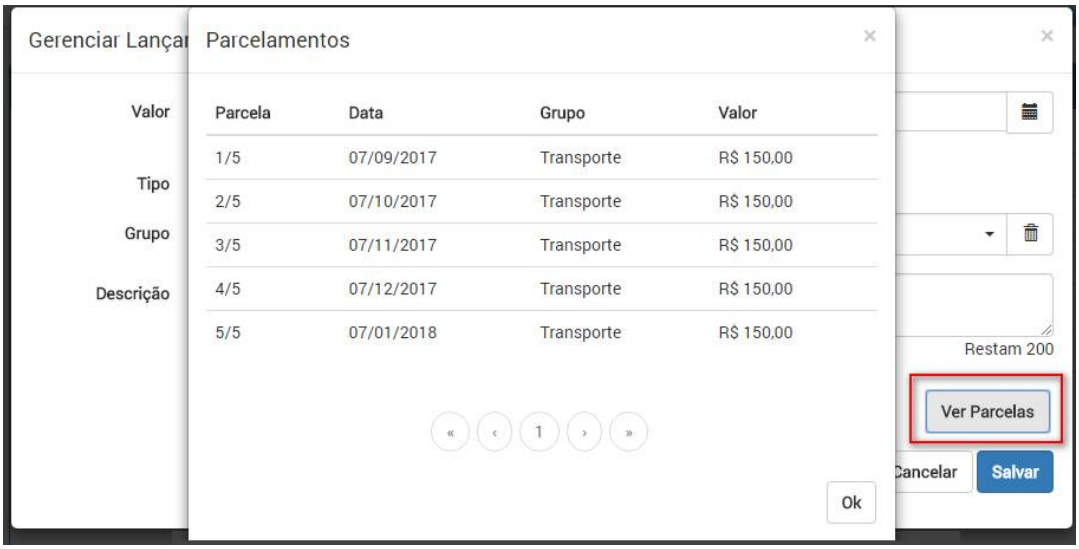


Figura 27 - Visualização parcelamentos

Quando o usuário salvar o registro dos lançamentos, será lançada a quantidade de parcelas informadas para os meses seguintes, considerando os demais parâmetros informados na tela. Caso o usuário exclua um desses registros que são parcelados, o sistema mostrará a confirmação de exclusão em outro formato, permitindo excluir uma ou todas as parcelas, conforme mostra a Figura 28.

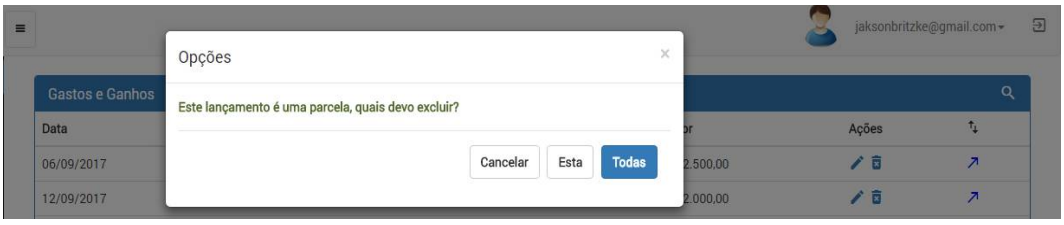


Figura 28 - Exclusão de Parcelamentos Recorrentes

Na tela de Transferências, exibida na Figura 29, o usuário poderá realizar transferência entre as contas, onde informa-se valor, data, descrição, conta de origem e conta de destino à qual será realizada transferência. Este processo é muito comum no dia a dia das pessoas, um exemplo clássico é quando nos dirigimos até o banco para realizar uma transferência, ou até mesmo para realizar um saque. A rotina conta com a validação de campos obrigatórios e também validação das contas selecionadas evitando que o usuário selecione a mesma conta para Origem quanto Destino.

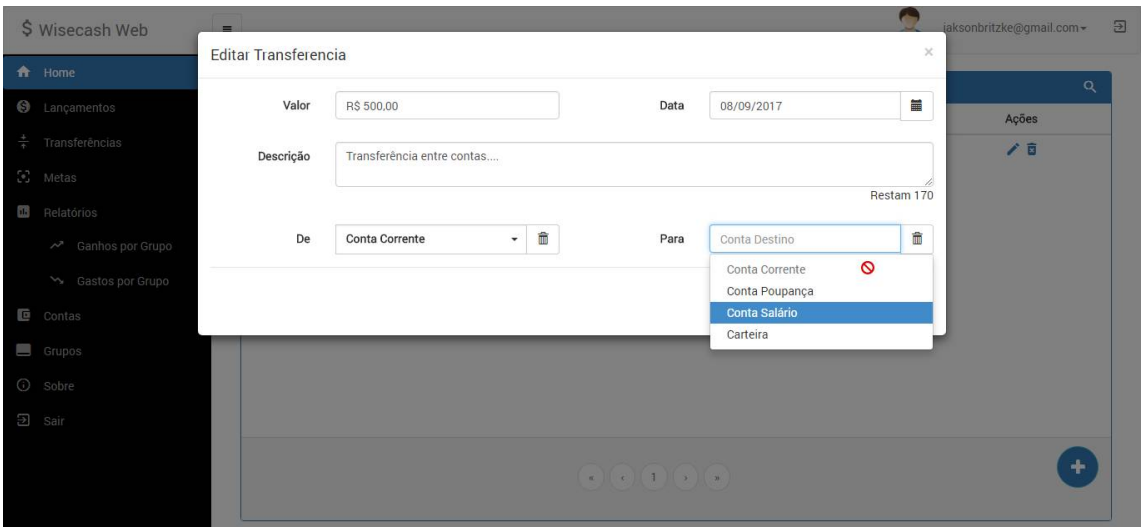


Figura 29 - Transferência entre Contas

Como foi descrito no parágrafo anterior, a rotina conta com a mesma estruturação de layout presente nas demais telas do sistema, permitindo que o usuário realize as principais alterações (incluir, editar, excluir) e também filtre pela transferência desejada.

Visando fornecer ao usuário alguma maneira na qual ele possa ter um controle quanto aos seus gastos, foi desenvolvido um recurso onde será possível lançar metas de gastos. Atualmente pode ser lançado tanto metas mensais quanto metas anuais. Por exemplo, digamos que o desejo é gastar R$: 600,00 em combustível durante o mês, na rotina é apresentada a meta a qual foi lançada, quanto já foi gasto e também quanto o usuário ainda poderá gastar naquele mês. A mesma regra equivale para as metas anuais, onde o usuário lança a meta desejada para o ano, e o sistema controla os gastos entre os meses do ano corrente.

