INSTITUTO DE GESTÃO EM TECNOLOGIA DA INFORMAÇÃO

ARQUITETURA DE MICROSERVIÇOS NA CRIAÇÃO DE APLICAÇÕES ESCALÁVEIS E TOLERANTE A FALHAS

Vanderson Zanoni Campanholi
Prof. orientador: Igor Ribeiro Lima
MBA em Arquitetura de Software

Resumo

A cada dia as aplicações são responsáveis por um grande tráfego de dados na rede e necessitam estar prontas e aptas para receberem ainda mais acessos e se manter estáveis e acessíveis o maior tempo possível. Com a arquitetura de microserviços é possível decompor grandes serviços ou módulos em pequenas partes independentes a fim de manter uma aplicação em operação com menos recursos de *hardware* e assim utilizar uma arquitetura escalável horizontalmente e assim permitir a separação de funcionalidades e recursos. Com base nessas afirmações e estudos foram identificadas formas de implementações com a utilização de microserviços para tornar um *software* escalável o suficiente para receber quantidades aceitáveis de acessos ou requisições. O objetivo principal deste projeto é definir formas de desenvolvimento de *softwares e* utilizar a tecnologia de microserviços. Neste contexto foi realizado um estudo bibliográfico para identificar vantagens e desvantagens da tecnologia na construção de aplicações escaláveis em estruturas que possuem um único módulo e que tendem a migrar e crescer, desta forma manter a aplicação funcional após falha em determinada parte, módulo ou em uma funcionalidade específica.

Palavras-chave: Microserviços; Escalabilidade; Arquitetura de Software; Tolerância a Falhas.

Abstract

Every day the applications are responsible for a large traffic of data in the network and they need to be ready and able to receive even more access and to remain stable and accessible as long as possible. With the microservice architecture it is possible to decompose large services or modules into small independent parts in order to keep an application in operation with less hardware resources and thus use a scalable architecture horizontally and thus allow the separation of features and resources. Based on these affirmations and studies were identified ways of implementations with the use of microservices to make a software scalable enough to receive acceptable amounts of access or requisitions. The main objective of this project is to define forms of software development and to use microservice technology. In this context, a bibliographic study was carried out to identify the advantages and disadvantages of the technology in the construction of scalable applications in structures that have a single module and that tend to migrate and grow, thus maintaining the functional application after failure in a particular part, module or in a Specific functionality.

Keywords: Microservices; Scalability; Software Architecture; Fault Tolerance.

1. INTRODUÇÃO

Visto a grande diversidade de abordagens e estilos arquiteturais para a construção de sistemas, conhecer as vantagens e desvantagens de cada tipo de arquitetura é fundamental para o sucesso do negócio e desenvolvimento de um *software* com qualidade e que não apresentem interrupções. Aplicações podem ser construídas em diferentes tecnologias com variadas linguagens de programação e em ambientes que demandam ou não uma maior quantidade de *hardware* para execução.

Conforme Newman (2016), um problema recorrente no desenvolvimento e manutenção de *softwares* são os impactos negativos encontrados por alterações de código em aplicações complexas com diferentes integrações e módulos que são dependentes entre si. Este estilo arquitetural atualmente empregada em aplicações monolíticas vem ganhando um novo formato passando a ser dividida e distribuída em serviços pequenos com um maior controle sobre alterações e paralelamente a isso dividir suas responsabilidade para obter ganhos de *performance* disponibilidade e recursos de *hardware*.

O objetivo principal dessa pesquisa é obter recursos perante a literatura existente e através de estudo de caso auxiliar desenvolvedores e arquitetos na criação de microserviços a fim de isolar partes específicas na execução de uma rotina em um *software* monolítico, garantindo a disponibilidade da aplicação, com ganho de qualidade e tempo em modificações. Reduzir a quantidade de falhas e disponibilidade do *software* em produção é fundamental e deve ser levada em consideração por todos os integrantes de um time de desenvolvimento além de permitir que a aplicação cresça independente do ambiente e do aumento considerável de acessos.

A pesquisa bibliográfica com o estudo de caso real não tem por objetivo realizar comparações entre aplicações monolíticas e microserviços, mesmo que em alguns casos a comparação ocorra, mas sim conceituar que é possível realizar a migração de uma rotina para um microserviço auto-suficiente para obter algumas vantagens dos serviços independentes como a tolerância a falhas e escalabilidade.

Esse artigo tem a seguinte estrutura, a próximo seção detalha a metodologia adotada no trabalho, dando continuidade, a seção 3 apresenta uma revisão da literatura

com os conceitos da arquitetura de microserviços, suas características dentro de um projeto de *software* e demonstrar suas vantagens e desvantagens, ainda na mesma seção apresenta as aplicações monolíticas, o legado delas na arquitetura e a possível refatoração deste legado para serviços independentes. Na seção 4 encontra-se o estudo realizado para a aplicação de um microserviço, onde é apresentada a empresa, como o processo é desenvolvido hoje, em seguida no próximo tópico a configuração, criação e implantação do serviço. Finalizando com a seção 6, os resultados obtidos referente ao estudo juntamente com as considerações finais.

