Centro Universitário Nossa Senhora do Patrocínio

TRABALHO DE CONCLUSÃO DO CURSO DE CIÊNCIAS DA COMPUTAÇÃO

ESTUDO COMPARATIVO ENTRE A ARQUITETURA MONOLÍTICA E DE MICROSSERVIÇOS

Centro Universitário Nossa Senhora do Patrocínio

ESTUDO COMPARATIVO ENTRE A ARQUITETURA MONOLÍTICA E DE MICROSSERVIÇOS

Johnny Rezende
Vinícius de Sordi

Salto – SP
2020

ESTUDO COMPARATIVO ENTRE A ARQUITETURA MONOLÍTICA E DE MICROSSERVIÇOS

Projeto apresentado na disciplina de Tendências em Ciências da Computação como requisito básico para a apresentação do Trabalho de Conclusão do Curso de Ciência da Computação

Orientador: Prof. José Luis Pagotto

Indaiatuba - SP
2020

JOHNNY REZENDE

VINICIUS DE SORDI

ESTUDO COMPARATIVO ENTRE A ARQUITETURA MONOLÍTICA E DE MICROSSERVIÇOS

Trabalho de Conclusão de Curso apresentada em 2020 do Centro Universitário Nossa Senhora do Patrocínio como requisito à obtenção do título de Bacharel em Ciência da Computação.

.

Aprovada em: ___/___/_____.

BANCA EXAMINADORA

_____

Prof. Título. Xxxxxxxxx Xxxxxxx

Faculdade

_____

Prof. Título Xxxxxxxxx Xxxxxxx

Faculdade

_____

Prof. Título. Xxxxxxxxx Xxxxxxx

Faculdade

RESUMO

Com o propósito de aplicar os conhecimentos didáticos obtidos durante o curso de Ciência da Computação, enriquecer a fonte de conhecimento e pesquisa em relação ao domínio das arquiteturas para desenvolvimento de software atuais, bem como apresentar testes técnicos mais expressivos e reais à respeito das arquiteturas de microsserviços e monolítica, este trabalho de conclusão de curso tem por objetivo apresentar um estudo entre as duas arquiteturas de desenvolvimento através de uma avaliação comparativa entre o desenvolvimento de uma aplicação monolítica de um sistema de lista de tarefas, desenvolvido previamente durante o curso, escrito utilizando as tecnologias *JavaScript* e *Node.JS* e uma outra aplicação idêntica construída com a arquitetura de microsserviços. Arquiteturas de desenvolvimento de software como microsserviços ganham cada vez mais destaque no mercado de tecnologia, e por isso, é interessante traçar um comparativo evidenciando as vantagens e desvantagens entre a arquitetura de desenvolvimento mais comum, a monolítica, e a arquitetura de microsserviços. Para estabelecer uma comparação justa e didática, foram desenvolvidos e analisados dois protótipos idênticos em termos de funcionalidades, um orientado a arquitetura monolítica e outro orientado a arquitetura de microsserviços, ambos escritos em JavaScript, sustentados sobre a plataforma Node.JS, utilizando um banco de dados relacional. A construção do protótipo foi baseada em um projeto já existente desenvolvido previamente durante o curso. Ambas as aplicações foram submetidas a critérios de testes de performance, quantidade de código escrito e análise estrutural para tornar a aplicação operante. E a partir dos resultados obtidos, foram apresentados e comentados suas características comparativas para facilitar a compreensão destes resultados. Levando em consideração as características únicas de cada aplicação, as tecnologias envolvidas, a análise e os testes aplicados, este trabalho, ao fazer um estudo comparativo entre essas duas arquiteturas, pode auxiliar, as organizações em sua tomada de decisão sobre qual arquitetura utilizar ao desenvolver uma nova aplicação, assim como estudantes interessados em ter um melhor compreendimento de sua aplicabilidade. Por conseguinte, o resultado obtido com os testes e análises

realizados neste estudo, foi observado que a arquitetura monolítica, em um projeto inicial, pode sim ter um melhor desempenho perante a arquitetura de microsserviços.

Palavras-Chave: Arquitetura de software. Padrões de desenvolvimento de software. Monolítico. Microsserviços.

LISTA DE ILUSTRAÇÕES

SUMÁRIO

1 INTRODUÇÃO

Nos últimos anos, uma nova arquitetura de desenvolvimento de software vem se tornando bastante popular no mundo da tecnologia. A palavra-chave "*microservices*" (microsserviços) teve um aumento de 9 pontos, em janeiro de 2015, para 81 pontos de interesse em maio de 2020 (GOOGLE TRENDS, 2020). Esta arquitetura surgiu com a finalidade de viabilizar o desenvolvimento de software otimizando a escalabilidade, implantação e melhorando a gestão de equipes de desenvolvimento as especializando em soluções específicas. Esta arquitetura possibilita um desenvolvimento mais ágil, com implantação facilitada, fornecendo a possibilidade de utilizar múltiplas linguagens de programação simultaneamente e proporcionando melhor adaptação para as aplicações em nuvem.

Mas para entender melhor o padrão de arquitetura orientado a microsserviços, é preciso entender primeiramente a arquitetura utilizada até então para a construção de aplicações, a arquitetura monolítica.

> Quase todas as histórias bem sucedidas de microsserviços começaram com um monólito que ficou muito grande e acabou sendo quebrado (FOWLER, 2010)

Arquitetura monolítica é um modelo de desenvolvimento de software onde toda a aplicação faz parte de uma mesma estrutura e é executada uma só vez. Ou seja, em uma aplicação monolítica, todas as funções e módulos do sistema são implementados em um único processo indivisível. O grande problema de uma aplicação monolítica é que ao longo do tempo a aplicação vai crescendo e consumindo mais recursos, assim, surgem muitos desafios de manutenção, tais como: limitação de escalabilidade, falta de agilidade para colocar novas alterações em produção e uma alta dependência dos componentes da aplicação. "Quando é preciso fazer uma mudança, é necessário mexer em toda a estrutura. Dependendo do grau de desenvolvimento, isso significa gastar um longo período." (SANTODIGITAL, 2019).

