UNIVERSIDADE FEDERAL DO ESTADO DO RIO DE JANEIRO
CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA
ESCOLA DE INFORMÁTICA APLICADA

Um exemplo do desenvolvimento em NodeJS e suas etapas

Hugo José dos Santos Pereira Bertoche

Orientador
Márcio de Oliveira Barros

RIO DE JANEIRO, RJ – BRASIL
AGOSTO DE 2020

Catalogação informatizada pelo autor

dos Santos Pereira Bertoche, Hugo José
d545 Um exemplo do desenvolvimento em NodeJS e suas etapas / Hugo José dos Santos Pereira Bertoche. -- Rio de Janeiro, 2020.
36

Orientador: Márcio de Oliveira Barros.
Trabalho de Conclusão de Curso (Graduação) - Universidade Federal do Estado do Rio de Janeiro, Graduação em Sistemas de Informação, 2020.

1. Node.js. 2. Docker. 3. Microsserviços. 4. Metodologias Ágeis. I. de Oliveira Barros, Márcio, orient. II. Título.

Um exemplo do desenvolvimento em NodeJS e suas etapas

Hugo José dos Santos Pereira Bertoche

Projeto de Graduação apresentado à Escola de Informática Aplicada da Universidade Federal do Estado do Rio de Janeiro (UNIRIO) para obtenção do título de Bacharel em Sistemas de Informação.

Aprovado por:

_____

Prof. Márcio de Oliveira Barros, DSc (UNIRIO)

_____

Prof. Gleison dos Santos Souza (UNIRIO)

RIO DE JANEIRO, RJ – BRASIL.
AGOSTO DE 2020

Agradecimentos

Primeiramente, gostaria de agradecer aos meus pais, Glória e Sérgio, pelo constante apoio às minhas decisões, sempre me auxiliando nos momentos mais difíceis, e me educando para ser cada vez uma pessoa melhor.

À minha melhor amiga, Bianca, pelo incessável apoio, paciência e por sempre me cobrar a me dedicar mais.

Ao meu irmão, Uriel, por ter me incentivado a escolher o curso de Bacharelado de Sistemas de Informação, sem ele eu possivelmente não estaria aqui agora.

Aos meus amigos por sempre me ajudarem a esvaziar a cabeça e a relaxar nos momentos de maior dificuldade e tensão.

Ao meu orientador Márcio Barros, por sempre ter se disposto a me ajudar a melhorar, corrigir, e entregar um trabalho cada vez melhor.

A todo o corpo docente, por sempre se fazerem presentes na vida do aluno durante o processo de Graduação, nos ensinando a sermos melhores profissionais na área de TI.

A todos os alunos com quem compartilhei algum momento aqui na Unirio, e aos que ainda estão se graduando, não desistam, pois é recompensador terminar essa jornada.

Ao corpo administrativo por sempre ter ajudado e estado presente ao longo de todo o período de graduação.

Aos meus amigos do trabalho, pois graças a eles pude realizar o estudo feito aqui, e pelo constante apoio e compreensão para finalizar o trabalho final.

E por fim, a todos aqueles que possibilitaram, e mantêm, a existência da Universidade Federal do Estado do Rio de Janeiro, mantendo-a como uma instituição de ensino superior pública, de qualidade, preocupada não só com a formação de profissionais, mas de seres humanos.

Obrigado a todos!

RESUMO

No crescente cenário de desenvolvimento em Node.js, Docker, soluções em nuvem, estruturação de aplicações em microsserviços e metodologias Agile, focamos em abordar como uma grande empresa usa essas tecnologias e metodologias. Para isso, utilizamos de um caso real, analisando as tecnologias e metodologias, como elas funcionam e os benefícios que elas trazem, trazendo informações a respeito de uma outra perspectiva sobre o desenvolvimento de software para os alunos do curso de Bacharelado em Sistemas de Informação da UNIRIO.

Palavras-chave: Node.js, Docker, Kubernetes, Agile, Microsserviços.

ABSTRACT

In a growing scenario of software development using Node.js, Docker, cloud solutions, structuring applications in microservices and agile methodologies, we focus on addressing how a large company implements these technologies and methodologies. To present such a scenario, we will use a real case scenario as an example, analyzing the technologies and methodologies, exploring its functionalities and benefits they bring, in the hope of showing a different perspective about software development to the students at the Information Systems School at UNIRIO.

Keywords: Node.js, Docker, Kubernetes, Agile, Microservices.

Índice

Índice de Figuras

1 Introdução

1.1 Motivação

No mundo atual, as novas tecnologias surgem com uma frequência cada vez maior. No âmbito da programação, esta dinâmica não é diferente. O surgimento das novas tecnologias para desenvolvimento de aplicações Web faz com que muitas tecnologias fiquem ultrapassadas e menos relevantes em um espaço de tempo muito pequeno se considerarmos tecnologias de desenvolvimento de software anteriores.

Manter-se atualizado não é apenas relevante para a indústria se manter nas novidades, mas para desenvolver soluções que tecnologias passadas não eram capazes de resolver. O desenvolvimento da tecnologia se dá justamente para resolver tais problemas que naquele momento não se via uma solução com as tecnologias existentes.

Neste sentido, este documento apresenta um projeto que está utilizando tecnologias modernas para viabilizar uma nova forma de desenvolver aplicações Web: programação na nuvem, com aplicações conteinerizadas e em ambientes isolados usando Node.js. Este projeto será utilizado como um exemplo para apresentar e discutir as tecnologias nas quais ele se apoia.

O Node.js (Nodejs, 2020) vem ganhando cada vez mais relevância e importância por simplificar o desenvolvimento *backend* usando JavaScript. O mercado começou a usá-lo para criar aplicações Web de forma rápida e escalável. Dito isto, no curso de BSI da Unirio não chegamos a nos aprofundar na linguagem JavaScript, de grande relevância para o mercado atual. Assim, vimos a possibilidade de abordar o tema neste projeto de conclusão de Graduação e informações a respeito das vantagens do Node.js.

O exemplo em estudo se trata de uma aplicação responsável por gerar taxas de financiamento na qual anteriormente utilizava-se de planilhas Excel e um sistema em tecnologias legadas que não podemos descrever por conta de confidencialidade. O meu papel nessa equipe era atuar como desenvolvedor, auxiliando também na definição dos requisitos por estar trabalhando com o sistema a mais tempo e com isso apoiar em decisões técnicas.

O novo sistema foi estruturado em microsserviços *dockerizados*, isto é, com cada microsserviço responsável por uma parte da aplicação, como a interface, conexão com o banco de dados, serviço de *login*, entre outras. O projeto utilizou o *OpenShift*[11], ferramenta desenvolvida pela Red Hat, baseada no *Kubernetes*[22], para orquestrar os contêineres onde rodam os microsserviços, gerenciar a rede, o armazenamento, a distribuição de carga, roteamento, monitoramento, segurança e escalabilidade.

Com isso, vimos também a possibilidade de abordar temas como metodologias de desenvolvimento, tecnologias que o mercado vem utilizando na infraestrutura (como banco de dados), estruturação dos microsserviços, além da implantação contínua, que hoje em dia é um dos tópicos mais relevantes para aplicações Web, seguindo a linha de *DevSecOps* (desenvolvimento, segurança e operações), pensando no desenvolvimento conectado com a operação, garantindo que os dados e a aplicação estejam menos suscetíveis a falhas de segurança e visando a aplicação bem estruturada, segura e que atenda às necessidades do cliente.

Com esse projeto usando tecnologias como Docker (Docker Documentation, 2020), OpenShift[1], Node.JS (Nodejs, 2020), achamos que seria uma oportunidade de descrever a experiência de como é desenvolver utilizando essas tecnologias, e explicar mais e mais sobre como elas funcionam.

