FACULDADE DE TECNOLOGIA ALCIDES MAYA - AMTEC
CURSO TECNOLÓGICO EM DESENVOLVIMENTO DE SISTEMAS PARA INTERNET

ANTONIO RICARDO MEDRONHA STEIN JUNIOR

ARQUITETURA REST API E DESENVOLVIMENTO DE UMA APLICAÇÃO WEB SERVICE

Porto Alegre
2019

ANTONIO RICARDO MEDRONHA STEIN JUNIOR

ARQUITETURA REST API E DESENVOLVIMENTO DE UMA APLICAÇÃO WEB SERVICE

Projeto de Pesquisa apresentado como requisito parcial para obtenção do título de Tecnólogo em Desenvolvimento de sistemas para internet, pelo Curso de Desenvolvimento de sistemas para internet da Faculdade de Tecnologia Alcides Maya - AMTEC

Orientador: Prof Me. Maicon dos Santos

Porto Alegre

2019

SUMÁRIO

1 INTRODUÇÃO 3
1.1 Definição do Tema ou Problema 4
1.2 Delimitações do Trabalho 5
1.3 Objetivos 5
1.3.1 Objetivo Geral 5
1.3.2 Objetivos Específicos 5
1.4 JUSTIFICATIVA 5
2 REFERENCIAL TEÓRICO 6
2.1 HTTP 6
2.2 Atlassian Jira 7
2.3 JavaScript 7
2.4 NodeJs 8
2.4.1 Módulos NodeJS 8
2.4.2 Módulo específicos 8
2.5 API Rest 9
2.6.1 Interface uniforme 10
2.6.2 Sem estado 10
2.6.3 Cache 11
2.6.4 Cliente-Servidor 11
2.6.5 Sistema de Camadas 11
2.6.6 Código sob demanda 12
2.7 Angular 12
2.7.1 NGX-Barcode 12
2.8 Web service vs Api Rest 13
2.9 Postman 14
3 METODOLOGIA 14
4 DESENVOLVIMENTO 15
4.1 Arquitetura do projeto 18
4.2 Arquitetura do servidor 19
4.2.1 Configuração Servidor Express 21
4.2.2 Rotas Api 23
4.2.3 Modelo de requisição 25
4.2.4 Controlador de Tarefas 26

4.2.3 Controlador de Links 33
4.3 Arquitetura cliente 34
4.3.1 Etiquetas 36
5 Validação 40
5.1 Manual versus automático 41
5.1.1 Gerando etiquetas pelo Jira 41
5.1.2 Gerando etiquetas automaticamente 42
5.2 Métrica de Geração de Etiquetas 45
5.2.1 Teste de estresse 46
5.2.2 Teste de múltiplas etiquetas 47
5.3 Criação de Tarefa Manual versus Automático 49
6 CONCLUSÃO 56
8 REFERÊNCIAS BIBLIOGRÁFICA 58

1 INTRODUÇÃO

O objetivo deste trabalho, é desenvolver um sistema que utiliza arquitetura Rest API [1], e linguagem de programação NodeJS[2] (MDN, 2019), bem como práticas e padrões de desenvolvimento da arquitetura de design de Apis, sendo o servidor a camada onde é feita as chamadas REST que nada mais são que interações com outro servidor disponibilizando uma informação via Api e a persistência de dados, assim como a integração e utilização das API'S do JIRA de modo que seja feito a automatização e criação de um processo de rastreamento e aplicação de etiqueta de itens. O back-end se apresenta separado do lado do cliente, que foi utilizado ANGULAR (ANGULAR, 2010), uma plataforma de aplicações web onde se encontram todas as regras de negócio, bem como exibição e captura de entrada de dados. Cada parte exposta aqui é considerado um subsistema.

As APIs que foram implementadas de forma estruturada lidam com as requisições na forma de HTTP[3], fazendo assim efetivo a comunicação entre os subsistemas e

[1] Significa interface de programação e aplicação, é uma arquitetura de desenvolvimento que permite uma interoperabilidade entre sistemas, ou seja, comunicação entre aplicações e usuários.
[2] Linguagem baseada em JavaScript utilizada para desenvolvimento em servidores.
[3] Protocolo de transferência e comunicação de sistemas da internet, distribuídos e afins. É a base de interligação de dados da internet.

sistemas externos, dentre os padrões impostos pelo design de Api, é gerado uma aplicação de independência no quesito manutenção, ou seja, é possível gerir e alterar os subsistemas com certa independência, sem afetar os outros módulos presentes, salvo que é necessário manter o padrão e compatibilidade.

O design proposto não é uma metodologia definida, mas uma forma de incrementar um software utilizando de boas práticas de desenvolvimento de software, que podem ser observadas como algo explícito em toda forma de apresentação relacionada a *web* APIs.

Normalmente, o ambiente mais propício para o desenvolvimento deste tipo de aplicação, que por sua natureza pode ser descrito como distribuído, sendo este a internet, visto que em ascendência em popularidade e reconhecimento, assim como complementado por umas das linguagens mais conhecidas para desenvolvimento *web* o JavaScript, ambos fazem com que Node JS (NODEJS, 2018) seja uma linguagem em constante crescimento e desenvolvimento, e expansão de JavaScript linguagens de resposta e servidores.

O protótipo final da aplicação é uma integração com os sistemas da Atlassian (ATLASSIAN SUPPORT, 2015), onde são feitas várias manipulações dentro do sistema em questão, bem como o desenvolvimento de uma solução pouco ou não presente no ecossistema Atlassian, o gerenciamento de itens, onde se tem um fluxo de automação para gerar códigos de barra a partir de tarefas do JIRA e também modificá-las.

1.1 Definição do Tema ou Problema

Atualmente, não existe uma solução simples para gerar etiquetas e criar códigos de barra a partir de um serviço web que obtenha informações de uma API do Jira Atlassian.

1.2 Delimitações do Trabalho

Delimita-se a criar um subsistema com arquitetura API para efetuar chamadas REST em um webservice JIRA, e retornar tarefas onde contendo um fluxo específico irá gerar etiquetas com código de barra que representam tarefas no JIRA.

1.3 Objetivos

Desenvolvimento, arquitetura e prototipagem funcional de uma aplicação de controle de ativos.

1.3.1 Objetivo Geral

Desenvolver uma interface que possa obter os dados da API JIRA e gerar etiquetas e códigos de barra.

1.3.2 Objetivos Específicos

a) identificar as tecnologias necessárias para construir a arquitetura com API;
b) analisar a viabilidade, e as tecnologia a serem aprendidas;
c) determinar a qualidade do sistema feito neste modelo, bem como agilidade total nos processos propostos pelo projeto.
d) finalizar uma aplicação funcional, ágil e simples.

1.4 JUSTIFICATIVA

Com o grande ecossistema e usabilidade que existe dentro da ferramenta Jira o mesmo é utilizado de forma extensa para diversas formas de controle de tarefas e outros (ATLASSIAN, 2019), e bem como tal existem também aqueles que utilizam o mesmo para controlar ativos e itens. Com este problema em mãos e o déficit que existe referente a gerenciamento de ativos de forma fácil, acessível e que atualize de fato a aplicação Jira e por fim tendo como diferencial utilizar etiquetamento para este fim. De forma a atingir principalmente varejo ou outros nichos de mercado que utilizam o etiquetamento com código de barra como forma de controle e utiliza ou quer utilizar os sistemas Jira Atlassian.

2 REFERENCIAL TEÓRICO

Neste Capítulo será descrito as tecnologias que fazem parte do trabalho e conceitos para fundamentação deste projeto.

2.1 HTTP

O princípio básico de trabalhar com Apis é a utilização de HTTP, ou em português, Protocolo de Transferência de Hipertexto, ou seja, é uma forma de comunicação, baseado no modelo OSI[4], este é presente na camada de aplicação. Normalmente este protocolo opera na porta 80, e normalmente é associado a comunicação de sites, utilizando a linguagem HTML (Marcação de hipertexto, ou na linguagem original, Hipertext Markup). Conforme especificação da versão 1.1 do protocolo em questão, (REDAÇÃO OFICINA, 2007), este protocolo é descrito como "stateless", ou seja, não conserva o estado de nenhuma aplicação, tornando-o em sua forma comum, escalável. Um importante característica, além da tipagem, e a representação de dados, permitindo que os sistemas se comuniquem e sejam construídos de forma independente dos dados que estão sendo transferidos.

Para o funcionamento do HTTP se faz necessário outros dois protocolos, conhecidos como TCP e IP (BERTULUCCI, 2005), bem como o modelo cliente-servidor, baseado em programa cliente(requisitante), que estabelece uma conexão com o servidor. Este protocolo em suma funciona como requisição/resposta. Um programa requisitante envia uma solicitação a um servidor contendo as seguintes informações: "*request method", URL,* versão do protocolo e os dados da mensagem. Toda a comunicação do sistema em questão é feita a partir desse modelo, então HTTP se torna uma das bases ao qual este sistema foi construído.

---

[4] Modelo de representação de uma arquitetura de rede e suas respectivas camadas de funcionalidade. (MICROSOFT, 2017)

2.2 Atlassian Jira

Jira é um software desenvolvido com objetivo principal de servir diversos times a gerenciar seus trabalhos diários. Esta plataforma oferece vários produtos e diversas formas de “lançar” a aplicação ao ar que são construídas com o propósito de servir áreas como TI, negócios, Times de operações e muito mais. Esse software auxilia vários times a planejar, distribuir, rastrear, gerenciar e fazer relatórios sobre estes. Basicamente uma plataforma que une desde times ágeis de desenvolvimento a times de suporte ao cliente, podendo, até mesmo ser utilizado para tarefas diárias da casa.

O sistema trabalha com uma premissa baseada em “fluxos”, ou seja, você define um tipo de tarefa, como tarefas de desenvolvimento ou tarefas de estudo, configura um fluxo de trabalho dentro desta tarefa que envolve desde “iniciar trabalhos”, a “finalizar tarefa”. Uma destas formas de lançar esta plataforma é pelo website da Atlassian, criadora da plataforma, que envolve utilizar uma instância baseada em Cloud. Esta plataforma Cloud é basicamente uma aplicação Jira software que enquadra uma utilização mais limitada do que a versão instalável do mesmo, porém, ainda assim permite a utilização das premissas da plataforma. (ATLASSIAN, 2019).

2.3 JavaScript

Popularmente conhecido com JS, é uma linguagem de interpretação e orientada a objetos, tendo suporte a heranças, propriedades e métodos. É possível ter polimorfismo, encapsulamento e vários outros paradigmas de objetos. A mesma, no entanto, é comumente associada a scripts para web, tendo também diversos outros usos fora do navegador. Conforme (JavaScript - Mozilla Developer Network, 2014) JS é uma linguagem baseado em protótipos, multiparadigmas e dinâmica. tendo total suporte a orientação a objeto, e programação funcional.

2.4 NodeJs

É considerado a expansão do JS, para o lado do *back-end* das aplicações, ou seja, o lado do servidor, sendo dirigido por eventos assíncronos em tempo de execução,

node foi desenvolvido para criar aplicações escaláveis, podendo lidar com múltiplas conexões. Conforme documentação (MDN, 2019), este método é um contraste ao mais comumente utilizado pela concorrência, onde os encadeamentos são feitos por unidades básicas do processador. Uma rede baseada em encadeamentos é relativamente ineficiente e difícil de ser utilizada além disso, os usuários do Node estão livres de preocupações com o bloqueio do processo, já que não há bloqueios. Quase nenhuma função no Node executa diretamente, portanto, o processo nunca é bloqueado. Como nada bloqueia, é muito razoável desenvolver sistemas escalonáveis no Node. Contamos também com todo o ecossistema de módulos para NodeJS. (NODE.JS FOUNDATION, 2011)

2.4.1 Módulos NodeJS

Node tem por si uma grande usabilidade, mas o que o torna tão robusto é o grande ecossistema de módulos sendo proprietários, livres e demais, são de comparação as bibliotecas de Java, ou para leigos “*plugins*”, que complementam e facilitam a vida diária do desenvolvedor.

2.4.2 Módulo específicos

Para a criação do sistema foram utilizados alguns módulos com mais frequência que outros, estes são:

- ExpressJS um framework para node que lida com rotas, métodos de utilidade HTTP, performance e outra utilidades para trabalhar com API. É o principal meio de definir as rotas, e configurar a arquitetura API, sem este módulo não é possível de forma fácil, trabalhar com REST (STRONGLOOP, IBM, 2017).

- Request uma integração básica que permite fazer as requisições HTTPS, de forma fácil e simples. (SCHOTT et al., 2018).

- Body-Parser um framework para Node que lida com rotas, métodos de utilidade HTTP, performance e outra utilidades para trabalhar com API, basicamente converte as URLs de forma que o navegador entenda. (WILSON, 2019)

2.5 API Rest

Até o ano de 2000, não existiam padrões em como design de API deveria ser. A integração necessária, uso de protocolos, como SOAP[5], que era notoriamente conhecido como algo complexo a ser construído, manusear e debugar. (SCHULTHESS, 2017)

Isso mudou quando um grupo de experts, liderados por Roy T. Fielding, mudaram para sempre a concepção de API, pode até se dizer que ele foi o criados da mesma. O trabalho de Fielding consistia em criar padrões que permitiriam dois servidores se comunicarem e trocarem dados em qualquer parte do mundo. Eles então definiram princípios, propriedade e constrições que nomearam “REST”. Sem absorver nenhum estado, sessão, ou cache, utilizando o protocolo HTTP e seus métodos. Rest é baseado em recursos, ou seja, objetos, ‘coisas’. Na modelagem de REST (FIELDING, 2004) é comumente referido a substantivos nos seus métodos, em uma requisição de HTTP alguns dos recursos (GET, POST, PUT, DELETE), em ordem “buscar, postar, colocar, deletar”, referem-se a os recursos. Uma chamada API que busque todos os nomes getAllNames(), utilizando os recursos a URL com o método seria, ‘GET’ Api.dominio.com/names.

