CENTRO UNIVERSITÁRIO DE ANÁPOLIS – UniEVANGÉLICA
BACHARELADO EM ENGENHARIA DE COMPUTAÇÃO

JOSÉ FRANCISCO DE OLIVEIRA JÚNIOR
MAYCON DA SILVA MOREIRA

ARQUITETURA DE MICROSSERVIÇO PARA INTEGRAÇÃO DE NOVOS SISTEMAS NOS CURSOS DE BACHARELADO EM COMPUTAÇÃO – UNIEVANGÉLICA.

ANÁPOLIS
2019

JOSÉ FRANCISCO DE OLIVEIRA JÚNIOR
MAYCON DA SILVA MOREIRA

ARQUITETURA DE MICROSSERVIÇO PARA INTEGRAÇÃO DOS NOVOS SISTEMAS NOS CURSOS DE BACHARELADO EM COMPUTAÇÃO – UniEVANGÉLICA.

Trabalho de Conclusão de Curso II apresentado como requisito parcial para a conclusão da disciplina de Trabalho de Conclusão de Curso II do curso de Bacharelado em Engenharia de Computação do Centro Universitário de Anápolis – UniEVANGÉLICA.

Orientador(a): Prof. Me. Millys Fabrielle.

ANÁPOLIS
2019

JOSÉ FRANCISCO DE OLIVEIRA JÚNIOR
MAYCON DA SILVA MOREIRA

ARQUITETURA DE MICROSSERVIÇO PARA INTEGRAÇÃO DE NOVOS SISTEMAS NOS CURSOS DE BACHARELADO EM COMPUTAÇÃO - UNIEVANGÉLICA

Trabalho de Conclusão de Curso II apresentado como requisito parcial para a obtenção de grau do curso de Bacharelado em Engenharia de Computação do Centro Universitário de Anápolis – UniEVANGÉLICA.

Aprovado(a) pela banca examinadora em 5 de junho de 2019, composta por:

Millys Fabrielle Araújo Carvalhaes
Presidente da Banca

Alexandre Moraes Tannus
Alexandre Moraes Tannus
Prof(a). Convidado(a)

Marcelo de Castro Cardoso
Marcelo de Castro Cardoso
Prof(a). Convidado(a)

Agradecimentos

Ao Criador dos céus e da terra, DEUS, Que além de nos formar, nos concede a capacidade para ir além, em busca dos nossos ideários. A ELE seja toda honra, glória e louvor, pois Dele, por Ele e para Ele são todas as coisas.

Á nossa nobre família que nos apoiaram em busca dos nossos ideários, nos incentivando e nos proporcionando todos os auxílios possíveis em todos os aspectos da nossa vida.

Eu Maycon também agradeço o meu irmão Filipe que ajudou grandemente no desenvolvimento desse projeto e a minha amada noiva, que nos momentos difíceis e corridos me deu grande apoio e incentivo para a conclusão deste trabalho de curso.

Eu José Francisco, agradeço aos meus pais, irmãos e amigos por apoiarem minhas boas decisões e me segurarem nas más, pela paciência, pelos conselhos, por estarem sempre ao meu lado, pelo companheirismo, pelas festas e pelas boas lembranças que tenho, estas pretendo multiplicar junto a vocês. A minha irmã Alice Maria, agradeço por estar comigo desde o início minha primeira e melhor amiga, que sempre me orientou, apoiou e cuidou de mim.

Agradecemos também ao nosso querido amigo Rafael Souza Oliveira, que nos ajudou neste projeto tanto teoricamente quanto tecnicamente compartilhando conosco suas experiências como arquiteto de software.

Ao nosso ilustre orientador, Prof. Me. Millys Fabrielle, que nos acolheu nesse trabalho tão complexo que demandou tempo e muito trabalho e que nos orientou no decorrer desse imenso projeto.

E a todos os docentes e técnicos administrativos do Centro Universitário de Anápolis UniEVANGÉLICA, que contribuíram direta e indiretamente no nosso percurso formativo, nos proporcionando um melhor aprendizado ao longo dessa caminhada.

Que DEUS abençoe a todos!

Resumo

Partindo da observação de vários projetos desenvolvidos nos cursos de bacharelado em computação, foi identificado que boa parte desses, são desenvolvidos por equipes, linguagens, arquiteturas e tecnologias diferentes, causando uma falta de padronização e consequentemente a dificuldade do gerenciamento entre os mesmos, já que não há comunicação entre estes projetos. O presente trabalho aborda a criação de uma arquitetura de microsserviços, cuja finalidade é padronizar os novos sistemas desenvolvidos para os cursos de bacharelado em computação através de sua utilização.

Abstract

Based on the observation of several projects developed in computing field (bachelor degree), it was identified that many of these are developed by different teams, languages, architectures and technologies, causing a lack of standardization and consequently the difficulty of management among them, since there is no communication between these projects. The present work deals with the creation of a microservice architecture, whose purpose is to standardize the new systems developed for computing bachelor degree, through its use.

Lista de Ilustrações

LISTA DE QUADROS

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
ESB	Enterprise Service Bus
HTTP	Hypertext Transfer Protocol
IP	Internet Protocol
JSON	JavaScript Object Notation
JWT	JSON Web Token
MVC	Model View Controller
MVP	Model View Presenter
MVVM	Model View View Model
POM	Project Object Model
REST	Representational State Transfer
SGBD	Sistema de Gestão de Base de Dados
SOA	Service-Oriented Architecture
SOAP	Simple Object Access Protocol
SQL	Structured Query Language
URL	Uniform Resource Locator
WAR	Web Application Archive
WEB	World Wide Web

Sumário

CAPÍTULO 1 - INTRODUÇÃO

O curso de Engenharia de Computação do Centro Universitário UniEVANGÉLICA, bem como diversos outros cursos, realiza tarefas manuais que podem ser automatizadas por meio de software. Entre essas tarefas estão, o controle de monitoria, planos de ensino, geração de horário, entre outros. Dessa maneira, acadêmicos propõem soluções para sanar essas demandas, tais como: o sistema para controle de monitoria, o sistema de Check-in cujo objetivo é controlar a frequência dos acadêmicos em eventos realizados pelo curso, entre outros projetos que poderão ser desenvolvidos.

Por outro lado, esses trabalhos são desenvolvidos por diferentes equipes e de forma isolada, cada qual usando tecnologias diferentes. Neste cenário, haverá um momento em que existirão diversos sistemas com funcionalidades específicas, tornando inviável o gerenciamento e manutenção dos mesmos. Entre algumas possíveis dificuldades de gerenciamento, pode ser citada a repetição de requisitos, tais como o de autenticação, além da falta de comunicação entre as aplicações.

Diante do cenário em que diversas atividades dos cursos de Bacharelado em Computação são solucionadas por aplicações de software distintas, as quais muitas das vezes possuem sobreposição de requisitos, tecnologias distintas, duplicidade de dados, entre outros, como a utilização da arquitetura de microsserviços possibilitará a criação de sistemas menores que funcionem de forma integrada na composição de um único sistema?

A contribuição desse estudo se dá devido a uma possível solução por meio da criação de uma arquitetura de software para a padronização de novos projetos. A definição de uma arquitetura de software adequada possibilitará a resolução de futuros problema, como por exemplo: a escolha de uma tecnologia para desenvolvimento sem analisar o contexto, o domínio e a adequação ao problema (VAROTO, 2002).

A arquitetura de software exerce um papel primordial na criação de um sistema, ela deve suportar os requisitos funcionais e principalmente atender aos requisitos não funcionais, os quais definem restrições e diversas características da aplicação, tais como: manutenibilidade, escalabilidade e interoperabilidade. (JAZAYERI ET AL apud VAROTO, 2002).

Uma das arquiteturas existentes que pode atender a essas especificidades é a de microsserviços, que é considerada como “serviços pequenos e autônomos que funcionam em conjunto.” (NEWMAN 2015 apud BACK, 2016, p.22). Stoiber (2017) destacou que:

> Um microsserviço é uma unidade única e independente que, juntamente com muitas outras, compõe uma grande aplicação. Ao dividir seu serviço 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.

Considerando o exposto, um dos motivos que justifica a utilização de microsserviços é sua independência, pois possibilita o desenvolvimento destes serviços em módulos, permitindo que diferentes equipes de desenvolvimento escolham a linguagem de programação e as tecnologias mais adequadas para cada necessidade.

Outro motivo que justificou a utilização da arquitetura de microsserviço foi através da revisão bibliográfica na monografia construída pelos autores: Carvalho; Anjos (2017). Da qual, aborda como temática a comparação entre a arquitetura monolítica e a de microsserviço. Nesse trabalho foi realizado a analise de testes em cima dessas arquiteturas tais como: teste de desempenho, de escalabilidade, interoperabilidade e modificabilidade, sendo viável a utilização da arquitetura de microsserviço nesse cenário.

Sendo assim o presente projeto justifica-se na utilização da arquitetura de microsserviços para ser usada como arquitetura referencial, para os novos aplicativos nos cursos de Bacharelado em Computação.

Esse estudo tem o objetivo de definir uma arquitetura de microsserviços para integração e padronização na construção de novos sistemas para os curso de Bacharelado em Computação do Centro Universitário UniEVANGÉLICA. Para a concretização desse objetivo é necessário seguir as etapas seguintes: Definir documento arquitetural, apresentando uma visão arquitetural abrangendo todos os aspectos envolvidos. Construir arquitetura de microsserviços para ser um modelo referencial para a construção de novos sistemas destinados aos cursos de bacharelado em computação da UniEVANGÉLICA, de forma integrada compondo um único sistema. Realizar a validação da arquitetura através de dois microsserviços.

1.1. Metodologia

O problema abordado nesse trabalho, foi identificado a partir da observação de vários projetos desenvolvidos nos cursos de bacharelado em Computação do Centro Universitário UniEVANGÉLICA.

Para a resolução da problematização encontrada, primeiramente foi realizada uma pesquisa bibliográfica qualitativa, sobre padrões arquiteturais e todos os aspectos que envolvem a arquitetura de microsserviços, por meio de livros, teses, dissertações e meios eletrônicos, que serviram como embasamento teórico para compreender detalhadamente a arquitetura e como deve ser trabalhada.

O próximo passo foi à elaboração do documento arquitetural, para isso foi usado um modelo, composto por objetivos, restrições, escopo da arquitetura, componentes, visões e seus artefatos. As visões foram apresentadas por meio do UML, tais como diagramas de componente, classe entre outros.

Para o desenvolvimento da arquitetura foi configurado o ambiente utilizando docker e docker compose. Posteriormente foi iniciado o projeto utilizando a linguagem de programação Java, juntamente com o *Framework Spring*, no qual, foram desenvolvidas as camadas composta por suas tecnologias, são elas: *Gateway* com Zuul, *Discovery* com Eureka, Auth Server com *Spring Securit*y, *Service* com Hystrix, além do banco de dados Mysql.

Para validar a arquitetura projetada foram desenvolvidos dois microsserviços dentro da mesma. Estes microsserviços foram desenvolvidos utilizando a linguagem de programação Java com o *Framework Spring* e com o banco de dados MySQL. O objetivo do protótipo é verificar se o modelo arquitetural criado conseguiu atender aos requisitos arquiteturais.

No último passo foi feita uma análise pelo método qualitativo do processo de desenvolvimento, para verificar o esforço depreendido na implementação da arquitetura, dos microsserviços e da comunicação entre eles, além das vantagens e desvantagens de se utilizar as tecnologias aplicadas durante o processo de implementação da arquitetura.

CAPÍTULO 2 - FUNDAMENTAÇÃO TEÓRICA: ARQUITETURA DE SOFTWARE

Neste capítulo é descrito a arquitetura de software, os padrões arquiteturais, as tecnologias utilizadas no projeto. Também são explanadas as definições, quando se aplica, estrutura arquitetural, além das vantagens e desvantagens de algumas arquiteturas de software existentes, são elas: monolítica, orientado a serviço e microsserviços.

2.1. Arquitetura de software

A arquitetura de software é formada por um conjunto de componentes computacionais, com características e relações, sendo fundamental na construção de um software. Sendo assim, Bass, Clements e Kazman (2003 apud FALBO, BARCELLOS 2011, p.85) descreve que a arquitetura de software são: "elementos de software, propriedades externamente visíveis desses elementos e os relacionamentos entre eles".

Para elaborar uma arquitetura é necessário conhecer as características do projeto, tais como os requisitos funcionais e não funcionais, para então decidir como a arquitetura será desenvolvida. Como descreve Krafzig, Banke e Slama (2004 apud OLIVEIRA, 2016, p.16):

> A arquitetura de software é um conjunto de declarações, que descreve os componentes de software e atribui funcionalidades de sistema para cada um deles. Ela descreve a estrutura técnica, limitações e características dos componentes, bem como as interfaces entre eles. A arquitetura é o esqueleto do sistema e, por isso, torna-se o plano de mais alto-nível da construção de cada novo sistema.

2.2. Padrões arquiteturais

A arquitetura de software é formada por vários elementos, dentre eles há o padrão arquitetural, que é um conjunto de modelos prontos que solucionam problemas arquiteturais comuns. O padrão arquitetural é uma maneira de exibir, partilhar e reusar o conhecimento.

Sommerville (2013, p.108) descreve que o padrão arquitetural é "uma forma de apresentar, compartilhar e reusar o conhecimento sobre sistemas de software". Há vários padrões arquiteturais existentes, são eles: Arquitetura em camadas, Cliente-servidor, MVC, MVP, MVVM, BROKER, Monolítico, arquitetura orientada a serviço, arquitetura orientada a Microsserviço entre outros. Serão descritos os padrões mencionados.

2.2.1 Arquitetura em camadas

Esta arquitetura é organizada em camadas, de modo que cada uma delas possui uma funcionalidade associada ao seu nível. Cada nível fornece serviços ao nível superior, sendo

assim, os inferiores representam os principais serviços, suscetíveis a serem usados em todo o sistema. (SOMMERVILLE, 2013).

Por ser modular, essa arquitetura permite a troca de uma camada específica, por outra equivalente que atenda os requisitos exigidos da camada removida, sem fazer necessário a trocas das demais. Além disso, ao se implementar novas funcionalidades, novas atualizações, apenas a camada adjacente será afetada (SOMMERVILLE, 2013).

Existem várias situações para o uso dessa arquitetura. Sugere-se que seja utilizada "na construção de novos recursos em cima de sistemas existentes; quando o desenvolvimento está espalhado por várias equipes, com a responsabilidade de cada equipe em uma camada de funcionalidade". (SOMMERVILLE, 2013, p. 110). O número de camadas existentes varia de acordo com a necessidade do projeto, podendo variar de duas camadas a n camadas.

A utilização desse tipo de arquitetura apresenta vantagens e desvantagens. Dentre suas vantagens, Sommerville (2013) destaca a troca de camadas inteiras desde que a interface seja mantida e a capacidade de se verificar a autenticação camada por camada para aumentar a confiança do sistema. Já dentre as desvantagens desta arquitetura, destaca-se a dificuldade da separação entre as camadas, e a possível necessidade de uma camada de alto nível precisar interagir com uma camada de baixo nível, ao invés de passar pela camada imediatamente inferior a ela. Ele também assinalou que o desempenho pode ser afetado pela existência de múltiplas camadas, onde em cada uma delas um serviço é processado.

