UNIVERSIDADE FEDERAL DE SANTA CATARINA

Desenvolvimento de Protótipo de Sistema Web para Gerenciamento de Agência Lotérica Utilizando Programação Funcional Reativa

João Victor Mendes de Oliveira

Florianópolis – SC

2016/2

UNIVERSIDADE FEDERAL DE SANTA CATARINA

DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

BACHARELADO EM SISTEMAS DE INFORMAÇÃO

Desenvolvimento de Protótipo de Sistema Web para Gerenciamento de Agência Lotérica Utilizando Programação Funcional Reativa

João Victor Mendes de Oliveira

Trabalho de conclusão de curso submetido à Universidade Federal de Santa Catarina como parte dos requisitos para obtenção do grau de Bacharel em Sistemas de Informação.

Florianópolis – SC

2016/2

João Victor Mendes de Oliveira

Desenvolvimento de Protótipo de Sistema Web para Gerenciamento de Agência Lotérica Utilizando Programação Funcional Reativa

Trabalho de conclusão de curso submetido à Universidade Federal de Santa Catarina como parte dos requisitos para obtenção do grau de Bacharel em Sistemas de Informação.

Orientador: Prof. Dr. Leandro José Komosinski

Banca examinadora

Prof. Dr. Frank Augusto Siqueira

Prof. Dr. Maurício Floriano Galimberti

Agradecimentos

Aos meus pais, Edlon e Marta, que desde sempre me deram o suporte necessário para que eu chegasse até aqui.

À minha namorada, Jéssica Vargas, por todo o incentivo e apoio, moral e material, sem os quais eu não teria chegado tão longe.

Ao meu orientador, Leandro Komosinski, pelo grande auxílio durante toda essa jornada.

Resumo

Ferramentas para gerenciamento de fluxo de caixa são essenciais no dia a dia de qualquer empresa, e em uma agência lotérica elas se tornam imprescindíveis. Apesar de existirem *softwares* no mercado voltados a esse objetivo, muitas agências ainda utilizam apenas planilhas eletrônicas, ou até mesmo papel e caneta, para gerenciar as operações de caixa diárias da empresa. Considerando isso, este trabalho se propõe a desenvolver um protótipo de aplicação web que visa auxiliar o gerente de agências lotéricas no controle das operações realizadas no dia a dia da sua empresa, bem como calcular automaticamente o saldo dos terminais operacionais ao fim de cada dia. Empregando o uso de programação funcional reativa, objetiva usufruir das qualidades desse paradigma, como também simular as relações funcionais entre as células de uma planilha eletrônica.

Palavras-chave: Aplicação web, agência lotérica, programação funcional reativa, websocket.

Abstract

Cash flow management tools are essential in the daily operations of any company, and indispensable to business such as lottery agencies. Despite the existence of softwares aiming this solution in the market, many agencies still use just an electronic spreadsheet, or even only pen and paper, to manage daily cash operations. Taking that into account, this work proposes to develop a web application prototype which seeks to help lottery agencies managers in the control of the daily operations of their company, as well as automatically calculate the balance of its operating terminals at the end of each day. By using functional reactive programming, it aims to enjoy the qualities of this paradigm, as well as simulate the functional relations between the cells of an electronic spreadsheet.

Keywords: Web application, lottery agency, functional reactive programming, websocket.

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas

FRP – *Functional Reactive Programming*

CEF – Caixa Econômica Federal

PF – Programação Funcional

API – *Application Programming Interface*

JS - *JavaScript*

VPN – *Virtual Private Network*

HTTP – *HyperText Transfer Protocol*

TCP – *Transmission Control Protocol*

HTML – *HyperText Markup Language*

CSS – *Cascading Style Sheets*

NPM – *Node Package Manager*

ER – Entidade Relacionamento

DOM – *Document Object Model*

Sumário

1. INTRODUÇÃO

Uma parte fundamental da gestão de qualquer tipo de empresa é o gerenciamento de fluxo de caixa. Ter conhecimento detalhado sobre todo dinheiro que está entrando e saindo, sabendo onde está sendo gasto o capital que entra na empresa, é imprescindível para o sucesso de qualquer empreendimento.

Em uma agência lotérica isso não é diferente. O gerente de uma casa lotérica precisa ter conhecimento sobre todas as operações realizadas no seu estabelecimento, como por exemplo, quantos – e quais – jogos foram realizados no dia, quantos boletos foram pagos, quantos saques e depósitos foram realizados de contas da Caixa Econômica Federal, quantos bolões foram vendidos, dentre outras operações possíveis de se realizar em uma empresa desse tipo. Necessita, também, ter conhecimento sobre os valores de cada operação, quanto dinheiro entrou e saiu, bem como o saldo de cada terminal lotérico existente em sua agência.

Para fazer a administração correta e o acompanhamento desses valores, buscando ter amplo controle sobre as operações e o capital da empresa, é imprescindível que se faça uso de ferramentas que auxiliem na execução das tarefas diárias de gerenciamento da agência lotérica. Uma ferramenta amplamente utilizada para esse objetivo é a planilha eletrônica, cuja principal qualidade é a possibilidade de ser pré-programada com diversas funções que ditam relações entre as células, calculando automaticamente os valores resultantes de uma célula na outra, no momento em que o usuário insere os dados de movimentação e controle da sua empresa.

Atualmente, com a popularização e utilização em larga escala de computadores e da internet, se faz possível a criação de sistemas complexos para auxiliar o gerente lotérico nas atividades necessárias para que consiga alcançar o

controle pleno das operações realizadas na sua agência, bem como a ter conhecimento total dos gastos e recebimentos envolvidos em cada uma das operações, tudo isso de forma eficiente, economizando tempo do gestor, para que ele possa se dedicar também às outras atividades da sua lotérica.

O presente trabalho apresenta o relatório de desenvolvimento de um protótipo de aplicação web, que, utilizando-se da Programação Funcional Reativa, implementa a principal qualidade apresentada pelas planilhas eletrônicas, visando alcançar todos os objetivos descritos acima.

1.1 Motivação e Objetivos

De acordo com “Perguntas frequentes sobre Licitações de Unidades Lotéricas” (CAIXA ECONÔMICA FEDERAL), a Caixa Econômica Federal não disponibiliza aos seus gerentes lotéricos ferramentas para fazer o gerenciamento e o controle das suas agências, ficando a cargo do próprio gerente buscar soluções que o auxiliem nesse trabalho. Apesar de existirem, sistemas informatizados com esse intuito não são unanimidade entre os agentes lotéricos, sendo que muitos ainda recorrem às tradicionais planilhas (tanto eletrônicas quanto físicas) na hora de fazer a contabilidade e o controle das operações diárias.

Nos dias de hoje, os usuários estão acostumados a resolver todos os seus problemas com a internet. As pessoas se habituaram a utilizar seu navegador preferido para acessar os mais diversos serviços, desde redes sociais até soluções empresariais. Essas mesmas pessoas não costumam gostar de precisar instalar softwares em seus computadores, pois além de acharem trabalhoso, muitas nem sabem ao certo como fazê-lo. Elas estão acostumadas a acessar um website e resolver tudo por ali, sem a necessidade de ficar reinstalando o mesmo programa sempre que houver uma nova atualização disponível.

Sendo assim, o presente trabalho objetivou desenvolver um protótipo de *software* que possibilitasse ao gerente de uma agência lotérica realizar o controle das atividades diárias da agência, como fechamento de caixa, cálculo de saldo da conta da agência, previsão de lucro de venda de jogos, quantidades de operações realizadas, bolões, fiados, etc. Tudo isso de forma simples, prática e eficiente.

O *software* foi desenvolvido na forma de uma plataforma *web*, acessível através de qualquer navegador de internet, utilizando tecnologias modernas e poderosas, como *JavaScript*, *Node.js* e *Bacon.js*.

A aplicação foi implementada utilizando o paradigma de Programação Funcional Reativa (*Functional Reactive Programming*, ou simplesmente FRP), através da API *Bacon.js*.

1.2 Descrição do Problema

Atualmente, existe no mercado uma quantidade alta de sistemas voltados para o controle e gerenciamento de agências lotéricas. Porém, muitas agências não utilizam nada mais sofisticado que uma planilha, seja ela eletrônica ou física. Por mais complexa que possa vir a ser uma planilha eletrônica, com diversos campos calculáveis e funções programáveis, não deixa de ser uma plataforma limitada, não só pelo critério de funcionalidades possíveis, mas também nos quesitos usabilidade e interface de usuário. Na figura 1 vemos um exemplo de planilha eletrônica utilizada para esse fim.

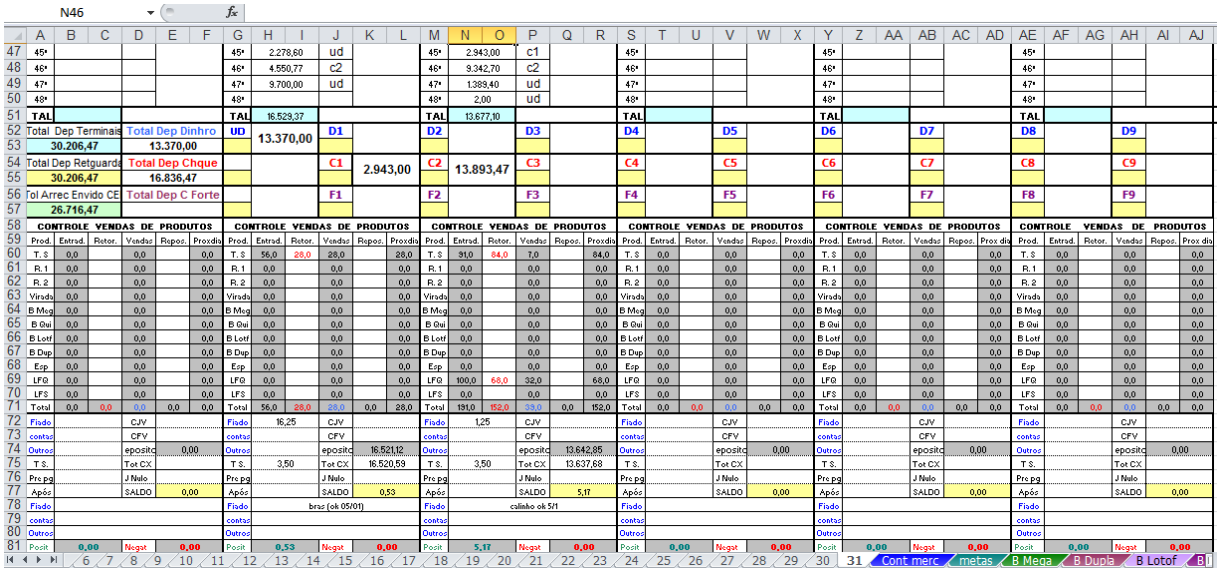

Figura 1 - Planilha de Controle de Agência Lotérica.

Fonte: Lotérica Gravatá.

No caso de usabilidade, planilhas causam certas inconveniências. Por exemplo, a necessidade de se criar uma planilha nova para informar os dados de cada dia, o que pode ser amenizado criando um arquivo para cada mês, contendo uma aba para cada dia do referente mês. Ainda assim, existe a necessidade de criar vários arquivos novos a cada curto espaço de tempo, e após alguns meses ou anos, acabaríamos com vários arquivos separados, sem ligação alguma, impossibilitando cruzamento de informações ou qualquer consulta eficiente.

Além dessas limitações, existe ainda a necessidade de se adquirir uma licença de *software* do fabricante (e.g., Microsoft Excel), resultando em um gasto extra por parte do usuário.

Mesmo os *softwares* existentes muitas vezes não oferecem uma solução satisfatória, entregando um produto muitas vezes ultrapassado e que não está adequado às necessidades e desejos do usuário moderno.

2. SOLUÇÕES EXISTENTES

Foi realizada uma pesquisa para encontrar soluções existentes para o problema que este trabalho se propõe a resolver. Os resultados encontrados são descritos na tabela 1.

