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

VALCIR BALBINOTTI JUNIOR

DESENVOLVIMENTO DE UM SISTEMA WEB PERSONALIZADO PARA CONTROLE FINANCEIRO

MONOGRAFIA DE ESPECIALIZAÇÃO

PATO BRANCO
2020

VALCIR BALBINOTTI JUNIOR

DESENVOLVIMENTO DE UM SISTEMA WEB PERSONALIZADO PARA CONTROLE FINANCEIRO

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
2020

Ministério da Educação
Universidade Tecnológica Federal do Paraná
Câmpus Pato Branco
Departamento Acadêmico de Informática
Curso de Especialização em Tecnologia Java

TERMO DE APROVAÇÃO

DESENVOLVIMENTO DE UM SISTEMA WEB PERSONALIZADO PARA CONTROLE FINANCEIRO

por

VALCIR BALBINOTTI JUNIOR

Este trabalho de conclusão de curso foi apresentado em 09 de março de 2020, 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), Andreia Scariot Beulke e Beatriz Terezinha Borsoi, membros de banca. Em seguida foi realizada a deliberação pela banca examinadora que considerou o trabalho aprovado.

_____
Vinicius Pegorini
Prof. Orientador (UTFPR)

_____
Andreia Scariot Beulke
(UTFPR)

_____
Beatriz Terezinha Borsoi
(UTFPR)

_____
Vinicius Pegorini
Coordenador do curso

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

RESUMO

Sistemas computacionais auxiliam todos os tipos de processos rotineiros, porém, como cada empresa possui sua própria rotina que faz parte da cultura organizacional, surgem necessidades de adaptação de software existente ou a contratação para desenvolvimento de um aplicativo específico para atender suas necessidades e interesses. O uso de *frameworks* como Angular, Angular Material e NodeJS auxiliam no desenvolvimento de um aplicativo, dando mais espaço para a fase de conexão com o cliente, que envolve o levantamento de requisitos, o acompanhamento do desenvolvimento, a realização de testes a etapa inicial quando o software é colocado em produção na empresa. Neste contexto, neste trabalho é apresentado o desenvolvimento de um sistema web para controle financeiro que foi desenvolvido para atender necessidades de determinado estabelecimento. O NodeJs foi utilizado para desenvolvimento do servidor e Angular e Angular Material para o desenvolvimento do cliente. Como banco de dados foi utilizado o MongoDB.

Palavras-chave: Angular. JavaScript. MongoDB. *Dynamic Systems Development Method*

ABSTRACT

Computer systems can assist all types of daily processes, however, as each company has an own routine, they arise to adapt an existing software or to hire a specific application for your needs. The use of frameworks such as Angular, Angular Material and NodeJS help to optimize the development phase of an application, giving more space to the connection phase with the customer. In this context, this work presents the development of a web system for money control of a given establishment. It was used NodeJs for server development, and Angular and Angular Material for the development of the client, MongoDB was chosen as the database.

Keywords: Angular. JavaScript. MongoDB. Dynamic Systems Development Method.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE SIGLAS

DSDM	*Dynamic Systems Development Method*
GUI	*Graphical User Interface*
HTML	*Hypertext Markup Language*
JSON	*JavaScript Object Notation*
NPM	*Node Package Manager*
RF	Requisito Funcional
RNF	Requisito Não Funcional*SCSS* *Sassy CSS*
URL	*Uniform Resource Locator* VSCode Visual Studio Code

SUMÁRIO

1 INTRODUÇÃO

Tendo em vista que a personalização é algo cada vez mais necessária no desenvolvimento de sistemas empresariais, a técnica de estudo de caso vem ganhado relevância, quando a finalidade é criar uma ferramenta que atenda o almejado pelo cliente, não necessitando, assim, transformação de ambiente ou adaptação na organização, nos processos e procedimentos e/ou na cultura organizacional que está adquirindo um sistema.

É comum que os sistemas ofertados no mercado, também conhecidos como software de prateleira, oferecem muitas funcionalidades, das quais não são necessárias para um grande número de empresas para as quais o software se destina, mas faltam detalhes que os usuários desejam e não encontram nesse tipo de produto. O que acontece muitas vezes, é, por exemplo, uma loja de roupas buscar por um sistema para o cadastro de produtos e o controle de suas vendas com bonificações direcionadas aos vendedores e encontrar um sistema que atenda o almejado, mas a obrigue a mudar aspectos rotineiros da sua atividade, gerando desconforto, dificuldade de adaptação e tempo de aprendizado em decorrência das adaptações necessárias.

Mediante esse cenário, conhecimentos devem ser aplicados objetivamente, como o uso de *frameworks*, padrões de projetos e metodologias de desenvolvimento para manter a vitaliciedade de uma empresa de desenvolvimento de softwares no mercado atual. E, principalmente, que atenda os interesses dos clientes, produzindo produtos dentro de custo e prazos planejados e com a qualidade almejada pelos usuários.

Logo, parte-se do princípio de que o entendimento das metodologias de desenvolvimento de software é primordial. Uma das metodologias que vem se destacando são as denominadas ágeis que surgiram em 2001. Segundo Pádua Filho (2019), os quatro pilares do desenvolvimento ágil são: interações entre indivíduos, documentação abrangente, colaboração do cliente além da negociação e respostas rápidas dentro de um planejamento estruturado.

Considerando os valores das metodologias ágeis e, que o desenvolvimento deste projeto foi realizado para uma empresa específica, o *Dynamic Systems Development Method* (DSDM) foi utilizado. O DSDM enfatiza o envolvimento constante dos usuários destinatários da solução. Além disso, foram usadas ferramentas de rápido desenvolvimento, como NodeJS, Angular e Angular Material.

O destinatário em questão é um grupo de estabelecimentos que fazem o fechamento de caixa de forma separada dos sistemas de controle de vendas, buscando uma segunda fonte de organização e conferência. Foi desenvolvido um aplicativo web para realizar o gerenciamento

de acordo com cada especificidade do processo, enfatizando tanto a parte estrutural do negócio quanto a de desenvolvimento do software em si.

1.2 OBJETIVOS

Os objetivos deste trabalho visam auxiliar na realização dos processos de trabalho pré-existentes.

1.2.1 Objetivo Geral

Desenvolver um sistema web de controle do fluxo de caixa de um estabelecimento comercial.

1.2.2 Objetivos Específicos

- Ter a possibilidade do cadastro de múltiplas contas monetárias.
- Implementar a interface do sistema atendendo padrões de usabilidade, visando reduzir o número de telas e de operações realizadas pelo usuário e tornar o uso do sistema mais intuitivo.
- Permitir a emissão de relatório com os dados financeiros e filtros.

1.3 JUSTIFICATIVA

Diante da necessidade de ter um controle financeiro para acompanhar a saúde fiscal de uma empresa, é utilizado as mais diversas maneiras para controlar as entradas e saídas das contas. Algumas empresas ainda utilizam processos manuais de controle, pois muitas vezes, os sistemas são muito complexos para o porte da empresa, os seus processos de negócio ou os procedimentos adotados pelas pessoas e que fazem parte da cultura organizacional.

Como forma de realizar o levantamento dos requisitos o mais fidedigno possível da realidade da empresa, durante alguns dias foi realizado o acompanhamento do processo manual de fechamento de caixa da empresa para a qual o software foi desenvolvido para identificar seu real modo de operação. E, assim, criar um sistema que torne a realização das atividades de gestão financeira mais práticas e rápidas, com funcionalidades extras para auxiliar a busca por informações a partir dos dados armazenados.

O desenvolvimento foi realizado aplicando formas estudadas durante o Curso de Especialização em Tecnologia Java, com um estudo estendido e maior imersão no *framework* Angular e o construtor NodeJS. O banco de dados escolhido foi o MongoDB, com seu gerenciamento feito pelo Compass.

1.4 ESTRUTURA DO TRABALHO

Este texto está organizado em capítulos. O Capítulo 2 apresenta os materiais e o método utilizados para o desenvolvimento do sistema. No Capítulo 3 estão os resultados da realização do trabalho, a apresentação do projeto desenvolvido por meio de suas telas e trechos do código fonte. Ao final, no Capítulo 4, está a conclusão do projeto, seguida das referências utilizadas na composição do texto.

2 FERRAMENTAS, TECNOLOGIAS E PROCEDIMENTOS

Para o desenvolvimento do software, obtido como resultado da realização deste trabalho, foram utilizadas ferramentas específicas em cada etapa. Para a modelagem do sistema e desenho do banco de dados foi utilizado o Adobe Illustrator, que é insuficiente para a disposição de padrões definidos pelos desenvolvedores em geral, mas permite maior personalização durante a criação dos mapas.

Para o desenvolvimento foi utilizado o Visual Studio Code, na versão 1.42.1, pois permite o uso de uma vasta biblioteca de extensões, facilitando a programação em todas as linguagens.