2. METODOLOGIA

Esta pesquisa teve como base um estudo da arquitetura de microserviços, onde foram analisadas as vantagens e desvantagens de usar esta arquitetura para tornar uma aplicação escalável e tolerante a falhas, podendo servir como referência para equipes de desenvolvimento e arquitetos de *software* na criação de novos sistemas ou funcionalidades em aplicações monolíticas, destacando a autonomia de pequenos serviços no conjunto do sistema.

Classifica-se como uma pesquisa aplicada por apresentar uma solução a um problema real, no caso específico a implantação de um pequeno serviço de validação de assinatura de arquivos, quanto aos objetivos trata-se de uma pesquisa exploratória pois buscou um exemplo real de desenvolvimento de *software* utilizando aplicações monolíticas. Segundo explica Gil (2002), a pesquisa exploratória tem como um objetivo válido proporcionar uma maior interação e familiaridade com o problema exposto tornando-o mais explícito.

Alguns pontos serão identificados na pesquisa bibliográfica, dentre eles estão:

- O que são microserviços, os benefícios e desvantagens?
- Como manter uma aplicação funcional e estável quando determinada funcionalidade falhar?
- Quais as formas de manter uma aplicação escalável utilizando microserviços?
- Como este modelo de arquitetura interfere o desenvolvimento da aplicação?

De posse deste material teórico, será realizado um estudo para a implementação de um microserviço e permitir que o mesmo funcione nas mesmas condições mas de forma isolada. Neste ponto a pesquisa irá:

- Identificar as ferramentas e configurações necessárias para a criação.
- verificar a viabilidade da criação do serviço em relação a estrutura do projeto.
- Implementar o microserviço e permitir que ele seja executado independentemente.
- Identificar possíveis pontos de falhas e o comportamento da aplicação monolítica com um serviço sendo executado isoladamente.

A técnica aplicada para levantar conhecimentos sobre o tema foi a pesquisa bibliográfica elaborada a partir de material já publicado, como livros, artigos, periódicos, Internet, também realizando um estudo de caso, que segundo Gil (2002) este formato de pesquisa do ponto de vista técnico aplica-se a este conceito pois irá realizar um estudo em um cenário existente.

Devido a dimensão da empresa o cenário em estudo será somente focado no setor de desenvolvimento restrito à desenvolvedores, arquitetos e analistas além de pessoas responsáveis pela qualidade e que diariamente tem contato direto com o desenvolvimento de aplicações.

3. REVISÃO DE LITERATURA

Sistemas distribuídos estão cada vez mais substituindo aplicações monolíticas com várias linhas de código para serviços menores e auto-suficientes, este tipo de arquitetura traz consigo benefícios da mesma forma que podem trazer desvantagens. A entrega de *software* mais rápida e a busca por novas tecnologias faz com que o uso de microserviços dê mais liberdade de reagir a mudanças e tomar decisões de forma diferente, além de permitir responder mais rapidamente à inevitável mudança. (NEWMAN, 2016).

3.1 Arquitetura de Microserviços

Para Fowler e Lewis (2014) microserviços descrevem um estilo arquitetural de desenvolvimento de *software*. Cada vez mais projetos vêm usando este formato e os resultados têm sido positivos, para alguns se tornou uma forma de padrão para desenvolver aplicações empresariais. É uma abordagem para desenvolver uma aplicação única com um conjunto de serviços, executados independentemente em seu próprio processo e mantendo uma comunicação, geralmente através de *Application Programming Interface* (API) e *Hypertext Transfer Protocol* (HTTP). Cada serviço possui pequenas responsabilidades até mesmo para a publicação em ambientes de produção. Podem ser desenvolvidos em diferentes linguagens de programação utilizando diferentes tecnologias de bancos de dados, como pode ser observado na Figura 1.