1.2 JUSTIFICATIVA

Hoje é possível encontrar na internet diversas linhas de pensamento sobre qual arquitetura de desenvolvimento utilizar dentro da comunidade, bem como suas características, vantagens e desvantagens. Todavia, mesmo com muito conhecimento teórico disponível na rede, ainda há dificuldade em se obter com clareza análises e comparações técnicas referente as duas arquiteturas citadas acima, bem como exemplos e testes práticos referente a performance.

1.3 OBJETIVO

Com o propósito de enriquecer a fonte de conhecimento e pesquisa em relação ao domínio das arquiteturas para desenvolvimento de software atuais, bem como apresentar testes técnicos mais expressivos e reais à respeito das arquiteturas de microsserviços e monolítica, este trabalho de conclusão de curso tem por objetivo apresentar um estudo entre as duas arquiteturas de desenvolvimento através de uma avaliação comparativa entre o desenvolvimento de uma aplicação monolítica de um sistema de lista de tarefas, desenvolvido previamente durante o curso, escrito utilizando as tecnologias *JavaScript* e *Node.JS* e uma outra aplicação idêntica construída com a arquitetura de microsserviços. Com este estudo, será possível visualizar em cenário real, as vantagens e desvantagens que cada arquitetura apresenta, bem como análises de performance e dificuldades de implantação

1.4 OBJETIVOS ESPECÍFICOS

Como objetivo específico, este projeto propõe demonstrar a comparação técnica entre as duas arquiteturas de desenvolvimento, apresentando testes de desempenho, análises de implantação e estrutura de projeto. Tendo como resultado, duas aplicações com funcionalidades equivalentes mas com propósitos diferentes, para que assim, sirva a comunidade como material de estudo e como guia para auxiliar no desenvolvimento de novas aplicações.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 ARQUITETURA MONOLÍTICA

Em uma arquitetura monolítica, todos os processos do negócio estão implementados em uma única aplicação, onde a interface de usuário e código de acesso aos dados são combinados em um único programa a partir de uma única plataforma. Ou seja, toda a aplicação é uma única base de código que é compilada e executada uma única vez na mesma máquina, consumindo os mesmos recursos de processamento, memória, banco de dados e arquivos. Com toda essa centralização, o nível de complexidade da coordenação é reduzido. O ambiente de um sistema monolítico é mais simples de configurar e entender, possui uma curva de aprendizado menor e, obviamente, existem mais desenvolvedores familiarizados com este modelo, o que é ideal para iniciar um *MVP (minimum viable product - mínimo produto viável)* para validar um negócio ou produto. Todavia, desenvolver nesta arquitetura implica em acrescentar itens sempre ao mesmo bloco, aumentando cada vez mais seu tamanho e complexidade, tornando sua manutenção cada vez mais cara e mais lenta, visto que existe uma alta dependência de componentes de código.

2.2 ARQUITETURA DE MICROSSERVIÇOS

Caracterizada pela descentralização de linguagens, automação de processos de implantação e organização de projetos e equipes, a arquitetura de microsserviços surgiu com a finalidade de viabilizar o desenvolvimento de software otimizando a implantação de aplicações modulares a partir da divisão de uma aplicação em pequenas unidades independentes.

> Um microsserviço é uma unidade de software autônoma que, juntamente com 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, 2017)

Este modelo visa criar serviços independentes, porém interligados. Desta forma, cada um dos módulos funcionam dentro do seu próprio processo, com seu banco de dados independente e utilizando a tecnologia apropriada para a sua aplicação. Esta arquitetura permite o desenvolvimento de acordo com a função, bem-delimitada e com um propósito específico. Se fosse possível resumir a arquitetura de microsserviços em uma palavra, seria especialização. Cada serviço tem uma implementação diferente, sendo assim, subir uma nova atualização em um serviço não atrapalha a disponibilidade dos outros. Assim como é possível escalar um único serviço que demande mais poder de processamento de forma independente do restante do sistema. Por ser uma estrutura mais complexa, esta arquitetura demanda um nível mais elevado de automação das implementações. Além disso, ter uma boa integração entre todos os serviços é essencial para que tudo funcione corretamente. A complexidade exige desenvolvedores com qualificação maior ou, ao menos, uma boa coordenação de *DevOps* (Desenvolvimento e Operações) para assegurar um bom funcionamento.

3 PLANEJAMENTO DO EXPERIMENTO

Para desenvolver os projetos que compõem este trabalho, foi utilizado como modelo um projeto de um Sistema de Lista de Tarefas, desenvolvido previamente durante o curso, no qual é utilizado como base a arquitetura monolítica 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, alterando apenas a sua arquitetura para torná-lo uma aplicação em microsserviços dotado das mesmas funcionalidades. Estes projetos, após implementados, foram comparados em termos de métodos de implantação, performance, escrita do código e infraestrutura. Ao final do experimento será apresentado um resultado conclusivo sobre as duas arquiteturas abordadas.

A aplicação de exemplo é composta por quatro funcionalidades/serviços. O serviço de “Task Scheduler”, responsável por realizar o registro de tarefas. Este serviço estará integrado com o serviço de notificação, chamado de “Notification Service”, que será o responsável por enviar ao usuário final notificações como confirmação de registro de tarefa e lembretes de tarefas que estão pendentes. O serviço “Default Tasks”, que irá fornecer ao usuário acesso rápido a um catálogo de tarefas padrão. E por fim, o serviço de lembrete de tarefas, o “Task Reminder”, encarregado de definir lembretes ao usuário referente a tarefas que precisam ser realizadas. Este serviço, por sua vez, estará vinculado aos serviços de notificação, “Notification Service” e ao serviço de agendamento de tarefas, o “Task Scheduler”.

