UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

João Paulo Duarte Lucio

ANÁLISE COMPARATIVA ENTRE ARQUITETURA MONOLÍTICA E DE MICROSSERVIÇOS

Florianópolis

2017

João Paulo Duarte Lucio

ANÁLISE COMPARATIVA ENTRE ARQUITETURA MONOLÍTICA E DE MICROSSERVIÇOS

Monografia submetida ao curso de Sistemas de Informação para a obtenção do Grau de Bacharel em Sistemas de Informação.
Orientador: Prof. Frank Siqueira

Florianópolis

2017

João Paulo Duarte Lucio

ANÁLISE COMPARATIVA ENTRE ARQUITETURA MONOLÍTICA E DE MICROSSERVIÇOS

Esta Monografia foi julgada aprovada para a obtenção do Título de "Bacharel em Sistemas de Informação", e aprovada em sua forma final pelo curso de Sistemas de Informação.

Florianópolis, 26 de setembro 2017.

Prof. Cristian Koliver
Coordenador do Curso

Banca Examinadora:

Prof. Leandro José Komosinski

Prof. Frank Siqueira
Orientador

Prof. Mario Antonio Ribeiro Dantas

Dedico o desenvolvimento deste trabalho à minha família, amigos e colegas de trabalho que apoiaram, suportaram, incentivaram e me ispiraram em todas as etapas do projeto.

AGRADECIMENTOS

Sou eternamente grato à minha irmã, Bruna Kalinoski e ao meu cunhado, Markian Kalinoski, pelo incondicional apoio em todos os aspectos sejam eles físicos, mentais e intelectuais e que se fazem presentes neste projeto. Também devo agradecer a minha querida mãe e querido pai, Sonia Duarte e João Batista Lucio, pelo apoio, paciência e principalmente por compreender minha ausência durante o período que estive focado no desenvolvimento deste trabalho. Agradeço meus grandes amigos, Vitor Arins e Thiago Diniz, por serem grandes colegas de curso e fontes de inspiração, não só para este trabalho, mas para a vida profissional e pessoal. Por fim, agradeço Aquele à quem confio e acredito que, sem o Seu desejo, nada disto seria possível!

Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior. Its heart is a series of small behavior preserving transformations. Each transformation (called a 'refactoring') does little, but a sequence of transformations can produce a significant restructuring. Since each refactoring is small, it's less likely to go wrong. The system is also kept fully working after each small refactoring, reducing the chances that a system can get seriously broken during the restructuring.

Martin Fowler at refactoring.com as cited in: Lawrence Bernstein, C. M. Yuhas (2005) Trustworthy Systems Through Quantitative Software Engineering. p. 266

RESUMO

Com o intuito de agregar os exemplos didáticos existentes, enriquecer a fonte de pesquisa para o domínio de arquiteturas de software e apresentar testes reais mais expressivos entre a arquitetura de microsserviços e monolítica, este trabalho tem por objetivo demonstrar uma análise comparativa entre duas arquiteturas de software através de uma avaliação comparativa entre o desenvolvimento de uma aplicação monolítica de um sistema gerenciador de cinemas, escrito utilizando a linguagem Javascript, e uma aplicação idêntica utilizando a arquitetura orientada à microsserviços. Novas arquiteturas de desenvolvimento como microsserviços ganham cada vez mais destaque nos campos de desenvolvimento, pesquisa e mercado tecnológico, e por isso, torna-se interessante traçar um comparativo entre os prós e contras encontrados entre as duas arquiteturas de desenvolvimento. A fim de estabelecer um comparativo fiel e didático, foram analisados dois protótipos idênticos em termos de funcionalidades sendo um orientado ao desenvolvimento monolítico e o outro, uma aplicação orientada à microsserviços, ambas escritas em Javascript sobre a plataforma NodeJs e utilizando banco de dados não-relacional. A construção do protótipo monolítico baseando-se em um projeto pré existente em microsserviços, evidencia características quantitativas, qualitativas e particularidades sobre o desenvolvimento em cada metodologia. Ambas aplicações foram submetidas à critérios de performance, quantidade de código escrito e análise estrutural e, à partir dos resultados obtidos, foram apresentados e comentados seus comparativos para facilitar a compreenção dos resultados obtidos e esperados. Considerando a singularidade das características de cada aplicação, as tecnologias envolvidas ou os aspectos de gestão mencionados, este trabalho, ao fazer um estudo comparativo entre estas duas arquiteturas pode auxiliar, seja uma organização em sue processo de tomada de decisão sobre a arquitetura a ser adotada em um projeto, seja profissionais da área de desenvolvimento de software, estudantes e entusiastas para terem um melhor entendimento de sua aplicabilidade. Como resultado obtido à partir dos testes realizados neste trabalho, foi observado que a arquitetura monolítica pode sim desempenhar uma melhor performance comparado ao microsserviço.

Palavras-chave: Microsserviços. Monolítico. Arquitetura de Software.

ABSTRACT

With the purpose of aggregating the existing didactic examples, enriching the research source for the domain of software architectures and presenting more expressive real tests between the microservice and monolithic architecture, this work aims to demonstrate a comparative analysis between two software architectures through a comparative evaluation between the development of a monolithic application of a cinema management system, written using the Javascript language, and an identical application using the microservice oriented architecture. New development architectures such as microservices are gaining more prominence in the fields of development, research and technological market, and it is therefore interesting to draw a comparison between the pros and cons found between the two development architectures. In order to establish a faithful and didactic comparison, two identical prototypes were analyzed in terms of functionalities being one orientated to the monolithic development and the other, a application oriented to the microservices, both written in Javascript on the platform NodeJs and using non- relational The construction of the monolithic prototype based on a pre-existing project in micro-services, shows quantitative, qualitative and developmental characteristics in each methodology. Both applications were submitted to performance criteria, written code quantity and structural analysis and, based on the obtained results, their comparatives were presented and commented to facilitate the understanding of the obtained and expected results. Considering the uniqueness of the characteristics of each application, the technologies involved or the aspects of management mentioned, this work, when doing a comparative study between these two architectures can help, either an organization in the process of decision making about the architecture to be adopted in a project, be software development professionals, students and enthusiasts to have a better understanding of its applicability. As a result of the tests performed in this work, it was observed that the monolithic architecture can perform better than the microservice.

Keywords: Microservices. Monolith. Software Architecture.

LISTA DE FIGURAS

LISTA DE ABREVIATURAS E SIGLAS

SUMÁRIO

1 INTRODUÇÃO

Microsserviços estão atualmente ganhando muito destaque no mercado de desenvolvimento de software e sendo discutidos em posts, blogs, artigos, redes sociais, conferências e apresentações. Alguns céticos da comunidade ignoram o termo por acreditar que não há nada de novo, apenas uma repaginação da arquitetura SOA (*Service-Oriented Architecture*). Entretanto, o padrão de arquitetura orientado a microsserviços, que surgiu com a finalidade de otimizar a escalabilidade, a implantação (*deploy*) e a gestão de equipes de desenvolvimento advindas de uma arquitetura monolítica, possui benefícios significativos especialmente quando isso possibilita um desenvolvimento mais ágil, com implantação facilitada, arquitetura de múltiplas linguagens simultâneas, e melhor adaptada para o novo cenário constituído por aplicações em nuvem. /

1.1 PROBLEMA

Para entender o padrão de arquitetura orientado a microsserviços precisamos primeiramente estudar a arquitetura que era utilizada até então para a construção de aplicações, a chamada arquitetura monolítica.