Como *builder* foi utilizado o NodeJS, servindo a programação do servidor. O sistema foi desenvolvido com o *framework* Angular e a estilização com o Angular Material. O banco de dados utilizado foi o MongoDB. Para gerenciamento do banco de dados foi utilizado o Mongo Compass. E para melhor execução das linhas de comando e organização de execução de processos, foi utilizado o ConEmu.

2.1 FERRAMENTAS E TECNOLOGIAS

O Quadro 1 apresenta as ferramentas e as tecnologias utilizadas na modelagem e no desenvolvimento do aplicativo.

Quadro 5 – Ferramentas e tecnologias

Ferramenta / Tecnologia	Versão	Disponível em	Aplicação
Adobe Illustrator	CC	https://www.adobe.com/sea/	Modelagem do sistema
ConEmu	191012	https://conemu.github.io/	Emulador de terminal
TypeScript	3.7	https://www.typescriptlang.org	Superconjunto de JavaScript para desenvolvimento da aplicação
HTML	5		Linguagem de marcação de textos para desenvolvimento da interface da aplicação
CSS	3		Mecanismo para adicionar estilo a um documento web
JavaScript	1.8.5	https://www.javascript.com/	JavaScript é uma linguagem de programação interpretada estruturada, de *script* em alto nível com tipagem dinâmica fraca e multi-paradigma.
NodeJS	12.16.1	https://nodejs.org/en/	*Runtime built* criado no mecanismo JavaScript V8 do Chrome.

Angular	9.0.2	https://angular.io/	Plataforma de aplicações web de código-fonte aberto e front-end baseado em TypeScript
Angular Material	9.0.1	https://material.angular.io/	Infraestrutura de componentes de interface do usuário e componentes de design de materiais para aplicativos.
MongoDB	4.0.0	https://www.mongodb.com/	MongoDB é um software de banco de dados orientado a documentos
MongoCompass	4.5	https://www.mongodb.com/products/compass	GUI para MongoDB

Fonte: autoria própria

Adobe Illustrator é um editor de imagens vetoriais desenvolvido e comercializado pela Adobe Systems. Inicialmente desenvolvido apenas para o sistema operacional Windows, permite criação de vetores com alta capacidade de edição e personalização, recursos importantes para a para criação da modelagem do sistema.

ConEmu é um emulador de terminal com guias de código aberto e gratuito para Windows. O ConEmu apresenta vários consoles e aplicativos simples da *Graphical User Interface* (GUI) como uma janela personalizável da GUI com guias e uma barra de *status*. ConEmu foi utilizado porque as execuções dos códigos são feitas em terminal.

TypeScript é um superconjunto de JavaScript desenvolvido pela Microsoft que adiciona tipagem e alguns outros recursos à linguagem. Necessário para funcionamento do Anuglar.

Node.js é um interpretador de JavaScript assíncrono com código aberto orientado a eventos, focado em migrar a programação do JavaScript do cliente para os servidores.

Angular é uma plataforma de aplicações *web* de código-fonte aberto e *front-end* baseado em TypeScript. Que, além de gama de *tags* que auxiliam a construção da página web, possuí vínculo com o Angular Material, que é uma implementação do Material Design no Angular e seus componentes.

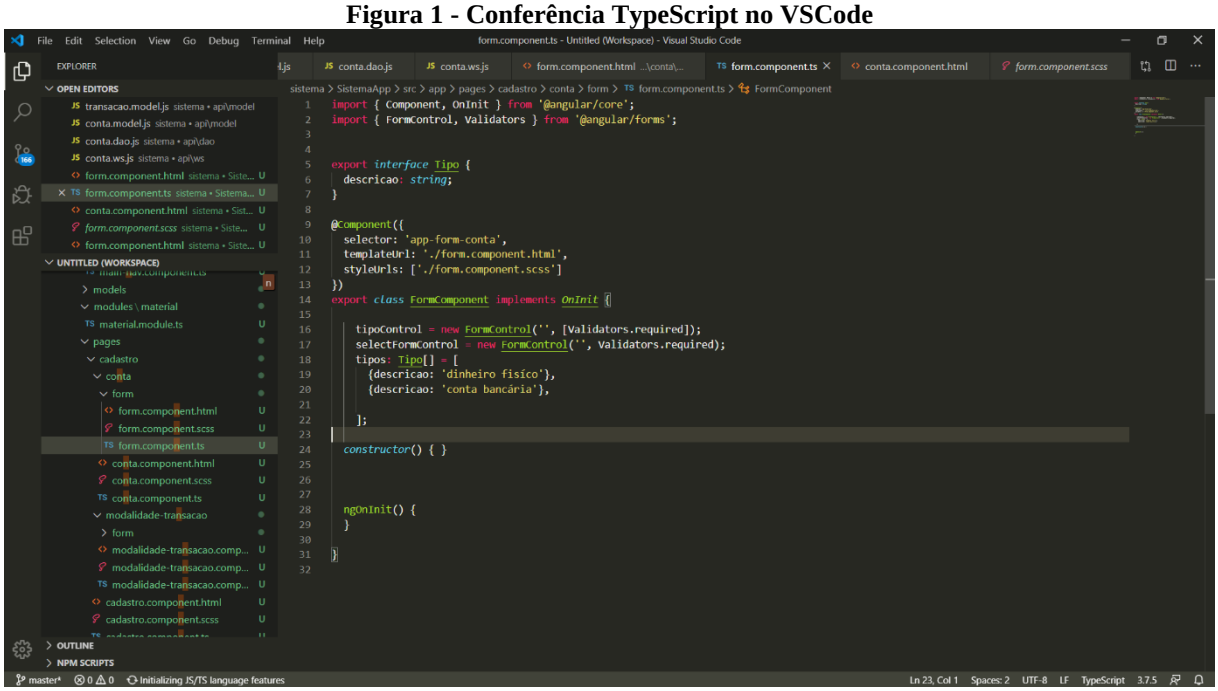
MongoDB é um software de banco de dados orientado a documentos livre, de código aberto e multiplataforma, escrito na linguagem C++. Classificado como um programa de banco de dados NoSQL, o MongoDB usa documentos semelhantes a *JavaScript Object Notation* (JSON) com esquemas. E o MongoDB Compass é uma interface gráfica, que simplifica muito o trabalho com o MongoDB.

2.2 PROCEDIMENTOS TÉCNICOS

Para preparação do ambiente de trabalho, primeiro é necessário instalar o NodeJS, efetuando o download do instalador na sua página principal. Importante ressaltar que o NodeJS traz consigo o *Node Package Manager* (NPM), que será muito útil para instalação e acompanhamento das outras ferramentas.

Na sequência vem a instalação do ConEmu e do Visual Studio Code (VSCode) que é feita de forma simples, sem nenhuma especificidade. Com o VSCode é instalado o TypeScript, que pode ser conferido na barra inferior do próprio VSCode em uma página TypeScript. Na Figura 1 é apresentada uma tela do VSCode.

Figura 1 - Conferência TypeScript no VSCode



Fonte: Autoria própria

Caso não seja apresentado TypeScript na barra inferior é necessário fazer o *download* na página do TypeScript e instalar como o passo-a-passo da própria página, ou, pelo próprio NPM, digitando nas linhas do terminal “npm install -g typescript”.

A instalação do Angular deve ser feita pelo NPM, executando nas linhas de código “npm install -g @angular/cli” e posteriormente, dentro da página do projeto, executar o comando “ng add @angular/material” e concordar com a instalação de todos os seus auxiliares.

Para a instalação do banco de dados, deve-se acessar a página da MongoDB e efetuar o *download* do executável. Durante os passos da instalação ele oferece o Mongo Compass, que deve ser marcado a *check box* respectiva para efetuar a instalação.

3 RESULTADOS

Este capítulo apresenta o resultado da realização deste trabalho que é uma ferramenta para controle financeiro de um grupo de restaurantes. Dentre suas funcionalidades estão a de fazer o fechamento de caixa diário, que será realizada pelo usuário de acordo com os resultados obtidos no dia, separando as entradas e saídas por múltiplas contas. Isso porque, além da conta bancária de cada empresa, mais de um administrador, produzindo relatórios específicos.

3.1 ESCOPO DO SISTEMA

O sistema desenvolvido possui interface simples que atende requisitos de usabilidade visando facilitar o acesso e o entendimento pelo usuário que realizará as operações no caixa do dia e para os administradores que irão consultar as informações cadastradas. O fechamento do caixa consiste em adicionar os valores movimentados pelo funcionário da empresa durante o dia, tanto as entradas quanto as saídas, mantendo os valores em cada caixa e conterá, ainda:

a) descrição - um título curto para identificação;
b) modalidade - cadastrada previamente, usada posteriormente para gerar os relatórios;
c) conta - utilizada nos lançamentos, também previamente cadastrada;
d) valor monetário - a quantia que foi movimentada;
e) tipo - se é valor de entrada ou de saída;
f) data - o dia da transação, podendo ser registrados lançamentos futuros ou passados.