Comumente os recursos são definidos por URLs e separados por sua representação no código. A representação pode ser um estado de um dos recursos, ou seja, um dado de transferência entre solicitante e receptor.

No paradigma de desenvolvimento de REST API o estilo arquitetural descreve seis constantes, as mesmas aplicadas ao desenvolvimento foram originalmente comunicadas por Roy Fielding na sua dissertação de doutorado e define as bases do estilo Rest. (WHAT IS REST, 2012)

[5] Protocolo de mensagens, serve para troca de dados estruturados.

2.6.1 Interface uniforme

Define a interface server e cliente, simplifica e desacopla a arquitetura, o que permite a evolução de cada parte de forma independente. Cada recurso individual é identificado nos requisitos usando as URLs como tal identificador. O servidor não envia o banco de dados, mas ao invés disto envia uma representação em HTML, XML ou Json, contendo os registros do banco por exemplo.

O lado do cliente, se munido de representação suficiente de um objeto, ele pode modificar ou deletar algum recurso do servidor tendo as permissões corretas. Sendo ainda responsabilidade da interface as mensagens auto descritivas, contendo informação suficiente para descrever o processo em execução, por exemplo. Por fim, o requisito neste paradigma entrega o estado pelo corpo de dados, conteúdo, parâmetros da chamada, URL e os cabeçalhos da chamada. As entregas são tecnicamente referidas como hipermídia. (WHAT IS REST, 2012)

2.6.2 Sem estado

Sendo REST, em português, acrônimo para estado representacional de transferência, não ter estado é a chave. De forma geral, o que isso quer dizer é que o estado necessário para lidar com as requisições está contido dentro da requisição, podendo ser em qualquer parte dela, corpo ou cabeçalho. A URL identifica de forma única o recurso, não ter estado, ou seja, recebendo apenas as peças como resposta de um estado apropriado representa uma grande usabilidade, tanto a termos uma grande escalabilidade, quanto balanço nos carregamentos. (WHAT IS REST, 2012)

2.6.3 Cache

Assim como no WWW (REDAÇÃO OFICINA, 2007), os clientes ou usuários podem ter suas respostas gravadas no cache. Porém as respostas, implícita ou explicitamente, devem definir-se como cacheáveis ou não, para prevenir o cliente de usar dados inapropriados na resposta e em próximas chamadas. Tendo esse tipo de

interação bem definido, elimina a necessidade de algumas interações no lado do servidor, melhorando a escalabilidade e performance. (WHAT IS REST, 2012)

2.6.4 Cliente-Servidor

A interface uniforme separa os clientes do servidor, essa separação significa que os clientes não precisam se preocupar com persistência, manipulação ou qualquer coisa relacionado a armazenamento de dados, que fica a cargo do servidor interno, fazendo com que a portabilidade do código cliente seja melhorada, visto que os servidores agora não se preocupam com a interface do usuário ou o estado do mesmo, fazendo com que os servidores sejam mais simples e escaláveis. Os servidores e clientes(front-end) podem ser substituídos e desenvolvidos de forma independente. (WHAT IS REST, 2012)

2.6.5 Sistema de Camadas

O front não pode diretamente direcionar se o mesmo está conectado ao servidor ou até mesmo a um intermediário no meio do caminho. Servidores intermediários servem para aprimorar o desempenho, balanço de carregamento escalonamento e provisionar caches múltiplos. Ou seja, é possível ter múltiplas camadas de servidor por exemplo.

2.6.6 Código sob demanda

Sendo opcional, os servidores podem, temporariamente estender ou customizar a funcionalidade do lado do cliente, transferindo dados de execução. Exemplos desse processo podem incluir componentes como scripts e outros. (WHAT IS REST, 2012)

2.7 Angular

Angular é uma plataforma de desenvolvimento *web* baseado em javascript, html e css, onde o mesmo é responsável por toda aplicação de lado cliente, botões, textos, transições, imagens e afins. Utilizando de filosofias de desenvolvimento voltado para cliente, a aplicação foi desenvolvida inicialmente com intuito de desacoplar manipulação direta de elementos de html[6] em aplicações e websites, especificamente falando, um modelo de “atribuição” de dados, onde era possível ter atualizações dinâmicas baseando o código nestas atribuições. A aplicação teve tamanha popularidade e aceitação que pouco tempo depois, uma versão nova introduziu novas formas de desenvolver com a plataforma, que mesmo em decadência na popularidade, trouxe consigo novas formas de desenvolver código para interfaces, sendo uma delas as “rotas”, que definiram com força o conceito de abrir páginas sem “recarregar” o site inteiro.(ANGULAR,2010)

2.7.1 NGX-Barcode

Módulo complementar do Angular que de forma fácil criar códigos de barra correspondentes a texto, utilizando de um bloco de código pré-definido pelo módulo é possível criar uma etiqueta passando um parâmetro para este bloco, esse parâmetro é um texto que é convertido então para a etiqueta em si. (BRYON, 2017)

2.8 Web service vs Api Rest

Embora ambos sejam parecidos, existe uma diferença substancial entre APIs e *web services*, por definição é qualquer tipo de software que se auto disponibiliza na internet e padroniza sua comunicação. Um cliente invoca este serviço enviando requisições normalmente no formato JSON, e o serviço envia em retorno uma resposta.

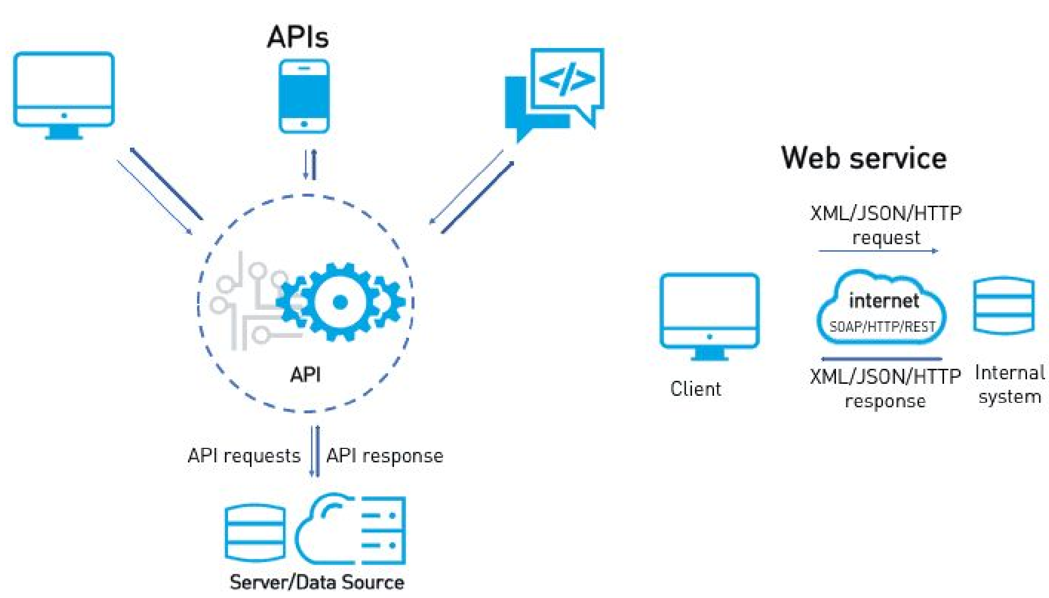
Em contraste uma API específica como componentes do software deve interagir uns com os outros, usando protocolo *HTTP,* o cliente normalmente não precisa saber qual procedimento chamar no lado do servidor, ao invés disso, é utilizado uma cadeia de

---

[6] Linguagem utilizada para criação de elementos visuais na tela. (MDN - HTML5)

comandos escritos como “verbos”. Um exemplo deste é o “GET” que faz a busca e retorno de dados. Quando ambas partes são comparadas normalmente sistemas em API são considerados mais fracamente acoplados, enquanto que os *web services* não, portanto são considerados menos frágeis, isto dá-se ao fato de que quanto mais independência entre subsistemas melhor. Ambas soluções servem como meios de comunicação entre cliente e servidor, ambos suportam envio e recebimento de dados JSON, a significante diferença está entre a quantidade de trabalho que deve ser feito ao cliente e serviço/servidor, isto quer dizer que o processo de serialização de dados ou seja envio e retorno, nos casos da API são muito menores do que quando comparado aos web services, (VERMA, 2018). A imagem a seguir representa de forma clara a diferença entre ambos:

Figura 1. API vs Web Service.



Fonte:(VERMA, 2018).

2.9 Postman

Postman é um software que auxilia no desenvolvimento de Api’s ele é necessário para efetuar testes nas rotas de envio de requisições http, enviar corpo de dados e cabeçalhos de requisição, assim como é responsável por receber dados de resposta,

pode ser considerado o ambiente onde se tem o design das Apis, quais dados enviar, respostas a receber, testar os *endpoints,* e afins. *(*POSTMAN INC, 2019*).*

3 METODOLOGIA

O método utilizado neste trabalho foi a pesquisa aplicada, onde é necessário ter alguma forma de aplicação para gerar dados, resultados e validação do trabalho, e a pesquisa exploratória de forma a ter familiarização com o tema proposto, análise de exemplos e o levantamento bibliográfico.

4 DESENVOLVIMENTO

A proposta do projeto é construir uma solução com arquitetura distribuída e de serviços *web* , ao qual tenha-se uma aplicação com acesso via REST a uma aplicação JIRA CLOUD da Atlassian, com intuito de prover um controle de varejo sob as mercadorias, as quais são provindas cadastradas do sistema Jira que receberá as chamadas de API, sob o domínio de um cliente que será referido como LOGIC-X. A funcionalidade é uma aplicação autoral contendo um front-end, back-end, estruturado, de forma a entender como times funcionam num ambiente real com metodologias de entrega, versionamento de código e entregas específicas em um determinado prazo de tempo. Este então, chamaremos de operações diárias consistem em nas seguintes funcionalidades:

- CRUD[7] – Tarefas;
- Mudança de status das tarefas;
- Relacionar tarefas;
- Mensageria de resposta API;
- Geração de etiquetas com código de barras.

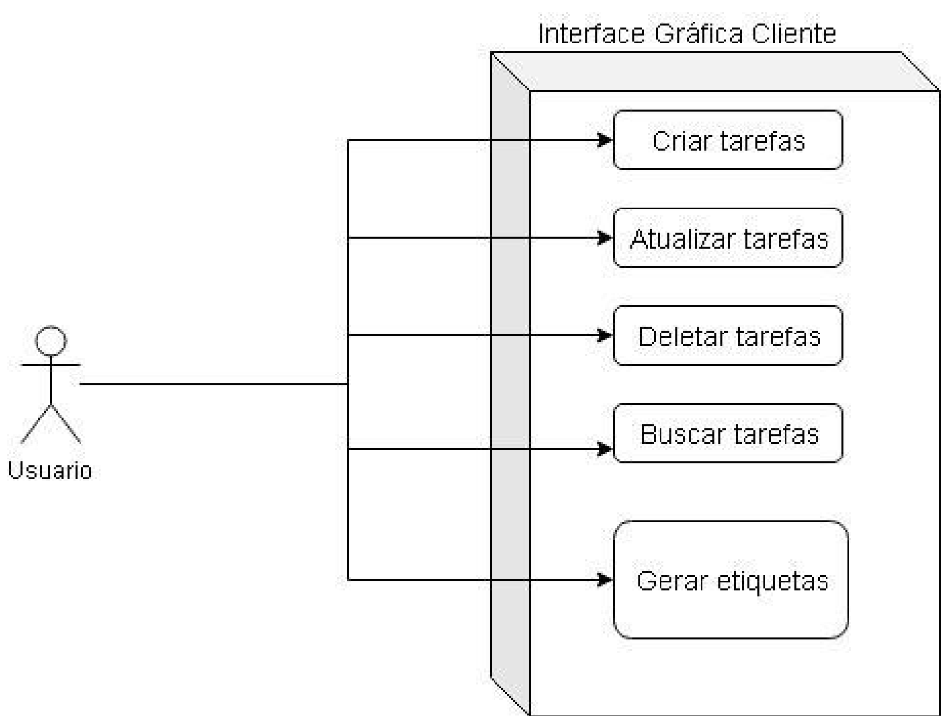
As tecnologias apresentadas foram escolhidas pela notoriedade em serem tecnologias de ponta na área de desenvolvimento web, também por iniciativa e curiosidade própria, onde foi considerado um ganho próprio adquiri qualquer forma de maestria sob tais tecnologias. A arquitetura do mesmo foi centrada em boas práticas e

[7] Operações básicas de um software, criar, ler, atualizar, destruir. (ARAÚJO, [2007-2018]).

design de API, sendo o ponto central a facilidade, escalabilidade e modelo pró-negócios da solução. Foi de iniciativa própria ter a habilidade de ter uma aplicação com desenvolvimento rápido, ágil, de fácil integração, para contínuas alterações e implementações.

Para tal objetivo, no lado do servidor foi construído um WebService de API Rest em Node JS, utilizando o framework Express como middleware, o npm Request para simplificar e agilizar requisições, Jira connector para criar a autenticação. A aplicação, contudo, ainda não possui implantação. Porém a mesma contém todas as funcionalidades de um sistema aptas a teste e implantação sob demanda. No lado do cliente, a aplicação possui integração com AngularJS, HTML, Less, Bootstrap. O servidor ainda conta com um sistema básico de envio de e-mail que também foi configurado como padrão API, onde se tem um *endpoint,* que recebe uma chamada https e faz o envio para uma lista Json de usuários do módulo futuro de cadastro de usuários. Na figura 2, é apresentado o diagrama de caso de uso do sistema. Após figura, segue as descrições detalhadas dos casos.