Tabela 1 - Soluções existentes

SISLOT	Promete executar diversas atividades de gerenciamento e controle de agência lotérica, porém é um aplicativo *desktop*, sem versão *web* – acesso remoto ao sistema é feito através de serviço de nuvem de terceiros. (MIDIA DREAM SOLUÇÕES, 2015).
SGCL	Apesar de indicar que possui versão *web*, sob a nomenclatura de “Aplicativo nas Nuvens”, nas demonstrações disponíveis no site mostra apenas versão *desktop*. (SGCL, 2013).
Prolotérica	Aparenta ser um sistema competente, porém não possui versão *web*. Para acessar remotamente, a possibilidade anunciada é o uso de VPN. (PROLOTÉRICA, 2015).
Megafllex	Afirma oferecer uma gama bastante grande de funcionalidades, mas também não possui versão *web*. (MEGAFLLEX,

	2015).
Dourasoft	Das ferramentas encontradas é a que possui o *website* mais condizente com os padrões atuais de interface e usabilidade, mas ainda assim, não possui versão *web* do seu produto. (DOURASOFT, 2015).
Loteria Exata	Garante funcionar puramente via *web*, sem a necessidade de instalação de nenhum *software* na máquina do usuário. Também alega ser acessível através de dispositivos móveis (*smartphones*, *tablets*), porém é a ferramenta que possui menos informações disponíveis para pessoas que não são clientes. Pelo informado, aparenta ser a ferramenta menos completa. (LOTERIA EXATA, 2013).
Planilha Excel pré-programada	Única ferramenta a qual foi possível ter contato total com suas funcionalidades, pois é a ferramenta utilizada na agência lotérica a qual o autor teve acesso. Apesar de oferecer diversas funcionalidades, é bastante limitada pela sua própria natureza de planilha. O

	ponto positivo dessa solução seria a consistência dos dados, devido às células serem bem “amarradas” através das diversas fórmulas já existentes na ferramenta.

3. FUNDAMENTAÇÃO TEÓRICA

3.1 Aplicação Web

Como sugerido em “*What Are Web Applications*?” (ACUNETIX, 2012), desde o seu surgimento, a internet tem evoluído constantemente. Tanto em questão de infraestrutura, com cada vez mais locais recebendo sinal de rede, e a qualidade dessa rede continuamente aumentando, quanto em questão de conteúdo, com o sempre crescente número de empresas e entusiastas em geral criando e disponibilizando seus trabalhos a um grupo cada vez maior de usuários.

Aplicações Web (ou *Web Application*) são programas de computador que se utilizam dessa infraestrutura para possibilitar aos usuários o acesso às suas funcionalidades, remotamente, através de um navegador de internet (WIKIPEDIA, 2015a). Basicamente, tratam-se de *softwares* que possibilitam aos internautas enviar e receber dados de bancos de dados remotos através do seu *browser* preferido. Enquanto o usuário interage com o sistema, informações são geradas dinamicamente no navegador pela aplicação, através de um servidor *web*.

3.2 WebSocket

A maneira tradicional de estabelecer conexões entre cliente e servidor em uma aplicação *web* é o paradigma de requisição e resposta do protocolo HTTP. O

cliente faz uma requisição ao servidor, este responde e nada acontece até que o usuário execute mais alguma ação que gere uma requisição ao servidor.

De acordo com Ubl e Kitamura (2010), surgiram algumas tecnologias que simulam ações originadas pelo servidor, utilizando técnicas como a sondagem longa, em que o cliente abre uma conexão com o servidor e esta se mantém aberta até que uma resposta seja enviada. Entretanto, essas técnicas se tornam ineficientes para mensagens pequenas, devido ao *handshake* do protocolo TCP e ao cabeçalho do HTTP.

Diferente do protocolo HTTP, explicam Ubl e Kitamura (2010), o protocolo *WebSocket* proporciona comunicação bidirecional sobre uma única conexão TCP. Ele facilita a transferência de dados em tempo real do cliente para o servidor e vice-versa. Essa maior interatividade é possível, pois o protocolo proporciona uma forma padronizada para o servidor enviar mensagens ao cliente sem ser solicitado, permitindo troca de mensagens entre as partes sem fechar a conexão, como demonstrado na Figura 2.

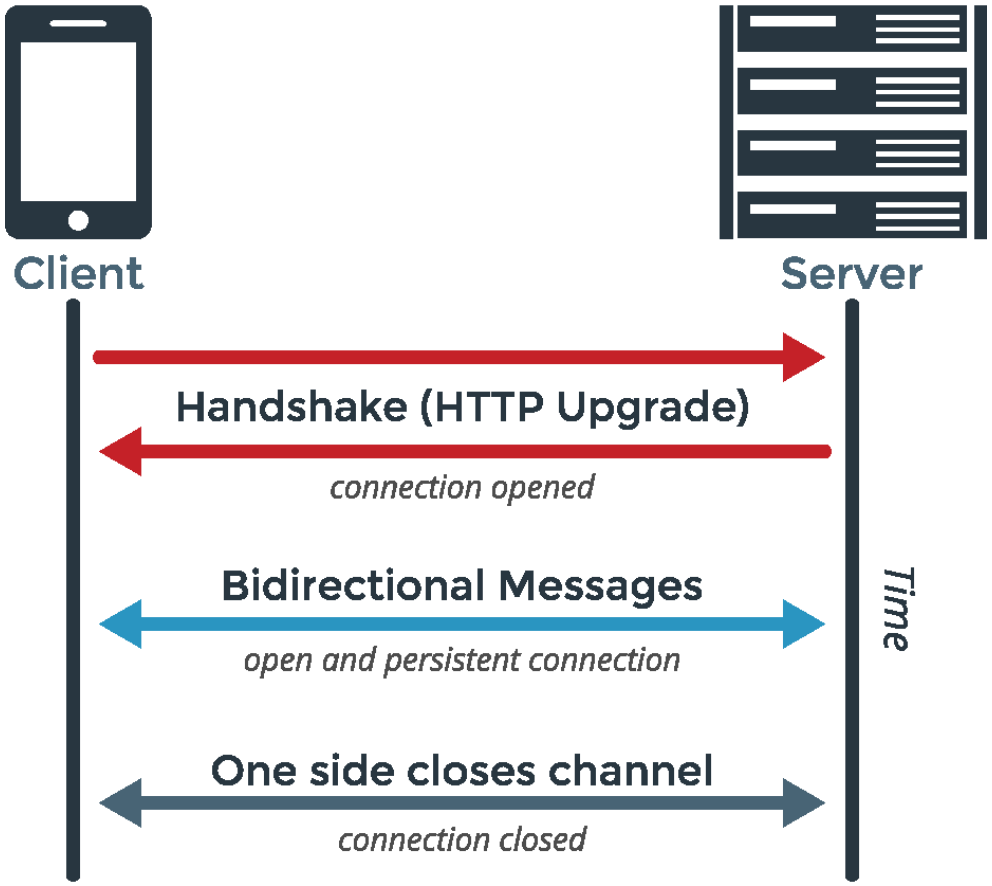

Figura 2 - Diagrama *WebSocket*.

Fonte: https://www.pubnub.com/blog/2015-01-05-websockets-vs-rest-api-understanding-the-difference/

A conexão é feita através da porta TCP 80, e dessa forma não corre riscos de ser impedido por firewalls que bloqueiam conexões Internet não-*web*. Atualmente, o protocolo *WebSocket* é suportado pela maioria dos principais navegadores, como Google Chrome, Mozilla Firefox, Internet Explorer, Opera, Safari (WIKIPEDIA, 2016).

Esse protocolo foi escolhido para o desenvolvimento deste trabalho para que se tenha uma padronização da programação orientada a eventos no projeto, através da biblioteca *Socket.io*, que facilita o uso do protocolo *WebSocket* na linguagem *JavaScript*.

3.3 Programação Funcional Reativa

Para entendermos FRP (*Functional Reactive Programming*) é preciso primeiro entender, separadamente, o que é programação reativa e o que é programação funcional.

Conforme (MEDEIROS, 2014; BONÉR, FARLEY, *et al.*, 2014), Programação Reativa é um paradigma de programação orientado a fluxos de dados assíncronos e a propagação de mudanças. Nesse paradigma, eventos (e.g., cliques em botões da tela) formam um fluxo (*stream*) de eventos assíncronos. Um *stream* é uma sequência contínua de eventos ordenados no tempo, como ilustrado na figura 3.

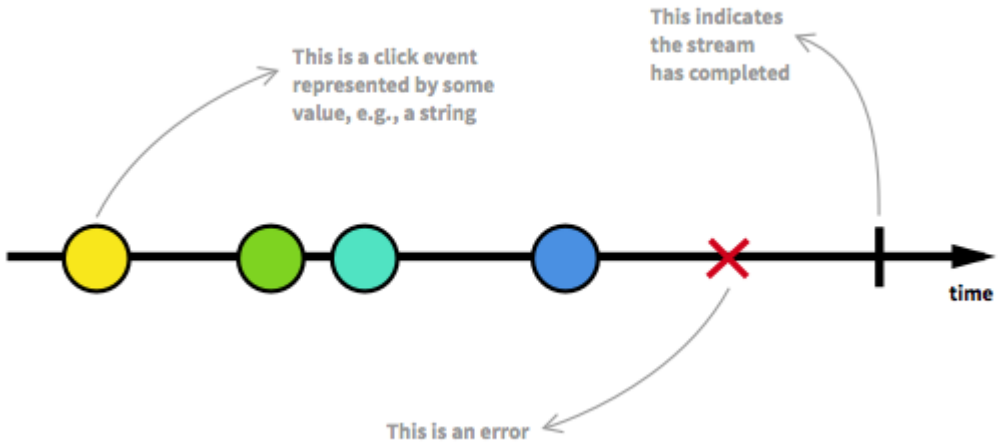

Figura 3 – Diagrama de *marbles* de um *Stream* de eventos.

Fonte: https://gist.github.com/staltz/868e7e9bc2a7b8c1f754

Um exemplo tradicional comparando com o paradigma de programação imperativa: a expressão “a := b + c”, define o valor de “a” como sendo o resultado do somatório dos valores de “b” e “c” no momento da execução. Sendo que, após isso, caso os valores de “b” ou “c” mudem, o valor de “a” se mantém inalterado.

Entretanto, na programação reativa, o valor de “a” seria atualizado com os novos valores de “b” ou “c” automaticamente (STOVELL, 2010). De certa forma, a Programação Reativa se assemelha às planilhas eletrônicas (e.g., Microsoft Excel, LibreOffice, Planilhas Google), pois quando uma célula é alterada, a mudança se propaga a todas as outras células que utilizam a primeira, alterando seus valores (STACK OVERFLOW, 2009). Portanto, o principal ponto positivo de se utilizar planilhas eletrônicas é mantido com o uso desse paradigma de programação.

Programação Funcional é outro paradigma de programação, que segundo Wikipedia (2015b), trata a computação como uma avaliação de funções matemáticas, enfatizando a aplicação de funções, ao contrário da programação imperativa, que enfatiza mudanças no estado do programa. É um paradigma declarativo, ou seja, a programação é feita com expressões ou declarações no lugar de instruções. Tem como base o cálculo lambda, que é um sistema formal para cálculo de funções desenvolvido na década de 1930 pelo matemático Alonzo Church. Como demonstrado na figura 4, o paradigma funcional pode ser entendido como um mapeamento dos valores de entrada nos valores de retorno, através de funções.

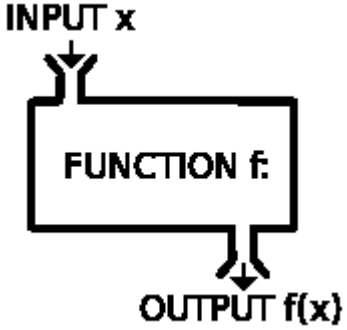

Figura 4 - Entrada -> Função -> Saída.