A página inicial apresentará as transações do dia, com um calendário para alteração. Os registros serão feitos para a data que está em apresentação, exclusivamente.

Deverá haver um cadastro de tipo de transação para produzir relatórios posteriores. Por exemplo, pode-se criar uma modalidade chamada “Débito”, assim, todas as vendas realizadas em débitos serão dadas como entrada em seu tipo. Ainda poderão ser editados e excluídos, sem necessidade de incluir justificativa.

Além disso, a hierarquização de usuários deverá ser realizada levando em consideração o cadastro de modalidades, contas e visualização de relatórios. Os relatórios apresentarão:

- As transações monetárias, por período, filtradas pela sua modalidade (ex.: vendas, pagamentos e vales salário);
- As transações monetárias, por período, filtradas pelo usuário que a registrou;
- O saldo relativo a cada modalidade de transação cadastrada.

Quanto à usabilidade, haverá botões de fácil visualização para alteração entre entradas e saídas, sem a necessidade de *pop-ups*. O método de inclusão de transação será realizado visando celeridade, priorizando o foco da ideia principal do sistema que é a visualização das quantias em cada conta.

Haverá, ainda, um cadastro de contas, que serão separadas por tipo previamente cadastrado, sendo dinheiro em espécie ou conta bancária. Por fim, uma última tela que mostrará o *status* atual de cada conta e respectiva soma.

3.2 MODELAGEM DO SISTEMA

O sistema possui os seguintes perfis: administrador e funcionário caixa. O administrador tem acesso a todas as funcionalidades do sistema. O funcionário caixa tem acesso apenas aos lançamentos diários, sem visualização dos relatórios e possibilidade de cadastros de modalidade ou contas.

O Quadro 2 apresenta os Requisitos Funcionais (RF) para o sistema considerando os dois perfis apresentados. Ressalta-se que os cadastros básicos não fazem parte do escopo da solução proposta por este trabalho.

Quadro 6 – Requisitos funcionais

Identificação	Nome	Descrição
RF01	Manter modalidades de lançamento	Administrador deverá efetuar o cadastro das modalidades referentes e necessárias ao lançamento das transações. Ex.: modalidade “vale salário”, para que no momento do lançamento seja seguida a lógica dos acontecimentos cotidianos da empresa.
RF02	Manter contas	Administrador deverá cadastrar contas que serão os depósitos de dinheiro, não apenas as contas bancárias, mas também as carteiras pessoais dos administradores e as reservas de caixa para os funcionários.
RF03	Lançar transações	Poderá ser feita pelas duas modalidades de usuário, sendo informados todos os campos requisitados no escopo do sistema. Deverá ser uma tela intuitiva, que busque praticidade no processo. Sendo a tela inicial após efetivada a autenticação do usuário, apresentado a data atual para os cadastros.
RF04	Relatórios para administradores	Relatório dos lançamentos por dia, dos lançamentos por tipo, dos lançamentos por modalidades e dos lançamentos por usuário.

RF05	Manter usuários	Manter usuários e seus perfis de acesso ao sistema será tarefa incumbida ao administrador, que terá seu cadastro feito no desenvolvimento do sistema.

Fonte: autoria própria

Os Requisitos Não Funcionais (RNF) definidos para o sistema são apresentados no Quadro 3.

Quadro 7 – Requisitos não funcionais

Identificação	Nome	Descrição
RNF01	Impressão de lançamentos	Deverá ser apresentada uma modalidade de impressão que traga apenas a tabela com os lançamentos de determinada data.
RNF02	Propriedades da tela de lançamentos	Não deverá ser feita como uma modalidade de cadastro, mas sim de forma funcional sem *pop-ups*, com aplicabilidade de estudos de usabilidade, favorecendo a celeridade.
RNF03	Impressão dos relatórios	Deverá ser apresentada uma modalidade de impressão que traga os relatórios de forma tabelada.

Fonte: autoria própria

Relatórios:

a) Lançamentos por dia. No formato:

Data: item

Descrição	Modalidade	Tipo/Valor
item	item	(+ ou -)/R$item

b) Lançamentos por modalidade:

Modalidade	Valor Total
item	R$item

c) Lançamentos por tipo:

Valores por /"item do tipo"/

item
item
soma

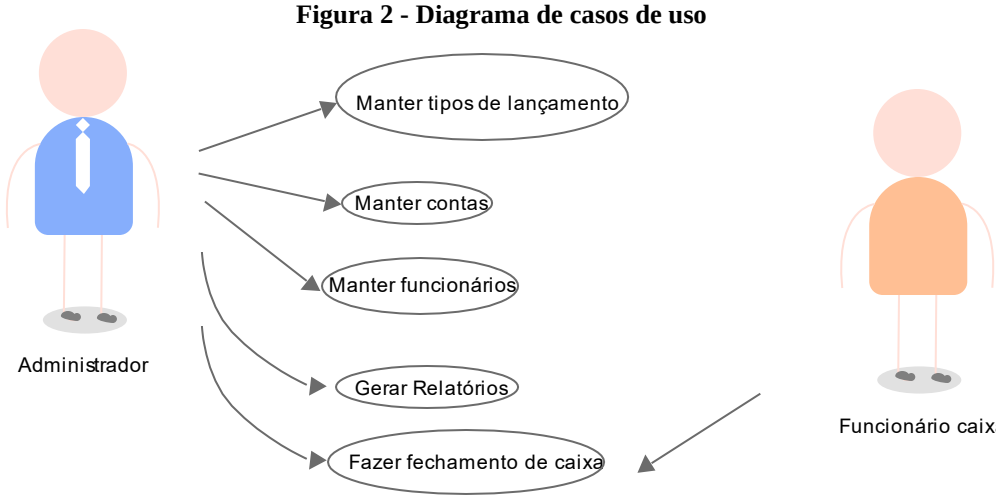
d) Lançamentos por usuário:

Nome do Usuario

Descrição	Modalidade	Tipo/Valor
item	item	(+ ou -)/R$item

A Figura 2 apresenta o diagrama de casos de uso.

Figura 2 - Diagrama de casos de uso



Fonte: autoria própria

No Quadro 4 está a descrição da operação incluir do caso de uso denominado manter tipos de lançamento.

Caso de uso: Manter tipos de lançamento. **Descrição**: São as modalidades de lançamento feitas durante o fechamento de caixa, ex.: cartão de crédito, vale salário, devolução ao cliente, entre outros **Atores**: Administrador. **Pré-condição**: Estar autenticado no sistema. **Sequência de Eventos**: 1. Administrador deve acessar a tela para cadastrar tipos de lançamento. 2. Sistema apresenta formulário de cadastro. 3. Administrador preenche os dados e envia formulário para que dados sejam salvos. 4. Sistema informa que os dados foram incluídos no banco de dados. **Pós-Condição**: Dados do novo tipo cadastrado pelo administrador.

Nome do fluxo alternativo (extensão)	Descrição
Linha 3: Ator informa campos inválidos ou não informa campos obrigatórios.	4.1 No momento de salvar, o sistema faz a verificação e constata que há dados inválidos ou de preenchimento obrigatório não informados. É emitida mensagem informando ao usuário. 4.2 Retorna para o formulário de cadastro em estado de edição – passo 3.

Quadro 8 – Caso de uso manter tipos de lançamento

No Quadro 5 está a descrição do caso de uso fazer fechamento de caixa, operação lançamento de transição.

Caso de uso:
Fazer fechamento de caixa: operação, lançamento de transição
Descrição:
Cadastro dos lançamentos monetários feito pelo responsável da carteira.
Atores:
Administrador ou Funcionário Caixa.
Pré-condição:
Estar autenticado no sistema.
Sequência de Eventos:
1. Ator entra na tela de fechamento de caixa
2. Ator informa os dados necessário para cadastrar o lançamento e pressiona a tecla Enter.
3. Sistema inclui os dados na tabela de lançamentos diários e refaz o cálculo do valor em caixa
Pós-Condição:
Lançamento salvo no banco de dados

Nome do fluxo alternativo (extensão)	Descrição
Linha 2: Ator informa dados incorretos ou não informa dados obrigatórios.	4.1 Sistema marca em cor vermelha os campos que devem ser revistos.