O projeto surgiu como uma forma de automatizar um processo de atualização que demorava três dias por pessoa para conseguir realizar o processo para um país. O sistema contava com mais de 32 países, o que gerava muito atraso e muita suscetibilidade a erro. No novo projeto em questão, para realizar o processo para um país são necessários alguns cliques e em 30 minutos o país teria suas novas taxas de financiamento prontas para aplicar nos produtos ofertados através do sistema.

As tecnologias utilizadas foram escolhidas por agilizarem o desenvolvimento e facilitarem a manutenção do software no futuro, trazendo uma grande melhoria nos tempos de implantação, correção de bugs e na escalabilidade da aplicação.

[1] OpenShift: https://openshift.com
[2] Kubernetes: https://kubernetes.io

1.2 Objetivos

O nosso objetivo é demonstrar tecnologias de ponta que o mercado vem usando, além de fornecer um material que auxilie alunos e professores que tenham interesse no Node.js, incluindo exemplos de como uma empresa de grande porte tem usado a linguagem, a infraestrutura e qual a metodologia de desenvolvimento que tem sido adotada para o projeto em análise. Desta forma, queremos motivar o estudo de Node.js, estimulando com que nossos discentes e docentes busquem conhecer mais do mercado, possibilitando crescimento dos nossos alunos e do alcance do nosso curso.

Além disso, demonstraremos um pouco da rotina de desenvolvimento em uma empresa de grande porte, preocupada com a documentação do código, boas práticas, e metodologia ágil desenvolvida pela empresa, explicando um pouco sobre como é o dia a dia, como são feitas as implantações entre ambientes usando integração e implantação contínuos.

1.3 Organização do texto

O presente trabalho está estruturado em capítulos e, além desta introdução, abordaremos os seguintes temas em cada um dos capítulos:

- Capítulo II: neste capítulo trataremos sobre a infraestrutura e as tecnologias utilizadas no projeto pela equipe, descrevendo-as e explicando o porquê de cada uma delas ter sido utilizada, focando no funcionamento de cada uma delas;

- Capítulo III: neste capítulo abordaremos metodologias de desenvolvimento, com foco na metodologia ágil desenvolvida pela empresa, explicando um pouco sobre o ciclo de desenvolvimento em cascata, e sobre a metodologia *Lean Software Development*. Também abordaremos questões do projeto acerca de ambientes utilizados e suas características, descrevendo as necessidades de cada ambiente, explicando sobre as ferramentas de integração contínua e implantação contínua (CI/CD);

- Capítulo IV: neste capítulo concluímos o estudo, sintetizando o que foi dito nos capítulos anteriores, focando nas motivações das escolhas, em como escolher a melhor ferramenta para um novo projeto, qual metodologia adotar, qual infraestrutura faz mais sentido para o que se está planejando construir.

2 Tecnologias Utilizadas no Projeto

2.1 Node.js

A linguagem de programação JavaScript [3] surgiu em 1995 como uma linguagem de script para complementar o desenvolvimento para Web com a dinamização das páginas Web. Os criadores da linguagem definiram que JavaScript teria uma sintaxe semelhante ao Java, já que serviria para complementar esta linguagem. Inicialmente, a linguagem seria chamada de Mocha. Porém, foi renomeada para JavaScript quando lançada em dezembro de 1995. O nome causou uma certa confusão por parecer uma derivação de Java. A princípio a linguagem era utilizada apenas para plataformas Web. Entretanto, em 1996 servidores Web passaram a suportar a implementação do JavaScript em páginas ASP e .NET. Em 2009, tivemos um grande aumento do interesse por JavaScript no lado servidor com a introdução do Node.js.

O Node.js (Nodejs, 2020) surgiu em maio de 2009 com a intenção de ajudar os programadores a criarem aplicações de alta escalabilidade como um servidor web, capazes de gerenciar muitas conexões simultaneamente graças a motor de execução V8, também conhecido como *engine*, desenvolvido pela Google com as linguagens C, C++ e JavaScript. Além disso, o Node.js foi desenvolvido sob a licença MIT, deixando o código aberto e seu uso comercial livre.

A máquina de execução V8 possui alto desempenho, usando os motores do JavaScript e do WebAssembly (Overview - WebAssembly 1.0, 2020) escrito em C++. Sua criação se deu como forma de aumentar o desempenho de JavaScript em navegadores Web, a fim de acelerar o processamento e permitir o desenvolvimento de aplicações complexas que estavam entre os projetos da Google. No entanto, esta máquina de execução não é somente utilizada no Node.js, dado que foi originalmente desenvolvida para o navegador Chrome. Importante ressaltar que ela implementa ECMAScript, uma padronização do JavaScript (ECMAScript, 2020), e WebAssembly e roda apenas em Windows 7 e versões posteriores, MacOS 10.12 e versões posteriores e sistemas Linux que façam uso de uma arquitetura de 64 bits, processadores MIPS, ARM ou IA-32. Além

[3] JavaScript: https://www.javascript.com

disso, a V8 pode rodar separada ou embutida em qualquer aplicação desenvolvida em C++. Por usar também o motor do WebAssembly, ela é capaz de compilar o código JavaScript em código de máquina. Dessa forma, as instruções já estão prontas para o servidor executá-las, aumentando a capacidade de processamento do mesmo.

Importante ressaltarmos o fato da V8 não gerar *bytecode* intermediário, removendo assim a necessidade de usarmos interpretadores. Outros motores, como o Rhino, geram algum *bytecode*. No exemplo do Rhino (Rhino - JavaScript Engine, 2020) ele compila o código JavaScript para Java *bytecode*, convertendo os scripts para classes. Com isso, o Java é responsável por enviar as instruções para o computador, sendo menos eficiente que a V8.

2.2 Solução em nuvem e Docker

Esta seção descreve o ambiente onde foi colocada em produção a aplicação descrita neste projeto. Este ambiente faz uso de contêineres (Docker, 2020) e orquestração (Kubernetes, 2020), se baseando em um ambiente de nuvem (OpenShift, 2020) que suporta estas tecnologias e simplifica o desenvolvimento de aplicações em NodeJS.

2.2.1 Docker

Utilizamos o Docker para tratar da conteinerização da aplicação. O Docker vem obtendo crescente relevância por solucionar problemas de compatibilidade, migrações de servidores e ser o principal agente na criação de aplicações conteinerizadas. Uma aplicação conteinerizada (Rouse, 2019) é uma virtualização de um nível do sistema operacional utilizada para executar e distribuir aplicações sem precisar executar uma máquina virtual inteira para cada aplicação. Com isso, é possível gerenciar múltiplas aplicações ou serviços isolados em um único hospedeiro e acessar o mesmo Kernel do sistema operacional. Contêineres funcionam em sistemas físicos, instâncias na nuvem e também em máquinas virtuais.

O Dockerfile (Docker Documentation, 2020) é um documento de texto que contém todos os comandos que o usuário pode chamar na linha de comando para montar uma imagem Docker que consiste de camadas de leitura representando cada instrução a ser executada no contêiner inicializado. É o arquivo responsável por criar uma imagem quando o usuário usa o comando *docker build*. Dockerfiles também são responsáveis por gerenciar variáveis de ambientes, conforme o exemplo na Figura 1 (abaixo).

```
FROM busybox
ENV foo /bar
WORKDIR ${foo}     # WORDKIR /bar
ADD . $foo         # ADD .    /bar
COPY \$foo /quux   # COPY /bar /quux
```

Figura 1: Exemplo de Dockerfile

Uma imagem Docker é um arquivo responsável por gerenciar tudo que será instalado e executado no contêiner do Docker antes de subir um contêiner. As imagens podem se utilizar de variáveis de ambientes, copiar informações do hospedeiro para o contêiner e disparar comandos para executar o projeto de diversas formas, de acordo com variáveis de ambiente.

Para subir um contêiner é necessário gerar uma imagem dele, rodando o comando *'docker build',* que prepara a aplicação em camadas, que são utilizadas quando se implanta uma aplicação com poucas mudanças. Dessa forma, geram-se novas imagens a partir das já construídas, o que além de tornar o processo mais rápido, o torna também mais eficiente, pois cada uma dessas camadas fica salva no disco e, dessa forma, evita-se dados duplicados.

