UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
DEPARTAMENTO ACADÊMICO DE INFORMÁTICA
TECNOLOGIA EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

IOHAN CAMARGO
LUCAS INOUE

DESENVOLVIMENTO DE UM SIMULADOR DE CHÃO DE FÁBRICA COM AUXÍLIO DE FERRAMENTAS GRÁFICAS DE REDES DE PETRI

TRABALHO DE CONCLUSÃO DE CURSO

PONTA GROSSA
2017

IOHAN CAMARGO

LUCAS INOUE

DESENVOLVIMENTO DE UM SIMULADOR DE CHÃO DE FÁBRICA COM AUXÍLIO DE FERRAMENTAS GRÁFICAS DE REDES DE PETRI

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção do título de Tecnólogo em Análise e Desenvolvimento de Sistemas, do Departamento de Informática / Coordenação do Curso de Tecnologia em Análise e Desenvolvimento de Sistemas, da Universidade Tecnológica Federal do Paraná.

Orientador: Prof. Dr. Richard Duarte Ribeiro

PONTA GROSSA

2017

Ministério da Educação
Universidade Tecnológica Federal do Paraná
Campus Ponta Grossa
Diretoria de Pesquisa e Pós-Graduação
Departamento Acadêmico de Informática
Tecnologia em Análise e Desenvolvimento de Sistemas

TERMO DE APROVAÇÃO

DESENVOLVIMENTO DE UM SIMULADOR DE CHÃO DE FÁBRICA COM AUXÍLIO DE FERRAMENTAS GRÁFICAS DE REDES DE PETRI

por

IOHAN CAMARGO
LUCAS INOUE

Este Trabalho de Conclusão de Curso (TCC) foi apresentado em 1 de novembro de 2017 como requisito parcial para a obtenção do título de Tecnólogo em Análise e Desenvolvimento de Sistemas. O candidato foi arguido pela Banca Examinadora composta pelos professores abaixo assinados. Após deliberação, a Banca Examinadora considerou o trabalho aprovado.

_____
Richard Duarte Ribeiro
Prof. Orientador

_____
Geraldo Ranthum
Membro titular

_____
Rafael dos Passos Canteri
Membro titular

_____
Profª. Helyane Bronoski Borges
Responsável pelo Trabalho de Conclusão de Curso

_____
Profª. Dra. Mauren Louise Sguario
Coordenadora do curso

- O Termo de Aprovação assinado encontra-se na Coordenação do Curso -

AGRADECIMENTOS

Em primeiro lugar, agradecemos a Deus pela vida.

Em segundo lugar, agracedemos também aos nossos pais e entes queridos, que não mediram esforços para que pudéssemos chegar neste lugar hoje e toleraram nossos momentos de ausência.

Agradecemos ao nosso orientador prof. Dr. Richard Duarte Ribeiro pela paciência e pelos puxões de orelhas em momentos válidos.

Agradecemos aos colegas de caminhada na UTFPR por todos os momentos, sejam os bons ou ruins. Afinal, tudo é aprendizado. Extendemos o agradecimento à comunidade de software livre, aos canais de compartilhamento de soluções de programação, ao StackOverflow e ao Github.

A todos os envolvidos, muito obrigado.

RESUMO

CAMARGO, Iohan. INOUE, Lucas. **Desenvolvimento de um simulador de chão de fábrica com auxílio de ferramentas gráficas de redes de Petri**. 2017. 62 p. Trabalho de Conclusão de Curso de Tecnologia em Análise e Desenvolvimento de Sistemas - Universidade Tecnológica Federal do Paraná. Ponta Grossa, 2017.

Este trabalho descreve a implementação de um simulador de linhas de produção no chão de fábrica em ambiente *web,* utilizando *frameworks* para a implementação de um sistema cliente-servidor. O sistema foi desenvolvido utilizando Angular para a aplicação cliente e Laravel como servidor, realizando o intermédio com o banco de dados PostgreSQL. O simulador utiliza conceitos de redes de Petri em âmbito visual, substituindo os lugares por elementos gráficos em formato de cartões. Este projeto visa atingir resultados significativos de redução de desperdício de tempo entre várias ordens de produção, optando pelo planejamento de determinadas manutenções, entradas e saídas de matéria-prima e melhor distribuição de carga entre as máquinas.

Palavras-chave: Simulação computacional. Cliente-servidor. Planejamento de produção. Redes de Petri.

ABSTRACT

CAMARGO, Iohan. INOUE, Lucas. **Development of a shop floor simulator with graphical tools of Petri nets**. 2017. 62 p. Tecnology in System Analysis and Development Final Paper - Federal Technology University - Parana. Ponta Grossa, 2017.

This work describes a shop floor simulator development in web environment, using frameworks to construct a client server system. The system was developed using Angular to client application and Laravel as a server, doing the intermediate layer with the PostgreSQL database. The simulator uses Petri nets concepts in visual field, replacing the places with a graphical element in format of cards. This project aims to achieve significant results concerning time-wasting between several production orders, allowing planning in advance certain maintenance, inputs and outputs of raw material and better load distribution between machines.

Keywords: Computational simulation. Client server architecture. Production planning. Petri nets.

LISTA DE FIGURAS

LISTA DE SIGLAS E ACRÔNIMMOS

ACID	Atomicidade, Consistência, Isolamento e Durabilidade.
AIX	*Advanced Interactive Executive*
BSD	*Berkeley Software Distribution*
CSS	*Cascading Style Sheets*
ECMA	*European Computer Manufacturers Association*
ES	ECMAScript
GNU	*GNU's Not Unix*
HP-UX	*Hewlett-Packard*
HTML	*Hypertext Markup Language*
HTTP	*HyperText Transfer Protocol*
IBM	*International Business Machines*
IRIX	*Integrated Raster Imaging Unix*
JSON	Javascript *Object Notation*
LESS	*Style Sheet Language*
LTS	Suporte de longo prazo.
MS	Microsoft
ORM	Mapeamento Relacional de Objetos
OSI	*Open System Interconnection*
PHP	*Hypertext Preprocessor*
REST	*Representational State Transfer*
SASS	*Syntatically Awesome Style Sheets*
SGBD	Sistemas de Gerenciamento de Banco de Dados
SGI	*Silicon Graphics*
SQL	*Structured Query Language*
SVG	Gráficos vetoriais escaláveis
URI	*Uniform resource identifier*
URL	*Uniform resource locator*
W3C	*World Wide Web Consortium*
XML	*Extensible Markup Language*

SUMÁRIO

1 INTRODUÇÃO

As fábricas priorizam o planejamento da sua produção antes da execução, em busca da redução de desperdícios e de tornar o seu produto mais competitivo, os japoneses desenvolveram um dispositivo que sugere a análise dos seus processos produtivos diante do conceito de manufatura enxuta (OHNO, 1997), onde se visa a redução de quaisquer tipos de desperdício: o Sistema Toyota de Produção.

Segundo Ohno (1997), há sete tipos diferentes de desperdício: superprodução, espera, transporte, processamento, estoque, movimento e defeitos. Esta proposta tem por objetivo atuar na redução de desperdícios em espera, movimento e transporte.

Neste trabalho foi desenvolvido um sistema de simulação de chão de fábrica em ambiente *web*, onde o usuário irá controlar de maneira virtual um painel com cartões, unindo-os para formar linhas de produção virtuais, permitindo análises e testes das referidas linhas.

Para a construção deste sistema, foram utilizados conceitos de Redes de Petri, que é uma técnica de especificação de sistemas que possibilita sua representação virtual e matemática, também possui mecanismos de análise poderosa que permitem a verificação de propriedades e a verificação da corretude de um sistema especificado (MACIEL et al, 1996).

Para Maciel (1996), a representação gráfica das redes de Petri tem sido útil para visualizar os processos e a comunicação entre eles. Através do uso dessa técnica, define-se dois componentes da representação gráfica: os lugares (processo) e as transições. Os processos representam as máquinas, enquanto as transições, as ligações entre elas (interdependência no processo produtivo), formando uma cadeia, onde um processo é cliente do seu anterior e fornecedor do próximo. Com isso, um processo é iniciado somente se o seu predecessor for completado.

Nesta aplicação a representação das máquinas foi realizada por objetos gráficos semelhantes a cartões de papel que poderam ser interligadas entre si através de linhas criadas graficamente e dispostas numa interface virtual dentro de um navegador.

Foi realizada uma implementação de Rede de Petri utilizando as linguagens Javascript (JAVASCRIPT, 2016), PHP – *Hypertext Preprocessor* – (PHP, 2016*)* e tecnologias *REST*, na qual foi possível identificar os processos produtivos na indústria metalúrgica e avaliar as conexões entre eles, permitindo a validação do funcionamento da linha de produção de maneira assíncrona, além de possibilitar a detecção de problemas como gargalos nas linhas de produção, convergindo para um melhoramento da produção.

1.1 OBJETIVO GERAL

Criar um sistema composto de um cliente em Javascript e de um servidor em PHP, e um banco de dados PostgreSQL que implemente um sistema de simulação de linha de produção representado por redes de Petri.

1.2 OBJETIVOS ESPECÍFICOS

- Criar uma biblioteca em Javascript para implementação de Redes de Petri em navegadores *web*;
- Validar o simulador implementado;
- Representar um estudo de caso no simulador;
- Analisar os resultados obtidos.

1.3 JUSTIFICATIVA

A motivação para o desenvolvimento deste projeto é melhorar a assertividade dos processos produtivos dentro das fábricas reduzindo desperdícios com tempo e retrabalho.

Para Ohno (1997), toda empresa deve buscar a redução de custos dentro do seu processo produtivo. Reduzir custos está intimamente ligado à redução de desperdícios em qualquer esfera. Uma das maneiras de atingir este controle é

desenvolver um pensamento de eficiência, conseguir produzir mais com menos, fazer o melhor uso possível de suas máquinas e suas instalações.

O sistema a ser desenvolvido neste projeto espera fornecer uma alternativa para o planejamento da produção antes que o bem de consumo seja colocado nesta de maneira física. Atuando principalmente na prevenção de gargalos no decorrer desta, no balanceamento da quantidade de peças produzidas por máquina, no planejamento de tempos de configuração mecânica das máquinas para as próximas ordens de produção utilizando de simulação de linhas de produção.

2 REFERENCIAL TEÓRICO

Para uma discussão ampla sobre o desenvolvimento de um *software* voltado para a área *web* é necessário o conhecimento das estruturas envolvidas que sustentam a aplicação, bem como metodologias e padrões que definem e ditam regras a serem seguidas. Neste capítulo, são discutidas as tecnologias envolvidas no projeto juntamente com seus conceitos e padrões.

2.1 SIMULAÇÃO COMPUTACIONAL

Simulação computacional é uma área de estudos de sistemas reais, permitindo que um *software* imite suas operações e características, podendo fornecer uma prévia de como um sistema se comporta quando exposto a uma série de condições pré estabelecidas (KELTON et al, 1998).

A simulação computacional pode ocorrer como uma linguagem de simulação e como um sistema de alto-nível de simulação. A linguagem de simulação é bastante usada, mas impede o uso de pessoas que não possuem conhecimento técnico sobre programação. O sistema alto-nível de simulação é um *software* que simula sistemas reais e fornece facilidade suficiente para que gestores e controladores o utilizem.

2.2 INTERNET

A utilização da Internet têm crescido ano após ano e possui números expressivos: 40% da população mundial possui uma conexão à mesma. O número de usuários aumentou cerca de dez vezes entre 1995 e 2013, e o número de usuários atingiu três bilhões em 2014 (ILS, 2016, tradução nossa).

Com o crescente número de usuários utilizando os serviços da Internet, algumas mudanças aconteceram na sociedade, entre elas o tempo de comunicação, que passou a ser em tempo real, conectando milhares de usuários do mundo inteiro.