Quadro 5 – Caso de uso fazer fechamento de caixa: operação lançamento de transação

No Quadro 6 está a descrição da operação gerar relatórios

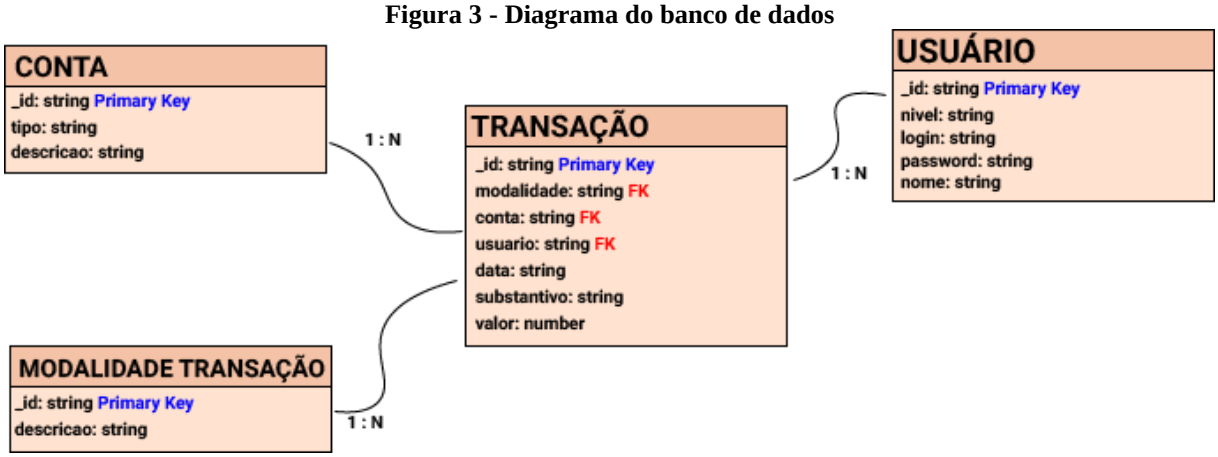
Caso de uso:
Caso de uso gerar relatórios.
Descrição:
Administrador acessa dados do sistema.
Atores:
Administrador.
Pré-condição:
Estar autenticado no sistema. Possuir lançamentos cadastrados.
Sequência de Eventos:
1. Administrador acessa a tela de relatórios
2. Administrador pressiona o botão visualizar para o relatório escolhido.
3. Sistema apresenta em tela dados requisitados.
Pós-Condição:
Dados requisitados exibidos em forma de relatório.

Nome do fluxo alternativo (extensão)	Descrição
Linha 3. Não há dados cadastrados para apresentação em tela	4.1 Sistema retorna a mensagem “Não constam dados cadastrados”

Quadro 6 – Caso de uso gerar relatórios

A Figura 3 apresenta o diagrama de entidades e relacionamentos do banco de dados.

Figura 3 - Diagrama do banco de dados



Fonte: autoria própria

No diagrama da Figura 3 está clara a simplicidade do banco de dados, tendo em vista que será executado em uma modalidade orientada a documentos. Desta forma, a ideia principal é o sistema funcionar em torno das transações, tornando-o muito leve. Os relatórios serão gerados com base nas tabelas relacionadas com a tabela transação e as datas registradas.

Além disso, a tabela “usuário” será utilizada para denominar o que será apresentado ao ator que efetuar o acesso, utilizando a propriedade nível.

3.3 APRESENTAÇÃO DO SISTEMA

Nesta seção, serão apresentadas as principais telas do sistema que inicia com a principal interface do sistema que é a tela de fechamento de caixa apresentada na Figura 4.

Figura 4 - Tela de registro de lançamentos

Fonte: Autoria própria

A tela foi desenvolvida com base no *framework* Angular Material, sendo disposto o seu conteúdo em duas colunas. Uma delas, à esquerda, contém os passos para efetuar um registro, que pode ser feito sem o auxílio do *mouse*, pois, depois de registrado o primeiro, o cursor será direcionado para o início do formulário novamente. E uma segunda coluna, à direita, na qual fica a tabela de apresentação dos dados cadastrados, sendo filtrados pela data apresentada no formulário da esquerda.

Na Figura 5 está a tabela da tela de fechamento de caixa com dados que exibem os valores aplicando cores para orientar os registros referentes às entradas e saídas. As ações editar e excluir exibem um *pop-up* no meio tela para confirmar a opção escolhida. Por fim, o total diário é exibido na parte inferior da tabela, servindo como um totalizador das transações diárias.

Figura 5 - Tabela registro lançamentos

Fonte: Autoria própria

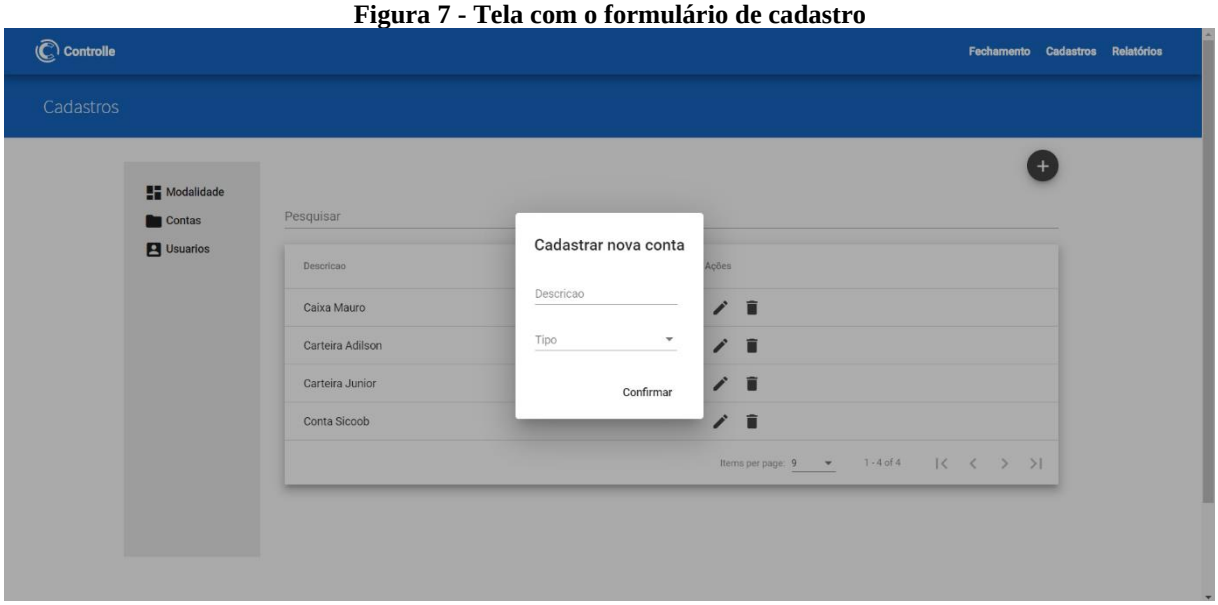
Na Figura 6, a tela de cadastros segue os padrões *Material Design*, aplicados pelo *Angular Material*. Possui o botão de adicionar localizado na parte superior direita da tabela e as opções de editar e excluir na própria tabela. Nota-se, no rodapé da tabela, a quantidade de itens dispostos por página, e o total, podendo ser alterado manualmente pelo usuário a quantidade exibida. Esta tela é acessada apenas pelo usuário com perfil de Administrador.

Figura 6 - Tela de modalidades

Fonte: Autoria própria

O menu posicionado na parte esquerda dá a escolha de qual tabela de cadastros será exibida. Assim, o botão adicionar chama o formulário referente à tabela apresentada, como na Figura 7.

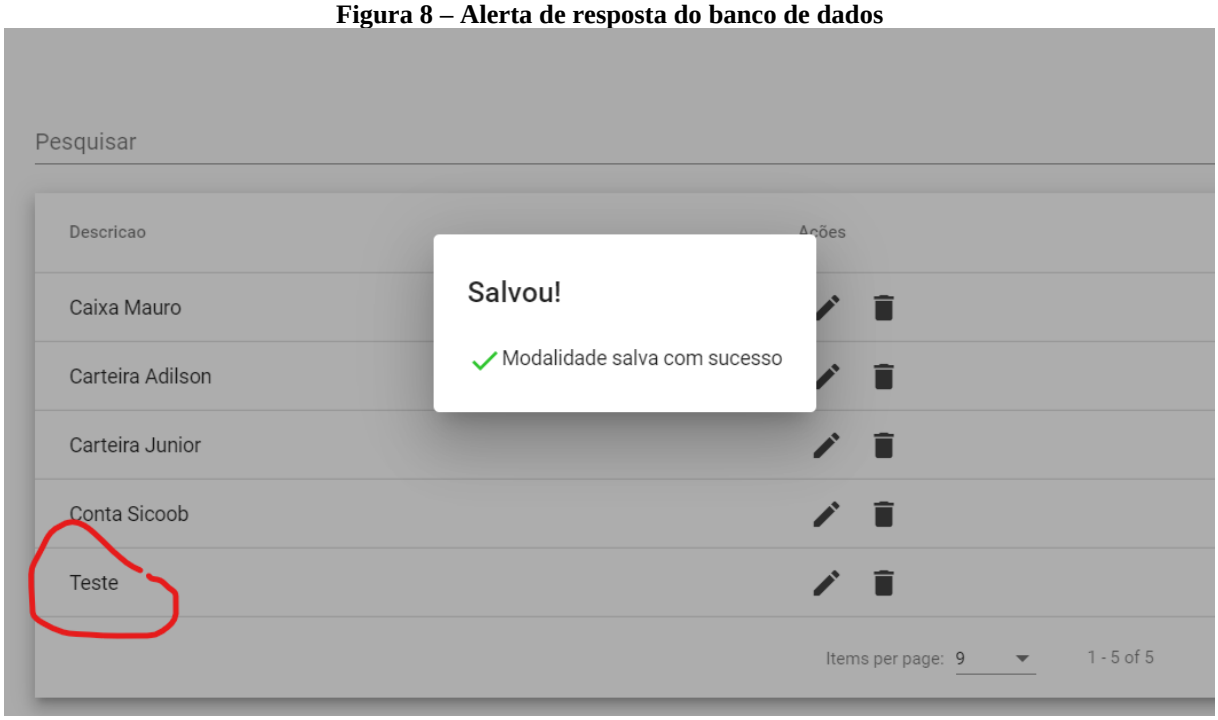
Figura 7 - Tela com o formulário de cadastro