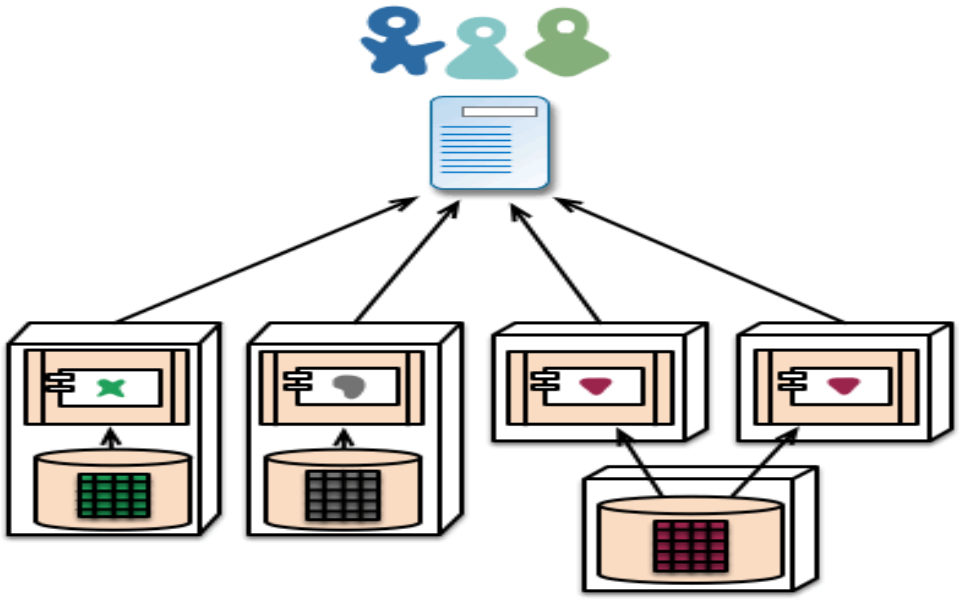

Figura 1: Microserviços com diferentes bancos de dados.
Fonte: o autor, 2017

Moreira (2017) reforça que a arquitetura de microserviços hoje é bem consolidada na área de tecnologia da informação, onde o termo vem sendo dissipado como uma forma de projetar aplicações de *software* como suítes de serviços desenvolvidas independentemente. No meio corporativo, mais e mais aplicações estão utilizando este estilo e obtendo resultados positivos.

3.1.1 Características de uma arquitetura de microserviços

Newman (2016) recomenda que algumas características devem ser seguidas para que microserviços possam obter todos os resultados esperados:

- Alta coesão: as responsabilidades devem ser única, comportamentos associados devem ser agrupados, enquanto que comportamentos desconexos não devem ser reunidos, desta forma alterações não são propagadas.
- Baixo acoplamento: um serviço deve saber o mínimo necessário dos serviços com os quais ele interage, garantindo que um serviço não sofra mudanças em razão de outros serviços, nem cause alterações em outros serviços devido a modificações em suas funcionalidades.
- Regras de negócio: ao delimitar o contexto de cada serviço, suas próprias funcionalidades não devem ser voltadas aos dados compartilhados com o mundo exterior, mas sim às operações que o serviço realiza e aos dados dos quais ele precisa para isso.
- Delimitação do contexto: microsserviços encapsulam informações que são pertinentes ao seu domínio, expondo através de interfaces apenas o que deve ser compartilhado com os consumidores.
- Comunicação condicionada aos conceitos de negócio: a comunicação entre os serviços deve obedecer ao modo em que os contextos que eles representam se comunicam na vida real. Isto facilita a implementação de mudanças em decorrência de alterações nas regras de negócio.

3.1.2 Vantagens

Para Newman (2015) a utilização da arquitetura de microserviços traz benefícios variados e podem ser utilizados em qualquer sistema distribuído, sendo que tendem a alcançar esses benefícios em maior grau principalmente quando levam os conceitos de sistemas distribuídos e orientados serviços.

Se a necessidade é melhorar o desempenho, há a possibilidade de utilizar

tecnologias capazes de atingir níveis melhores de desempenho e garantir que serviços possam evoluir sua tecnologia independentemente um do outro, conseguindo assim uma heterogeneidade da tecnologia (NEWMAN, 2015).

Possibilidade de escalonamento de serviços independentemente, além de identificar gargalos e endereça-los diretamente. Partes do sistema que não representam nenhum tipo de gargalo podem permanecer simples e não acoplados de acordo com Schroeder (2016).

Se um componente do sistema falha é possível isolar o problema e continuar utilizando a aplicação. Em uma aplicação monolítica isto não é possível, uma falha pode parar a aplicação toda, já as aplicações com arquitetura de microserviços podem ser executados em várias máquinas para reduzir a chance de falhas, com microserviços é possível construir sistemas que lidam com a falha total de serviços e degradar a funcionalidade em conformidade (NEWMAN, 2015).

A implantação independente permite realizar escalas seletivas ou sob demanda, como por exemplo, um microserviço que frequentemente necessita de alta carga, nele pode-se realizar outra implantação ou mover para um ambiente com mais recursos, sem precisar aumentar a capacidade de *hardware* para todo o sistema corporativo (NADAREISHVILI et. al, 2016).

Conforme Newman (2015), outra vantagem é o alinhamento organizacional, grandes equipes e grandes bases de códigos trazem problemas recorrentes o que pode ser reduzido utilizando microserviços, pois a equipe irá trabalhar de forma distribuída e assim alinhando melhor a arquitetura. Também é possível transferir a responsabilidade de serviço entre equipes para tentar manter pessoas trabalhando alocadas em serviços.

3.1.3 Desvantagens

Segundo Moreira e Beder (2015) alguns pontos considerados como desvantagem da arquitetura são: complexidade de desenvolvimento, chamadas remotas e gerenciamento de múltiplos bancos de dados e transações.