Na Figura 30 é mostrada a tela na qual as metas podem ser visualizadas. Na listagem além das ações básicas de inclusão, edição e exclusão, a tela também conta com o filtro de metas no cabeçalho.

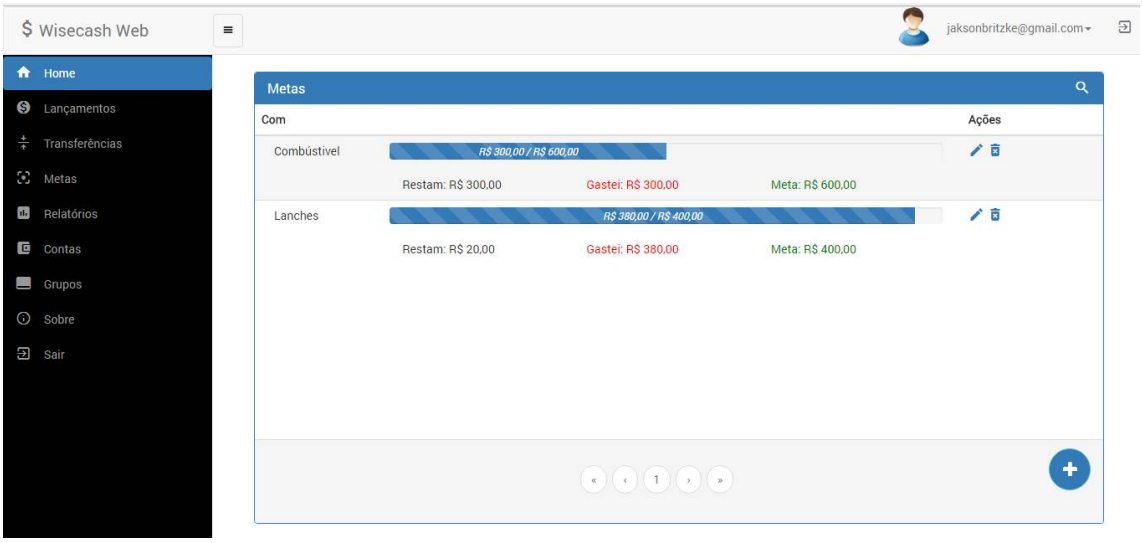


Figura 30 - Metas

Na Figura 31 é apresentado o cadastro das metas. A tela possui as validações básicas, fazendo com que o botão permaneça desabilitado até que sejam informados os campos necessários, alertando o usuário.

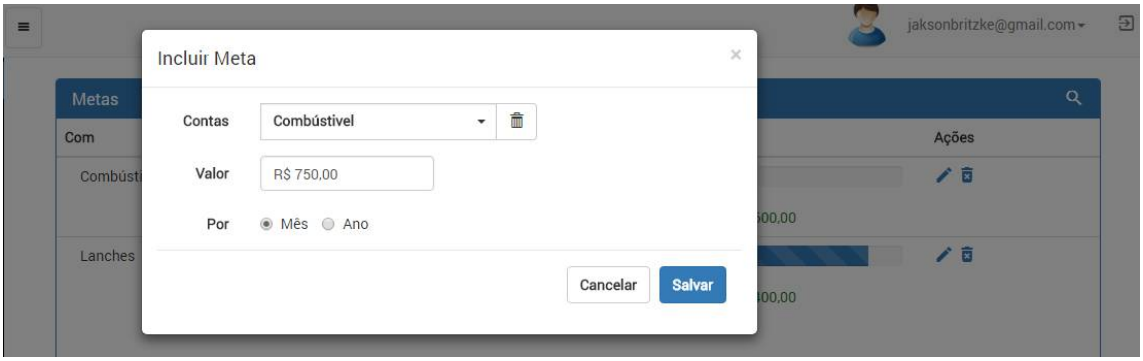


Figura 31 - Cadastro de Metas

Com base nos lançamentos e nos controles propostos foram implementado relatórios com as mesmas características diferenciando apenas os gastos de ganhos, onde o usuário poderá filtrar conforme sua necessidade, informando o período inicial e final, o(s) grupo(s), a(s) conta(s), e também possibilitará o usuário a filtrar pelos lançamentos Pago(s) ou Recebido(s), como pode ser observado na Figura 32.

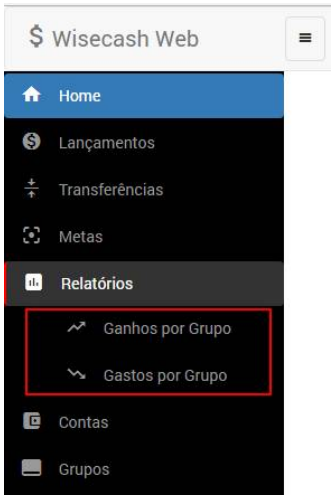


Figura 32 - Menu Relatórios

Na tela de cada relatório foram criados filtros *auto-complete* tanto para grupos quanto para contas facilitando a seleção do usuário e também oferecendo uma melhor experiência em usabilidade, conforme apresentado na Figura 33.

