

Greenhouse Delivery On Minikube

João Adherval C. de Barros¹, Milena S. C. Carneiro¹

¹Centro de Informática – Universidade Federal de Pernambuco (UFPE)
50.740-560 – Recife – PE – Brasil

{jacob,mscc}@cin.ufpe.br

Resumo. *Este trabalho teve como objetivo principal fazer as configurações necessárias para a aplicação Greenhouse delivery ser executada, orquestrada e escalada através do Minikube. Implementar vantagens e boas práticas de DevOps, como facilidade de deploy e build e integração contínua, além de acrescentar o Elasticsearch aos serviços foram objetivos não alcançados plenamente. Enxerga-se a possibilidade de acrescentar o Elasticsearch e até mesmo a stack ELK para agregar mais boas práticas de arquitetura de microsserviços como trabalhos futuros.*

Palavras-chave: *minikube, container, configuração, ambiente, microsserviços*

Repositório: Greenhouse delivery on Minikube

1. Introdução

Uma das maiores preocupações no mercado tecnológico, assim como no mercado em geral, é a rapidez e a eficiência na entrega de um produto que não só agregue valor, mas também satisfaça o cliente. Além disso, vivemos em tempos de imersão tecnológica, smartphones, tablets, notebooks, ultrabooks, exposição a dados em todos os momentos, big data. Logo, mais importante do que um produto, a atualização do mesmo de forma que o cliente tenha acesso a elas também de forma rápida e eficaz. É dentro desse contexto que existe a valorização da arquitetura em microsserviços, assim como um pipeline de desenvolvimento mais integrado e uma entrega contínua do produto.

1.1. Caracterização do problema

O Greenhouse Delivery é um projeto em Java Spring Boot que tem como objetivo colocar em prática os conceitos de arquitetura de microsserviços, com troca de mensagens, separação em pequenos módulos e uma abertura para facilidade em manutenção e novas implementações independentes, onde um serviço não depende de outro. Porém, está limitado a isso.

Dado que atualmente o sistema como um todo funciona somente com todos os microsserviços, broker de mensagens e banco de dados localmente (em uma única máquina), como modificar a arquitetura para que seja possível isolar, orquestrar e escalar os serviços baseados na nuvem?

2. Fundamentação teórica

O projeto em sua primeira versão, conta com tecnologias como *Spring Boot*, *RabbitMQ* e *Eureka*. Optamos por não modificar nenhuma dessas implementações, a menos que necessário, pois elas oferecem muitos benefícios para a arquitetura em microsserviços.

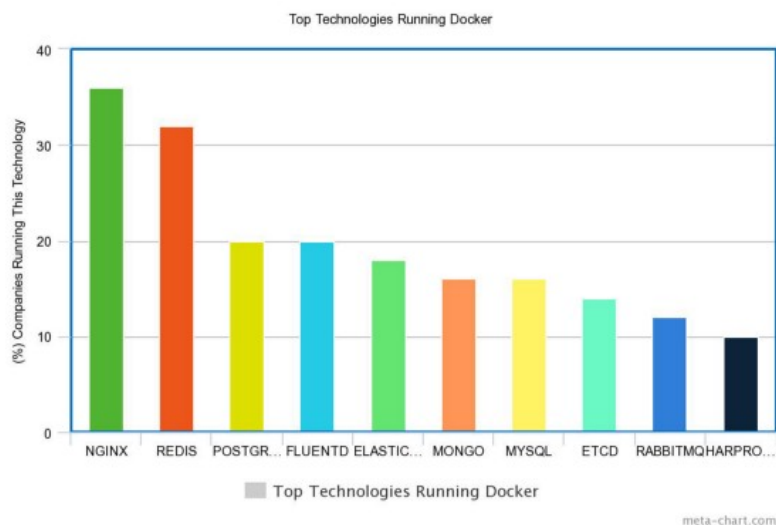
Spring Boot é baseado no *Spring Framework*, que por si só já tem um histórico de desenvolvimento na casa de 15 anos. Dessa maneira, a aplicação é construída em cima de uma tecnologia madura, flexível. Além disso é *open source* mas com apoio de grandes empresas, uma tecnologia já baseada em *Cloud*, um grande benefício já que a intenção é justamente fazer a transição da aplicação de local para nuvem. [Hackernoon 2018]