2.2.2 Aplicações em uma camada

O modelo de uma camada mantém todos os recursos do sistema, que são banco de dados, regras de negócios e interfaces de usuário em computadores de grande porte. Os computadores conhecidos como mainframes tinham a responsabilidade de realizar todas as tarefas e processamento, e por isso, são considerados computadores de grande porte. Nestes sistemas, os terminais clientes eram conhecidos como terminais burros ou mudos, devido ao fato de não possuírem recursos de armazenamento ou processamento (GRANATYR, 2007).

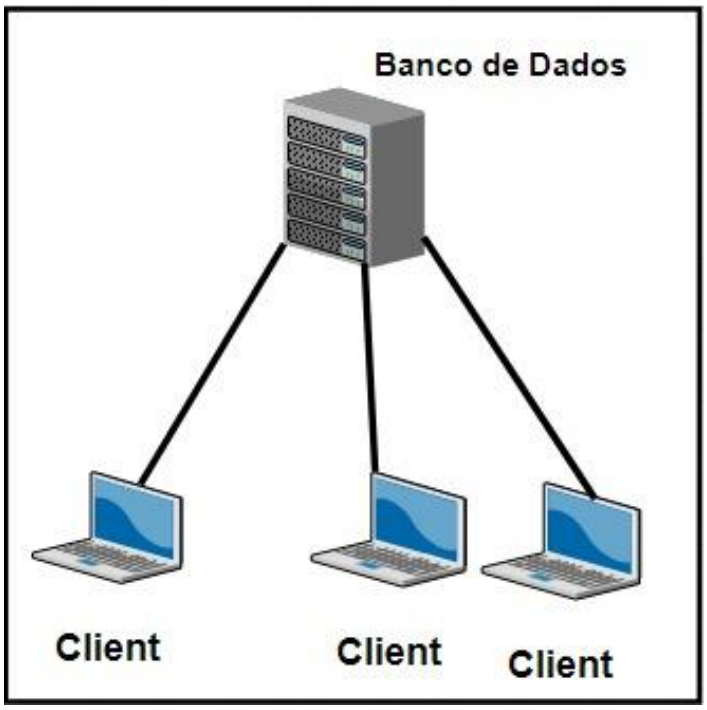
2.2.3 Aplicações em duas camadas

O uso do modelo de uma camada deu espaço ao modelo de duas camadas, pois com o passar do tempo houve o aumento do uso de computadores pessoais, evolução da tecnologia e a expansão das redes de computadores, o que acabou ocasionando a descentralização dos sistemas onde apenas o servidor era responsável pelo processamento. Os microcomputadores

passaram então a assumir parte deste processamento, ficando dividido entre o servidor e as máquinas clientes (GRANATYR, 2007).

A Figura 01 mostra a arquitetura de duas camadas.

Figura 01- A Organização da arquitetura de duas camadas



FONTE: Adaptado de GRANATYR (2007)

No modelo de duas camadas temos as regras de negócio no programa instalado no cliente. Este programa possui as regras de acesso ao banco de dados residente em um servidor de banco de dados. Do lado do cliente fica a apresentação, que é a interface visível do programa e onde o usuário acessa e interage com sistema, através dos formulários, menus e demais elementos visuais.

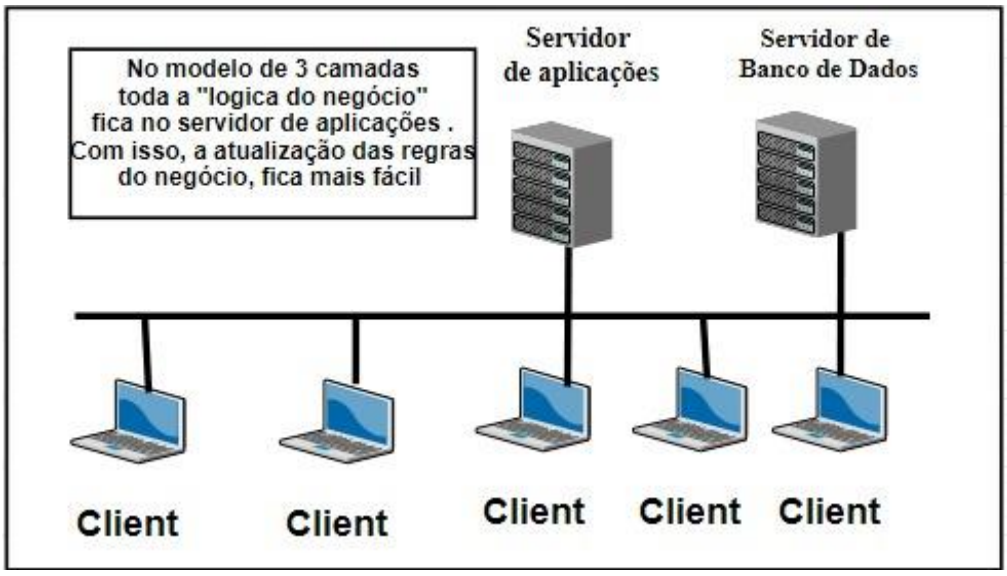
2.2.4 Aplicações em três camadas

O modelo de três camadas veio como forma de evolução do modelo de duas camadas, a diferença mais notável em comparação com o modelo de duas camadas foi a "retirada" das regras de negócio da aplicação do cliente, centralizando-a em um servidor de aplicações, deixando mais fácil a atualização das regras de negocio. Nesse modelo o desenvolvimento é um pouco mais demorado no início se comparado com o modelo de duas camadas, pois se faz necessário um maior suporte à camada adicionada. Porém, tem-se um ganho de desempenho em suas requisições (ALBUQUERQUE, 2012).

O cliente somente tem acesso à camada de apresentação. Ela faz requisições ao servidor de aplicações, e o servidor acessa o banco de dados de acordo com as regras contidas nele. (Idem, 2012).

A Figura 02 mostra a arquitetura de três camadas.

Figura 02- A Organização da arquitetura de três camadas



FONTE: Adaptado de ALBUQUERQUE (2012).

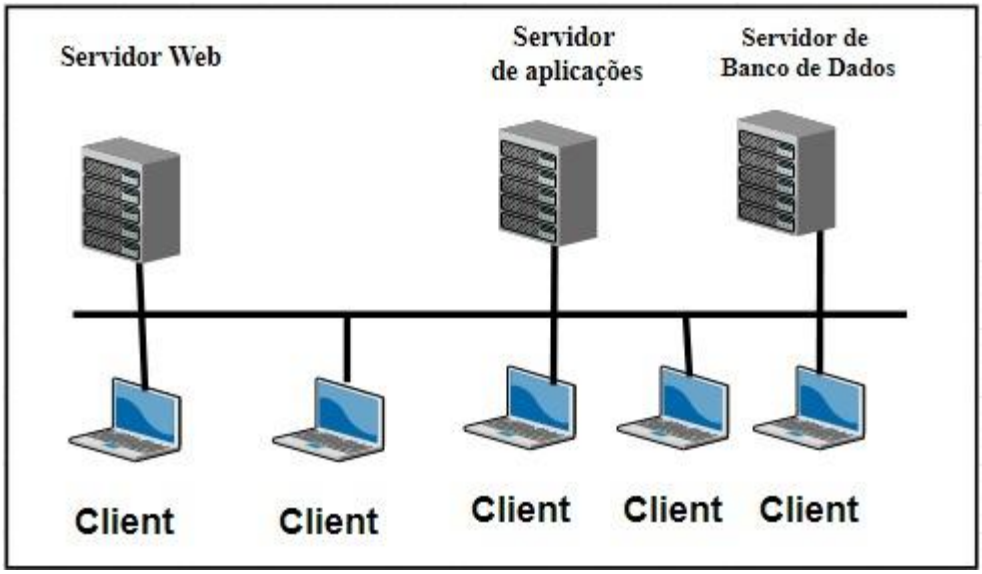
2.2.5 Aplicações em quatro camadas

Surge o modelo de quatro camadas como uma evolução do modelo de três camadas. Uma das maiores dificuldades do modelo de três camadas era a atualização da aplicação do lado do cliente, porém, com o modelo de quatro camadas, essa dificuldade pôde ser resolvida. Nos modelos de duas e de três camadas, havia a necessidade de atualizar a aplicação em centenas ou milhares de computadores. Agora, as atualizações são simples e fáceis, sendo necessário realizar as atualizações apenas nos servidores. (ALBUQUERQUE, 2012).

Isso foi possível devido ao fato dos clientes não terem uma aplicação instalada em seus computadores. Agora o acesso à aplicação faz-se através de um navegador web, como Chrome, Firefox e muitos outros.

A Figura 03 mostra a arquitetura de quatro camadas.

Figura 03- A Organização da arquitetura de quatro camadas



FONTE: Adaptado de ALBUQUERQUE (2012).

O acesso do cliente a aplicação se faz por meio dos navegadores com o cliente acessando o endereço da aplicação. Desse modo, assim como o padrão em três camadas, o cliente não acessa diretamente o banco de dados, o que faz com que com seja acrescentada mais segurança a integridade do sistema (ALBUQUERQUE, 2012).

A camada de apresentação, anteriormente mantida no cliente, agora permanece em servidores Web. Este servidor acessa o servidor de aplicações e este acessa o banco de dados. O cliente passa a ser o navegador *Web*, onde o usuário acessa a aplicação (Idem, 2012). Entretanto, fisicamente um único servidor pode desempenhar o papel de servidor Web, de aplicações e de banco de dados e, nesse caso, ainda segundo o autor, devem-se considerar as questões de desempenho, segurança, entre outras (Idem, 2012).

2.2.6 Cliente-servidor

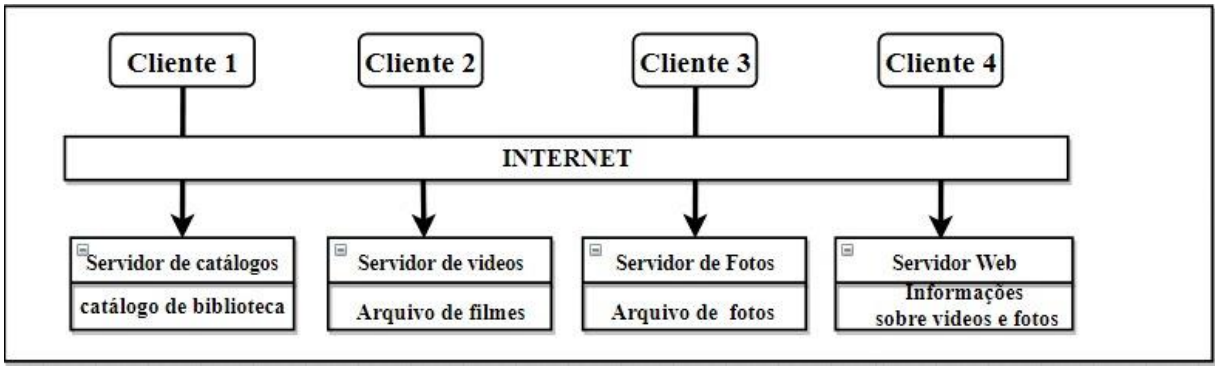
A arquitetura cliente-servidor foi desenvolvida utilizando o padrão de duas camadas, em que um cliente solicita um recurso a uma rede de servidores e algum dos servidores disponíveis o responde diretamente. Tem como base a descentralização de dados e recursos de processamento. Esse é um dos motivos para que ela seja muito utilizada em sistemas distribuídos. (ALBUQUERQUE, 2012).

A arquitetura cliente-servidor, segundo Sommerville (2013), possui três componentes principais, são eles os clientes, os quais podem chamar os serviços e consumir seus recursos oferecidos pelos servidores. O conjunto de servidores, o qual é os responsáveis por disponibilizar os serviços a outros componentes. E a rede de computadores que é a

responsável por ligar cliente e servidor para que haja a comunicação entre eles. Por sua vez os clientes podem saber os nomes dos servidores a serem acessados caso necessitem, no entanto os servidores não necessitam conhecer a identidade e nem a quantidade de seus clientes.

A organização da arquitetura cliente-servidor pode ser observada na figura 04.

Figura 04 - A Organização da arquitetura cliente-servidor



FONTE: Adaptado de SOMMERVILLE (2013).

A figura 04 mostra uma arquitetura cliente-servidor para uma biblioteca de filmes. Ela exemplifica a distribuição dos servidores, dos clientes e da rede responsável pela comunicação entre eles.

Na escolha dessa arquitetura, deve-se analisar as necessidades do projeto e se atende a maioria delas, pois há situações em que se adequa perfeitamente e outras nem tanto. A arquitetura cliente-servidor deve ser utilizada "quando os dados em um banco de dados compartilhado precisam ser acessados a partir de uma série de locais. Como os servidores podem ser replicados, também pode ser usado quando a carga em um sistema é variável." (SOMMERVILLE, 2013, p.113).

Existem diversas vantagens e desvantagens de se utilizar a arquitetura cliente-servidor. Dentre elas, Sommerville (2013, p. 113) destaca a seguinte vantagem como sendo a principal:

Os servidores podem ser distribuídos através de uma rede. A funcionalidade geral (por exemplo, um serviço de impressão) pode estar disponível para todos os clientes e não precisa ser implementada por todos os serviços.

Em contrapartida, também apresenta as seguintes desvantagens:

> Cada serviço é um ponto único de falha suscetível a ataques de negação de serviço ou de falha do servidor. O desempenho, bem como o sistema, pode ser imprevisível pois depende da rede. Pode haver problemas de gerenciamento se os servidores forem propriedade de diferentes organizações. (SOMMERVILLE, 2013, p.113).