> Quase todas as histórias bem sucedidas de microsserviços começaram com um monólito que ficou muito grande e acabou sendo quebrado. E, quase todos os casos em que ouvi falar de um sistema que foi construído como um sistema de microsserviços a partir do zero, acabou com sérios problemas. (FOWLER, 2010)

Arquitetura monolítica, relativo à monólito, refere-se ao modelo de desenvolvimento de uma aplicação dentro de uma única estrutura executável, onde o comportamento obtido no resultado final do seu desenvolvimento poderemos encontrar uma única entidade, compacta, indivisível e impenetrável, que posteriormente será executada.

1.2 OBJETIVO

Com o intuito de agregar os exemplos didáticos existentes, enriquecer a fonte de pesquisa para o domínio de arquiteturas de software e apresentar testes reais mais expressivos entre a arquitetura de microsserviços e monolítica, este trabalho tem por objetivo demonstrar uma análise comparativa entre duas arquiteturas de software através de uma avaliação comparativa entre o desenvolvimento de uma aplicação monolítica de um sistema gerenciador de cinemas, escrito utilizando a linguagem Javascript, e uma aplicação idêntica utilizando a arquitetura orientada à microsserviços, destacando os pontos fortes e fracos dentro do modelo adotado para cada uma, assim como suas particularidades e cenários de melhor aplicabilidade.

Para fazer esta correlação serão executadas duas aplicações idênticas. Uma utilizando microsserviços e a outra com base na arquitetura monolítica. Serão destacados durante a sua construção cenários que apresentam vantagens e desvantagens, análises de performance e processos de implantação abordando aspectos de escalabilidade em nuvem.

1.2.1 Objetivos específicos

Como objetivo específico deste projeto, pretende-se demonstrar a comparação entre as duas arquiteturas tanto no âmbito de testes de desempenho quanto à realização de análises estruturais como quantidade de código escrito necessário e estrutura de projeto. Obtendo assim, como resultado final, dois projetos arquiteturais de funcionalidades equivalentes para que eventualmente sirvam a toda a comunidade como guia auxiliar para a elaboração de novos projetos.

1.3 JUSTIFICATIVA

Hoje podemos encontrar diversas linhas de pensamento dentro da comunidade de software que criticam a arquitetura de microsserviços como sendo uma renomeação para arquiteturas preexistentes, e não sendo constituída, portanto, de alguma originalidade. Percebemos a dificuldade de se obter clareza e solidez quanto às informações disponíveis na rede, bem como exemplos demonstrativos das arquiteturas supracitadas.

1.4 BENEFÍCIOS

Podemos ressaltar o benefício deste projeto como sendo uma contribuição à comunidade de desenvolvedores de software, bem como alunos e professores, quanto às diferenças observadas dentro dos dois paradigmas arquiteturais e também a disponibilização de ambos os projetos desenvolvidos para que possam servir de guias para futuras implementações.

1.5 METODOLOGIA

Com a intenção de atingir os objetivos propostos, este trabalho adota uma metodologia de estudo de caso comparativo de dois sistemas com características funcionais semelhantes, cada um desenvolvido com abordagens, tecnologias e arquiteturas distintas.

1.6 ORGANIZAÇÃO DO TEXTO

Para uma melhor compreensão e separação dos conteúdos, este trabalho será organizado em 5 capítulos e um apêndice com um artigo do referente à esta monografia.

Depois da introdução realizada no primeiro capítulo, o capítulo 2 apresenta a fundamentação teórica com as definições das abordagens de desenvolvimento de software que serão adotadas no projeto e também descreve como são aplicados ambos os modelos de implementação.

O capítulo 3 é composto pelo planejamento do experimento, as formas de avaliação comparativas, bem como as arquiteturas que serão adotadas neste trabalho para avaliar o desenvolvimento dos dois sistemas.

O capítulo 4 contém as informações relacionadas ao resultado dos testes aplicados, bem como a avaliação dos sistemas e também a apresentação dos dados obtidos através das metodologias escolhidas no capít ulo anterior.

No quinto capítulo estão as conclusões obtidas através dos resultados deste trabalho e as sugestões para trabalhos futuros sobre o assunto em questão.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 MONOLÍTICO

Em uma arquitetura monolítica, uma única aplicação é responsável por todos os processos. Ela apresenta a interface para interação com o usuário, acessa os dados persistidos no banco de dados, processa pedidos de clientes e todas as outras ações que a aplicação necessitar.

A aplicação monolítica é auto-suficiente e independente de outras aplicações de computação. A filosofia do projeto é que o aplicativo é responsável não apenas por uma determinada tarefa, mas pode também executar todos os passos necessários para completar uma macro função.

Em Engenharia de Software, uma aplicação monolítica descreve uma aplicação de software que é projetada sem modularidade externa, ou seja, sem a preocupação de construir uma aplicação que possa vir a ser um módulo para uma outra aplicação. A modularidade é desejável, em geral, uma vez que suporta a reutilização de partes da lógica da aplicação e também facilita a manutenção, permitindo a reparação ou substituição de partes da aplicação sem a necessidade de substituição total. Entretanto, as aplicações monolíticas em geral podem optar por uma modularização interna.

A modularidade pode ser conseguida em graus diferentes por diferentes abordagens de modularização. Modularidade baseada em código permite aos desenvolvedores reutilizar e reparar partes da aplicação, mas ferramentas de desenvolvimento são necessárias para executar essas funções de manutenção (por exemplo, a aplicação pode precisar ser recompilada). Modularidade baseada em objeto fornece o aplicativo como uma coleção de arquivos executáveis separados, que pode ser independentemente mantidos e substituídos sem reimplantar o aplicativo inteiro. Alguns recursos de mensagens de objetos permitem que aplicações baseadas em objetos sejam distribuídas em múltiplas plataformas, por exemplo, o Microsoft COM+. Arquiteturas orientadas a serviços usam especificamente um padrão de comunicação e protocolos para comunicação entre módulos.

Dizer que um software é, de fato, um monólito implica em adotar uma perspectiva sobre o assunto. Por exemplo, um software pode não estar orientado à serviço e pode ser descrito como um monólito, embora também possa ser baseado em objeto e estar virtualmente distribuído.