Além disso, existe um repositório de imagens (*Docker Registry*) padrão pelo Docker, responsável por guardar algumas das imagens mais relevantes e mais utilizadas como por exemplo uma imagem do Debian, partição *Linux-based* muito utilizada como base de outras imagens. É possível também salvar imagens próprias neste repositório para compartilhá-las com a comunidade.

O Docker (Docker Documentation, 2020) usa uma arquitetura cliente-servidor. Conforme descrito na Figura 2, o cliente Docker conversa com o *Docker daemon*, responsável por construir/puxar as imagens, executar e distribuir os contêineres. O cliente e o *daemon* se comunicam usando REST API, em cima de um host UNIX via soquetes ou uma interface de rede. Já o *Docker Registry* é um repositório de imagens para Docker.

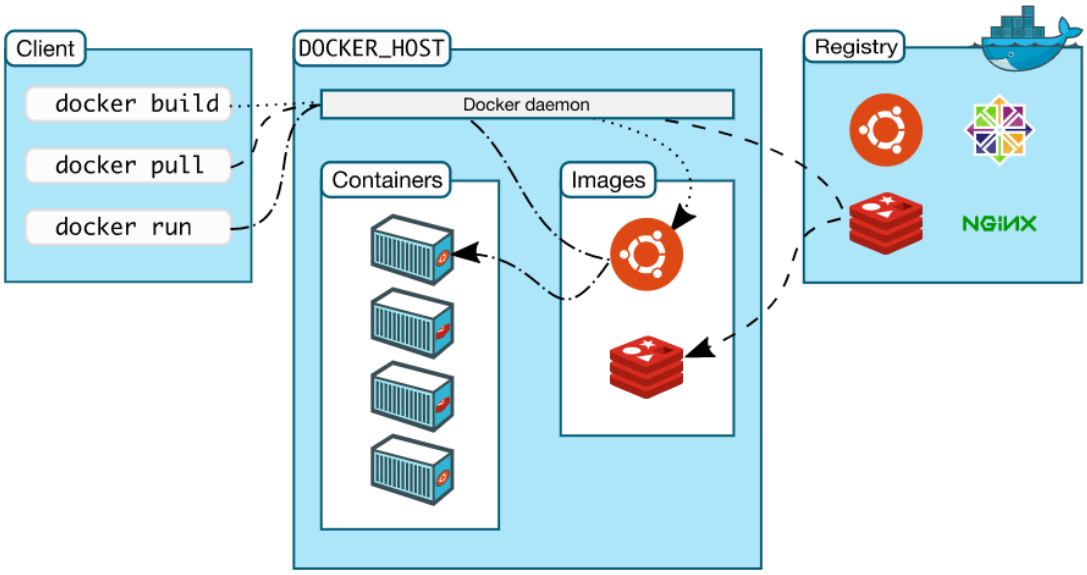

Figura 2: Arquitetura do Docker

2.2.2 Kubernetes

Desenvolvido pela Google em 2014, o Kubernetes é um sistema *Open Source*, programado na linguagem GO, para gerenciar contêineres (Kubernetes, 2020). Essa ferramenta surgiu com a necessidade de orquestrar, organizar, balancear, checar a saúde de cada *Pod*. Um *Pod* (Pod Overview, 2020) é responsável por encapsular um contêiner de aplicação e, em alguns casos, até múltiplos contêineres. Além disso, possui recursos de armazenamento, um IP único de rede e opções de como os contêineres devem rodar. Um *Pod* representa cada instância da aplicação que está no Kubernetes, podendo ser um único contêiner ou um grupo, que compartilham recursos.

Dentre as principais capacidades do Kubernetes, podemos citar o balanceamento de cargas, orquestração de armazenamento, auto-cura e gerenciamento e configuração de chaves. O balanceamento é responsável por monitorar o tráfico de requisições e encaminha a requisição para o *Pod* que estiver menos ocupado. A orquestração de armazenamento permite que o usuário monte um sistema de armazenamento de acordo com a preferência. A auto-cura é responsável por fazer o monitoramento dos *Pods*, reiniciar quando algum *Pod* falhar, conforme definido pelo usuário. O gerenciamento e configuração de chaves é responsável por guardar informações de segurança como senhas, OAuth token e chaves SSH. Dessa forma, todas as configurações de ambiente ficam salvas e não ficam expostas.

2.2.3 Solução em nuvem

Dentre as opções de solução em nuvem, temos o *OpenShift*, que é uma nuvem híbrida para gerenciamento de softwares baseados em contêineres (Pousty e Miller, 2014). Com isso, ele traz portabilidade de carga de trabalho, orquestração e gerenciamento entre dois ou mais ambientes. É uma distribuição suportada do Kubernetes, usando Docker e DevOps para o desenvolvimento acelerado de aplicações.

Lançado em 2011, o OpenShift começou a ganhar popularidade em 2017, quando sua versão estável foi lançada. Com essa crescente popularidade, a empresa que desenvolveu o projeto que aqui é apresentado criou programas de incentivo para que seus desenvolvedores usem as ferramentas da Red Hat, oferecendo treinamentos, visita de especialistas e licenças do OpenShift.

A ferramenta desenvolvida pela Red Hat, empresa conhecida por sua filosofia de desenvolver códigos e ferramentas *open source*, disponibiliza também o OpenShift em versão gratuita, com armazenamento e memória RAM limitados, porém com uso irrestrito da plataforma, o que cria os incentivos para que desenvolvedores o testem, conheçam e adotem a ferramenta. Na Figura 3 podemos perceber a abrangência do OpenShift. Além de fornecer bancos de dados conteinerizados e ferramentas para CI/CD, também fornece rápida configuração para um projeto inicial em algumas linguagens de programação, como Node, Python, PHP, .NET, GO, entre outras.

O OpenShift é uma ferramenta recente e grandes empresas estão investindo nesse tipo de solução. A Google, por exemplo, começou a oferecer esse tipo de solução com a criação do Google Anthos[4] e a Amazon com o Amazon EKS[5]. Essas soluções estão se tornando cada vez mais comuns em grandes empresas por oferecerem serviços na forma "pague conforme o uso" de memória, armazenamento e rede, além de oferecerem ferramentas que facilitam o desenvolvimento e o gerenciamento da infraestrutura.

[4] Google Anthos: https://cloud.google.com/anthos/
[5] Amazon EKS: https://aws.amazon.com/eks/