2.2.7 Model View Controller

A arquitetura MVC do inglês *Model-View-Controller* que tem a tradução como Modelo-Visa-Controlador. É considerada como um padrão arquitetural que visa separar a exibição e a interação dos dados sistemáticos (dados dos sistemas). Nesse contexto, Sommerville (2013, p.109) descreve:

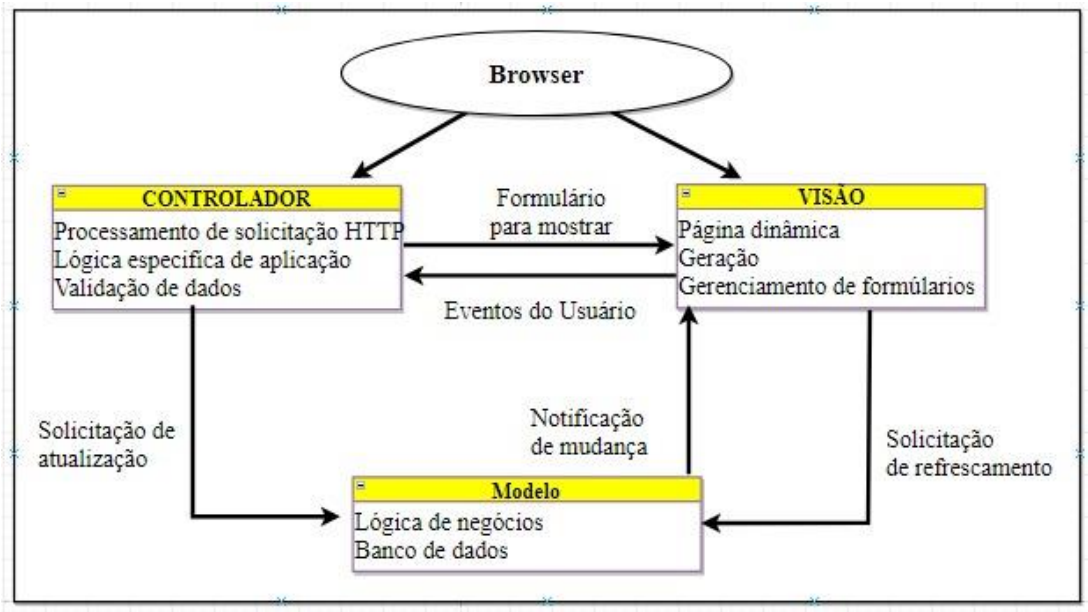
> O sistema é estruturado em três componentes lógicos que interagem entre si. O componente Modelo gerencia o sistema de dados e as operações associadas a esses dados. O componente Visão define e gerencia como os dados são apresentados ao usuário. O componente Controlador gerencia a interação do usuário (por exemplo, teclas, cliques do mouse etc.) e passa essas interações para a Visão e o Modelo.

Conforme o autor mencionado anteriormente, nota-se que a organização do MVC acontece mediante os três componentes lógicos (Visão, Controlador e Modelo), cada qual interagindo entre si e tendo atributos específicos dentro do sistema.

- *View* ou visão: É a camada que é mostrada no navegador, na qual o usuário se interage, enviando dados e visualizando as respostas geradas pelo sistema.
- *Model* ou modelo: É a camada que contém toda a lógica do sistema, por meio dela é realizada a conexão com a base de dados, a regra de negócio e a manipulação dos dados do banco.
- *Controller* ou Controlador: Ele controla e mapeia as requisições entre a *View* e o *Model*: é o intermediador entre essas duas camadas.

Para exemplificar esse tipo de padrão arquitetural, Sommerville (2013) expõe uma figura que demonstra a estrutura de um aplicativo que tem como base a internet, organizado pelo uso do padrão MVC. Exemplo disso pode ser observado na figura 05.

Figura 05 – Arquitetura e Aplicações Web usando o padrão MVC



FONTE: Adaptado de SOMMERVILLE (2013).

Esse modelo de padrão arquitetural é mais utilizado em desenvolvimento web, pois por ser independente, obtém-se mais desempenho, segurança e facilidade na manutenção da aplicação. Está presente em várias estruturas como *Ruby on Rails*, *Laravel*, *Django*, ASP.NET MVC entre outros (MACORATTI, 2011). Segundo Sommerviller (2013) o MVC também é usado "quando os futuros requisitos de interação e apresentação de dados são desconhecidos".

Segundo Sommerville (2013, p.109) podemos alencar como vantagem: "Permite que os dados sejam alterados de forma independente de sua representação, e vice-versa. Apoia a apresentação dos mesmos dados de maneiras diferentes, com as alterações feitas em uma representação aparecendo em todas elas."

E como desvantagens os seguintes aspectos: quando o modelo de dados e as interações são simples, haverá envolvimento de código adicional e até mesmo uma complexidade maior de código (SOMMERVILLE, 2013).

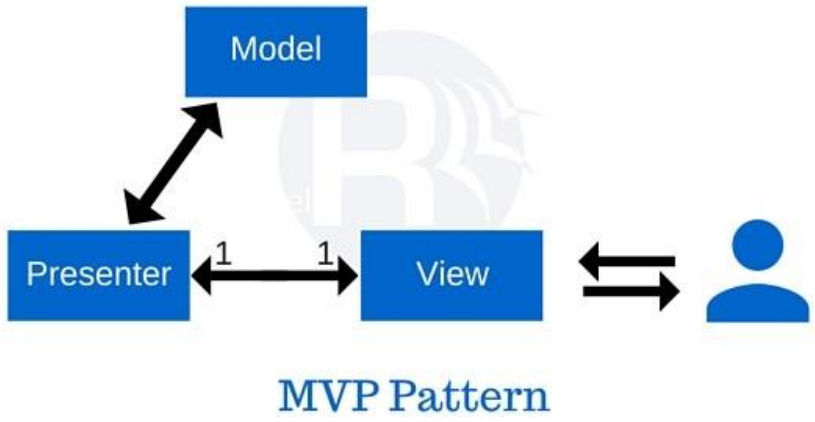
2.2.8 Model View Presenter

O MVP é semelhante ao MVC, porém o *Presenter* substitui o *Controller*, e a *View* precisa do *Presenter* para conseguir acesso aos dados dos modelos do sistema. (FILHO, 2017)

> [...] o Presenter tem mais responsabilidades que o Controller, pois todo o acesso aos dados e a lógica de negócios da aplicação são realizados através dele, seja do Model em relação à View (uma notificação web) ou da View em relação ao Model (o clique de um botão). (RISHABH, 2016 apud FILHO, 2017, p.26).

Na figura 06 demonstra os componentes e suas ligações dentro do padrão MVP.

Figura 06 - A Organização do MVP



FONTE: (RISHABH, 2016).

Segue respectivamente abaixo os componentes e suas definições que foram citados por Filho (2017, p.26) mediante Rishabh (2016):

- *View*: A *View* integra todos os elementos das camadas gráficas do sistema, interfaces, transições, efeitos, eventos e interações. Neste padrão a *View* só envia ou recebe comandos a partir do *Presenter*.
- *Model*: Assim como no MVC, o *Model* contém as regras de negócio e a lógica do sistema. Para ter acesso a *View*, só é possível por meio de um mediador que é o *Presenter*.
- *Presenter*: É responsável por controlar a interação entre a *View* e o *Model*, o mesmo trata as ações da *View*, ficando a cargo de decidir quando e como ela será atualizada. Além do acesso aos dados e da lógica de negócio sendo realizado através dele, seja do *Model* em relação à *View* ou da *View* em relação ao *Model*.

Ele é bastante utilizado na construção de interfaces com usuário. Pois foi projetado para melhorar o mecanismo de programação modular e orientada a objetos. Rishabh afirma que "O MVP é usado principalmente para aplicativos *ASP.NET Web Forms*". (Idem, 2016).

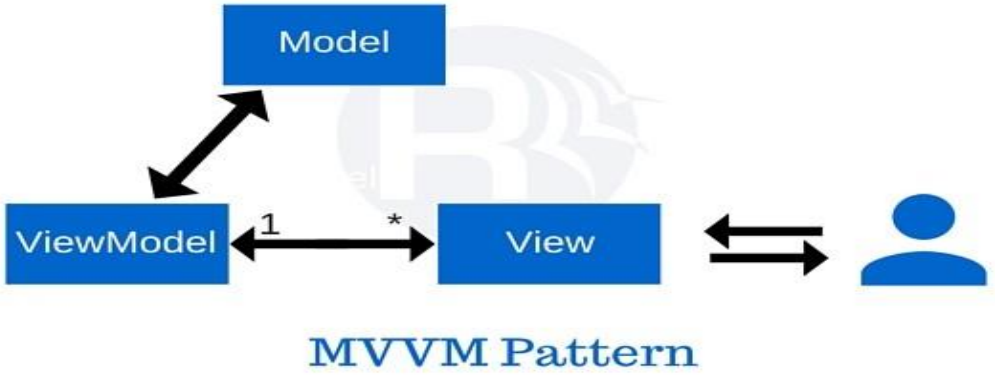
2.2.9 Model View View Model

O model View View Model (MVVM) é bem similar com os últimos dois padrões mencionados, nele a *View* se comunica com o *ViewModel* e vice versa, permitindo que alterações sejam atualizadas em tempo real. Normalmente para se conectar e informar alterações, entre o *ViewModel* e o *Model*, é utilizado o padrão observador. Este padrão

arquitetural é usado em sistemas como: Windows *Presentation Foudation*, *Caliburn*, *Silverlight* e *nRoute*. (RISHABH, 2016)

A figura 07 demonstra como o padrão arquitetural MVVM é organizado.

Figura 07 - A Organização do MVVM



FONTE: (RISHABH, 2016).

Rishabh (2016) menciona a definição dos componentes do padrão exposto, são eles:

- Modelo: Representa uma coleção de classes que explica o modelo de negócios e o modelo de dados. Define as regras de negócios para dados, ou seja, como os dados podem ser alterados e manipulados.
- Visão: Representa os componentes da interface do usuário, como CSS, *jQuery*, *HTML*, etc. O modo exibir, mostra os dados que são recebidos do controlador como o resultado, isso também altera o (s) modelo (s) na interface do usuário.
- *ViewModel*: O modelo de visualização é responsável por exibir métodos, comandos e outras funções que auxiliam na manutenção do estado da visualização, manipulando o Modelo como resultado de ações na visualização e acionando os eventos na própria visualização.

O MVVM traz consigo todas as vantagens de ser divido em camadas, por exemplo, a manutenção se torna fácil, pois alterações podem ser feitas sem se preocupar com outras camadas, a testabilidade se torna mais simples já que as dependências estarão em pedaços separados de código. (MICROSOFT, 2017)

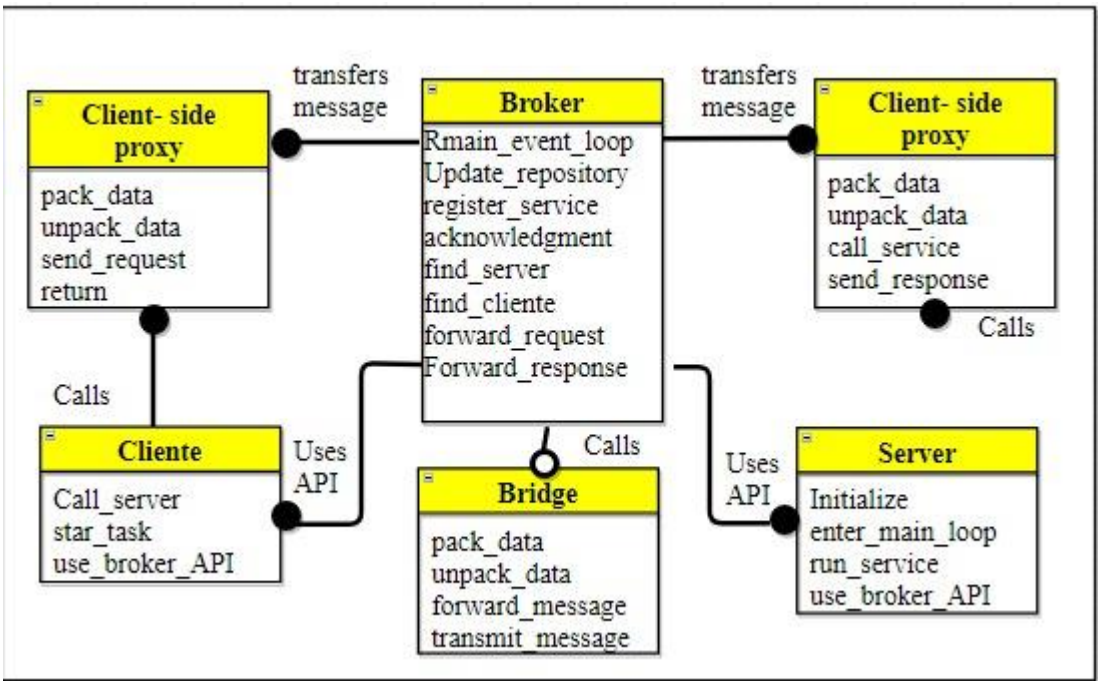
Algumas desvantagens de usar o padrão MVVM, são elas: aplicar em sistemas muito simples se torna um exagero, ou usar o MVVM em aplicações maiores, ou seja, aplicações complexas, pois torna difícil a implementação do ViewModel. (Idem, 2017)

2.2.10 Broker

É utilizado em ambiente distribuído cujos componentes são desacoplados e se comunicam através de invocação remota de serviço. O Broker é responsável por coordenar a comunicação entre o cliente e o servidor (VAROTO, 2002).

Na figura 08 demonstra os componentes e suas ligações no padrão arquitetural Broker, cuja estrutura é separada em seis componentes: Clientes são aplicações que acessam os serviços de pelo menos um servidor. Cliente-Proxy é a camada que interliga o cliente e o Broker, que esconde os detalhes de implementação do cliente. O Broker é o mensageiro que transmite as requisições entre o cliente e o servidor. Os Bridges escondem os detalhes de implementação quando dois brokers interoperam. O Servidor-Proxy é responsável por receber requisições, desempacotar mensagens e chamar serviços apropriados. E por ultimo o servidor que possui objetos que expõem suas funcionalidades através de interfaces com operações e atributos. (VAROTO, 2002)

Figura 08 - A Organização do BROKER



FONTE: Adaptado de BUSCHMANN (2001).

Varoto descreve as seguintes vantagens sobre o padrão arquitetural Broker:

> Transparência para o cliente na localização do servidor. Facilidade de troca e evolução de componentes uma vez que suas interfaces não mudem. Portabilidade em função do Broker. Interoperabilidade entre diferentes Broker. Reuso dos serviços já existentes quando da construção de novas aplicações cliente. (VAROTO, 2002, p.29).

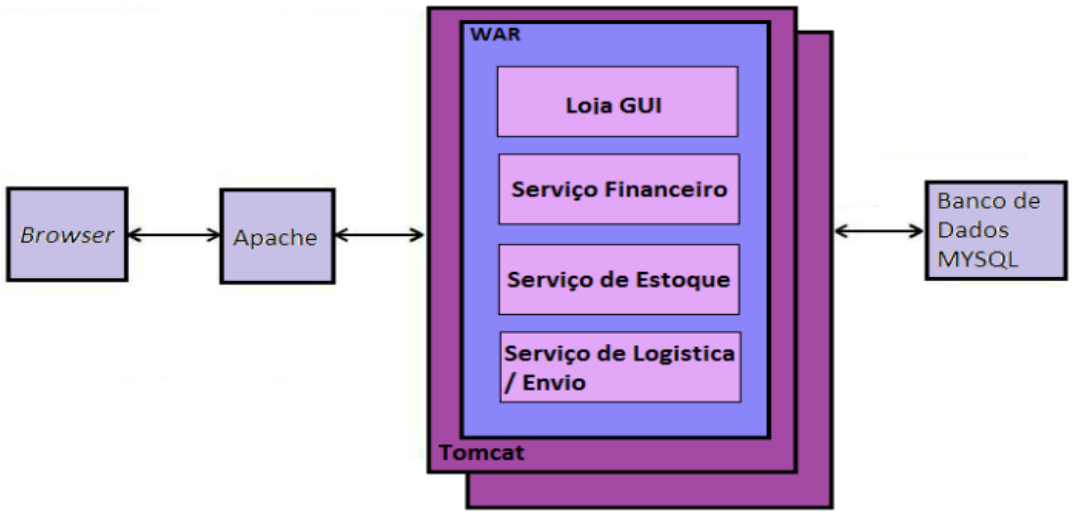
O autor ainda cita as desvantagens do padrão Broker, são elas: Eficiência restrita uma vez que os servidores são localizados dinamicamente; Baixa tolerância a falhas; Teste e debug são dificultados por envolver muitos componentes. (VAROTO, 2002, p.29)

2.3 Arquitetura monolítica

A arquitetura monolítica é o padrão arquitetural mais usado para o desenvolvimento de aplicações corporativas. Uma aplicação monolítica é um bloco no qual contem toda a sua base de código, onde são definidas todas as funcionalidades do sistema (CARVALHO, 2017).