Fonte: https://zeroturnaround.com/rebellabs/there-is-more-to-code-quality-than-just-pretty-vs-ugly

Existem várias linguagens puramente funcionais, ou que foram desenvolvidas com o objetivo de suportar o paradigma funcional (e.g., Hope, Haskell, F#), entretanto várias outras linguagens que originalmente não possuíam essa intenção, atualmente suportam esse tipo de programação, como por exemplo, Java, C++, C#, PHP, Perl. Essas linguagens passaram a empregar funções básicas de PF, como *map*, *filter*, *reduce*, lambda (WIKIPEDIA, 2015b).

Programação Funcional Reativa se trata de um paradigma de programação que engloba esses dois paradigmas. Portanto, trata-se de *streams* – ou fluxos – de dados assíncronos que sofrem alterações ao longo do tempo, que podem ser mapeados, filtrados, reduzidos, compostos (JONES e BLACKHEATH, 2015; WIKIPEDIA, 2015c). Ou seja, um *stream* pode ser utilizado como entrada para outro, ou ambos podem ser fundidos, ou ainda mapear os valores de um *stream* para um novo, ou filtrar os valores de um *stream* para conseguir outro contendo apenas os eventos que interessam.

3.4 Tecnologias Utilizadas

Nesta seção serão descritas brevemente algumas tecnologias utilizadas no desenvolvimento do projeto, além das tradicionais HTML, CSS e JavaScript.

3.4.1 Node.js

Trata-se de um ambiente de execução *open-source* e multiplataforma que possibilita o desenvolvimento no lado do servidor de aplicações web na linguagem *JavaScript*, interpretando essa linguagem através do motor “V8” da Google (NODE.JS FOUNDATION, 2016a).

Segundo Ribeiro Júnior (2012), *Node.js* utiliza uma arquitetura de entrada e saída de dados direcionada a eventos e assíncrona. Essas características tem por

objetivo otimizar o processamento e a escalabilidade de aplicações *web* com um elevado número de operações de entrada e saída, bem como aplicações em tempo real.

Operando em um único *thread* e utilizando chamadas I/O não bloqueantes (assíncronas), explica Ribeiro Júnior (2012), uma aplicação *Node* é capaz de suportar dezenas de milhares de conexões concorrentes sem o custo de troca de contexto de *threads*.

Node.js possui uma extensa coleção de módulos que auxiliam no desenvolvimento das aplicações, os quais são disponibilizados através da ferramenta NPM (*Node Package Manager*), disponível no *website* https://www.npmjs.com. Foram utilizados no desenvolvimento deste trabalho os módulos: Express, um *framework* que simplifica ainda mais o desenvolvimento de aplicações *web* com *Node* (NODE.JS FOUNDATION, 2016b); Socket.io, que possibilita o uso do protocolo *WebSocket* para comunicação bidirecional em tempo real e baseada em eventos entre servidor e cliente (SOCKET.IO, 2016); mysql, um driver para fazer a conexão com o banco de dados MySQL (GEISENDÖRFER, 2016).

3.4.2 Bacon.js

Conforme Nilsson (2013), *Bacon.js* é uma biblioteca que possibilita a implementação de programação assíncrona com *streams* observáveis – conceitos de Programação Funcional Reativa – em *JavaScript*. Criada por funcionários da empresa Reaktor, da Finlândia, é apoiada pela mesma, sendo utilizada no dia a dia da empresa para desenvolver seus produtos, além de ter seu código aberto e

receber auxílio da comunidade, através do repositório GitHub (PAANANEN, 2013; PAANANEN, 2016a).

Segundo Paananen (2012), *Bacon* não é um *plug-in* e nem dependente de *jQuery*, mas caso encontre essa biblioteca no projeto, adiciona o método "*asEventStream*" ao *prototype* do objeto *jQuery*. Esse método é utilizado para capturar eventos em variáveis de tipo "*EventStream*", ou fluxos de eventos que, por sua vez, podem ser mapeadas, filtradas, fundidas, dentre várias outras operações funcionais. Essas variáveis são também chamadas de "*observables*", ou "observáveis", em português.

Outro conceito presente em *Bacon.js* é chamado de "*Property*" – ou Propriedade, em português. Uma *property* – que também é um *observable* – é quase como um *EventStream*, com a diferença de que possui um "valor atual". Para exemplificar essa diferença podemos pensar que cliques do mouse (eventos discretos) são *EventStream*, enquanto que a posição do ponteiro na tela (valor dinâmico, mas que possui um "estado atual") é uma *Property* (PAANANEN, 2016b).

Na figura 5, abaixo, vemos um exemplo simples, em que eventos "*keyup*" (o ato de soltar uma tecla após pressioná-la) em qualquer local do documento são capturados em um fluxo de eventos (através do método "*asEventStream*"). Em seguida, esse *stream* é filtrado para capturar apenas os eventos "*keyup*" em teclas de espaço, e então, para cada novo evento nesse *stream* é emitido um alerta para o usuário.

```
var allKeyUps = $(document).asEventStream("keyup")

var spaceBarKeyUps = allKeyUps
  .filter(function(event) { return event.keyCode == 32 })

spaceBarKeyUps.onValue(function(event) { alert("you pressed space") })
```

Figura 5 - Exemplo *Bacon.js*.

Fonte: http://raimohanska.github.io/bacon.js-slides/0.html

4. SOLUÇÃO PROPOSTA

No processo de realização deste projeto, para que fosse possível definir com mais clareza quais são as principais necessidades do sistema, foi feito, em um primeiro momento, o levantamento de requisitos do *software* em questão. Analisando-se a planilha a qual se teve acesso, e em contato direto com uma gerente de agência lotérica, foi possível levantar os requisitos que serão descritos a seguir.

4.1 Requisitos Funcionais

i. Deve ser possível cadastrar os terminais existentes na agência.

ii. Deve ser possível cadastrar os operadores de terminais.

iii. Deve ser possível cadastrar turnos de trabalho.

iv. Deve ser possível inserir, para cada terminal, para um determinado turno:

   a. Informações de depósitos categorizados por dinheiro ou cheque;

   b. Informações de vendas de produtos externos ao terminal (*e.g.*, Tele Sena, Bolões);

   c. Informações de atividades financeiras particulares à agência (*e.g.*, fiados, pagamento de contas);

d. O valor do saldo final descrito no relatório gerado pelo próprio terminal;
e. Possuindo as informações descritas nos itens anteriores, o sistema deve calcular o saldo (positivo ou negativo) de cada terminal;
f. Informações de pagamentos de benefícios (*e.g.*, INSS, FGTS);
g. Informações de quantidades e valores de venda de jogos (*e.g.*, Mega-Sena, Quina), devendo calcular o valor da comissão da agência;
h. Informações de prêmios pagos, contabilizando a quantidade e o valor total;
i. Informações de transações financeiras (saques e depósitos) de contas Caixa, bem como saques para contas do Banco do Brasil;
j. Informações de quantidade e valores de estornos e invalidações de jogos;

v. Deve ser possível inserir informações de valores enviados por malote para a Caixa Econômica Federal.

vi. Deve ser possível inserir informações detalhadas de cada depósito feito na CEF (*e.g.*, Valor, Número do Malote, Hora de Envio), tanto em dinheiro quanto em cheques.

vii. Deve possibilitar registro de créditos e débitos nas contas 043 e 003 da agência, bem como o saldo atual disponível nessas contas.

viii. Deve calcular a diferença entre valor arrecadado e valor depositado em um mesmo dia.

4.2 Requisitos Não-Funcionais

i. O sistema deve ser compatível com os principais navegadores do mercado.

ii. As telas do sistema devem apresentar a mesma identidade visual.

iii. A licença de software do sistema deve ser de código aberto.

4.3 Arquitetura da Solução

Como toda aplicação *web*, a solução proposta é composta de uma interface com o usuário, chamada de “cliente”, e outra parte que consiste da lógica da aplicação e do armazenamento dos dados, chamado de “servidor”. O cliente roda em um navegador *web*, e é onde o usuário insere dados, busca informações, interage com o sistema como um todo, gerando eventos. O servidor, hospedado remotamente e acessado via internet, é onde esses eventos são validados e processados, persistindo os resultados em uma base de dados e enviando as informações necessárias ao cliente.

A comunicação entre cliente e servidor é feita através do protocolo *WebSocket*, implementado no presente trabalho utilizando o *framework Socket.io*. O cliente foi implementado utilizando HTML, CSS e *JavaScript*, com o auxílio das bibliotecas *jQuery* e *Bacon.js*. A parte de servidor foi desenvolvida utilizando também *JavaScript*, através do *Node.js*, facilitada pelo *framework Express*. No servidor também foi utilizado *Bacon.js* para implementar o paradigma de Programação Funcional Reativa. Para conexão com o banco de dados foi utilizado o módulo “mysql” do *Node.js*.

A figura 6 ilustra a arquitetura do sistema proposto.

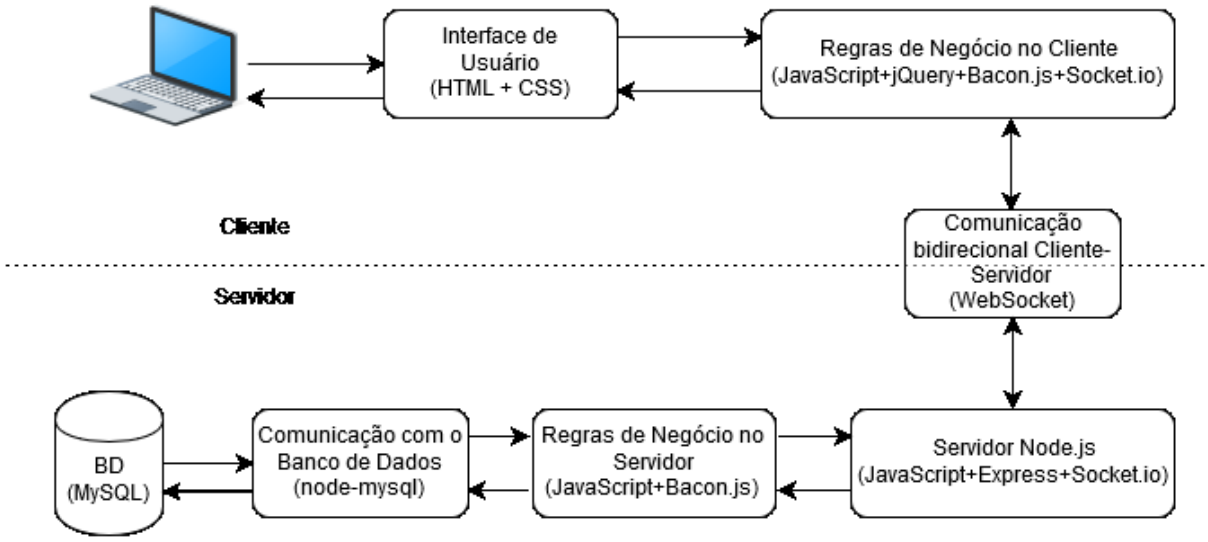

Figura 6 - Arquitetura da solução proposta.

5. DESENVOLVIMENTO

5.1 Banco de Dados

Após analisar todos os requisitos, considerando também o conhecimento do autor com relação ao negócio, foi criado o modelo Entidade-Relacionamento da aplicação, que pode ser visualizado na figura 7.