2.1.1 Modelo de Implementação

A abordagem monolítica é demonstrada neste trabalho através de uma aplicação única que conterá os mesmos serviços implementados no projeto desenhado com microsserviços, com as mesmas regras de negócio. Este modelo altamente acoplado nos obriga a subir todos os serviços simultaneamente a cada implantação. Utilizaremos o Javascript como linguagem principal de programação, ShellScript para automação de algumas tarefas de implantação e outras linguagens de especificação de API, como o RAML. / /

2.2 MICROSSERVIÇOS

> Um microsserviço é uma unidade de software autônoma que, juntamente com muitas outras, compõe uma grande aplicação. Ao dividir seu aplicativo em unidades pequenas, cada parte pode ser independentemente implantada e escalada; pode ser escrita por diferentes equipes de desenvolvimento, em diferentes linguagens de programação; e pode ser testada individualmente. (STOIBER,)

O objetivo de otimizar a implantação de softwares modulares, com ciclos de vida independentes, foi o que fomentou o aparecimento do termo Microsserviços. Ao longo dos anos, este termo utizado para descrever este novo tipo de arquitetura foi se constituindo de características semelhantes relativas à descentralização de linguagens, automação do processo de implantação, alinhamento ao negócio e organização de projetos e equipes.

> O termo "Arquitetura de Microserviços"surgiu nos últimos anos para descrever uma maneira particular de projetar aplicativos de software como suítes de serviços implementáveis independentemente. Embora não haja uma definição precisa desse estilo arquitetônico, existem certas características comuns em torno da organização, da capacidade comercial, implantação automatizada, inteligência nas chamadas da interface do sistema e controle descentralizado de idiomas e dados. (FOWLER, 2015)

Este novo termo, embora seja às vezes um pouco contraditório e

carente de informações a seu respeito, causando certa estranheza, não deve ser menosprezado. Este novo design arquitetural para o desenvolvimento de software tem chamado muito a atenção e atraído muitos olhares dentro e fora da comunidade tecnológica. Dentre os desbravadores do novo conceito estão, por exemplo, os gigantes do *streaming* Netflix e Spotify, e um dos maiores provedores de nuvem e de *e-commerce* mundiais, a Amazon.

> Os microsserviços são uma abordagem para sistemas distribuídos que promovem o uso de serviços finamente granulado com seus próprios ciclos de vida, que colaboram em conjunto. (NEWMAN, 2015)

Não temos como explicar devidamente o estilo microsserviços de se escrever um software sem compará-lo ao seu modelo antagônico, chamado monolítico. Aplicações web corporativas são basicamente constituídas de um *Frontend*, que é a interface do usuário, um banco de dados, normalmente relacional, para a persistência dos dados e um *Backend*, que consiste basicamente de um servidor de aplicação com as regras de negócio. Este último, quando desenvolvido monoliticamente, para toda regra de negócio, função ou serviço que se deseja fazer qualquer modificação, pode repercutir em um processo mais custoso de teste e implantação de todos os outros serviços, embora ainda assim seja construído de forma mais ágil. O fato de que todos os serviços são, em geral, escritos com a mesma linguagem, não possibilita usufruir o que cada uma delas tem de melhor a oferecer, porém facilita o intercâmbio na manutenção do código de todos os serviços entre diversos times de desenvolvimento. Importante destacar que nesta aplicação toda a sua lógica para lidar com uma solicitação é executada em um único processo, o que pode conferir ao monólito uma grande vantagem em termos de processamento.

2.2.1 Modelo de Implementação

Esta nova abordagem será demonstrada neste trabalho através de uma suíte de pequenos serviços que se comunicarão através de requisições HTTP. Estes serviços serão construídos sobre as regras de negócio de um sistema gerenciador de uma rede de cinemas, onde cada serviço terá seu deploy independente e automatizado. Apenas um gateway será responsável por centralizar e reconhecer estes serviços. Embora pudessem ser escritos em diferentes linguagens, utilizaremos o Javascript

como linguagem principal de programação, ShellScript para automação de algumas tarefas de implantação e outras linguagens de especificação como o RAML. /

2.3 ESCALABILIDADE

Denomina-se escalabilidade horizontal a capacidade de aumentar o poder de processamento ou a capacidade de armazenamento de uma aplicação essencialmente criando réplicas ao invés de melhorar um servidor pre-existente. Dessa forma, em vez de comprar um servidor mais robusto e mover toda a carga para ele, o responsável pela aplicação pode comprar um servidor adicional e distribuir a carga horizontalmente entre todos eles.

O escalonamento horizontal é usado quando existe a capacidade de executar simultaneamente múltiplas instâncias da aplicação em servidores distintos. É necessário destacar que, para que isso ocorra, a aplicação necessita ser projetada para trabalhar de forma distribuída. Normalmente, é muito mais difícil ir de 1 servidor para 2 servidores, do que ir de 2 para 5, justamente pela complexidade de se projetar uma aplicação distribuída.

Em alguns casos esta abordagem pode se fazer necessária, por exemplo, para guardar sessões de autenticações em disco. A não ser que este disco possa ser compartilhado entre diversos servidores, a abordagem mais adequada seria o escalonamento vertical. Outros casos que demandariam a mesma necessidade seria o caso de armazenamento por blocos, e bancos de dados que não possam ser distribuídos.

Na escalabilidade vertical a forma de escalar a aplicação, em que se amplia o tamanho do servidor/contêiner com mais CPU e memória RAM ou com maior espaço de armazenamento em disco, pode ser menos apropriada, mas é necessária ou até mesmo a única opção em algumas situações. Este tipo de escalabilidade ainda hoje é a mais comum e a mais utilizada para suportar a maioria das aplicações do mercado.

3 PLANEJAMENTO DO EXPERIMENTO

Para o desenvolvimento dos projetos que embasam este trabalho é tomado como exemplo e modelo um projeto de um Sistema Gerenciador de Cinemas (RAMIREZ, 2017a), no qual são utilizados microsserviços como arquitetura fundamental e cujo código foi disponibilizado publicamente na plataforma para compartilhamento de softwares de código livre, o Github. A partir deste trabalho foi desenvolvido um novo projeto com características funcionais idênticas, alterando-se basicamente a sua arquitetura para torná-lo um monólito provido das mesmas funcionalidades. Estes projetos, após implementados, foram comparados em termos de performance, construção do código, infraestrutura necessária e processos de implantação. Ao final do experimento será apresentado um resultado conclusivo sobre as duas plataformas.

A temática da aplicação de exemplo foi um Sistema de Gerenciamento de Cinemas composto por cinco serviços principais. Temos o serviço de "Booking" responsável por efetuar a compra dos assentos para um determinado filme em uma sala de cinema específica e também fará a verificação de uma ordem de compra já efetuada. O serviço "Cinema Catalog" retornará a lista de todos os cinemas cadastrados, bem como efetuará buscas por cinemas específicos sendo realizadas tanto por identificadores únicos dos cinemas como por identificadores de cidade e filme. O serviço de notificação será chamado de "Notification Service" e este estará responsável por enviar ao usuário final a confirmação da sua ordem de compra por email. Este serviço estará atrelado ao serviço de pagamento, "Payment Service", encarregado de validar e recuperar as ordens de compra e que, por sua vez, estará vinculado ao serviço de compra "Booking", previamente descrito. Por fim, o serviço de catálogo de filmes, denominado "Movies", proverá a lista dos filmes cadastrados, dos lançamentos de filmes (*premières*) e a obtenção de um filme específico através do seu identificador.