- Complexidade de desenvolvimento: um grande problema em relação à microserviços é o alto nível de complexidade no desenvolvimento, levando em consideração que é muito mais fácil escrever sistemas em um único projeto realizando chamadas internas entre as classes, quando se trabalha com diferentes projetos e módulos faz-se necessário o gerenciamento de suas dependências e manter as versões atualizadas. Microserviços são desenvolvidos para trabalhar de forma autônoma, se os diferentes módulos do projeto não forem atualizados, os microserviços não tirarão proveito das novas alterações que foram liberadas.

  Em projetos com uma grande quantidade de módulos sendo desenvolvidos ao mesmo tempo é essencial à comunicação entre as equipes para um bom andamento.
- Chamadas remotas: chamadas remotas são muito mais custosas do que chamadas de classes internas, merecendo uma melhor atenção sobre a comunicação entre os microserviços. Caso um serviço seja muito pequeno, a comunicação via rede irá se tornar um problema, cada serviço deve ser único e suficiente para executar sua funcionalidade evitando a realização de processos adicionais, diminuindo a coesão.
- Gerenciamento de múltiplos bancos de dados e transações: controles transacionais em sistemas distribuídos são extremamente complexos, principalmente onde diversos bancos de dados são consumidos por vários serviços. Um grande esforço inicial é aplicado para realizar todas as configurações e ainda manter vários bancos funcionando tem um alto custo, nos casos de sistemas distribuídos é difícil manter recursos transacionais de suporte a *rollback* uma vez que ocorra a falha diversos tratamentos e controles necessitam ser realizados.

3.2 Aplicações monolíticas

Um *software* monolítico é um aplicativo composto por módulos que não são independentemente da aplicação a que pertencem. Como os módulos de um monolito

dependem dos referidos recursos compartilhados, eles não são independentes (DRAGONI et. al, 2016).

Grande maioria das linguagens de desenvolvimento possuem meios de isolar e quebrar a complexidade dos sistemas em módulos, geralmente são projetadas para a criação de um único executável, onde todos os módulos são executados em uma única máquina, compartilhando processamento, memória, bancos de dados e arquivos.

A Figura 2 representa uma arquitetura típica de um sistema monolítico, onde as funções do negócio são desenvolvidas em um único processo.

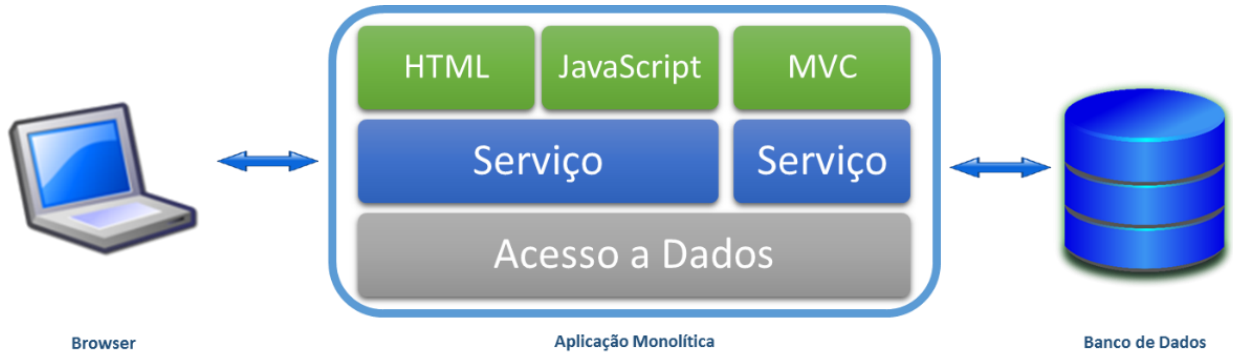

Figura 2: Arquitetura de um sistema monolítico.
Fonte: o autor, 2017.

3.2.1 Legado monolítico

Conforme Machado (2017) sistemas crescem constantemente e tornam-se cada vez mais complexos, dependendo de vários recursos, surgindo assim grandes desafio em relação à sua manutenção. Dentre os desafios encontram-se:

- Aumento de complexidade e tamanho ao longo do tempo: A alta complexidade das aplicações faz com que a manutenção se torne cada vez mais cara e lenta, devido a grande quantidade de código que desenvolvedores necessitam analisar.
- Alta dependência de componentes de código: Incluir ou realizar manutenções em um componente pode causar problemas ou comportamentos inesperados, devido a grande quantidade de funções interdependentes ou entrelaçadas com

outros componentes.

- Escalabilidade do sistema é limitada: O sistema é tratado como um todo e exige que seja replicado mesmo que apenas parte de sua funcionalidade seja necessária, o que ocasiona custos muito maiores que o esperado.
- Falta de flexibilidade: Aplicações neste tipo de arquitetura deixam de ser flexíveis quando estão vinculadas originalmente a uma tecnologia escolhida, mesmo que em algumas situações não seja a melhor escolha.
- Dificuldade para colocar alterações em produção: Por menores que sejam as mudanças é necessário que o sistema seja reiniciado, podendo trazer riscos operacionais necessitando acompanhamento de toda equipe.