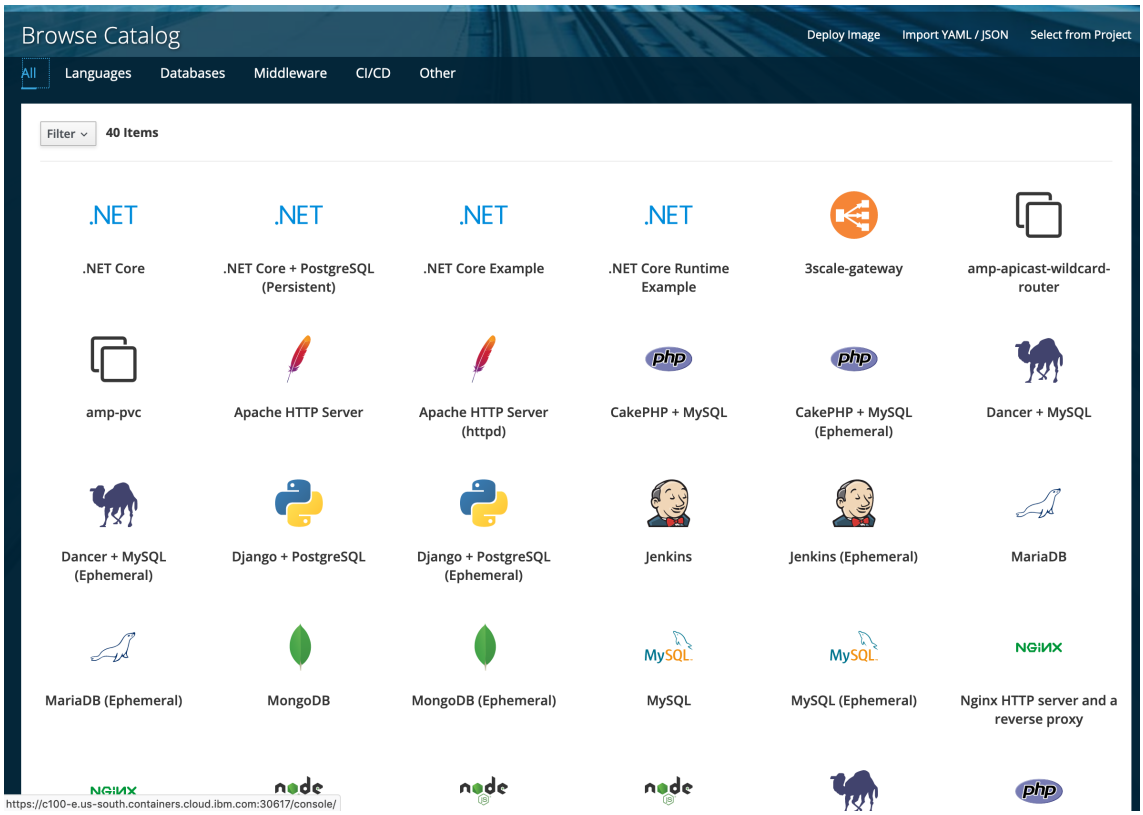

Figura 3: Catálogo do OpenShift.

A estrutura do OpenShift permite uma divisão eficiente entre o módulo de implantação, serviços e roteamento da aplicação. O módulo de implantação serve para, por meio de uma interface web, o usuário levantar um *Pod* (Kubernetes, 2020) com o novo código que deve ser subido. Para alterar as configurações de implantação, dos serviços ou mesmo dos roteamentos da aplicação, utiliza-se um documento YAML, que pode ser editado na própria página, trazendo praticidade e conforto para que o arquiteto faça alterações pequenas num curto espaço de tempo. YAML (YAML, 2020) é uma linguagem amigável de serialização de dados para todas as linguagens de programação.

No projeto em estudo, graças à mudança de uma estrutura monolítica para uma aplicação *dockerizada*, as implantações realizadas na empresa que desenvolveu o projeto passaram de um processo que demorava dois meses para uma semana, devido à necessidade de aprovações e a burocracia da empresa. Mas ainda assim é possível medir e mostrar evidências da melhoria, notando o quão prático e simples o processo de implantação ficou graças ao OpenShift.

2.3 Banco de dados

Quanto à escolha do banco de dados utilizado pela aplicação, a decisão não foi tão simples quanto os casos descritos nas seções anteriores. A equipe responsável pelo projeto tentou utilizar o MongoDB no desenvolvimento, mas concluiu que o projeto demandava um banco de dados relacional. O MongoDB (MongoDB, 2020) é um banco de dados orientados a documentos muito utilizado por ser mais simples de configurar, e não precisar modelar as tabelas como em um banco relacional. Na tentativa de uso do MongoDB a equipe teve que executar alguns scripts para alterar a estrutura dos dados e assim relacioná-los corretamente. Com isso, o processo se tornou muito pesado e demorado, gerando um alerta de que o banco de dados deveria migrar para um relacional.

Após a migração para um banco de dados relacional, o processo de tratar os dados no MongoDB, que demorava cerca de 30 minutos, passou a demorar 2 minutos, confirmando assim que a estrutura relacional era a melhor para ser utilizada. Após um longo período de conversas e discussões, a equipe optou pelo uso do DB2[6], que, por ser um banco relacional, melhorou o desempenho da aplicação. O ganho foi tão significativo que mesmo a maior parte da equipe não tendo conhecimento em DB2, preferiu-se usar uma tecnologia que poucas pessoas conheciam bem.

Depois de usar o DB2 por alguns meses, a equipe encontrou diversas dificuldades por falta de conhecimento, decidindo então alterar novamente a estrutura do banco para PostgreSql[7]. Com isso, vieram problemas de migração de banco de dados, que foram resolvidos com certa facilidade. Para decidir qual banco usar, o correto é avaliar qual a estrutura dos dados que serão armazenamos e com qual banco a equipe se sente mais confortável em trabalhar. Dessa forma, não haverá tanto retrabalho e se evitará frustrações de ter que alterar lógica no sistema devido há mudanças de banco.

É importante ressaltar, que apesar da aplicação ser estruturada em microsserviços (ver a próxima seção), a equipe não utiliza bancos de dados conteinerizados. Isso pode acarretar problemas quanto a backups, persistência e afins, havendo necessidade de se trabalhar muito mais no container e sua orquestração. Para simplificar esse processo, é melhor usar um banco de dados dentro de um servidor, além de utilizar pacotes/módulos de migração de dados. A equipe optou por utilizar o db-migrate, que possibilita uma

[6] DB2: https://www.ibm.com/support/knowledgecenter/SSEPGG_10.1.0/com.ibm.db2.luw.common.doc/doc/c0011568.html

[7] PostgreSQL: https://www.postgresql.org/docs/12/index.html

transição mais fácil entre alguns bancos de dados, além de servir como uma documentação de todas as mudanças nas estruturas dos dados.

O db-migrate (db-migrate) serve para criar as tabelas/views, usando um documento de configuração (Figura 4). Além de remover a dependência de linguagem do banco, pois o código é escrito em JavaScript, o *db-migrate* tem APIs nativas para uso de bancos SQL (Figura 5) e NoSQL (Figura 6), facilitando a criação de novos modelos de dados. O maior benefício do db-migrate foi o fato de mudanças constantes na estrutura dos dados e com ele a equipe de desenvolvimento conseguiu melhorar o código, documentando cada nova mudanças dentro de um script em JavaScript.

```
{
 "dev": {
   "driver": "sqlite3",
   "filename": "~/dev.db"
 },
 "test": {
   "driver": "sqlite3",
   "filename": ":memory:"
 },
 "prod": {
   "driver": "sql1",
   "user": "root",
   "password": "root"
 },
 "pg": {
   "driver": "pg",
   "user": "test",
   "password": "test",
   "host": "localhost",
   "database": "mydb",
   "schema": "my_schema"
 },
 "mongo": {
   "driver": "mongodb",
   "database": "my_db",
   "host": "localhost"
 },
 "other": "postgres://uname:pw@server.com/dbname"
}
```

Figura 4. Exemplo de configuração dos tipos de bancos

```
// with no table options
exports.up = function (db, callback) {
 db.createTable('pets', {
   id: { type: 'int', primaryKey: true, autoIncrement: true },
   name: 'string'  // shorthand notation
 }, callback);
}

// with table options
exports.up = function (db, callback) {
 db.createTable('pets', {
   columns: {
     id: { type: 'int', primaryKey: true, autoIncrement: true },
     name: 'string'  // shorthand notation
   },
   ifNotExists: true
 }, callback);
}
```

Figura 5. Exemplo SQL API

```
exports.up = function (db, callback) {
 db.createCollection('pets', callback);
}
```

Figura 6. Exemplo NoSQL API

2.4 Microsserviços

Os microsserviços são uma arquitetura e uma abordagem para escrever programas de software utilizando componentes separados que juntos realizam a mesma tarefa (Red Hat Microsserviços, 2020). Cada microsserviço é uma pequena parte da implementação da aplicação, responsável por parte das funcionalidades da aplicação, dessa forma há uma maior independência de cada parte da aplicação e maior facilidade para orquestrar cada parte.

O uso de uma solução em nuvem híbrida e com infraestrutura como um serviço facilita o uso de microsserviços pelo fato de nativamente utilizar o Docker e Kubernetes, ferramentas que estimulam o uso microsserviços, apesar de também suportarem aplicações monolíticas (Bagorolo, Di Cocco, Marinelli e Pichetti, 2017).