3.1 PROJETO MONOLÍTICO

Na aplicação monolítica, os serviços do projeto estão todos agrupados dentro da mesma aplicação, seus pontos de chamada estão agrupados dentro de um mesmo arquivo de *endpoints* (terminal de comunicação). Quando algum ponto de chamada é acionado por uma requisição http, o terminal de comunicação aciona a funcionalidade requisitada e devolve a resposta para o usuário.

Algoritmo - Trecho do arquivo de funcionalidades do projeto monolítico

```
/**
* Endpoint para retornar todas as tarefas cadastradas
*
* @param Request
* @param Response
*
* @return json
*/
async index (Request, Response)
{
    const tasks =
        await knex('TASKS')
        .select('TASKS.*');

    const serializedTasks = tasks.map(task => {
        return {
            ID TASK TAS: task.ID TASK TAS,
            ST TASK TAS: task.ST TASK TAS,
            ST STATUS TAS: task.ST STATUS TAS,
            DT TASK TAS: format(task.DT TASK TAS, "dd'/'MM'/'yyy HH':'mm")
        }
    });

    return Response.status(200).json(serializedTasks);
}

/**
 * Endpoint para criar uma tarefa
 *
 * @param Request
 * @param response
 *
 * @return success
 * @throws error
 */
async create (Request, Response)
{
    const  {ST TASK TAS, ST STATUS TAS, DT TASK TAS} = Request.body;

    const transaction = await knex.transaction();

    const taskDateTime = startOfMinute(parseISO(DT TASK TAS));

    if (isPast(taskDateTime)) {
```

```
        await transaction.rollback();

        return Response
            .status(400)
            .json({error: "Não é possivel inserir uma tarefa para o
passado!"});
    }

    const task = {
        ST TASK TAS,
        ST STATUS TAS: ST STATUS TAS.toUpperCase(),
        DT TASK TAS: taskDateTime
    };

    const taskExists =
        await transaction('TASKS')
        .where('ST TASK TAS', ST TASK TAS)
        .andWhere('DT TASK TAS', task.DT TASK TAS)
        .andWhere('ST STATUS TAS', task.ST STATUS TAS)
        .first();

    if (taskExists) {
        await transaction.rollback();
        return Response
            .status(400)
            .json({error: 'Tarefa já criado!'});
    };

    const taskCreated = await transaction("TASKS").insert(task);

    await transaction.commit();

    Queue.add('scheduleReminder', {method: 'remind',task:`lembrete:
${ST TASK TAS}`, date: DT TASK TAS});

    return Response
        .status(200)
        .json(`Tarefa#${taskCreated} criado com sucesso!`)
}

/**
 * Endpoint para remover uma tarefa
 *
 * @param Request
 * @param Response
 *
 * @return success
 * @throws error
 */
async remove (Request, Response)
{
    const {ID TASK TAS} = Request.body;

    const reminder = await knex('TASKS').where('ID TASK TAS',
ID TASK TAS).del();

    if (reminder == 0) {
        return Response
            .status(200)
            .json('Tarefa já deletada!');
    }
```

```
    return Response
        .status(200)
        .json('Tarefa deletada!');
}
```

3.2 PROJETO EM MICROSSERVIÇOS

Na arquitetura em microsserviços, o projeto inteiro foi construído de tal forma que cada serviço seja uma única aplicação sendo executada de forma independente da outra. Ou seja, cada aplicação é responsável pelo seu próprio ciclo de vida desde a concepção até a sua implantação. Sendo possível utilizar uma linguagem de programação que seja mais apropriada para o propósito do serviço. Para estabelecer a troca de informações entre os serviços, foi necessário criar um serviço responsável por capturar as ações realizadas pelos serviços e transmiti-las para os outros serviços em forma de eventos, esse serviço foi chamado de “Event bus”. Este serviço funciona da seguinte forma: O usuário acessa o serviço Task schedule e cria uma nova tarefa com lembrete marcado para uma data específica. Este serviço registra essa tarefa e emite um evento de tarefa criada para o event bus, que irá transmitir esse evento para todos os outros serviços. Dessa forma, o serviço Task Reminder receberá a notificação desse evento transmitido pelo Event bus, e irá agendar o lembrete desta tarefa. Por conseguinte, quando chegar na data em que a tarefa precisa ser notificada, este serviço envia um evento para o Event bus, dizendo que a tarefa precisa ser notificada para o usuário final, e o event bus encaminha esse evento para o serviço de notificação, o Notification service, que por fim, notificará o usuário final que a tarefa está quase vencendo.

Algoritmo - Trecho do arquivo responsável por transmitir os eventos para todos os outros serviços.

```
// Recebendo eventos
app.post('/events', (req, res) => {
  const event = req.body;
  console.log("evento recebido no event bus!")
  //Disparando events
  axios.post(
      `${api.API URL QUERY SERVICE}/events`,
      event
      ).catch(function (error) {
          console.log(error.message)
  });
```

```
  axios.post(`${api.API URL DEFAULT TASKS SERVICE}/events`, event)
       .catch(function (error) {
          console.log(error.message)
  });

  axios.post(`${api.API URL NOTIFICATION SERVICE}/events`, event)
       .catch(function (error) {
          console.log(error.message)
  });

  axios.post(`${api.API URL TASK REMINDER SERVICE}/events`, event)
       .catch(function (error) {
          console.log(error.message)
  });

  axios.post(`${api.API URL TASK SCHEDULE SERVICE}/events`, event)
       .catch(function (error) {
          console.log(error.message)
  });

  res.json("Eventos transmitidos.")
});
```

Cada serviço possui seu próprio banco de dados, porém, os dados de cada serviço são persistidos em um serviço de armazenamento, chamado de Query service. Esse serviço se fez necessário pois caso uma interface deseje consumir dados de dois serviços que estão relacionados (como por exemplo consultar as tarefas cadastradas junto com o seu status de notificação), não será preciso estabelecer uma comunicação com os dois serviços para obter tal informação. A vantagem de ter os dados de todos os serviços persistidos em uma base de dados centralizada é que se caso algum serviço deixe de funcionar, os dados ainda estarão disponíveis para visualização do usuário final. Além de funcionar também como uma espécie de *cache*, que é uma área de armazenamento onde dados ou processos frequentemente utilizados são guardados para um acesso futuro mais rápido.