Fonte: Autoria própria

Depois de preenchido o formulário de cadastro e confirmado, é apresentado um *pop-up* de resposta ao usuário, podendo ser de sucesso ou não (caso, como exemplo, já houver uma conta com a mesma descrição, ou extrapolar o número máximo de caracteres), esta tela pode ser visualizada na Figura 8.

Figura 8 – Alerta de resposta do banco de dados



Fonte: Autoria própria

O elemento cadastrado é adicionado instantaneamente na tabela apresentada e esse resultado é informado ao usuário.

A tela de relatórios, também acessada apenas pelo usuário Administrador, segue os padrões *Material Design* e aplica gráficos da biblioteca *ChartJs*, disponível em *Chartjs.org*. Um exemplo de gráfico pode ser visualizado na Figura 9.

Figura 9 - Tela dos relatórios

Fonte: Autoria própria

A tela inicial de relatórios apresenta o gráfico de vendas da última semana e os valores disponíveis em cada carteira, aplicando cores para as positivas e as negativas. As modalidades de relatório disponíveis estão apresentadas em um menu tipo barra na parte superior do relatório.

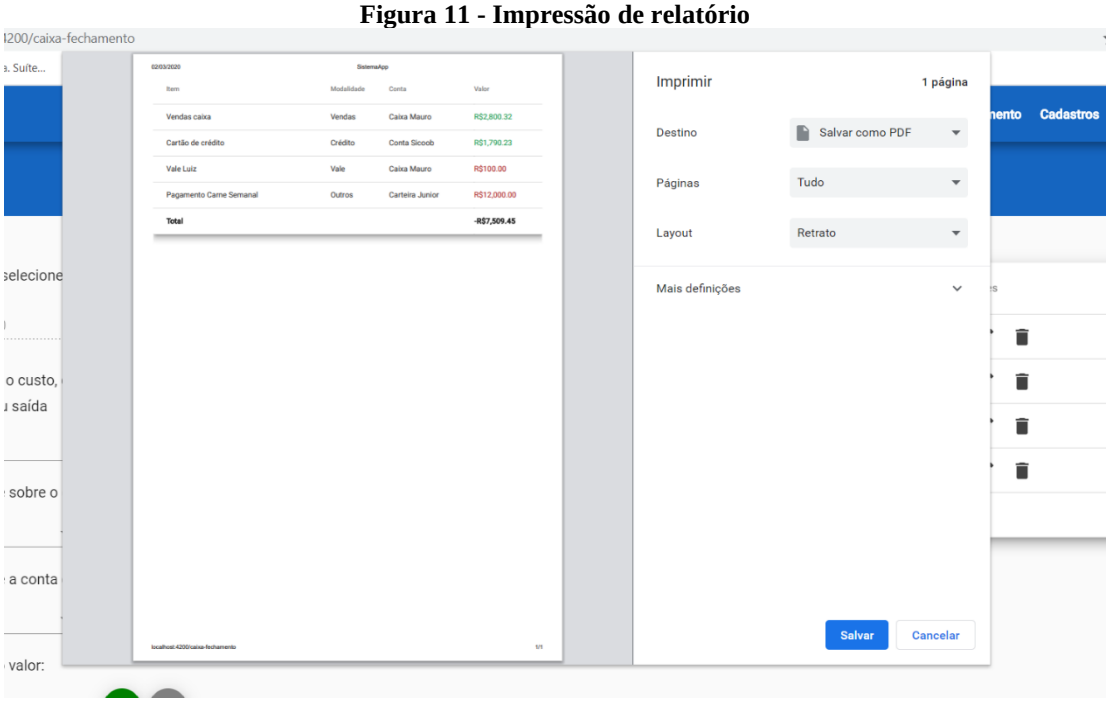
Os outros relatórios são exibidos no formato de tabela, como pode ser visualizado na Figura 10.

Figura 10 - Tela relatórios lançamentos por modalidade

Fonte: Autoria própria

O relatório de lançamentos por modalidade apresenta as transações lançadas em um determinado período de tempo da modalidade selecionada. Caso seja solicitada a impressão, será exibida apenas a tabela, como pode ser observado na Figura 11.

Figura 11 - Impressão de relatório



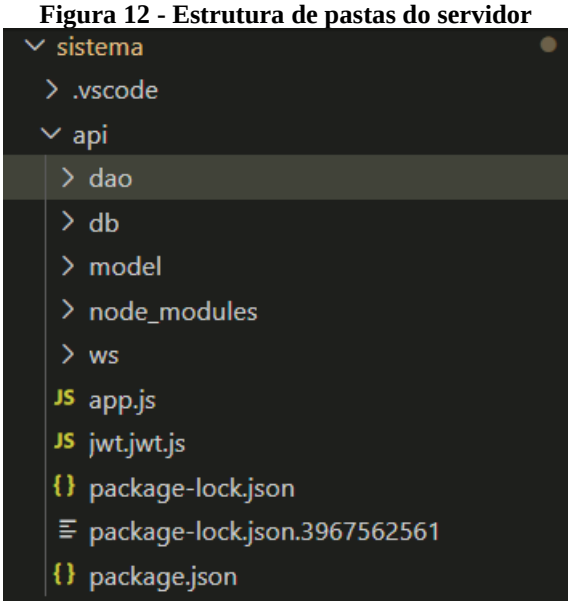
Fonte: Autoria própria

Assim, caso seja necessária a impressão física de algum documento, será oferecida a opção ao usuário.

3.4 IMPLEMENTAÇÃO DO SISTEMA

Nesta seção será apresentada a implementação do sistema. O desenvolvimento é divido em *front-end*, página de apresentação ao usuário, e *back-end*, desenvolvimento das funções de servidor, que neste caso, como foi utilizado o NodeJs, ainda visa as aplicações JavaScript serem trazidas para o servidor. Então, a estruturação do *back-end* é dividida em, conforme apresentado na Figura 12: “dao”, os arquivos com funções de relações com o banco de dados, “db”, arquivo com funções de conexão e desligamento do banco de dados, “model”, arquivos que estruturam objetos, e “ws”, arquivos que fornecem os *services* para um caminho *Uniform Resource Locator* (URL) sugerido.

Figura 12 - Estrutura de pastas do servidor



Fonte: Autoria própria

São criadas subpastas para armazenar os aquivos JavaScript *“.js”* de acordo com suas funções. A pasta “dao” armazena os arquivos que fazem as funções de conexão com o banco, como salvar, excluir, listar e atualizar. Nessa pasta também estão os arquivos com as estruturas de busca, conforme pode ser observado na Listagem 1.

Listagem 1 - Exemplo de arquivo DAO

```
exports.listar = (fnCallback) => {
    db.connect()

    let q = Transacao.find().sort('substantivo').populate('modalidade').populate('conta')
    q.exec( (e, ret) => {
        db.disconnect()
        fnCallback(ret)
        if(e){
            console.log(e.toString())
        }
    })
}

exports.litarDebitos = (fnCallback) => {
    db.connect()

    let q = Transacao.find().where('substantivo').equals('D').sort('-data').populate('modalidade').populate('conta')
    q.exec((e, ret)=>{
        db.disconnect()
        fnCallback(ret)
        if(e){
            console.log(e.toString())
        }
    })
}
```

Fonte: Autoria própria

Imagem que faz parte do arquivo *DAO* para as transações, exemplificando a busca de todas as transações em uma função JavaScript, ou da busca de apenas débitos. Outro exemplo está na Listagem 2, que mostra a função de busca por modalidade, que aceita tanto um objeto do tipo modalidade, ou apenas uma *string* da modalidade requerida, por fim faz a busca no banco e retorna a lista de transações desejadas.