Outra grande vantagem é a compatibilidade [Spring 2019] com uma grande variedade de bancos, tais como *Oracle*, *PostgreSQL*, *MySQL*, *MongoDB*, *Redis*, fora a opção de conectividade com outras tecnologias tais como *RabbitMQ*, *ElasticSearch*, *Eureka*. [HackerNoon 2019]

O *RabbitMQ* é um sistema de mensageria que permite que os serviços se comuniquem através de mensagens, de forma a desacoplá-los mais uns dos outros. A conectividade e integração do *RabbitMQ* com o *Spring Boot* e *Docker*, além do seu amplo uso no mercado não nos deu motivo para uma mudança nesse sentido.

Para fazer a transição para nuvem, foram escolhidos o *Docker* e o *Minikube* (uma versão generica do *Kubernetes*). Com a crescente da arquitetura de microsserviços mundialmente, essas duas stacks de tecnologias foram as que mais se destacam nos últimos anos. A habilidade de criar uma aplicação autocontida incluindo todas as configurações e bibliotecas dependentes empacotadas de forma portátil tem sido revolucionária [Sparks 2018].

Figura 1. Uso de orquestradores com Docker.

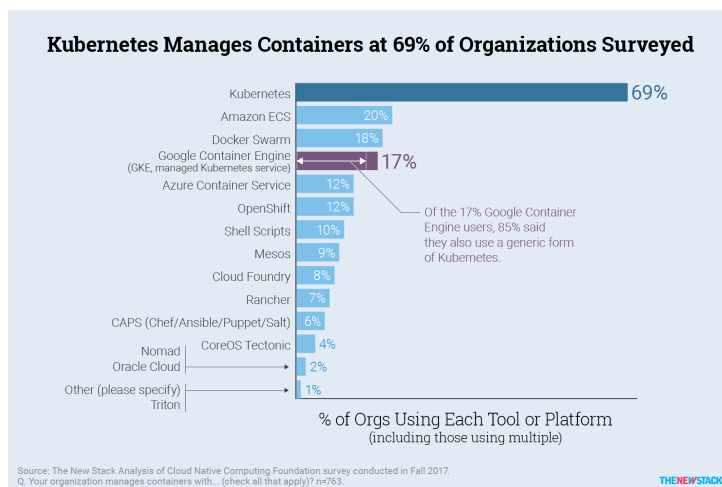


O motto de "*Build, Ship, Run*" faz total sentido com *Docker*, onde podemos escolher o sistema operacional base para a imagem do *container*, juntamente com a flexibilidade de configurar a maneira desejada como se estivesse no seu próprio computador, não importante se o uso final da imagem seja numa máquina física, virtual ou em cloud. Tornando assim cada vez próximo o contexto de Desenvolvimento e Infraestrutura, um paradigma que domina o campo da tecnologia a anos. Conforme mostra a imagem 1, é possível usar *Docker* em diversos tipos de tecnologias e ferramentas.

Grandes empresas fazem o uso de Docker, tais como ThoughtWorks, Spotify, Twitter, Pinterest, e de campos que variam entre Tecnologia da Informação em geral, Saúde, Finanças, Consultoria, Recursos Humanos, Telecomunicações.

E para orquestrar esses serviços em containers, *Kubernetes* é usado para orquestração em 69 por cento dos casos, mais aproximadamente 10 por cento de um genérico *Kubernetes*[Inc. 2018]. Entre as plataformas de orquestração existentes, *Kubernetes* é a mais popular, sendo portátil, configurável e modular e é amplamente suportado em diferentes tecnologias de computação na nuvem como Amazon AWS, IBM Cloud, Microsoft Azure, e Google Cloud [Xu et al. 2018].