Para atingir um cenário mais próximo de um cenário de produção, e para que seja mais simples lidar com a implantação destes serviços, foi utilizado a ferramenta *Docker*, que irá fornecer um ambiente de computação responsável por implantar os serviços da aplicação. E para realizar a orquestração, gerenciamento e implantação desses ambientes, foi utilizado o *Kubernetes*, que também tornará a comunicação dos serviços desses ambientes mais fácil. Assim como também é responsável por escalonar esses serviços, caso se faça necessário.

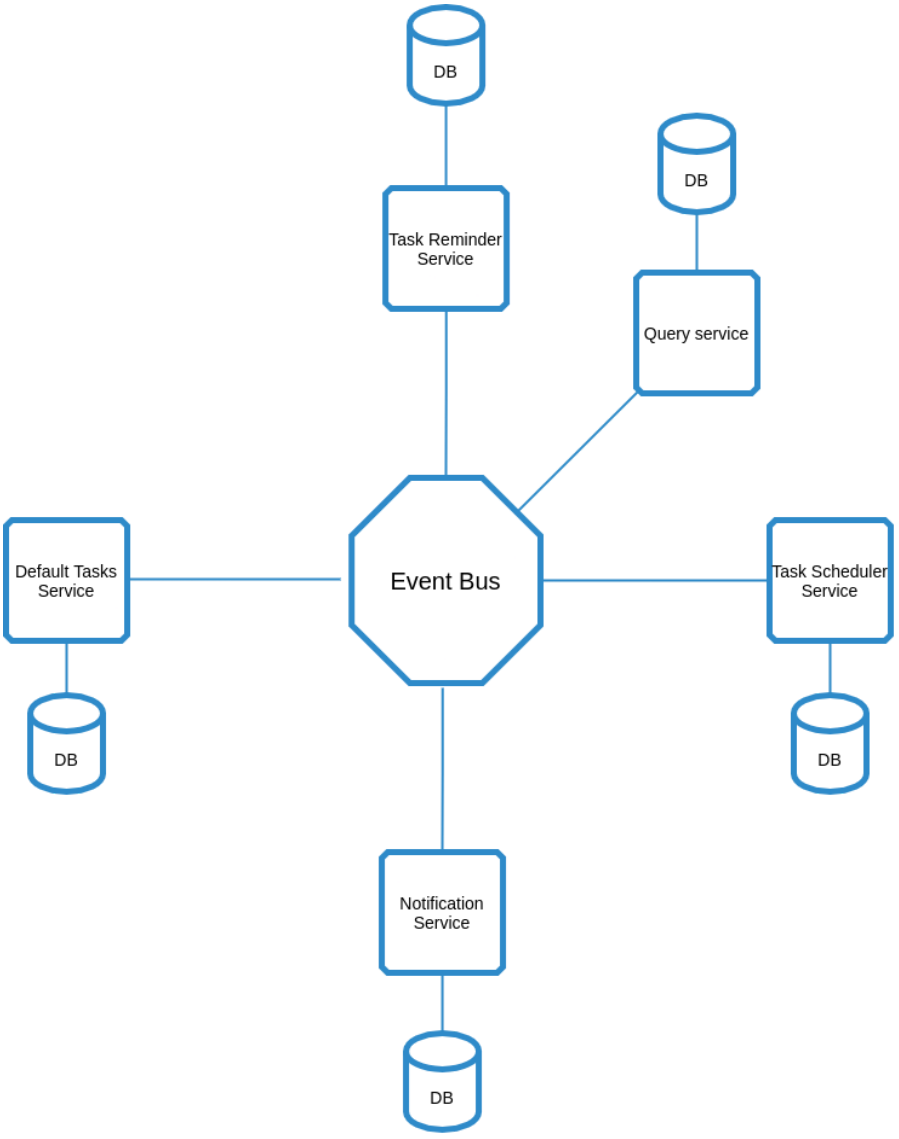


Figura 1 - Design da arquitetura do projeto em microsserviços.

Fronte (própria, 2020)

3.3 FERRAMENTAS E TECNOLOGIAS DE DESENVOLVIMENTO

3.3.1 Node.Js

Node.js é uma plataforma para a construção de aplicativos web escalonáveis e de alto desempenho utilizando o JavaScript. Essa plataforma foi construída em cima do motor V8 (JavaScript V8 Engine, 2008), criado pela Google e utilizado em seu navegador, o Google Chrome. Essa plataforma permite utilizar a linguagem pelo lado do servidor (server-side).

O Node.js trabalha com uma arquitetura voltada a eventos. Ou seja, utilizando um loop de eventos, o Node.js interpreta solicitações assíncronas em um único nó de filamento (*thread*), em vez de solicitações sequenciais, e não permite o bloqueio. Isso o torna incrivelmente rápido e muito adequado para lidar com um grande número de solicitações.

3.3.2 Docker

O Docker é uma plataforma de código aberto, que desembaraça a geração e administração de ambientes únicos. Com a utilização do Docker, é possível realizar a geração de contêineres, que são uma virtualização de um sistema operacional Windows ou Linux, onde é possível implantar uma aplicação, instalar todas as suas dependências e executá-las em um processo separado. Transformando a aplicação em uma ferramenta portátil para qualquer outra máquina, seja ela para desenvolvimento ou ambiente de servidor que contenha o Docker devidamente instalado. Sendo assim, é criar, copiar, implantar e migrar de um ambiente para outro com maior facilidade.