Nesse sentido a figura 09 demonstra uma aplicação Java Web utilizando a arquitetura monolítica, tal sistema consiste em apenas um único arquivo *Web Application Archive* (WAR) sendo executado em um container web como o *Tomcat* (TRIPOLI; CARVALHO, 2016).

Figura 09 – Arquitetura tradicional de uma aplicação Web



FONTE: (TRIPOLI; CARVALHO, 2016).

Podem-se citar algumas vantagens de se utilizar a arquitetura monolítica, como por exemplo, a simplicidade da arquitetura por não conter muitas camadas. A implementação em apenas uma linguagem de programação, o que facilita o desenvolvimento e a coesão entre a equipe. A facilidade no desenvolvimento e nos testes, já que as funcionalidades estão localizadas em uma única unidade de implantação. (CARVALHO, 2017, p.21)

O autor afirma ainda que “Quando uma aplicação monolítica se torna grande, surgem às desvantagens dessa arquitetura”. Tais como: a enorme quantidade de linhas de código, o que dificulta a compreensão do código, principalmente quanto a novos desenvolvedores. A dificuldade em realizar testes, pois tudo se torna lento e pesado, fazendo com que haja uma diminuição de produção do desenvolvedor. Dificulta a entrega contínua, já que uma grande

aplicação monolítica também é um obstáculo para *deploys* frequentes, pois para atualizar um componente é necessário reimplantar o aplicativo inteiro. Além da limitação de adoção de novas tecnologias, pois uma aplicação que foi projetada e desenvolvida com certas tecnologias, linguagem de programação e banco de dados adequado à época, deverá continuar na mesma pilha de tecnologia, independente do que seja mais adequado.

2.4 Arquitetura orientada a serviço

Além da arquitetura monolítica descrita acima, existe a arquitetura orientada a serviço (SOA) que é descrita por Sommerville (2013, p.356) como:

> [..] uma forma de desenvolvimento de sistemas distribuídos em que os componentes de sistema são serviços autônomos, executando em computadores geograficamente distribuídos. Protocolos-padrão baseados em XML, SOAP e WSDL foram projetados para oferecer suporte à comunicação de serviço e à troca de informações. Consequentemente, os serviços são plataforma e implementação independentes de linguagem. Os sistemas de software podem ser construídos pela composição de serviços locais e serviços externos de provedores diferentes, com interação perfeita entre os serviços no sistema.

A arquitetura orientada a serviços permite que os diversos serviços integrados, sejam implementados em diferentes linguagens de programação. Isso permite uma maior flexibilidade para o desenvolvimento de cada serviço. Assim cada serviço pode ser implementado na linguagem mais adequada de acordo com seus requisitos. (OLIVEIRA, 2013).

Estes serviços implementados em diferentes linguagens proporciona uma maior interoperabilidade ao sistema, o que poderia ser um problema para a comunicação entre os serviços. Porém, cada serviço possui uma interface exposta para tais comunicações, que são invocadas via mensagens. Estas interfaces são responsáveis por disponibilizar os recursos oferecidos por cada serviço sem que seja necessário o conhecimento da implementação de cada serviço (OLIVEIRA, 2013).

Para a disponibilização e comunicação de serviços do SOA, existe um importante componente chamado *Enterprise Service Bus* (ESB), que disponibiliza os serviços em um local único e centralizado. Isto possibilita o reuso dos serviços e baixa o acoplamento entre os consumidores e provedores de serviços (OLIVEIRA, 2013).

O ESB ele é um importante componente responsável por muitas funções de SOA, mas não todas, dentre outras responsabilidades pode se citar: o suporte a web services: ele tem a capacidade de invocar web services baseados em WSDL e SOAP dentre outros. Roteamento: os barramentos disponibilizam diferentes formas de realizar roteamento de mensagens,

roteamento baseado em um serviço de regras e roteamento baseado em políticas. Orquestração: muitos ESB realizam orquestração através de um serviço de proxy, o qual coordena a execução de múltiplos serviços. Alguns barramentos delegam a orquestração para motores BPEL. Dentre outras responsabilidades (SANTANA, 2017).

Essa arquitetura possui uma grande capacidade de reuso, aumentando a produtividade. Segundo Oliveira (2013) também destaca como vantagens a: flexibilidade, manutenibilidade, interoperabilidade, integração, governança, disponibilidade, segurança, entre outros. O autor mencionado anteriormente também cita as desvantagens, são elas: complexidade no gerenciamento, desempenho, robustez, disponibilidade, testabilidade, segurança entre outros.

2.4.1 Web Service

Web Service é uma arquitetura que faz a comunicação entre aplicações diferentes, usada na integração de sistemas. Utilizando Web Service é possível fazer com que aplicações antigas comuniquem com novas aplicações, eles são componentes que permitem às aplicações enviar e receber dados. Cada aplicação pode ter a sua própria "linguagem", que é traduzida para uma linguagem universal, um formato intermediário como XML, Json, CSV, entre outros (FERREIRA, MOTA, 2014). Com a Web Service surgiram algumas padronizações estipuladas, as duas de maior destaque são o protocolo SOAP e o modelo de design REST, as quais serão descritas nos próximos tópicos.

2.4.2 Protocolo Simple Object Access Protocol

O Simple Object Access Protocol mais conhecido como (SOAP), é um protocolo feito para facilitar a chamada remota de funções via internet, permitindo que dois programas se comuniquem. Os pedidos SOAP podem ser feitos em três padrões, sendo eles GET, POST e SOAP, os pedidos do SOAP parte do princípio da utilização de XMLs para a transferência de objetos entre aplicações, e a utilização, como transporte, do protocolo de rede HTTP (SANT’ANNA, 2015).

O autor exposto também cita que o padrão Web Service Description Language (WSDL) que é definido pelo SOAP, descreve perfeitamente os objetos e métodos que estão disponíveis, através de paginas XML acessíveis por intermédio da WEB (SANT’ANNA, 2015).

2.4.3 Arquitetura REST

O REST é uma arquitetura criada para ser simples de se usar, ele é aplicado na integração de *Web Services* e também é responsável por fazer a comunicação entre microsserviços. Com a utilização deste protocolo há um ganho em agilidade, flexibilidade e praticidade (LACERDA, 2018).

O REST pode ser usado em vários formatos de texto, como por exemplo, o JSON que significa *JavaScript Object Notation*, ele é uma formatação leve de troca de dados, Porém, só pode ser utilizado com o protocolo HTTP/HTTPS, utilizando os métodos GET, POST, PUT, DELETE dentre outros para as requisições (ARMIGLIATTO, 2017).

O REST tem por base cinco princípios fundamentais, que os seguindo consegue explorar a arquitetura da Web em seu beneficio, são eles: Dê a todos os recursos um Identificador; Vincule os recursos; Utilize métodos padronizados; Recursos com múltiplas representações; Comunique sem estado.

Tilkov (2008) e Ferreira (2017) descrevem sobre os princípios fundamentais, sendo o primeiro: use URIs para identificar tudo o que precisar ser identificado, especifique todos os recursos de "alto nível" que seu aplicativo oferece, se eles representam itens individuais, conjuntos de itens, objetos virtuais e físicos, ou resultados de computação. Os benefícios de se ter um único esquema de nomes a nível global aplicável tanto para a Web em seu navegador como para comunicação de máquina para máquina (TILKOV, 2008).

O segundo princípio: Usar links para referenciar os recursos que possam ser identificados sempre que for possível. Hiperlinks são o que fazem a Web ser a Web (TILKOV, 2008).

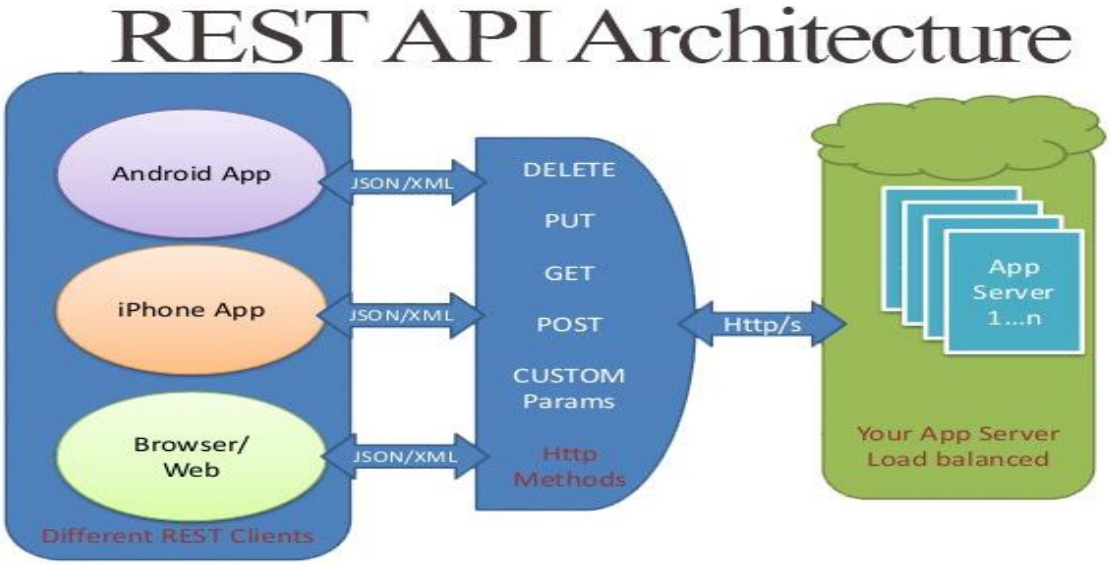
O terceiro princípio: Para que clientes possam interagir com seus recursos, eles devem implementar o protocolo de aplicação padrão (HTTP) corretamente, isto é, utilizar os métodos padrão: GET, PUT, POST e DELETE (TILKOV, 2008).

O quarto princípio: Ofereça diversos formatos dos recursos para diferentes necessidades, ou seja, este pode retornar um XML ou um JSON dependendo da necessidade do cliente (TILKOV, 2008).

O quinto princípio é a comunicação sem estado: As requisições feitas por um cliente a um serviço REST devem conter todas as informações necessárias para que o servidor as interprete e as execute corretamente. Qualquer informação de estado deve ser mantida pelo cliente e não pelo servidor. Isso reduz a necessidade de grandes quantidades de recursos físicos, como memória e disco, e também melhora a escalabilidade de um serviço REST (FERREIRA, 2017).

A figura 10 demonstra um diagrama API REST, onde tem três clientes, sendo eles o Android, o iPhone e o Browser. Nesse caso é utilizado o API REST para buscar ou fornecer informações de um serviço da web, utilizando o JSON para transportar essas informações entre o App Server e os clientes.

Figura 10 – Arquitetura REST API



FONTE: JAMES, Geordy, 2017.

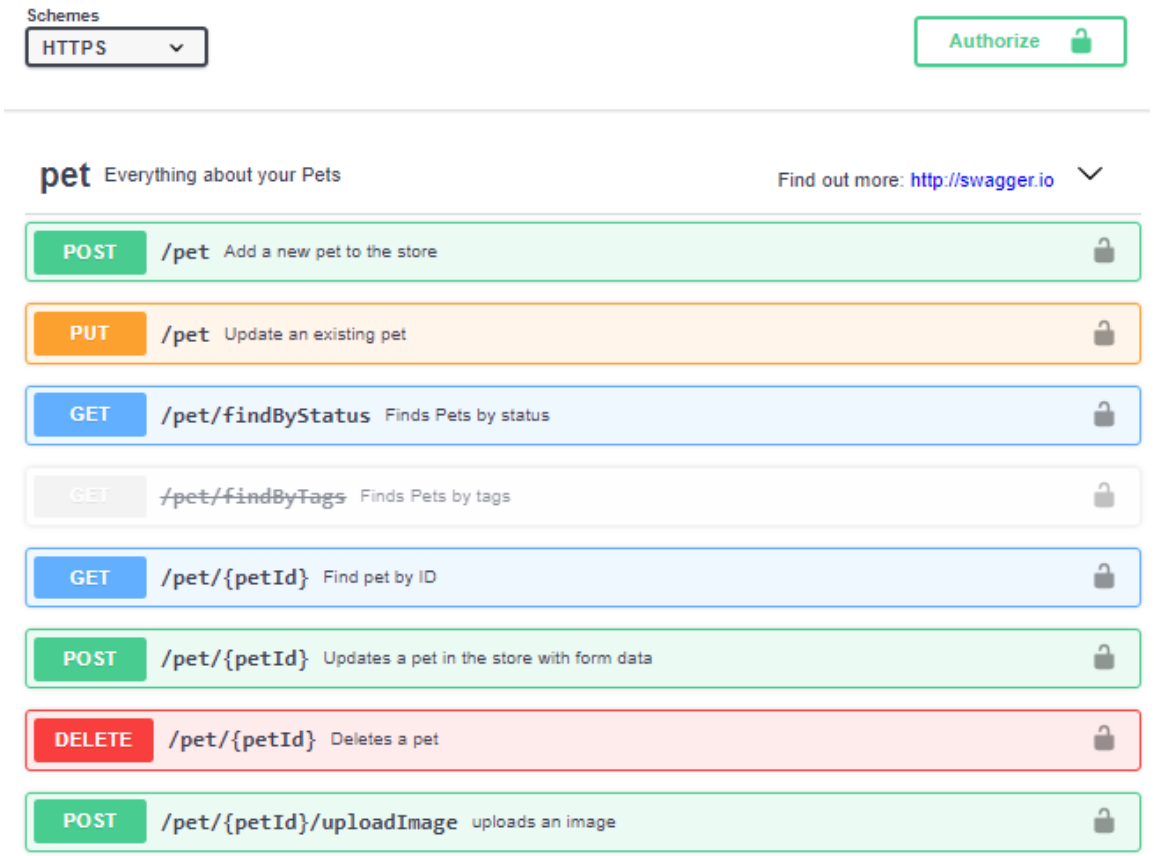
2.4.4 Swagger

O *Swagger* é um software *open-source* poderoso e fácil de usar, ele possui varias ferramentas que apoiam os desenvolvedores durante o ciclo de vida da API, como o suporte para documentação automatizada (SWAGGER, 2018).