3.3 Refatorar aplicações monolíticas

Nem sempre é possível trabalhar em projetos novos que ainda estão em desenvolvimento, há grandes possibilidades de se trabalhar em equipes responsáveis por aplicações monolíticas extremamente grandes e que já estão há algum tempo em produção. para este tipo de situação existem algumas técnicas utilizadas para que estas aplicações sejam divididas em conjuntos de serviços (RICHARDSON, 2014).

Conforme Richardson (2014) o primeiro passo é não propagar o problema, ou seja, o que for criado de novo já deve ser implementado com serviços independentes, mesmo não sendo uma tarefa fácil pois uma parte da nova funcionalidade geralmente vai estar integrada com uma parte monolítica, neste caso utilizam-se códigos que não possuem funcionalidade em relação à aplicação mas que serviriam como um elo entre as partes (*glue code*).

O segundo passo seria a identificação de componentes da aplicação monolítica que seriam transformados em serviços independentes. Geralmente componentes que estão em constante modificação ou que possuem conflitos de recursos, além da camada de apresentação. Mesmo não sendo muito simples o processo de migração para serviços independentes é uma atividade que permite gradativamente a migração para a arquitetura de microserviços (RICHARDSON, 2014).

4. APRESENTAÇÃO DA PESQUISA

4.1 A empresa

Para aplicação prática da área de estudo escolhida, foi utilizada a própria empresa com vínculo empregatício, porém será utilizado o nome fictício *InfoCloud* durante o desenvolvimento da pesquisa. A empresa trabalha na área de desenvolvimento de *softwares* para gestão e contabilidade pública atuando no mercado a mais de 30 anos, com seus sistemas sendo utilizados por mais de 400 mil usuários distribuídos entre 900 municípios brasileiros. Atualmente possui mais de 500 colaboradores divididos em diversas funções, na área técnica de desenvolvimento estão divididos entre arquitetos de *software*, analistas de sistemas, analistas de requisitos, desenvolvedores e testes, todos divididos entre equipes.

A empresa passou por diversas reestruturações em relação à tecnologias de desenvolvimento e banco de dados, passando por aplicações *desktop* atualmente ainda desenvolvidas em *PowerBuilder* que utiliza banco de dados *Sybase* até aplicações *web* utilizando Java, *AngularJs*, *JavaScript* e banco de dados Oracle.

A análise do trabalho será realizada no setor de desenvolvimento da empresa em questão aplicando o conteúdo sobre uma rotina de uma aplicação real criada para estudos. Para fins de aprendizagem, foi incorporado à aplicação um microserviço responsável por validar a assinatura digital de um arquivo contendo informações de coletas de preços de um processo administrativo.

4.2 Atual processo de desenvolvimento aplicado

Atualmente o processo de desenvolvimento da empresa ainda mantém grandes aplicações em versões *desktop* que já são utilizadas há muito tempo e estão passando por um processo de migração contínuo. Novos projetos vêm sendo desenvolvidos nesses últimos anos a fim de eliminar essas versões antigas e migrar todas as aplicações para *web*, vários projetos já estão em fase final. Para o desenvolvimento das aplicações estão sendo utilizadas novas tecnologias mas ainda não partiram para uma

arquitetura de microserviços, o software ainda é concebido em módulos onde todos são dependentes.

Fazem parte da equipe dois analista de sistemas, um arquiteto de *software* e quatro programadores, devido a grande quantidade de correções e melhorias que estão sendo realizadas para que o produto possa ser liberado em produção muitas vezes os ambientes locais e de testes passam por instabilidades devido a alguma falha que por vezes interrompe o funcionamento de toda aplicação.

5. DISCUSSÃO DOS RESULTADOS

5.1 Configuração e implantação do microserviço

Devido ao desenvolvimento ser executado dentro de um ambiente organizacional alguns códigos fontes não podem ser apresentados devido ao negócio envolvido e por regras de conduta da empresa. Atualmente a organização possui um setor de pesquisa e desenvolvimento voltado para a construção de *frameworks*, publicações em ambientes de produção, gerenciamento de banco de dados e infraestrutura, por este motivo o desenvolvimento do microserviço será realizado no ambiente de desenvolvimento sem ter relação nenhuma com ambientes de testes e produção.

Como não é possível expor a aplicação onde à rotina será implantada, foi realizado um estudo para a criação de uma aplicação simples que serve como um exemplo para possíveis implementações futuras em algum ambiente local de estudos. Desta forma faz-se necessário a utilização de ferramentas, configurações e alguns pré-requisitos.

- *Java* 1.8: Plataforma Java com *Java Development Kit* (JDK) 1.8;
- *Maven* 3.2: Construir aplicação e gerenciar bibliotecas;
- *Spring Boot*: Configurações;
- Eclipse: IDE de desenvolvimento;

O *Spring Boot* é um *framework* Java para a construção de microserviços baseado

na estrutura de injeção de dependência da *Spring*. O *Spring Boot* permite que desenvolvedores criem microserviço reduzindo a configuração por parte do desenvolvedor, favorecendo a configuração automática e convencional por padrão além de simplificar pacotes de aplicações (POSTA, 2016).