Assim que criado um contêiner, a aplicação será transformada em uma imagem Docker, podendo ser utilizado em qualquer ambiente de preferência do usuário, isso significa que pode essa imagem pode ser utilizada em um ambiente de desenvolvimento da mesma forma que em um ambiente de produção.

As imagens Docker são construídas através de arquivos de definição, contudo, é possível garantir que um certo padrão seja seguido e então aumentar a confiança em sua replicação, então, basta que as imagens sigam as melhores práticas de construção para que seja melhor escalar a estrutura rapidamente.

3.3.3 Kubernetes

Kubernetes (ou K8s, como é chamado pela comunidade) é uma ferramenta de código aberto responsável por gerenciar e orquestrar contêiner. A ferramenta automatiza o dimensionamento, gestão de aplicações e implementação de contêineres, retirando grande parte dos processos manuais indispensáveis para implementar e escalar as aplicações em contêineres. O Kubernetes é formado por

uma sucessão de elementos, cada um com um propósito diferente. Para afirmar que exista uma divisão de responsabilidades e que o sistema seja resiliente, o Kubernetes utiliza um *cluster* (conjunto de máquinas usado para executar aplicações em container) para ser executado sendo dividido em diversas formas:

A primeira forma é chamada de Node ou Nó, que são máquinas que realizam as tarefas solicitadas e atribuídas, controladas pela máquina master do Kubernetes. O propósito de um Node é executar os contêineres que encapsulam as aplicações sendo orquestradas pelo Kubernetes.

A segunda é chamada de Node master, responsável pelos principais componentes do Kubernetes que são executados, como o Scheduler, ferramenta que tem a missão de controlar a alocação de recursos no cluster.

Com o Kubernetes, é possível facilitar o desenvolvimento e estabilização de aplicações em produção, como também o monitoramento e gerenciamento de todas aplicações.

3.3.4 Express

Express é um *framework* (pacote de códigos prontos que podem ser utilizados no desenvolvimento de sites) do Node.js de código aberto que permite estruturar aplicações web ou mobile para lidar com várias solicitações http diferentes em uma url específica. Foi projetado para facilitar o desenvolvimento de aplicações em Node.js facilitando a resposta a solicitações com suporte de rota para que seja possível escrever respostas para URLs específicas.

3.3.5 JMeter

Como descrito em sua própria documentação, “o Apache JMeter™ é um software de código aberto, um aplicativo 100% em Java puro, projetado para testar o comportamento funcional de carga e medir o desempenho. Ele foi originalmente projetado para testar aplicativos da Web, mas desde então foi expandido para outras funções de teste” (APACHE, 2020). Com esta ferramenta, é possível medir a performance de aplicações através de testes unitários unitários, testes de carga e processos nativos do sistema. Com o JMeter, é possível realizar testes em diversos

tipos de aplicações, servidores e protocolos, como aplicações web através de requisições HTTP, conexão com bancos de dados, servidores de envio de email, entre outros. Essa ferramenta suporta parametrização de variáveis, asserções (validação de resposta), cookies por segmento, variáveis de configuração e uma grande variedade de relatórios para visualização dos resultados obtidos.

4 AVALIAÇÃO COMPARATIVA

4.1 PLANO DE TESTES

Para realizar a análise comparativa destas suas arquiteturas, foram realizadas comparações teóricas, analíticas e de performance. A abordagem teórica considera a comparação de vantagens e desvantagens entre as arquiteturas estudadas. Na comparação analítica, foram examinadas as diferenças na estrutura organizacional dos dois projetos, esforço para desenvolvimento e número de linhas de código. Na comparação performática, os dois projetos foram submetidos a exaustivos e estressantes testes de carga e performance através da ferramenta JMeter. No fim deste capítulo, será apresentado as dificuldades de escalabilidade dessas duas arquiteturas.

Para fins de teste, o cenário utilizado é constituído de uma máquina cujas configurações possuíam memória ram de 16 GB, processador AMD Ryzen 5 3400G de 4 núcleos e frequência de 3.7 GHz, e SSD com capacidade de armazenamento de 240 GB.

4.2 VANTAGENS E DESVANTAGENS DAS ARQUITETURAS

4.2.1 Vantagens da arquitetura monolítica

A palavra que melhor define essa arquitetura é simplicidade. Desenvolver nesta arquitetura é bem fácil, bem como implantar a aplicação e escrever rotinas de testes. Isso porque o projeto e todas as suas funcionalidades são centralizadas em apenas uma aplicação, como se fossem uma única entidade. Sendo assim, realizar a comunicação entre os serviços implementados é bem acessível.

4.2.2 Desvantagens da arquitetura monolítica

Quando uma alteração no código é realizada, é preciso reiniciar novamente toda a aplicação, tornando-a indisponível por um curto período de tempo. se caso

houver uma única linha de código com erro, poderá causar instabilidade em toda aplicação.

Como a aplicação é uma entidade única, todo o projeto foi desenvolvido em cima de uma única linguagem de programação. Isso torna a aplicação inflexível, pois ao desenvolver uma nova funcionalidade, não será possível definir uma outra linguagem de programação mais vantajosa para essa funcionalidade em específico.

Com o crescimento da aplicação, a dificuldade para realizar a sua manutenção se torna cada vez maior. A dificuldade de interpretação do código se torna cada vez mais difícil, assim como a dificuldade de realizar modificações.

4.2.3 Vantagens da arquitetura em microsserviços

Uma vez que desenvolvido, a estrutura de um serviço é separada e independente de qualquer outro serviço. Isso traz mais objetividade e simplifica a responsabilidade de cada serviço desenvolvido. Com isso, realizar testes e manutenções no código se tornam bem mais simples. Outra vantagem é que as equipes de desenvolvimento podem ser divididas por serviço, criando especialistas em cada ponto chave do negócio.