A velocidade de comunicação proporcionou benefícios não somente nessa área, como também alterou a dinâmica de trabalho de diversas empresas ao redor do mundo. Segundo Merkle e Richardson (2000), as aplicações comerciais na Internet surgiram em meados dos anos 80, e desde então, cada vez mais foi notável o número de aquisições de sistemas para o auxílio e controle da produção, informações, entre outros temas.

2.3 SISTEMAS DISTRIBUÍDOS

Segundo Colouris (COULOURIS, 2007, p. 15) um sistema distribuído é definido como: aquele no qual os componentes de *hardware* ou *software* localizados em computadores interligados em rede, se comunicam e coordenam suas ações.

A palavra-chave para compreender a dimensão dos sistemas distribuídos é compartilhamento. Os compartilhamentos disponíveis pela *web* e dispositivos móveis são tão naturais, que nem notamos a dimensão dessas estruturas. Segundo Coulouris (2007, p. 28), a heterogeneidade das estruturas encontram-se nos aspectos de: redes, *hardware* de computadores, sistemas operacionais, linguagens de programação, e a implementação dos diferentes desenvolvedores.

Mesmo com essas diversas heterogeneidades, um sistema distribuído se comporta de forma consistente e transparente para os usuários, durante as transições desses diferentes meios, a camada responsável por mascarar essa heterogeneidade denomina-se *middleware*, que, segundo Coulouris (2007, p. 29), “fornece uma abstração, assim como, o mascaramento da heterogeneidade”. É nessa camada que as conversões de tipo de dados acontece, e que possibilita a comunicação em sistemas heterogêneos.

Um modelo de arquitetura de um sistema distribuído nada mais é, segundo Coulouris (2007, p. 39) do que o da seguinte definição: “da forma pela qual os componentes dos sistemas interagem e a maneira pela qual eles são mapeados em uma rede de computadores subjacente”.

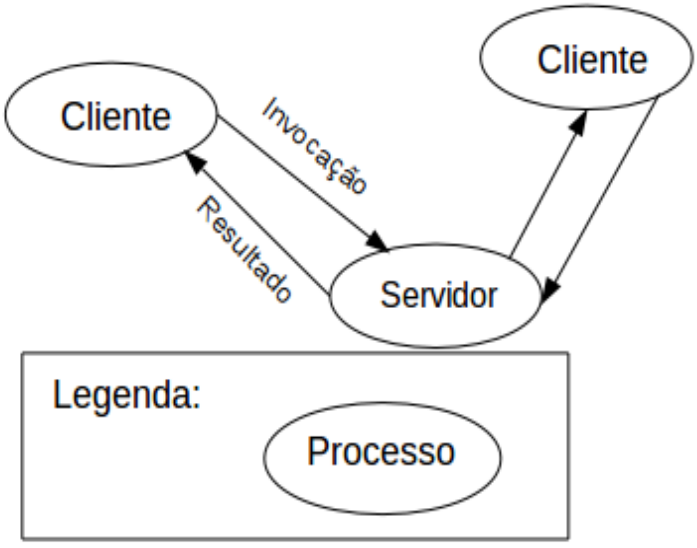
Existem diversos modelos de sistemas distribuídos, que possuem algumas características próprias, entretanto, algumas características são notáveis a todos os

modelos, tais como: tolerância à concorrência, possibilidade de adição ou remoção dos componentes neles envolvidos, existência de um meio seguro durante a repercussão das informações, gerenciamento da escalabilidade conforme a demanda dos acessos ao sistema, transparência aos olhos dos usuários e também tolerância à falhas (COULORIS, 2007, p. 11).

Dentre os modelos existentes, é discutido a seguir o modelo cliente-servidor, que foi o escolhido para a implementação do projeto. Essa arquitetura contém processos clientes interagindo com processos servidores visando o compartilhamento de recursos. Em determinado ponto, servidores podem se tornar clientes devido à dependência de recursos (COULOURIS, 2007, p. 43).

A figura 1 representa de maneira simples como uma arquitetura cliente-servidor trabalha.

Figura 1 – Diagrama representativo de um sistema cliente-servidor



Fonte: Autoria própria, adaptado de Couloris (2007).

2.4 CLIENTE-SERVIDOR

A arquitetura cliente-servidor é um dos estilos mais encontrados para aplicações distribuídas baseadas em redes. Essa arquitetura trabalha com dois elementos, o cliente e servidor. Enquanto o papel do servidor é esperar solicitações de um cliente e retornar um *feedback* para o mesmo, o papel do cliente é enviar requisições através de solicitações a um servidor (FIELDINGS, 2000).

Para melhor gerenciar os recursos compartilhados pelas aplicações, utiliza-se do conceito de separar a demanda de tarefas do servidor para aumentar a escalabilidade da aplicação e as tarefas do cliente, no qual localiza-se a iteração de ações tomadas por usuários. Dessa forma, os componentes podem evoluir independentemente um do outro (FIELDINGS, 2000).

Com essas informações, podemos classificar o servidor denominando-o de *back-end*, onde a parte lógica da requisição é processada, e o cliente como *front-end*, responsável pela interação humano-computador. Essa junção com a adição de um sistema operacional e as infraestruturas envolvidas caracterizam a arquitetura (SALEMI, 1993).

2.5 REST

REST é um acrônimo utilizado para nomear a transferência de estado representacional, é um método de desenvolvimento de *web services* (SAUDATE, 2014, p. 4). Ele é definido por Fieldings (2000) como:

> “A transferência de estado representacional é uma abstração de sistemas arquiteturais contidos em um sistema de hipermídia distribuído. O REST ignora os detalhes da implementação do componente e da sintaxe do protocolo, a fim de se concentrar nos papéis dos componentes, nas restrições em relação à sua interação com outros componentes e na sua interpretação de elementos de dados significativos. Abrange as restrições fundamentais sobre componentes, conectores e dados que definem a base da arquitetura da Web e, portanto, a essência de seu comportamento como uma aplicação baseada em rede” (FIELDINGS, 2000, tradução nossa).

O REST é um sistema baseado no modelo cliente-servidor. Esta definição permite que vários clientes implementados em diferentes linguagens e tecnologias consigam realizar a comunicação com o servidor, simplificando seu desenvolvimento e tornando-o escalável (FIELDINGS, 2000, p. 78).

Fieldings é co-autor do HTTP – *Hypertext Transfer Protocol*, o que torna o REST um modelo arquitetônico guiado pelas melhores práticas do HTTP (SAUDATE, 2014, p. 4):

- Uso adequado de métodos HTTP;
- Uso adequado de URL;
- Uso de códigos padronizados para mensagens de *status* (sucesso ou falha);
- Uso adequado de cabeçalhos HTTP;
- Interligações entre vários recursos.

Recurso é a unidade mínima do REST, está presente em todos os aspectos do protocolo e trata-se do conjunto de dados que trafegam por ele (SAUDATE, 2014, p. 4). Para acessar estes recursos, são utilizados identificadores globais, chamados URI – *Universal Resource Identifier*, que utilizam o protocolo HTTP para realizar esta comunicação. Desta forma cliente e servidor trocam recursos: o cliente envia um recurso que contém uma solicitação para um outro recurso, enquanto o servidor envia o recurso que foi solicitado.

Uma característica importante de um sistema REST é o não armazenamento de nenhum dado entre as requisições. Esta característica é conhecida como *stateless*. O que significa que cada uma das requisições do cliente para o servidor precisa conter toda a informação necessária para compreender a requisição sem depender de uma informação obtida previamente (FIELDINGS, 2000, p. 78). No entanto, esta definição pode reduzir o desempenho por ter de tratar os dados de maneira repetida (FIELDINGS, 2000, p. 78).

2.6 CLIENTE

Nesta seção são expostos os temas relacionados à aplicação cliente do projeto.

2.6.1 HTML

O HTML – *Hypertext Markup Language* – é uma linguagem de marcação de textos e a principal linguagem do conteúdo exibido na Internet (MOZILLA, 2016a). Esta linguagem descreve o documento exibido estruturalmente bem como semanticamente.

O HTML é um padrão internacional definido e mantido pela W3C – *World Wide Web Consortium* (MOZILLA, 2016a).

Para o compartilhamento de arquivos através da Internet, Tim Berners-Lee disponibilizou uma maneira organizada e que se diferenciou da maneira vigente na época. A comunicação era possível apenas com envio de texto puro e a partir deste avanço era possível acessar documentos com um *software* chamado de navegador, facilitando o acesso a arquivos e tornando-os mais ricos, isto é, contendo além de texto, imagens e métodos sofisticados de formatação (MOZILLA, 2016b).

2.6.2 CSS

As CSS – *Cascading Style Sheets* – são arquivos que contém uma linguagem de estilo e são utilizados para estilizar elementos HTML. Em suma, descrevem como um elemento HTML deve ser interpretado.

As regras do CSS possuem duas partes (SCHMITT, 2010, p. 36):

- Seletores: fornecem ao navegador informação sobre quais elementos do documento devem ser estilizados;
- Propriedades: informam ao navegador que propriedade do elemento alterar e que valor devem receber.

Para Robbins (2012), dentre as vantagens do uso do CSS, pode-se destacar uma maior mantenabilidade dos projetos, pois basta realizar uma alteração em um único arquivo CSS para que a aplicação toda receba as mesmas configurações em seus elementos. Outra vantagem é “a habilidade de ajustar a apresentação do documento dependendo do dispositivo de saída“ (MOZILLA, 2016c).

2.6.3 Bootstrap

O Bootstrap é um *framework* para desenvolvimento de *front-end* desenvolvido pela equipe do Twitter[1] em 2011. A justificativa para a criação do Bootstrap foi dada pelo criador Mark Otto em seu artigo de lançamento, no qual descrevia que cada desenvolvedor utilizava uma biblioteca que lhe era mais familiar, onde futuramente gerou problemas de integração, escalabilidade e manutenção das aplicações (SILVA, 2015, p. 21).

O *framework* funciona através de pré-processadores de CSS LESS[2] e SASS[3], que são ferramentas que geram folhas de estilo (CSS). A utilização destas ferramentas permitem a geração de folhas de estilo mais flexíveis do que as folhas de estilo não processadas (SILVA, 2015, p. 21), devido ao uso de variáveis, aninhamento de classes e operadores.

O Bootstrap fornece uma série de classes CSS para utilização rápida, por exemplo, um sistema completo de linhas e colunas para diagramação do conteúdo, formatação de cabeçalhos, botões, modais, ícones, barra de navegação, menus, além de um ecossistemect Metadata do IEEE LTSC) PREMIS (PREservation Metadataa inteiro de componentes javascript para tornar a experiência do usuário satisfatória (BOOTSTRAP, 2016).

1 http://www.twitter.com
2 http://less.org
3 http://sass-lang.com

2.6.4 Angular

Angular é um *framework* utilizado para a construção de aplicativos do lado do cliente, utilizando as linguagens HTML, javascript e também typescript. Neste *framework* é possível escrever aplicações compondo modelos HTML com marcação angularizada escrevendo ect Metadata do IEEE LTSC) PREMIS (PREservation Metadataclasses de componentes para gerenciamento desses modelos (ANGULAR, 2016, tradução nossa).

O principal objetivo do *framework* angular é facilitar o dinamismo da exibição das informações contidas em um documento HTML, com ele, é possível, através de eventos, requisições e até mesmo rotinas programadas, alterar o conteúdo visual que está sendo exibido sem precisar recarregar a aplicação.