3.1 EXPERIMENTO MONOLÍTICO

3.1.1 Design

Na aplicação monolítica, como os serviços estão agrupados dentro de um mesmo projeto e os pontos de chamadas para a interface da aplicação foram agrupados em um mesmo arquivo de *endpoints* para

facilitar e ilustrar didaticamente a arquitetura da aplicação como demonstrado no trecho de código que segue.

Algorítmos 3.1 – Arquivo da API monolítica

```
'use strict'
const status = require('http-status')

module.exports = ({repo}, app) => {

  //BOOKING API
  app.post('/booking', (req, res, next) => {
    const validate = req.container.cradle.validate

    Promise.all([
      validate(req.body.user, 'user'),
      validate(req.body.booking, 'booking')
    ])
    .then(([user, booking]) => {
      const payment = {
        userName: user.name + ' ' + user.lastName,
        currency: 'mxn',
        number: user.creditCard.number,
        cvc: user.creditCard.cvc,
        exp_month: user.creditCard.exp_month,
        exp_year: user.creditCard.exp_year,
        amount: booking.totalAmount,
        description: `
          Tickect(s) for movie ${booking.movie},
          with seat(s) ${booking.seats.toString()}
          at time ${booking.schedule}`
      }

      return Promise.all([
        makePuchase(payment, validate),
        Promise.resolve(user),
        Promise.resolve(booking)
      ])
    })
    .then(([paid, user, booking]) => {
      return Promise.all([
        repo.makeBooking(user, booking),
        Promise.resolve(paid),
        Promise.resolve(user)
      ])
    })
    .then(([booking, paid, user]) => {
      return Promise.all([
        repo.generateTicket(paid, booking),
        Promise.resolve(user)
      ])
    })
    .then(([ticket, user]) => {
      const payload = Object.assign({},
      ticket, {user: {name: user.name + user.lastName, email: user.email}})
      repo.sendEmail(payload)
      .then(info => {
        res.status(status.OK).json(ticket)
      })
      .catch(next)
    })
    .catch(next)
  })

  app.get('/booking/verify/:orderId', (req, res, next) => {
    repo.getOrderById(req.params.orderId)
      .then(order => {
        res.status(status.OK).json(order)
      })
      .catch(next)
  })

  //MOVIES API
  app.get('/movies', (req, res, next) => {
    repo.getAllMovies().then(movies => {
      res.status(status.OK).json(movies)
    }).catch(next)
  })

  app.get('/movies/premieres', (req, res, next) => {
```

```
    repo.getMoviePremiers().then(movies => {
      res.status(status.OK).json(movies)
    }).catch(next)
  })

  app.get('/movies/:id', (req, res, next) => {
    repo.getMovieById(req.params.id).then(movie => {
      res.status(status.OK).json(movie)
    }).catch(next)
  })

//CINEMA-CATALOG API
app.get('/cinemas', (req, res, next) => {
    repo.getCinemasByCity(req.query.cityId)
      .then(cinemas => {
        res.status(status.OK).json(cinemas)
      })
      .catch(next)
  })

  app.get('/cinemas/:cinemaId', (req, res, next) => {
    repo.getCinemaById(req.params.cinemaId)
      .then(cinema => {
        res.status(status.OK).json(cinema)
      })
      .catch(next)
  })

  app.get('/cinemas/:cityId/:movieId', (req, res, next) => {
    const params = {
      cityId: req.params.cityId,
      movieId: req.params.movieId
    }
    repo.getCinemaScheduleByMovie(params)
      .then(schedules => {
        res.status(status.OK).json(schedules)
      })
      .catch(next)
  })

  //NOTIFICATION SERVICE API
  app.post('/notification/sendEmail', (req, res, next) => {

    const validate = req.container.cradle.validate
    validate(req.body.payload, 'notification')
      .then(payload => {
        return repo.sendEmail(payload)
      })
      .then(ok => {
        res.status(status.OK).json({msg: 'ok'})
      })
      .catch(next)
  })

  app.post('/notification/sendSMS', (req, res, next) => {
    const {validate} = req.container.cradle

    validate(req.body.payload, 'notification')
      .then(payload => {
        return repo.sendSMS(payload)
      })
      .then(ok => {
        res.status(status.OK).json({msg: 'ok'})
      })
      .catch(next)
  })

  //PAYMENT SERVICE API
  app.post('/payment/makePurchase', (req, res, next) => {
    const {validate} = req.container.cradle

    validate(req.body.paymentOrder, 'payment')
      .then(payment => {
        return repo.registerPurchase(payment)
      })
      .then(paid => {
        res.status(status.OK).json({paid})
      })
      .catch(next)
  })

  function makePuchase(paymentOrder, validate) {
```

```
    return new Promise((resolve, reject) => {
      validate(paymentOrder, 'payment')
        .then(payment => {
          return repo.registerPurchase(payment)
        })
        .then(paid => {
          resolve(paid)
        })
        .catch(e => reject(new Error('Payment error, err: ' + e)))
    })
  }

  app.get('/payment/getPurchaseById/:id', (req, res, next) => {
    repo.getPurchaseById(req.params.id)
      .then(payment => {
        res.status(status.OK).json({payment})
      })
      .catch(next)
  })

}
```

3.1.2 Arquitetura

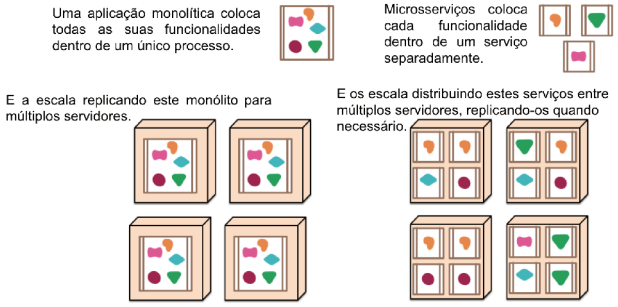

Figura 1 Comparativo entre Monolítico e Microsserviços, Adaptado de (FOWLER, 2015)

Para desenvolver a arquitetura deste monólito, tendo como premissa a execução de um único processo, todo o projeto foi refatorado para que então pudesse ser aplicada a bateria de testes comparativos. Portanto, nesta arquitetura encontraremos um único projeto a ser executado, que será responsável por prover todas as funcionalidades do sistema seguindo o modelo demonstrado na Figura 1.

3.2 EXPERIMENTO MICROSSERVIÇO

3.2.1 Design

No desenvolvimento da arquitetura de microsserviços, diferentemente da arquitetura proposta anteriormente, o projeto estará construído de tal forma que cada serviço seja um processo sendo executado independentemente. Nesta abordagem, cada processo é responsável pelo seu próprio ciclo de vida desde a sua concepção na fase de planejamento, podendo eventualmente utilizar uma linguagem que seja mais propícia ao propósito do serviço, até o ponto da sua implantação. Também necessitaremos utilizar um gerenciador dos serviços implantados que chamaremos de "gateway", visto que poderemos ter diversas instâncias do mesmo serviços implantadas individualmente. Este "gateway" será responsável por gerenciar as instâncias de todos os serviços implantados. Para que possamos lidar de forma mais eficiente na implantação destes serviços utilizaremos a ferramenta chamada Docker Machine, que nos possibilitará a construção de máquinas virtuais facilmente inicializáveis para que possamos implantar os serviços da aplicação.