Figura 2. Uso de orquestradores com Docker.



3. Estado Atual do Trabalho

3.1. Gerência de Configuração e Ambiente

Para possibilitar a implementação desse sistema, utilizou-se o Spring Boot, um framework na linguagem Java que fornece configurações mais simplificadas através de *annotations* e arquivos de configuração em *.yaml* ou *.properties*, agilizando o desenvolvimento de aplicações e possuindo vasta quantidade de bibliotecas para implantar as configurações necessárias, a depender do provedor de dependências utilizado. Neste projeto, foi utilizado *Apache Maven* como este gerenciador de dependências, assim como ferramenta de *build*.

Com sua arquitetura feita primordialmente para a aplicação ser executada na rede local de uma única máquina, foi utilizado o *Eureka* para prover *Service discovery*, em que os microsserviços que precisam comunicar-se entre si utilizem o servidor *Eureka* ao invés de endereços IP fornecidos manualmente nos arquivos de configuração dos módulos.

Também é feito o monitoramento de requisições HTTP das APIs dos serviços através do *Hystrix*, uma biblioteca que fornece configurações para tolerância a falhas caso os módulos que estejam habilitados a utilizá-la estejam fora do ar, e provê um *dashboard* para que sejam acompanhadas informações como quantidade total de requisições, quantidade de requisições bem ou mal sucedidas e entre outras.

O *RabbitMQ* é utilizado nesse projeto para permitir emissão de eventos atuando como um broker de mensagens e emitindo eventos de um dado módulo aos serviços aos quais estão interessados naquele evento para ser executada uma decisão ou tomar conhecimento de um estado do sistema. Finalizando a apresentação das tecnologias já existentes no sistema, tem-se o banco de dados não relacional *MongoDB*, onde cada serviço cria seu próprio *database* de acordo com a lógica necessária, em questão de entidades pertinentes, para seu funcionamento.

Partindo para falar das tecnologias que só se tornaram presentes neste trabalho, devemos começar com *Docker*, que fornece uma camada de abstração logo acima da camada do sistema operacional através de *container*, garantindo efemeridade e isolamento

dos serviços, de forma que seja possível monitorar com maior granularidade caso haja problemas e também de fazer implantação de novas versões, a depender da demanda.

Além disso, os *containers* de *Docker* são uma tecnologia chave para habilitar o uso de microsserviços, oferecendo um encapsulamento leve de cada componente de forma que facilite a manutenção e a atualização independentes [Docker 2019a]. Entre as muitas outras funções que *Docker* pode fornecer, utilizou-se nesse projeto não somente seus *containers* como também foram criadas imagens, em que era possível configurar um *container* para executar os módulos da aplicação e expor a porta configurada no arquivo de cada um no intuito de habilitar a comunicação dos serviços e sua disponibilidade através de requisições HTTP. Os *containers* isolam o software de seu ambiente e garante que funcione uniformemente, desconsiderando diferenças entre instâncias em fases de desenvolvimento ou de implantação [Docker 2019b].

Para podermos orquestrar os microsserviços e implantá-los numa arquitetura inspirada com base na nuvem, utilizamos o *Minikube*. Segundo [Google 2019b], *Minikube* executa em um único nó de *clusterKubernetes* dentro de uma máquina virtual em seu computador pessoal para usuários que estão experimentando *Kubernetes* ou usando-o em seu cotidiano de desenvolvimento. O *Minikube* nos permitiu obter uma base de conhecimento inicial sobre como funciona e quão complexo é no mundo real para orquestrar, expor e escalar uma aplicação em microsserviços numa arquitetura em nuvem de verdade