Com ele é possível criar a documentação de três formas diferentes, sendo elas: manual, que é escrito as especificações dos serviços pelo desenvolvedor. Automática, quando é criado a API e é gerada a documentação, e por meio do Codegen que converte as anotações do *swagger* contidas no código fonte das APIs REST em documentação. (COSTA, 2016)

A figura 11 exemplifica a documentação da API usando o *swagger*, onde mostra na integra os pontos de extremidades e descreve as operações permitidas em cada uma delas.

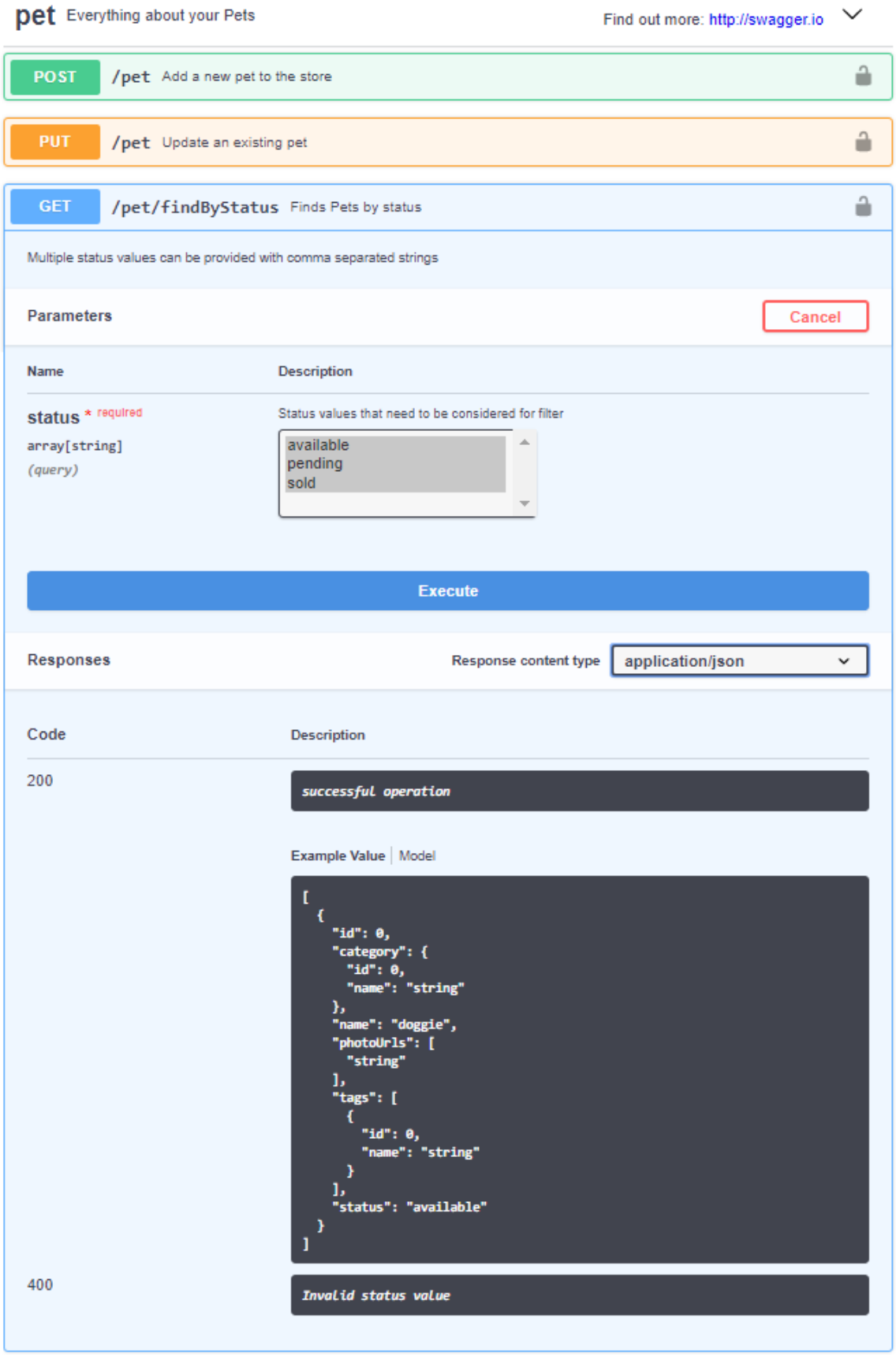
Figura 11 – Exemplo Swagger Documentation



FONTE: EDITOR SWAGGER, 2019.

Na figura 12 é demonstrado o ponto de extremidade chamado findByStatus, na documentação é possível verificar os parâmetros, o tipo de resposta que nesse caso é JSON, o exemplo JSON e os código de estado do HTTP.

Figura 12 – Exemplo do ponto de extremidade findByStatus



FONTE: EDITOR SWAGGER, 2019.

2.5 Arquitetura de microsserviços

A arquitetura de microsserviços é o principal foco desse projeto, esta é compreendida quando o sistema é dividido em serviços menores e autônomos, capazes de executar em seu próprio espaço de memória e dimensionar com autonomia um dos outros em diversas maquinas desassociada uma das outras (ELLIOTT, 2015).

Segundo Fowler (2014) essa nomenclatura surgiu com o intuito de descrever uma forma particular de projetar aplicativos de software como suítes de serviços implementáveis de forma independente. Uma das empresas que foi pioneira na utilização de microsserviços, foi a Netflix, cuja necessidade a levou a adoção desse modelo para o desenvolvimento de software.

A arquitetura de microsserviços surgiu com o objetivo de solucionar alguns problemas de complexidade nos softwares, ocasionados pelo crescimento do sistema ao longo do tempo. Tais como: escalabilidade, a falta de flexibilidade, alta dependência, e as dificuldades para fazer alterações de código.

A arquitetura de microsserviços tem como uma de suas características a autonomia e independência de serviços, possuindo um ciclo de vida independente. Podendo ser implementado por equipes diferentes, escolhendo separadamente a melhor tecnologia para cada microsserviços. Sendo assim, Newman (2015) afirma que “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.”.

Tripoli (2016) cita as vantagens e desvantagens da arquitetura de microsserviços, dentre as vantagens estão: Escalabilidade; Isolamento de falhas; Implantação de serviço individual; Código pequeno com limites bem definidos; Flexibilidade para escolher as melhores linguagens e tecnologias; Desenvolvimento independente, ciclos de desenvolvimento e implementação e iteração de recursos mais rápida; Menos caminho de resistência para adotar uma nova tecnologia no futuro;

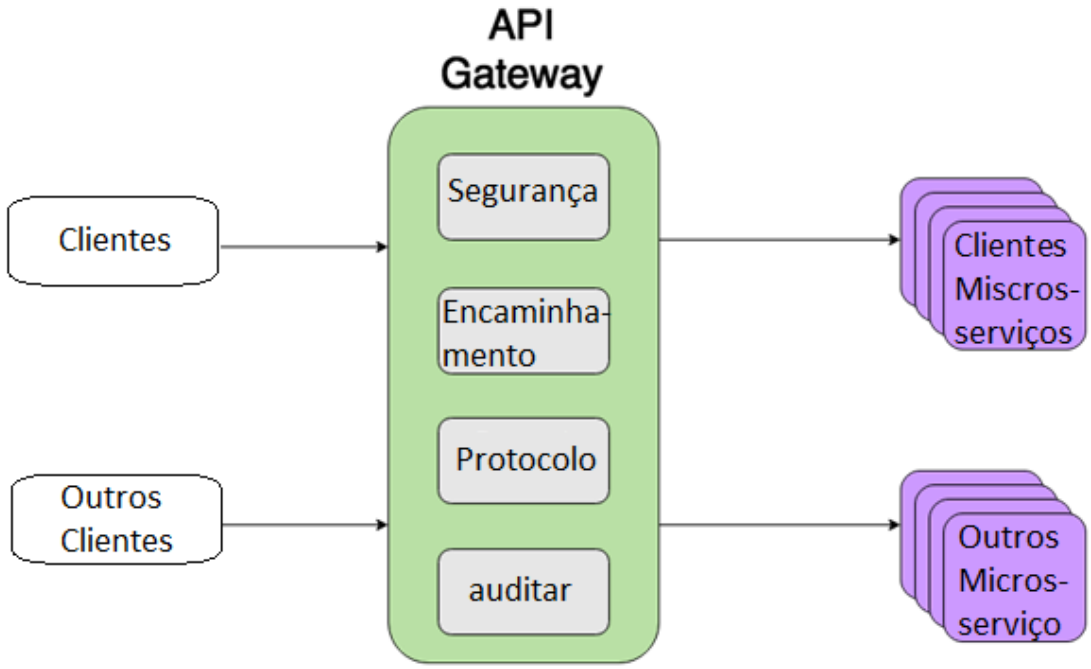
Todavia, como toda arquitetura, ela também possui suas desvantagens, Tripoli (2016) menciona algumas delas, tais como: Desenvolvedores precisam de habilidades de *DevOps* substanciais; Sobrecarga de operações significativas; Chamadas remotas são mais caras do que as chamadas de processos internos; Consumo de memória geralmente aumenta; Desafios de testabilidade; Fica mais complexo gerenciar por ter muitos microsserviços;

A arquitetura de microsserviços é composta por varias camadas, onde cada uma é um serviço, entre elas podemos citar a camada Gateway, Discovery, Auth Server, além do banco de dados, abaixo será explicado o que é cada uma delas.

2.5.1 Gateway

A camada *Gateway* é a porta de entrada única para o sistema, ele gerencia as requisições, roteando-as para o serviço de *back–end* apropriado, ou seja, podem-se ter vários microsserviços, porém ao utilizar o *gateway* existirá apenas uma única URL, como é demonstrado na figura 13. O *gateway* também faz requisições de vários serviços do *back-end*, e depois juntam os resultados (LUKYANCHIKOV, 2016).

Figura 13 – Estrutura do Gateway



FONTE: Produção Própria dos Autores

Segundo Lukyanchikov (2016) o *Gateway* "pode ser usado de diversas formas, como: autenticação, insights, testes de estresse, migração de serviço, tratamento de resposta estática, gerenciamento de tráfego ativo".

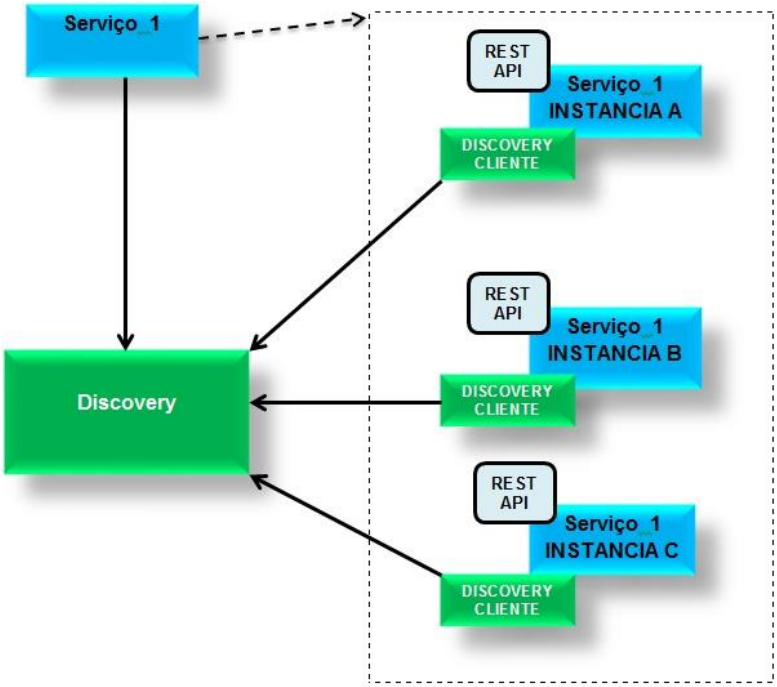
2.5.2 Discovery

A descoberta de serviço *discovery*, é responsável por registrar todos os microsserviços. Quando se usa nuvem, os serviços podem ser instanciados em locais de rede dinamicamente, seja por falhas, reinicialização ou escalonamentos automáticos. O *Discovery* resolve esse problema permitindo a detecção automática de locais de rede para instâncias de serviço. (LUKYANCHIKOV, 2016).

A figura 14 mostra um exemplo usando a camada Discovery, na qual sempre que um serviço é executado ele se auto registra no Discovery, deixando-o a par da localização de cada

seviço/instancia. Na imagem temos o Serviço_1 e suas varias instancias dinâmicas tais como a instancia A, instancia B e a instancia C, essas possuem um IP, Porta e outras informações referentes à sua localidade. Vale ressaltar que está sendo utilizado o REST via protocolo HTTP para a comunicação entre estes serviços.

Figura 14 – Estrutura usando discovery



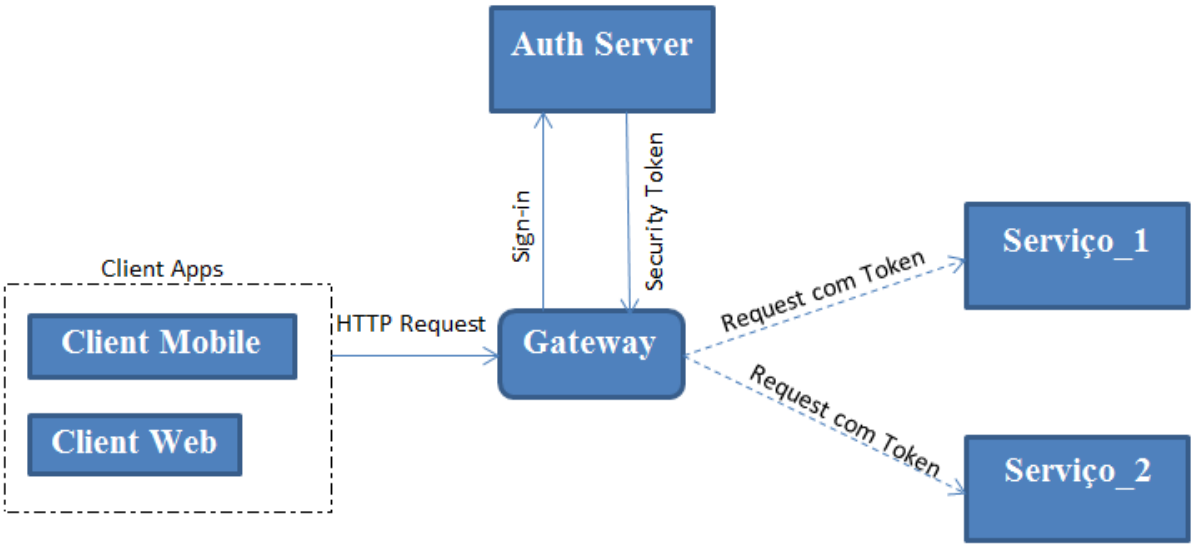
FONTE: Produção Própria dos Autores

2.5.3 Auth Server

O servidor de autenticação é uma camada separada, todas as responsabilidades de autorização são redirecionadas pra ele, que concede *tokens* de autorização para serviços de recursos de *back-end*, esses *tokens* são fornecidos pelo serviço OAuth2. O servidor de autenticação é usado para autorização do usuário, bem como para comunicação máquina a máquina segura dentro de um perímetro. (LUKYANCHIKOV, 2016)

É possível observar no diagrama da figura 15 o fluxo com o Auth Server, a sequencia começa com os Clients Apps enviando uma requisição para o gateway para acessar um determinado serviço, o gateway vai até o Auth Server e verifica se o Client é autentico e se ele tem permissão para acessar o serviço solicitado, se caso houver a permissão, o Auth Server retorna para o Gateway um Security Token e o gateway envia a requisição com o token para o serviço solicitado.