Figura 2. Diagrama de Caso de Uso



Fonte: Elaborado pelo autor.

Caso de Uso: Operar tarefas do varejo (CRUD)

Ator: Usuário

Descrição: ambos através da interface da aplicação, criam, exibem tarefas de projetos, alteram, excluem e buscam tarefas.

Caso de Uso: Operar links

Ator: Usuário

Descrição: através da interface da aplicação, relacionam as tarefas, excluem links, buscam tarefas pelos tipos de links disponíveis.

Caso de Uso: Gerar Etiqueta

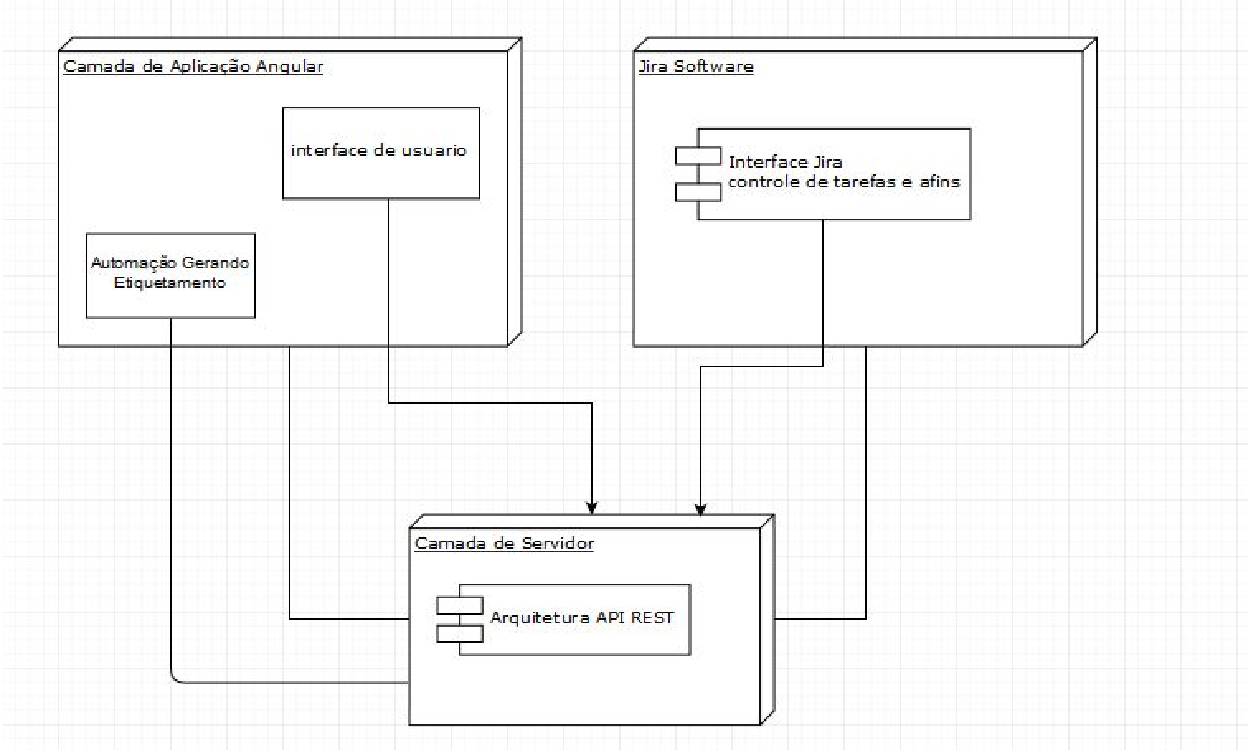
Ator: Usuário

Descrição: através da interface da aplicação, gera etiqueta após fluxo necessário para gerar a mesma.

4.1 Arquitetura do projeto

O sistema é dividido em dois subsistemas ou módulos, um sendo o lado servidor e o outro sendo o lado cliente. O lado do front-end ou lado cliente:

Figura 3. Arquitetura e visão geral diagrama.



Fonte: Elaborado pelo autor.

A figura representa a arquitetura do projeto como um todo, tendo em visão clara as 3 subcamadas que compõem o mesmo, a primeira parte representada pela aplicação do usuário, contendo as tecnologias de front-end, como Angular, css, e html. Também utiliza a biblioteca de utilidade ngx-barcode do Angular utilizada para gerar os códigos de barra e etiquetamento que são editados pelo css na camada de interface. A camada de servidor constitui todos os endpoints de API que retornam dados para o front-end, com o Middleware do express e a biblioteca request e feito as chamadas para o JIRA que retornam informações da aplicação para o servidor e por consequência o front também. A aplicação de Jira Cloud não chega a ser bem um subsistema, porém como ela funciona neste caso apenas como plataforma de coleta de dados, consideramos como, esta camada serve apenas para receber as chamadas e retornar dados em formato JSON.

4.2 Arquitetura do servidor

No servidor de *back-end* que é a camada importante para este projeto, de forma detalhada na figura 4, exemplificando as chamadas com o fluxo do webservice.

Figura 4. Fluxo de requisição para Web Service.

Fonte: Elaborado pelo autor .

Sendo a funcionalidade principal abordada aqui, a estrutura definida pelo back-end foi a utilização firme de Express como middleware[8], e também a utilização do servidor do node, junto com as rotas de requisição como um subsistema de camadas, sendo o servidor node a primeira camada, e a aplicação JIRA a segunda, a aplicação que retorna as informações está sendo o JIRA, foi tratado como um subsistema, pois a Atlassian, criadora da plataforma lida com as end points de formas diferente, e esta aplicação dá LOGIC-X está alocada em um servidor físico. Foram definidas rotas específicas para o tratamento do NODEJS, porém não são os recursos que chegam a requisição, um exemplo é a URL de chamada da API é diferente da requisição que o JIRA recebe sendo estas no NODE (URL.domain.com/Jira/issue) e no JIRA recebendo

[8] Meio de campo entre as chamadas, uma espécie de software que lida com as requisições como meio de campo.

(LOGIC-X.com/rest/Api/3/issue/). É possível perceber que a chamada é recebida de forma diferente para ambos subsistemas. Seguindo o estilo de arquitetura proposto pela REST API. O banco de dados é acessado apenas para persistir os dados de usuários sendo este o banco, neste caso é a própria instância do Jira, todas as outras transações são feitas em requisições e guardando os resultados em memória de navegador.

Algumas das integrações do Jira tiveram que ser construídas como micro serviços e acessando instâncias diferentes do Jira pois as requisições são feitas em tempo de resposta, e se múltiplas conexões para mesma aplicação ficassem ocorrendo o tempo todo, geraria problemas de performance. A modelagem foi feita da seguinte forma, temos a estrutura padrão de diretórios que constitui em, controladores, modelos e rotas, conforme imagem:

Figura 5. Diretórios do projeto.

Fonte: Elaborado pelo autor.

Está estrutura permite um controle maior, tendo em vista que dentro dos controladores é onde é feito as requisições e manipulação de erros. Os modelos correspondem a modelagem de dados reutilizáveis, como tabela de dados de usuários, URLs de requisição para Api's, e por fim as rotas, onde é feito a configuração dos *endpoints* das requisições, ou seja, onde vai ser a URL final que será feita a requisição.

4.2.1 Configuração Servidor Express

Neste arquivo, denominado de “index.js”, é feito a configuração do *middleware* express, que faz o controle das rotas e cria o webservice, bem como existe uma pré-configuração de banco de dados que não é utilizado neste projeto, conforme imagem:

Figura 5. Código Express

```
1   'use strict'
2   const express = require('express');
3
4   const app = express();
5
6   //const cors = require('cors');
7   const apiRoutes = require("./routes/api-routes");
8   const jiraRoutes = require('./routes/jira-api-routes');
9
10
11  const user = require('./models/userModel');
12  const bodyParser = require('body-parser');
13  const mongoClient = require('mongodb').MongoClient;
14  app.use(bodyParser.urlencoded({ extended: true }));
15  app.use(bodyParser.json());
16  apiRoutes(app);
17  jiraRoutes(app);
18
19
20  const connection = mongoClient.connect('mongodb://localhost:27017/logistica', { useNewUrlParser: true });
21
22  // connections config
23  const port = process.env.PORT || 3000;
24  const host = '0.0.0.0';
25  const db = connection;
26
27  //app.use(cors());
28
29  app.get('/', function(req, res) {
30      res.json(({ Hello: 'World' }));
31  });
32  app.listen(port, host, function() {
33      console.log(port);
34      console.log("Server started.......");
35  });
```

Fonte: Elaborado pelo autor.

Este código representa de forma correta o design e boas práticas de modelagem de Express. No começo do código temos a utilização do modo “strict[9]”, e então importamos a biblioteca do express com a constante express na linha dois, e na linha quatro criamos a instância do aplicativo, denominada de “app”, ela serve como uma referência para o nosso *middleware* do express. Nas linhas 7,8,11,12,13 são feitas as importações de bibliotecas e arquivos necessários para o funcionamento e utilização do

[9] Forma de semântica da linguagem javascript que altera vários padrões e permite utilizar uma forma diferente de escrita de JS. (MDN MOZILLA, [2005-2019])

express sendo uma delas o *body-parser*. Logo após temos um bloco interessante, que efetua uma grande parte no projeto e que permite os arquivos de rotas utilizarem o express, as linhas 16,17 representam uma chamada nos arquivos "ApiRoutes & JiraRoutes", ou seja, duas rotas que recebem a instância do nosso aplicativo previamente criado, isso serve para que esses dois arquivos, recebam uma referência do aplicativo express e saibam se comunicar com o mesmo. Na linha 20 temos uma configuração de conexão de banco de dados fictícia que não utilizamos no projeto. As linhas seguintes representam a configuração de um servidor local de desenvolvimento, ou seja, uma forma de ver em uma página ou com chamadas de API, os resultados do nosso desenvolvimento, em ordem o "port" é a porta do servidor local que utilizaremos, ou seja acessaremos a URL "http://localhost:3000" o "host" está sendo configurado de forma fixa para que a aplicação entenda como "utilize qualquer meta rota disponível."(IWAYA, 2015). Nas linhas subsequentes a 29, nós temos a primeira utilização de express, com a requisição do tipo "GET", para a URL " / " ou seja para a página inicial, recebendo uma resposta de sucesso igual a um JSON com as palavras "Hello World", após temos a configuração do express para "escutar" ou seja atualizar sempre o servidor toda vez que é dado o comando de iniciar o servidor que retorna o número da porta de acesso e a mensagem de servidor iniciado. Na imagem a seguir é feito a inicialização do servidor via terminal de comandos utilizando o comando npm start:

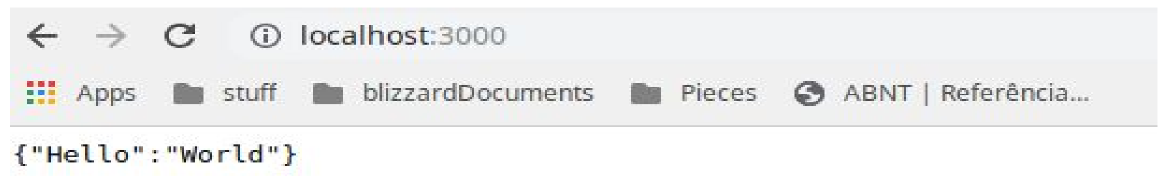
Figura 6. Inicialização do Servidor

```
objectedge@objectedge-Aspire-E5-574:~/Trabalho/Projects/NodeWebService$ npm start

> nodewebservice@1.0.0 start /home/objectedge/Trabalho/Projects/NodeWebService
> node index.js

3000
Server started.......
```

Fonte: Elaborado pelo autor.

O resultado é um ambiente de desenvolvimento com o seguinte aspecto:

Figura 7. Ambiente de desenvolvimento.



Fonte: Elaborado pelo autor.

4.2.2 Rotas Api

A seguir configuramos as rotas de AApi, onde serão feitas as chamadas responsáveis por trazer dados, essas chamadas serão utilizadas pelo *front-end* para requisitar informações do servidor que por sua vez efetua a chamada para o Jira e retorna dados. a seguinte imagem representa a configuração das rotas de Api.

Figura 8. Rotas API Jira

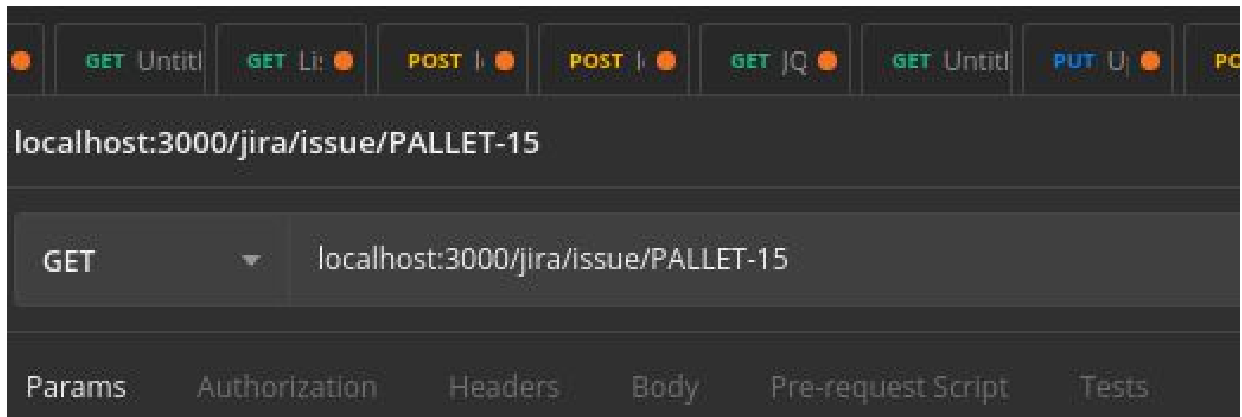
```
'use strict'
module.exports = function(app) {

    const issueController = require('../controller/issueController');
    const issueLinkController = require('../controller/issueLinkController');
    const issueTransitionController = require('../controller/issueTransitionController')

    // ************** Issue Controller **************
    app.route('/jira/issue')
        .post(issueController.create);

    app.route('/jira/jql/search')
        .post(issueController.findByJQL)

    app.route('/jira/issue/:issue_id')
        .put(issueController.update)
        .get(issueController.find)
        .post(issueController.comment)
        .delete(issueController.delete);

    // ************** Transition Controller **************

    app.route('/jira/issueTransition/:issue_id')
        .post(issueTransitionController.transitionIssue);

    // ************** Link Controller **************
    app.route('/jira/issue-link')
        .post(issueLinkController.create);

    app.route('/jira/issue-link/:issue_id')
        .get(issueLinkController.find)
        .delete(issueLinkController.getLinkId,
            issueLinkController.delete);

};
```

Fonte: Elaborado pelo autor.