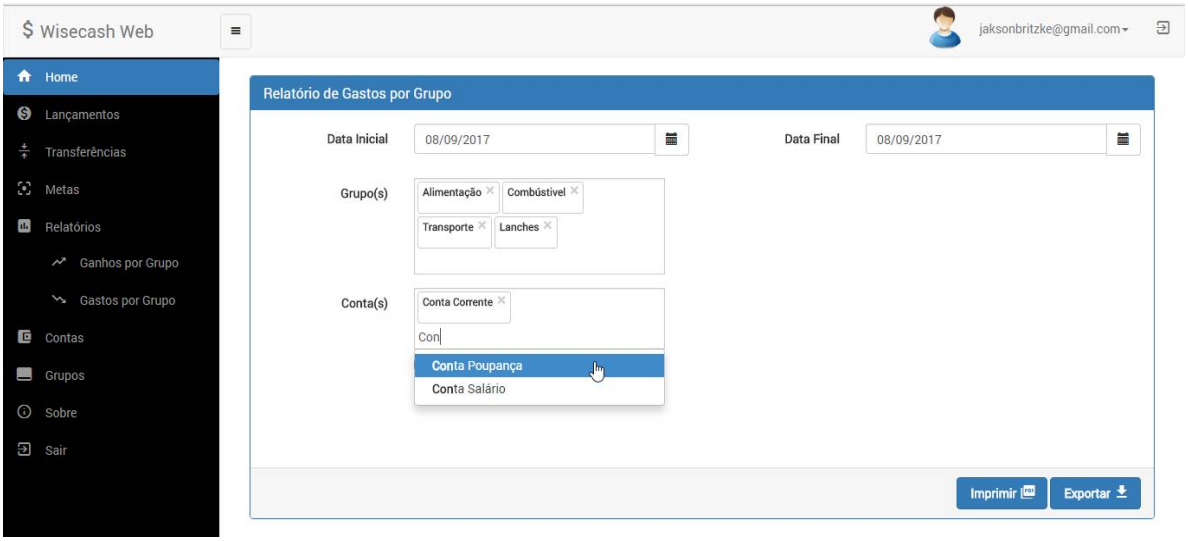


Figura 33 - Filtros Relatórios

Como pode ser visto na Figura 33 o usuário poderá visualizar o resultado dos relatórios no formato PDF, e também exportar o arquivo no formato ".xls". Nos relatórios

foram realizados tratamentos caso o usuário deixe algum campo em branco, nesta condição o sistema trará todos os dados.

Na Figura 34 é apresentado um exemplo do relatório que pode ser extraído tanto para ganhos como para gastos.

Figura 34 - Exemplo de Relatório

A aplicação foi desenvolvida em 3 idiomas (Português, Inglês e Espanhol), basta o usuário clicar e selecionar o idioma desejado. Esta função foi incluída no cabeçalho no campo de seleção que mostra o usuário logado, permitindo a alteração para qualquer um dos idiomas citados não sendo necessário sair da aplicação ou ter que recarregar a página. Na Figura 35 é melhor representada esta funcionalidade.

Figura 35 - Aplicação Multilíngue

4.3 DESENVOLVIMENTO DO SISTEMA

Nesta fase as tecnologias foram preparadas, instaladas e configuradas para que a implementação pudesse ocorrer. O *back-end* foi desenvolvido separado do *front-end*, na parte de *front-end* para início de projeto foi utilizado o *Grunt*, que nada mais é do que um automatizador de tarefas em JavaScript, onde o qual roda no Node.JS, também construído em *JavaSript* com intuito de criar aplicações rápidas e escaláveis.

A estrutura definida no projeto AngularJS(*front-end)* está organizado da seguinte maneira:

Config: Responsável por manter as configurações globais, como rotas, configuração das notificações, calendários e traduções;

Controllers: É o comportamento por trás do DOM, também responsável por inicializar e/ou adicionar comportamentos ao objeto, permitindo a comunicação entre *View* e *Controller*.

Directives: São elementos do AngularJS afim de encapsular uma lógica de apresentação de um módulo, após a criação de uma diretiva é possível passar para o compilador incluir um determinado comportamento específico.

Interceptors: Responsável por interceptar as requisições HTTP. No projeto é utilizado para validação do token.

Services: Os services mantêm as tarefas específicas comuns da aplicação, nele foram criadas as requisições (*get, post, put, update*);

Value: Criado para definição do endereço para comunicação com o servidor (*back-end)*, no projeto foi criado para facilitar se em algum momento for necessário a alteração do

endereço com o servidor, desta forma não haverá necessidade de percorrer todas as requisições e alterá-las para o novo endereço, basta apenas alterar no *value*.

***Views*:** É aonde ficam concentradas as páginas HTML;

A Figura 36 mostra a estruturação do projeto (*front-end).*

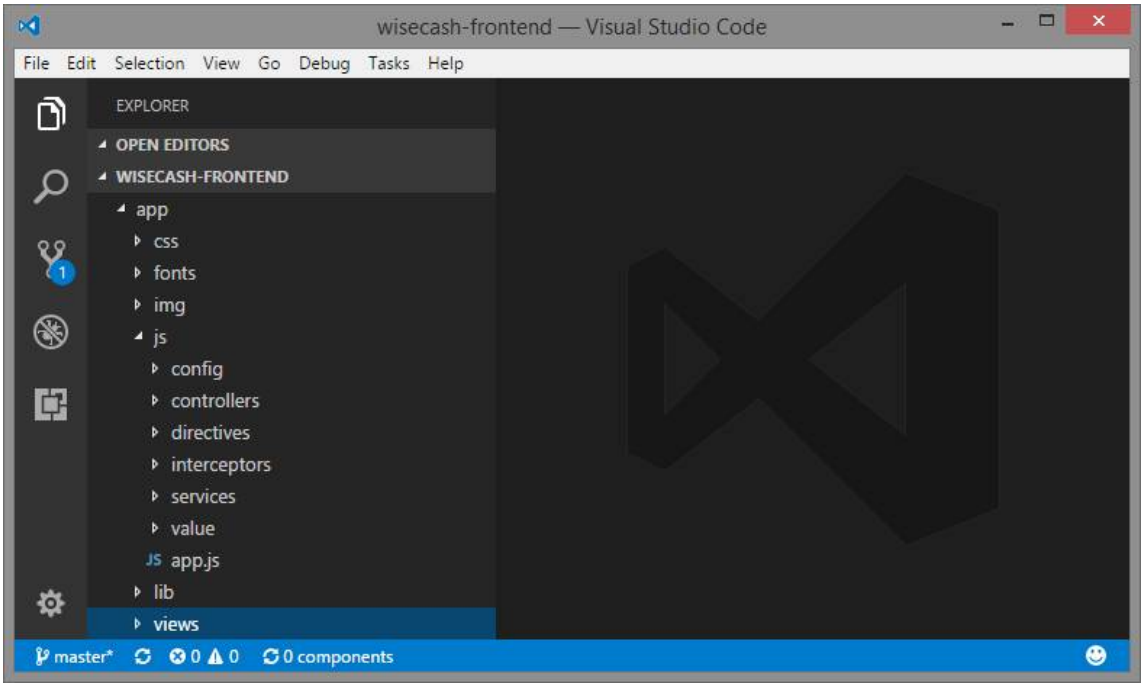


Figura 36 - Projeto AngularJs

No *back-end* foi seguido o uso do padrão MVC, organizando o projeto conforme mostra os pacotes (*package)* abaixo:

Component: Foi criado afim de chamar alguns métodos comuns da aplicação, por exemplo *UsuarioComponent*, responsável por buscar o usuário logado, *UtilComponent*, retorna mês atual, ano atual.