Para tal manipulação dos dados, é necessário que o conteúdo dessas rotinas e eventos fique na parte do cliente da aplicação, sendo assim, o navegador interpreta toda a lógica contida no angular. Através de *tags* de marcação é possível construir no HTML elementos desse *framework*, e, a partir dessas tags são interpretado os valores que elas contêm. Caso uma rotina altere o valor de um objeto ou variável do Angular, no mesmo instante o elemento visual marcado é alterado também.

A estrutura do *framework* Angular, segundo a documentação da organização, é considerada uma estrutura Modelo-Visão-Controlador (FREEMAN, 2015), contendo uma arquitetura que possui os seguintes componentes (ANGULAR, 2016, tradução nossa):

Módulo: *Tag* de marcação responsável por modularizar a parte do HTML na qual o *framework* atua.

Componente: É responsável pelo controle de uma *view*. Após a definição da lógica desse componente ele é responsável pela exibição dos dados.

Template: É uma parte do HTML que junto ao componente exibe informações na tela.

Metadata: Dita ao angular como processar uma classe.

Data Binding: Responsável pela ligação da parte funcional e a parte visual da aplicação, mais especificamente, é o que liga as funções com o conteúdo da *view* (ligação entre *templates* e componentes).

Injeção de dependência: É a forma de fornecer instâncias de classes para a aplicação, ou seja, o conteúdo de manipulação de dados.

A empresa Google, utiliza em diversas áreas este *framework*, tais como: Google *Atmosphere*, Google *Carrer*, Google *Education*, entre outras. Muitas empresas também estão começando a usar este *framework*, que é disponibilizado com um código aberto devido a sua simplicidade de uso e também devido ao dinamismo que a biblioteca oferece.

Diferentemente da sua versão anterior, conhecida como AngularJS 1.x, o Angular é construído utilizando o *Typescript*, que é um super conjunto do *javascript* em sua versão ES6 – ECMAScript 6 – (TYPESCRIPT, 2017). Na prática, o *Typescript* é uma adaptação do *javascript* para a produção de *software* em larga escala.

Alguns navegadores não são compatíveis as novas características do ES6. Então surgiu uma nova ferramenta que converte um código gerado com as características da ES6 para a sua versão anterior ES5. Esta ferramenta foi chamada de *transpiler*.

O Angular é constituído basicamente por componentes. Trata-se da menor parte possível dentro de uma aplicação. Segundo Murray et al (2017, p. 12, tradução nossa), é como se fossem ensinadas novas *tags* ao navegador, que possuem funções e comportamento personalizadas.

Um componente é formado por duas partes (MURRAY et al, 2017, p. 12):

- Um decorador do tipo *Component*;
- Uma classe de definição deste componente.

Um decorador é o que fornece semântica a uma classe e informa ao Angular como aquela classe deve se comportar (MURRAY et al, 2017, p. 14).

Além de componentes, uma aplicação contém módulos. Módulos são recipientes para demais componentes, serviços e diretivas. Segundo Fain e Moiseev (2017, p. 32, tradução nossa), um módulo "é uma biblioteca de componentes e

serviços que implementa as funcionalidades do domínio dos negócios". Toda aplicação Angular possui pelo menos um módulo, que é utilizado para dar o início aos demais componentes.

2.7 SERVIDOR

Nesta seção são expostos os temas relacionados à aplicação servidor do projeto.

2.7.1 Laravel

O Laravel[4] é um *framework* PHP de código aberto. Este *framework* fornece suporte a instalação de pacotes externos através do gerenciador de dependências Composer[5].

O ciclo de vida das aplicações construídas com Laravel inicia com todas as requisições sendo recebidas pelo arquivo *index.php* (SILVA, 2015, p. 4). Dentro deste arquivo, é gerada uma instância do componente HTTP *Kernel* que inicializa os demais componentes do Laravel.

O *framework* gera uma instância do HTTP *Kernel*, que é o componente responsável por processar uma imensa lista de inicializadores que terão a responsabilidade de capturar as variáveis de ambiente, executar todas as configurações, *handlers*, *logs*, etc. Tudo isso é feito antes mesmo da requisição ser processada (SILVA, 2015, p. 10).

São também responsabilidades do *Kernel* (SILVA, 2015, p. 10):

- Inicializar os serviços da aplicação;
- Receber requisições HTTP;
- Responder requisições HTTP.

4 http://laravel.com
5 http://getcomposer.org

O Laravel utiliza o conceito de rotas para distribuir as requisições para seus respectivos recursos. As rotas fazem o mapeamento de método e recurso que uma requisição está solicitando (SILVA, 2015, p. 11).

2.7.2 Banco de dados

Para se ter informações armazenadas de forma consistente em um sistema, é indispensável um sistema de armazenamento de dados, mais conhecido como banco de dados. Elmasri e Navathe (2011) definem um banco de dados como:

> “Um banco de dados representa algum aspecto do mundo real, às vezes chamado de mini mundo ou de universo de discurso (UOD - Universe of Discourse). As mudanças no mini mundo são refletidas no banco de dados. Um banco de dados é uma coleção logicamente coerente de dados com algum significado inerente. Uma variedade aleatória de dados não pode ser corretamente chamada de banco de dados. Um banco de dados é projetado, construído e populado com dados para uma finalidade específica. Ele possui um grupo definido de usuários e algumas aplicações previamente concebidas nas quais esses usuários estão interessados” (ELMASRI. NAVATHE, 2011, p. 3).

Um banco de dados é usado para a abstração de informações do mundo real, onde os dados relacionados entre si, formam um conjunto de informações, que podem ser visualizadas, atualizadas, e também deletadas. Esses dados que estão presentes no banco de dados são utilizados no sistema proposto para manipulação de informações.

2.7.3 Banco de dados relacional

Segundo Elmasri e Navathe (2016), o modelo de banco de dados relacional foi introduzido por Ted Codd da empresa IBM em 1970, utilizando relações matemáticas que parecem com uma tabela de valores, baseadas em conjunto e lógica de predicado de primeira ordem.

Este modelo representa o banco de dados baseado em relações, que é representado por uma tabela que contém linhas e colunas. As linhas representam a

abstração de informações do mundo real, e o nome da tabela juntamente com suas colunas são usadas para a interpretação dessa abstração.

A linguagem característica dos bancos de dados relacionais se denomina SQL – *Struct Query Language* – que é uma linguagem de consulta estruturada. A criação da linguagem SQL é de autoria da IBM nos meados dos anos 70, e passou a ser um padrão para os bancos de dados relacionais, segundo Elmasri e Navathe (2016, p. 57).

Ainda segundo Elmasri e Navathe (2016, p. 508), as principais propriedades que garantem que as transações aconteçam e forneçam informações confiáveis neste modelo são as denominadas ACID:

Atomicidade: Dita que uma transação tem que ser executada até o final, ou então não ser executada. Casa haja alguma falha no meio que impossibilite uma transação de ser concluída, o SGBD voltará ao estado inicial antes dessa transação ter alterado qualquer informação.

Consistência: Determina que uma transação deve levar o SGBD de um estado consistente a outro estado consistente.

Isolamento: Dita que uma transação tem que ser executada sem interferência de outras transações.

Durabilidade: Impõe que alterações ocorridas no SGBD devam ser gravadas no banco de dados, e não podem ser perdidas devido à alguma falha.

2.7.4 PostgreSQL

É um Sistema Gerenciador de Banco de Dados (SGBD) que tem a estrutura de um modelo de banco de dados relacional citado anteriormente. Ele possui uma comunidade ativa de desenvolvedores, que melhoram cada vez mais a sua codificação, uma documentação rica e fornece ferramentas robustas para tratar as transações.

Além disso, todos os grandes sistemas operacionais são suportados pelo PostgreSQL (GNU/Linux, Unix (AIX, BSD, HP-UX, SGI IRIX, Mac OS X, Solaris,

Tru64), e MS Windows). Também é um *software* de código aberto (POSTGRESQL, 2016).

2.8 JSON

O método utilizado para alterar informações exibidas no cliente é através da utilização do protocolo HTTP, onde o cliente realiza uma solicitação ao servidor, que responde com uma coleção de dados. Neste projeto, a coleção é devolvida no formato JSON – *Javascript Object Notation*.

O JSON é um subconjunto do javascript, e apesar disso, não se trata de uma linguagem de programação (SMITH, 2015, p. 55). Este subconjunto apresenta uma construção que o torna, de fato, um formato para a troca de dados entre plataformas (SMITH, 2015, p. 55). Este formato veio como alternativa ao XML – *Extensible Markup Language*, este mais estruturado e detalhado (RISCHPATER, 2015, p. 18).

Figura 2 – Um arquivo JSON

```
{
   "piloto": {
      "nome": "Ayrton",
      "sobrenome": "Senna"
   }
}
```

Fonte: Autoria própria

Os dados em um arquivo JSON podem estar organizados em (SMITH, 2015, p. 56):

- Um conjunto chave/valor;
- Uma lista ordenada de valores.

A figura 2 representa uma estrutura básica de um documento JSON. Neste modelo, temos um objeto chamado “piloto”, possuindo as chaves “nome” e “sobrenome”, separados de seus valores através de vírgula.

A norma ECMA-404 é clara no que diz respeito aos valores que um JSON pode assumir. Sejam eles: Objeto, numérico, *string*, *true*, *false* ou *null* (ECMA, 2013).

A escolha deste formato para conduzir os dados deste projeto é devido à simplicidade para ler, escrever e entender dados (IHRIG, 2013, p. 270), além da facilidade de transferí-los entre linguagens diferentes.

2.9 REDES DE PETRI

O conceito de redes de Petri surgiu a partir da tese de doutorado de Carl Adam Petri, intitulada “*Kommunication mit Automaten*” em 1962. O objetivo dessa rede é a modelagem de sistemas componentes concorrentes. Os sistemas distribuídos têm a característica de terem a concorrência entre as transições das informações entre suas estruturas e componentes.

Define-se redes de Petri como uma técnica utilizada para modelagem de sistemas, utilizando como alicerce uma base matemática. Entre as particularidades dessa técnica, é possível realizar a modelagem de sistemas paralelos, concorrentes, assíncronos e não determiníticos (MACIEL, 1996).

A aplicação das redes de Petri permite observar como a concorrência entre determinado estado do processo de fabricação das máquinas simuladas no sistema funcionam, dessa forma levantando uma análise concreta de como os processos estão se comunicando e intercalando a fila de produção industrial.

As redes de Petri como ferramenta gráfica fornecem uma comunicação entre o usuário e o cliente (ZURAWSKI; ZHOU, 1994, p. 541, tradução nossa). Neste sentido, é possível simplificar modelos de dados, ao invés de usar notações matemáticas, ou textos ambíguos. Essa combinação entre conceito e ferramenta gráfica permite colocar engenheiros no controle do processo de modelagem de sistemas complexos. (ZURAWSKI; ZHOU, 1994, p. 541, tradução nossa).

As características das redes de Petri se mostram vantajosas para a modelagem de sistemas devido a vários fatores (CHILE PALOMINO, 1997, p. 2):

- A simplicidade da rede de Petri como ferramenta gráfica de fácil compreensão;
- Permite a verificação de concorrência, disparo de eventos e precedência lógica em um sistema;
- Pode-se decompor uma rede em partes menores (modularidade);
- Funciona como uma linguagem de comunicação entre profissionais de diferentes áreas do conhecimento.

Segundo MARRANGHELLO (2005, p. 7), os elementos básicos para a formação da estrutura topológica das redes de Petri são:

Estados: Através de um conjunto, modelam componentes passivos do sistema. Representados pela forma geométrica elipse.