Microserviços podem criar aplicações de muitos componentes simples, de uso único, fáceis de usar e que permitem a entrega de um software melhor mais rápido. Mesmo as arquiteturas monolíticas existentes poderiam ser transformadas usando o padrão arquitetural de microsserviços.

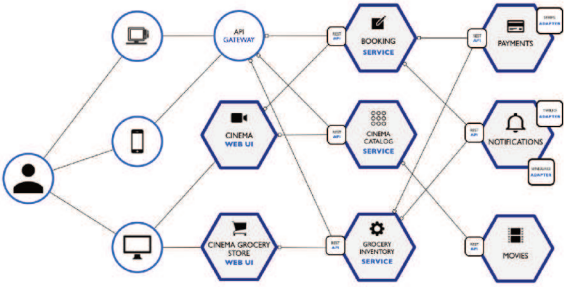

Figura 2 Visão geral do design da arquitetura do projeto em microsserviços. Fonte: (RAMIREZ, 2017a)

Nesta arquitetura, diferentemente da arquitetura monolítica, uma

estrutura específica denominada “API Gateway”, conforme pode ser observado na Figura 2, será utilizada para gerenciar os serviços que estarão disponíveis. Este serviço reconhecerá e disponibilizará as rotas dos serviços que estiverem a ela conectados. A classe do programa responsável por estas tarefas está demonstrada no exemplo à seguir.

Algorítmos 3.2 – Arquivo da API em microsserviços

```
'use strict'
const Docker = require('dockerode')

const discoverRoutes = (container) => {
  return new Promise((resolve, reject) => {
    const dockerSettings = container.resolve('dockerSettings')

    const docker = new Docker(dockerSettings)

    const getUpstreamUrl = (serviceDetails) => {
      const {PublishedPort} = serviceDetails.Endpoint.Spec.Ports[0]
      return `http://${dockerSettings.host}:${PublishedPort}`
    }

    const addRoute = (routes, details) => {
      routes[details.Spec.Name] = {
        id: details.ID,
        route: details.Spec.Labels.apiRoute,
        target: getUpstreamUrl(details)
      }
    }

    docker.listServices((err, services) => {
      if (err) {
        reject(new Error('an error occured listing containers, err: ' + err))
      }

      const routes = new Proxy({}, {
        get (target, key) {
          console.log(`Get properties from -> "${key}" container`)
          return Reflect.get(target, key)
        },
        set (target, key, value) {
          console.log('Setting properties', key, value)
          return Reflect.set(target, key, value)
        }
      })

      services.forEach((service) => {
        addRoute(routes, service)
      })

      resolve(routes)
    })
  })
}

module.exports = Object.assign({}, {discoverRoutes})
```

3.2.2 Arquitetura

Neste formato arquitetural implantaremos cada serviço de forma independente, podendo escalá-los entre as três máquinas virtuais (*docker machines*), conforme apresenta a Figura 3, disponibilizadas para tal. Nesta arquitetura também é importante destacar a presença de um serviço que até então não foi necessário para a concepção do pro-

jeto monolítico. Trata-se de um serviço responsável por ser o nosso único ponto de entrada na aplicação, já que esta aplicação poderá ter um ou mais serviços iguais, ou seja, ele tratará o reconhecimento, a distribuição de requisições e a redundância dos serviços.

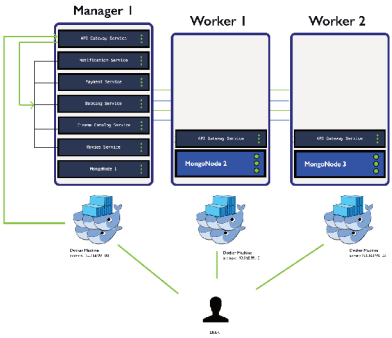

Figura 3 Arquitetura inicial das máquinas virtuais rondando microsserviços. Fonte: (RAMIREZ, 2017b)

3.3 STACK DE DESENVOLVIMENTO

Para o desenvolvimento deste projeto foi adotada uma pilha de tecnologias emergentes que estão sendo muito utilizadas no desenvolvimento de aplicações web e que podem ser aplicadas para ambas arquiteturas, com a finalidade de poder traçar um comparativo fidedigno entre elas.

3.3.1 Docker

Docker é uma tecnologia de software que fornece contêineres para a implantação de aplicações, promovido pela empresa Docker, Inc. (DOCKER,) O Docker fornece uma camada adicional de abstração e automação da virtualização do sistema operacional em Windows e Linux. O Docker usa os recursos de isolamento de recursos do kernel do Linux, como *cgroups* e *namespaces* do kernel, e um sistema de arquivos de junção compatível, como OverlayFS e outros, para permitir que os contêineres independentes sejam executados em uma única instância do Linux, evitando a sobrecarga de iniciar e manter máquinas virtuais

(VMs).

Em outras palavras, o Docker nos permite construir imagens para a geração de contêineres que contenham uma "máquina virtual" onde podemos implantar uma aplicação e instalar todas as suas dependências e executá-lo como um processo isolado, compartilhando o kernel com outros contêineres no sistema operacional hospedeiro ou em qualquer plataforma IaaS.

/ /

3.3.2 Rancher

Rancher é uma plataforma de gerenciamento para contêineres Docker desenvolvido e distribuido pela Rancher Labs. Ele inclui uma distribuição Kubernetes, bem como a opção de escolher entre Docker Swarm e Apache Mesos. Todos eles usam o Docker como suporte de tempo de execução do contêiner subjacente e coordenam os contêineres em execução entre múltiplos nós físicos discretos. Rancher também inclui serviços de infraestrutura modular, incluindo redes, balanceamento de carga, descoberta de serviços, monitoramento e recuperação. (RANCHER,)

3.3.3 Docker Swarm

Docker Swarm é uma ferramenta de agrupamento e agendamento para contêineres Docker. Com o Swarm, os administradores e desenvolvedores de TI podem estabelecer e gerenciar um cluster de nós Docker como um único sistema virtual. /

3.3.4 NodeJs

Node.js é uma plataforma aberta, multiplataforma e um ambiente de tempo de execução de JavaScript para executar código em JavaScript lado do servidor (NODEJS,). Historicamente, o JavaScript foi usado principalmente para scripts do lado do cliente, nos quais os scripts escritos em JavaScript estavam incorporados em arquivos HTML de uma página Web para serem executados no lado do cliente por um maquina de JavaScript no navegador do usuário. O Node.js permite que o JavaScript seja usado para construir e executar os scripts JavaScript do lado do servidor para produzir conteúdo Web dinâmico antes que

a página seja enviada para o navegador da web do usuário. Consequentemente, o Node.js tornou-se um dos elementos fundamentais do paradigma "JavaScript em todos os lugares", permitindo que o desenvolvimento de aplicativos web se unisse em torno de uma única linguagem de programação, em vez de exigir o uso de linguagens diferentes para escrever scripts do lado do servidor e do lado do cliente. /

3.3.5 Mongo

MongoDB é um programa de banco de dados livre, aberto e multiplataforma . Classificado como um programa de banco de dados NoSQL, o MongoDB usa documentos similares a JSON com esquemas. O MongoDB é desenvolvido pela MongoDB Inc. e é publicado sob uma combinação da Licença GNU Affero General Public e da Licença Apache. /

3.3.6 Express

Express.js, ou simplesmente Express, é uma estrutura de aplicação web para o Node.js, lançado como software livre e de código aberto sob a Licença MIT. É projetado para a construção de aplicativos web e APIs. Na verdade, é a estrutura padrão do servidor para Node.js. /

O autor original, TJ Holowaychuk, descreveu-o como um servidor inspirado em Sinatra, o que significa que é relativamente mínimo com muitos recursos disponíveis como plugins. Express é a parte do backend da pilha MEAN, juntamente com a plataforma Node.Js, o banco de dados MongoDB e o framework para *frontend* AngularJS.