3.2. Características da aplicação

A aplicação *Greenhouse delivery* é uma aplicação onde o cliente pode escolher o restaurante desejado através do nome, fazer pedidos utilizando um menu, assim como quantidade, além de notas para fazer observações de restrição alimentar, por exemplo, assim como especificar o endereço de entrega. Com o pedido feito, o cliente paga, utilizando o número do cartão, data de validade e código de segurança. Para contemplar esse escopo, a aplicação foi dividida nos seguintes serviços:

- **Restaurant service:** Parte responsável por cadastro e listagem tanto de restaurantes quanto de itens de cardápio de cada estabelecimento
- **Order service:** Módulo em que é feito o registro do pedido, dada a escolha de restaurante, prato, quantidade e endereço de entrega
- **Payment service:** Módulo responsável pelo fornecimento dos dados necessários para o pagamento (cartão de crédito)
- **Payment distribution:** Parte da aplicação que dirá a ordem do pedido e avisar através de evento (com RabbitMQ) a realização desse pedido para outros módulos do sistema que necessitam dessa informação
- **Order complete updater:** Parte responsável por "concluir" o pedido de forma que o cliente possa acompanhar a estimativa de tempo para entrega

3.3. Visão de Análise e Projeto (Arquitetura)

Tem-se em mente que, ao escolhermos o *Minikube*, temos um ponto crítico de falha por executar localmente em uma única máquina. Porém, temos uma separação, orquestração e gerenciamento mais refinado e robusto de toda aplicação em si.

Dado este cenário, a arquitetura do sistema pode ser visualizada no diagrama abaixo:

3.4. Visão de Implantação

Nesta seção serão apresentadas as tecnologias utilizadas para ampliar o escopo da modelagem em microsserviços, assim como incluir conceitos de *DevOps*, como facilidade de build e deploy. Dessa maneira, foi feita uma mudança de servidor local para nuvem e de configuração para garantir mais vantagens que a arquitetura em microsserviços pode fornecer.

No presente trabalho, o *Minikube* foi configurado para não ser utilizado em uma máquina virtual, visto que é possível usá-lo em um computador desde que atenda os requisitos de utilizar um sistema operacional baseado em Linux e ter o *Docker* instalado. Com tudo isso em mente, a arquitetura para dar base ao projeto deste trabalho foi feita da seguinte forma, através do *Minikube*:

1. Criação de deployments para cada microsserviço através do comando *kubectl run*
2. Criação dos services de cada um dos deployments através do comando *kubectl expose deployment*

Para que a arquitetura fique de fato bem especificada com base no que foi dito nos passos ditos acima, o que acontece no *Minikube* é a criação de um *pod* para cada *deployment*, e de acordo com a necessidade de expor esses deployments para que sejam acessados tanto pelos microsserviços quanto para que sejam acessados pelo usuário, são criados *services*. Nesse caso, temos um *pod* para cada microsserviço, ligado a seu respectivo controlador *Deployment*, e com seus *Services*. Isso garante o encapsulamento dos microsserviços, do banco de dados e do broker de mensagens, utilizando o protocolo TCP no ambiente do *Minikube* para serem expostos na rede, e usando o endereço IP associado ao *service* de um módulo para reconfigurar o microsserviço que depende dele, sabendo qual é o seu *host* para permitir, por exemplo, requisições HTTP para CRUD no banco de dados.

Definindo cada um dos termos do parágrafo anterior, um *Pod* encapsula o(s) contêiner(es) de uma aplicação, armazena recursos, uma rede IP única, e opções que administram como os *containers* devem executar. Um *Pod* representa uma unidade de desenvolvimento: uma instância única de uma aplicação em *Kubernetes* [Google 2019c]. No caso dos *Deployments*, ele é usado para descrever um estado desejado para o *Pod*, e o controlador *Deployment* muda o estado atual para o estado desejado a uma taxa controlada [Google 2019a]. Sendo assim, esse objeto serve para regressar versão de um módulo, escalar o módulo, pausar ou executar uma forma de rollback caso haja algum problema. Por fim, o *Service* no contexto de *Kubernetes* é uma abstração que define um conjunto lógico de *Pods* e uma política a qual deve ser seguida para acessá-los [Google 2019d].