Neste código, temos ainda a utilização de strict, e na primeira linha de código após temos o “module.exports” que diz para o sistema que esta função está disponível para ser acessada por outros módulos e arquivos, na “function” descrita entre parênteses estamos recebendo a referência do “App” que configuramos previamente no servidor express. Nas próximas linhas até a número sete, temos importações de arquivos que utilizam estas rotas, como os controladores de tarefas do Jira, contamos então a partir da linha nove com o roteamento que se classifica como, à determinação de como um aplicativo responde a uma solicitação do cliente por um endpoint específico, que é uma URI (ou caminho) e um método de solicitação HTTP específico (GET, POST, e assim por diante), StrongLoop inc (2012 e 2019). Cada rota tem uma URL específica onde será feito o endpoint, e logo após os métodos de requisição, cada método de requisição está ligado a uma operação CRUD ou transição dos arquivos importados, por exemplo na linha 15, temos a URL “/Jira/issue/:issue_id” que traduz basicamente para a

requisição utilizar este *endpoint* para receber as chamadas do front, neste temos as operações comuns a API, por exemplo a de encontrar um registro, "*.get(IssueController.find)*" isto se traduz a dentro do controlador de tarefas(*issue Controller")* ache a função *find* e retorne para a URL especificada. o resultado é algo como, visto de dentro do postman:

Figura 9. Requisição de exemplo no postman



Fonte: Elaborado pelo autor.

De acordo com a imagem nove, a URL de servidor local que utilizamos para os endpoints criados pelo express, recebem após a "/" as URLS presentes no código visto nas rotas de API do Jira.

4.2.3 Modelo de requisição

O modelador pode ser utilizado como base para certas operações recorrentes, e como base para criar um modelo de dados para quando houver interações com banco. A figura a seguir representa a estrutura das requisições, ou seja, as opções necessárias e os cabeçalhos da requisição:

Figura 10. Controlador de requisição

```
const email = 'fakeEmail@gmail.com';
const password = '******';

module.exports.options = function(url, method, bodyData) {
    return {
        method: method,
        url: url,
        headers: {
            'Accept': 'application/json',
            'Content-Type': 'application/json',
            'Authorization': 'Basic ' + Buffer.from(email + ':' + password).toString( encoding: 'base64'),
        },
        body: bodyData,
        json: true
    };
};

module.exports.HeadersForRequest = function(req, res) {
    var allowance = res.header("Access-Control-Allow-Origin", "*");
    var origin = res.header("Access-Control-Allow-Headers", "Origin, X-Requested-With, Content-Type, Accept");
    return {
        allowance,
        origin
    };
};
```

Fonte: Elaborado pelo autor.

Neste código, temos no topo o e-mail do usuário do Jira e a senha do mesmo, na primeira versão existe somente autenticação básica no sistema. Nas options, que consiste em opções necessárias no corpo de uma requisição, é passado o método (GET, POST, etc..), a URL que é enviada através do controlador, e os cabeçalhos, são definidos pela própria API, (ATLASSIAN, [entre 2004 e 2016]). Temos mais cabeçalhos que são enviados, são cabeçalhos de resposta HTTP não essenciais, porém utilizados para ter um maior controle. O código de buffer na autorização corresponde a seguinte conforme documentação, antes da introdução das listas tipadas, a linguagem JavaScript não tinha nenhum mecanismo para ler ou manipular fluxos de dados binários. A classe Buffer foi introduzida como parte da API Node.js para permitir a interação com fluxos de octetos em fluxos TCP, operações do sistema de arquivos e outros contextos (NODE JS V12.4 DOC, 2019).

4.2.4 Controlador de Tarefas

O controlador de issues contém as operações de CRUD e a manipulação de erro por resposta, o que é de certa forma uma quebra de paradigma, pois a manipulação de erro deveria ser feita pelo sistema, porém estamos criando validações customizadas para enviar uma resposta ao subsistema *front-end* customizada.

Figura 11. Configurações iniciais controlador

```
1   'use strict'
2
3   const request = require('request');
4
5   const email = 'FakeEmail@gmail.com';
6   const password = '*******';
7   const base_url = 'https://logisticareversa.atlassian.net';
8
9   const rest_api_issue_url = base_url + '/rest/api/3/issue/';
10  const search_with_jql = base_url + '/rest/api/3/search';
11  const jiraModel = require('../models/jiraModel');
```

Fonte: Autor

As importações consistem nos dados do usuário caso necessário, a URL base de onde as requisições de API feitas em direção ao JIRA , a linha nove representa a URL base da documentação JIRA para trabalhar com tarefas, está é a URL que sera utilizada para enviar requisições de tarefas para o Jira, sendo a extensão total, por exemplo : “Https://logisticareversa.atlassian.net/rest/Api/3/issue”, está e a URL definida pela documentação de Api da Atlassian para gerar dados nesta URL, a linha 10 representa a URL que a documentação sugere utilizar para efetuar buscas usando um termo utilizado para pesquisa de tarefas dentro da plataforma Jira, conhecido como JQL[10]. A figura a seguir contém a função de criação de uma tarefa usando API no Jira.

Figura 12. Cabeçalho da função de criar tarefas

```
14  module.exports.create = function(req, res, next) {
15
16      jiraModel.HeadersForRequest(req, res); //headers
17      let json = req.body;
18
19      const options = jiraModel.options(rest_api_issue_url, method: 'POST', json);
20      request(options, options: function(error, response, body) {
```

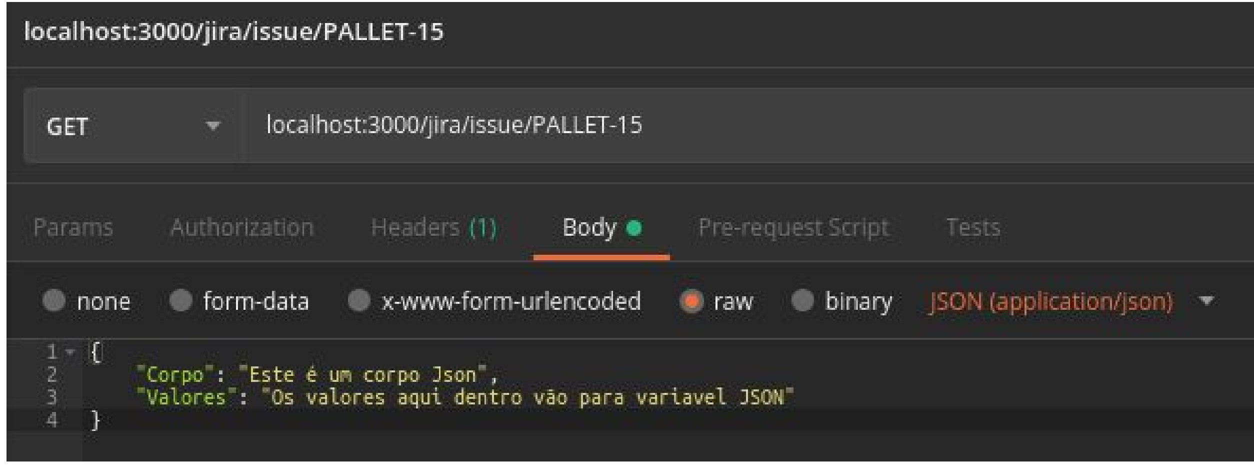
Fonte: Elaborado pelo autor.

Neste pequeno bloco, acontece toda a mágica do processo, onde na primeira linha, onde declaramos a função, passamos como parâmetro “req(requisitos),

[10] Uma forma de fazer pesquisas e criar filtros dentro do software Jira, (RADIGAN, 2017).

res(resposta), next (próximo)”, que são declarações pertinentes ao *middleware express* estes códigos servem para que o express intérprete nossas requisições, na linha 16 apenas dizemos ao código buscar a função *“HeadersForRequest”* dentro do arquivo *JiraModel*. Temos a seguir um exemplo de utilização dos parâmetros do *express*, onde dizemos que uma variável chamada “*json*”, irá receber o valor de “*req.body*” que traduz para “dê-me o corpo da requisição e jogue para este código chamado *json*”, um exemplo do que isto representa utilizando Postman:

Figura 13. Exemplo Postman



Fonte: Elaborado pelo autor.

Após a variável ter sido preenchida, declaramos outra variável que irá receber valores para as opções da requisição, esta é a presente na linha 19, “*options*”. Ela efetua uma chamada para uma função de mesmo nome presente no arquivo *JiraModel* e passa quais valores colocar em cada campo conforme visto anteriormente, neste caso a requisição que o modelo de opções dentro do arquivo de modelo Jira.

Figura 14. Resultado da função Options

```
5   module.exports.options = function('/rest/api/3/issue/', POST, JSON) {
6       return {
7           method: method,
8           url: url,
9           headers: {
```

Fonte: Elaborado pelo autor.

Por fim, conforme figura 12 , na linha 20 é utilizado a biblioteca *request* para receber todos estes parâmetros e efetuar a requisição de fato, como resposta ela irá retornar um erro se for o caso, uma resposta e um corpo de dados, A resposta ou “*response”* neste caso, é apenas um número que indica o sucesso ou erro, conforme respostas em HTTP,(IANA ORG, ca.1983). O código “*body”* traz as respostas de dados desta requisição, por exemplo, efetuado a criação de uma tarefa temos a seguinte resposta no *body.*

Figura 16. Resposta de criação

Fonte: Elaborado pelo autor.

Na parte inferior da tela temos a mensagem “*SucessData*” que nos reporta dados sobre a criação da tarefa, como *id, key,* e a URL de criação, observe que no canto superior direito, contém também o código de status e o tempo de resposta. O bloco de código explicado até aqui é o que efetua a criação e resposta das tarefas usando API para o Jira, o resto de código apresentado serve para validar respostas se der erro ou se der sucesso.

Figura 17. Manipulando Erros

```
21            //handle error parameter
22            if (error === null || !error) {
23                if (body != undefined) {
24                    var errors = body.errors
25                };
26            } else res.send(res.status(400).send({ error: "Error, request error param is not null check create request" }));
27
28            //handle response status if error
29            if (response.statusCode === 400 || response.statusCode >= 400) {
30                console.error('Response Error : ' + response.statusCode, body);
31                if (errors && errors.length > 0) {
32                    res.send({ ErrorData: errors });
33                } else res.send(body);
34                next();
35            };
36
37            //handle response if success
38            if (response.statusCode <= 210 || !response.statusCode >= 400) {
39                console.log('Response Success: ' + response.statusCode, body); // Dev purposes
40                res.send({ SuccessData: body });
41                next();
42            };
43        });
44    };
```

Fonte: Elaborado pelo autor.

Este código tem como escopo, por o bloco entre 22 a 35 manipula erros e retorna uma resposta legível para o desenvolvedor e para o front que pode exibir esta mensagem como aviso para o usuário. O escopo a partir de 38 a 44 faz a manipulação de resposta de sucesso, como visto na figura 13. Este bloco de manipulação de erro está presente em todas as funções de CRUD desta parte do sistema. Todas as funções de tarefas contêm a mesma lógica apresentada anteriormente, segue exemplo as funções de deletar e buscar registro.

Figura 18. deletar

```
47  module.exports.delete = function(req, res, next) {
48
49      jiraModel.HeadersForRequest(req, res);
50      const issue_key = req.params.issue_id;
51      const url = rest_api_issue_url.concat(issue_key);
52
53      const options = jiraModel.options(url,  method: 'DELETE',  bodyData: null);
54      request(options,  options: function(error, response, body) {...});
78  };
```

Fonte: Elaborado pelo autor.

Figura 19. Encontrar

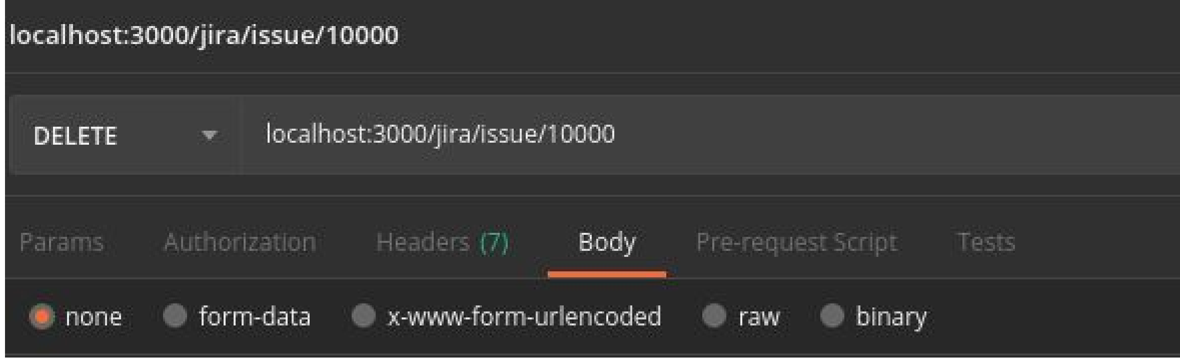
```
81  module.exports.find = function(req, res, next) {
82      jiraModel.HeadersForRequest(req, res);
83      const issue_key = req.params.issue_id;
84      const url = rest_api_issue_url.concat(issue_key);
85
86      const options = jiraModel.options(url,  method: 'GET');
87      request(options,  options: function(error, response, body) {...});
111 };
```

Fonte: Elaborado pelo autor.