Grande parte das configurações do ambiente é realizada utilizando *command-line interface* (CLI) do próprio *Spring* Boot através de linhas de comando, devido à facilidade de uso.

Verificar a versão da instalação:

```
$ spring –version
Spring CLI v1.3.3.RELEASE
```

Navegar até o diretório onde será armazenado o exemplo em questão, utilizando o comando:

```
spring init --build maven --groupId com.infocloud .exemplo \
--version 1.0 --java-version 1.8 --dependencies web \
--name apptest-springboot apptest-springboot
```

Após comando executado será criado um diretório chamado *apptest-springboot* com um aplicativo *Spring Boot*. Após este procedimento temos alguns comandos disponíveis, dentre eles:

--build

A ferramenta de gerenciamento de compilação que será usada. *Maven* ou *gradle*. São as duas opções válidas neste momento.

--groupId

O *groupId* será usado em coordenadas do *Maven* para o uso no arquivo pom.xml;

Infelizmente isso não se estende corretamente para o pacote Java precisam ser modificados manualmente.

--version

A versão da aplicação. Será usado em iterações posteriores, definido para 1.0.

--java-version

Permite verificar a versão do compilador de compilação para o JDK.

--dependencies

Verifica o conjunto de dependências.

Navegar para o diretório *apptest-springboot* e executar o comando:

```
$ mvn spring-boot:run
```

Caso nenhum problema tenha ocorrido e possível navegar até a *url* http://localhost:8080 e a seguinte página será demonstrada, conforme Figura 3. A informação de erro é padrão e esperada já que a aplicação não possui teste, não possui nenhuma funcionalidade e não executa nenhum processo.

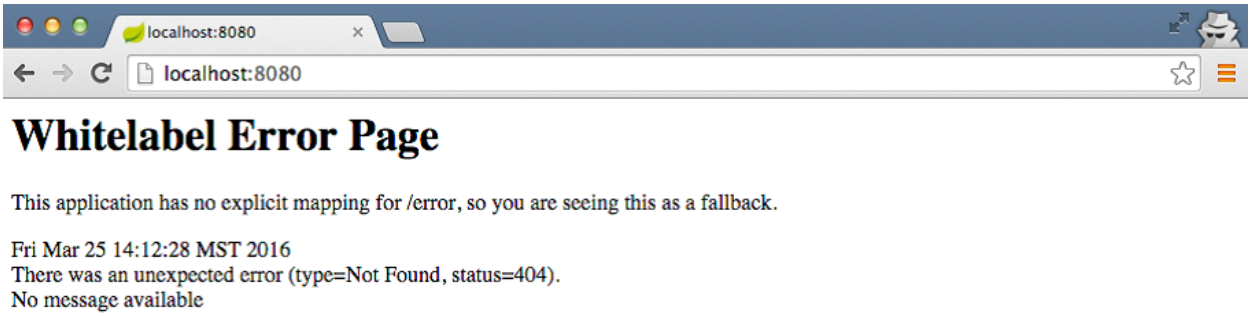

Figura 3: Página recebida pelo navegador.
Fonte: o autor, 2017.

5.2 Criação do microserviço

Até o momento uma aplicação simples foi criada utilizando *Spring Boot* com a finalidade de exemplificar a criação de uma aplicação. Nesta parte da implementação será codificado um serviço simples que será construído pelo *jenkins* gerando ao final um container *Docker* em execução, o processo de construção será uma máquina contendo a aplicação rodando. O microserviço realiza algumas validações na assinatura digital de um arquivo, utilizando outro certificado já assinado pela empresa. Para a criação foi utilizado o *Dropwizard*[1] que possui algumas características:

- Utiliza *JAX-RS*, através do projeto *Jersey*;
- Usa *Jackson*, para serialização *JSON*;

[1] *Dropwizard*: Estrutura Java para o desenvolvimento de serviços web restful de alto desempenho (POSTA, 2016).

- Usa uma combinação de "*path*" e método HTTP, que decide qual método Java será invocado usando um Container *Jetty* embutido, agregando agilidade ao serviço. O *Jetty* é uma biblioteca que torna a aplicação capaz de lidar com HTTP, sem ser uma aplicação Java EE. Considerado um projeto Java comum, que gera um arquivo JAR (POSTA , 2016).