3.5. Visão de Uso

Dada que a implementação do sistema em si não fornece uma interface gráfica para usuários comuns, e somente telas de visualização das entidades do banco de dados, elas servem apenas para ver as coleções de objetos e realizar requisições HTTP GET, listando ou encontrando uma entidade em específico. Porém, visto que a aplicação trata-se de um serviço de pedido e entrega de comida, assemelhando-se a aplicações já conhecidas como **Uber Eats** e **Ifood**, sua visão de uso consiste em:

- Ver restaurantes disponíveis e poder escolher um para fazer um pedido

- Ver as opções de refeição de um restaurante e seus respectivos preços por unidade ou porção
- Fazer um pedido a esse restaurante, fornecendo a quantidade de pratos e visualizando quanto deve pagar ao total
- Cadastrar o endereço de entrega do pedido
- Saber o tempo estimado de entrega do pedido ao ser finalizado
- Cadastrar um cartão de crédito para realizar o pagamento do pedido

4. Descrição e Avaliação dos Resultados

4.1. Contribuições

Para executarmos as propostas do projeto em si, foi decidido seguir uma sequência de passos definida de acordo com o escopo do projeto e o que seria feito, dividindo em etapas, priorizando o que seria essencial naquele determinado momento.

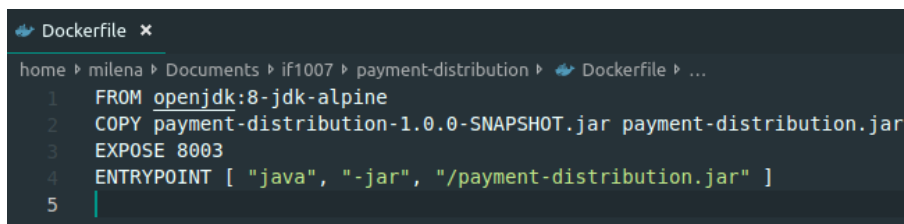
1. Remover a execução local de todos os serviços.
2. Escrever o *Dockerfile* para os serviços em Spring.
3. Conectar todos os serviços em *Spring* com o banco de dados, *RabbitMQ*, *Hystrix* e *Eureka*.
4. Containerizar todos os serviços juntos, em um docker-compose.
5. Utilizar o minikube para rodar os serviços fora da máquina local e em custers.
6. Adicionar o componente do *ElasticSearch* aos serviços já existentes.

Durante a execução do projeto, foi possível aprender mais sobre o uso de *Docker*, assim como a comunicação entre os containers de forma efetiva. A familiarização com ferramentas como o *Eureka* e o *Hystrix* também foi uma grande lição aprendida durante a execução, além do uso do *Minikube* para orquestração dos serviços. Vale ressaltar também entre as lições aprendidas, a necessidade de um razoável poder de processamento e memória necessária para dar conta de uma stack de serviços como essa, assim como um gerenciamento e configuração de redes dentro do *Docker* e *Minikube*.

4.2. Revisão do Projeto

Tendo em mente os pontos necessários para o escopo, iniciou-se as implementações necessárias. Primeiramente, foi feita uma análise do projeto, de forma que pudessemos nos familiarizar com as funcionalidades e como ele funciona, em seguida, foi realizada a retirada dos *scripts* manuais que eram utilizados para inicializar os serviços na máquina local, seguido da escrita de um *Dockerfile* para cada serviço *Spring*, iniciando assim o processo de *dockerização* de todos os microsserviços.

Neste ponto já nos deparamos com um erro, onde devido a uma dependência do *payment service* com o serviço de troca de mensagens, o *RabbitMQ*, não conseguimos subir esse *container* isoladamente. Decidimos que após a migração de todos os módulos para *Docker* conforme ilustra a figura 3, faríamos um novo teste com todos os módulos integrados, para resolver essa dependência. Ao colocar todos os serviços em *container*, ainda que com a criação de um *docker network*, os serviços não "se enxergavam", ou seja, eles não conseguiam se comunicar, nem se registrar no servidor *Eureka*. Por isso, foi criado um arquivo docker-compose com a configuração de *network_mode* para usar a rede local.