Neste escopo mudam poucas coisas, como no deletar e encontrar precisamos passar um *ID* de tarefa, temos alguns parâmetros como “concat” que serve para concatenar textos ou seja formatar textos no código, a *issue_key* interpreta o valor que colocamos na requisição do postman ou seja o ID da tarefa, ambas têm o mesmo conceito, recebem um *id* via URL de requisição e fazem requisições correspondentes,

para o *delete* a requisição é “*DELETE*” e para a *find* a requisição correspondente é “GET”, segue exemplo de como a requisição é feita para deletar um registro:

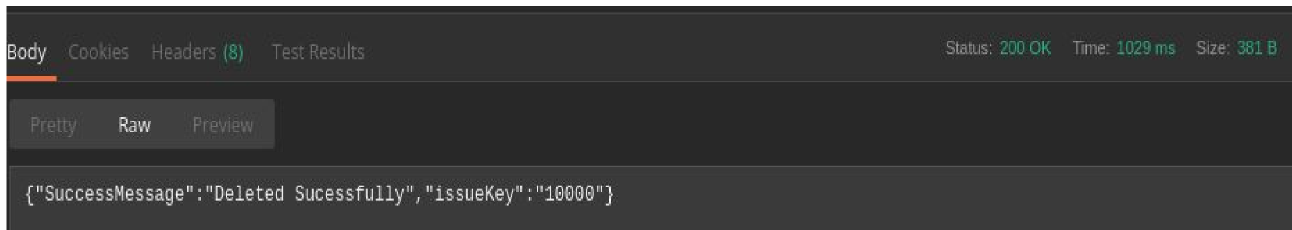
Figura 20. Deletar um registro



Fonte: Elaborado pelo autor.

O body da requisição deve ir vazio, ou nulo como pode ser visto no código de *body-data* na figura 15, observe também que após o “*issue”* colocamos o ID da nossa tarefa anteriormente criada, O método ao lado esquerdo da URL sendo o *delete,* temos assim o seguinte resultado;

Figura 21. Resultado da deleção



Fonte: Elaborado pelo autor.

As outras funções do CRUD, ou seja, atualizar corresponde a mesma metodologia apresentada anteriormente, conforme imagem a seguir corresponde a última operação do crud a atualização;

Figura 22**.** Atualizar registros

```
114  module.exports.update = function(req, res, next) {
115
116      jiraModel.HeadersForRequest(req, res);
117      const issue_key = req.params.issue_id;
118      const url = rest_api_issue_url.concat(issue_key);
119      const json = req.body;
120
121      const options = jiraModel.options(url, method: 'PUT', json);
122      request(options, options: function(error, response, body) {...});
143  };
```

Fonte: Elaborado pelo autor.

Existem outras duas funções que também existem rotas para as mesmas, porém não contemplam nesta versão funcionalidades utilizadas, pelo sistema como um todo, estas são a habilidade de enviar comentários e fazer buscas por JQL dentro do Jira, segue a imagem contendo o código correspondente a ambos, que se baseia no mesmo fluxo de deletar, onde se recebe o *id* da tarefa na URL de requisição, junto com o body, ambas contêm a mesma lógica, contendo a "*issue_key"* que é preenchida com o parâmetro do *id* da tarefa a ser atualizada, a URL que contém alguma peculiaridade baseado na operação a ser feita, por exemplo o comentário deve ter "/*comment"* presente na URL, ambos são requisições do tipo *"POST".*

Figura 22**.** Comentar e encontrar via jql

```
145  module.exports.findByJQL = function(req, res, next) {
146
147      jiraModel.HeadersForRequest(req, res);
148      const url = search_with_jql;
149      const json = req.body;
150      console.log(url);
151      const options = jiraModel.options(url, method: 'POST', json);
152      request(options, options: function(error, response, body) {...});
172  };
173
174  module.exports.comment = function(req, res, next) {
175
176      jiraModel.HeadersForRequest(req, res);
177      const issue_key = req.params.issue_id;
178      const url = rest_api_issue_url.concat(issue_key).concat('/comment');
179      const json = req.body;
180
181      const options = jiraModel.options(url, method: 'POST', json);
182      request(options, options: function(error, response, body) {...});
203  };
```

Fonte: Elaborado pelo autor.

4.2.3 Controlador de Links

Os *links* contêm uma estrutura similar a do controlador de tarefas, contém um *CRUD* que manipula as relações entre tarefas, com a mesma funcionalidade, é possível perceber no código que será exibido a seguir a imensa similaridade, a única alteração real são os *endpoints* das chamadas, pois toda a lógica aplicada no *CRUD* anterior são exatamente as mesmas, os endpoints para o controlador de *link* são os seguintes :

Figura 23. *Endpoints* do controlador de *link*

```
var rest_api_issue_link_url = base_url + '/rest/api/3/issueLink';
var rest_api_issueLink_id = base_url + '/rest/api/3/issue/';
```

Fonte: Elaborado pelo autor.

As operações são basicamente as mesmas, como pode ser observado a seguir com a relação inteira do *CRUD* de *links*.

Figura 24. *Crud* de *Links*

```
11  module.exports.create = function(req, res, next) {
12      jiraModel.HeadersForRequest(req, res);
13
14
15      const json = req.body;
16      const options = jiraModel.options(rest_api_issue_link_url, method: 'POST', json);
17      request(options, options: function(error, response, body) {...});
47  };
48
49
50  module.exports.delete = function(req, res, next) {
51      jiraModel.options(req, res);
52
53      const url = rest_api_issue_link_url.concat('/').concat(exports.link_id);
54      const options = jiraModel.options(url, method: 'DELETE', bodyData: null);
55      request(options, options: function(error, response, body) {...});
80  };
81
82
83  module.exports.find = function(req, res, next) {
84
85      const urlFinal = rest_api_issueLink_id.concat(req.params.issue_id).concat('?fields=issuelinks');
86      const options = jiraModel.options(urlFinal, method: 'GET', bodyData: null);
87      request(options, options: function(error, response, body) {...});
110 };
```

Fonte: Autor

Pode ser observado que o código é virtualmente o mesmo, com a diferença de nomes de variáveis, e algumas peculiaridades de URL, como dita a documentação Atlassian quanto a quais parâmetros passar via API, qual endereço enviar, e qual corpo enviar, (ATLASSIAN, 2004).

4.3 Arquitetura cliente

A estrutura do cliente é parecida com a do servidor, embora contenha muita lógica de apresentação de telas, não é o foco deste trabalho, que visa apenas a utilização da Api e a criação dos códigos de barras, mais importante sendo este. A estrutura utiliza Angular, que por si próprio utiliza muito javascript, daí a aparência similar ao *back-end* que utiliza *Node.* A aplicação angular também faz uso do *express* para rotear suas chamadas de API, pois é assim que o front-end consome as informações no backend, ele faz sua própria requisição para o *back-end* a configuração é exatamente a mesma da

vista anteriormente no item 5.2.1, entretanto a utilização do express também serve para testar em modo local as telas e consumação de dados. A primeira configuração disponível é o serviço de requisição dos dados das tarefas, que faz basicamente a mesma coisa que o modelo de requisição visto no item 5.2.3, porém ao invés de efetuar as chamadas API's para o jira, ele efetua para o servidor backend, conforme imagem temos as configurações de URL que serão usados como *endpoints* :

Figura 25**.** URLS base

```
1  export const API_CONFIG = {
2    SERVER_BASE_URL: 'http://localhost:3000',
3    SERVER_JIRA_URL: 'http://localhost:3000/jira'
4  };
```

Fonte: Elaborado pelo autor.

Estas configurações são utilizadas nas funções apresentadas a seguir. que efetuam as requisições HTTP para o servidor as operações de *CRUD*:

Figura 26**.** *Crud front-end*

```
15  export class IssueService {
16
17    constructor(private httpClient: HttpClient) { }
18
19    create(json: string): Observable<any> {
20      return this.httpClient.post( url: `${API_CONFIG.SERVER_JIRA_URL}/issue/`, json, httpOptions);
21    }
22
23    delete(issueKey: string) {
24      return this.httpClient.delete(`${API_CONFIG.SERVER_JIRA_URL}/issue/`.concat(issueKey));
25    }
26
27    findByKey(issueKey: string): Observable<any> {
28      return this.httpClient.get(`${API_CONFIG.SERVER_JIRA_URL}/issue/`.concat(issueKey), options: { observe: 'response' });
29    }
30
31    update(issueKey: string, json: string): Observable<any> {
32      return this.httpClient.put(`${API_CONFIG.SERVER_JIRA_URL}/issue/`.concat(issueKey), json, httpOptions);
33    }
```

Fonte: Elaborado pelo autor.

Conforme imagem anterior, o processo de requisição de dados do front-end é muito parecido com o back-end, ele efetua a requisição recebendo como parâmetro um corpo *json*, ou uma chave de tarefa, e retorna a resposta HTTP com os dados do

back-end. Da mesma forma o controle de links e requisição no front-end é o mesmo, conforme imagem a seguir,

Figura 27. Controle de links no cliente

```
15  export class IssueLinkService {
16
17    constructor(private httpClient: HttpClient) { }
18
19    create(json: string) {
20
21      return this.httpClient.post(`${API_CONFIG.SERVER_JIRA_URL}/issue-link`, json, httpOptions);
22    }
23
24    delete(issueKey: string) {
25      return this.httpClient.delete(`${API_CONFIG.SERVER_JIRA_URL}/issue-link/`.concat(issueKey));
26    }
27
28    find(issueKey: string): Observable<any> {
29      return this.httpClient.get(`${API_CONFIG.SERVER_JIRA_URL}/issue-link/`.concat(issueKey), httpOptions);
30    }
31
32  }
```

Fonte: Elaborado pelo autor.

Este serviço, como é denominado as funções de operação e chamada *Api* no *Angular*, representa o *CRUD* de operações de *linkar* as tarefas, que o *front* consome para efetuar o relacionamento de tarefas.

4.3.1 Etiquetas

O último serviço relevante neste projeto na parte do *front-end* constitui a habilidade através do sistema de receber uma string contendo o número de uma tarefa do Jira e transformá-la em um código de barra equivalente, o serviço de criar etiquetas se da seguinte forma, uma função de chamada posterior que recebe parâmetros compondo o resultado final, conforme imagem a seguir, onde primeiro temos a configuração da criação que envolve parâmetros de estilo, para formar uma etiqueta apresentável ao usuário;

Figura 28. Configuração da etiqueta de estilo

```
export class BarcodeConfig {

    elementType: string;
    format: string;
    lineColor: string;
    width: number;
    height: number;
    displayValue: boolean;
    fontOptions: string;
    font: string;
    textAlign: string;
    textPosition: string;
    textMargin: number;
    fontSize: number;
    background: string;
    margin: number;
    marginTop: number;
    marginBottom: number;
    marginLeft: number;
    marginRight: number;
    printCSS: string[];
    printStyle: string;

```

Fonte: Elaborado pelo autor.

Neste primeiro bloco temos declarado as variáveis e o tipo de dados que elas receberão, e a seguir os dados passados a tais variáveis por um construtor:

Figura 29. Variáveis recebendo valores

```
24    constructor() {
25      this.format = 'CODE39';
26      this.elementType = 'svg';
27      this.lineColor = '#000000';
28      this.width = 1;
29      this.height = 50;
30      this.displayValue = true;
31      this.fontOptions = '';
32      this.font = 'monospace';
33      this.textAlign = 'center';
34      this.textPosition = 'bottom';
35      this.textMargin = 2;
36      this.fontSize = 15;
37      this.background = '#ffffff';
38      this.margin = 0;
39      this.marginTop = 0;
40      this.marginBottom = 0;
41      this.marginLeft = 0;
42      this.marginRight = 0;
43
44      this.printCSS = ['http://cdn.bootcss.com/bootstrap/3.3.7/css/bootstrap.min.css'];
```

Fonte: Elaborado pelo autor.

Neste bloco acima temos as variáveis recebendo os valores de estilo para criar um código de barra apresentável ao usuário, contendo dados relevantes ao código de barra conforme o código de geração do mesmo, primeiro o construtor criado que recebe a o texto com o nome da tarefa:

Figura 30. Construtor do código de barra

```
3   import { Issue } from 'src/app/models/jira/issue';
4
5   export class BarcodeFactory {
6
7     private html: string;
8
9     constructor(html: string) {
10      this.html = html;
11    }
12
13    getBarcodeHTML() {
14      return this.html;
15    }
16  }
```

Fonte: Elaborado pelo autor.

Após o construtor, existe a função na linha 13 que funciona como chamada reversa, ou seja, ela é chamada em algum ponto do código recebendo o valor de *string* ou texto, da tarefa do Jira. Nas linhas subsequentes temos a função que efetua a construção do código em si, trazendo informações extras e chamando o construtor:

Figura 31. Construtor do *Palletbarcode*

```
export function getPalletBarcode(data_hora: string, issue: Issue, element: any) {
  var html = getPalletBarcodeHtml(data_hora, issue, element);
  return new BarcodeFactory(html);
}
```

Fonte: Elaborado pelo autor.