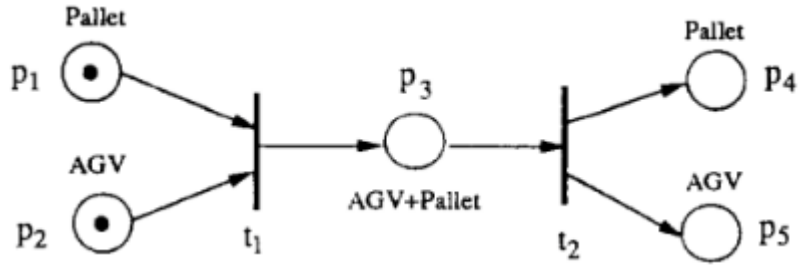
Ações: Através de um conjunto, modelam componentes ativos do sistema. Representadas pela forma geométrica de um retângulo.

Relação de fluxo: Através da união dos conjuntos de estado e ações, é usado para modelar as transições ocorridas no sistema através de ações determinadas. Representada pela figura de uma seta.

Tokens: São utilizados para representar a existência ou não de um estado. Cada lugar pode conter um ou mais *tokens*, representados por pontos. Esses *tokens* permitem modelar a dinâmica do sistema.

A Figura 3 mostra a representação de uma rede de Petri.

Figura 3 – Representação de uma rede de Petri



Fonte: Zurawski e Zhou (1994)

A rede de Petri é identificada como um tipo de grafo composto por 3 objetos: Lugares, transições e arcos, com arcos conectando transições e lugares, bem como lugares e posições. Graficamente, os lugares são representados por círculos e as transições por barras (ZURAWSKI; ZHOU, 1994, p. 569, tradução nossa).

Dentro do conceito de redes de Petri, pode-se classificá-las em graus de abstração (MARRANGHELLO, 2005, p. 5):

Baixo Nível: São as redes de Petri cujo significado de seus *Tokens* não são diferenciáveis a não ser pela estrutura da rede à qual estão associadas, e são subdivididas em elementares e lugar/transição.

Elementares: São as redes de Petri que são extremamente restritivas do ponto de vista da modelagem porque permitem a existência de apenas um *token* em cada elemento da rede.

Lugar/Transição: Não tão restritivas como as elementares, permitem a utilização de mais de um *token* em cada elemento da rede.

Alto Nível: São as redes as quais incorporam uma semântica, viabilizando a diferenciação por meio á atribuição de valores ou cores àos *tokens* permitindo assim a adoção de dados abstratos.

2.10 FABRIC.JS

Fabric.js é uma biblioteca de código aberto escrita em javascript que trabalha com elementos e objetos interativos com *canvas* para HTML5. Essa biblioteca teve início como um editor de *design* para uma loja online, onde a ideia era facilitar a customização de camisetas com imagens e formas vetoriais. Com o sucesso do editor a comunidade de desenvolvedores disponibilizou a biblioteca aos desenvolvedores através da sua conta no GitHub (FABRICJS,2016).

Na linguagem HTML5 existe uma *tag* de marcação denominada *canvas*, essa *tag*, na estrutura HTML por definição:

> "Adicionado ao HTML5, o elemento HTML <canvas> é um elemento que pode ser usado para desenhar gráficos via código (nomalmente JavaScript). Por exemplo, ele pode ser usado para desenhar gráficos, fazer composição de fotos, criar animações ou até mesmo fazer processamento ou renderização de vídeo em tempo real (MOZILLA, 2017)."

A biblioteca do Fabric.js, proporciona uma manipulação do elemento *canvas.* Todas as funcionalidades disponíveis na biblioteca são exclusivamente para a manipulação dessa *tag* (FABRICJS, 2016*)*.

Na biblioteca está pré determinada a criação de formas geométricas simples, como: retângulos, linhas, círculos. Também é possível dimensionar (ângulo de rotação de zero até trezentos e sessenta graus), passando uma largura e altura ou então um tamanho de raio (para círculos), isso determinada a posição do objeto no elemento *canvas*. É possível também realizar animações de objetos criados no canvas, alterar a cor dos objetos, e também pré determinar manipulações e modificações nos objetos vinculados a um ou mais eventos (FABRICJS, 2016).

Para definir um elemento *canvas* na linguagem HTML5 e trabalhar com a biblioteca Fabric.js, deve-se declarar a *tag canvas* pré determinando sua dimensão de altura (*height*) e sua dimensão de largura (*width*) e atribuindo um "id" para a *tag*. A figura 4 demonstra a criação de um elemento *canvas* em um documento HTML.

Figura 4 – Declaração de um elemento canvas no HTML

```
<canvas id="canvasSimulator" width="1000" heigth="1000"></can
```

Fonte: Autoria própria

Após essa declaração inicial, pode-se declarar uma variável do tipo *fabric.Canvas*, recuperando a *tag* de marcação do *canvas* pelo seu "id". A figura 5 exemplifica como é a declaração de uma variável do tipo *canvas*.

Figura 5 – Declaração de uma variável canvas no Angular

```
this.canvas = new fabric.Canvas('canvasSimulator');
```

Fonte: Autoria própria

Feita a declaração da variável *canvas*, pode-se criar variáveis que são objetos pertencentes à biblioteca Fabric.js. A Figura 6 exemplifica a criação de um elemento SVG com a forma de um quadrado.

Figura 6 – Criação de um quadrado

```
var square = new fabric.Rect({
    width: 35,
    heigth:35,
    left:500,
    top:500,
    stroke: '#000',
    strokeWidth: 2,
    fill: '#fff',
    selectable:false
});
```

Fonte: Autoria própria

Na Figura 6, foi criado um elemento de altura e largura de 35 *pixels*, na posição de altura 500 *pixels* a esquerda e 500 *pixels* abaixo da posição inicial do *canvas*. Após criação do objeto, para visualizar o mesmo no *canvas*, é necessário realizar a adição dele utilizando: *this.canvas.add(square)*. Como resultado, é incluído um desenho de um polígono no *canvas*.

3 METODOLOGIA

3.1 OBJETO DE ESTUDO

Para colocar à prova o simulador desenvolvido, foram coletados dados acerca de uma linha de produção de uma indústria metalúrgica de Ponta Grossa, com foco em produção de peças para máquinas madeireiras com predominância de usinagem.

A usinagem é um processo de fabricação mecânica que consiste na remoção de material através da ação de uma ferramenta de corte (CHIAVERINI, 1986, p. 194). A quantidade de operações de usinagem e ferramentas de corte são de uma variedade bastante grande (CHIAVERINI, 1986, p. 194).

A linha de produção analisada é a fabricação de um guia de apoio para a lâmina de uma serra de fita, utilizada nas máquinas de corte de madeira. O guia de apoio é uma peça composta por dez componentes diferentes. Três deles são fabricados diretamente na unidade fabril desta indústria: um rolete, um eixo e uma tampa.

Foram analisados os processos produtivos das peças fabricadas diretamente pela indústria do estudo, não considerando as peças que são produzidas por terceiros.

3.2 SISTEMAS SIMILARES

De alguma maneira, todos os *softwares* pesquisados apresentaram pontos em comum com a ideia deste trabalho. Pontos como simulação de uma linha de produção, modelagem de uma fábrica em 3D, redução de tempo e controle de informações agrupados em um só programa, que já é uma realidade. Não existe, porém, um aprofundamento extenso acerca de suas funcionalidades, pois se tratam de programas de corporações privadas e muitas vezes sob demanda.

Dentre os *softwares* pesquisados, destacam-se: See-the-future[6], Arena[7] e DELMIA[8].

O Quadro 1 exibe um comparativo entre as ferramentas de estudo similares.

Quadro 1 – Quadro comparativo entre ferramentas similares

Software	**Fabricante**	**Características**
See-the-future	Trilha	Descreve o roteiro de produção de cada produto acabado ou peça; Gera relatórios da simulação; Sistema desenvolvido para clientes com suas particularidades (sob demanda);
Arena	Rockwell Automation	Simulação em 3D com gráficos baseados em jogos; Painel em tempo real apresentando informações; Planejamento condicionais representando uma linha de produção;
DELMIA	Dassault Systèmes	Com o *software* é possível planejar e testar novos métodos dentro de um ambiente de produção simulado; Operação em tempo real, podendo realizar agendamentos de mudança na fábrica;

Fonte: Autoria própria

3.3 RESTRIÇÕES DO PROJETO

Este projeto fornece uma simulação de uma linha de produção funcionando com uma ordem de produção.

> "Uma ordem de produção é o documento que dá início aos processos de fabricação dos produtos, sendo essencial para gerar as requisições dos materiais e considerando todos os componentes necessários e as fases para a fabricação determinadas pela sua estrutura (NOMUS, 2016)".

No *software*, não é possível visualizar a produção de uma fábrica com diversas ordens de produção acontecendo ao mesmo tempo. Diferentemente dos sistemas similares revisados na seção 3.2, o sistema não exibe gráficos em três dimensões.

O *software* não calcula nenhuma informação relativa a custo com matérias-primas, tampouco com funcionários ou faz referência à gastos monetários diretamente.

6 http://www.trilhaprojetos.com.br
7 http://www.paragon.com.br/softwares/arena/
8 https://www.3ds.com/products-services/delmia/

Este projeto não possui armazenamentos de senha e nem autenticação de qualquer natureza que possa ser validada pelo usuário.

Quanto às requisições, estas são apenas aceitas via protocolo HTTP. O envio e recebimento de informações se dá apenas em arquivos JSON, não aceitando-se requisições com conteúdo em XML.

Operacionalmente, não é possível alterar as redes de Petri diretamente através de cliques. Quaisquer alterações no desenho devem ser precedidos de alterações nos painéis de cadastro da linha. Também não é possível gerar uma simulação sem informações inseridas pelo usuário, tais como máquinas, processos produtivos e produtos.

Este projeto é preferencialmente executado no navegador de um computador, não sendo recomendado o uso em *smartphones* para uma melhor visualização da rede na totalidade.

O *software* não contabiliza o tempo gasto pelo funcionário em atividades que não fazem parte da rede, como por exemplo, emergências, deslocamento de funcionário para outra tarefa ou necessidades naturais.

4 DESENVOLVIMENTO

Este capítulo expõe os detalhes da implementação do projeto do simulador. Trata também das dificuldades encontradas durante o desenvolvimento e finaliza com uma discussão acerca dos resultados obtidos.

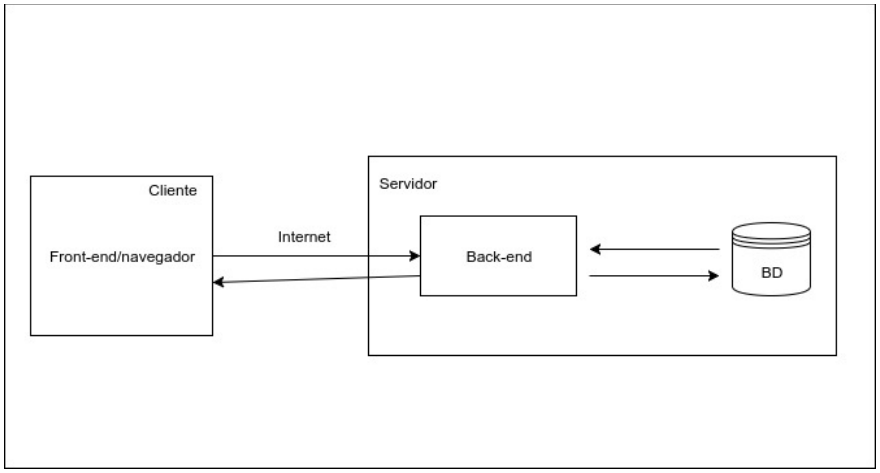
4.1 SISTEMA CONSTRUÍDO

Para a realização deste projeto, foram definidos alguns requisitos mínimos que nortearam as escolhas. Primeiramente, era preciso que o sistema tivesse seu funcionamento na Internet para que fosse acessível de qualquer lugar. Em segundo lugar, era necessário desenvolver em uma tecnologia mais recente para ter acesso a recursos mais novos dos navegadores.