4.2.4 Desvantagens da arquitetura em microsserviços

O desenvolvimento de novos recursos em mais de um serviço deve ser cuidadosamente planejado em todas as equipes. Se os serviços não forem bem documentados a medida que foram desenvolvidos, sua complexidade para interpretação pode se tornar bem alta. Desenvolver em microsserviços pode ser uma tarefa bem complicada, visto que é preciso pensar em como os serviços irão se comunicar, como será realizado o escalonamento destes serviços e em como um serviço deve se comportar quando outro serviço que esteja interligado a este esteja com instabilidades.

4.2.5 Comparação analítica

4.2.5.1 Estrutura de diretórios

Para listar a estrutura de diretórios de ambos os projetos apresentadas nas imagens abaixo, foi utilizada a biblioteca Tree, disponibilizada publicamente pelo usuário Yangshun Tay na plataforma de compartilhamento de softwares de código livre, o Github.

4.2.5.2 Árvore de diretórios

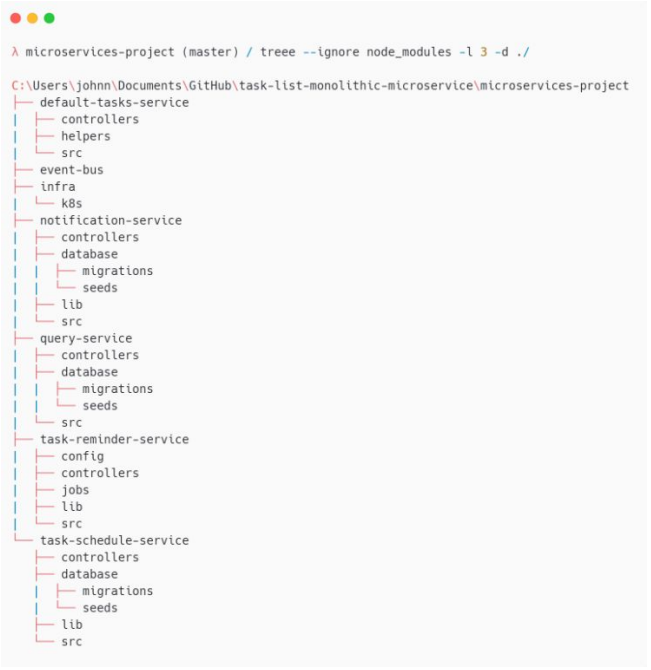


Figura 2 - Estrutura de diretórios do projeto em microsserviços

Fonte: (própria, 2020)

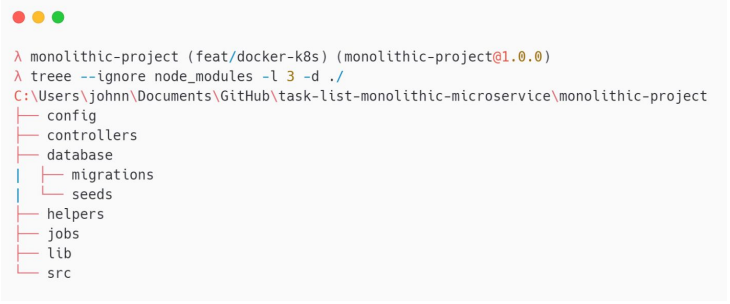


Figura 3 - Estrutura de diretórios do projeto monolítico

Fonte: (própria, 2020)

4.2.5.3 Avaliação

Nas figuras listadas acima, é possível perceber pela quantidade de diretórios criados, o aumento da complexidade causada pela arquitetura de microsserviços. A explicação vem da necessidade de criar um ambiente totalmente independente para cada serviço, visto que eles foram desenvolvidos, testados e implantados de forma independente, como se fossem aplicações distintas que conversam entre si. Muitos desses diretórios são repetidos, pois a configuração base para implantar esses serviços são bem semelhantes. Bem como a configuração do banco de dados de cada serviço. São um total de 33 diretórios na arquitetura de microsserviços, contra apenas 9 diretórios na arquitetura monolítica.

4.2.5.4 Linhas de código

Para obter com assertividade o total de linhas de código escritas, apresentado nas imagens a seguir, foi utilizada a ferramenta de código aberto *Node-sloc*. Também disponibilizada publicamente no plataforma Github e publicada pelo usuário Edvin Havic. Para realizar a contagem em ambas as aplicações, foram ignoradas bibliotecas de terceiros utilizadas no projeto.

Figura 4 - Contagem de linhas do projeto em microsserviços

Fonte: (própria, 2020)

Figura 5 - Contagem de linhas do projeto monolítico

Fonte: (própria, 2020)

5.3.2.1 Resultados

A contagem de linhas se refere tanto ao código da regra de negócio dos serviços, quanto aos códigos de configuração do ambiente. Nota-se que o projeto em microsserviços necessitou um esforço de escrita de código três vezes maior do que o projeto monolítico. Essa necessidade discrepante do projeto em microsserviços é explicada por duas necessidades da arquitetura: A necessidade de realizar a configuração completa de ambiente para cada serviço e a necessidade de criar dois novos serviços para realizar a comunicação entre todos os serviços utilizados, o Event bus e o Query service.

4.2.6 Análise performática

4.2.6.1 Testes de carga e estresse

Para se obter uma comparação de performance das duas arquiteturas, foram aplicados conjuntos de testes de estresse e de carga simulando o acesso e requisições de usuários reais de forma assíncrona através da ferramenta JMeter. O plano de teste aplicado é idêntico para ambas as arquiteturas, contendo o mesmo número de usuários fictícios para envio das requisições, de forma que seja possível

obter uma avaliação mais assertiva sobre o comportamento das aplicações em iguais condições.