Figura 15 – Estrutura usando Auth Server



FONTE: Produção Própria dos Autores

2.5.4 Banco de dados

O banco de dados é um sistema de registro de dados, sua funcionalidade é armazenar os dados de forma que o usuário busque, atualize ou apague os dados registrados nele (MATIOLI, 2010).

Para acessar o banco de dados existem sistemas chamados de sistemas de gerenciamento de banco de dados (SGBD), este possui um conjunto de requisitos e funcionalidades como segurança, integridade, controle de concorrência e recuperação/tolerância a falhas. Dentre os mais populares SGBDs existentes, existem o SQL, MySQL, PostgreSQL dentre outros (OLIBONI, 2016).

O MySQL é um sistema de gerenciamento de banco de dados que utiliza a linguagem SQL, este é atualmente um dos sistemas de gerenciamento de banco de dados mais populares. Esse SGBD é conhecido por possuir integração com varias linguagens de programação existente, além de suportar praticamente qualquer plataforma atual (PIZA, 2012). O autor mencionado anteriormente também cita algumas vantagens, que são: portabilidade, compatibilidade, desempenho, facilidade em manuseio, suporte a triggers, é um software livre, além de varias outras vantagens. (PIZA, 2012).

2.6 Quadro comparativo de vantagens e desvantagens entre arquiteturas

Com o intuito de simplificar a descrição das arquiteturas, foi construída uma tabela com base nos autores Carvalho (2017); Oliveira (2013); Tripoli (2016) e na NBR ISO/IEC 9126-1

(2003) que compara as características, vantagens e desvantagens das arquiteturas descritas no projeto em sistemas de grande porte: arquitetura monolítica, orientada a serviço e de microsserviços. Como pode ser observado no quadro 01 a seguir.

Quadro 01 – Comparação das arquiteturas de Software

Características	**Descrição**	**Arquiteturas**		
		Monolítico	**Orientado a serviço**	**Microsserviços**
Disponibilidade	Capacidade para executar uma função requisitada num dado momento.	3	2	2
Interoperabilidade	Capacidade de interagir com um ou mais sistemas especificados.		3	3
Manutenabilidade	Capacidade do produto de software de ser modificado.	1	3	3
Escalabilidade	Capacidade de aumentar as funcionalidades de um software, aumentar o consumo de memória, de processos, atender demandas de alta exigência de processamento e possivelmente concorrência de acesso	2	2	3
Flexibilidade	Capacidade para poder escolher as melhores linguagens e tecnologias		3	3
Testabilidade	Capacidade de permitir que o software, quando modificado, seja validado.	1	2	2
Segurança	Capacidade do produto de software de proteger informações e dados	2	2	2
Resiliência	Capacidade te tolerar falhas			3
Classificação de vantagens:				
Classificação 1	Possui vantagem em ambiente não tão complexo.			
Classificação 2	Vantagem parcial.			
Classificação 3	Vantagem integral.			

FONTE: Produção Própria dos autores

CAPÍTULO 3 - FUNDAMENTAÇÃO TEÓRICA: TECNOLOGIAS DE SOFTWARES

Neste capítulo é descrito sobre as tecnologias que compõem este trabalho, é explanado sobre suas definições, vantagens e desvantagens além de suas estruturas.

3.1 Java

Existem diversas linguagens de programação que dão suporte a arquitetura de microsserviços como, por exemplo, JAVA, PHP e ASP. Dentre elas destaca-se o JAVA que é uma linguagem de programação e plataforma computacional, utilizada para o desenvolvimento web, foi lançada pela primeira vez pela Sun Microsystems em 1995.

Java é uma linguagem orientada a objetos e sua característica mais forte, é que o código compilado não é nativo da plataforma. Este é executado por uma máquina virtual, o que permite aos desenvolvedores criarem um programa uma única vez, podendo ser executado em qualquer uma das plataformas suportadas pela tecnologia, por sua vez ele é rápido, seguro e confiável. (FERNÁNDEZ, 1998)

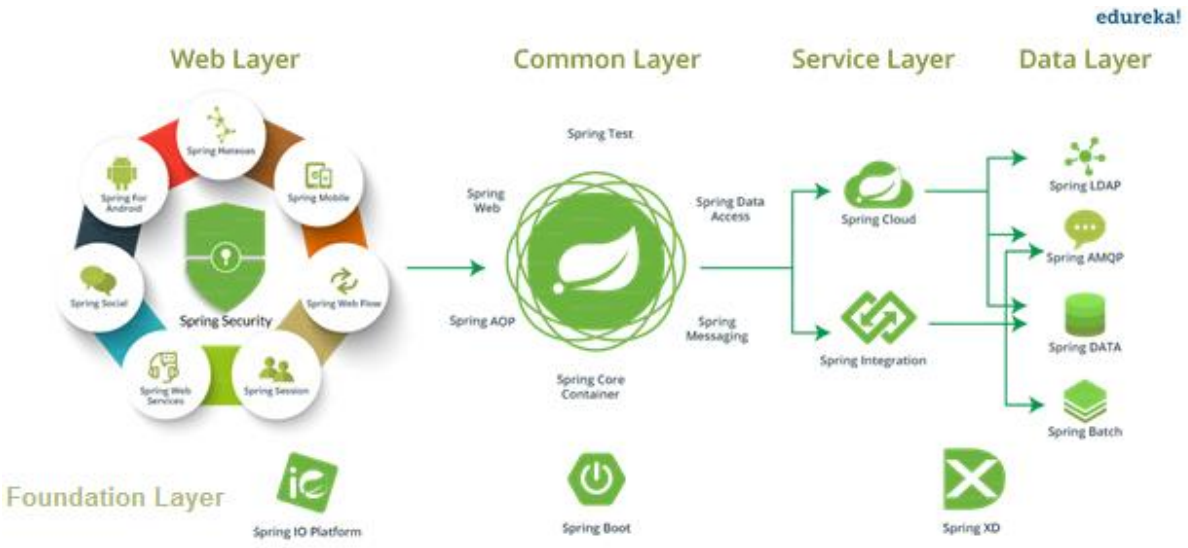
3.2 Spring

O *Framework Spring* foi desenvolvido para a plataforma Java, que tem como objetivo facilitar o desenvolvimento, explorando, para isso, os conceitos de Inversão de Controle e Injeção de Dependências. Essa tecnologia fornece não apenas recursos necessários à grande parte das aplicações, como também um conceito a seguir que permite criar soluções menos acopladas, mais coesas e, consequentemente, mais fáceis de compreender e manter (SOUZA, 2018). O Spring é composto por micro frameworks como, por exemplo, o Spring Boot e o Spring Cloud.

Abaixo a figura 16 mostra o ecossistema do framework Spring, onde ele é composto por projetos diferentes que por sua vez possuem subprojetos, além de mostrar as cinco camadas que dividem o spring, sendo elas a web layer, common layer, service layer, data layer, além da quinta parte que são os três projetos na parte inferior da imagem.

Dentre os projetos mostrados na figura, serão explicados dois deles, que são o spring boot e o spring cloud.

Figura 16 – Ecossistema do Spring Framework



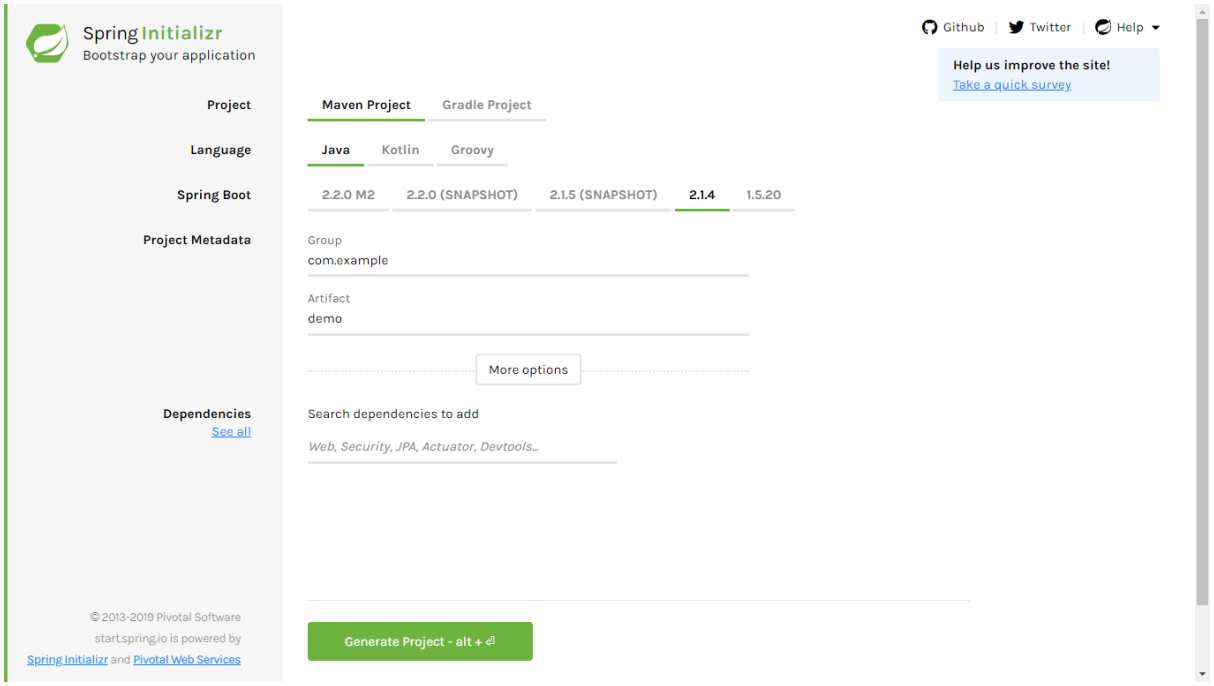
FONTE**:** Adaptado de CHAND, 2018.

3.2.1 Spring Boot

O Spring Boot foi projetado para colocar o projeto em operação o mais rápido possível, com configuração inicial mínima do Spring. Nele é possível selecionar as tecnologias que farão parte do projeto, onde elas já veem com a configuração inicial pronta (SPRING, 2014).

Uma das maneiras de iniciar o projeto usando o spring boot é pelo link <https://start.spring.io/> ao acessar o inicializador, é mostrado uma interface onde é possível configurar e selecionar as dependências necessárias para o projeto de acordo com suas necessidades, a figura 17 mostra a pagina inicial do inicializador de projetos spring boot.

Figura 17 – Tela inicial do Spring Initializr



FONTE: Spring Initializr, 2019

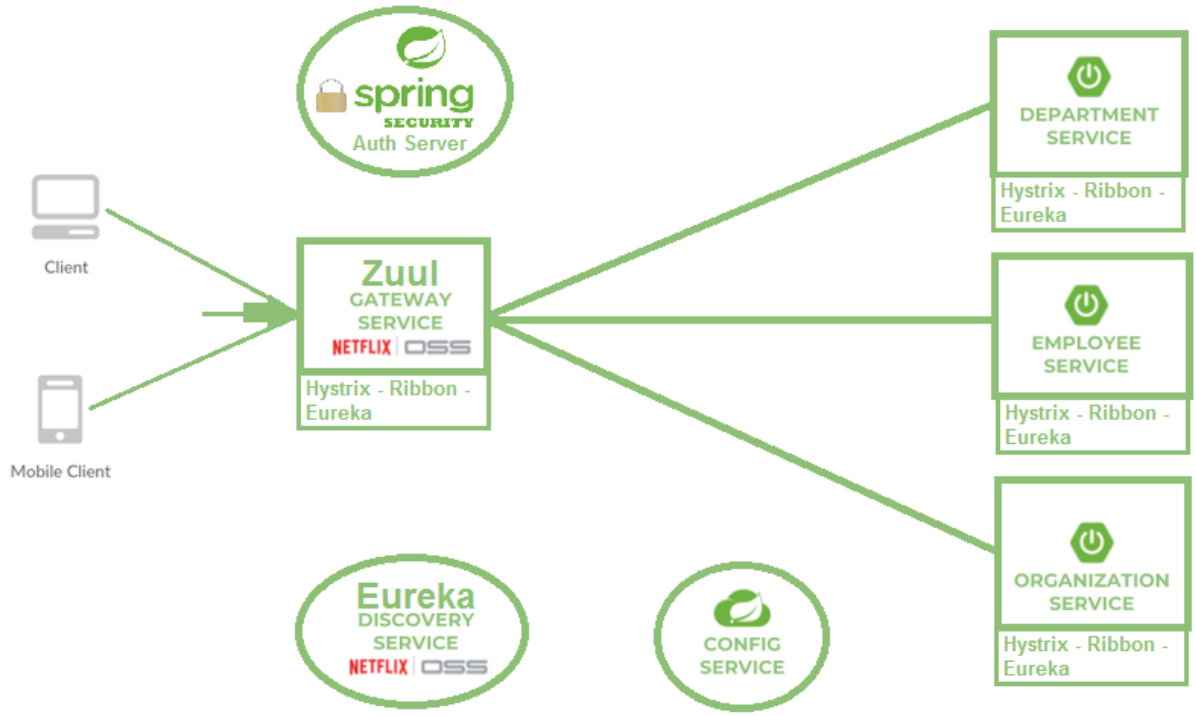
3.2.2 Spring Cloud

O Spring Cloud foi criado diretamente na abordagem inovadora do Spring Boot para o Java corporativo, o Spring Cloud simplifica a arquitetura distribuída no estilo de microsserviços implementando padrões comprovados para trazer resiliência, confiabilidade e coordenação aos seus microsserviços (SPRING, 2016).

O Spring Cloud contém uma integração com o spring boot e o projeto Netflix OSS. O Netflix OSS é um conjunto de frameworks e bibliotecas que a Netflix criou para resolver problemas comuns em sistemas distribuídos em escala, entre esses conjuntos podemos citar o Eureka, Hystrix/Turbine, Zuul e o Ribbon. Com algumas notações simplificadas é possível ter alguns destes componentes sendo executados no projeto (SALERNO, 2017).

Na figura 18 é estruturada a arquitetura de microsserviços usando Spring Cloud e Netflix OSS, o diagrama mostra alguns dos componentes citados anteriormente, estes são detalhados nos tópicos seguintes.