3.3.7 JMeter

O Apache JMeter é um projeto Apache que pode ser usado como uma ferramenta de teste de carga para analisar e medir o desempenho de uma variedade de serviços, com foco em aplicativos da web. Adaptado de (APACHE,).

JMeter pode ser usado como uma ferramenta de teste unitário para conexões de banco de dados JDBC, FTP, LDAP, Web Services, JMS, HTTP, conexões TCP genéricas e processos nativos do sistema operacional. Pode-se também configurar o JMeter como uma ferramenta de monitoramento, embora este seja tipicamente considerado

apenas para teste em vez de monitoramento avançado. Ele também pode ser usado para alguns testes funcionais. / / / / /

O JMeter suporta parametrização de variáveis, asserções (validação de resposta), cookies por segmento, variáveis de configuração e uma variedade de relatórios.

A arquitetura JMeter é baseada em plugins. A maioria dos recursos são implementados como plugins. Os usuários da ferramenta podem facilmente estender o JMeter desenvolvendo plugins personalizados.

4 AVALIAÇÃO COMPARATIVA

4.1 PLANO DE TESTES PARA AVALIAÇÃO COMPARATIVA

As análises comparativas a serem aplicadas neste projeto consistem de comparações teóricas, analíticas e performáticas. As análises teóricas abordarão as comparações como vantagens e desvantagens entre as arquiteturas. Os testes analíticos apresentarão a estrutura de diretórios e as diferenças encontradas na estrutura organizacional dos projetos de maneira qualitativa e qualitativamente as linhas de código entre ambos os projetos. A performance obtida através da execução exaustiva entre os dois projetos será apresentada na parte de avaliação final deste capítulo. Por fim será demonstrado um comparativo entre a escalabilidade entre as duas arquiteturas.

Para a realização dos testes, o cenário utilizado foi constituído de máquinas virtuais semelhantes cujas configurações possuiam 2.19 GHz de processamento, 1.96 GiB de memória e 17.9 GiB de capacidade de armazenamento.

4.2 VANTAGENS E DESVANTAGENS

4.2.1 Vantagens da arquitetura em Microsserviços

> Uma arquitetura baseada em Microservice divide sistemas de software em muitos pequenos serviços que podem ser implantados de forma independente. Cada equipe trabalha em seus próprios Microservices e, portanto, está desacoplada de outras equipes. Isso permite escalar facilmente os processos ágeis. A modularização no Microservices protege o sistema contra a decomposição da arquitetura. Conseqüentemente, os sistemas baseados em Microservices permanecem mantidos no longo prazo. Além disso, os sistemas legados podem ser migrados para Microservices sem ter que alterar o código legado. Além disso, a entrega contínua é mais fácil de implementar em sistemas baseados em Microservice. (WOLFF, 2015)

- A aplicação inicia mais rápido, o que torna os desenvolvedores mais produtivos e acelera as implantações.

- Cada serviço pode ser implantado independentemente de outros serviços, portanto, é mais fácil implantar novas versões de serviços com maior freqüência.

- Desenvolvimento mais fácil de dimensionar e também pode ter vantagens de desempenho.

- Elimina qualquer compromisso a longo prazo com uma pilha de tecnologia. Ao desenvolver um novo serviço, você pode escolher uma nova pilha de tecnologia.

- Os serviços são geralmente melhor organizados, uma vez que cada microsserviço possui um trabalho muito específico e não se preocupa com os trabalhos de outros componentes.

- Os serviços desacoplados também são mais fáceis de recompor e reconfigurar para servir os propósitos de diferentes aplicativos (por exemplo, atendendo os clientes da web e a API pública).

/

4.2.2 Desvantagens da arquitetura em Microsserviços

- Os desenvolvedores devem lidar com a complexidade adicional da criação de um sistema distribuído.

- Complexidade de implantação. Na produção, há também a complexidade operacional de implantação e gerenciamento de um sistema composto por muitos tipos de serviços diferentes.

- À medida que você está construindo uma nova arquitetura de microsserviços, é provável que você descubra muitas preocupações transversais que você não antecipou em tempo de design.

4.3 ANALÍTICO DE CÓDIGO

4.3.1 Estrutura de diretórios

Para capturar a estrutura das pastas dos projetos apresentadas nas figuras 4 e 5, foi utilizada a biblioteca Tree, disponibilizada pelo Centro de Computação biológica através do link: http://mama.indstate.edu/users/ice/t

.

4.3.1.1 Árvores dos diretórios

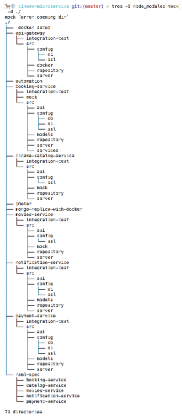

Figura 4 Estrutura de diretórios do código em microsserviços

Figura 5 Estrutura de diretórios do código monolítico

4.3.1.2 Avaliação

Pode-se perceber nas figuras 4 e 5 o aumento da complexidade ocasionado pela arquitetura de microsserviços pela quantidade de diretórios criados para se organizar o projeto. São um total de 71 diretórios em microsserviços, muitos deles se repetindo para cada serviço, contra 17 diretórios no projeto monolítico. Isso acontece devido à necessidade de se criar uma estrutura de diretórios para cada serviço, visto que eles poderão ser desenvolvidos, testados e implantados independentemente, o que confere à arquitetura de microsserviços a independência

de serviços esperada.