Config: Responsável por manter as configurações referente a autenticação, permissões de usuário etc;

Controller: Recebe todas as requisições do usuário, e retorna dados para a view;

Filter: Filtra as requisições, verifica o token recebido afim de autenticar o usuário;

Model: Onde são definidas as classes e seus devidos atributos. Dentro desse pacote foi criado um novo pacote chamado ***to (Object Transaction)*** e ***enumeration***, onde o ***to*** armazena classes que servirão apenas para montar um resultado, não sendo necessário sua persistência no banco, utilizada nos gráficos por exemplo e o ***enumeration*** usado para definição do tipo do grupo (receita ou despesa), e o tipo da meta (mensal ou anual);

Repository: Implementa parte das regras de negócio no que se refere a composição das entidades;

Security: Responsável pela segurança da aplicação (geração do token, tempo de expiração, chave para descodificar o token etc);

Service: Armazena as regras comuns da aplicação.

A Figura 37 exibe a organização de acordo com o padrão MVC abordado:

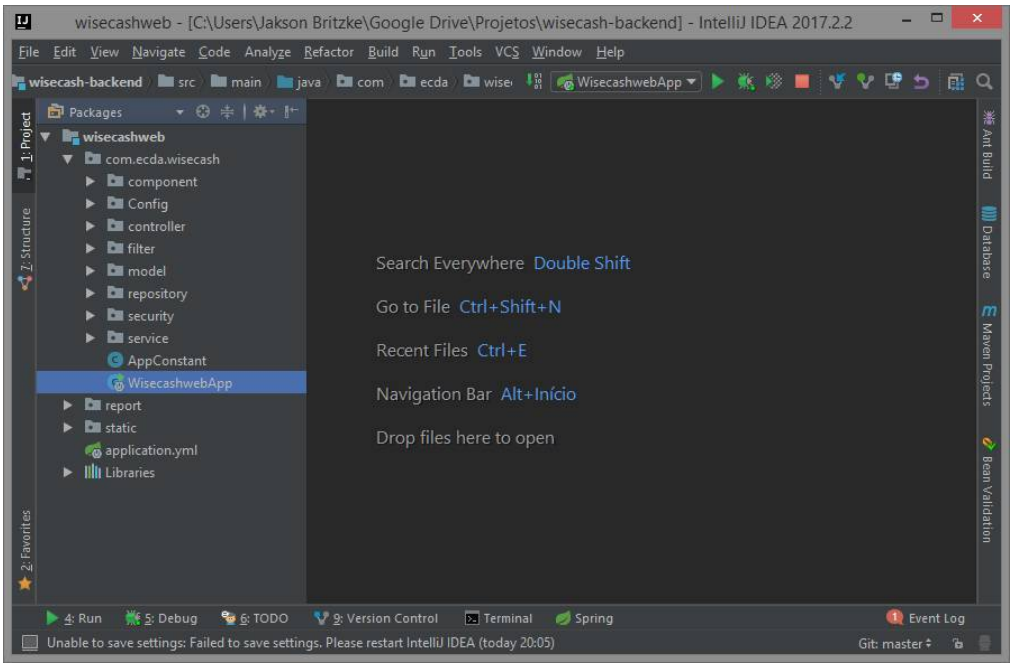


Figura 37 - Organização BackEnd

Para ambas as partes da aplicação (*front-end* e *back-end*) foi utilizado um controlador de versão chamado *Bitbucket*, plataforma que tem o intuito de gerenciar diferentes versões, possibilitando a obtenção de históricos de todas as modificações, trazendo agilidade e segurança no decorrer do desenvolvimento desta aplicação.

Conforme citado na descrição das ferramentas para este projeto foi utilizado o Spring boot, o qual foi criado o projeto utilizado Maven para gerenciamento dos builds e integração contínua. A partir disso fazendo o uso da dependência do Spring, foram obtidos a maior parte dos recursos necessários, como o Hibernate por exemplo para o mapeamento das tabelas do banco de dados e um servidor de aplicação que é embutido no empacotamento do Jar, neste caso o Tomcat.

No decorrer do texto serão demonstrados alguns códigos implementados para o desenvolvimento do trabalho, o intuito é exemplificar a implementação realizada.

O processo de autenticação, ou seja, a validação de um usuário e senha no sistema é controlado a partir da geração de um *token*, no qual o usuário envia usuário e senha para o servidor, servidor processa, busca pelo usuário no banco de dados e retorna um *token* para o navegador, permitindo ou não a autenticação. A Listagem 1 melhor representa esta descrição.

```
$scope.autenticar = function () {
    autenticarAPI.autenticarUsuario($scope.usuario).then(function (response) {
        $scope.token = response.data.token;
        sessionStorage.setItem("currentUser", JSON.stringify(response.data.token));
        sessionStorage.setItem("userLogado", JSON.stringify($scope.usuario.username));
        $location.path("/");
    }, function (response) {
        if (response.status === 500) {
            $scope.erroLogin = ($filter('translate')(response.data.message + " - ") + $scope.usuario.username);
        } else {
            $scope.erroLogin = ($filter('translate')("Verifique seu usuário ou senha") + " :(");
        }
    });
}
```

Listagem 1 - Autenticação do Usuário (AngularJs)

Para fins de organização todas as chamadas para o servidor foram separadas e criado serviços por funcionalidades, um dos exemplos é a Listagem 2 apresentada abaixo.

```
app.factory("autenticarAPI", function($http, config){
    var _autenticarUsuario = function (usuario){
        return $http.post(config.baseUrl + "/auth", usuario);
    };
    return {
        autenticarUsuario: _autenticarUsuario
    };
});
```

Listagem 2 - Requisições Servidor (AngularJs)

Como pode ser visto na Listagem 1 e 2, quando o sistema executar a função $scope.autenticar, o sistema realizará uma requisição para o servidor (“autenticarAPI”), que retornará um *token*. Se o servidor *back-end* encontrar o usuário e a requisição tenha sido realizada com sucesso a página será direcionada para tela inicial do sistema, caso contrário retornará uma mensagem informando para verificar usuário ou senha, ou *status http* 500, nesse caso o sistema exibe uma mensagem informando que o usuário deve ativar a conta por e-mail.