Listagem 2 - Exemplo 2 de arquivo DAO

```
exports.listarpmod  = async (mod, fnCallBack) =>{

    db.connect()
    let modalidadepesq
    console.log(mod)

    if(typeof mod[0].modalidade  === "string" ){
        modalidadepesq =  mod[0].modalidade
        console.log("aqui")
    }else{
        modalidadepesq = mod[0].modalidade.descricao
        console.log("ou aqui")
    }

    let qmod = Modalidade.find({'descricao': modalidadepesq})
    qmod.exec((e, retmod) =>{

        if(e){
            console.log(e)
        }

        retmod[0]._id

        let q = Transacao.find({'modalidade': retmod[0]._id}).sort('data').populate('modalidade').populate('conta')
        q.exec((e,ret) =>{

        if(e){
            console.log(e.toString())
        }

        this.retorno = ret

       fnCallBack(ret)
       db.disconnect()

    })
    })
```

Fonte: Autoria própria

A segunda pasta é a “db”, que possui os arquivos de configuração para a conexão com o banco de dados, exportando as funções de conectar e desconectar, apresentadas na Listagem 3.

Listagem 3 - Arquivo para conexão com o Banco de Dados

```
const mongoose = require('mongoose')

exports.connect = () => {
    mongoose.connect('mongodb://localhost:27017/sistema',  { useNewUrlParser: true })
}

exports.disconnect = () => {
    mongoose.disconnect()
}
```

Fonte: Autoria própria

Ressalta-se o uso da biblioteca *mongoose*, que possui todas as funções necessárias para esse sistema em sua estrutura.

A terceira pasta é a “model”, que armazena os arquivos de definição de objetos e suas propriedades. Como o arquivo “transacao.model.js” que estrutura uma transação para se relacionar de forma mais adequada com o banco de dados e o *front-end*, apresentado na Listagem 4.

Listagem 4 - Exemplo de arquivo Model no servidor

```
const mongoose = require('mongoose')
const types = mongoose.Schema.Types

const transacaoSchema = new mongoose.Schema({
    descricao : {
        type : String,
        required : true
    },
    valor : {
        type : Number,
        required : true
    },
    data : {
        type: Date,
        require: true,
    },
    substantivo: {
        type: String,
        require: true
    },
    modalidade : {
        type: types.ObjectId,
        ref: 'ModalidadeTransacao',
        required: true
    },
    conta:{
        type: types.ObjectId,
        ref: 'Conta',
        required: true
    }
})

module.exports = mongoose.model('Transacao', transacaoSchema)
```

Fonte: Autoria própria

Assim, depois de definidas suas propriedades, é exportado como um “model” da estrutura *mongoose*. Agregando ao objeto as funções disponíveis de comunicação com o banco, dando ao *objetoDAO* a possibilidade de se formar nos moldes dessa estrutura.

A próxima pasta “node_modules”, armazena os arquivos do próprio NodeJs e suas bibliotecas, como a *Mongoose* ou *JWS*. Os arquivos não foram desenvolvidos neste trabalho, mas foram importados a partir de bibliotecas criadas por outros desenvolvedores.

A última pasta, a “ws”, define as rotas de comunicação com o *front-end* da aplicação, dando a cada arquivo uma organização de buscas e retornos, apresentado na Listagem 5.

Listagem 5 - Exemplo arquivo WebService no servidor

```
const dao = require('../dao/modalidade-transacao.dao')

module.exports = (app) => {

    app.route('/modalidade-transacao/listar').get( (req, res) => {
        dao.listar( (result) => {
            res.json(result)
        })
    })

    app.route('/modalidade-transacao/salvar').post( (req, res) => {
        let dados = req.body

        dao.salvar(dados, () => {
            res.json({})
        })
    })

    app.route('/modalidade-transacao/excluir/:id').get( (req, res) => {
        let id = req.params.id
        dao.excluir(id, () => {
            res.json({})
        })
    })

    app.route('/modalidade-transacao/atualizar').post( (req, res) => {
        let dados = req.body
        dao.atualizar(dados, () => {
            res.json({})
        })
    })

}
```

Fonte: Autoria própria

Na pasta raiz do projeto está o arquivo “app.js, que é responsável pela integração entre os módulos do sistema e pela execução do sistema, o código pode ser observado na Listagem 6.

Listagem 6 - Arquivo de execução do servidor

```
const cors = require('cors')
const express = require('express')

const app = require('express')()
const server = require('http').Server(app)

const wsTransacao = require('./ws/transacao.ws')
const wsModalidadeTransacao = require('./ws/modalidade-transacao.ws')
const wsUsuario= require('./ws/usuario.ws')
const wsConta = require('./ws/conta.ws')
const wsDateArray = require('./ws/date-array.ws')

app.use(cors())
app.use(express.json())

wsTransacao(app)
wsModalidadeTransacao(app)
wsUsuario(app)
wsConta(app)
wsDateArray(app)

server.listen(3000, () => {
    console.log("Servidor iniciado...")
})
```

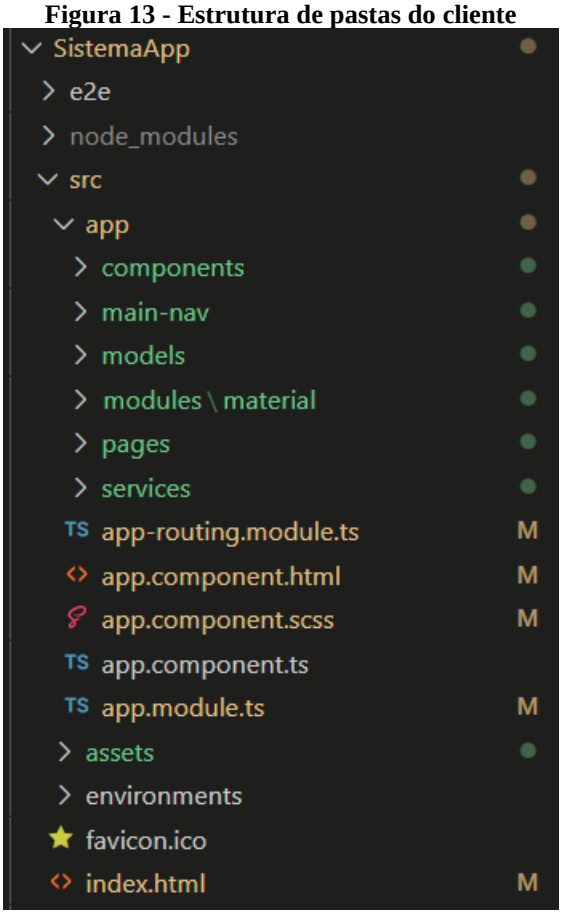
Fonte: Autoria própria

Por fim, este arquivo correlaciona os *Wss* e coloca o servidor em execução, disponibilizando caminhos de conversa que serão acessados pelo *fron-tend* e com o banco de dados. O próprio *NodeJS* constrói a aplicação através do comando “node /nome do arquivo/”.

A segunda etapa é o desenvolvimento do *front-end*, feito com a biblioteca “Angular” e a biblioteca “Angular Material”, desenvolvida na seguinte estrutura, pasta “components”, arquivos de componentes genéricos para uso futuro, pasta “main-nav”, possui o código *HyperText Markup Language* (HTML), *Sassy CSS* (SCSS) e *TypeScript* da barra de menu presente em todas as páginas do sistema, “models”, como no servidor, possuí a estruturação de objetos, “modules”, possuí arquivos de importação de modulos, “pages”, todos os códigos

HTML, CSS e TypeScriptdas páginas do sistema, “service”, os arquivos com funções para se comunicar com o servidor por meio de rotas pré-estabelecidas. Esta estrutura é apresentada na Figura 13.

Figura 13 - Estrutura de pastas do cliente



Fonte: Autoria própria

É importante ressaltar, primeiramente, a pasta “models”, que contém, da mesma maneira que no servidor, a estruturação de um objeto com seus atributos. Assim, no momento de comunicação, a estruturação do objeto se dá de maneira muito simples, lembrando que o MongoDB é baseado em documentos, logo, sua manuseabilidade é menos rígida. Assim, a estruturação de um objeto no aplicativo de *front-end*, se dá como na Listagem 7.

Listagem 7 - Exemplo de arquivo Model no cliente

```
import { Conta } from './conta.model';

export class Transacao{
    _id : string
    descricao : string
    valor : number
    substantivo : string
    data : Date
    modalidade : ModalidadeTransacao
    conta: Conta
}
```

Fonte: Autoria própria

O *framework* Angular é desenvolvido na linguagem TypeScript, usando, assim, além de sua estruturação padrão, as bibliotecas fornecidas por ela, como “Date”.