No plano de testes definido, foram criados pelo JMeter mil usuários fictícios para enviar requisições para as aplicações de forma assíncrona, com o objetivo de estressar as aplicações para se obter métricas comparativas, como tempo médio de resposta e a porcentagem de erro de cada requisição, assim como uma tabela listando a distribuição do tempo de resposta entre cada serviço que pode ser observado nas imagens a seguir.

Dentro desse plano de testes, será executada a primeira bateria de testes irá consultar todas as tarefas já cadastradas com o status de notificação. Em seguida, a outra bateria de testes retornará a lista de tarefas padrões da aplicação. Por fim, a última bateria de testes irá cadastrar uma nova tarefa, cadastrar um lembrete e enviar uma notificação para o usuário final.

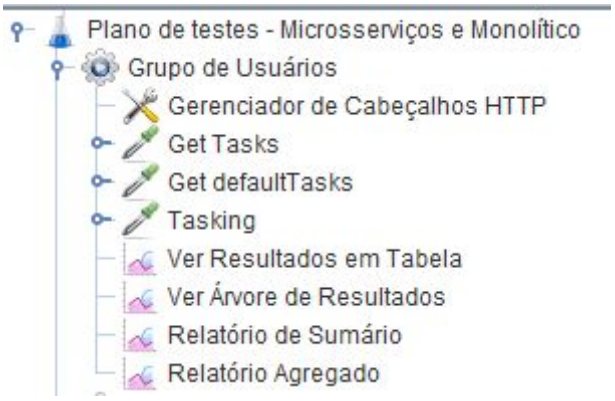
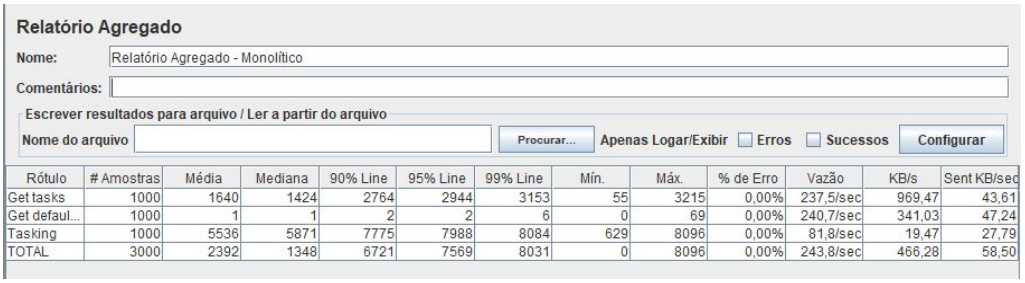


Figura 6 - Plano de testes criado no JMeter

Fonte: (própria, 2020)

4.2.6.2 Performance projeto monolítico

Ao finalizar a execução do plano de testes no projeto monolítico, foi observado que a média total do tempo das requisições realizadas pelos mil usuários fictícios ficou em 2,392 segundos, a mediana ficou em 1,348 segundos e 0% em porcentagem de erro na execução total do plano de testes



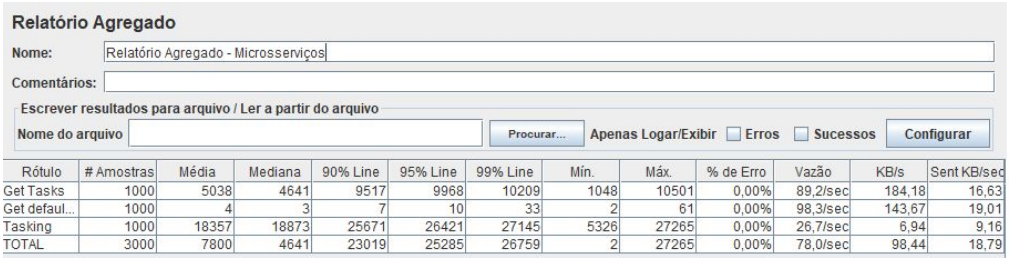
Rótulo	# Amostras	Média	Mediana	90% Line	95% Line	99% Line	Mín.	Máx.	% de Erro	Vazão	KB/s	Sent KB/sec
Get tasks	1000	1640	1424	2764	2944	3153	55	3215	0,00%	237,5/sec	969,47	43,61
Get defaul...	1000	1	1	2	2	6	0	69	0,00%	240,7/sec	341,03	47,24
Tasking	1000	5536	5871	7775	7988	8084	629	8096	0,00%	81,8/sec	19,47	27,79
TOTAL	3000	2392	1348	6721	7569	8031	0	8096	0,00%	243,8/sec	466,28	58,50

Figura 7 - Relatório agregado nos testes do projeto monolítico

Fonte: (própria, 2020)

4.2.6.3 Performance projeto em microsserviços

Os resultados obtidos na execução do plano de testes no projeto em microsserviços foram de 7,8 segundos na média total do tempo das requisições que foram realizadas. Sendo a mediana em 4,641 segundos e tambem 0% de porcentagem de erro para as mesmas requisições.



Rótulo	# Amostras	Média	Mediana	90% Line	95% Line	99% Line	Mín.	Máx.	% de Erro	Vazão	KB/s	Sent KB/sec
Get Tasks	1000	5038	4641	9517	9968	10209	1048	10501	0,00%	89,2/sec	184,18	16,63
Get defaul...	1000	4	3	7	10	33	2	61	0,00%	98,3/sec	143,67	19,01
Tasking	1000	18357	18873	25671	26421	27145	5326	27265	0,00%	26,7/sec	6,94	9,16
TOTAL	3000	7800	4641	23019	25285	26759	2	27265	0,00%	78,0/sec	98,44	18,79

Figura 8 - Relatório agregado nos testes do projeto em microsserviços

Fonte: (própria, 2020)

4.2.6.4 Avaliação

Com base no plano executado, observa-se que o projeto monolítico teve uma menor taxa de tempo de respostas perante ao projeto em microsserviços. Essa diferença pode ser justificada pelas comunicações internas que precisam ser realizadas entre os microsserviços para completar todo o processo nos planos de testes, comunicações essas que são feitas através de requisições HTTP, resultando em um maior tempo de comunicação entre os serviços. Acredita-se que a diferença no desempenho das duas aplicações poderia ser reduzida se fosse implementado

um *middleware* (software que permite a comunicação e o gerenciamento de dados para aplicações distribuídas) no lugar de requisições HTTP para realizar a comunicação entre os serviços.