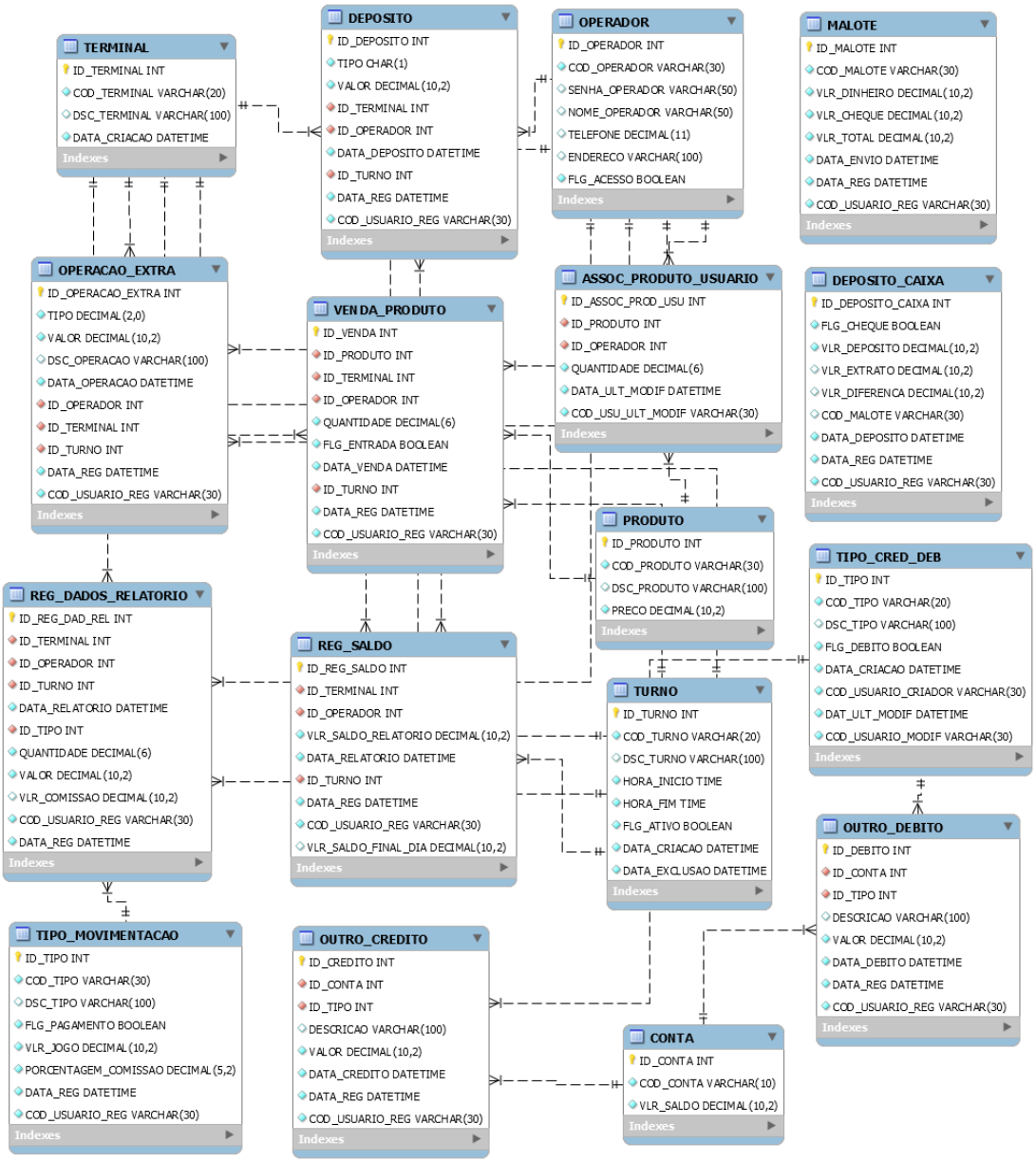

Figura 7 - Modelo ER da Aplicação.

Para possibilitar a implementação de todos os requisitos, foram criadas as seguintes tabelas:

- “TERMINAL”, que guarda informações dos terminais existentes na agência;
- “USUARIO”, para guardar informações de cada operador dos terminais, bem como os usuários que possuem acesso à aplicação desenvolvida neste trabalho;
- “DEPOSITO”, servindo o propósito de salvar informações de depósitos realizados no dia pelos operadores dos terminais;
- “OPERACAO_EXTRA”, para persistir os dados referentes às operações extra-terminal (como fiados, pagamento de contas da agência, entre outros);
- “REG_DADOS_RELATORIO”, para registro dos dados de movimentação presentes no relatório diário dos terminais;
- “TIPO_MOVIMENTACAO”, vinculada à tabela anterior, contendo informações sobre os tipos de movimentação presentes no relatório;
- “PRODUTO”, referente aos produtos oferecidos pela lotérica que não são processados pelos terminais (e.g., raspadinhas);
- “ASSOC_PRODUTO_USUARIO”, responsável por guardar as quantidades de cada produto que cada operador possui;
- “VENDA_PRODUTO”, que registra as vendas de produtos que cada operador efetuou em um determinado dia;
- “REG_SALDO”, responsável por manter o registro de saldo final do terminal para um determinado dia, levando em conta o valor do relatório e todos os outros dados de movimentação;

- “TURNO”, que possibilita o cadastro de turnos de trabalho, em caso de operadores que não trabalhem o dia inteiro;
- “CONTA”, para cadastrar as contas bancárias da agência;
- “OUTRO_DEBITO”, “OUTRO_CREDITO” e “TIPO_CRED_DEB”, referentes às operações financeiras dessas contas (além das contempladas pelas demais tabelas);
- “DEPOSITO_CAIXA”, para salvar informações de depósitos feitos à CEF;
- “MALOTE”, contendo informações de malotes enviados à CEF.

5.2 Requisito: Cadastro de Terminais

Após análise dos requisitos, foram definidas as classes, sendo Terminal composta pelos atributos: código, descrição e data de criação. Cadastro de Terminais foi o primeiro a ser desenvolvido. Foi criada uma página HTML simples, que pode ser vista na figura 8, com apenas duas caixas de texto para inserção dos dados, um botão para efetuar o cadastro e um botão para voltar à tela anterior. No *script* da página foi utilizado *jQuery* para manipulação dos elementos do DOM, *socket.io* para comunicação com o servidor e *Bacon*.js para validação dos valores inseridos e liberação do botão.

Figura 8 - Página de cadastro de terminais, utilizando navegador Firefox.

Na figura 9 vemos uma parte da implementação que faz uso de *Bacon.js*, em que eventos "*keyup*" em um campo de texto são capturados em um fluxo de eventos (através do método "*asEventStream*"), onde são mapeados para o valor do campo de texto no momento do evento (ou seja, é criado um novo *stream* no lugar do original, onde cada evento "*keyup*" é substituído pela *string* que corresponde ao conteúdo do campo de texto naquele momento), e então esse fluxo de eventos é transformado em uma *property*, com o valor inicial sendo o resultado da função "*value*" nesse primeiro momento. Essa *property* é então atribuída a variáveis para uso posterior, utilizando os respectivos campos de texto como parâmetro.

```
function valorCampoDeTexto(campoDeTexto) {
    function valor() { return campoDeTexto.val() }
    return campoDeTexto.asEventStream("keyup").map(valor).toProperty(valor());
}
var codTerminal = valorCampoDeTexto($("#codTerminal"));
var dscTerminal = valorCampoDeTexto($("#dscTerminal"));
```

Figura 9 - Uso de Bacon.js.

As *properties* recém-criadas foram utilizadas para informar se os campos estão ou não preenchidos, e com isso habilitar, ou desabilitar, o botão de cadastro. A figura 10 demonstra como isso é feito via código: é criada uma nova *property* que mapeia os eventos da *property* “codTerminal” utilizando a função “preenchido”. Ou seja, a cada novo evento em “codTerminal” (nesse caso, “evento” trata-se da *string* contida na caixa de texto após uma nova tecla ser pressionada), esse mesmo evento é convertido para o retorno da função “preenchido”, utilizando o próprio evento (*string*) como parâmetro. Dessa forma, criamos a *property* “codigoPreenchido”, que trata-se de um fluxo de eventos booleanos, sendo que o último evento transmitido é utilizado como “valor atual”.

A mesma funcionalidade foi implementada para o campo referente à descrição do terminal, e então essas duas *properties* de eventos booleanos foram combinadas para produzir uma nova, denominada “botaoAtivo”, que a cada novo evento executa a função “setAtivo”, passando o botão de cadastro e o valor do seu próprio evento como parâmetros, a fim de definir se o botão é habilitado ou desabilitado.

```
function preenchido(x) { return x.length > 0 }
function setAtivo(elemento, ativo) { elemento.attr("disabled", !ativo) }
var codigoPreenchido = codTerminal.map(preenchido);
var descricaoPreenchida = dscTerminal.map(preenchido);
var botaoAtivo = codigoPreenchido.and(descricaoPreenchida);
botaoAtivo.assign(setAtivo, btnCadastro);
```

Figura 10 - Controle de botão com Bacon.js.

Podemos perceber que, ao utilizar os conceitos de *streams* e *properties*, o tratamento de eventos é simplificado, além de melhorar consideravelmente a legibilidade do código. Para um melhor entendimento dos fluxos de evento e suas transformações, na figura 11 é demonstrado, através de um diagrama de *marbles*, um exemplo dos eventos que podem ocorrer nessa etapa do processo.

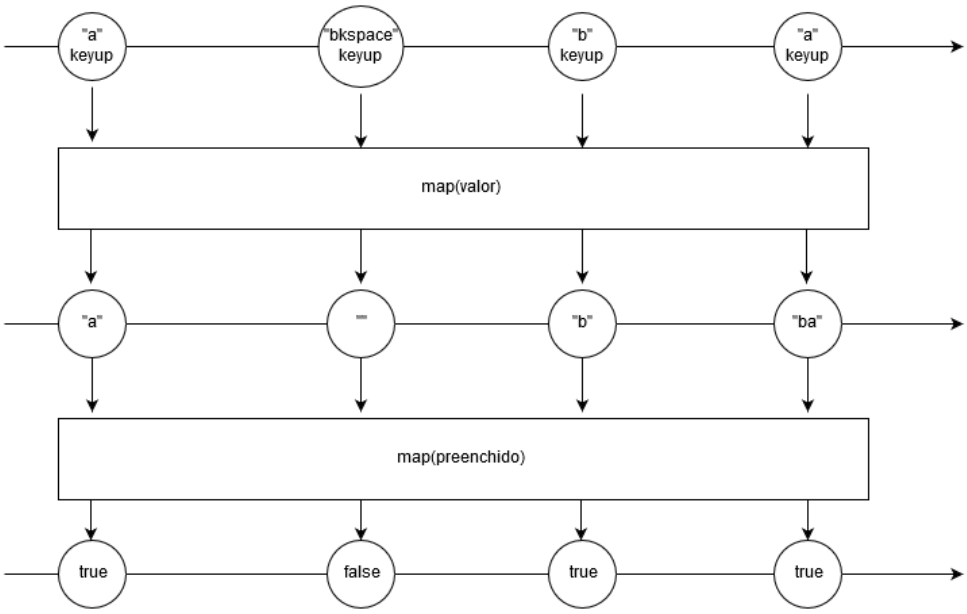

Figura 11 - Diagrama de *marbles* demonstrando fluxos de possíveis eventos e suas transformações.

O envio dos dados de cadastro para o servidor é feito através de um disparo de evento da biblioteca *socket.io*, denominado "*emit*", que é então capturado dentro do evento "*connection*" no lado servidor, onde os dados recebidos são utilizados para criar um novo objeto da classe Terminal e são então salvos no banco de dados MySQL, através do módulo *node-mysql*. Esse funcionamento é demonstrado nas figuras 12 e 13.

```
function limpaInputs() { $('input[type="text"]').val(''); }

btnCadastro.click(function () {
    socket.emit('cadastroTerminal', $("#codTerminal").val(), $("#dscTerminal").val());
    limpaInputs();
});

socket.on("terminalCadastrado", function (strMsg) { alert(strMsg); });
```

Figura 12 - Envio dos dados pelo cliente via *WebSocket.*

```
io.on('connection', function (socket) {
    socket.on('error', function (err) {
        console.log(err);
    });

    socket.on('cadastroTerminal', function (codTerminal, dscTerminal) {
        var novoTerminal = new Terminal(codTerminal, dscTerminal);
        novoTerminal.save();
    });

    Terminal.prototype.save = function () {
        var self = this;
        connection.connect('SELECT COUNT(*) as cont FROM TCC.TERMINAL WHERE COD_TERMINAL = ' + self.codigo + ' ;', function (rows) {
            if (rows[0].cont === 0) {
                connection.connect('INSERT INTO TCC.TERMINAL (COD_TERMINAL, DSC_TERMINAL, DATA_CRIACAO) VALUES ("' +
                    self.codigo + '","' + self.descricao + '","' + self.dataCriacao + '");', function (rows) { });
                socket.emit('terminalCadastrado', 'Terminal cadastrado com sucesso!');
            } else {
                socket.emit('terminalCadastrado', 'Já existe um Terminal com esse código!');
            };
        });
    }
});
```

Figura 13 - Recebimento dos dados pelo servidor via *WebSocket.*

5.3 Requisito: Cadastro de Operadores

Definiu-se a classe Operadores como possuindo os atributos: código, senha, nome, telefone, endereço e um booleano para informar se o operador em questão possui acesso ao sistema.