Está chamada efetua a criação do template do código de barras e seguinte temos o bloco que faz a estilização da imagem final, recebendo *tags* de *html,* conforme imagem:

Figura 32. Template *html* da etiqueta

```
31  function getPalletBarcodeHtml(data_hora: string, issue: Issue, peso: string): string {
32    var html = `
33      <div id="codigo_barra" #codigo_barra style="...">
34        <div style="...">
35          <img style="..." src="http://www.viabrasiltransaereo.com.br/img-l/transportadora-via-brasil-transaereo-lt-br.png" />
36          <img style="..." src="../../../../assets/images/qr_code_viabrasil.png" />
37          <div style="...">
38            <h3>Logística Reversa</h3>
39            <span style="...">São Paulo - (11) 2085-4400 | </span>
40            <br/>
41            <span style="...">www.viabrasil.com.br</span>
42          </div>
43        </div>
44
45        <div class="row" >
46            <div>
47                <h5 style="...">PALLET ID</h5>
48            </div>
49
50            <div style="...">
51              ${document.getElementsByClassName( classNames: 'barcode')[0].innerHTML}
52            </div>
53            <h4 style="...">${data_hora}</h4>
54        </div>
55
56        <h2 style="...">${issue.summary}</h4>
57        <h2 style="...">${peso}</h4>
58      </div>`;
59
60    return html;
61  }
```

Fonte: Elaborado pelo autor.

O resultado pode ser observado na seguinte imagem:

Figura 33. Resultado

Fonte: Elaborado pelo autor.

5 Validação

Para a validação deste projeto, foi utilizado algumas métricas, utilizando um método analítico de comparação de resultados extraídos com objetivos do projeto, relatando o sucesso do experimento, os cenários propostos foram:

1. Manual versus automático:
    a. Como é feito a geração de etiquetas pelo Jira atualmente de forma manual comparado a forma criada neste projeto.
2. Métrica de geração de etiquetas automáticas:
    a. Quantidade de Etiquetas Geradas em 2 minutos
    b. Quantidade máxima de etiquetas em uma impressão
3. Criação de tarefas manual versus automático:

a. Três usuários, utilizando uma aplicação Jira criando 10 tarefas cada, comparado a utilização da requisição de criação da Api, usando dados não variáveis.

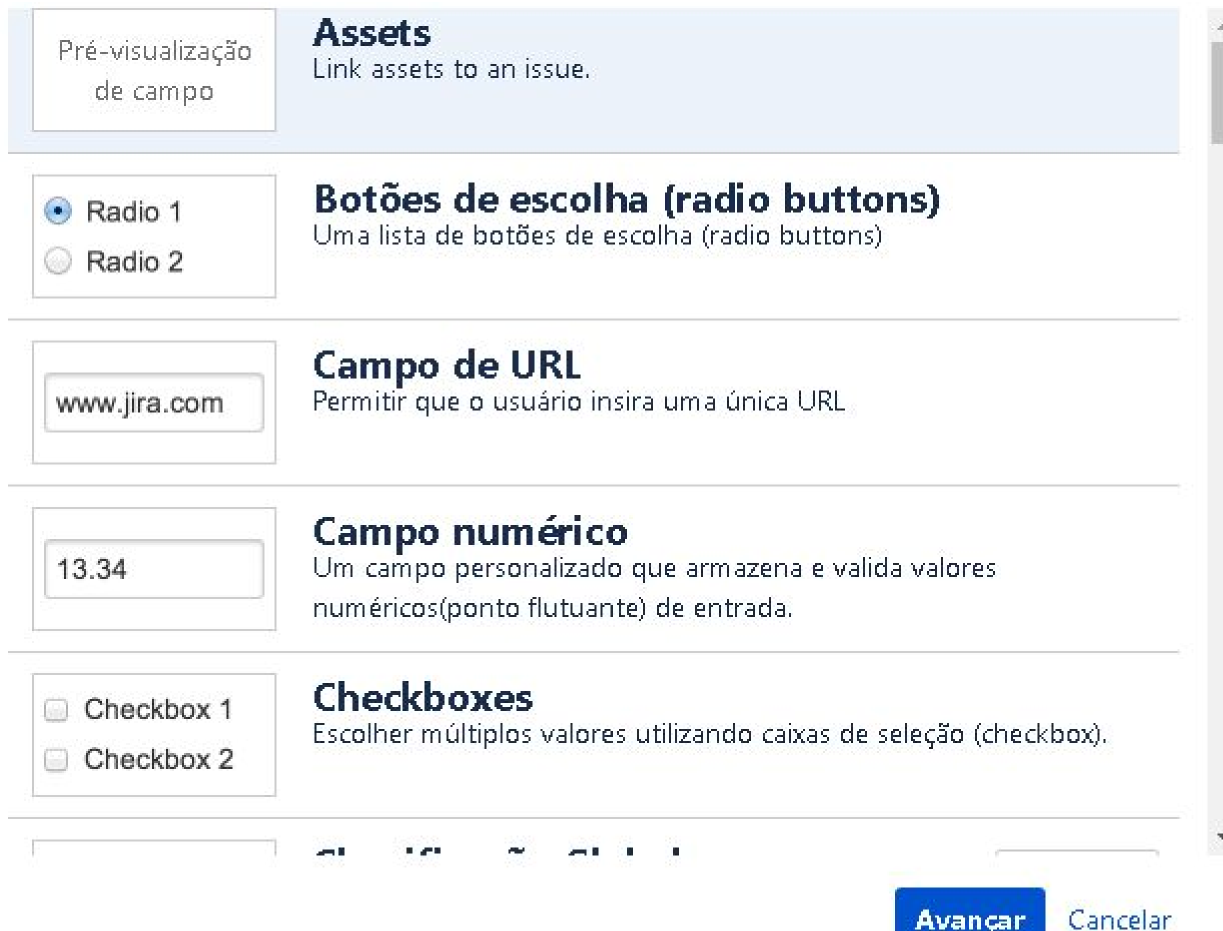
5.1 Manual versus automático

Este experimento foi conduzido de acordo com a necessidade de averiguar a diferença entre gerar códigos de barras no Jira, comparando-o com a geração feita pela solução proposta.

5.1.1 Gerando etiquetas pelo Jira

Atualmente, existe na plataforma da Atlassian, uma forma concreta de gerenciar ativos, que normalmente, é o foco de se ter códigos de barras e etiquetamento, com o propósito de gerenciar itens. A plataforma permite uma estrutura coordenada, conforme documentação, Ray (2014). Porém, fica evidente o problema, não existe nenhuma forma de gerar etiquetas utilizando somente o Jira, ou até mesmo plugins que de forma fácil permitem este tipo de solução, ou seja, não é possível gerar nenhuma forma de etiqueta, baseado em tarefas, ou em numeração de tarefas provindas somente do Jira. Desta forma, navegando por todas as configurações disponíveis no Jira nativo, e criando todos os tipos de campos disponíveis, fica claro a impossibilidade de efetuar esta operação, conforme imagem, demonstra alguns dos tipos de campos disponíveis para criação;

Figura 34. Lista de Campos

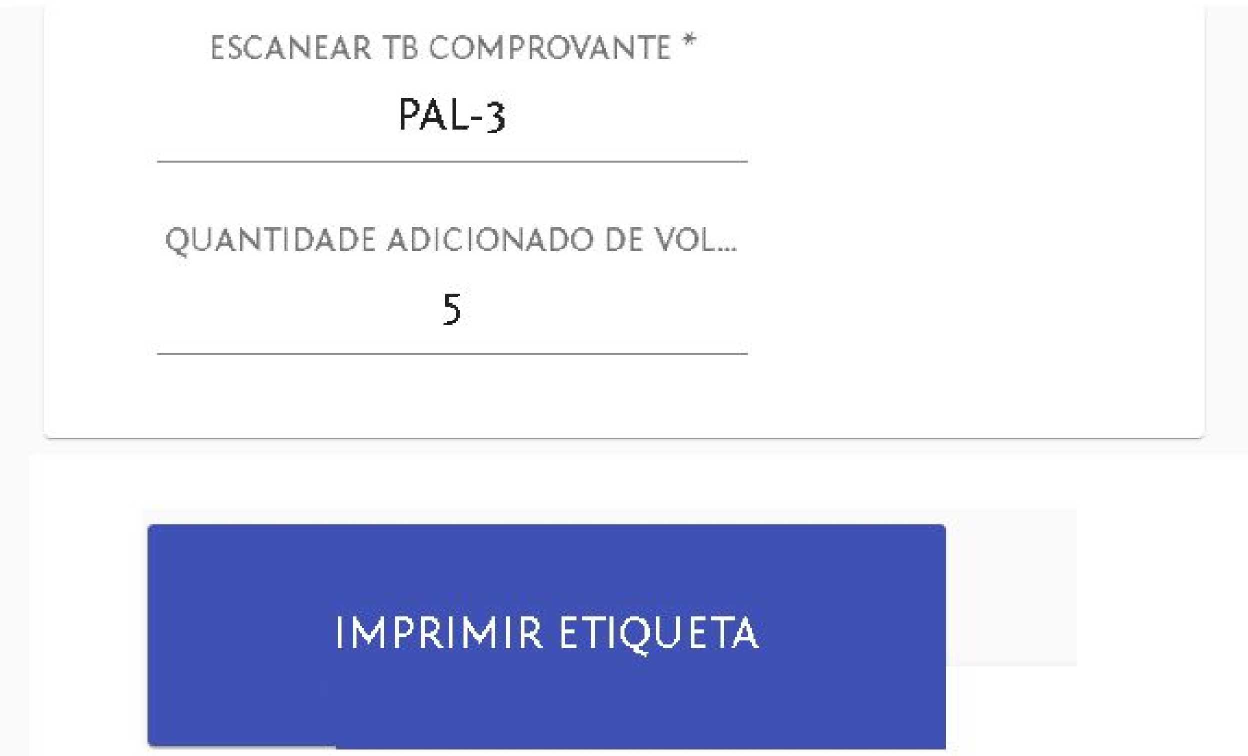


Fonte: Elaborado pelo autor.

5.1.2 Gerando etiquetas automaticamente

Comparando com o método disponível na aplicação Jira, temos uma aplicação simples, porém potente, que consegue criar códigos de barra equivalente as tarefas do Jira contendo a nomenclatura desejável para qualquer tipo de cliente, neste caso um varejista que lida com pallets de mercadoria, o primeiro passo é acessar a aplicação e na aba de “*checkout*” preencher os campos necessários e clicar para “imprimir” as etiquetas, e pronto, recebemos um pop-up contendo-as e pronto para impressão, conforme imagens a seguir;

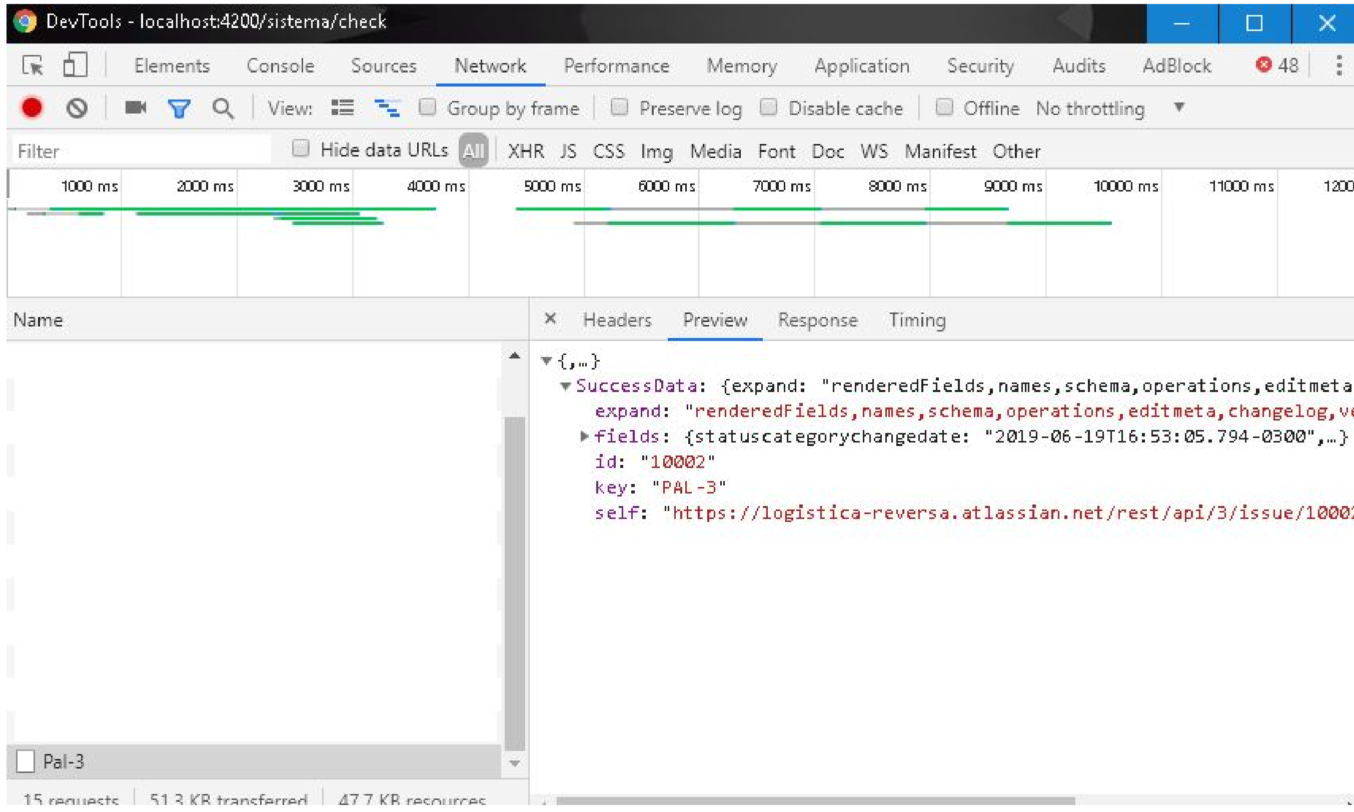
Figura 35. Página para preenchimento de dados



Fonte: Elaborado pelo autor.