4.2.7 Infraestrutura dos projetos

Para que não existisse nenhuma interferência nos resultados do teste de performance, foi determinado a mesma estrutura de escrita de código e separação de arquivos em ambos os projetos, assim como foi utilizado o mesmo banco de dados para as duas arquiteturas, o SQLite.

A estrutura adotada para a execução das aplicações requerem baixa complexidade, sendo esta responsável por gerir um único processo a qual iniciará os scripts escritos em JavaScript e sustentará a comunicação das requisições através do framework Express. Quanto a infraestrutura utilizada para gestão do projeto, foram utilizadas as ferramentas Docker e Kubernetes para facilitar a abstração de criação e implantação dos ambientes de execução das aplicações.

5 CONCLUSÃO

Este trabalho, em sua maioria, aborda assuntos sobre a concepção de uma aplicação em sua fase inicial. Seja ela utilizando a arquitetura monolítica ou a de microsserviços. Todavia, não pode-se 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 influenciará em boa parte deste ciclo, assim como as regras de negócio do produto a ser desenvolvido, testes de integração, implantação, monitoramento e principalmente a manutenção.

Com base nessas considerações, é possível afirmar que a arquitetura em microsserviços tem seu destaque em grandes aplicações onde seus benefícios são identificados facilmente. Um projeto tem seu tempo de inicialização bem mais rápido, visto que cada serviço é independente. Isso faz com que os desenvolvedores ganhem maior agilidade desenvolvendo e implantando com essa arquitetura. Pela mesa premissa de independência entre os serviços, publicar novas versões tende a ser mais frequente, assim como também proporciona aos gestores uma maior precisão no dimensionamento de projetos e equipes.

Com serviços menores e desacoplados, a facilidade de utilizar uma tecnologia diferente e mais apropriada para a construção de cada serviço também pode ser identificado como um benefício proporcionado por essa arquitetura. Essa liberdade na escolha da tecnologia pode aferir ao projeto grandes melhorias de performance, o que não foi comparado nas análises e testes aplicados neste trabalho. Assim como também não foi analisado o cenário de escalonamento das aplicações. Mas acredita-se que neste caso, a arquitetura de microsserviços apresenta vantagem sobre a arquitetura monolítica. Pois na arquitetura de microsserviços, pode ser escalonado apenas novas instâncias do serviço que está sendo mais requisitado no momento, enquanto na arquitetura monolítica seria necessário escalonar novas instâncias de toda a aplicação.

Quanto a arquitetura monolítica, evidencia-se a vantagem na facilidade para desenvolver e implantar pequenas aplicações em sua fase inicial, visto que é possível abstrair várias preocupações que a arquitetura em microsserviços exige

atenção, preocupações relacionadas ao reconhecimento e registro de novos serviços, bem como estabelecer uma sólida e assertiva forma de realizar a comunicação entre os microsserviços. Como na arquitetura monolítica os serviços estarão acoplados dentro de uma mesma aplicação, essas preocupações podem ser abstraídas. E por fim, como foi visualizado nos projetos utilizados para fins de comparação, também podem existir vantagens de desempenho de uma aplicação monolítica em sua fase inicial, visto que o acesso à memória é mais rápida do que a comunicação entre os microsserviços.

5.1 TRABALHOS FUTUROS

Com relação a trabalhos futuros referente a este tema, outras arquiteturas de desenvolvimento de software podem ser desenvolvidas para traçar suas respectivas demonstrações de utilização e comparações. Outra abordagem mais detalhada a respeito deste projeto seria desenvolver testes e especializações com mais profundidade a respeito das arquiteturas abordadas neste trabalho. Além dos tópicos diretamente relacionados às arquiteturas abordadas, este estudo comparativo também abre portas para a área de desenvolvimento e operações (Devops), virtualização de ambientes através do uso de contêineres, ferramentas de automação e gerenciamento de aplicações desenvolvidas utilizando a arquitetura de microsserviços.

REFERÊNCIAS

APACHE. JMeter. <http://jmeter.apache.org/>. Acessado em 25/09/2020.

DOCKER. Docker overview. <https://docs.docker.com/get-started/overview/>. Acessado em 26/09/2020.

FOWLER, Susan. **Microsserviços prontos para a produção**: construindo sistemas padronizados em uma organização de engenharia de software. São Paulo: Novatec Editora, 2017. 224 p.

FOWLER, M. **Microservices, a definition of this new architectural term**. Março 2014. <https://www.martinfowler.com/articles/microservices.html>. Acessado em 10/09/2020.

GRIDER, S. **Microservices with Node JS and React**. Setembro 2020 <https://medium.com/ucan-learn-to-code/stephen-grider-microservices-with-node-js-and-react-review-7fa541610388>. Acessado em 15/10/2020.

NODEJS. NodeJs. <https://nodejs.org/en/>. Acessado em 25/09/2020.

RAMIREZ, C. **Build a NodeJS cinema microservice and deploying it with docker part 1**. 2017. <https://medium.com/@cramirez92/builda-nodejs-cinema-microservice-and-deploying-it-with-docker-part-1-7e28e25bfa8b>. Acessado em 25/09/2020.

RIOS, A. **Tem certeza de que seu time pode usar micro serviços?**. Dezembro 2019.<https://dev.to/alextrending/tem-certeza-de-que-seu-time-pode-usar-micro-servicos-30i7>. Acessado em 17/09/2020.

THE LINUX FOUNDATION. Kubernetes. <https://kubernetes.io/>. Acessado em 26/09/2020.