Pensando no futuro, o *software desenvolvido* foi dividido em dois programas menores, funcionando de maneira distribuída:

- Uma *software* que fornece acesso visual aos formulários para cadastros menores e operações em geral através de um navegador - o *front-end* ;
- Uma plataforma que se comunica diretamente com o banco de dados, processa as requisições que são enviadas pelo *front-end* e as formata, sem que o usuário possua acesso diretamente ao banco de dados - o *back-end*;

Figura 7 – Esquema de comunicação entre cliente e servidor



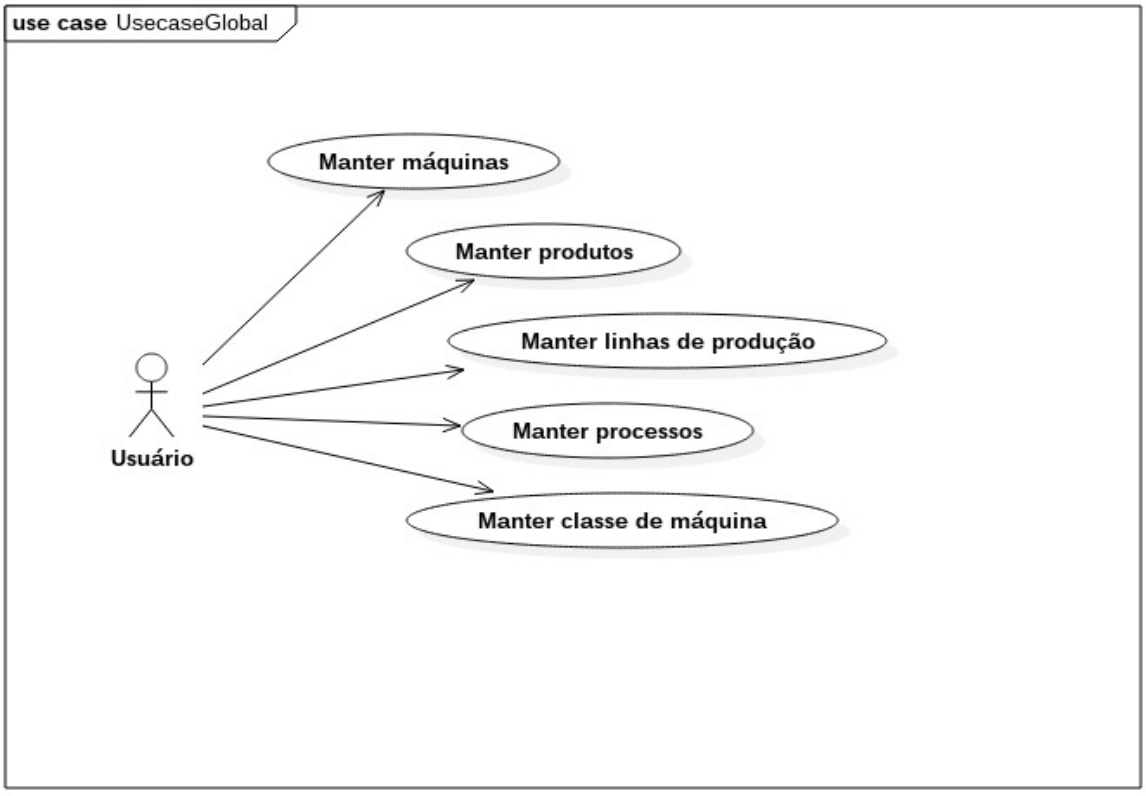
Fonte: Autoria própria

A figura 7 ilustra o funcionamento das requisições entre o cliente e o servidor. O cliente faz a requisição diretamente ao *back-end.* O *back-end,* então, faz a consulta ao banco e devolve um resultado válido para que o cliente o interprete.

Esta separação permite que em um determinado momento, quando o projeto estiver estabilizado, possam ser inseridas novas maneiras de acesso utilizando o mesmo banco de dados. Por exemplo, o desenvolvimento de um aplicativo ou uma interface que possa ser iniciada como um *software* executado nativamente dentro de um sistema operacional.

Quanto às funcionalidades do projeto, a Figura 8 apresenta um diagrama de caso de uso global do projeto.

Figura 8 – Diagrama de caso de uso global



Fonte: Autoria própria

Os casos de uso são:

Manter máquinas: A possibilidade do usuário realizar a manutenção dos registros das suas máquinas, fornecendo um nome, uma marca e um modelo;

Manter produtos: A possibilidade do usuário realizar a manutenção dos registros dos seus produtos (internos ou externos), fornecendo o nome do produto, descrição, comprimento, altura, largura e peso;

Manter linhas de produção: A possibilidade do usuário realizar a manutenção de suas linhas de produção, onde o usuário cadastra as etapas que fazem parte desta linha, bem como as máquinas que fazem parte desta etapa;

Manter processos: A possibilidade do usuário realizar a manutenção dos registros dos seus processos produtivos (por exemplo, o torneamento de um eixo), fornecendo um nome e uma descrição;

Manter classe de máquina: A possibilidade do usuário realizar a manutenção dos registros uma classe para a sua máquina (por exemplo, uma furadeira pode pertencer a classe ferramenta).

Para o desenvolvimento das interfaces e suas respectivas funcionalidades, foram utilizados o sistema operacional Ubuntu 16.04[9] LTS – *Long Term Support*, sendo este um sistema operacional de código aberto construído utilizando-se um núcleo Linux, sendo o mesmo derivado do sistema operacional Debian[10].

Para a criação dos arquivos de código fonte, foram utilizados os editores de texto Atom[11] e Visual Studio Code[12], ambos gratúitos e de código aberto. Para o teste das funcionalidades implementadas, foi utilizado o navegador Google Chrome[13].

A seguir, serão levantados os detalhes da implementação do servidor e do cliente, bem como a sua integração e a montagem da simulação.

4.2 SERVIDOR

De maneira geral, a aplicação *back-end* é alocada diretamente em um servidor da Internet, deixando as rotas de acesso aos recursos disponível para uso.

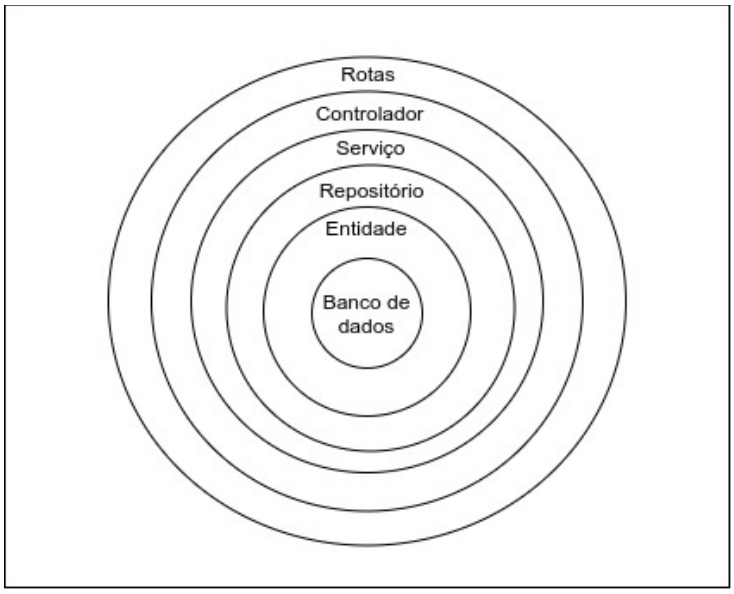
As rotas funcionam como porta de entrada para as funções de inclusão, atualização, remoção e leitura. Através delas, as aplicações direcionam as

9 https://www.ubuntu.com
10 https://www.debian.org/index.pt.html
11 https://atom.io
12 https://code.visualstudio.com/
13 https://www.google.com.br/chrome/browser/desktop/index.html

requisições entre as classes do *software* até que as informações necessárias sejam operadas no banco de dados.

A estrutura desenvolvida funciona através de camadas. A figura 9 mostra um esquema geral da estrutura do projeto. Para acessar o banco de dados, é obrigatório que requisição seja repassada por todas as camadas, iniciando pelo arquivo de rotas. O arquivo de rotas faz o uso da classe *Route* do *framework*, que permite registrar as rotas e as deixando disponível para qualquer função do HTTP.

Figura 9 – Estrutura do back-end em camadas



Fonte: Autoria própria

Utilizando os métodos fornecidos pelo *Route*, são passados dois parâmetros: $uri e $callback. O primeiro parâmetro é o ponto de acesso da requisição a partir da URI do servidor. O segundo é a função a ser executada com o valor recebido pela URI.

No caso deste projeto, o primeiro parâmetro é o recurso que deve ser acessado e o segundo é o caminho para a função que deve ser executada no controlador quando os parâmetros da rota forem atendidos.

Para realizar uma consulta e resgatar todas as máquinas, o cliente realizará, em ambiente local, uma requisição para *http://localhost/maquinas*. Quando o arquivo de rotas identificar o verbo presente na requisição, encaminhará os parâmetros para

o seu controlador *MaquinaController*, apontando em seguida qual método do controlador o sistema deverá executar. A função deve vir precedida de um sinal de arroba (@). Então tem-se:

Route::get(‘maquinas’, ‘MaquinaController@lista’).

O controlador tem seu funcionamento baseado no direcionamento das informações para a camada de serviços. Esta classe recebe a requisição completa e condiciona a um objeto que pode ser operado dentro do *back-end*. Observando o segundo parâmetro para *Route::get(‘maquinas’, ‘MaquinaController@lista’)* no exemplo supracitado, tem-se o nome da classe do controlador *MaquinaController* e o nome de uma função presente na classe, neste caso ‘lista’.

Dentro da função lista, é onde ocorre o apontamento mapeado para a classe de serviço. Neste caso, é realizada uma chamada para um método também existente na classe *MaquinaService*. Quando a função do controlador termina de processar o valor do retorno, este retorno é dado já convertido em JSON.

A camada de serviços é responsável por implementar as regras de negócio da aplicação. Nesta camada são realizados os tratamentos de exceção, validação de campos importantes vazios, e tratamento de mensagens de alto nível para o usuário. Sendo possível também construir consultas mais complexas em que não seja possível utilizar os assistentes de mapeamento de objetos através da camada de repositórios.

A camada de repositório é a camada responsável por acessar os assistentes de mapeamento de objetos (ORM – *Object Relational Mapper*). O *framework* Laravel fornece por padrão o seu ORM Eloquent[14]. O mapeamento de objetos realiza a representação das tabela do banco de dados em forma de orientação a objetos e é realizada de maneira automática quando utiliza-se o Laravel e o Eloquent.

Para que a classe de repositório saiba qual é a tabela que deve operar, é criada uma classe auxiliar chamada Entidade. Esta classe possui o papel de trabalhar com o Eloquent para facilitar a tarefa do desenvolvedor em criar funções básicas de inserção, edição, remoção e leitura de dados. As entidades possuem atributos básicos:

14 https://laravel.com/docs/5.3/eloquent

- $*table*: atributo que informa qual a tabela no banco que é espelhada na classe;
- $*primaryKey*: informa a chave primária da tabela;
- $*fillable*: este atributo informa quais os campos poderão receber valor ao realizar operações de manipulação de dados.