Nesta tela, o campo “Escanear TB Comprovante”,e é responsável por buscar a tarefa, neste caso a “PAL-3”, e abaixo temos a quantidade de etiquetas que serão impressas, ou “volumes” existentes, a busca de tarefa é comprovada pela aba de chamadas e respostas, no console de desenvolvedor do Google Chrome, conforme imagem:

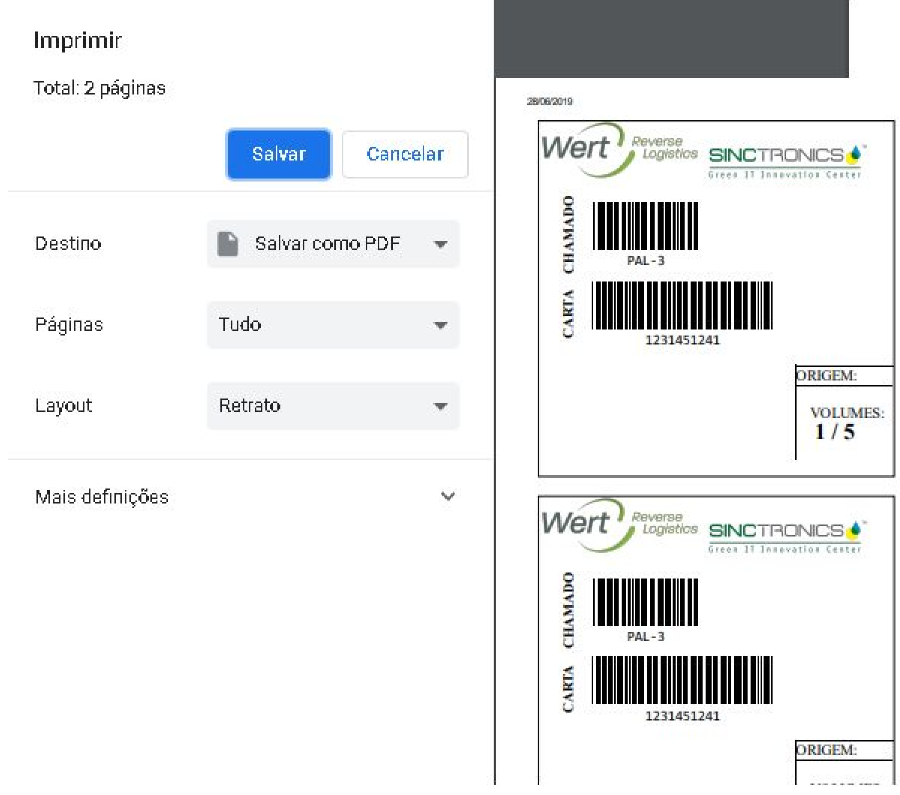
Figura 36. Chamada Api



Fonte: Elaborado pelo autor.

Está figura acima, representa no ambiente de desenvolvimento do navegador Google, uma requisição API, para o servidor Jira, mais especificamente a resposta após a requisição do tipo GET, na aba “*Preview”* contém os dados de sucesso da requisição, sendo estes “*Fields* “ ou campos da tarefa que foi buscada (Pal-3), o *ID* da tarefa sendo este o identificador único da mesma, uma chave ou “*key*” que é um método de catalogar tarefas do próprio JIRA, e o *link* ao qual a chamada foi feita. Após a busca trazer os resultados e o usuário preencher os campos com os dados que desejar, como volume de pallets e afins, o mesmo imprime as etiquetas clicando no botão de impressão, o resultado é o seguinte;

Figura 37. Resultado da impressão



Fonte: Elaborado pelo autor.

5.2 Métrica de Geração de Etiquetas

O primeiro critério para este teste, foi a quantidade de etiquetas que eram possíveis gerar sem estourar o sistema com requisições, foi estipulado que o tempo limite, era de 2 minutos, após este tempo, o sistema perdia toda a estabilidade e perdia totalmente a funcionalidade, cabe dizer, que a quantidade de etiquetas criadas, dificilmente será alcançada pela utilização normal deste sistema, sendo esta acima de quinhentos mil.

Quadro 1 - Cenários de validação

Caso	**Dados**	**Tempo**
Estresse de criação de etiquetas em determinado espaço tempo.	Fixos, sem variação de dados ou quantidade.	2 minutos ou 120000 ms

Quadro 1 - Cenários de validação

(Conclusão)

Caso	Dados	Tempo
Quantidade de volumes em uma única impressão utilizando valor surreal.	Fixos, sem variação de tarefa, somente quantidade	Instantâneo
Quantidade de volumes em uma única impressão utilizando um valor mais próximo do real	Fixos, sem variação de tarefa, somente quantidade	Instantâneo

Fonte: Elaborado pelo autor, 2019.

5.2.1 Teste de estresse

Para este teste, foi gerado um script de teste de criação, aguardando o resultado conforme criação e tempo estabelecido foi utilizado, outra interface de desenvolvimento para efetuar os testes, segue o script feito.

Figura 38. Script de estresse.

```
144    getBarcodeHtml() {
145      let html = '';
146      let dataInicial = new Date();
147      let dataFinal;
148      console.log("Começando a gerar as etiquetas em: ", dataInicial);
149      var vetorHTML = [];
150      var x = 0;
151      for(var i = 0;i<9999999999999999999999999999;i++){
152          vetorHTML[x] += barcodeFactory.getTBLEXBarcode(i, this.uf, 99999).getBarcodeHTML();
153          vetorHTML[x] += '</br>';
154          if(vetorHTML[x].length >= 1000000){
155            x++;
156          }
157          dataFinal = new Date();
158          if(dataInicial.getMinutes()+2 === dataFinal.getMinutes()){
159            console.log("Número de Etiquetas geradas para impressao em 2 minutos:", i);
160            for(var y = 0;y<x;x++){
161              this.printComponent.print(vetorHTML[y]);
162            }
163            return null;
164          }
165        }
```

Fonte: Elaborado pelo autor.

Neste código, é criado variáveis para controlar a data e hora do início da geração de tarefas, é criado um vetor que irá receber as *strings* pois como estamos trabalhando com uma grande quantidade de texto, é impossível receber em uma variável única do tipo string, como normalmente é feito. É criado um algoritmo que irá se repetir por alguns n ciclos , ou seja, só irá terminar se atingir este número, ou conforme linha 158, se passe dois minutos, durante este tempo é feito a contagem de etiquetas geradas a partir da linha 154. O resultado foi o seguinte.

Figura 39. Resultado do teste de estresse

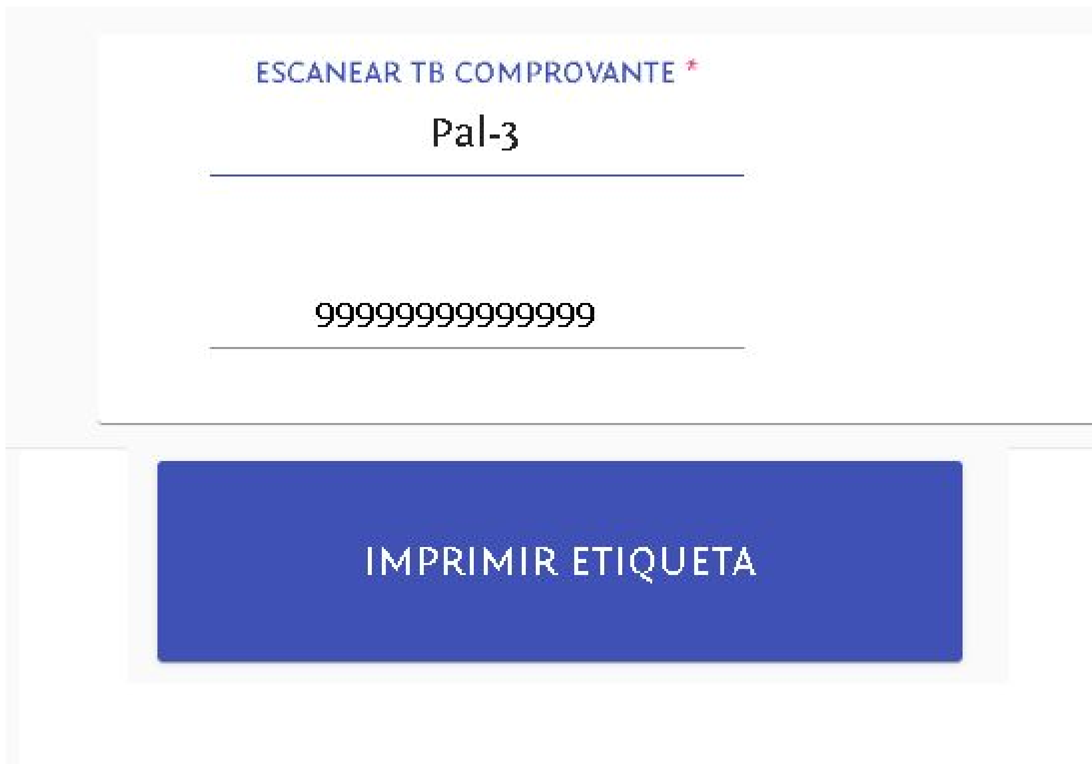
```
DevTools - localhost:4200/sistema/check
Elements  Console  Sources  Network  Performance  Memory  Application
top  Filter  Default levels
Angular is running in the development mode. Call enableProdMode() to enable the     core.js:16829
production mode.
Começando a gerar as etiquetas em:  Sat Jun 22 2019 21:48:29 GMT-0300     check.component.ts:148
(Brasilia Standard Time)
Número de Etiquetas geradas para impressao em 2 minutos: 621793     check.component.ts:159
```

Fonte: Autor

5.2.2 Teste de múltiplas etiquetas

Para este caso, simplesmente via interface de geração de etiquetas, foi posto um número absurdo de volume, e verificado se a aplicação se compromete a entregar tal quantidade, o teste se refere a seguinte.

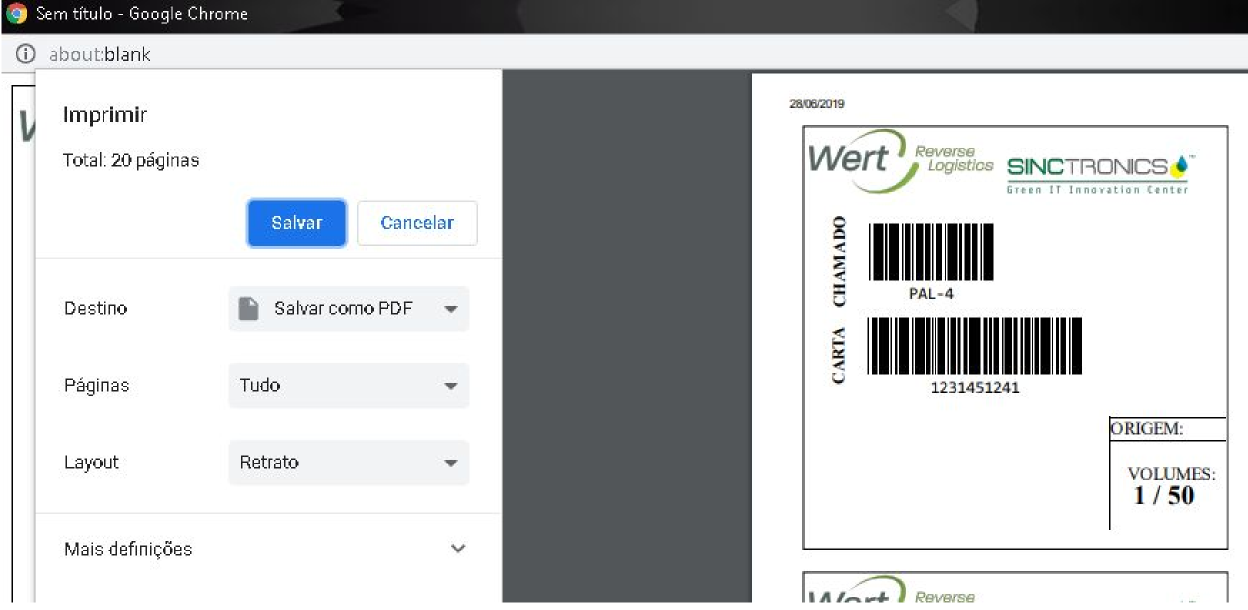
Figura 40. Teste de múltiplas impressões



Fonte: Elaborado pelo autor.

Conforme o teste, a averiguação de uma alta quantidade de volumes impressos se mostrou falho, a aplicação não responde e não gera o pop-up, então, foi testado com um valor mais próximo do que poderá ser utilizado em um ambiente corporativo, tendo em mente uma grande empresa que carrega, por exemplo, 50 caixas do mesmo produto, o teste seguinte será feito, utilizando este critério, e o resultado foi o seguinte:

Figura 41. Resultado de teste realista



Fonte: Elaborado pelo autor.

5.3 Criação de Tarefa Manual vs Automático

Para este caso, foi feito a utilização de três usuários, sendo estes dois proficientes com a ferramenta Jira e outro não proficiente, foram solicitados que criassem dez tarefas em cada uma da instância do Jira, tendo o resultado seguinte.