Figura 18 – Estrutura de Microsserviços com Spring Cloud e Netflix OSS



FONTE**:** Produção Própria dos autores

3.2.3 Spring Cloud Zuul (Gateway)

Como explicado anteriormente, o gateway é a única porta de entrada do sistema. Uma das maneiras de implementar o Gateway é usando a tecnologia Zuul, é possível ativa-lo de forma rápida através da notação entre parêntese (@EnableZuulProxy). O Zuul permite roteamento dinâmico, monitoramento, resiliência e segurança (LUKYANCHIKOV, 2016).

O Zuul usa vários filtros diferentes que permitem aplicar de forma rápida funcionalidades ao gateway, eles ajudam a executar funções como, por exemplo: autenticação e segurança, na qual identifica os requisitos de autenticação para cada recurso e rejeita solicitações que não os satisfazem; Roteamento dinâmico, este direciona dinamicamente as solicitações para diferentes clusters de back-end, conforme necessário; Rejeição de Carga faz a alocação de capacidade para cada tipo de solicitação e elimina as solicitações que ultrapassam o limite (NETFLIX, 2018)

3.2.4 Spring Cloud Eureka (Discovery)

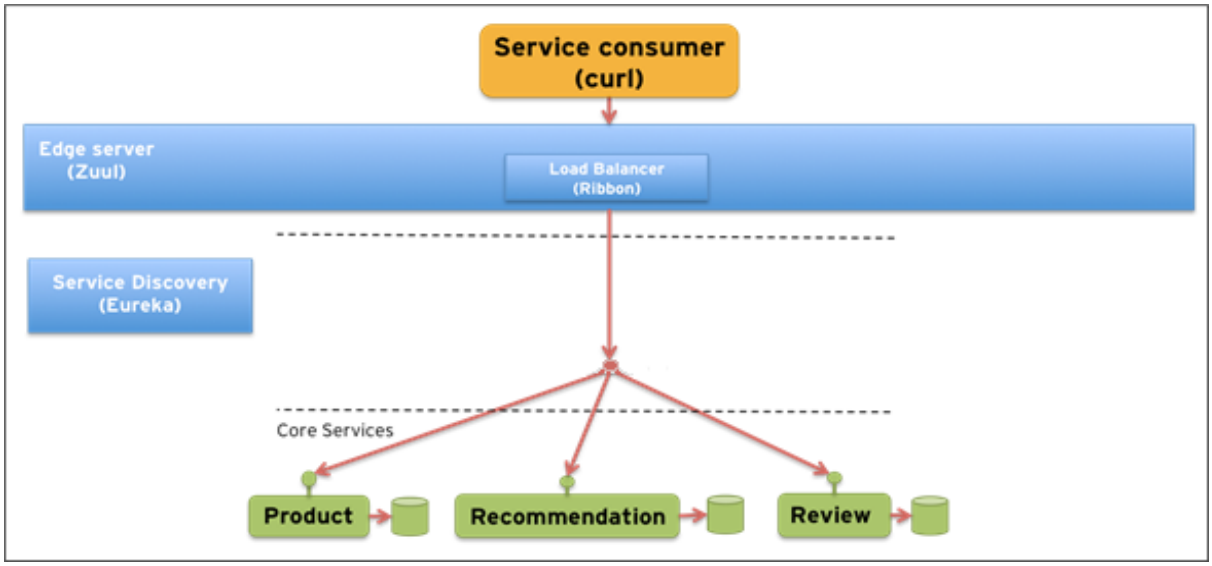
O Discovery é a camada que faz a localização de serviços além do balanceamento de carga. Uma tecnologia desenvolvida para facilitar a implementação dessa camada é o *Eureka*, ele permite que os microsserviços se registrem em tempo de execução, sempre que aparecem no campo de visão do sistema, quando o serviço se registra no Eureka, este fornece metadados sobre si mesmo, tais como o Host, a porta, a URL, dentre outros (LARSSON, 2015).

O Eureka é composto pelos módulos Eureka Server e o Eureka Client, o Eureka Server é um componente que permite que outros serviços se registrem nele, ele mantem esses registros atualizados e sinaliza quando um serviço não esta disponível, para ativar esse componente precisa usar a notação entre parêntese (@EnbaleEurekaServer). Já o componente Eureka Client permite que o serviço se registre no Eureka Server, para ativar o Eureka Cliente é necessário usar a notação entre parêntese (@EnbaleEurekaClient) (SOUZA, 2018).

É apresentado o diagrama na figura 19 contendo uma estrutura de microsserviços utilizando as tecnologias Eureka o Zuul, além de outra tecnologia chamada Ribbon.

Para que o serviço consumidor (*Service Consumer*) consiga acessar o serviço chamado produto (Product), primeiro é necessário passar pelo Gateway (Zuul), que centraliza a comunicação com todos os serviços, o *Gateway* acessa o *Discovery* (Eureka) no qual tem cadastrado todos os serviços, e questiona o endereço do serviço produto (*Product*), após o Gateway receber o endereço do serviço, ele acessará o serviço produto, e só então retornará ao *service consumer* o serviço solicitado. O *Ribbon* está conectado ao *Discovery*, e tem a função de balancear a carga que distribui as requisições uniformemente entre as instancias.

Figura 19 – Estrutura do Gateway com o Discovery



FONTE**:** Produção Própria dos autores

3.2.5 Spring Cloud Security (Auth Server)

A camada Auth Server ou também chamado de servidor de autenticação, é responsável pela parte de autenticação e autorização, é possível implementa-la utilizando o *Spring Cloud Security*, este torna aplicações e serviços mais seguros e com menos problemas casuais, pois se consegue facilmente fazer configurações externamente ou centralmente, estando apto para

implementação de grandes sistemas de componentes remotos cooperantes, que geralmente são serviços de autenticação de usuário.

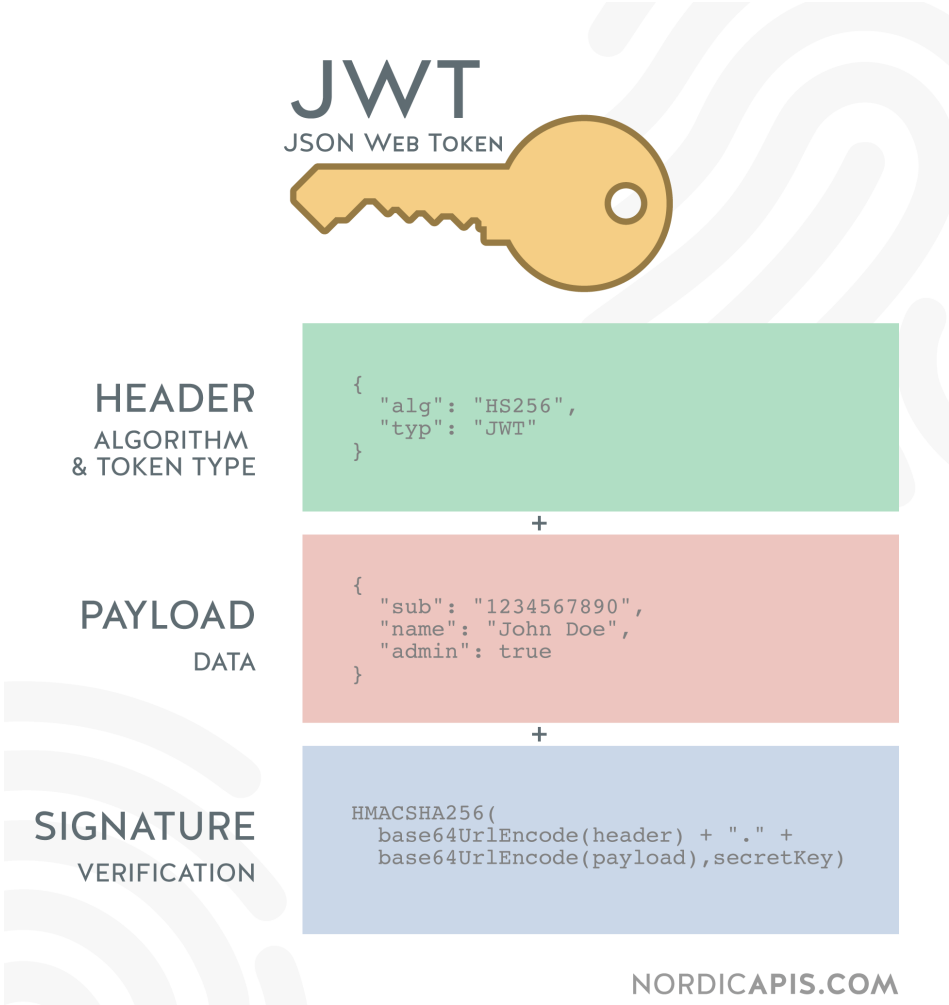
Com base no *Spring Boot* e no *Spring Security OAuth2*, este pode criar rapidamente sistemas que implementam padrões comuns, como logon único, relé de *token* e troca de *tokens*. O Token pode ser uma senha gerada randomicamente por um algoritmo, dentre os tokens mais usados se destaca o Json Web Token (SPRING, 2014).

O JSON Web Token (JWT) consegue transmitir com segurança informações entre as partes como um objeto JSON, de modo independente e compacto. Essas informações podem ser verificadas e confiáveis por serem assinadas digitalmente. As JWTs podem ser assinadas usando um segredo (com o algoritmo HMAC) ou um par de chaves pública / privada usando RSA ou ECDSA (JWT, 2014).

Em sua forma compacta o JSON Web Tokens consiste em três partes separadas por pontos, Que são: *Header*, *Payload* e *Signature*. O *Header* normalmente consiste em duas partes: o tipo do *token*, que é JWT, e o algoritmo de assinatura como HMAC SHA256 ou RSA. A segunda parte do *token* é a carga útil, que contém as reivindicações, elas são declarações sobre uma entidade (normalmente, o usuário) e dados adicionais. Existem três tipos de reivindicações, elas podem ser registradas, públicas e privadas. A ultima parte é assinatura, você precisa pegar o cabeçalho codificado, a carga codificada, um segredo, o algoritmo especificado no cabeçalho e sinalizar isso. A assinatura é usada para verificar se a mensagem não foi alterada ao longo do caminho e, no caso de *tokens* assinados com uma chave privada, ela também pode verificar se o remetente do JWT é quem diz ser (NASCIMENTO, 2018).

Na figura 20 exemplifica as partes que compõe o Token JWT como explicado no paragrafo anterior.

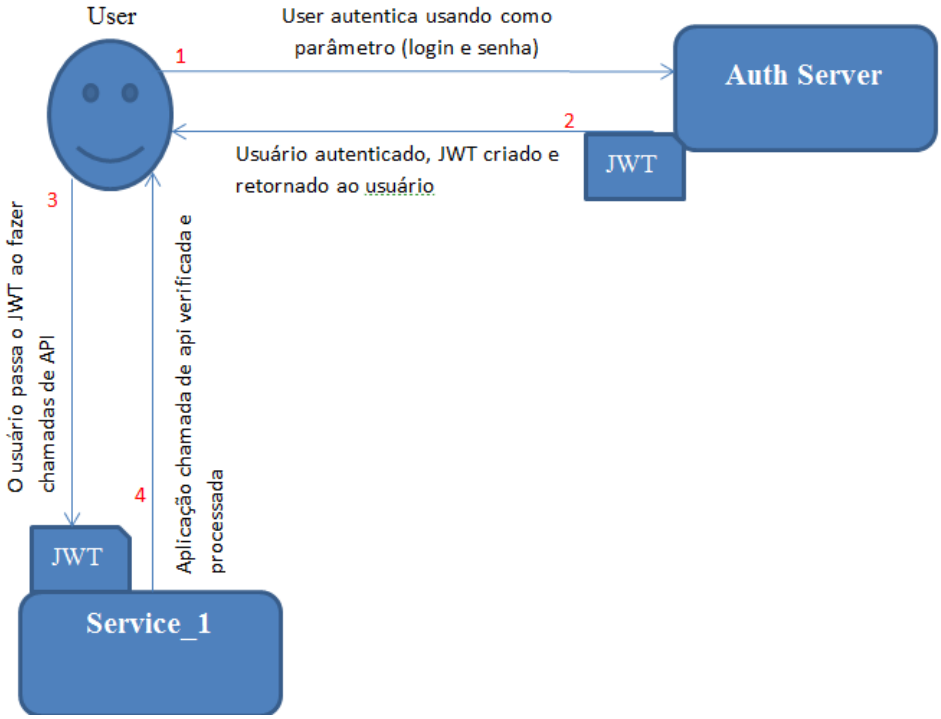
Figura 20 – Estrutura exemplificando as partes do JWT



FONTE: SANDOVAL, 2017.

Na figura 21 é possível observar um fluxo utilizando o JWT, onde o usuário se autentica no servidor de autenticação e o servidor gera e retorna um JWT, este é usado durante um limite de tempo para fazer as próximas requisições, como acessar o serviço service_1.

Figura 21 – Diagrama com o fluxo do JWT



FONTE**:** Produção Própria dos autores

3.2.6 Spring Cloud Ribbon (Balanceador de carga)

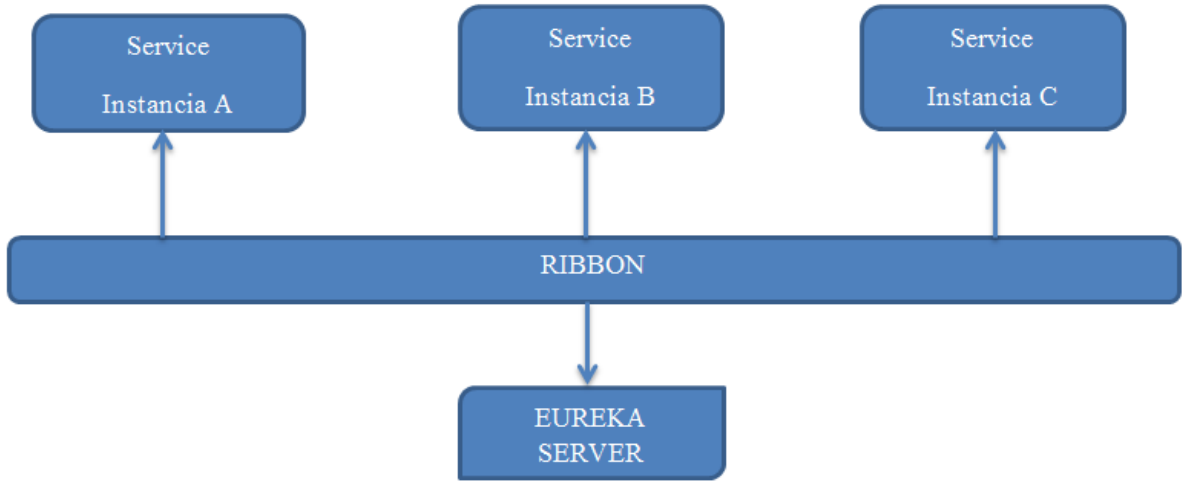
Em uma aplicação web que possui um grande trafego, normalmente é utilizado instancias de serviços para conseguir atender a demanda. É nessa hora que o balanceador de carga entra em ação, este é um serviço que tem a função de balancear a carga uniformemente entre as instancias.