Na pasta raiz do projeto se encontra o arquivo “app-routing.module.ts”, que estrutura os caminhos feitos pelas páginas acessadas pelo usuário, apresentada na Listagem 8.

Listagem 8 - Arquivo de rotas no cliente

```
const routes: Routes = [
  {path: 'caixa-fechamento', component: FechamentoComponent},
  {path: 'cadastro', component: CadastroComponent, children:[
    {path:"modalidade-transacao", component: ModalidadeTransacaoComponent } ,
    {path:"conta", component: ContaComponent}
  ]},
  {path: 'relatorios', component: RelatoriosComponent, children:[
    {path: "saldos", component: SaldosComponent},
    {path: "lancpMod", component: LancamentoPorModalidadeComponent},
    {path: '', redirectTo: 'saldos', pathMatch: 'full'}
  ]},
  {path: 'imprimir-caixa-dia', component: CaixaDiaComponent},
  {path: 'login', component: LoginComponent},
  {path: '', redirectTo: 'login', pathMatch: 'full'}

];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule {


}
```

Fonte: Autoria própria

Nota-se a estruturação das páginas do sistema, disponibilizando está a biblioteca a possibilidade de subcaminhos, tornando mais rápido o acesso de páginas que pertencem a um determinado grupo. Como no *path* “cadastro”, que se subdivide em “modalidade-transacao”, conta e “usuário”.

Também são criados previamente os arquivos “services” que fazem a comunicação com o servidor, alimentando os modelos, ou transferindo-os. Neste arquivo, são enviados pacotes às rotas definidas nos arquivos “ws” do *back-end*, como pode ser visualizado na Listagem 9.

Listagem 9 - Exemplo de arquivo Service no cliente

```
const WS_SALVAR = "http://localhost:3000/transacao/salvar"
const WS_LISTAR = "http://localhost:3000/transacao/listar"
const WS_LISTARDEBITOS = "http://localhost:3000/transacao/listarDebitos"
const WS_LISTARCREDITOS = "http://localhost:3000/transacao/listarCreditos"
const WS_LISTARDATA = "http://localhost:3000/transacao/listarData"
const WS_ATUALIZAR = "http://localhost:3000/transacao/atualizar"
const WS_EXCLUIR = "http://localhost:3000/transacao/excluir/"
const WS_LISTARPMOD = "http://localhost:3000/transacao/listarpmod"

@Injectable({
  providedIn: 'root'
})

export class TransacaoService {

  constructor(private http : HttpClient) {

  }

  salvar(transacao : Transacao, cb : () => void) {
      this.http.post(WS_SALVAR, transacao).subscribe(cb, this.tratarErro)
  }

  listar(cb : (ret : any) => void){
      this.http.get(WS_LISTAR).subscribe(cb, this.tratarErro)
  }
  listarDebitos(cb : (ret : any) => void){
      this.http.get(WS_LISTARDEBITOS).subscribe(cb, this.tratarErro)
  }
  listarCreditos(cb : (ret : any) => void){
      this.http.get(WS_LISTARCREDITOS).subscribe(cb, this.tratarErro)
  }
```

Fonte: Autoria própria

Portanto, utilizando-se da biblioteca “HttpClient”, são realizados os *posts* e *gets*, do cliente para o servidor. Caso seja necessário, pode-se tratar algum dado antes de enviá-lo, mas isto foi feito nas páginas *typescript* dos componentes de apresentação.

O primeiro componente criado no desenvolvimento foi o “main-nav”, que é o menu superior presente em todas as páginas. Em seu HTML, usa as *tags* disponibilizadas pelo Angular Material, visualizadas na Listagem 10.

Listagem 10 - Exemplo de arquivo HTML

```
<mat-sidenav-container class="sidenav-container">
  <mat-sidenav #drawer class="sidenav"
      [ngClass]="{hidden: !(isHandset$ | async)}"
      fixedInViewport="false"
      [attr.role]="(isHandset$ | async) ? 'dialog' : 'navigation'"
      [mode]="(isHandset$ | async) ? 'over' : 'side'"
      [opened]="!(isHandset$ | async)">
    <mat-toolbar > Menu </mat-toolbar>
    <mat-nav-list>
      <a mat-list-item href="caixa-fechamento">Fechamento</a>
      <a mat-list-item href="cadastro">Cadastros</a>
      <a mat-list-item href="relatorios">Relatórios</a>
    </mat-nav-list>
  </mat-sidenav>
  <mat-sidenav-content >
    <mat-toolbar color="primary">
      <button
        type="button"
        aria-label="Toggle sidenav"
        mat-icon-button
        (click)="drawer.toggle()"
        *ngIf="isHandset$ | async">
        <mat-icon aria-label="Side nav toggle icon" class="font-color-nav">menu</mat-icon>
      </button>
      <span class="font-color-nav"><div class="logo-toolbar" ></div></span>
      Controlle
      <span class="spacer"></span>
      <a  [ngClass]="{hidden: (isHandset$ | async)}" class="font-color-nav" href="caixa-fechamento">Fechamento</a>
      <a  [ngClass]="{hidden: (isHandset$ | async)}" class="font-color-nav" href="cadastro">Cadastros</a>
      <a  [ngClass]="{hidden: (isHandset$ | async)}" class="font-color-nav" href="relatorios">Relatórios</a>

    </mat-toolbar>
      <div class="padding-content">
      <router-outlet></router-outlet>
    </div>
  </mat-sidenav-content>
</mat-sidenav-container>
```

Fonte: Autoria própria

Dá-se notoriedade as opções disponibilizadas pela biblioteca, como [ngClass], que permite executar uma estrutura “if” para definir qual classe será atribuída ao objeto, ainda acessando variáveis ao código TypeScript ligado a esta página, como “isHandset$”, que é uma variável *booleana* para definir se o tamanho da tela é maior ou menor que X.

Ainda abaixo do *toolbar* é definido uma *tag* da estrutura angular, para apresentar o disposto nas rotas definidas anteriormente. Está é a *tag* “<router-outlet>”.

Também foram criados componentes de uso geral, como o “date-picker” personalizado nos padrões pt-BR e *dialogs* para alertar ou confirmar funções executadas no sistema, que podem ser vistos, respectivamente nas Listagens 11, 12 e 13.

Listagem 11 - Exemplo de arquivo TS para componente de uso futuro

```
import { Component, OnInit } from '@angular/core';
import { NativeDateAdapter } from '@angular/material';

const MY_DATE_FORMATS = {
  parse: {
      dateInput: {month: 'short', year: 'numeric', day: 'numeric'}
  },
  display: {
      // dateInput: { month: 'short', year: 'numeric', day: 'numeric' },
      dateInput: 'input',
      monthYearLabel: {year: 'numeric', month: 'short'},
      dateA11yLabel: {year: 'numeric', month: 'long', day: 'numeric'},
      monthYearA11yLabel: {year: 'numeric', month: 'long'},
  }
};

export class MyDateAdapter extends NativeDateAdapter {
    format(date: Date, displayFormat: Object): string {
        if (displayFormat =="input") {
            let day = date.getDate();
            let month = date .getMonth() + 1;
            let year = date.getFullYear();
            return this._to2digit(day) + '/' + this._to2digit(month) + '/' + year;
        } else {
            return date.toDateString();
        }
    }

    private _to2digit(n: number) {
        return ('00' + n).slice(-2);
    }
}
```

Fonte: Autoria própria

A classe que adapta o componente “DatePicker” estende da própria classe do Angular Material, assim, se mudam apenas propriedades específicas, como apresentação da data atual e usa o tema disponibilizado pela Google.

Listagem 12 - Exemplo de arquivo HTML para componente de uso futuro

sistema > SistemaApp > src > app > pages > dialog > alert > alert.component.html > h1

```
<h1 mat-dialog-title>{{data.titulo}}</h1>
<div mat-dialog-content>
  <div>

    <mat-icon [ngClass]="{success: (data.type == 'success')
                        , fail: (data.type == 'fail')}"
              class="float-left"
              sytle="font-size: 40px">
    {{data.icone}}</mat-icon>

    <span class="spacer"></span>

    <mat-dialog-content class="mat-typography float-right">
          {{data.mensagem}}
    </mat-dialog-content>

    <div class="correct-float"></div>
  </div>
</div>
```

Fonte: Autoria própria

Listagem 13 - Exemplo 2 de arquivo HTML para componente de uso futuro

sistema > SistemaApp > src > app > pages > dialog > confirm > confirm.component.html > h2

```
<h2 mat-dialog-title>{{data.titulo}}</h2>
    <mat-dialog-content class="mat-typography">
        {{data.mensagem}}
    </mat-dialog-content>
<mat-dialog-actions align="end">
  <button mat-button mat-dialog-close>Cancelar</button>
  <button mat-button [mat-dialog-close]="true" cdkFocusInitial>Confirmar</button>
</mat-dialog-actions>

```

Fonte: Autoria própria

Esses componentes de uso genérico, que são construídos durante a criação da página, devem ser especificados no arquivo de configurações de módulo, na pasta raiz “app.module.ts” como componentes de entrada, da forma apresentada na Listagem 14.