A screenshot of a code editor showing a Dockerfile. The file path is 'home > milena > Documents > if1007 > payment-distribution > Dockerfile > ...'. The code contains five lines: 1. FROM openjdk:8-jdk-alpine, 2. COPY payment-distribution-1.0.0-SNAPSHOT.jar payment-distribution.jar, 3. EXPOSE 8003, 4. ENTRYPOINT ["java", "-jar", "/payment-distribution.jar"], 5. (empty line).

```
1 FROM openjdk:8-jdk-alpine
2 COPY payment-distribution-1.0.0-SNAPSHOT.jar payment-distribution.jar
3 EXPOSE 8003
4 ENTRYPOINT [ "java", "-jar", "/payment-distribution.jar" ]
5
```

Figura 3. Dockerfile do serviço Payment-distribution

Desta forma, foi possível tanto a familiarização com o *Docker compose* quanto com a realização de testes do sistema para saber se ele funcionava da forma como já funcionava antes. Obtendo sucesso nessa fase, passou-se a pesquisar entre usar o *Minikube* ou usar o próprio *Kubernetes* através da *Google Cloud*. Por conta da possibilidade de gastos elevados com o uso da API, apesar do fornecimento de créditos de US\$300 pela própria Google durante 12 meses, decidiu-se usar o *Minikube*. O grande desafio foi configurá-lo da forma correta e em qual sistema operacional. As primeiras tentativas, mal sucedidas, foram no ambiente do Windows, que possui configurações conflitantes, especialmente com Docker em relação ao VirtualBox, o que não permitiu o uso deste para usar como *vm-driver* para criar uma máquina virtual para o *Minikube*. Ao tentar usar o HyperV, já presente no Windows 10 versão 1803, houve um pico de uso de CPU que foi o suficiente para não conseguir dar conta de gerenciar e de "matar" essa máquina virtual, e sequer de fazer a instalação dos requisitos do *Minikube*.

Por conta dessas dificuldades com o Windows, passou-se para o Linux. Usando Ubuntu 18.04 e com Docker instalado, não se fez necessário usar um driver de máquina virtual, porém houve certa dificuldade para descobrir o que impedia de abrir o *dashboard* do *Minikube*, recebendo um código de erro 503 por mau funcionamento inesperado do proxy. Após muito pesquisar em documentações e issues do Github, encontrou-se uma falha já conhecida quando executado o *Minikube* no ambiente em questão. Bastava passar um arquivo de configuração que permitia o funcionamento regular do CoreDNS do *kubelet* usado no *Minikube*, que provavelmente entrava em conflito com o DNS do próprio sistema operacional.

A partir do ponto acima, criar os Pods, Deployments e Services no *Minikube* foi menos trabalhoso. Apesar de que, durante as tentativas de usar *Minikube* no Windows, foi utilizado o *kompose*, que transforma arquivos de *docker-compose* em arquivos de configuração de Deployments e Services das imagens, com extensão *.yaml*. Porém, de todos os arquivos gerados pelo *kompose*, foi utilizado apenas o de Deployment do *RabbitMQ*, visto que o comando *kubectl run* para criação de Deployments e Pods não permitia fornecer mais de uma porta. A justificativa de usar *kubectl run* é a de ter disponível o Docker da própria máquina para especificar as imagens sem a necessidade de *docker push*, algo que foi longe de ser possível no ambiente Windows por conta dos problemas de configuração conflitante.