4.3.2 Linhas de código

Para efetuar a contagem das linhas de código, conforme apresentado nas figuras 6 e 7, utilizou-se uma ferramenta de código aberto chamada node-sloc, disponibilizada através do link https://www.npmjs.com/package/node-sloc e abstrairemos desta contagem em ambos os casos tanto o diretório contendo bibliotecas de terceiros utilizadas no projeto quanto o repositório *mock*, responsável por criar objetos de testes.

```
cinema-monolithic git:(master) ✗ node-sloc "./" --ignore-pat
hs "node_modules, mock"
Reading file(s)...

+-----------------------------+-----------------+
| SLOC                        | 1073            |
|-----------------------------|-----------------|
| Lines of comments           | 60              |
|-----------------------------|-----------------|
| Blank lines                 | 159             |
|-----------------------------|-----------------|
| Files counted               | 30              |
|-----------------------------|-----------------|
| Total LOC                   | 1133            |
+-----------------------------+-----------------+
```

Figura 6 Contagem de linhas da aplicação monolítica

```
cinema-microservice git:(master) ✗ node-sloc "./" --ignore-p
aths "api-gateway/node_modules, booking-service/node_modules, cin
ema-catalog-service/node_modules, movies-service/node_modules, no
tification-service/node_modules, payment-service/node_modules, ap
i-gateway/src/mock, booking-service/src/mock, cinema-catalog-serv
ice/src/mock, movies-service/src/mock, notification-service/src/m
ock, payment-service/src/mock"
Reading file(s)...

+-----------------------------+-----------------+
| SLOC                        | 3386            |
|-----------------------------|-----------------|
| Lines of comments           | 232             |
|-----------------------------|-----------------|
| Blank lines                 | 595             |
|-----------------------------|-----------------|
| Files counted               | 130             |
|-----------------------------|-----------------|
| Total LOC                   | 3618            |
+-----------------------------+-----------------+
```

Figura 7 Contagem de linhas da aplicação em microsserviços

4.3.2.1 Avaliação

Como a contagem de linhas refere-se tanto aos arquivos de código da aplicação propriamente dita quanto aos arquivos de configuração de ambiente, nota-se que microsserviços necessitou de pelo menos três vezes mais esforço de escrita de código. Devemos também considerar a necessidade da criação do serviço que atua como ponto único de acesso para a utilização de todos os outros serviços, o qual chamamos de "Gateway" e que possui aproximadamente trezentas e trinta linhas, como um segundo fator para a discrepância entre os valores obtidos.

4.4 PERFORMANCE

4.4.1 Teste de carga

Para poder comparar a performance entre ambas arquiteturas serão aplicados conjuntos de testes de carga (Figura 8) formados por requisições assíncronas através da ferramenta JMeter. As requisições aplicadas às duas implementações do sistema serão idênticas, contendo o mesmo número de *threads* para envio de requisições, de modo que possamos avaliar o comportamento das aplicações em iguais condições.

Dentro deste conjunto de testes, teremos uma primeira bateria de testes responsáveis por buscar as agendas dos cinemas por cidade e identificadores únicos do filme (*Fetch schedules by city and movieId*). Em seguida um outra bateria de testes é realizada para buscar todos os cinemas de uma determinada cidade (*Fetch cinemas by city*). Outra sequência de testes trará os filmes de um determinado cinema através do seu identificador único. A seguir, uma tarefa de teste dará uma resposta de todos os filmes cadastrados e outra mostrará apenas os que possuem a propriedade "premiere". A tarefa subsequente buscará os detalhes de um filme específico baseado no seu identificador único. Por fim, os dois últimos grupos de testes serão responsáveis por realizar a reserva, pagamento e notificação dos usuários e também retornar as reservas realizadas por um determinado usuário através do número do pedido.

Durante a execução dos testes foram criadas 1000 *threads* em cada teste pelo JMeter, a fim de estressar os serviços e assim conseguir obter métricas comparativas como tempo médio de resposta e a porcentagem de erros de requisição, bem como um gráfico da distribuição do

tempo de resposta entre os serviços que pode ser observado nas figuras 9 e 10.

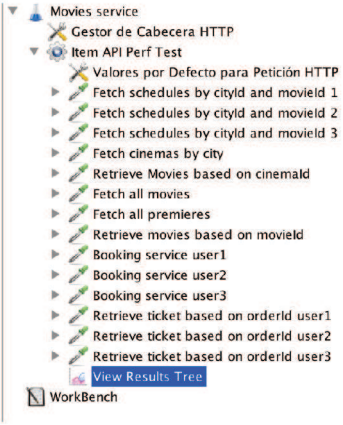

Figura 8 Plano geral de testes criados no JMeter

4.4.1.1 Monolítico

Ao executar a bateria de testes de carga no sistema monolítico foi observado que a média total do tempo das requisições realizadas ficou em 2,659 segundos, sendo a mediana de 1,158 segundos. Já a porcentagem de erro total na execução dos processos ficou em 0.19% e a mediana to tempo de resposta para as mesmas requisições foi de 1,158 segundos.

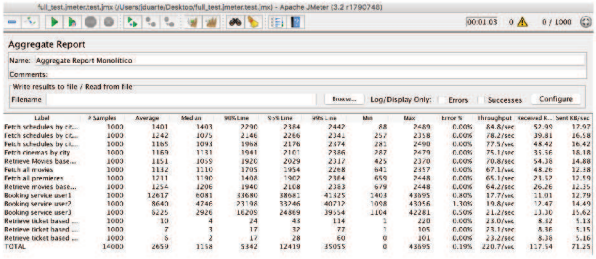

Figura 9 Aggregate Report Monolítico

4.4.1.2 Microsserviços

Na execução dos testes utilizando microsserviços distribuídos obtivemos o resultado da média total do tempo das requisições realizadas em 11,900 segundos. A mediana obtida para as mesmas requisições foi de 5,867 segundos enquanto a porcentagem de erro total ficou estimada em 0.21%.

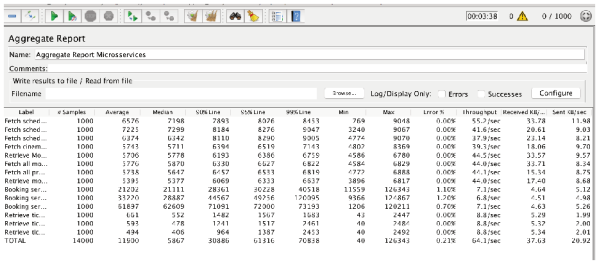

Figura 10 Aggregate Report Microsservices

4.4.1.3 Avaliação

A comparação entre os resultados pode ser justificada através da observação da alta taxa de tempo de resposta para as requisições realizadas entre os serviços dentro da arquitetura de microsserviços, pois para realizar a comunicação entre os serviços, requisições HTTP internas são realizadas, resultando em um maior tempo de comunicação entre os processos. Acreditamos que essa diferença no desempenho das duas implementações poderia ser reduzida se, ao invés de requisições HTTP, fosse utiizado um *middleware* de mensagens para comunicação entre os microsserviços.

4.5 INFRAESTRUTURA

Nesta seção iremos abordar a infraestrutura por trás de cada projeto e as premissas utilizadas para a execução dos testes de carga utilizados para mensurar a performance das aplicações. Para que não haja qualquer tipo de interferência nos resultados do teste de performance quanto à escrita e leitura no banco de dados, ficou determinada

a utilização do mesmo banco NoSQL, o MongoDB, para as duas aplicações.

A infraestrutura adotada para a execução da aplicação monolítica requer uma baixa complexidade, sendo esta responsável por gerir um único processo a qual iniciará os scripts NodeJs e manterá a comunicação das requisições através do framework ExpressJs.

Quanto à infraestrutura utilizada para gerir este projeto, foram adotadas algumas ferramentas para facilitar e abstrair a criação de maquinas virtuais com, por exemplo, Docker Machine, ambiente clusterizado com Docker Swarm, serviços escalonáveis através do Docker Service, conteinerização via Docker e gerenciamento da infraestrutura utilizando a plataforma Rancher apresentada na Figura 11.