A Listagem 3 representa a classe responsável pela autenticação do usuário, onde se o usuário estiver ativo o servidor gera o *token* e retorna o mesmo para o navegador onde esse *token* é gravado na sessão, permitindo que o angular busque toda vez que necessário para realizar as chamadas *Rest.*

```java
@RestController
@RequestMapping("auth")
public class AuthenticationController extends ControllerBase {

    @Autowired
    private AuthenticationManager authenticationManager;

    @Autowired
    private TokenUtils tokenUtils;

    @Autowired
    private UsuarioService usuarioService;

    @RequestMapping(method = RequestMethod.POST)
    public ResponseEntity<?> authenticationRequest(@RequestBody AuthenticationRequest authenticationRequest) throws Exception {
        Authentication authentication = this.authenticationManager.authenticate(new UsernamePasswordAuthenticationToken(
                authenticationRequest.getUsername(), authenticationRequest.getPassword()));

        SecurityContextHolder.getContext().setAuthentication(authentication);
        UserDetails userDetails = this.usuarioService.loadUserByUsername(authenticationRequest.getUsername());

        Usuario usuario = usuarioComponent.getUsuario();
        if (usuario.isAtivada() == true) {
            String token = this.tokenUtils.generateToken(userDetails);
            return ResponseEntity.ok(new AuthenticationResponse(token));
        } else {
            throw new Exception("Acesse seu email para ativar sua conta");
        }
    }
```

Listagem 3 - Autenticação do Usuário (Java)

A partir da autenticação, as requisições ($http.get) e respostas contendo algum tipo de erro (responseError) são passadas por um interceptor, pois como o próprio nome já diz sua principal função é interceptar todas as requisições http ($http.get, $http.post, etc...). Além dessa funcionalidade ele adiciona um header nas requisições, verifica se o token existe e serve também para tratar respostas genéricas oriundas do backend (ex.: erro 404, erro500). A Listagem 4, apresenta a implementação do Interceptor.

```javascript
app.factory("tokenInterceptor", function ($q, $injector, $location, config) {
    return {
        'request': function (config) {
            var token = JSON.parse(sessionStorage.getItem('currentUser'));
            if (token != null){
                config.headers['X-Auth-Token'] = token;
                config.headers['Accept'] = 'application/json;odata=verbose';
            }
            return config;
        },
        'responseError': function (rejection) {
            if (rejection.status == 401) {
                $location.path("/login");
            }
            return $q.reject(rejection);
        }
    };
});
```

Listagem 4 - Interceptor(AngularJs)

A cada requisição enviada para o servidor é enviado o *token* no cabeçalho, uma constante comum entre as aplicações (*front-end e back-end*) neste projeto chamada *X-Auth-Token*, onde no *back-end* é lido essa informação por um filtro que valida o *token*, como pode ser visto na Listagem 5.

```
public class AuthenticationTokenFilter extends UsernamePasswordAuthenticationFilter {
    @Autowired
    private TokenUtils tokenUtils;
    @Autowired
    private UsuarioService userDetailsService;
    @Override
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
            throws IOException, ServletException {

        tokenUtils = WebApplicationContextUtils
                .getRequiredWebApplicationContext(this.getServletContext())
                .getBean(TokenUtils.class);
        userDetailsService = WebApplicationContextUtils
                .getRequiredWebApplicationContext(this.getServletContext())
                .getBean(UsuarioService.class);

        HttpServletResponse resp = (HttpServletResponse) response;
        resp.setHeader( s: "Access-Control-Allow-Origin", s1: "*");
        resp.setHeader( s: "Access-Control-Allow-Methods", s1: "POST, GET, PUT, OPTIONS, DELETE, PATCH");
        resp.setHeader( s: "Access-Control-Max-Age", s1: "3600");
        resp.setHeader( s: "Access-Control-Allow-Headers", s1: "Origin, X-Requested-With, Content-Type, Accept, "
                        + AppConstant.tokenHeader);
        HttpServletRequest httpRequest = (HttpServletRequest) request;
        String authToken = httpRequest.getHeader(AppConstant.tokenHeader);
        String username = this.tokenUtils.getUsernameFromToken(authToken);

        if (username != null && SecurityContextHolder.getContext().getAuthentication() == null) {
            UserDetails userDetails = this.userDetailsService.loadUserByUsername(username);
            if (this.tokenUtils.validateToken(authToken, userDetails)) {
                UsernamePasswordAuthenticationToken authentication =
                    new UsernamePasswordAuthenticationToken(userDetails, credentials: null, userDetails.getAuthorities());
                authentication.setDetails(new WebAuthenticationDetailsSource().buildDetails(httpRequest));
                SecurityContextHolder.getContext().setAuthentication(authentication);
            }
        }
        chain.doFilter(request, response);
    }
}
```

Listagem 5 - Filtro de Requisições (Java)

A partir da implementação do *interceptor* no *front-end* ele foi adicionado também para o arquivo de rotas, o arquivo de rotas nada mais é do que o apontamento que define para onde a página deve ser direcionada seguido do controlador responsável pela lógica programada, as rotas são alteradas quando o usuário clicar em algum menu por exemplo.

Conforme mostra a Listagem 6 o interceptor foi adicionado no cabeçalho do arquivo de rotas, fazendo com que a cada mudança de rota o interceptor entre em ação interceptando todas as requisições, verificando se o token existe etc. Note também que nessa listagem ainda possui um "*resolve*", o objetivo desse resolve é carregar as informações antes que a *view* seja carregada.

```
app.config(function ($routeProvider, $httpProvider, $translateProvider) {
    $httpProvider.interceptors.push("tokenInterceptor");
    $routeProvider
        .when('/login', {
            templateUrl: 'views/login/login.html'
        })
        .when('/', {
            templateUrl: 'views/dashboard/dashboard.html',
            controller: 'dashboardCtrl',
            controllerAs: 'ctrl',
            resolve: {
                gastosporgrupo: function (dashboardAPI) {
                    return dashboardAPI.getGastosPorGrupo();
                },
                ganhosporgrupo: function (dashboardAPI) {
                    return dashboardAPI.getGanhosPorGrupo();
                },
                saldoDespesaPorconta: function (dashboardAPI) {
                    return dashboardAPI.getSaldoDespesaPorConta();
                },
                gastosMensaisPorAno: function (dashboardAPI) {
                    return dashboardAPI.getGastosMensaisPorAno();
                }
            }
        })
```

Listagem 6 - Rotas x Interceptor(AngularJs)

Abaixo das rotas foi implementada a função referente ao ponto de partida da aplicação(*.run)*, onde foi implementado o *$routeChangeStart* que é responsável por entender todas as alterações de rota. Em cada alteração o sistema verifica se existe um *token* armazenado no *sessionStorage*,[1] caso sim o sistema permite direcionamento para a rota desejada, caso contrário direciona a página para o *login.* Listagem 7 representa esta implementação

```
app.run(['$rootScope', '$location', function ($rootScope, $location, $httpProvider) {
    $rootScope.$on('$routeChangeStart', function (event) {
        $rootScope.rota = $location.path();
        var token = JSON.parse(sessionStorage.getItem('currentUser'));
        $rootScope.usuarioLogado = JSON.parse(sessionStorage.getItem('userLogado'));
        if (token == null) {
            $location.path('/login');
        }
    });
}]);
```

Listagem 7 - Valida Token Existente

[1] Salva dados no navegador ficando vinculados ao (e apenas acessíveis) pelo seu domínio (BELEM, Thiago, 2012).