O desenvolvimento deste requisito ocorreu de forma similar ao de Cadastro de Terminais, utilizando *Bacon.js* para validação dos campos código, senha e nome, para então liberar o botão de cadastro; *socket.io* para comunicação com o servidor *node*; e o módulo *node-mysql* para conexão com o banco de dados. Na figura 14 é possível visualizar a página referente a esse requisito.

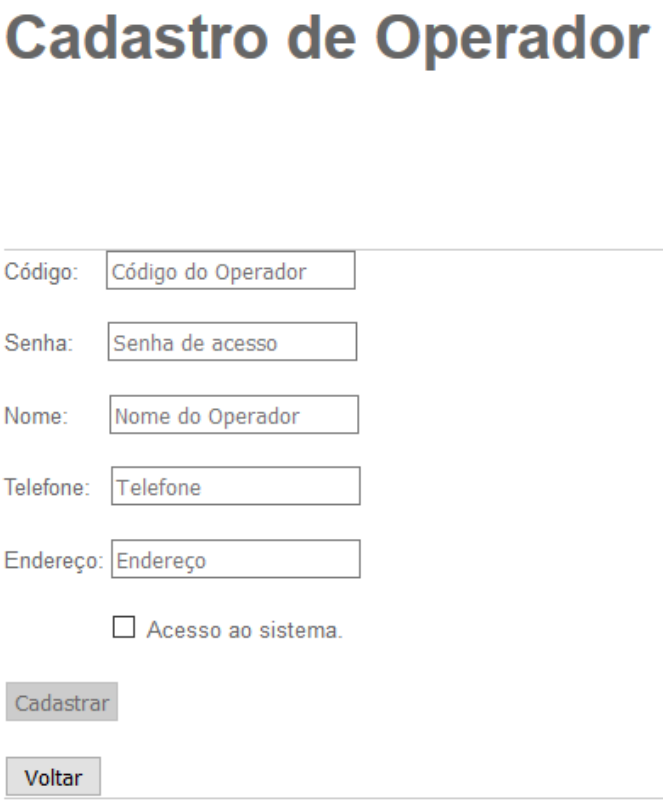

Figura 14 - Página de cadastro de operador, utilizando navegador Firefox.

A implementação desses dois requisitos, justamente por serem os primeiros a serem desenvolvidos, envolveu outros aspectos do sistema, como a criação das páginas HTML e CSS de *login* e de menu principal, bem como seus respectivos arquivos de script, tanto no lado cliente como no lado servidor, para validações e comunicação com o banco de dados. Na figura 15 vê-se a página de Login, enquanto na figura 16 pode-se visualizar a página de Menu.

Figura 15 - Página de *login*, utilizando navegador Edge.

A página de *login* possui um campo para o código de usuário e outro para a senha, bem como um botão para entrar no sistema. Semelhante ao que foi feito nos dois requisitos anteriores, foi utilizado *Bacon.js* para validar se os campos estavam preenchidos e então liberar o botão de entrada. Também foi utilizado *socket.io* para comunicação com o servidor *node* e o módulo *node-mysql* para consulta no banco de dados, que busca pela quantidade de registros existentes com a combinação de usuário e senha digitados, retornando uma mensagem de "usuário/senha não encontrado" no caso de a consulta retornar zero. Nesse momento também é verificado o atributo booleano do usuário para decidir se libera ou não a entrada do mesmo no sistema.

A página de menu principal, por sua vez, apenas mostra as opções do sistema, com *links* para levar o usuário às páginas desejadas, como Cadastro de Terminais ou de Operadores, por exemplo.

Figura 16 - Página de menu do sistema, utilizando navegador Edge.

5.4 Requisito: Cadastro de Turno

Pelo fato de alguns funcionários trabalharem em tempo integral, enquanto outros trabalham apenas meio período, e levando em consideração também que a agência pode optar por um horário diferenciado aos sábados, se faz necessário o conceito de turnos de trabalho. A classe foi definida com os atributos: código,

descrição, hora de início, hora final, um booleano para marcar se determinado turno está ativo, data de criação e data de exclusão. Não se pode excluir um registro dessa tabela, pois ela está vinculada a vários registros do sistema, então se faz necessária uma exclusão lógica, em vez de física, e para isso existem os três últimos atributos citados.

Como pode ser visto na figura 17, a página apresenta campos de texto para inserção do código, descrição, hora inicial e hora final. Os demais atributos são inseridos de forma automática, sendo o booleano inserido como “verdadeiro”, e a data de criação como a data e hora do momento do cadastro. Nesse requisito também é realizada a validação do preenchimento dos campos para liberação do botão de cadastro, através de *Bacon.js*, de forma bastante parecida com os anteriores.

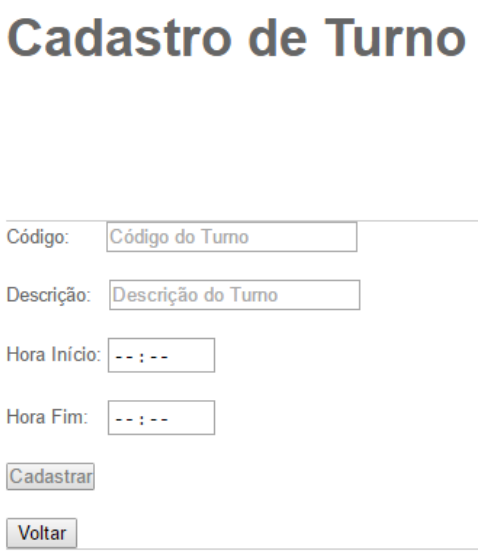

Figura 17 - Cadastro de turno, utilizando navegador Chrome.

5.5 Requisito: Registro de Depósito

Depósito se refere às quantias, em dinheiro ou cheque, que entram nos caixas dos terminais e que os operadores depositam, no decorrer do dia, no cofre da

agência. O valor da soma desses depósitos, junto das operações extra-terminais realizadas pelo operador naquele dia, é calculado, e o resultado é comparado com o valor de saldo final apresentado no relatório gerado pelo terminal lotérico ao final de cada dia. Sendo assim, a classe Depósito foi definida com os seguintes atributos: terminal, operador, data e hora do depósito, tipo (cheque ou dinheiro), valor, turno, data do registro e operador que efetuou o registro.

A página de registro de depósito, como demonstrado na figura 18, é composta por três caixas de seleção, para que o usuário escolha o terminal, operador e turno referentes ao depósito. Esses valores são capturados do banco de dados no momento em que a página é carregada, através de eventos *socket.io*. Também presentes na página estão campos para inserção da data, hora, tipo e valor do depósito.

Figura 18 - Página de registro de depósito, utilizando navegador Chrome.

De forma semelhante às páginas de cadastro, é realizada a validação de preenchimento dos campos, pois todos são necessários ao registrar um novo depósito. Um diferencial nessa página, porém, é a validação do conteúdo do campo valor, possibilitando apenas a inserção de números. Esse tratamento pode ser visto na figura 19, a seguir.

```
$("#valor_dep").keydown(function (e) {
    // backspace, delete, tab, esc, enter e "."
    if ($.inArray(e.keyCode, [8, 46, 9, 27, 13, 110, 190]) !== -1 ||
        (e.keyCode == 65 && e.ctrlKey === true) || // Ctrl+A
        (e.keyCode == 67 && e.ctrlKey === true) || // Ctrl+C
        (e.keyCode == 88 && e.ctrlKey === true) || // Ctrl+X
        (e.keyCode >= 35 && e.keyCode <= 39)) {    // home, end, esquerda, direita
        return; // não faz nada
    }
    // garante que não é um number e não deixa digitar
    if ((e.shiftKey || (e.keyCode < 48 || e.keyCode > 57)) && (e.keyCode < 96 || e.keyCode > 105)) {
        e.preventDefault();
    }
});
```

Figura 19 - Validação do campo valor.

No lado servidor, quando são recebidos os dados de registro de um novo depósito, é feita uma validação da data e hora recebidos, pois o navegador Mozilla Firefox não possui suporte aos tipos “*date*” e “*time*” das caixas de texto HTML, então nesse navegador é preciso que o usuário digite manualmente os valores, o que aumenta as chances de erro. Para tal validação foi utilizada a biblioteca “*moment.js*”, que contém uma variedade bastante grande de operações relacionadas a datas. Como podemos ver na figura 20, ao receber um novo evento “registroDeposito”, é checado se a data recebida é válida, tanto no formato que deve ser inserido no Firefox, quanto no formato gerado pelo “*datepicker*” dos navegadores Chrome, Edge, Internet Explorer e Opera. Caso a data esteja em um formato válido, ela é então convertida para o formato aceito pelo banco de dados MySQL. Em caso

contrário, é emitida uma mensagem de erro para o cliente. Também são validados os conteúdos das variáveis valor e tipo.

```
if ((moment(dataDeposito, "DD/MM/YYYY HH:mm", true).isValid() ||
    moment(dataDeposito, "YYYY-MM-DD HH:mm", true).isValid()) && (!(isNaN(valor))) &&
    (tipo.toUpperCase() === 'C' || tipo.toUpperCase() === 'D')) {
    var dataHora = moment(dataDeposito, ['DD/MM/YYYY HH:mm', 'YYYY-MM-DD HH:mm']).format('YYYY-MM-DD HH:mm:ss');
```

Figura 20 - Validação dos dados no servidor.

5.6 Requisito: Registro de Saldo

Ao final de cada dia, cada terminal emite um relatório contendo todas as operações – e seus respectivos valores – realizadas naquele terminal naquele dia. Também é apresentado nesse relatório o valor do saldo final do caixa, que se trata da quantia de dinheiro resultante de todas as operações, e que deve ser a quantia de dinheiro presente no caixa ao final do dia.

Para efetuar o registro desse valor no sistema foi definida a classe “RegSaldo”, que contém os atributos: terminal, operador, turno, valor do saldo apresentado no relatório, data do relatório, data do registro, código do operador que efetuou o registro e o valor do saldo final do terminal naquele dia, que é o resultado da diferença entre o somatório de todas as operações do terminal e o valor do saldo presente no relatório. Esse valor resultante é chamado também de “quebra de caixa”, que pode ser tanto positiva, no caso da quantia em caixa ser maior que a quantia mostrada no relatório, quanto negativa, no caso do caixa chegar ao fim do dia com menos dinheiro do que deveria.

A página criada para o registro do saldo, demonstrada na figura 21, contém três campos de seleção para que o usuário escolha o terminal, operador e turno correspondentes, bem como um campo para inserção do valor do saldo informado no relatório, um campo para inserir a data a que se refere o relatório, e um campo “somente leitura” que exibe o valor calculado da quebra de caixa.

Figura 21 - Página Registro de saldo, utilizando navegador Opera.

Nessa página, as validações são diferentes das demais, dado que o objetivo dessa tela não é somente cadastrar ou registrar alguma informação no banco, mas também calcular o valor da quebra de caixa. Aqui o sistema simula um comportamento de planilha, uma vez que existe uma fórmula vinculada ao campo "Quebra de caixa", e assim que os outros campos são preenchidos, é calculado esse valor e inserido na caixa de texto correspondente. Somente após o cálculo da quebra de caixa é que o botão de registro é liberado.