4.3 CLIENTE

O cliente possui dois modos de execução: *modo desenvolvimento* e *modo produção*. O modo desenvolvimento é iniciado quando o sistema é executado localmente através do comando *ng serve*, enquanto o modo produção é utilizado quando a codificação está pronta. Em seguida o código-fonte é compilado em javascript puro utilizando-se do comando *ng build.*

A aplicação cliente tem seu ponto de acesso no arquivo *index.html*, dentro do diretório *src*. Este diretório contém armazenados os códigos-fonte que são exclusivos da aplicação, isto é, os arquivos feitos para este trabalho bem como os arquivos de configuração do *framework*.

O diretório *src* fornece acesso ao diretório *app*, que contém dois dos principais arquivos da aplicação: *app.module.ts* e *app.component.ts*. Estes são os arquivos que possuem as configurações mínimas do projeto.

No arquivo *app.module.ts* são inseridos os módulos para que o servidor do cliente consiga dar o início à aplicação. Todas as dependências que são globais devem estar inseridas nestes arquivos.

O arquivo *app.component.ts* armazena a classe que é executada primeiro e que pode ter inseridos comportamentos do arranque. Em alguns casos, são inseridas opções de autenticação nesta classe, fazendo o roteamento de usuários que já estão autenticados e direcionando o usuário que não está para a página de *login*.

Um dos arquivos mais importantes do projeto é o arquivo *app-http.service.ts*. Este arquivo contém as configurações válidas e compatíveis do cabeçalho para que o servidor aceite as requisições. Esta classe presta um serviço global para todas as

outras classes que se utilizam das requisições, uma vez que as informações do cabeçalho são compartilhadas entre elas.

Outra função de extrema importância é a função *builder*() desta mesma classe. Esta função faz a montagem da URI do servidor com o recurso a ser acessado que está disponível através de uma rota no servidor. O *apêndice A* contém o arquivo de serviço na íntegra.

Dentro do mesmo diretório *src*, foram criados diretórios para armazenar cada uma das páginas a serem exibidas dentro do navegador, bem como os componentes auxiliares. Em geral, cada um dos diretórios contém os arquivos:

- <*classe*>.*component.ts*
- <*classe*>.*component.html*
- <*classe*>.*component.scss*
- <classe>.module.ts

Assim como a classe *AppComponent* criada pelo arquivo *app.component.ts*, cada uma destes módulos tem seu comportamento similar. Uma vez instanciada a classe através das rotas, cada uma pode ter seu comportamento individual diferente entre si.

Do ponto de vista funcional, a classe *app.component.ts* faz com que o usuário inicie a navegação na aplicação na página principal, que serve como um guia para o usuário conseguir acessar as demais funções do sistema. Cada uma das telas podem ser acessadas utilizando o menu superior, que contém os *links* necessários para a navegação.

O arquivo que possui a lógica do processo de navegação é o arquivo de módulo de qualquer uma das páginas. Este arquivo possui acesso a uma constante definida para o projeto como *appRoutes*. Esta constante é do tipo *Routes* e permite que haja uma lista de objetos de configuração da rota, onde cada um dos objetos dá acesso aos atributos *path* e *component*, que recebem, respectivamente, o caminho e a classe que deve ser iniciada.

Primariamente, cada um das telas é iniciada com uma lista dos objetos cadastrados para aquela classe, com acesso aos botões de cadastro, edição e remoção. Como um exemplo básico, a tela de produtos é iniciada com uma tabela

de produtos, possuindo, logo acima um botão para o cadastro. Em cada uma das linhas da tabela há *links* para que o usuário tenha acesso à edição e remoção de um registro em específico.

A Figura 10 exibe a interface de lista de produtos.

Figura 10 – Interface de produtos

Fonte: Autoria própria

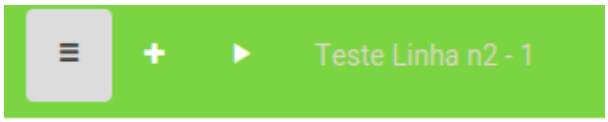
O mesmo padrão é adotado para todas as demais interfaces que possuem acesso a um formulário de cadastro.

A interface que não obedece aos padrões convencionais supracitados é a interface de montagem da linha de produção. Esta interface permite que um usuário selecione uma máquina, representada como um cartão, dentre uma lista de máquinas, clique nela e arraste para dentro de uma área destinada à etapa de produção.

A interface inicia realizando o carregamento das máquinas e das linhas de produção pré cadastradas em listas que ficam acessíveis através de painéis tabulados, possíveis de serem acessados através do botão de menu localizado no canto superior esquerdo. Ao realizar o clique neste botão, toda a interface é deslocada para a direita, revelando assim as listas de máquinas. A Figura 11 exibe parte da barra de navegação com acesso ao menu.

Através do botão de inclusão caracterizado pelo sinal de mais, é possível inserir novas etapas, necessitando somente de um nome e um processo, previamente cadastrado.

Figura 11 – Ponto de acesso ao menu



Fonte: Autoria própria

A Figura 12 mostra a lista de máquinas que podem ser arrastadas para as etapas.

Figura 12 – Lista de máquinas em exibição

Fonte: Autoria própria

A figura 13 mostra algumas etapas cadastradas em ordem temporal. No simulador, representa uma etapa sucedendo outra. No momento mostrado, existe uma divisória dentro do espaço em branco da coluna da etapa, pronta para receber uma máquina da lista de máquinas.

Figura 13 – Etapas inseridas na interface

Fonte: Autoria própria

O procedimento de inclusão de máquinas nas etapas é executado quando o usuário solta o cartão sobre a lista. Neste caso, o evento *soltar* do cartão dispara a requisição para o *back-end*, enviando os identificadores da linha e da etapa, bem como a ordem das máquinas dentro das suas etapas.

Ainda no menu superior, é possível utilizar o botão de iniciar. Este botão dá início à execução da montagem da simulação.

4.4 CONSTRUÇÃO DA REDE DE PETRI

Para desenhar uma simulação das linhas de produção de um produto final para uma metalúrgica, na qual vários processos paralelos ocorrem e são dependentes entre si, foi construído um *framework.* Este permite gerar, calcular e desenhar uma rede de Petri vinculando as máquinas utilizadas para produção e também os produtos gerados em cada etapa.

Para a construção do *framework* foi essencial a biblioteca do *Fabric.js. C*om base nos componentes oferecidos pela biblioteca, foi realizada a divisão das funcionalidades:

1) Criação dos componentes visuais.
2) Cálculo da posição de cada elemento feito em *pixels*.

3) Mapeamento da rede de Petri organizando as posições na quais os *tokens* atuam.

4) Animação dos *tokens*.

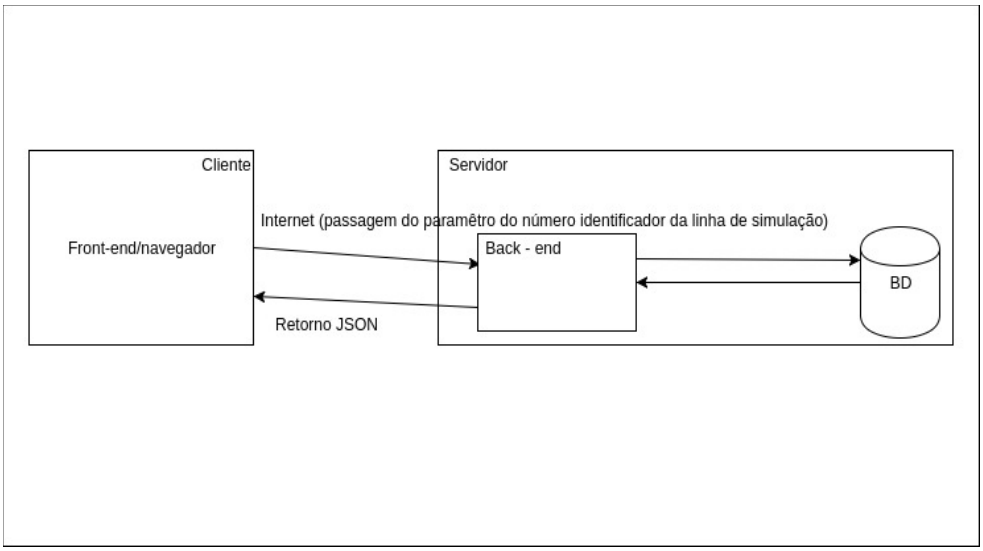
5) Exibição dinâmica dos movimentos dos *tokens.*

6) Exibição dinâmica das informações do produto fabricado e tempo gasto para cada etapa.

7) Exibição do tempo final gasto da linha de produção na totalidade.

Para atender as necessidades das funcionalidades acima, primeiramente é feita uma requisição ao *back-end* para obter os dados referente a simulação completa, passando como parâmetro o número identificador da linha. A Figura 14 exibe o esquema de envio e recepção de dados entre cliente e o servidor.

Figura 14 – Passagem de parâmetro do cliente para o servidor



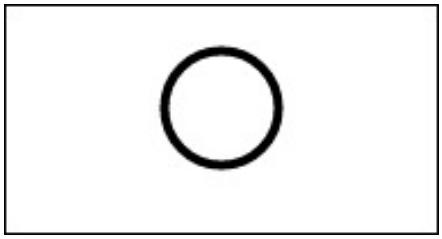
Fonte: Autoria própria

Após o retorno ao *front-end* com os dados no formato JSON, tem-se o seguinte processamento: A rota indicada pelo navegador chamará o recurso de um *provider* que retornará os dados do JSON. Paralelo à requisição existe a chamada do diretório “novo-simulador” (responsável pelos itens 4, 5, 6 e 7) que irá realizar chamadas para o diretório *factories* (responsável pelos itens 1, 2 e 3).

O diretório *factories* foi incluído no projeto apenas para trabalhar com a criação e manipulação dos objetos da biblioteca *Fabric.js*. Existem duas classes dentro desse diretório: *canvas.factory.ts* e *calculated-positions-factory.ts*. Enquanto a primeira classe contém os métodos para a criação dos objetos geométricos, a segunda classe fornece métodos de cálculo automático das posições dos objetos dentro do *canvas*. Para representar elementos de uma rede de Petri, foi pré determinado a exibição visual de alguns componentes da rede:

Início/fim de uma rede de Petri: originalmente esse círculo não existe em uma rede de Petri. Foi inserido para marcar o início/fim da simulação. A Figura 15 mostra a representação deste marcador.

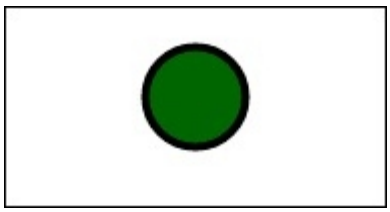
Figura 15 – Marcador de inicio e fim



Fonte: Autoria própria, adaptado de ZURAWSKI; ZHOU, 1994

Tokens: Representação feita através de um círculo verde. A Figura 16 mostra a representação deste marcador.

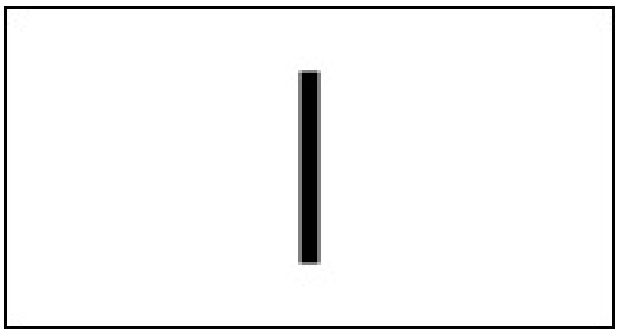
Figura 16 – Representação do Token



Fonte: Autoria própria, adaptado de ZURAWSKI; ZHOU, 1994

Triggers: Representação através de linhas verticais pretas, conforme a quantidade de níveis da simulação cresce, seu tamanho cresce também, para manter o alinhamento das setas fiel a uma rede de Petri. É a representação de uma transição na rede. A Figura 17 mostra a representação deste marcador.