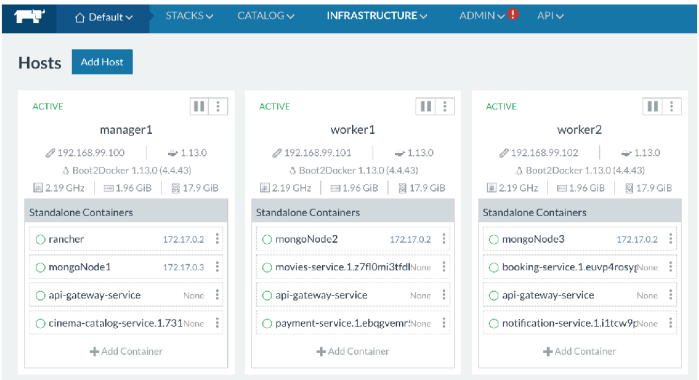

Figura 11 Rancher Web Interface

4.5.1 Escalabilidade

Com o propósito de demonstrar a escalabilidade do sistema, iremos escalar apenas os serviços de "Booking" para demonstrar como um serviço pode ser implantado independentemente do restante do projeto conforme pode ser observado na Figura 12.

Podemos perceber, olhando para a interface do Rancher na figura 12 que dois novos serviços de "Booking" foram escalados, ou seja, cada maquina virtual "Docker Machine" agora possui um contêiner com este serviço implantado.

No caso de uma aplicação orientada a microsserviços, toda a

aplicação deveria ser replicada em uma nova máquina.

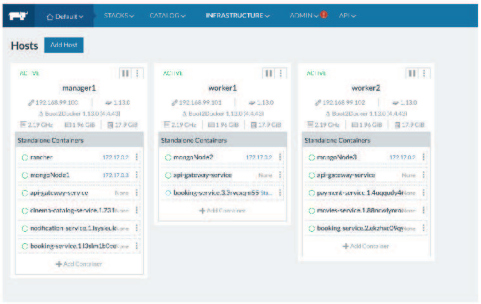

Figura 12 Escalonamento dos serviços de Booking

5 CONCLUSÃO

Para concluir os resultados obtidos a partir deste trabalho serão comentados os pontos fortes e fracos de cada arquitetura, bem como as avaliações obtidas. Na maior parte deste trabalho foram abordados assuntos sobre a concepção de uma aplicação, seja ela monolítica ou em microsserviços. Porém, não podemos deixar de lembrar que o ciclo de vida de uma aplicação em um ambiente real vai muito além da sua construção propriamente dita. A arquitetura adotada para a concepção do projeto acabará por impactar em boa parte deste ciclo, desde as regras de negócios a serem suportadas, passando pelos testes de integração, implantação e principalmente a manutenção.

A partir destas considerações pode-se dizer que a arquitetura em microserviços destaca-se em grandes aplicações onde seus benefícios são facilmente identificados. O tempo de iniciação de um projeto é muito mais rápido, visto que cada serviço é independente. Isto faz com que os desenvolvedores ganhem maior agilidade, tanto no desenvolvimento quanto nas implantações. Com a mesma premissa de independência dos serviços, as liberações de versões dos serviços tendem a ser mais frequentes, e também é proporcinada aos projetistas uma maior precisão no dimensionamento de projetos e equipes.

Com módulos menores e independentes, a facilidade de trocar a pilha tecnológica composta em um serviço ou construir um novo serviço utilizando outra pilha tecnológica também podem ser identificadas como benefícios alcançados por esta arquitetura. A independência da linguagem utilizada em cada serviço também pode aferir ao projeto uma melhor performance, o que não foi observado nas análises e testes aplicados neste trabalho. Alguns especialistas também sugerem que o fato de que na composição de cada serviço tenha apenas funcionalidades específicas para o interesse do próprio serviço possa trazer mais organização ao projeto, embora muitos serviços possam apresentar funções redundantes.

Serviços de baixo acoplamento também são mais fáceis para serem reaproveitados e reconfigurados para servir propósitos de diferentes soluções, por exemplo, atendendo uma aplicação web e fornecendo uma API pública.

Quanto à arquitetura monolítica, evidencia-se a vantagem de abstrair preocupações relacionadas ao reconhecimento e registro de novos serviços, visto que todos os serviços estarão dentro de um mesmo monólito, a limitação dos recursos de segurança, trilhas de auditoria,

proteção contra ataques de negação de serviço (DOS), e a conexão e comunicação entre os componentes da aplicação. Como vimos na aplicação utilizada para fins de comparação, também podem haver vantagens de desempenho, já que o acesso à memória é mais rápido do que a comunicação entre processos.

/

5.1 TRABALHOS FUTUROS

A respeito dos trabalhos futuros relacionados a este tema, pode-se abordar temas em que outras arquiteturas de softwares sejam desenvolvidas para traçar os respectivos demonstrativos e comparativos. Outra abordagem mais detalhista seria aprofundar-se no desenvolvimento de mais testes comparativos e também na especialização de uma das arquiteturas trabalhadas neste projeto. Além destes temas diretamente relacionados ao tema fundamental do projeto, este trabalho também abre portas para o desenvolvimento de aplicações distribuídas, utilização de contêineres, ferramentas de automação e gerenciamento de aplicações desenvolvidas utilizando-se microsserviços, entre outros.

REFERÊNCIAS

APACHE. *JMeter*. <http://jmeter.apache.org/>. Acessado em 03/06/2017.

DOCKER. *What is Docker*. <https://www.docker.com/>. Acessado em 13/05/2017.

FOWLER, M. *MonolithFirst*. Junho 2010. <https://martinfowler.com/bliki/MonolithFirst.html>. Acessado em 8 mar. 2017.

FOWLER, M. *Microservices, a definition of this new architectural term*. Junho 2015. <https://martinfowler.com/articles/microservices.html>. Acessado em 13/07/2017.

NEWMAN, S. *Building Microservice*. [S.l.]: OReilly, 2015. 473 p.

NODEJS. *NodeJs*. <https://nodejs.org/en/>. Acessado em 03/06/2017.

RAMIREZ, C. *Build a NodeJS cinema microservice and deploying it with docker part 1*. 2017. <https://medium.com/@cramirez92/build-a-nodejs-cinema-microservice-and-deploying-it-with-docker-part-1-7e28e25bfa8b>. Acessado em 07/08/2017.

RAMIREZ, C. *Deploy Nodejs microservices to a Docker Swarm Cluster [Docker from zero to hero]*. 2017. <https://towardsdatascience.com/deploy-a-nodejs-microservices-to-a-docker-swarm-cluster-docker-from-zero-to-hero-464fa1369ea0>. Acessado em 13/05/2017.

RANCHER. *Rancher*. <https://rancher.com/>. Acessado em 03/06/2017.

STOIBER, M. *Build your first Node.js microservice*. <https://mxstbr.blog/2017/01/your-first-node-microservice/>. Acessado em 11/06/2017.

WOLFF, E. *Flexible Software Architectures*. [S.l.]: Leanpub, 2015. 324 p.

APÊNDICE A – Artigo