Para efetuar o cálculo da quebra de caixa, foram criadas cinco *properties* correspondendo aos cinco campos que precisam ser validados. As caixas de seleção de terminais, operadores e turnos tem seu preenchimento avaliado através da observação do evento "*onChange*" desses campos. Sempre que for alterada a

seleção do item desse campo, é alterado o valor da *property* correspondente para "*true*", caso o item contenha algum texto. Similarmente, o preenchimento do campo correspondente ao saldo também é avaliado pela observação do seu evento "*onChange*". Não foi utilizado o evento "*keyup*" para avaliação desse campo para evitar a utilização de valores incompletos no cálculo da quebra de caixa. No caso da validação do campo de data, é avaliado não só o preenchimento, mas também se o conteúdo do campo se trata de uma data válida. O código referente a essas validações pode ser visto na figura 22.

```
function valorCampoChange(campoDeTexto) {
    function valor() { return campoDeTexto.val() }
    return campoDeTexto.asEventStream("change").map(valor).toProperty(valor());
}
var saldoRelatorio = valorCampoChange(inputSaldo); //var inputSaldo = $("#saldoRelatorio");
var dataRegSaldo = valorCampoChange(inputData); //var inputData = $("#dataRelSaldo");
var P_Terminal = valorCampoChange(cboTerminais); //var cboTerminais = $('#listaTerminais');
var P_Operador = valorCampoChange(cboOperadores); //var cboOperadores = $('#listaOperadores');
var P_Turno = valorCampoChange(cboTurnos); //var cboTurnos = $('#listaTurnos');

function preenchido(x) { return x.length > 0 }
function dataValida(strData) {
    return (moment(strData, "DD/MM/YYYY", true).isValid() || moment(strData, "YYYY-MM-DD", true).isValid());
}
var saldoRelatorioPreenchido = saldoRelatorio.map(preenchido);
var terminalPreenchido = P_Terminal.map(preenchido);
var operadorPreenchido = P_Operador.map(preenchido);
var turnoPreenchido = P_Turno.map(preenchido);
var dataRelatorioPreenchido = dataRegSaldo.map(dataValida);
```

Figura 22 - Validação dos campos para cálculo da quebra de caixa.

Os valores dessas cinco *properties* são combinados para produzir uma nova, que a cada alteração dos campos observados recebe um evento booleano referente ao preenchimento dos campos. Essa nova *property* é então filtrada, e a cada evento "*true*" recebido é emitido um evento *socket.io* ao servidor para efetuar o cálculo da quebra de caixa, passando os valores dos campos como parâmetro, como demonstrado na figura 23.

```
var calculaQuebraCx = saldoRelatorioPreenchido.and(dataRelatorioPreenchido).and(terminalPreenchido)
    .and(operadorPreenchido).and(turnoPreenchido);

calculaQuebraCx.filter(function (x) { return x === true }).onValue(function () {
    var obj = {
        terminal: cboTerminais.val(),
        operador: cboOperadores.val(),
        turno: cboTurnos.val(),
        data: inputData.val(),
        saldo: inputSaldo.val()
    };
    socket.emit("calcularQuebraCx", obj);
});
```

Figura 23 - Requisição do cálculo da quebra de caixa.

Na figura 24 vemos o lado servidor, onde esse evento é capturado em um *stream* e são filtrados apenas os que tenham data e saldo válidos. A cada novo evento válido, é executada uma consulta no banco de dados buscando os valores de depósitos referentes ao terminal, operador, turno e data recebidos. A função de consulta retorna um *array* de objetos, em que cada objeto se refere a uma linha do resultado. É criado, então, um *stream* a partir do *array* de resultados da consulta, tornando cada item do *array* em um evento. Esse *stream* é mapeado para que cada evento corresponda ao seu atributo valor, e então é transformado em uma *property* contendo apenas um evento, que se trata do resultado da soma de todos os eventos do stream. Essa *property* contendo a soma de todos os depósitos é combinada com o *stream* que contém o valor do saldo, resultando na diferença entre os dois valores, ou seja, a quebra de caixa. Esse valor é emitido através de um evento *socket.io* para o cliente.

```
Bacon.fromEvent(socket, "calcularQuebraCx")
    .filter(function (dados) {
        return ((moment(dados.data, "DD/MM/YYYY", true).isValid() ||
            moment(dados.data, "YYYY-MM-DD", true).isValid()) &&
            (!(isNaN(dados.saldo)))) === true;
    }).onValue(function (dados) {

        var data = moment(dados.data, ['DD/MM/YYYY', 'YYYY-MM-DD']).format('YYYY-MM-DD');
        var saldo = Bacon.fromArray([dados.saldo]).map(function (s) { return -s; });

        var query = 'SELECT IFNULL(DP.VALOR,0) AS VALOR FROM TCC.DEPOSITO DP ' +
            ' JOIN TCC.TERMINAL TE ON DP.ID_TERMINAL = TE.ID_TERMINAL ' +
            ' JOIN TCC.OPERADOR OP ON DP.ID_OPERADOR = OP.ID_OPERADOR ' +
            ' JOIN TCC.TURNO TU ON DP.ID_TURNO = TU.ID_TURNO ' +
            ' WHERE TE.COD_TERMINAL = "' + dados.terminal + '" AND OP.COD_OPERADOR = "' + dados.operador +
            '" AND TU.COD_TURNO = "' + dados.turno + '" AND DATE(DP.DATA_DEPOSITO) = "' + data + '" ';

        connection.connect(query, function (rows) {
            var soma = function (a, b) { return a + b };
            var extraiValor = function (obj) { return obj.VALOR };
            var depositos = Bacon.fromArray(rows).map(extraiValor).fold(0, soma);
            var resultado = depositos.combine(saldo, soma).onValue(socket, "emit", "resultQuebraCx");
        });
    });
```

Figura 24 - Cálculo da quebra de caixa.

Como pode ser visto na figura 25, ao receber o valor da quebra de caixa no lado cliente, é preenchido o campo correspondente e liberado o botão registrar, que ao ser pressionado, salva um novo registro de saldo no banco de dados, utilizando os valores presentes nos campos da página.

```
Bacon.fromEvent(socket, "resultQuebraCx").onValue(function (valor) {
    //arredonda para duas casas decimais
    var v = Math.round(valor * 100) / 100;
    vlrQuebraCx.val(v);
    setAtivo(btnRegistro, true);
});

btnRegistro.click(function () {
    socket.emit("registroSaldo", cboTerminais.val(), cboOperadores.val(), cboTurnos.val(), inputSaldo.val(),
        inputData.val(), cboOperadores.val(), vlrQuebraCx.val());
    limpaInputs();
    setAtivo(btnRegistro, false);
});
```

Figura 25 - Recebimento do resultado e clique do botão.

6. CONCLUSÃO

Para utilizar programação funcional reativa, é exigida uma mudança do pensamento tradicional de programação imperativa e de orientação a objetos. Existe uma curva de aprendizado vinculada a essa mudança, que não se trata de algo tão trivial quanto possa ser avaliado de início. Porém, à medida que essa curva de aprendizado vai sendo superada, os benefícios do uso de FRP vão sendo percebidos, cada vez mais. Tal curva de aprendizado pôde ser observada no desenvolvimento deste trabalho, pois o autor não possuía qualquer conhecimento prévio em programação funcional reativa, nem em nenhuma das tecnologias descritas no item 4.3.

Ao utilizar *streams*, eventos, *properties* e as diversas funções existentes para manipulá-los, o código se torna mais legível, além de mais sucinto, necessitando de menos linhas de código para produzir a mesma funcionalidade. Especialmente ao se fazer uso do *Node.js*, por possuir uma arquitetura de entrada e saída de dados direcionada a eventos e assíncrona, o que combina muito bem com programação funcional reativa.

O objetivo do trabalho – produzir um protótipo de sistema web para gerenciamento de lotéricas utilizando FRP, a fim de simular as relações funcionais entre células de planilhas – foi atingido ao completar o requisito descrito no item 5.6, por se tratar de uma das funções mais importantes de um sistema desse tipo, pois o cálculo e visualização de quebras de caixa é uma atividade vital e diária nessas empresas. Para implementar os demais requisitos seria mera questão de tempo, uma vez que já foi compreendido o uso de streams e demais conceitos de FRP, bem como as outras tecnologias utilizadas no trabalho.

6.1 Trabalhos Futuros

Como sugestão de trabalhos futuros, pode ser destacada, principalmente, a implementação dos demais requisitos do sistema, sempre utilizando programação funcional reativa. Outra melhoria interessante ao protótipo seria na questão visual, melhorando a interface e a experiência do usuário, preferencialmente após uma análise de usabilidade de *software*.

Por não ser o foco deste trabalho, não foi implementada nenhuma forma de criptografia na comunicação dos dados entre cliente e servidor, e nem no armazenamento de senhas no banco de dados. Essas e outras questões de segurança poderiam também ser abordadas em trabalhos futuros.

REFERÊNCIAS BIBLIOGRÁFICAS

ACUNETIX. What Are Web Applications? **Web application security with Acunetix**, 2012. Disponivel em: <http://www.acunetix.com/websitesecurity/web-applications/>. Acesso em: Outubro 2015.

BONÉR, J. et al. The Reactive Manifesto. **The Reactive Manifesto**, 16 Setembro 2014. Disponivel em: <http://www.reactivemanifesto.org>. Acesso em: Novembro 2015.

CAIXA ECONÔMICA FEDERAL. Perguntas frequentes sobre Licitações de Unidades Lotéricas. **Caixa**. Disponivel em: <http://www.caixa.gov.br/compras-caixa/licitacoes-lotericas/perguntas-frequentes/Paginas/default.aspx#empresario-loterico>. Acesso em: Outubro 2015.

DOURASOFT. Funcionalidades | DouraSoft do Brasil. **DouraSoft - Programa AGIL para lotéricas**, 2015. Disponivel em: <https://dourasoft.com.br/programa-para-lotericas/>. Acesso em: Outubro 2015.

GEISENDÖRFER, F. mysql. **npm**, 2016. Disponivel em: <https://www.npmjs.com/package/mysql>. Acesso em: Junho 2016.

IBM DEVELOPERWORKS. O que exatamente é o Node.js? **iMasters - Desenvolvimento, Criação e Inovação**, 2011. Disponivel em: <http://imasters.com.br/artigo/22016/javascript/o-que-exatamente-e-o-nodejs/>. Acesso em: Maio 2016.

JONES, A.; BLACKHEATH, S. **Functional Reactive Programming**. New York: Manning, 2015.

LOTERIA EXATA. Loteria Exata. **Loteria Exata**, 2013. Disponivel em: <https://www.loteriaexata.com.br/site/home/>. Acesso em: Outubro 2015.

MEDEIROS, A. The introduction to Reactive Programming you've been missing. **GitHub Gist**, 2014. Disponivel em: <https://gist.github.com/staltz/868e7e9bc2a7b8c1f754>. Acesso em: Novembro 2015.

MEGAFLLEX. Megafllex - Sistema de gerenciamento de lotéricas e correspondentes. **Megafllex - Sistema de gerenciamento de lotéricas e correspondentes**, 2015. Disponivel em: <http://www.megafllex.com.br/software-megafllex>. Acesso em: Outubro 2015.

MIDIA DREAM SOLUÇÕES. SISLOT. **Midia Dream Soluções**, 2015. Disponivel em: <http://www.midiadream.com.br/sislot/>. Acesso em: Outubro 2015.

NILSSON, P. Implementing Snake in Bacon.js. **GitHub Pages**, 2013. Disponivel em: <http://philipnilsson.github.io/badness/>. Acesso em: Setembro 2016.

NODE.JS FOUNDATION. Node.js. **Node.js**, 2016a. Disponivel em: <https://nodejs.org/en/>. Acesso em: Maio 2016.

NODE.JS FOUNDATION. Express. **Express - Node.js web application framework**, 2016b. Disponivel em: <http://expressjs.com/>. Acesso em: Maio 2016.