Figura 42. Kanban de Tarefas Jira

LISTA DE PENDÊNCIAS 30
Teste 01 - Leo Lothamer 01 PAL-2
Teste 02 - Leo Lothamer 02 PAL-3
Teste 03 - Leo Lothamer 03 PAL-4
Teste 04 - Leo Lothamer 04 PAL-5

LISTA DE PENDÊNCIAS 30
BUG001 - Fix this thing PAL-12
BUG002 - Fix this other thing PAL-13
BUG003 - The site broke!!!! PAL-14
BUG004 - Revert the fix of BUG-002. PAL-15

LISTA DE PENDÊNCIAS 30
Teste 02 - Mário PAL-23
Teste 03 - Mário
Teste 03 - Mário PAL-24
Teste 04 - Mário
Teste 04 - Mário PAL-25
Teste 05 - Mário
Teste 05 - Mário

Fonte: Elaborado pelo autor.

Neste kanban existem 30 tarefas, criadas por 3 usuários diferentes, cada um com respectivos tempos de criação diferentes, agora, como resposta de requisição Api, temos o tempo de uma tarefa criada, sendo este o maior tempo registrado durante o processo:

Figura 43. Criação via postman

Body Cookies Headers (8) Test Results Status: 200 OK Time: 2124 ms Size: 440 B

Pretty Raw Preview JSON


```
1 {
2     "SuccessData": {
3         "id": "10031",
4         "key": "PAL-32",
5         "self": "https://logistica-reversa.atlassian.net/rest/api/3/issue/10031"
6     }
7 }
```


Fonte: Elaborado pelo autor.

Para receber os dados de tempo de criação das tarefas, foi efetuado um bloco de script que cria 30 tarefas e retorna o tempo em milissegundos que cada tarefa levou para ser criada, o script é o seguinte;

Figura 44. Script de criação de tarefas

```
12 module.exports.create = function (req, res, next) {
13
14     for(var i=0; i === 30; i++) {
15
16         jiraModel.HeadersForRequest(req, res); //headers
17         let json = req.body;
18         let time = true;
19
20
21         const options = jiraModel.options(rest_api_issue_url, method: 'POST', json, time);
22         console.log(options);
23         request(options, options: function (error, response, body) {
24             var numbs = response.elapsedTime++
25
26             console.log(numbs);
27         })
28     }
29 };
```

Fonte: Elaborado pelo autor.

Este nada mais é que o mesmo bloco de criação, porém está encapsulado dentro de um repetidor, que efetua trinta vezes a mesma operação, retornando os “*numbs*”, variável que aloja o tempo de cada tarefa individualmente. A fim de compararmos ambas soluções, temos a seguinte tabela para criação feita pelos usuários;

Tabela 1. Análise da criação Manual

Tipo	Responsável	Tempo
Criação manual	Leonardo Lothamer, Desenvolvedor *Full Stack*, com conhecimentos Jira	00:08:52 (Oito minutos, cinquenta e dois segundos).
Criação manual	Arthur Balestro, Desenvolvedor *Back end,* sem conhecimentos Jira	00:17:09 (Dezessete minutos, nove segundos).
Criação manual	Mário de Araújo, Desenvolvedor Full Stack, com conhecimentos Jira	00:04:50 (Quatro minutos, cinquenta segundos)

Fonte: Elaborado pelo autor, 2019.

A seguir para comparação, a tabela com os dados das criações automáticas.

Tabela 2. Tempo de criação automático de tarefas Jira

Tipo	Responsável	Tempo
Criação automática	Sistema	1037 ms[11]
Criação automática	Sistema	1045 ms
Criação automática	Sistema	1103 ms
Criação automática	Sistema	1093 ms
Criação automática	Sistema	1105 ms
Criação automática	Sistema	1130 ms
Criação automática	Sistema	1139 ms
Criação automática	Sistema	1149 ms

[11] Abreviação de milisegundo

Tabela 2. Tempo de criação automático de tarefas Jira

(Conclusão)

Tipo	Responsável	Tempo
Criação automática	Sistema	1172 ms
Criação automática	Sistema	1152 ms
Criação automática	Sistema	1176 ms
Criação automática	Sistema	1174 ms
Criação automática	Sistema	1197 ms
Criação automática	Sistema	1217 ms
Criação automática	Sistema	1227 ms
Criação automática	Sistema	1254 ms
Criação automática	Sistema	1258 ms
Criação automática	Sistema	1295 ms
Criação automática	Sistema	1335 ms
Criação automática	Sistema	1371 ms
Criação automática	Sistema	1390 ms
Criação automática	Sistema	1400 ms
Criação automática	Sistema	1427 ms
Criação automática	Sistema	1466 ms
Criação automática	Sistema	1494 ms
Criação automática	Sistema	1610 ms
Criação automática	Sistema	1572 ms
Criação automática	Sistema	1656 ms
Criação automática	Sistema	1690 ms
Criação automática	Sistema	1723 ms

Fonte: Elaborado pelo autor, 2019.

Tendo em mão as duas tabelas contendo as métricas de tempo de geração, mesmo sem cálculo já fica fácil perceber qual das mesmas tem o maior ganho de tempo, quando utilizado o script de gerar tarefas, foi feito instantaneamente. Claro que quando se trata de gerar tarefas dinâmicas, ou seja contendo campos diferentes para cada caso, se resguarda o fato de ter scripts diferentes, mas ainda sim, se mantém rápido, uma vez que só é necessário alterar um ou dois campos. Segue a tabela final contendo a média de tempo entre ambos métodos:

Quadro 2. Média Final

Tipo	Média
Manual	Para estes participantes a média foi de 00:10:50 (dez minutos, cinquenta segundos)
Automático	Para trinta tarefas a média foi de 1362 (mil trezentos e sessenta e dois milissegundos), que convertendo dá um total de 1,3(segundos).

Fonte: Elaborado pelo autor, 2019.

6 CONCLUSÃO

A finalização deste projeto mostrou algumas falhas substanciais e muito sucesso referente a agilidade, utilização de ferramentas inovadoras, e a criação de um protótipo funcional de algo não existente na plataforma Jira, as falhas consistem em ter uma pouca flexibilidade, troca-se agilidade por flexibilidade no quesito de criar tarefas, enquanto que atualizá-las torna-se uma tarefa simples somente se os campos estiverem presentes em um formulário fixo, onde o usuário não tenha que preencher uma grande quantidade de informações, contudo, o objetivo era construir uma aplicação ágil e simples, o que cumpre de forma memorável, tanto a criação de tarefas quanto a ênfase principal, está sendo o processo de gerar etiquetas e códigos de barra. Existem implementações a serem feitas, como cadastro funcional de usuários, interface de usuário finalizada, implementação online e também testes que se fazem necessários, como por exemplo, máquinas que leem código de barras, conseguem interpretar a numeração criada pelo aplicativo e utilizar de forma coerente em seus subsistemas, entretanto, para o escopo deste projeto, conclui-se que foi bem sucedido a criação e aperfeiçoamento a nível alfa da aplicação com arquitetura Rest Api, e implementação de um subsistema que gere e imprima etiquetas com códigos de barras.

8 REFERÊNCIAS BIBLIOGRÁFICA

ANGULAR. Introduction to Angular Docs. 2010. Disponível em: <https://angular.io/docs>. Acesso em: 26 abr. 2019.

ARAÚJO, Ronaldo. O que é um CRUD. [*S. l.*], [2007-2018]. Disponível em: http://wordpressdescomplicado.com.br/o-que-e-um-crud/#more-36. Acesso em: 17 abr. 2019.

ATLASSIAN. Jira Software. [S. l.], 2019. Disponível em: https://br.atlassian.com/software/jira. Acesso em: 6 fev. 2019.

ATLASSIAN. The jira cloud platform REST API. [*S. l.*], [entre 2004 e 2016]. Disponível em: https://developer.atlassian.com/cloud/jira/platform/rest/v3/. Acesso em: 18 jun. 2019.

ATLASSIAN SUPPORT. Atlassian Cloud System. [*S. l.*], 7 out. 2015. Disponível em: https://confluence.atlassian.com/cloudkb/atlassian-cloud-system-691011820.html. Acesso em: 1 mar. 2019.

BERTULUCCI, Cristiano Silveira. Entenda como Funciona o Protocolo TCP-IP. [*S. l.*], [2005 e 2017]. Disponível em: https://www.citisystems.com.br/protocolo-tcp-ip/. Acesso em: 8 jun. 2019

BRYON, Yo. NGX-Barcode. [S. l.], 2017. Disponível em: https://www.npmjs.com/package/ngx-barcode. Acesso em: 6 mar. 2019.

FIELDING, Roy *et al*. Hypertext Transfer Protocol. [*S. l.*], 1 set. 2004. Disponível em: https://www.w3.org/Protocols/rfc2616/rfc2616.html. Acesso em: 2 jun. 2019.

IANA ORG. HTTPS Status code registry. [*S. l.*], ca.1983. Disponível em: https://www.iana.org/assignments/http-status-codes/http-status-codes.xhtml. Acesso em: 18 jun. 2019.

IWAYA, Akemi. What is the difference between 127.0.0.1 and 0.0.0.0. [*S. l.*], 16 ago. 2015. Disponível em: https://www.howtogeek.com/225487/what-is-the-difference-between-127.0.0.1-and-0.0.0.0/. Acesso em: 18 jun. 2019.

KRUKOWSKI, Ilya Bodrov. Angular Introduction: What It Is, and Why You Should Use It. [*S. l.*], 22 abr. 2018. Disponível em: https://www.sitepoint.com/angular-introduction/. Acesso em: 8 maio 2019.

MASSÉ, Mark. Rest API Design Rulebook. USA: O'Reilly Media, Inc, 2012. Disponível em: https://pepa.holla.cz/wp-content/uploads/2016/01/REST-API-Design-Rulebook.pdf. Acesso em: 6 maio 2019.

MDN. HTML5. [*S. l.*], 23 abr. 2019. Disponível em: https://developer.mozilla.org/pt-BR/docs/Web/HTML/HTML5. Acesso em: 2 jun. 2019.

MDN. Node.js. [*S. l.*], 2019. Disponível em: https://developer.mozilla.org/en-US/docs/Glossary/Node.js. Acesso em: 6 jun. 2019.

MDN MOZILLA. Strict mode - Javascript. [*S. l.*], [2005-2019]. Disponível em: https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Strict_mode. Acesso em: 18 jun. 2019.

MICROSOFT. Windows Network Architecture and the OSI Model. [*S. l.*], 19 abr. 2017. Disponível em: https://docs.microsoft.com/en-us/windows-hardware/drivers/network/windows-network-architecture-and-the-osi-model. Acesso em: 16 maio 2019.

NODE.JS FOUNDATION. About. [*S. l.*], [ca.2011]. Disponível em: https://nodejs.org/en/about/. Acesso em: 8 fev. 2019.

NODEJS. Web Survey Report 2018. [*S. l.*], 2018. Disponível em: https://nodejs.org/en/user-survey-report/. Acesso em: 2 maio 2019.

NODE JS V12.4 DOC. Buffer. [*S. l.*], 2019. Disponível em: https://nodejs.org/api/buffer.html. Acesso em: 18 jun. 2019.

POSTMAN INC. Postman. [*S. l.*], 2019. Disponível em: https://www.getpostman.com/products. Acesso em: 18 jun. 2019.

RADIGAN, Dan. JQL: The most flexible way to search jira. [*S. l.*], 29 jan. 2017. Disponível em: https://www.atlassian.com/blog/jira-software/jql-the-most-flexible-way-to-search-jira-14. Acesso em: 18 jun. 2019.

REDAÇÃO OFICINA. O Protocolo HTTP. [*S. l.*], 3 set. 2007. Disponível em: https://www.oficinadanet.com.br/artigo/459/o_protocolo_http. Acesso em: 21 maio 2019.

REVANTHKUMAR, Setikam et al. JavaScript. [S. l.], 13 jun. 2019. Disponível em: https://developer.mozilla.org/en-US/docs/Web/JavaScript. Acesso em: 13 jun. 2019.

RICHARDSON, Chris. What are MicroServices. [*S. l.*], [entre 2014 - 2018]. Disponível em: https://microservices.io/. Acesso em: 19 jun. 2019.

SCHOTT, Fred k *et al*. Request. [*S. l.*], 2018. Disponível em: https://www.npmjs.com/package/request. Acesso em: 25 abr. 2019.

SCHULTHESS, Coline. History of REST APIs. [*S. l.*], 26 jan. 2017. Disponível em: https://www.mobapi.com/history-of-rest-apis/. Acesso em: 10 abr. 2019.

STRONGLOOP, IBM. Express - NodeJS web application framework. [*S. l.*], 2017. Disponível em: https://expressjs.com/. Acesso em: 5 mar. 2019.

STRONGLOOP INC. Roteamento básico no Express. [*S. l.*], 2012 e 2019. Disponível em: https://expressjs.com/pt-br/starter/basic-routing.html. Acesso em: 18 jun. 2019.

VERMA, Sanjna. Apis Versus web services. [*S. l.*], 18 jan. 2018. Disponível em: https://blogs.mulesoft.com/dev/api-dev/apis-versus-web-services/. Acesso em: 5 jun. 2019.

W3C. Soap. [*S. l.*], 2000. Disponível em: https://www.w3.org/TR/soap/. Acesso em: 16 maio 2019.

WHAT is REST. [*S. l.*]: REST API Tutorial, [ca.2012]. Disponível em: https://www.restapitutorial.com/lessons/whatisrest.html. Acesso em: 16 maio 2019.

WILSON, Doug. Body-Parser. [*S. l.*], 2019. Disponível em: https://www.npmjs.com/package/body-parser. Acesso em: 28 maio 2019.