No projeto em questão, existem cinco microsserviços em operação, sendo eles: a interface com o usuário, login e autenticação, carga de dados, banco de dados

(postgreSQL) e consumo dos dados. O microsserviço de interface com o usuário é autoexplicativo: ele contém uma aplicação rodando o *framework* Angular 6[8], com algumas telas de gerenciamento dos dados, além de fazer muitas requisições para o microsserviço de carregamento dos dados. Além disso, se conecta com o microsserviço de login e autenticação, verificando se o usuário conectado está apto a usar o sistema.

Quanto ao microsserviço de login e autenticação, o mais interessante de ressaltar é a utilização de *SSO* (*Single Sign-On*), que funciona com um login único entre um conjunto de sistemas. Uma vez que o usuário faça login em algum sistema dentro do conjunto no qual o SSO está ativado, o usuário não precisará fazer login novamente. Caso o usuário esteja autenticado e tenha acesso ao sistema, ele será redirecionado para o microsserviço de interface com o usuário.

Os microsserviços de carga e consumo dos dados são bem simples. O microsserviço de carga de dados é responsável por buscar dados em alguns sistemas, fazer um tratamento destes dados e gerar um resultado com a junção dos dados. O microsserviço de consumo de dados é uma API que, dado uma consulta com os parâmetros corretos, retorna um conjunto de dados referentes a essa consulta.

O microsserviço de banco de dados nada mais é do que o serviço responsável por gerenciar o banco de dados. Sua implementação utiliza o *db-migrate* para fazer alterações, quando necessárias, além de utilizar para desenvolvimento local como um banco de dados *dockerizado*.

2.5 Considerações finais

Este capítulo apresentou as tecnologias utilizadas no projeto utilizado como exemplo de uso das tecnologias em foco, falando sobre suas funcionalidades e sua relevância para o caso em estudo e para o mercado.

Apresentamos o funcionamento do Node.js e sua máquina de execução. Além disso, as tecnologias Docker, Kubernetes e OpenShift possibilitam novos métodos para o desenvolvimento de aplicações, mantendo-se atualizado com novas práticas de desenvolvimento como CI/CD, aplicações *dockerizadas* e uso de tecnologias em nuvem. Essas tecnologias e práticas são, conforme demonstra nosso caso de uso, importantes e utilizadas pelo mercado.

[8] Angular: https://angular.io/

O próximo capítulo vai tratar da equipe e das metodologias de desenvolvimento utilizadas no projeto. Além disso falaremos sobre os ambientes utilizados no caso de uso analisado, expondo a relevância de separar os ambientes de produção, testes de usuário e desenvolvimento.

3 Metodologia e Ambientes

3.1 Metodologias de desenvolvimento de software

Existem diversas metodologias para o desenvolvimento de software das quais analisaremos a metodologia ágil da empresa, Lean Software Development e Cascata. Cada uma destas metodologias tem sua aplicação e usabilidade, ou seja, dependendo do projeto faz sentido usar uma ou outra. Abordaremos brevemente a Lean Software Development e a Cascata, a fim de compararmos com a metodologia da empresa. Porém, focaremos na metodologia da empresa, já que o projeto fez uso desta metodologia.

3.1.1 O modelo em cascata

A metodologia Cascata (Balaji, 2012) também conhecida como *Waterfall* (cascata em inglês), é um modelo sequencial, geralmente usado para processos de desenvolvimento de software, nos quais o progresso é visto conforme os processos vão sendo concluídos. É importante ressaltar que nessa metodologia, após uma etapa terminar, ela não é revisitada. Sendo assim, após o fim de cada etapa, para retornar a ela, é necessário refazer todo o processo, começando pela etapa de requisitos, conforme indicado na Figura 7.

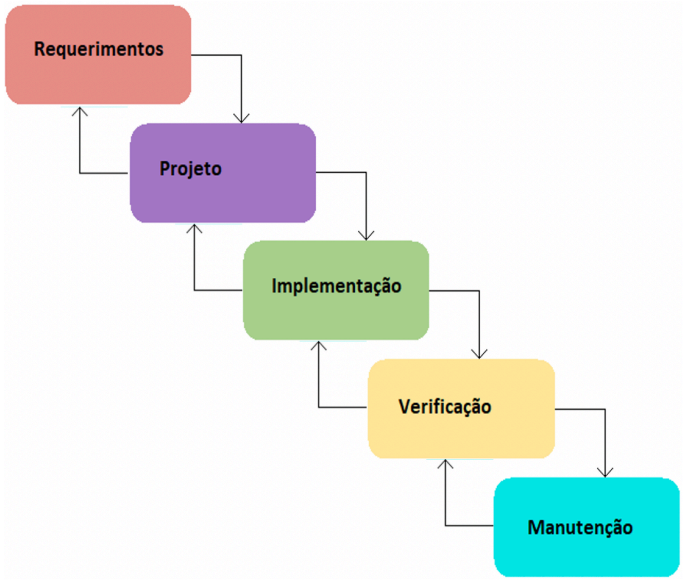

Figura 7. Etapas do modelo em cascata

3.1.2 Metodologia Lean

O modelo *Lean Software Development* (Poppendieck, 2007) é a aplicação dos princípios do Sistema de Desenvolvimento de Produtos da Toyota em desenvolvimento de software. Os sete princípios da metodologia Lean são: eliminar desperdícios, construir com qualidade, criar conhecimento, adiar o compromisso, entregar rapidamente, respeitar as pessoas e otimizar o todo. Com base nesses pilares, podemos analisar que o foco é na eficiência do trabalho, além da qualidade do produto e em um ambiente bom para os envolvidos no projeto.

- O princípio de eliminar desperdícios é entender tudo que não agrega valor ao cliente ou que atrasa o recebimento do valor como algo que pode ser simplificado;
- Construir com qualidade envolve utilizar metodologias que aumentem a qualidade do código, fazendo testes unitários e corrigindo problemas logo que encontrados;
- Criar conhecimento implica em fazer um *design* completo da aplicação, uma vez que um *design* incompleto pode não antecipar a complexidade da implementação;
- Quanto a adiar compromisso, o princípio não está relacionado com adiar entregas e afins, mas sim adiar decisões irreversíveis para o último momento possível para que, dessa forma não sejam necessárias grandes reestruturações;
- Entregar rapidamente implica que os desperdícios de tempo já foram eliminados, que o software desenvolvido tem poucos defeitos e que a equipe entende bem as prioridades do cliente, entregando assim o valor o mais rápido possível;
- Respeitar as pessoas é ter uma empresa que permita que seus funcionários tenham engajamento para que possam criar ótimos produtos, desta forma, fomentando a especialização de seus funcionários;
- Otimizar o todo é otimizar a cadeia de geração de valor, desta forma, aumentando o valor que se entregará para o cliente. Para atingir isso, basta otimizar cada processo da cadeia de geração de valor.

Na Figura 8, podemos observar a cadeia de geração de valor e como a otimização de cada etapa do ciclo otimizaria o próximo passo. Podemos perceber a mudança evidente do modelo em cascata para o *Lean Software Development*. Enquanto no modelo em cascata o foco estava nos processos e nos artefatos produzidos durante a execução destes

processos, no modelo *Lean* o foco é nas pessoas, no valor entregue ao cliente e na redução de custos, tentando sempre manter a eficácia no trabalho.