A classe AssinaturaResource.java foi criada e contém o método principal, conforme pode ser observado na Figura 4:

```
1  /*Vanderson.Campanholi 10/04/2017*/
2  @Path("/assinatura")
3  public class AssinaturaResource {
4      private String caminho;
5      private String alias;
6      private String senha;
7      @POST
8      @Timed
9      @Consumes(MediaType.APPLICATION_JSON)
10     public Response verifica(SampleDocument document) {
11         boolean resultado = false;
12         String mensagem = "";
13         String situacao = "";
14         int httpsituacao = 200;
15         try {
16             resultado = VerificaAssinatura.verifica(document.getHexSignature(),
17                 document.getTexto(),this.caminho, this.alias, this.senha);
18             if (!resultado) {
19                 situacao = "falhou";
20                 mensagem = "assinatura incorreta";
21             } else {
22                 situacao = "sucesso";
23                 mensagem = "assinatura correta";
24             }
25         } catch (Exception e) {
26             situacao = "erro";
27             mensagem = e.getMessage();
28             httpsituacao = 500;
29         }
30         String saidaJSON = "{ \"situacao\": \"" + situacao + "\",\"data\": "
31                 + "{\"mensagem\": \"" + mensagem + "\"}}";
32         return Response.status(httpsituacao).entity(saidaJSON).build();
33     }
34 }
```

Figura 4: Classe do serviço contendo o método principal.

Fonte: o autor, 2017.

Para que o serviço possa ser consumido, um pequeno serviço em *NodeJS* foi criado, como pode ser observado na Figura 5. Ele exemplifica bem o conceito de interoperabilidade, podendo se comunicar de forma transparente com outro sistema.

```
var http = require('http');
var fs = require('fs');
var file = fs.readFileSync("C:/Users/vanderson.campanholi/assinatura/src/test/resources/arquivo.txt", "utf8");
var exemplo = {
  texto: file,
  hexSignature: '8ed7b4235f21db78c92e69082df3874c03d4135515cb04ff10f07715f0cb65beb839dbf33d691acc30b'
};
var exemploString = JSON.stringify(exemplo)
var headers = {
  'Content-Type': 'application/json',
  'Content-Length': exemploString.length
};

var options = {
  host: 'localhost',
  port: 8080,
  path: '/assinatura',
  method: 'POST',
  headers: headers
};
var req = http.request(options, function(res) {
  res.setEncoding('utf-8');
  var responseString = '';
  res.on('data', function(data) {
    responseString += data;
  });
  res.on('end', function() {
    console.log(responseString)
    var resultObject = JSON.parse(responseString);
  });
});
req.write(exemploString);
req.end();
```

Figura 5: Código *Javascript* para consumir o serviço.
Fonte: o autor, 2017.

Basicamente a aplicação possui estas classes principais entre outras de configurações e parâmetros, para que a aplicação possa ser executada basta compilar o projeto com o comando do Maven: **mvn clean package** e executar o .jar resultante desta compilação com: **java -jar assinatura-0.0.1-SNAPSHOT.jar server assinatura.yml**

O programa cliente pode ser executado desde que o Node.js esteja instalado e com o comando *node request*, que retorna o seguinte resultado em caso de sucesso:

enterprise:dockertest vandersoncamp$ node request

{"situação":"sucesso","data": {"mensagem": "assinatura correta"}}

enterprise:dockertest vandersoncamp$

5.3 – Distribuição do serviço

Para a distribuição do serviço foi utilizado *Container*, rodando em uma máquina virtual pelo fato de estar usando *Windows*, sendo iniciado pelo *Boot2Docker*. Após instalação do *Docker* e da máquina virtual além de algumas possíveis configurações de variáveis de ambientes em alguns casos, é possível executar comandos *Docker*.

O Container *Docker* depende de três fatores para ser criado, um arquivo .jar, o arquivo .yml usado pelo *Dropwizard* e um arquivo *Dockerfile* que está localizado dentro do projeto.

```
FROM dockerfile/java:openjdk-7-jdk
ADD assinatura-0.0.1-SNAPSHOT.jar /data/assinatura-0.0.1-SNAPSHOT.jar
ADD assinatura.yml /data/assinatura.yml
CMD java -jar /data/assinatura-0.0.1-SNAPSHOT.jar server /data/assinatura.yml
EXPOSE 3000
```

Este programa utiliza imagem do sistema operacional para gerar um Container *Docker*, ao final demonstra a porta (EXPOSE 3000). Ao copiar o arquivo .jar , o arquivo assinatura.yml e o *Dockefile* para a máquina virtual do *Boot2Docker* é possível criar um Container com os comandos:

```
docker build -t "signatureimage"
docker run -d -p 3000:3000 --name assinaturaserver assinaturaimage
```

Em relação à entrega contínua do microserviço é possível utilizar o *Jenkins* como ferramenta para deste processo. Sua configuração é simples após a instalação. Pode ser criado um *job* para o *jenkins*, criando um *build* para pegar do repositório onde está o código fonte e rodar os comandos de *build* do *Maven*. O próximo passo será executar alguns comandos de *Shell Script*.

```
export DOCKER_HOST=tcp://192.168.0.108:1752
export DOCKER_CERT_PATH=/Users/vandersoncamp/.boot2docker/certs/boot2docker-vm
```

```
export DOCKER_TLS_VERIFY=1
cp $WORKSPACE/assinatura/target/assinatura-0.0.1-SNAPSHOT.jar .
cp $WORKSPACE/assinatura/src/main/resources/assinatura.yml .
cp $WORKSPACE/assinatura/Dockerfile .
/usr/local/bin/docker ps
set +e
/usr/local/bin/docker rm -f assinaturaserver
set -e
/usr/local/bin/docker build -t "assinaturaimage" .
/usr/local/bin/docker run -d -p 3000:3000 --name assinaturaserver assinaturaimage
```