PAANANEN, J. Tutorials. **Bacon.js - Functional Reactive Programming library for JavaScript**, 2012. Disponivel em: <http://baconjs.github.io/tutorials.html>. Acesso em: Setembro 2016.

PAANANEN, J. Juha Paananen (Reaktor) @ MLOC.JS. **Ustream**, 2013. Disponivel em: <http://www.ustream.tv/recorded/29299079>. Acesso em: Setembro 2016.

PAANANEN, J. FAQ - baconjs. **GitHub**, 2016a. Disponivel em: <https://github.com/baconjs/bacon.js/wiki/FAQ>. Acesso em: Setembro 2016.

PAANANEN, J. baconjs - FRP library for Javascript. **GitHub**, 2016b. Disponivel em: <https://github.com/baconjs/bacon.js#intro>. Acesso em: Setembro 2016.

PROLOTÉRICA. Software para Gestão Lotérica - Prolotérica. **Software para Gestão Lotérica - Prolotérica**, 2015. Disponivel em: <https://www.proloterica.com.br/>. Acesso em: Outubro 2015.

RIBEIRO JUNIOR, F. D. A. **Programação Orientada a Eventos no lado do servidor utilizando Node.js**. Ifactory Solutions. Fortaleza, p. 5. 2012.

SGCL. Gestão Lotérica. **SGCL-Sistemas Gestão Comercial Ltda**, 17 Maio 2013. Disponivel em: <http://www.sgcl.com.br/home/?page_id=172>. Acesso em: Outubro 2015.

SOCKET.IO. Socket.IO. **Socket.IO**, 2016. Disponivel em: <http://socket.io/>. Acesso em: Maio 2016.

STACK OVERFLOW. What is (functional) reactive programming? **Stack Overflow**, 23 Junho 2009. Disponivel em: <http://stackoverflow.com/questions/1028250/what-is-functional-reactive-programming>. Acesso em: Novembro 2015.

STOVELL, P. What is Reactive Programming? **Paul Stovell**, 2010. Disponivel em: <http://paulstovell.com/blog/reactive-programming>. Acesso em: Abril 2016.

UBL, M.; KITAMURA, E. Apresentando WebSockets: trazendo soquetes para a web. **HTML5 Rocks**, 2010. Disponivel em: <http://www.html5rocks.com/pt/tutorials/websockets/basics/>. Acesso em: Maio 2016.

WIKIPEDIA. Web Application. **Wikipedia**, 2015a. Disponivel em: <https://en.wikipedia.org/wiki/Web_application>. Acesso em: Outubro 2015.

WIKIPEDIA. Functional Programming. **Wikipedia**, 2015b. Disponivel em: <https://en.wikipedia.org/wiki/Functional_programming>. Acesso em: Novembro 2015.

WIKIPEDIA. Functional reactive programming. **Wikipedia**, 2015c. Disponivel em: <https://en.wikipedia.org/wiki/Functional_reactive_programming>. Acesso em: Novembro 2015.

WIKIPEDIA. WebSocket. **Wikipedia**, 2016. Disponivel em: <https://en.wikipedia.org/wiki/WebSocket>. Acesso em: Maio 2016.

APÊNDICE A – Artigo

Desenvolvimento de Protótipo de Sistema Web para Gerenciamento de Agência Lotérica Utilizando Programação Funcional Reativa

João Victor Mendes de Oliveira[1], Leandro José Komosinski[1]

[1]Dpt. Informática e Estatística – Universidade Federal de Santa Catarina (UFSC)

Postal 476 – 88040-900 – Florianópolis – SC – Brasil

jvmdeoliveira@gmail.com, leandro.komosinski@ufsc.br

Abstract. *Cash flow management tools are essential in the daily operations of lottery agencies. Many agencies still use just an electronic spreadsheet, or even only pen and paper, to manage daily cash operations. Taking that into account, this work proposes to develop a web application prototype which seeks to help lottery agencies managers in the control of the daily operations of their company, as well as automatically calculate the balance of its operating terminals at the end of each day. By using functional reactive programming, it aims to enjoy the qualities of this paradigm, as well as simulate the functional relations between the cells of an electronic spreadsheet.*

Resumo. *Ferramentas para gerenciamento de fluxo de caixa são essenciais no dia a dia de uma agência lotérica. Muitas agências ainda utilizam apenas planilhas eletrônicas, ou até mesmo papel e caneta, para gerenciar as operações de caixa diárias da empresa. Considerando isso, este trabalho se propõe a desenvolver um protótipo de aplicação web que visa auxiliar o gerente de agências lotéricas no controle das operações realizadas no dia a dia da sua empresa, bem como calcular automaticamente o saldo dos terminais operacionais ao fim de cada dia. Empregando o uso de programação funcional reativa, objetiva usufruir das qualidades desse paradigma, como também simular as relações funcionais entre as células de uma planilha eletrônica.*

1. Introdução

O gerente de uma casa lotérica precisa ter conhecimento sobre todas as operações realizadas no seu estabelecimento. Necessita, também, ter conhecimento sobre os valores de cada operação, quanto dinheiro entrou e saiu, bem como o saldo de cada terminal lotérico existente em sua agência.

Uma ferramenta amplamente utilizada para esse objetivo é a planilha eletrônica, cuja principal qualidade é a possibilidade de ser pré-programada com diversas funções que ditam relações entre as células.

O presente trabalho apresenta, de forma sucinta, o relatório de desenvolvimento de um protótipo de aplicação web, que, utilizando-se da Programação Funcional Reativa, implementa a principal qualidade apresentada pelas planilhas eletrônicas, visando alcançar todos os objetivos descritos acima.

1.1 Programação Funcional Reativa

Programação Reativa [1] é um paradigma de programação orientado a fluxos de dados assíncronos e a propagação de mudanças. Nesse paradigma, eventos formam um fluxo (*stream*) de eventos assíncronos. Um *stream* é uma sequência contínua de eventos ordenados no tempo, como ilustrado na figura 1.

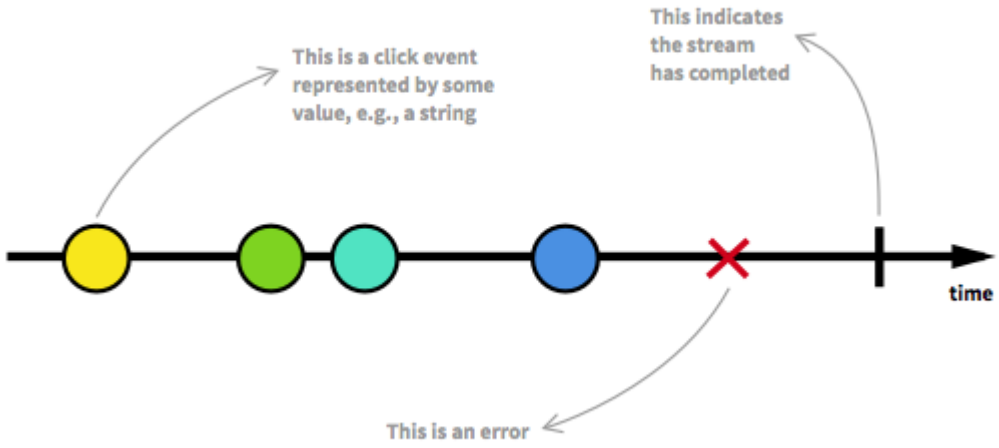

Figura 26 - Diagrama de marbles de um Stream de eventos.

Fonte: https://gist.github.com/staltz/868e7e9bc2a7b8c1f754

Programação Funcional [2] é outro paradigma de programação, que trata a computação como uma avaliação de funções matemáticas, enfatizando a aplicação de funções, ao contrário da programação imperativa, que enfatiza mudanças no estado do programa. É um paradigma declarativo, ou seja, a programação é feita com expressões ou declarações no lugar de instruções. Como demonstrado na figura 2, o paradigma funcional pode ser entendido como um mapeamento dos valores de entrada nos valores de retorno, através de funções.

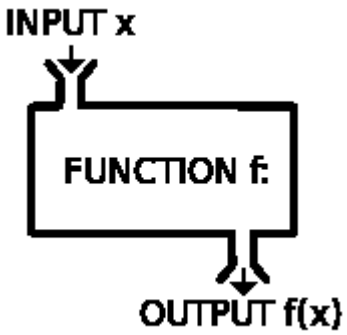

Figura 27 - Entrada -> Função -> Saída.

Fonte: https://zeroturnaround.com/rebellabs/there-is-more-to-code-quality-than-just-pretty-vs-ugly

Programação Funcional Reativa se trata de um paradigma de programação que engloba esses dois paradigmas. Portanto, trata-se de *streams* – ou fluxos – de dados

assíncronos que sofrem alterações ao longo do tempo, que podem ser mapeados, filtrados, reduzidos, compostos [3][4].

1.2 Bacon.js

Bacon.js [5] é uma biblioteca que possibilita a implementação de programação assíncrona com *streams* observáveis – conceitos de Programação Funcional Reativa – em *JavaScript*. Captura eventos em variáveis dos tipos "*Property*" – ou Propriedade – e "*EventStream*" – ou fluxos de eventos – que, por sua vez, podem ser mapeados, filtrados, fundidos, dentre várias outras operações funcionais. Essas variáveis são também chamadas de "*observables*", ou "observáveis", em português.

1.3 Node.js

Trata-se de um ambiente de execução *open-source* e multiplataforma que possibilita o desenvolvimento no lado do servidor de aplicações *web* na linguagem *JavaScript*, interpretando essa linguagem através do motor "V8" da Google [6].

Possui uma extensa coleção de módulos que auxiliam no desenvolvimento das aplicações. Foram utilizados no desenvolvimento deste trabalho os módulos: Express, um framework que simplifica ainda mais o desenvolvimento de aplicações web com Node [7]; Socket.io, que possibilita o uso do protocolo WebSocket para comunicação bidirecional em tempo real e baseada em eventos entre servidor e cliente [8]; mysql, um driver para fazer a conexão com o banco de dados MySQL [9].

2. Solução Proposta

Como toda aplicação web, a solução proposta é composta de uma interface com o usuário, chamada de "cliente", e outra parte que consiste da lógica da aplicação e do armazenamento dos dados, chamado de "servidor".

A comunicação entre cliente e servidor é feita através do protocolo *WebSocket*, implementado no presente trabalho utilizando o *framework Socket.io*. O cliente foi implementado utilizando HTML, CSS e *JavaScript*, com o auxílio das bibliotecas *jQuery* e *Bacon.js*. A parte de servidor foi desenvolvida utilizando também *JavaScript*, através do *Node.js*, facilitada pelo *framework Express*. No servidor também foi utilizado *Bacon.js* para implementar o paradigma de Programação Funcional Reativa. Para conexão com o banco de dados foi utilizado o módulo "mysql" do *Node.js*.

A figura 6 ilustra a arquitetura do sistema proposto.

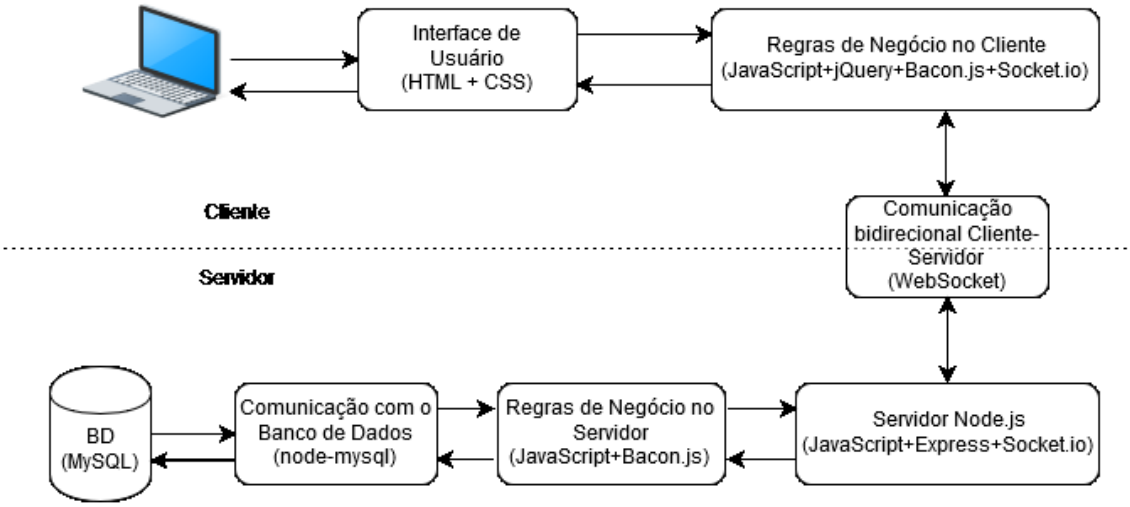

Figura 28 - Arquitetura da solução proposta.