Para criação dos gráficos foram utilizados os mesmos conceitos, ou seja, uma requisição é enviada para o servidor, onde o mesmo processa e retorna os dados formatados. Para o gráfico “Para onde vai meu dinheiro”, foi criada uma consulta que retorna: código, descrição do grupo, mês, ano e total. A Listagem 8 demonstra essa requisição.

```
var _getGastosPorGrupo = function () {
    return $http.get(config.baseUrl + "/dashboard/gastosporgrupo");
};
```

Listagem 8 - Requisição Gráfico (AngularJs)

Na Listagem 9 demonstra-se o desenvolvimento Java que recebe esta requisição, e executa a consulta *gastosByGrupo.*

```
@RestController
@RequestMapping("dashboard")
public class DashboardController extends ControllerBase implements Serializable {

    @Autowired
    private DashboardRepository dashboardRepository;

    @Autowired
    private ContaRepository contaRepository;

    @GetMapping("/gastosporgrupo")
    public List<GastosPorGrupoTo> gastosPorGrupo() {
        return dashboardRepository.gastosByGrupo(usuarioComponent.getUsuario(),
                utilComponent.mesAtual(), utilComponent.anoAtual());
    }
```

Listagem 9 - Consulta – Gráfico (Java)

Listagem 10, refere-se à consulta *gastosByGrupo,* que será executada quando for invocado o método gastosPorGrupo().

```
@Query("select new com.ecda.wisecash.model.to.GastosPorGrupoTo(grupo.codigo,grupo.descricao, l.mes, l.ano, sum(valor)) " +
        "from Lancamento l " +
        "where grupo.tipo = 'D' and l.usuario = :usuario and l.mes = :mes and l.ano = :ano " +
        "group by grupo.codigo,grupo.descricao, l.mes, l.ano")
List<GastosPorGrupoTo> gastosByGrupo(@Param("usuario") Usuario usuario, @Param("mes") int mes, @Param("ano") int ano);
```

Listagem 10 - Consulta JPA - Gráfico Para onde vai meu dinheiro (Java)

Para o gráfico “Ganhos e Gastos Anuais” foram criadas duas consultas especificas que retornam uma lista contendo Mês, Ano e Valor referente a “Gastos” e outra para “Ganhos”. Como pode-se notar na Listagem 10 e 11 foi utilizado um *new* após o *select,* especificando uma classe, isso quer dizer que para cada linha retornada, será criada uma instância desta classe.

```java
@Query("select new com.ecda.wisecash.model.to.GastosMensalAnoTo(l.mes, l.ano, sum(valor)) " +
        "from Lancamento l " +
        "where grupo.tipo = 'D' and l.usuario = :usuario and l.ano = :ano " +
        "group by l.mes, l.ano")
List<GastosMensalAnoTo> gastosMensaisByAno(@Param("usuario") Usuario usuario, @Param("ano") int ano);
```

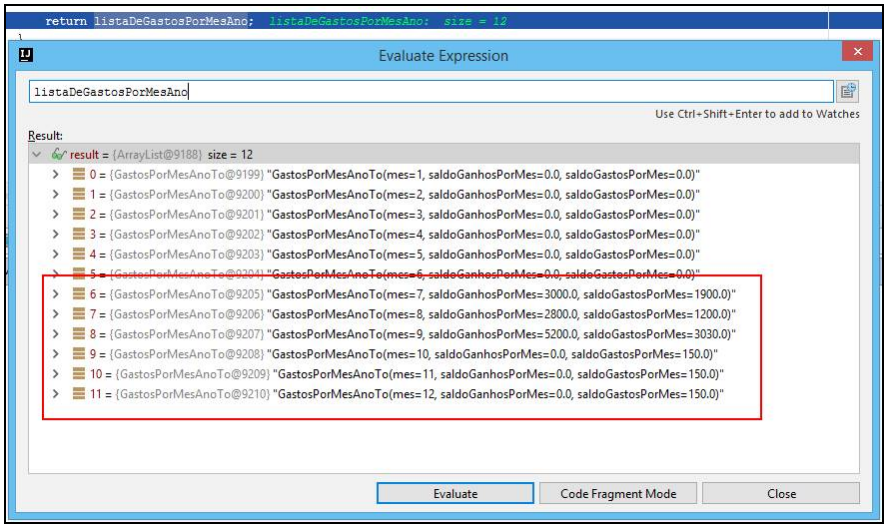
Listagem 11 - Consulta JPA - Gráfico Ganhos e Gastos Anuais (Java)