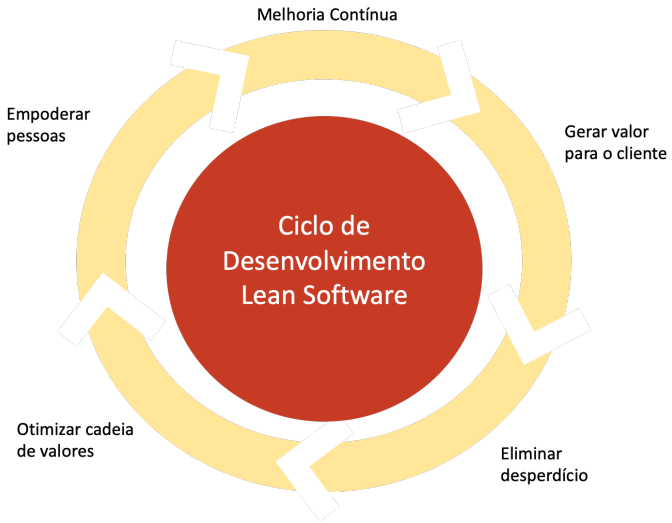

Figura 8. Cadeia de geração de valor da metodologia Lean

3.1.3 Metodologia ágil da Empresa

A metodologia da empresa surge com as mudanças rápidas de tecnologia, onde os requisitos não precisam ser completamente definidos e deve-se entregar o mínimo de valor, o mais rápido possível. Na metodologia em questão, assim como no Lean Software Development, o foco é também na entrega de valor da forma mais eficaz e eficiente possível. Para realizar isso (Agile Manifesto, 2001) a metodologia se baseia em doze princípios, além de quatro diretrizes centrais. As quatro diretrizes são: Indivíduos e interações acima de processos e ferramentas, um software que funcione ao invés de uma documentação compreensível, colaboração do cliente acima de negociação do contrato, responder a mudanças ao invés de seguir um plano. Podemos perceber que o foco está nos indivíduos, no cliente e na entrega de valor, a fim de atender as mudanças que possam acontecer. Citaremos os doze princípios ágeis que estão descritos no Manifesto for Agile Software Development:

- Satisfação do cliente através de entregas contínuas de valor;
- Receber mudanças de requisitos ainda que no fim do desenvolvimento;
- Entregar um software funcional constantemente, com a preferência de uma curta escala de tempo;

- As pessoas da área de negócios e os desenvolvedores devem trabalhar diariamente juntos durante o projeto;
- Construir projetos ao redor de indivíduos motivados, dando o ambiente e o suporte que eles precisam para realizar o trabalho;
- A comunicação mais eficiente e eficaz para transmitir informação entre e para a equipe de desenvolvimento é a conversa face a face;
- Um software funcional é a primeira medida de progresso;
- Agile promove desenvolvimento sustentável: desenvolvedores, patrocinadores e usuários devem ser capazes de manter um ritmo constante indefinidamente;
- Atenção contínua à excelência técnica e bom design melhoram a agilidade;
- Simplificar é essencial, reduzir o trabalho desnecessário;
- As melhores arquiteturas, requisitos e *designs* são produzidos por equipes auto organizáveis;
- Em períodos regulares, a equipe deve refletir em como se tornar mais eficaz e fazer os ajustes para efetivamente serem mais eficazes.

Cada um desses princípios é responsável por criar um alicerce em como a equipe deve se estruturar e se comunicar com o cliente. O principal foco da metodologia ágil da empresa são nas mudanças, pois são elas que geram valor e competitividade para o cliente. É poder modificar um requisito que estava definido e pronto para ser entregue em um novo requisito que, do ponto de vista do cliente, agregará mais valor ao seu negócio, pois, assim como na *Lean*, é necessário focar no que traz valor para o cliente.

3.2 Aplicação da metodologia no projeto

O projeto em estudo foi baseado na metodologia ágil da empresa, fazendo uso de reuniões diárias de quinze minutos com os integrantes da equipe técnica e entregas parciais a cada *sprint* (intervalo de tempo entre as entregas parciais), definidas pela equipe para durar duas semanas.

Na Figura 9 podemos observar como é um ciclo ágil. Reuniões de troca de experiências a respeito de como foi a *sprint*, o que poderia melhorar, quais práticas foram positivas e quais foram negativas. Planejamento da *sprint*, montar *backlog* (definição de

requisitos) do produto, *backlog* da *sprint*, *Planning Poker*, prática para avaliar o peso das tarefas que a equipe irá realizar durante a *sprint*. Isso tudo auxilia a equipe a entender onde pode haver falhas de entendimento de requisitos e, pelo fato das *sprints* durarem duas semanas, quaisquer mudanças necessárias serão aplicadas em curto espaço de tempo, tornando os requisitos adaptáveis à necessidade do cliente.

Figura 9. Ciclo de uma *sprint* ágil

Durante o projeto, havia muitos requisitos mal definidos, inconclusivos, trazendo insegurança para a equipe desenvolver requisitos mal formulados. A equipe passou por um momento conturbado por não estar bem definido quem era o dono do produto, o que acarretou em uma série de requisitos que foram entregues de forma incorreta, necessitando de uma revisão próxima ao período de implantar o projeto no ambiente de produção. Graças a este acontecimento, a equipe conseguiu incorporar uma pessoa que entendia do negócio e de suas regras. Este novo recurso tornou-se uma ponte entre a equipe e o cliente, sendo ela uma analista de negócios do cliente e tendo bastante experiência no produto que estava sendo desenvolvido.

Ter o cliente mais próximo da equipe de desenvolvimento foi crucial para um andamento melhor das *sprints*, pois sempre que surgia alguma dúvida quanto aos requisitos que a equipe desenvolveria durante a *sprint*, o contato com a equipe de negócios era eficiente, possibilitando à equipe de desenvolvimento resolver mais rápido e resolver as questões pendentes. Além disso, conhecendo melhor o cliente e suas demandas, a equipe foi capaz de minimizar erros e trazer maior velocidade ao processo de desenvolvimento, graças ao melhor entendimento dos requisitos.

Dois conceitos muito importantes na metodologia em estudo são a definição de pronto e definição de feito. No começo do projeto, a equipe tinha dificuldades em entender esses dois conceitos, ainda mais que a definição de feito é dada pelo dono do produto. Foi muito comum, por muito tempo, ouvir a frase "Está pronto, mas falta...", demonstrando um despreparo da equipe em entender o que é pronto e feito.

Devido a esses problemas, e depois de diversas reuniões, foi definido que pronto (*ready*) é quando o programador sobe um *pull-request*, explicando a funcionalidade que foi desenvolvida, porque foi feita e como testar para garantir que tudo está operacional, além de um processo rigoroso quanto aos testes unitários, permitindo apenas códigos com cobertura de comandos pelos testes acima de 85%.

Em relação ao conceito de feito, o foco é no valor para o cliente, ao invés da funcionalidade no ambiente de desenvolvimento. Então, uma tarefa só é considerada feita (*done*) quando o usuário final faz os testes que julgue necessários (geralmente, definidos junto com a tarefa) e, com isso, os desenvolvedores têm mais informações sobre o comportamento esperado da funcionalidade.

Quanto às tarefas que a equipe tinha anotado por conta das reuniões com a equipe de negócios, a equipe começou registrando-as em *post it* e utilizando um quadro no estilo *Kanban* (Vacanti, 2018), sendo uma estratégia para melhorar o fluxo de valor para os clientes, que serve para visualizar o fluxo de trabalho em três principais partes: a fazer, fazendo e feito. A equipe subdividiu a seção de feito em duas seções: pronto e feitos, seguindo a lógica descrita acima.

Entretanto, com a chegada de muitos membros, o quadro acabou caindo em desuso e por vezes ficava desatualizado. Com isso, a equipe decidiu utilizar o *Git Projects*, que é uma funcionalidade embutida do GitHub[9] e funciona como um Kanban interligado ao GitHub, no qual as equipes de desenvolvimento e negócios cadastram as tarefas que a ser feitas ou bugs encontrados na aba de *issues* do GitHub. Apesar de por vezes estar desatualizado, com o uso dos *pull-requests* ficou mais fácil fazer uma auditoria de quem fez qual parte do código e por qual necessidade aberta pelo cliente.