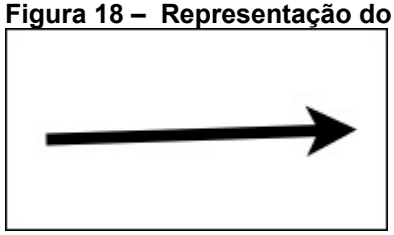
Figura 17 – Representação da transição (trigger)



Fonte: Autoria própria, adaptado de ZURAWSKI; ZHOU, 1994

Setas: Representam um arco de passagem entre um processo e outro na rede. As setas sempre estarão representadas na horizontal, ou seja, não há ângulo de inclinação para a representação. É a representação dos arcos da rede de Petri. A Figura 18 mostra a representação deste marcador.

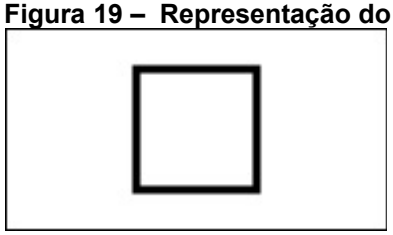
Figura 18 – Representação do arco



Fonte: Autoria própria, adaptado de ZURAWSKI; ZHOU, 1994

Quadrados: É a representação de lugar na rede de Petri. Representam as máquinas executando os processos cadastrados, a ideia da representação do quadrado é aproximar o cadastro visual das interface das linhas com a representação. A Figura 19 mostra a representação deste marcador.

Figura 19 – Representação do lugar



Fonte: Autoria própria, adaptado de ZURAWSKI; ZHOU, 1994

A classe *calculated-positions.factory.ts* é responsável pela leitura do objeto JSON. Conforme a leitura ocorre, a mesma calcula as posições dos elementos e realiza uma chamada para a classe *canvas.factory.ts* passando a posição do objeto como parâmetro e recebendo como retorno o objeto em si, adicionando o mesmo no *canvas*.

A lógica para exibição dinâmica dos *tokens* e também das informações dinâmicas exibidas, encontra-se no diretório “novo-simulador”, que a partir do mapa das posições dos objetos encontrados no canvas, realiza o cálculo das posições dos *tokens* e das informações.

O resultado final desse desenvolvimento é de uma rede representando a execução dos processos das máquinas onde é visualmente exibido cada tempo de cada linha de produção e do processo.

4.5 RESULTADOS OBTIDOS

O estudo de caso realizado para a análise dos resultados foi efetuado com base em linhas de produção da peça guia de apoio a uma lâmina de serra de fita. As informações apresentadas neste processamento são reais.

A simulação acontece com quatro linhas de produção, sendo elas:

1) Produção do Rolete. É realizada a produção de 136 peças, nessa linha existem seis etapas de produção, que são:
   a) Serra da barra de ferro em barras de 45 milímetros. A cada cinco minutos é produzida uma peça, contendo um tempo total de etapa de onze horas e vinte minutos.
   b) *Setup* do comando numérico central. Nessa etapa é feita a configuração da máquina, e gasta um tempo total de uma hora e meia.
   c) Torneamento utilizando o comando numérico central. Nessa etapa, o tempo total de conclusão para 136 peças é de doze horas.
   d) Tratamento térmico. Essa etapa é processada em um local externo da empresa. Gasta um tempo total, contando transporte, recebimento etc, de cinco dias, ou seja, cento e vinte horas.
   e) *Setup* do comando numérico central. Nessa etapa é feita a configuração da máquina, e gasta um tempo total de uma hora.
   f) Retífica do rolete. Essa etapa gasta seis horas para ser concluída.

2) Produção do Eixo. Também é realizada a produção de 136 peças, contando com seis etapas de produção, que são:
   a) Serra da barra de ferro em barras de mil milímetros, na qual se gasta um tempo total de processamento de quarenta minutos.
   b) *Setup* do comando numérico central para torneamento do eixo. Essa etapa demora um total de uma hora e meia.
   c) Torneamento da tampa. Tem um tempo de conclusão total de dez horas.
   d) Furação da tampa. Essa etapa é concluída em duas horas.
   e) Rosqueamento do eixo. Nessa etapa o tempo de conclusão é de duas horas.
   f) Zincagem do eixo. Dois dias é o tempo de conclusão dessa etapa, ou seja, vinte e quatro horas.

3) Produção da Tampa.

a) Serra barra vinte milímetros. Essa etapa tem um tempo total de quinze minutos para ser concluída.
b) *Setup* do comando numérico central para torneamento da tampa. Nessa etapa o tempo de configuração da máquina gasto é de uma hora.
c) Torneamento da tampa. Tem um tempo de conclusão total de quatro horas e trinta minutos.
d) Zincagem da tampa. Nessa etapa o tempo total de conclusão é de dois dias, ou seja, quarenta e oito horas.

4) Montagem da peça. Essa linha possui apenas uma etapa, que conta com a mão-de-obra humana. Geralmente é apenas um operador que realiza a montagem. Com um operador o tempo total gasto é dezoito horas.

Inicialmente, pode-se verificar que existem etapas paralelas que dependem de uma mesma máquina para realizar a execução, o que causa um tempo de espera e acaba atrasando a linha de produção como um todo. Nesse caso, cada linha tem um tempo total de execução, que se efetuando os cálculos são:

Produção do Rolete: 138 horas e 50 minutos.

Produção do Eixo: 64 horas e 10 minutos.

Produção da Tampa: 53 horas e 45 minutos

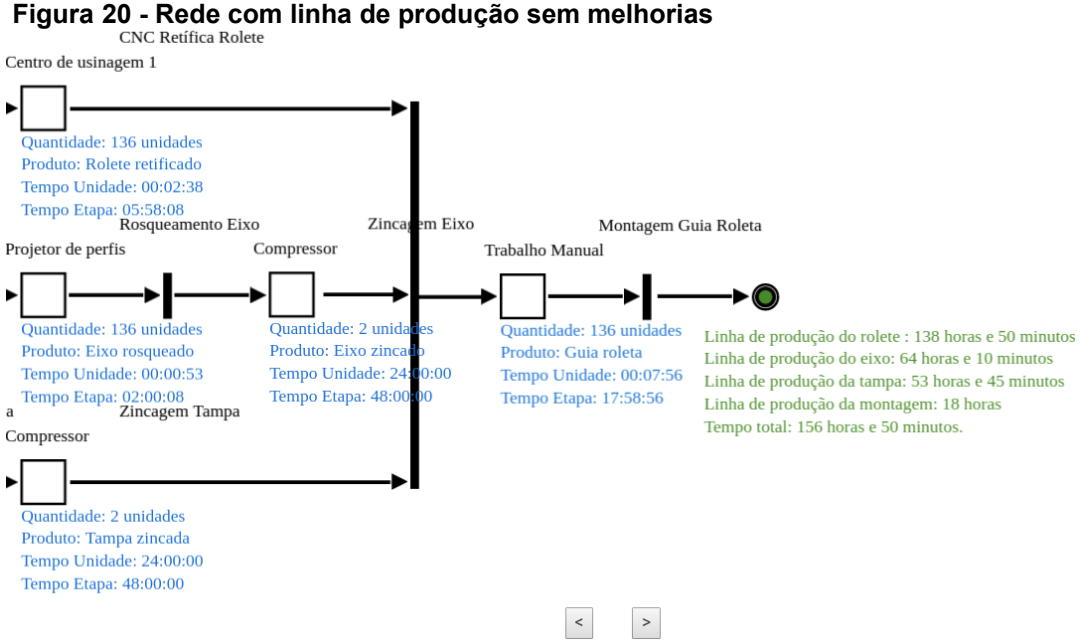
Montagem da peça: 18 horas.

O tempo gasto nesses processamento não são somados diretamente, uma vez que eles acontecem em paralelo A determinação de um tempo total das três linhas iniciais seria o maior tempo entre elas, que no caso seria a linha de produção do rolete. Para obter o tempo total da simulação, usa-se o tempo das três linhas mais o tempo da quarta linha de produção, o que resultaria em um total de 156 horas e 50 minutos.

A primeira melhoria a ser notada está relacionada com a etapa do trabalho manual, uma vez que se fosse dividida entre dois trabalhadores o tempo da linha de produção passaria a ser de nove horas, levando a um resultado total de 147 horas e 50 minutos.

A identificação do tempo na qual um processo não é executado, pois depende de uma máquina que está sendo utilizada, é denominado de *gargalo* nas fábricas. Com a montagem da simulação, é notável onde esse tempo de gargalo ocorre, facilitando a identificação de um problema. Pode-se ainda gerar outras simulações com outras posições, obtendo um resultado com menos gargalos. A Figura 20 exibe um fragmento da linha em que não houveram quaisquer melhorias.

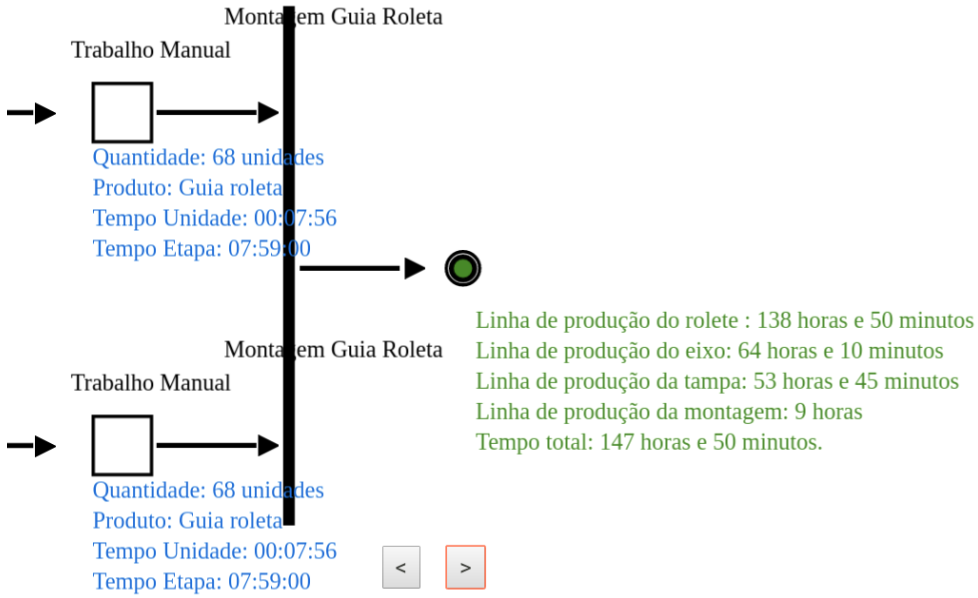
Figura 20 - Rede com linha de produção sem melhorias



Fonte: Autoria própria

No caso analisado, foi conseguida uma melhoria de nove horas para toda a produção. A Figura 21 demonstra um fragmento da rede montada com a melhoria em foco. A parte mais demorada dessa simulação é a dependência de um processo externo de produção (zincagem), e com a análise pode ser cogitada a mudança desse processo externo para a execução na própria fábrica. Porém, nem sempre isso é possível. A ilustração deste modelo está contido no apêndice B.

Figura 21 - Rede com linha de produção alterada



Fonte: Autoria própria

5 DISCUSSÕES FINAIS

Nesta seção são tratadas as discussões finais acerca do desenvolvimento deste projeto, bem como o prosseguimento do projeto em trabalhos futuros.

5.1 CONSIDERAÇÕES FINAIS

Conhecer a própria fábrica e utilizar simuladores como o desenvolvido neste trabalho para melhorar o processo, aumentando a assertividade do tempo de produção é um diferencial em um mercado onde há uma disputa intensa entre diversas empresas por determinadas fatias de mercado.