Por fim, apesar de o *RabbitMQ* enxergar o *binder* conectado a ele, ao visualizar a tela de Queues nada acontece. O que pode-se assumir, devido a problemas de gerenciamento e divisão de tarefas e esforço, é que isso possa ser um problema de implementação do *binder* e não da configuração dos módulos do serviço e do *RabbitMQ* no ambiente do *Minikube*. É necessária uma revisão do código para poder concluir de fato se o problema

está nele.

4.2.1. 12 Factors

Visto que as boas práticas de arquitetura em microsserviços está alinhada com os fatores listados no *The Twelve-Factor App*, a seguir temos quais deles foram atingidos neste trabalho e de que maneira:

- **Dependências:** Ainda que, para ser possível configurar os serviços ligados ao RabbitMQ e MongoDB precisassem do endereço de ClusterIP dentro do *Minikube*, essas dependências não foram implantadas nos serviços de forma que gerasse uma repetição de código, mas sim cada um desses módulos executava de forma independente. Além disso, cada pod continha apenas um único container para cada módulo do sistema
- **Processos:** Nenhuma das aplicações foi feita para guardar estados, e a utilização dos serviços através de API RESTful também garantia que não haveria como guardar estado atual
- **Vínculo de porta:** A partir do momento em que os serviços foram "containerizados", para comunicar-se com outros módulos bastava se comunicar através de *port binding*
- **Serviços de apoio:** O acesso aos serviços, ao banco de dados e ao RabbitMQ é sempre através de uma URL, especificando em qual *host* e a porta para acessar um determinado serviço
- **Concorrência:** graças ao *Minikube*, ainda que no ambiente local, é possível escalar um serviço caso seja necessário, e fazer isso em um serviço de maneira individual, já que cada um contém seu deployment (controlador que permite a escalabilidade).
- **Descartabilidade:** O *Minikube* gerenciava a reinicialização dos deployments caso houvesse lançado alguma falha.

Já tratando-se dos fatores os quais não foram alcançados, mas explicitando de que maneira poderia ser feito, temos:

- **Base de código:** Visto que foi utilizado o repositório da primeira versão do Greenhouse delivery e ele próprio já armazenava o código e os builds de todos os serviços nesse mesmo repositório e na mesma branch, além de ter sido desenvolvida por outra equipe, não foi feita a transição para que cada serviço tivesse sua própria base. Nosso repositório, de forma semelhante, armazena todos os arquivos de todos os serviços em um mesmo repositório, apenas fazendo a separação de cada um por pastas. Ao passo em que fosse dada a continuação de implantar um pipeline completo, com integração e entrega contínua, em nuvem de fato, poderia ser feita essa separação da base de código.
- **Configurações:** Não utiliza-se banco de dados, nem algum *ServerConfig* para armazenar as configurações, que seria o ideal quando trata-se de um projeto continuamente em manutenção e expansão. Neste projeto, são usados os arquivos de configuração em cada módulo do *Spring Boot* em extensão *.yaml*.
- **Build, release, run:** Visto que estamos lidando com um ambiente de máquina local para rodar o *Minikube*, não há ainda a automação dessas etapas. Apesar disso, o Jenkins provavelmente seria uma escolha para fornecer essa capacidade.

- **Logs:** Apesar de no nosso documento de *Goal, Questions and Metrics* termos colocado como um dos objetivos inserir o *Elasticsearch* nos serços e aproveitá-lo para também termos log, não foi possível nesta versão devido a problemas de gerenciamento de esforço da equipe. A famosa stack ELK seria suficiente para termos não apenas logs mas também um *dashboard* próprio para fazer o monitoramento.
- **Processos de admin:** Como apenas um único membro da equipe quem fez todo o processo de implantação da aplicação para o *Minikube*, não houve problemas quanto à configurações semelhantes e versões dos requisitos necessários, visto que tudo foi feito num mesmo computador.
- **Dev/prod semelhantes:** Dado que a aplicação foi desenvolvida por uma equipe e o intuito deste projeto foi implantar um pipeline para que fosse possível orquestrar e disponibilizar o serviço em um ambiente ligeiramente diferente, o mais próximo possível de cloud graças ao *Minikube*, certamente não teve esse alinhamento de ambientes de desenvolvimento e produção. Além disso, pouco código foi alterado durante a execução deste trabalho, portanto foi mais um trabalho sobre configurar ambientes e ferramentas do que de implementação.