Posteriormente foi criado uma estrutura que atende o retorno deste gráfico, um método que percorre 12 vezes afim de encontrar os “Gastos” e “Ganhos” para cada mês do ano especificado, e dentro dele existe outra função que percorre a lista citada acima (Listagem 11), onde já traz o mês ano e valor e caso não existe lançamento para um determinado mês é alimentando 0. A classe responsável por esta ação pode ser observada na Listagem 12.

```java
    @GetMapping("/gastosMensaisPorAno")
    public List<GastosPorMesAnoTo> gastosMensaisPorAno() {

        List<GastosMensalAnoTo> listaGastos = dashboardRepository.gastosMensaisByAno(usuarioComponent.getUsuario(),
                utilComponent.anoAtual());
        List<GastosMensalAnoTo> listaGanhos = dashboardRepository.ganhosMensaisByAno(usuarioComponent.getUsuario(),
                utilComponent.anoAtual());

        List<GastosPorMesAnoTo> listaDeGastosPorMesAno = new ArrayList<>();
        for (int i = 1; i < 13; i++) {
            int mes = 0;
            Double saldoGanhosPorMes = 0.0;
            Double saldoGastosPorMes = 0.0;
            GastosPorMesAnoTo gastosPorMesAnoTo = new GastosPorMesAnoTo(mes, saldoGanhosPorMes, saldoGastosPorMes);
            gastosPorMesAnoTo.setMes(i);

            for (GastosMensalAnoTo listaD : listaGastos) {
                if (gastosPorMesAnoTo.getSaldoGastosPorMes() == 0.0) {
                    if (listaD.getMes() == i) {
                        gastosPorMesAnoTo.setSaldoGastosPorMes(listaD.getValor());
                        break;
                    } else {
                        gastosPorMesAnoTo.setSaldoGastosPorMes(0.0);
                    }
                }
            }

            for (GastosMensalAnoTo listaR : listaGanhos) {
                if (gastosPorMesAnoTo.getSaldoGanhosPorMes() == 0.0) {
                    if (listaR.getMes() == i) {
                        gastosPorMesAnoTo.setSaldoGanhosPorMes(listaR.getValor());
                        break;
                    } else {
                        gastosPorMesAnoTo.setSaldoGanhosPorMes(0.0);
                    }
                }
            }

            listaDeGastosPorMesAno.add(gastosPorMesAnoTo);
        }

        return listaDeGastosPorMesAno;
    }

}
```

Listagem 12 - Gráfico Ganhos e Gastos Anuais (Java)

A Listagem 13 apresenta um resultado do método apresentado na Listagem 12.



Listagem 13 - A Listagem 13 apresenta um exemplo de resultado do método citado.

Com o resultado disponível no AngularJS é necessário passar para o gráfico a lista de dados a qual será convertida no gráfico. Segue Listagem 14 demonstrando este código.

```
googleChartApiPromise.then(function () {
    var table = new google.visualization.DataTable();
    table.addColumn('number', 'Mes');
    table.addColumn('number', ($filter('translate')("Ganhos")));
    table.addColumn('number', ($filter('translate')("Gastos")));

    angular.forEach(gastosMensaisPorAno, function (row) {
        table.addRow([
            row.mes,
            row.saldoGanhosPorMes,
            row.saldoGastosPorMes,
        ]);
    });
    $scope.lineChart = {
        type: "LineChart",
        options: {
            hAxis: {
                ticks: [{ v: 1, f: ($filter('translate')("Jan")) },
                        { v: 2, f: ($filter('translate')("Fev")) },
                        { v: 3, f: ($filter('translate')("Mar")) },
                        { v: 4, f: ($filter('translate')("Abr")) },
                        { v: 5, f: ($filter('translate')("Mai")) },
                        { v: 6, f: ($filter('translate')("Jun")) },
                        { v: 7, f: ($filter('translate')("Jul")) },
                        { v: 8, f: ($filter('translate')("Ago")) },
                        { v: 9, f: ($filter('translate')("Set")) },
                        { v: 10, f:($filter('translate')("Out")) },
                        { v: 11, f: ($filter('translate')("Nov")) },
                        { v: 12, f: ($filter('translate')("Dez")) }]
            },
            curveType: 'function',
            legend: { position: 'bottom' }
        },
        data: table,

    };
    var formatter = new google.visualization.NumberFormat(
        { prefix: '$', negativeColor: 'red', negativeParens: true });
    formatter.format(table, 1);

});
```

Listagem 14 - Gráfico Ganhos e Gastos Anuais (AngularJs)

Na página em que o gráfico será exibido foi necessário acrescentar a *tag* responsável pelo gráfico conforme mostra a Listagem 15.

```
<div class="col-md-7">
    <section class="panel gmd-2">
        <div class="panel-body columnchart">
            <div class="top-stats-panel">
                <div class="gauge-canvas">
                    <h4 class="widget-h" translate="Ganhos e Gastos Anuais"></h4>
                </div>
                <div class="chart" google-chart chart="lineChart"></div>
            </div>
        </div>
    </section>
</div>
```

Listagem 15 - Gráfico Ganhos e Gastos Anuais (html)

Conforme apresentado na Listagem 15 é possível a visualização da *tag translate*, ela é responsável pela tradução de cada título utilizado na aplicação.

As rotinas de cadastros, fazem o uso das mesmas regras – Requisições enviadas ao servidor a qual o servidor processa e retorna algum resultado para a aplicação. No cadastro de lançamentos foi optado por gravar além da data de lançamento, o mês e ano do lançamento, sendo um facilitador ao criar consultas personalizadas. Na Listagem 16 será mostrada a classe responsável pelo *CRUD* de lançamentos.

```java
@RestController
@RequestMapping("lancamentos")
public class LancamentoController extends ControllerBase implements Serializable {

    @Autowired
    private LancamentoRepository lancamentoRepository;


    @GetMapping("/")
    public List<Lancamento> Lancamentos() {
        return lancamentoRepository.findAllByUsuarioByMesByAno(usuarioComponent.getUsuario(), utilComponent.mesAtual(),
                utilComponent.anoAtual());
    }

    @PostMapping("/lancamentosByMesAno")
    public List<Lancamento> Lancamentos(@RequestBody Lancamento lancamento) {
        int mes = lancamento.getMes();
        int ano = lancamento.getAno();
        return lancamentoRepository.findAllByUsuarioByMesByAno(usuarioComponent.getUsuario(), mes, ano);
    }

    @DeleteMapping("del/{codigo}")
    public void delete(@PathVariable Long codigo) {
        lancamentoRepository.delete(codigo);
    }


    @DeleteMapping("deleteParcelamento/{codigoParcela}")
    public void deleteParcelas(@PathVariable Long codigoParcela) {
        lancamentoRepository.apagarParcelas(codigoParcela);
    }


    @PutMapping(path = "/edit")
    public void alterar(@RequestBody Lancamento lancamento) {
        lancamentoRepository.save(lancamento);
    }
```

Listagem 16 - Crud Lançamentos (Java)

Para os lançamentos parcelados o usuário informa o número de parcelas que deseja, o qual é enviado uma requisição para o servidor, onde o servidor processa os parcelamentos e devolve uma lista formatada para *view*, contendo id, data, grupo, valor, permitindo o usuário visualizar as parcelas para posteriormente salvar o lançamento.