Este trabalho demonstra parte do potencial da utilização da rede de Petri para o planejamento de produção. Planejamento este que é de suma importância para a indústria que deseja lucrar mais e ter um produto mais competitivo.

Como ferramenta visual, a rede de Petri deu visibilidade a problemas nas linhas de produção que antes só estavam no mundo das ideias ou não eram documentados até então. Não é possível buscar uma melhoria sem antes identificar o que precisa ser melhorado.

Neste sentido, percebe-se que a rede de Petri é uma ferramenta valiosa na identificação de problemas relacionados ao desperdício de tempo, uma vez que pode-se verificar o atraso de determinadas etapas de produção, quando causado por outra etapa que necessita de um recurso que está sendo utilizado. Isso permite que os gestores das fábricas realizem um planejamento para minimizar os impactos dessa concorrência entre recursos.

5.2 TRABALHOS FUTUROS

Os próximos passos para este projeto se tornar uma ferramenta robusta é a adoção de novas funcionalidades que o aproximem do dia-a-dia das indústrias e o tornem uma ferramenta imprescindível dentro das fábricas, sejam elas de pequeno, médio ou grande porte.

Dentre as funcionalidades que podem ser implementadas, estão a possibilidade de representar mais linhas de produção sendo executadas ao mesmo tempo; o cálculo do quanto é gasto por funcionário que está operando a máquina através da utilização do seu custo/hora; a criação de um sistema de autenticação que permita que um gestor dê a um funcionário privilégio para testar novos modelos de produção.

Uma melhoria importante também seria o refinamento do gráfico da rede, incluindo informações mais relevantes e controles mais precisos de tempo. A introdução de ferramentas que tornem a visualização em três dimensões também não é descartada.

REFERÊNCIAS

ANGULAR, **Angular Organization.** Disponível em: <https://angular.io/docs/ts/latest/guide/architecture.html>. Acessado em: 27 nov. 2016.

BUSCHMANN, F. Meunier, R. Rohnert, H. Sommerland, P. Stal, M. **Pattern-Oriented software Architectur A System of Patterns.** John Wiley & Sons, 1996.

CHIAVERINI, Vicente. **Fabricação Mecânica:** Processos de fabricação e tratamento . 2. ed. São Paulo: Makron Books, 1986.

CHILE PALOMINO, Reynaldo. **Modelagem e análise de sistemas de montagem utilizando redes de Petri.** Engenep. Gramado, 1997.

COULOURIS, George. DOLLIMORE, Jean. KINDBERG, Tim. **Sistemas Distribuídos**: Conceitos e Projeto. 4. ed. São Paulo: Bookman Editora, 2007.

ECMA. **Standard ECMA-404**: The JSON data interchange format. 1 ed. Genebra: Ecma International, 2013.

ELMASRI, Ramez. NAVATHE, Shamkant B. **Sistemas de Bancos de Dados.** 6 ed. Person, 2011.

FIELDING, Roy T. ***Architectural Styles and the Design of Network-based Software Architectures.*** Tese de Doutorado, Universidade da California, Irvine, 2000.

FREEMAN, Adam. **Pro AngularJS -** Learn to harness the power of moden web browsers from whin your application's code. Apress, 2015.

HULTQUIST, Steve. **FAQ about Client/Server.** Disponível em: <http://non.com/news.answers/client-server-faq.html>. Acesso em: 27 nov. 2016.

IHRIG, Colin J. **Pro Node. js for Developers**. Nova Iorque: Apress, 2013.

ILS. **Internet live stats**. Disponivel em <http://www.internetlivestats.com/internet-users/>. Acesso em: 26 nov. 2016.

MACIEL, Paulo R. M. et al **Introdução às redes de Petri e Aplicações**. X Escola de Computação, Campinas, 1996.

MARRANGHELLO, Norian. **Redes de Petri: Conceitos e Aplicações**. Disponível em <https://www.dcce.ibilce.unesp.br/~norian/cursos/mds/ApostilaRdP-CA.pdf> Acessado em 1 dez. 2016.

MOZILLA, **HTML.** Disponível em: <https://developer.mozilla.org/pt-BR/docs/Web/HTML>. Acesso em: 27 nov. 2016a.

_____, Introdução ao HTML**.** Disponível em: <https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/Introduction>. Acesso em: 27 nov. 2016b.

_____, CSS**.** Disponível em: <https://developer.mozilla.org/pt-BR/docs/Web/CSS>. Acesso em: 27 nov. 2016c.

_____, Canvas**.** Disponível em: <https://developer.mozilla.org/pt-BR/docs/Web/HTML/Canvas>. Acesso em: 15 out. 2017.

OHNO, T. **O Sistema Toyota de Produção:** além da produção em larga escala. 5. ed. Porto Alegre: Bookman, 1997. 149 p.

PETRI, C.A. **Communication with Automata**, Technical Report RADC-TR-65-377, Griffiths Air Force Base, Nova Iorque, 1966.

POSTGRES. **PostgreSQL organization**. Disponível em <https://www.postgresql.org.br/sobre>. Acessado em: 28 nov. 2016.

FABRICJS. **Fabricjs Javascript canvas library**.Disponível em <https://github.com/kangax/fabric.js/>. Acessado em: 28 nov. 2016.

PRIKLADNICKI, Rafael; MILANI, Fabiano; WILLI, Renato. **Métodos ágeis para desenvolvimento de software**. 1. ed. Porto Alegre: Bookman Companhia Ed., 2014.

RISCHPATER, Ray. **Javascript JSON Cookbook**. 1. ed. Birmingham: Packt, 2015.

ROBBINS, Jennifer N. **Learning Web Design**: A Beginner's Guide to HTML, CSS, JavaScript, and Web Graphics. 4. ed. Sebastopol: O'Reilly, 2012.

SALEMI, Joe. **Banco de Dados Cliente/Servidor** . IBPI Press, 1993.

SAUDATE, Alexandre. **REST**: Construa API's inteligentes de maneira simples. 1. ed. São Paulo: Casa do Código, 2014.

SCHMITT, Christopher. **CSS Cookbook**: Soluções rápidas para problemas comuns com CSS. 3. ed. São Paulo: Novatec, 2010.

SILVA, Mauricio S. **Bootstrap 3.3.5**: Aprenda a usar o framework Bootstrap para criar layouts CSS complexos e responsivos. 1. ed. São Paulo: Novatec, 2010.

SMITH, Ben. **Beggining JSON**: Learn the preferred data format on the web. 1. ed. Nova Iorque: Apress, 2015.

ZURAWSKI, Richard. ZHOU, Meng Chu. **Petri Nets and Industrial Applications: A tutorial.** IEEE Transactions on Industrial Electronics. v. 40, n. 6, 1994.

APÊNDICE A – Arquivo de configuração app-http.service.ts

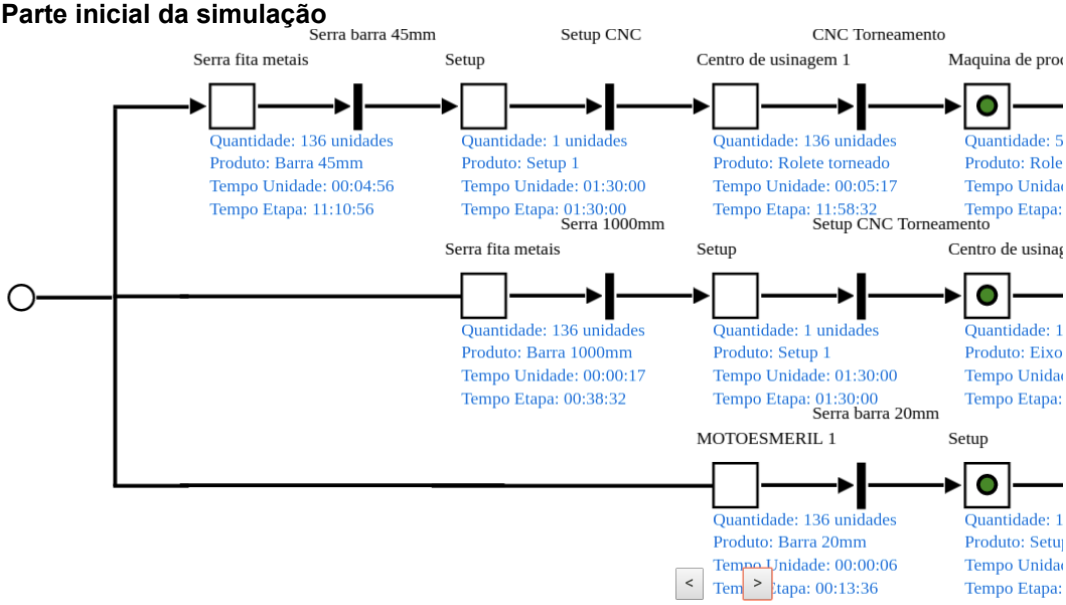
Arquivo De Configuração *app-http.service.ts*

```
import { Injectable } from '@angular/core';
import { Http, Headers, RequestOptions } from '@angular/http';

import 'rxjs/add/operator/toPromise';

@Injectable()
export class AppHttpService {

private url: string;
private header: Headers;

constructor (private http: Http) {

}

setAccessToken(): RequestOptions{
        let token = '';
        this.header = new Headers([
        {'Authorization' : 'Bearer '+ token},
        {'Access-Control-Allow-Origin': '*'},
        {'Access-Control-Request-Method': '*'}
        ]);
        let options: RequestOptions = new RequestOptions({ headers:
this.header });
        return options;
}

builder(resource: string): string{
        this.url = 'http://machboard.app/api/' + resource;
        return this.url;
}

}
```

APÊNDICE B - Imagem da rede de petri do modelo real pelo simulador

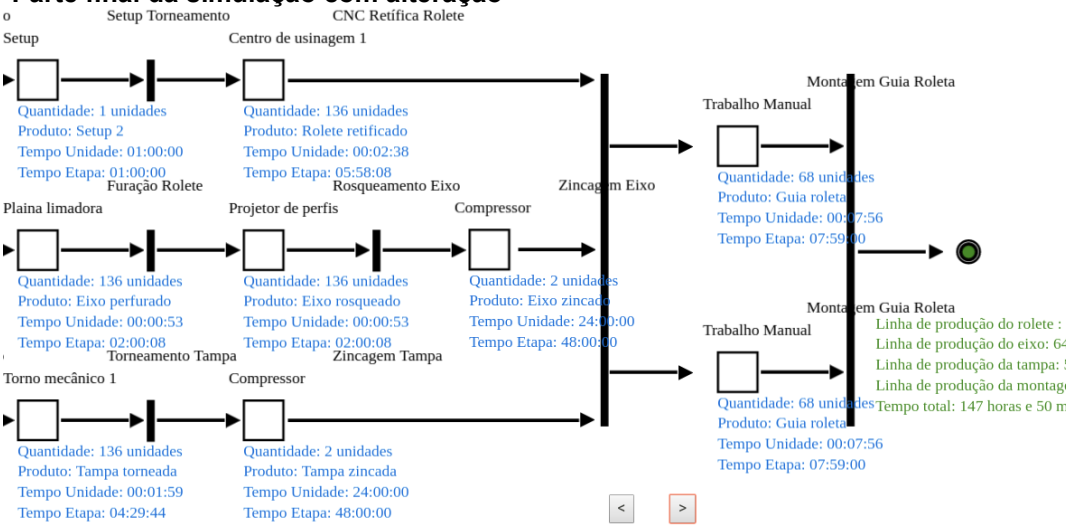
Imagem da rede de petri do modelo real pelo simulador

Figura 22 – Parte inicial da simulação



Fonte: Autoria própria

Figura 23 – Parte final da simulação com alteração



Fonte: Autoria própria