4.3. Revisão Individual

- **João:** A lição mais importante certamente foi a importância do gerenciamento de tempo e atividades. A conciliar todas as atividades foi um grande desafio na reta final do projeto. Entretanto, uma dos maiores pontos positivos desse projeto foi justamente se ver diante de problemas práticos que se enfrentam no uso das tecnologias envolvidas, principalmente no que envolve a comunicação de containers em rede local, ou na própria rede do Docker. Como quando colocamos todo o conceito de orquestração, clusters e pods, que o *Kubernetes* trás. Certamente as dificuldades enfrentadas nos ajudarão bastante quando nos encontrarmos em um cenário com as tecnologias utilizadas. Foi possível ver de perto o funcionamento de um serviço de aplicações, mesmo que de maneira básica, dentro dessa arquitetura de microsserviços, e na prática observando as vantagens e desvantagens. Consegui exergar com mais clareza também, como outros aspectos que envolvem Microsserviços e DevOps, mas que não fizeram parte do escopo do projeto, podem ser acoplados e como funcionariam dentro desse contexto, tais como o monitoramento dos modulos de serviços e *clusters* do *Kubernetes*, e a relevância que uma ferramenta como o ElasticSearch ou o ELK podem oferecer a um produto.
- **Milena:** Uma das lições mais importantes foi justamente a de que é preciso obter conhecimento através das documentações das tecnologias utilizadas antes de querer usá-las, pois não sabendo como e por qual razão deve ser usada tais configurações ou quais as limitações existentes ao implantar configurações específicas, perde-se muito tempo em força bruta para fazer funcionar. Adicionalmente, que através do bom entendimento das tecnologias e das configurações, não teria dedicado esforço no ambiente Windows, que é bastante problemático para a implantação das tecnologias usadas neste projeto. Porém, apesar das dificuldades enfrentadas, foi possível conhecer tecnologias e possibilidades completamente novas, e que, ainda com possíveis melhorias e sempre algo a acrescentar neste projeto, este trabalho mostra e fornece algumas das vantagens que a arquitetura em microsserviços pode proporcionar. Adicionalmente, as vantagens que

o *Minikube* trouxe, pois ainda que limitado à máquina local, ele é bastante poderoso, no sentido de poder automatizar ou tornar um deploy muito mais simples, além de ter autonomia para reiniciar os serviços caso dê algum problema e tornar fácil a adaptação da versão executada localmente. Mesmo com todas as dificuldades, vendo que não foram extremamente complexas, foi possível adaptar mesmo não totalmente a versão executada localmente do Greenhouse delivery para o *Minikube*, o que consequentemente funcionaria em *cloud* com *Kubernetes* dado os conhecimentos adquiridos através do presente trabalho.

Referências

- Docker (2019a). Microservices delivery management.
- Docker (2019b). What is a container?
- Google (2019a). Deployments.
- Google (2019b). Installing kubernetes with minikube.
- Google (2019c). Pod overview.
- Google (2019d). Service.
- Hackernoon (2018). Why spring boot is an excellent choice for your next applications.
- HackerNoon (2019). Hacker noon: What is containerization?
- Inc., L. U. E. (2018). Interesting facts: Companies and the use of docker.
- Sparks, J. (2018). Enabling docker for hpc. *Concurrency and Computation: Practice and Experience*, page e5018.
- Spring (2019). Spring: the source for modern java.
- Xu, C., Rajamani, K., and Felter, W. (2018). Nbwguard: Realizing network qos for kubernetes. In *Proceedings of the 19th International Middleware Conference Industry*, Middleware '18, pages 32–38, New York, NY, USA. ACM.