3. Desenvolvimento

Após análise dos requisitos, foram definidas as classes do sistema, modelado o banco de dados, e decididos os requisitos mínimos para que o protótipo tivesse as funcionalidades mínimas necessárias para simular a funcionalidade de planilha utilizando programação funcional reativa. Esses requisitos são: Página de Login, Cadastro de Terminal, Cadastro de Operador, Cadastro de Turno, Registro de Depósito e Registro de Saldo.

Depósito se refere às quantias, em dinheiro ou cheque, que entram nos caixas dos terminais e que os operadores depositam, no decorrer do dia, no cofre da agência. Ao final de cada dia, cada terminal emite um relatório contendo todas as operações – e seus respectivos valores – realizadas naquele terminal naquele dia. Também é apresentado nesse relatório o valor do saldo final do caixa, que se trata da quantia de dinheiro resultante de todas as operações, e que deve ser a quantia de dinheiro presente no caixa ao final do dia – o somatório de todos os depósitos dos operadores.

Para efetuar o registro desse valor no sistema foi definida a classe “RegSaldo”, que contém os atributos: terminal, operador, turno, valor do saldo apresentado no relatório, data do relatório, data do registro, código do operador que efetuou o registro e o valor do saldo final do terminal naquele dia, que é o resultado da diferença entre o somatório de todas as operações do terminal e o valor do saldo presente no relatório. Esse valor resultante é chamado também de “quebra de caixa”, que pode ser tanto positiva, no caso da quantia em caixa ser maior que a quantia mostrada no relatório, quanto negativa, no caso do caixa chegar ao fim do dia com menos dinheiro do que deveria.

A página criada para o registro do saldo, demonstrada na figura 4, contém três campos de seleção para que o usuário escolha o terminal, operador e turno correspondentes, bem como um campo para inserção do valor do saldo informado no relatório, um campo para inserir a data a que se refere o relatório, e um campo “somente leitura” que exibe o valor calculado da quebra de caixa.

Figura 29 - Página de Registro de saldo.

Nessa página o sistema simula um comportamento de planilha, uma vez que existe uma fórmula vinculada ao campo "Quebra de caixa", e assim que os outros campos são preenchidos, é calculado esse valor e inserido na caixa de texto correspondente. Somente após o cálculo da quebra de caixa é que o botão de registro é liberado.

Para efetuar o cálculo da quebra de caixa, foram criadas cinco *properties* correspondendo aos cinco campos que precisam ser validados. As caixas de seleção de terminais, operadores e turnos tem seu preenchimento avaliado através da observação do evento "*onChange*" desses campos. Sempre que for alterada a seleção do item desse campo, é alterado o valor da *property* correspondente para "*true*", caso o item contenha algum texto. Similarmente, o preenchimento do campo correspondente ao saldo também é avaliado pela observação do seu evento "*onChange*". O código referente a essas validações pode ser visto na figura 5.

```
function valorCampoChange(campoDeTexto) {
    function valor() { return campoDeTexto.val() }
    return campoDeTexto.asEventStream("change").map(valor).toProperty(valor());
}
var saldoRelatorio = valorCampoChange(inputSaldo); //var inputSaldo = $("#saldoRelatorio");
var dataRegSaldo = valorCampoChange(inputData); //var inputData = $("#dataRelSaldo");
var P_Terminal = valorCampoChange(cboTerminais); //var cboTerminais = $('#listaTerminais');
var P_Operador = valorCampoChange(cboOperadores); //var cboOperadores = $('#listaOperadores');
var P_Turno = valorCampoChange(cboTurnos); //var cboTurnos = $('#listaTurnos');

function preenchido(x) { return x.length > 0 }
function dataValida(strData) {
    return (moment(strData, "DD/MM/YYYY", true).isValid() || moment(strData, "YYYY-MM-DD", true).isValid());
}
var saldoRelatorioPreenchido = saldoRelatorio.map(preenchido);
var terminalPreenchido = P_Terminal.map(preenchido);
var operadorPreenchido = P_Operador.map(preenchido);
var turnoPreenchido = P_Turno.map(preenchido);
var dataRelatorioPreenchido = dataRegSaldo.map(dataValida);
```

Figura 30 - Validação dos campos para cálculo da quebra de caixa.

Os valores dessas cinco properties são combinados para produzir uma nova, que a cada alteração dos campos observados recebe um evento booleano referente ao preenchimento dos campos. Essa nova property é então filtrada, e a cada evento “true” recebido é emitido um evento socket.io ao servidor para efetuar o cálculo da quebra de caixa, passando os valores dos campos como parâmetro, como demonstrado na figura 6.

```
var calculaQuebraCx = saldoRelatorioPreenchido.and(dataRelatorioPreenchido).and(terminalPreenchido)
    .and(operadorPreenchido).and(turnoPreenchido);

calculaQuebraCx.filter(function (x) { return x === true }).onValue(function () {
    var obj = {
        terminal: cboTerminais.val(),
        operador: cboOperadores.val(),
        turno: cboTurnos.val(),
        data: inputData.val(),
        saldo: inputSaldo.val()
    };
    socket.emit("calcularQuebraCx", obj);
});
```

Figura 31 - Requisição do cálculo da quebra de caixa.

Na figura 7 vemos o lado servidor, onde esse evento é capturado em um stream e são filtrados apenas os que tenham data e saldo válidos. A cada novo evento válido, é executada uma consulta no banco de dados buscando os valores de depósitos referentes ao terminal, operador, turno e data recebidos. A função de consulta retorna um array de objetos, em que cada objeto se refere a uma linha do resultado. É criado, então, um stream a partir do array de resultados da consulta, tornando cada item do array em um evento. Esse stream é mapeado para que cada evento corresponda ao seu atributo valor, e então é transformado em uma property contendo apenas um evento, que se trata do resultado da soma de todos os eventos do stream. Essa property contendo a soma de todos os depósitos é combinada com o stream que contém o valor do saldo, resultando na diferença entre os dois valores, ou seja, a quebra de caixa. Esse valor é emitido através de um evento socket.io para o cliente.

```
Bacon.fromEvent(socket, "calcularQuebraCx")
    .filter(function (dados) {
        return ((moment(dados.data, "DD/MM/YYYY", true).isValid() ||
            moment(dados.data, "YYYY-MM-DD", true).isValid()) &&
            (!(isNaN(dados.saldo)))) === true;
    }).onValue(function (dados) {

        var data = moment(dados.data, ['DD/MM/YYYY', 'YYYY-MM-DD']).format('YYYY-MM-DD');
        var saldo = Bacon.fromArray([dados.saldo]).map(function (s) { return -s; });

        var query = 'SELECT IFNULL(DP.VALOR,0) AS VALOR FROM TCC.DEPOSITO DP ' +
            ' JOIN TCC.TERMINAL TE ON DP.ID_TERMINAL = TE.ID_TERMINAL ' +
            ' JOIN TCC.OPERADOR OP ON DP.ID_OPERADOR = OP.ID_OPERADOR ' +
            ' JOIN TCC.TURNO TU ON DP.ID_TURNO = TU.ID_TURNO ' +
            ' WHERE TE.COD_TERMINAL = "' + dados.terminal + '" AND OP.COD_OPERADOR = "' + dados.operador +
            '" AND TU.COD_TURNO = "' + dados.turno + '" AND DATE(DP.DATA_DEPOSITO) = "' + data + '" ';

        connection.connect(query, function (rows) {
            var soma = function (a, b) { return a + b };
            var extraiValor = function (obj) { return obj.VALOR };
            var depositos = Bacon.fromArray(rows).map(extraiValor).fold(0, soma);
            var resultado = depositos.combine(saldo, soma).onValue(socket, "emit", "resultQuebraCx");
        });
    });
```

Figura 32 - Cálculo da quebra de caixa.

Como pode ser visto na figura 8, ao receber o valor da quebra de caixa no lado cliente, é preenchido o campo correspondente e liberado o botão registrar, que ao ser pressionado, salva um novo registro de saldo no banco de dados, utilizando os valores presentes nos campos da página.

```
Bacon.fromEvent(socket, "resultQuebraCx").onValue(function (valor) {
    //arredonda para duas casas decimais
    var v = Math.round(valor * 100) / 100;
    vlrQuebraCx.val(v);
    setAtivo(btnRegistro, true);
});

btnRegistro.click(function () {
    socket.emit("registroSaldo", cboTerminais.val(), cboOperadores.val(), cboTurnos.val(), inputSaldo.val(),
        inputData.val(), cboOperadores.val(), vlrQuebraCx.val());
    limpaInputs();
    setAtivo(btnRegistro, false);
});
```

Figura 33 - Recebimento do resultado e clique do botão.

4. Conclusão

Para utilizar programação funcional reativa, é exigida uma mudança do pensamento tradicional de programação imperativa e de orientação a objetos. Existe uma curva de aprendizado vinculada a essa mudança, que não se trata de algo tão trivial quanto possa ser avaliado de início. Porém, à medida que essa curva de aprendizado vai sendo superada, os benefícios do uso de FRP vão sendo percebidos, cada vez mais.

Ao utilizar streams, eventos, properties e as diversas funções existentes para manipulá-los, o código se torna mais legível, além de mais sucinto, necessitando de menos linhas de código para produzir a mesma funcionalidade. Especialmente ao se fazer uso do Node.js, por

possuir uma arquitetura de entrada e saída de dados direcionada a eventos e assíncrona, o que combina muito bem com programação funcional reativa.

Referências

[1] MEDEIROS, A. The introduction to Reactive Programming you've been missing. **GitHub Gist**, 2014. Disponivel em: <https://gist.github.com/staltz/868e7e9bc2a7b8c1f754>. Acesso em: Novembro 2015.

[2] WIKIPEDIA. Functional Programming. **Wikipedia**, 2015a. Disponivel em: <https://en.wikipedia.org/wiki/Functional_programming>. Acesso em: Novembro 2015

[3] JONES, A.; BLACKHEATH, S. **Functional Reactive Programming**. New York: Manning, 2015.

[4] WIKIPEDIA. Functional reactive programming. **Wikipedia**, 2015b. Disponivel em: <https://en.wikipedia.org/wiki/Functional_reactive_programming>. Acesso em: Novembro 2015.

[5] NILSSON, P. Implementing Snake in Bacon.js. **GitHub Pages**, 2013. Disponivel em: <http://philipnilsson.github.io/badness/>. Acesso em: Setembro 2016.

[6] NODE.JS FOUNDATION. Node.js. **Node.js**, 2016a. Disponivel em: <https://nodejs.org/en/>. Acesso em: Maio 2016.

[7] NODE.JS FOUNDATION. Express. **Express - Node.js web application framework**, 2016b. Disponivel em: <http://expressjs.com/>. Acesso em: Maio 2016.

[8] SOCKET.IO. Socket.IO. **Socket.IO**, 2016. Disponivel em: <http://socket.io/>. Acesso em: Maio 2016.

[9] GEISENDÖRFER, F. mysql. **npm**, 2016. Disponivel em: <https://www.npmjs.com/package/mysql>. Acesso em: Junho 2016.