4. CONSIDERAÇÕES FINAIS

Com uma simples implementação de um microserviço é possível destacar algumas vantagens, dentre elas a que tem um papel fundamental na utilização de serviços independentes é a escalabilidade seletiva. Também é possível perceber que quando se tem um sistema monolítico geralmente é necessário aumentar a capacidade do servidor, na utilização de serviços pode-se escalar somente parte do sistema, ou seja, o que realmente necessita de mais desempenho, sem que o resto seja afetado.

No exemplo prático não houve a necessidade de tantos recursos já que se tratava de um simples serviço, mas se pensar em uma aplicação com uma quantidade considerável de rotinas e caso elas sejam executadas simultaneamente. Uma boa prática seria a migração de partes da aplicação, onde seria possível gerenciar pequenas partes evitando um grande problema encontrado hoje em sistemas monolíticos que seria a falha. Uma pequena parte apresentaria problema o que não afetaria toda a aplicação, reduzindo consideravelmente o tempo de manutenção e facilitando o processo de entrega contínua.

Como toda tecnologia sempre tem algo a melhorar ou a acrescentar, um ponto que deixa a desejar na experiência com microserviço é o *deploy*, se existir um problema ao realizar um *deploy* de uma aplicação monolítica ao migrar para microserviços esses problemas serão multiplicados pela quantidade de serviços independentes

disponibilizados. O que se tem que saber ao iniciar um projeto de migração monolítica para microserviço é qual o problema que se quer resolver e depois disso alinhar os seus prós e contras.

Quando se fala em microserviços não podemos ter em mente que uma aplicação monolítica está ultrapassado ou não tem mais capacidade suficiente para se manter operante, as duas formas de arquitetura presentes no projeto não são dois extremos, cada uma possui características diferentes, além de existirem diversas aplicações que não se enquadram em nenhuma das duas, seja por uma outra arquitetura utilizada ou até mesmo por mesclar arquiteturas para obter o melhor de cada uma em determinado cenário.

É possível levantar pontos fortes e fracos de projetos orientados a microserviços em determinados cenários, principalmente em conjunto com sistemas monolíticos, basta seguir uma estratégia de arquitetura e aplicar bem seus conceitos conforme indicado por renomados autores e especialistas no assunto.

REFERÊNCIAS

DRAGONI Nicola; LAFUENTE Alberto Lluch; MAZZARA Manuel; MONTESI Fabrizio. **Microservices: yesterday, today, and tomorrow**, United States, v. 1, n. 1, p. 209-215, jun 2016.

FOWLER, Martin. LEWIS, James. **Microservices a definition of this new architectural term**. 2014. Disponível em: <https://martinfowler.com/articles/microservices.html>. Acesso em: 04 abr. 2017.

GIL, Antonio Carlos. **Como elaborar projetos de pesquisa**. 4ª ed. São Paulo, SP: Atlas. 2002.

MACHADO, Marcos Giacometti. **Micro Serviços: Qual a diferença para a Arquitetura Monolítica?** 2017. Disponível em: <http://www.opus-software.com.br/microservicos-diferenca-arquietura-monoliticas/>. Acesso em: 05 abr. 2017.

MATTAR, F. N. **Pesquisa de marketing.** 3.ed. São Paulo: Atlas, 2001.

MOREIRA, Thiago. **Arquitetura de Integração de Aplicações**. IGTI. Instituto de Gestão de Tecnologia da Informação. 2017.

MOREIRA, Pedro Felipe Marques; BEDER, Delano Medeiros. **Desenvolvimento de Aplicações e Micro Serviços: Um estudo de caso**. T.I.S, São Carlos, v. 4, n. 3, p. 209-215, set-dez 2015.

NADAREISHVILI Irakli, MITRA Ronnie, MCLARTY Matt, AMUNDSEN Mike. **Microservice Architecture: Aligning Principles, Practices, and Culture**. United States: O'ReillY Media, 2016.

NEWMAN, Sam. **Building Microservices**. United States: O'ReillY Media, 2015.

POSTA Christian. **Microservices for Java Developers**. United States: O'ReillY Media, 2016.

RICHARDSON, C. **Microservices: Decomposição de Aplicações para Implantação e Escalabilidade**, 2014. Disponível em: <http://www.infoq.com/br/articles/microservicesintro>. Acesso em: 01 de jul. 2015.

SCHROEDER, Andreas. **Microservice Architectures**. Deutschland: Codecentric, 2016.

VIANNA, Ilca Oliveira de Almeida. **Metodologia do Trabalho Científico: Um enfoque didático da produção científica.** São Paulo: EPU, 2001.