Ao salvar os lançamentos é enviado uma lista de parcelas para o servidor, onde existe o método que grava no banco de dados. Segue Listagem 17 representando o processo realizado na geração das parcelas.

```
    @PostMapping(path = "/geraParcela")
    public List<ParcelamentoTo> gerarParc(@RequestBody ParcelamentoTo parcelamentoTo) {
        final int NUMERO_PARCELAS = parcelamentoTo.getNumParcela();
        final Calendar DATA_EMISSAO = Calendar.getInstance();

        List<ParcelamentoTo> listParc = new ArrayList<>();
        for (int i = 1; i <= NUMERO_PARCELAS; i++) {
            ParcelamentoTo parc = new ParcelamentoTo();
            DATA_EMISSAO.add(Calendar.MONTH,  amount: 1);
            if (i == 1) {
                DATA_EMISSAO.setTime(parcelamentoTo.getData());
            }
            parc.setId((long) i);
            parc.setData(DATA_EMISSAO.getTime());
            parc.setValor(parcelamentoTo.getValor());
            parc.setGrupo(parcelamentoTo.getGrupo());

            listParc.add(parc);
        }
        return listParc;
    }
}
```

Listagem 17 - Gerar Parcelas

5 CONSIDERAÇÕES FINAIS

O objetivo deste trabalho foi implementar um sistema *web* com interface rica, utilizando do conceito de *Single-Page-Application* (SPA), evitando que toda a página seja recarregada em cada requisição. O intuito principal da aplicação além de fornecer conhecimento referente às tecnologias e *frameworks* foi também fornecer uma interface simples e intuitiva aos usuários, por isso utilizado nomenclaturas amigáveis (Ganhos, gastos...), buscando facilitar o entendimento.

Para implementação do sistema o conjunto de *frameworks* utilizado foi o *AngularJs* e o *Spring*, os quais trouxeram vários benefícios, apesar de já existirem versões mais recentes do *Angular*, o *AngularJs* continua proporcionando alta produtividade, e o principal é a curva de aprendizado muito positiva, desde documentações de fácil entendimento a exemplos reais de desenvolvimento. A mesma opinião aplica-se ao *Spring*, além da facilidade em iniciar um projeto, existem muitos materiais afim de facilitar a vida do desenvolvedor.

Contudo, o *AngularJS* apresenta uma desvantagem referente a sua atualização, a qual a sua escrita mudou totalmente da versão do 1 para 2. Onde na versão 1 era escrito em *Javascript* e na versão 2 foi reescrita para *TypeScript*, mudando totalmente sua sintaxe. Porém para este projeto foi considerado a facilidade no aprendizado e a quantidade de materiais disponíveis relacionados ao AngularJs. De modo geral foi de grande valia obter o conhecimento sobre as tecnologias.

Como trabalho futuro toda estrutura criada principalmente no *back-end* será utilizada para integração com um aplicativo móvel já existente (*Wisecash*), onde o objetivo é permitir que o usuário realize lançamentos *off-line* via plataforma *mobile* e posteriormente esses dados sejam integrados sincronizados para a nuvem, permitindo que o usuário os manipule também via *Web*.

REFERÊNCIAS

JUNIOR, EID; GARCIA, F. G. **Finanças pessoais: como fazer o orçamento familiar.** São Paulo: Publifolha.

MACEDO Jurandir**. A Árvore do Dinheiro: guia para cultivar a sua independência financeira.** Rio de Janeiro: Elsevier, 2007.

OLIVEIRA, Eduardo **Ria - Rich Internet Applications: Uma Revisão dos principais Expoentes Da Area**. Recife: Unibratec, 2008.

LAWTON, George. **New ways to build rich internet applications. Industry Tends, IEEE Computer Society**, Agosto 2008, p. 10-12.

JUNIOR, Normandes; AFONSO, Alexandre. **Produtividade no Desenvolvimento de Aplicações Web com Spring Boot**, Janeiro 2017.

KLAUS, Allan. **O que é Single Page Application?**, 2016. Disponível em: < http://blog.locaweb.com.br/artigos/desenvolvimento-artigos/o-que-e-single-page-application> Acesso em: 29 Agosto 2017.

ANGULARJS, **AngularJs**, 2014. Disponível em: < https://angularjs.org//> Acesso em: 01 julho 2017.

SAXENA, Rahul. **Single Page Application (SPA) Using AngularJS, Web API and MVC 5**, 2014. Disponível em: <http://www.c-sharpcorner.com/uploadfile/rahul4_saxena/single-page-application-spa-using-angularjs-web-api-and-m/> Acesso em: 29 Agosto 2017.

MARETTI, André. **Requisitos Funcionais, Não Funcionais e De Domínio5**, 2014. Disponível em: <https://andremaretti.wordpress.com/2014/03/09/requisitos-funcionais-nao-funcionais-e-de-dominio/> Acesso em: 16 Setembro 2017.

BELEM, Thiago. **Armazenando informações com localStorage e sessionStorage**, 2012. Disponível em: < http://blog.thiagobelem.net/armazenando-informacoes-no-computador-do-visitante-com-localstorage-e-sessionstorage/> Acesso em: 28 Setembro 2017.

WALTENBERG, Rodrigo. **AngularJS: O que é e porquê utilizar**, 2016. Disponível em: < http://blog.algaworks.com/o-que-e-angularjs/> Acesso em: 30 Agosto 2017.

POSTGRESQL. **Postgresql x Recursos Presentes**. Disponível em < https://wiki.postgresql.org/wiki/Introdução_e_Histórico>. Acesso em: 29 Agosto 2017.

VISUAL PARADIGM. **Visual Paradigm for UML 13.2 Community Edition**. Disponível em http://www.visual-paradigm.com/product/vpuml/editions/community.jsp. Acesso em: 29 Agosto 2017.

CONTROLE DE VERSÃO. **Controle de Versão,Git, Github e Bitbucket – Afinal, o que é tudo isso ?**. Disponível em < https://aprendendowww.wordpress.com/2015/02/20/controle-de-versaogit-github-e-bitbucket-afinal-o-que-e-tudo-isso/>. Acesso em: 30 Agosto 2017.

GOOGLE TRENDS. **Google Trends**. Disponível em < https://trends.google.com.br/trends//>. Acesso em: 30 Agosto 2017.

RAMOS, Allan. **MVC – Afinal, é o quê ?**, 2015. Disponível em: < https://tableless.com.br/mvc-afinal-e-o-que/ > Acesso em: 27 Agosto 2017.

BERNARDO, Edson. **Introdução Jaspersoft Studio**, 2015. Disponível em: < https://edsonbernardo.wordpress.com/2015/01/23/introducao-jaspersoft-studio/ > Acesso em: 29 Agosto 2017.