Essa estrutura serviu para apoiar a equipe de desenvolvimento, uma vez que fazendo uso dessas ferramentas e processos, fazer *rollbacks* (voltar um código defeituoso) seria mais prático e rápido. Toda essa estrutura e processos ajudaram a equipe a desenvolver um fluxo bem definido, que visou à redução de erros e falhas. O uso da

[9] GitHub: https://github.com/

ferramenta também trouxe maior confiabilidade ao desenvolvimento, pois no sistema antigo não havia ferramenta para versionamento de código e, caso algum código defeituoso fosse para produção, o processo de checagem era manual, assim como era manual o processo de restaurar os backups de código, dependente de uma equipe que ficava alocada em São Paulo.

Tudo isso o novo projeto se propunha a mudar, e conforme citado anteriormente, a estrutura de *pull-requests* com dois aprovadores foi um dos fatores de sucesso do trabalho. O processo de *pull-request* é uma forma dos programadores inserir código na principal *branch* (ramificação) do código, seguindo um padrão definido pela equipe. O desenvolvedor tinha que referenciar o requisito que o código resolve, descrever a solução proposta, quanto tempo demorou para desenvolver, o quanto de cobertura seus testes unitários oferecem e um roteiro de teste para a funcionalidade.

Além disso, para aprovar uma *pull-request* eram necessários dois aprovadores e a equipe fazia um rodízio acerca de quem seriam os aprovadores da rodada. Com isso, além de termos melhor segurança do código que estava subindo para produção, toda a equipe ficava ciente de como cada um estava codificando, aprendendo mais sobre como o sistema funciona e como cada parte foi codificada.

A equipe decidiu separar o projeto em quatro ambientes e explicaremos como cada um deles funciona e os riscos que a equipe pretendia mitigar com a criação destes ambientes.

3.3 Ambiente de desenvolvimento

Antes de falar sobre o ambiente de desenvolvimento, é importante ressaltar que a equipe utiliza o GitHub para manter as versões do código-fonte. Conforme falado anteriormente, para desenvolver um novo código ou melhoria, o desenvolvedor deveria pegar o código mais recente no GitHub, que fica dentro da principal ramificação do código (*master*) e, a partir desta, criar um novo ramo. A partir deste novo ramo, o desenvolvedor faz o seu trabalho de código e abre uma *pull-request* para juntar o novo código ao ramo principal, onde outros dois desenvolvedores eram responsáveis por testar, verificar a qualidade e aprovar a junção do código.

A equipe decidiu usar o Jenkins como ferramenta para configurar o CI/CD, onde cada junção de código com o ramo principal iniciava um processo que atualizava o servidor de desenvolvimento com este novo código, ao qual tanto os desenvolvedores

quanto os analistas tinham acesso. O Jenkins (Jenkins, 2020) é uma ferramenta de automação de servidores que pode ser utilizada para automatizar tarefas relacionadas ao desenvolvimento, testes, disponibilização ou implantação de softwares. Ele pode ser instalado a partir do *Docker* ou rodar de forma independente. No projeto, a equipe utilizou o *Docker* com o *OpenShift*, por proporcionar facilidades para a implementação de CI/CD.

Um dos problemas que a equipe enfrentou foi o uso de um GitHub privado, onde para acessar os dados é necessário registrar uma chave SSH e uma série de protocolos. Mas depois de persistirem, conseguiram pegar uma chave SSH para o projeto, dando a possibilidade para o Jenkins ficar observando o ramo *master* e rodar os scripts de atualização de código sempre que houvesse alguma atualização neste ramo.

A partir disso, o Jenkins foi configurado para escutar o ramo *master* e rodar scripts de testes unitários com tolerância de 90% de cobertura (maior do que a necessária para aprovar uma *pull-request*) e garantindo que todas as funções presentes no código que está sendo migrado estão funcionais segundo os testes. Isso trouxe grande praticidade e rapidez para a equipe realizar uma mudança de código entre ambientes, e com o uso do *OpenShift* isso foi potencializado, uma vez que este último guarda todos os históricos de *pods* que ele subiu, tornando qualquer *rollback* trivial.

O Jenkins se conectava com o ramo *master* e ao servidor de pré-produção. Neste servidor, os analistas faziam testes de integração e verificavam se as funcionalidades estavam entregando os resultados esperados. Com isso, a equipe não se responsabilizava por cobrar a equipe de negócios a testar novas funcionalidades, exceto se estes recursos fossem cruciais para resolver um erro. Fora isso, as novas funcionalidades ficavam à disposição do cliente para testar e aprovar o processo e seguir em frente, conforme explicaremos mais adiante.

Importante ressaltar que qualquer problema encontrado nesse ambiente não gera impacto nos outros ambientes e é reportado à equipe de desenvolvimento, que trabalha nesses problemas como prioridade máxima, já que um problema encontrado nesta etapa afeta a mudança do código ir para o ambiente de produção ou não.

3.4 Ambiente de pré-produção

Este é o ambiente onde o código, depois de aprovado pelos analistas, pode ser testado pelos usuários, que fazem testes de funcionalidade, de comportamento e se as soluções estão satisfatórias ao que foi pedido, analisando os requisitos levantados e seus

critérios de aceitação. Quaisquer problemas encontrados nesse ambiente são analisados pelos analistas para entender se é um problema do código ou se foi um problema de dados mal formatados. Sendo um problema de dados mal formatados ou incongruentes, os usuários devem consertar os dados e testar a funcionalidade novamente.

Infelizmente, esse ambiente no novo projeto é um gargalo porque os usuários não estavam acostumados com o novo modelo de gerar os resultados do sistema (planilhas com as taxas de financiamento) e com isso a incongruências acabam demandando muito tempo da equipe de analistas para entender o erro e pedir as alterações. Apesar disso, a equipe de desenvolvimento sofreu menor impacto com os testes feitos no ambiente de pré-produção, dado que grande parte dos erros são relacionados aos dados e acabam voltando para os analistas e usuários.

Separar o ambiente neste ponto foi crucial para dar maior segurança aos desenvolvedores, isolando-os do problema caso não seja um problema de código e aliviando a carga deles em relação aos problemas analíticos. Este é um dos ambientes chave para que possam levar o código para produção, pois somente após a aprovação dos usuários, garantindo que as funcionalidades foram entregues conforme as suas necessidades, a equipe poderia alterar o código de produção.

3.5 Ambiente de produção

Este é o principal ambiente da aplicação, onde uma vez que um novo código tenha passado por todas as etapas de validação, ele é aplicado. Neste ambiente, devido à política da empresa, os desenvolvedores não possuem acesso nem ao servidor, nem ao código-fonte que este está executando. Desta forma, quaisquer erros encontrados no servidor de produção são prioridade máxima e são de responsabilidade dos usuários e da equipe de negócios, já que estes foram os últimos aprovadores.

Apesar de não ser um problema da equipe de desenvolvimento, todas as equipes se responsabilizam em consertar o problema devido à urgência e por ser o principal servidor da nossa aplicação, que os usuários acessam constantemente. Dado aos processos e ambientes separados, esses tipos de erro tendem a não acontecer com frequência, uma vez que diversas equipes com diferentes áreas de atuação fazem as verificações durante o processo de desenvolvimento. Isto traz grande segurança ao projeto e a cada equipe por ter realizado os testes e garantido que os ambientes estão funcionando conforme esperado.

3.6 Considerações finais

Neste capítulo, nosso objetivo foi falar sobre a rotina de desenvolvimento usando a metodologia ágeis, mostrar os ambientes nos quais o projeto em tela foi executado e mostrar o funcionamento de ferramentas de CI/CD. Com isso, mostramos alguns dos problemas que podem acontecer no projeto e quem é o responsável por cada problema.

No próximo capítulo abordaremos as considerações finais, explicando como cada etapa é importante para a realização de um projeto e como as escolhas auxiliaram a criar o projeto aqui citado. Além disso, falaremos de trabalhos futuros que possam ser feitos e como esta monografia pode auxiliar a seguir as próximas etapas.