Listagem 14 - Declaração dos componentes de entrada

```
],entryComponents: [
  ModalFormComponent,
  AlertComponent,
  ConfirmComponent,
  FormMTComponent,
  FormContaComponent
],
```

Fonte: Autoria própria

Então, foram desenvolvidos os componentes principais, que são os de apresentação das funcionalidades. O primeiro exemplo é o componente para lançamento das transações, chamado de fechamento de caixa. Ele possui um desenvolvimento de arquivos HTML, TypeScript e SCSS que ultrapassa quinhentas linhas de código, portanto, será resumido seu funcionamento.

Depois de especificadas as *tags html* e suas opções. O arquivoTypeScript entra em ação, sendo implementada a interface “OnInit” que permite chamar funções durante o carregamento inicial da página, portanto, dentro da sua implementação foram chamadas as funções de população de *arrays* relativos aos objetos cadastrados no banco utilizando através o arquivo *service* denominado na construção do TypeScript, dispostas na Listagem 15.

Listagem 15 - Exemplo 2 de arquivo TypeScript de tela do cliente

```
constructor(private ts: TransacaoService,
            private mts: ModalidadeTransacaoService,
            private dialog : MatDialog,
            private cs: ContaService) { }

listar(date : Time){
  this.ts.listarData(date, (ret)=>{
    this.transacoes = ret
    console.log(this.transacoes)
  })
}

ngOnInit() {
  this.transacao.data = moment().toDate()
  sessionStorage.setItem('dataFind', this.transacao.data.toString())

  this.dataPEB = new Time()
  this.dataPEB.dataString = moment().toDate().toString()
  this.dataPEB.dataDate = moment().toDate()
  this.listar(this.dataPEB)

  sessionStorage.setItem('dataPEB', this.dataPEB.dataString.toString())

  this.mts.listar((ret) =>{
    this.modalidades = ret
  })
  this.cs.listar(ret =>{
    this.contas = ret
  })
}
```

Fonte: Autoria própria

Logo, nas primeiras linhas mostra-se a aplicação dos *services*, colocados no momento da construção do arquivo *TypeScript*, depois, a criação de uma função que utiliza o *service* para recorrer ao servidor e alimentar as transações que serão exibidas, por fim, a chamada do *ngOnInit* que executa essas funções no momento da construção da página HTML.

Outro exemplo é a construção de gráficos feita nos relatórios. Lá, foi usada a biblioteca “chart.js”, que fornece gráficos prontos, é necessário apenas fornecer os dados de maneira correta. Então, na página que exibe as últimas vendas em gráfico, foi preciso desenvolver uma função para criar o intervalo de datas. O código de um dos gráficos pode ser observado na Listagem 16.

Listagem 16 - Função para gerar intervalo de datas

```
getDates(){

  this.strDate.dataString = moment(this.inicial, "YYYY-MM-DD").toDate().toString()
  this.strDate.dataDate = moment(this.final, "YYYY-MM-DD").toDate()
  this.endDate.dataString = moment().toDate().toString()
  this.endDate.dataDate = moment().toDate()

  this.ds.btwDates(this.strDate, this.endDate, (ret)=>{
      ret.arr.forEach(dt => {
        let dtsplit = dt.split("T")
        let jsdate = new Date(dtsplit[0])
        this.dateArray.push(jsdate.toLocaleTimeString('pt-br',{
          year: 'numeric',
          month: 'short',
          day: 'numeric'
        }))

      });
  })
}
```

Fonte: Autoria própria

Depois, na Listagem 17, a função que por meio dos *services* busca os dados no banco.

Listagem 17 - Função para buscar dados referentes ao relatório

```
  getTransacpMod(){
    this.ts.listarPorModalidade('Vendas', (ret)=>{
      this.trasacpmodArray = ret
      ret.forEach(el =>{

        let dtsplit = el.data.split("T")
        let jsdate = new Date(dtsplit[0])

        this.valoresArray.push(el.valor)

        this.dtTransac.push(jsdate.toLocaleTimeString('pt-br',{
          year: 'numeric',
          month: 'numeric',
          day: 'numeric',
        }).toString().split(" ")[0])

      })
      this.chartgen()
    })
  }
```

Fonte: Autoria própria

E por fim, a função que constrói o gráfico, nos parâmetros do *Chart.Js*, apresentados na Listagem 18.

Listagem 18 - Função para gerar gráfico

```
chartgen(){

  this.chart = new Chart(this.canvas.nativeElement, {
    type: 'line',
    data: {
        labels:  this.dtTransac,
        datasets: [{
            label: 'Vendas',
            data: this.valoresArray,
            backgroundColor: [
                'rgba(255, 99, 132, 0.2)'
            ],
            borderColor: [
                'rgba(255, 99, 132, 1)'
            ],
            borderWidth: 1
        }]
    },
    options: {
      maintainAspectRatio :false,
      responsive: true,
      legend: {
        label: {
          fontSize: 12
        }
      }
    }
})
```

Fonte: Autoria própria

Também utilizado nas páginas de relatório, foi implementado ao arquivo *scss* global uma classe para definir o que será impresso ou não, apresentado na Listagem 19.

Listagem 19 - Classe para definir impressão

```
@media print {
    body * {
      visibility: hidden;
    }
    #printable, #printable * {
      visibility: visible;
    }
    #printable {
      position: fixed;
      left: 0;
      top: 0;
    }
    .mat-drawer{
        display: none!important
    }
  }
```

Fonte: Autoria própria

4 CONSIDERAÇÕES FINAIS

O objetivo deste trabalho foi o desenvolvimento de uma aplicação web para gerenciar o controle financeiro de um grupo de sócios e funcionários que administram o dinheiro de uma empresa.

O desenvolvimento de sistemas, tecnologias que solucionam funcionamentos de forma computacional em sua essência, tem passado por fases. A primeira disponibilizava ao desenvolvedor recursos escassos, quanto ao usuário era mais precária ainda, possibilitando produzir um software para apenas atender funcionalidades simples, como armazenar dados cadastrais que poderiam ser distribuídos de forma ilimitada para todos os usuários do sistema. Porém, com a acelerada expansão computacional, cada vez mais se dá a quem produz maior flexibilidade para fornecer ferramentas computacionais não somente completas, mas essenciais aos usuários.

Compreende, então, neste contexto, que é necessário o programador, produzir bibliotecas de forma inteligente para alcançar a necessidade de cada requisito para cada sistema específico, aumentando a produção, expandindo a gama de processos de negócio transformados para processos tecnológicos.

O apresentado neste texto relata a produção, desde a fase de conhecimento do ambiente real até a entrega do sistema em ambiente computacional de um sistema web personalizado visando atender requisitos específicos de um cliente. Desta forma, o projeto foi estruturado, tendo um aprofundamento considerável nos componentes do Angular Material, muito utilizados na apresentação das funções nos componentes. Também, um estudo longo nas bibliotecas disponibilizadas pelo Angular e pelo Node.js e na própria programação TypeScript e JavaScript.

As principais dificuldades encontradas durante o desenvolvimento do trabalho foram criar maior contato com as funções definidas pelos *frameworks*, que, por mais que tragam produtividade no desenvolvimento, o primeiro contato é trabalhoso.

Como trabalho futuro pretende-se desenvolver outros relatórios e possibilitar a divisão das transações por diversas empresas simultaneamente. Além disso, definir uma classe de fornecedores, para dar destino as saídas de caixa e com isso criar a modalidade transferência, além de entrada/saída.

REFERÊNCIAS

ANGULAR UNIVERSITY. **Angular Security - Authentication With JSON Web Tokens (JWT):** The Complete Guide. Disponível em: < https://blog.angular-university.io/angular-jwt-authentication//>. Acesso em: 22 fev. 2020.

GOOGLE. Angular Documentation. Disponível em: < https://angular.io/docs >. Acesso em: 29 fev. 2020.

GOOGLE. Componentes Angular Material. Disponível em: < https://material.angular.io/components/categories >. Acesso em: 29 fev. 2020.

RAIMUNDO BARBOSA, J. **Usando Angular Chart,** Saiba como criar gráficos usando Angular Chart. Disponível em: < https://ifpb.github.io/jaguaribetech/2016/09/01/usando-angular-chart/>. Acesso em: 26 fev. 2020.

OPENJS FUNDATION. NodeJS Documentation. Disponível em: < https://nodejs.org/en/docs/ >. Acesso em: 29 fev. 2020.

PÁDUA P FILHO, W. Engenharia de Software - Projetos e Processos - Vol. 2: Volume 2, LTC – Sâo Paulo – 2019.