4 Conclusão

Ao longo do texto, descrevemos o projeto em estudo, explicando as tecnologias que foram utilizadas e comparando-as com outras tecnologias na indústria de desenvolvimento de software. Isso tudo foi realizado para explicitar a relevância do Node.JS e das tecnologias de desenvolvimento de soluções em nuvem. Com esse panorama, conseguimos discorrer sobre a estrutura de uma aplicação em microsserviços e porque hoje em dia essa é uma tecnologia relevante. Explicitamos conceitos sobre metodologias de desenvolvimento, focalizando nas metodologias ágeis, utilizando e aplicando de conhecimentos ministrados na disciplina de Processos de Software, lecionada na UNIRIO.

4.1 Objetivos e limitações

Analisando o projeto, podemos elaborar a respeito do que deu certo e errado durante o projeto e quais decisões devem ser tomadas com cautela e objetividade. Quanto ao que deu certo, podemos garantir que o uso de metodologias ágeis foi um fator que trouxe muitos benefícios assim como a mudança para um banco relacional. Quanto ao que deu errado, o uso de tecnologias mais modernas sem uma avaliação dos tipos de dados pode acabar causando retrabalho, e problemas de performance. É de fundamental importância avaliar o modelo que vai ser utilizado para usar a tecnologia adequada.

Consideramos que esse projeto alcançou o objetivo de elucidar mais as novas tecnologias que estão sendo lançadas na indústria, fornecendo um material que explique brevemente o funcionamento destas tecnologias. Esperamos que este documento sirva de base para que os alunos do curso de BSI da UNIRIO adquiram novos conhecimentos, visando à capacitação individual.

Devido à falta de tempo e ferramentas, as comparações de tecnologias se mantiveram no campo teórico. No caso da comparação do Node.js, seria interessante fazer uma análise com outras linguagens em termos de desempenho, executando os mesmos trechos de código e comparando a velocidade de execução de cada uma das linguagens.

4.2 Trabalhos futuros

Existem diferentes formas de aprimorar este trabalho, seja fazendo uma melhor avaliação entre linguagens, seja trazendo outros casos de usos ou mesmo tratando de outras tecnologias relevantes no mercado, como Python[10], Swift[11], TypeScript[12] e GO[13].

Além disso, não conseguimos cobrir outras tecnologias e ferramentas relevantes a orquestração de contêineres, como o Istio (Istio, 2020), responsável por controlar o fluxo de conexões entre APIs, segurança dos serviços e monitorar os serviços e *logs*, sendo também uma possibilidade de trabalhos futuros o uso dessas tecnologias.

Também não cobrimos outros fornecedores de soluções em nuvem além do *OpenShift*. Porém, existem outros provedores, como a Amazon Web Services, o Google Anthos, entre outras. É possível fazer um estudo sobre estes provedores de solução em nuvem e os catálogos de serviços fornecidos por cada um deles.

O nosso foco neste Trabalho de Conclusão de Curso foi motivar que os alunos busquem conhecimento do que está acontecendo na indústria de desenvolvimento de softwares e, com isso, consigam acompanhar as mudanças tecnológicas. Esperamos ter atingido sucesso neste sentido e poder ajudar os alunos do curso de Bacharelado em Sistemas de Informação da Unirio.

[10] Python: https://www.python.org/
[11] Swift: https://developer.apple.com/swift/
[12] TypeScript: https://www.typescriptlang.org/
[13] GO: https://golang.org/

Referências Bibliográficas

Bagorolo, Di Cocco, Marinelli e Pichetti (2019) “Dynamic Deployment of An Application Based On Micro - Services”. Disponível em: <https://patentimages.storage.googleapis.com/e0/ec/96/3f4bbfe5d4e528/US10255052.pdf>. Acesso em 04/01/2020.

Goyal, S. (2014) “Public vs Private vs Hybrid vs Community - Cloud Computing: A Critical Review”. Disponível em: <http://www.mecs-press.net/ijcnis/ijcnis-v6-n3/IJCNIS-V6-N3-3.pdf>. Acesso em: 18/11/2019.

M. Poppendieck, "Lean Software Development," 29th International Conference on Software Engineering (ICSE'07 Companion), Minneapolis, MN, 2007, pp. 165-166, doi: 10.1109/ICSECOMPANION.2007.46.

Manifesto for Agile Software Development. Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R., Mellor, S., Schwaber, K., Sutherland, J., Thomas, D. (2001). Disponível em: <https://agilemanifesto.org/>. Acessado em: 31/01/2020

Pousty, S. and Miller, K. (2014) “Getting Started with OpenShift: A Guide for Impatient Beginners”. O’Reilly

Vacanti, Daniel. (2018) "O Guia do Kanban para Times Scrum", Scrum.org.

WATERFALLVs V-MODEL Vs AGILE: A COMPARATIVE STUDY ON SDLC. Balaji, S., Sundararajan Murugaiyan, M. (2012) "International Journal of Information Technology and Business Management", 29 June, Vol.2 No.1. JITBM & ARF.

About | Node.js. **Nodejs**. Disponível em: <https://nodejs.org/en/about/>. Acesso em: 12/11/2019.

Anthos. **Anthos**. Disponível em: <https://cloud.google.com/anthos/?hl=pt-BR>. Acesso em: 04/05/2020.

Db-migrate. **db-migrate**. Disponível em: <https://www.npmjs.com/package/db-migrate>. Acesso em 07/09/2020

Docker Documentation. **Docker**. Disponível em: <https://docs.docker.com/engine/reference/builder/>. Acesso em 31/01/2020.

Documentation - V8. **V8**. Disponível em: <https://v8.dev/docs>. Acesso em: 20/01/2020.

ECMAScript. **ECMAScript**. Disponível em <https://wikipedia.org/wiki/ECMAScript>. Acesso em: 03/09/2020.

Home - db-migrate. **db-migrate**. Disponível em: <https://db-migrate.readthedocs.io/en/latest/>. Acesso em: 04/01/2020.

Istio. **Istio**. Disponível em: <https://istio.io/>. Acesso em 05/03/2020

JAVA vs Node JS. **EDUCBA**. Disponível em: <https://www.educba.com/java-vs-node-js/>. Acesso em: 12/11/2019.

Jenkins User Documentation. **Jenkins**. Disponível em: <https://jenkins.io/doc/>. Acesso em 18/03/2020.

MongoDB. **MongoDB**. Disponível em: <https://www.mongodb.com/>. Acesso em: 03/09/2020.

Node.js. **Wikipedia**. Disponível em: <https://en.wikipedia.org/wiki/Node.js>. Acesso em: 12/11/2019.

O que é uma arquitetura de microsserviços. **Red Hat Microsserviços**. Disponível em: < https://www.redhat.com/pt-br/topics/microservices>. Acesso em 07/09/2020

Overview - WebAssembly 1.0 **WebAssembly**. Disponível em: <https://webassembly.github.io/spec/core/intro/overview.html#concepts>. Acesso em: 20/01/2020.

PHP vs Node JS, **Geeks for Geeks**. Disponível em: <https://www.geeksforgeeks.org/php-vs-node-js/>. Acesso em: 12/11/2019.

PostgreSQL. **PostgreSQL**. Disponível em: <https://www.postgresql.org/>. Acesso em: 03/09/2020.

Rhino Documentation. **Mozilla**. Disponível em: <https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino/Documentation>. Acesso em: 12/11/2019.

Rhino (JavaScript Engine). **Wikipédia**. Disponível em: <https://en.wikipedia.org/wiki/Rhino_(JavaScript_engine)>. Acesso em: 20/01/2020

Rouse, Rosencrance, McKenzie, Courtemanche, Kinard (2019). What is Application containerization. Disponível em: <https://searchitoperations.techtarget.com/definition/application-containerization-app-containerization>. Acesso em 11/02/2020

The Official YAML Web Site. **YAML**. Disponível em: <https://yaml.org/>. Acesso em 12/02/2020.

What is a Container? | App Containerization | Docker. **Docker**. Disponível em: <https://www.docker.com/resources/what-container>. Acesso em: 31/01/2020.

What is Kubernetes. **Kubernetes**. Disponível em: <https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/>. Acesso em 12/02/2020