O *Ribbon* é um balanceador de carga do lado do cliente, que permite ter o controle sobre o comportamento dos clientes HTTP além do TCP. É permitido o acesso diretamente com o serviço desejado. Ele se integra nativamente com o *Spring Cloud Eureka*. O *Eureka Cli*ent fornece uma lista dinâmica de servidores disponíveis para que a Ribbon possa equilibrar entre eles (LUKYANCHIKOV, 2016).

Uma vantagem de se utilizar o *Ribbon* é que tem a possibilidade de controlar o balanceamento de carga programaticamente, pois a faixa de opções fornece esse recuso (MITRA, 2017).

Na figura 22 é possível ver como o Ribbon atua, este fica posicionado entre o Eureka Server e as instancias dos serviços, fazendo o balanceamento de carga.

Figura 22 – Exemplo utilizando Ribbon



FONTE**:** Produção Própria dos autores

3.2.7 Spring Cloud Hystrix (Circuit breaker)

O serviço disjuntor (*Circuit Breaker*) entra em ação quando um serviço solicita acesso a outro serviço, e este pode estar indisponível ou até mesmo com uma latência alta, tornando inutilizável e levando-o ao esgotamento de recursos, o que causaria a incapacidade de lidar com outras solicitações. Tal falha pode causar um efeito cascata possivelmente para outros serviços afetando todo o aplicativo (FOWLER, 2014).

Hystrix surgiu com objetivo de cessar falhas em cascata em ambientes distribuídos com microsserviços, foi implementado utilizando o padrão disjuntor, ou seja, irá fornecer controle sobre falhas e problemas com latência. Com a utilização do Hystrix será rápida a recuperação quando houver falhas (LUKYANCHIKOV, 2016).

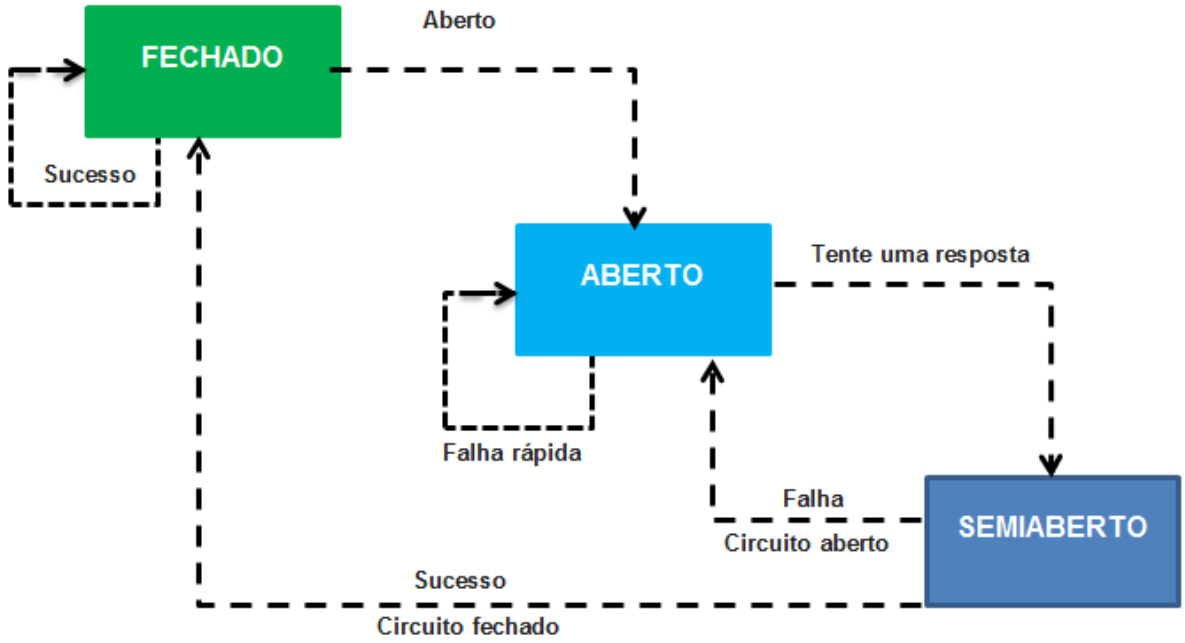
Na figura 23 é mostrado o ciclo de vida de um disjuntor através de um diagrama de estado baseado no funcionamento do Hystrix, nessa imagem existem três estados, o fechado, aberto e o semiaberto. Gopal (2015) explica cada um desses estados:

O estado fechado é a função normal, ou seja, quando um sistema está funcionando sem problemas, a resiliência é medida pelo estado de seus contadores de sucesso, enquanto quaisquer falhas são rastreadas usando os medidores de falha. Esse design garante que, quando o limite de falhas for atingido, o disjuntor passa de fechado para aberto, evitando chamadas adicionais ao recurso dependente (GOPAL, 2015).

O estado aberto é o estado de falha, nesse instante, todas as chamadas para a dependência estão em curto-circuito, é possível obter uma indicação clara de sua causa, usando algumas notações do Hystrix. Uma vez que o intervalo de sono passa, o disjuntor Hystrix se move para um estado semiaberto (GOPAL, 2015).

No estado semiaberto o Hystrix se encarrega de enviar a primeira solicitação para verificar se o sistema esta disponível, permitindo que outras solicitações falhem rapidamente até que a resposta seja obtida. Se a chamada for bem sucedida, o disjuntor é reiniciado para Fechado, se houver falha, o sistema volta ao estado Aberto e o ciclo continua (GOPAL, 2015).

Figura 23 – Diagrama do disjuntor Hystrix



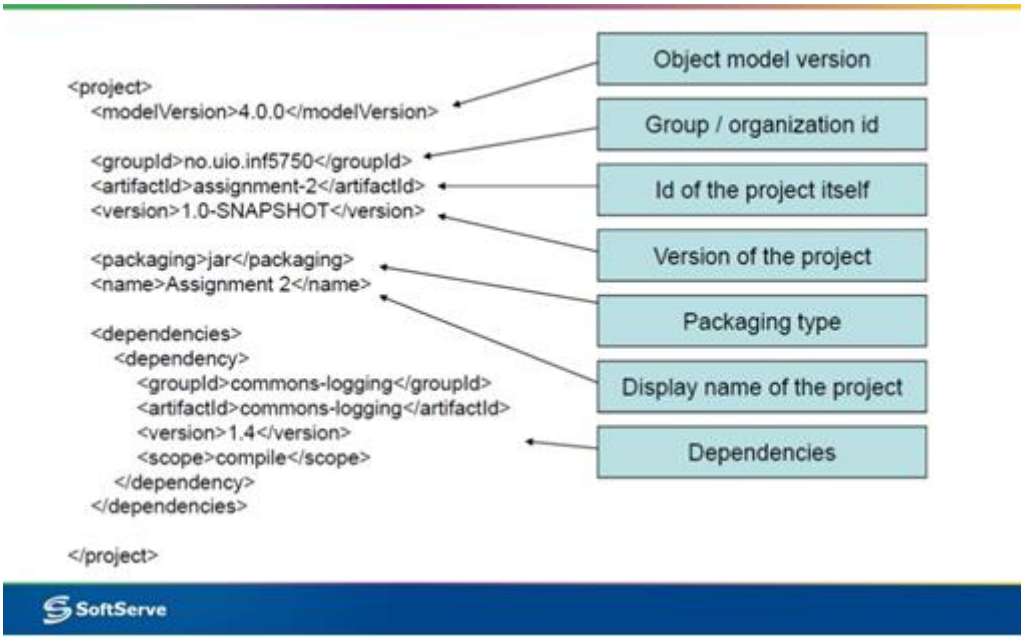
FONTE: Produção Própria dos autores

3.3 Maven

O Apache Maven é uma ferramenta de gerenciamento de dependências de software no Java. Com base no conceito de um modelo de objeto de projeto cuja sigla é (POM), é um arquivo XML, o Maven tem como objetivo gerenciar as bibliotecas externas ao projeto, baixando bibliotecas Java e seus plug-ins dinamicamente de um ou mais repositórios (MAVEN, 2011).

Na figura 24 é possível ver um exemplo do arquivo pom.xml, este possui algumas linhas que serão explicadas a seguir: o *object model version* define a versão do Maven; o groupId define o id do grupo do projeto; ArtifactId é o id do artefato do projeto; *version of the Project* é a versão do artefato no grupo do projeto; *packaging* é o tipo do pacote, como padrão é .jar; *display name of the Project* é o nome de exibição do projeto; *dependencies* são bibliotecas ou plug-ins que o java utiliza.

Figura 24 – Arquivo pom.xml



FONTE**:** РУСИН, 2015.

3.4 Infraestrutura

Nesse tópico é abordada a parte de infraestrutura da arquitetura, descrevendo a tecnologia Docker, que tem o intuito de facilitar a vida do desenvolvedor, livrando de preocupações relacionadas à infraestrutura.

3.4.1 Docker e Docker Compose

O Docker é uma ferramenta que através do uso de contêineres, facilita o desenvolvimento, distribuição e reprodução de aplicações de forma isolada. Com ele é possível fornecer um ambiente ideal para trabalhar e publicar serviços com velocidade, gestão de isolamento e ciclo de vida (STENBERG, 2015).

Através da criação do arquivo chamado Dockerfille, é possível preparar todo ambiente a partir de um script de execução, ele contem comandos que devem seguir uma ordem de formatação correta para o seu funcionamento, respeitando o padrão INSTRUÇÃO argumento, onde INSTRUÇÃO é o comando a ser executado ao montar a imagem e o argumento são instruções que de fato serão realizadas (DIEDRICH, 2015).

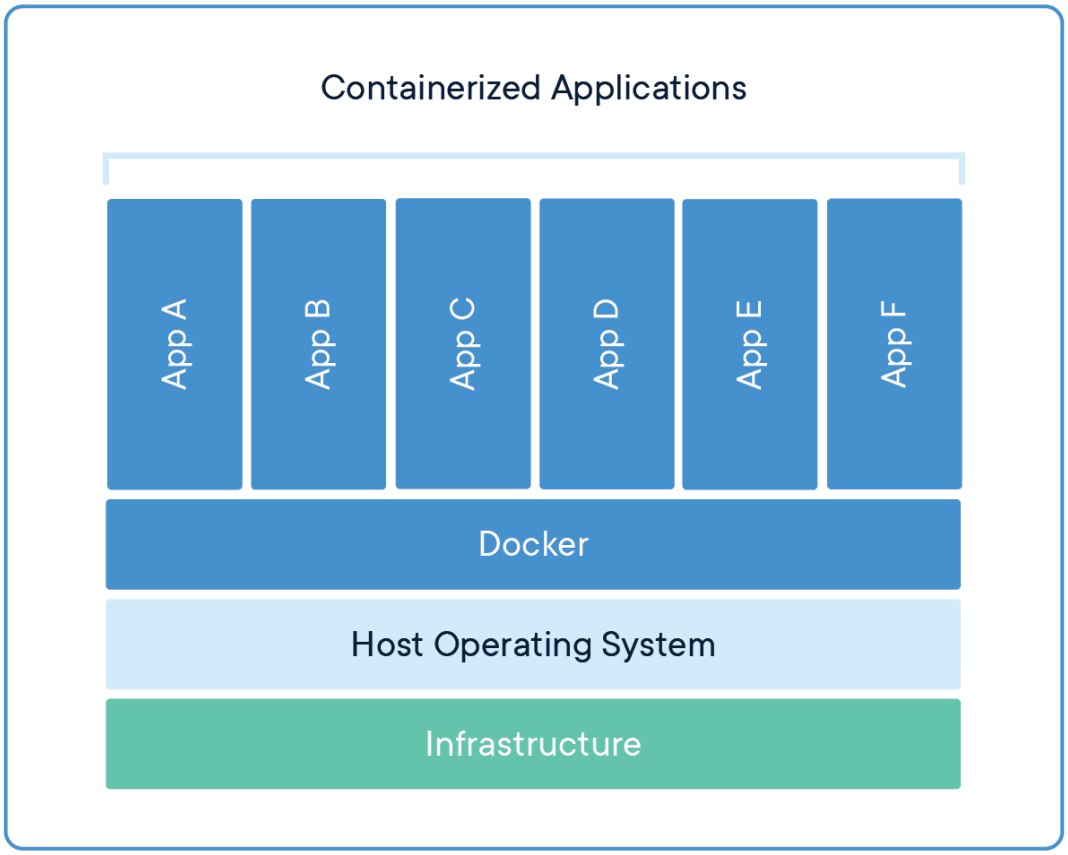
O autor citado anteriormente descreve os seguintes comandos: a instrução FROM, informa a imagem utilizada na criação da nova imagem. Já a instrução VOLUME, mapeia um diretório do host para ser acessível pelo container. A instrução COPY, copia arquivos ou diretórios locais para dentro da imagem, que neste exemplo copia o executável gateway.jar. A

instrução CMD define um comando a ser executado quando um container baseado nessa imagem for iniciado (DIEDRICH, 2015).

O Docker possui uma ferramenta nativa chamada de Docker Compose, que possibilita a criação e execução de vários contêineres. Ou seja, é possível que existam vários contêineres em um único projeto, por exemplo, se tratando da arquitetura de microsserviços, pode existir um contêiner para o banco de dados, um para o Gateway, um para o Auth Server, etc. Neste tipo de cenário, o Docker Compose centralizará as configurações desses contêineres em um único arquivo, facilitando a execução dos contêineres (TRUCCO, 2017).

Na figura 25 é exemplificada a estrutura do docker, onde temos a *infrastructure* que é o hardware, como por exemplo, um servidor. O *host operating system* é o sistema operacional instalado no servidor. O Docker é uma ferramenta de container que provê um modelo de implantação com base em imagens. Em seguida temos os contêineres rodando individualmente seus serviços, por exemplo, o App A, App B, App C e assim por diante.

Figura 25 – Exemplo da estrutura do docker



FONTE: Docker, 2013

CAPÍTULO 4 - DESENVOLVIMENTO

Esse capítulo apresenta os passos realizados para o desenvolvimento da arquitetura de microsserviços. É demonstrado o processo de desenvolvimento que incluem: documento arquitetural, preparação do ambiente de desenvolvimento, desenvolvimento dos serviços que compõem a arquitetura e desenvolvimento de dois microsserviços para validação dessa arquitetura.

4.1 Definição da arquitetura

Antes de iniciar o desenvolvimento da arquitetura, foi definida a linguagem de programação, o framework, banco de dados, a estrutura da arquitetura com suas camadas e tecnologias, além da definição do modelo arquitetural e seus componentes.

Foi definido o documento arquitetural (em anexo) e sua estrutura, que contempla o objetivo, escopo da arquitetura, requisitos e restrições arquiteturais, visões e seus artefatos que serão apresentadas por meio do UML. Tais como diagramas de pacotes, camadas, classes, sequencia dentre outros.

A estrutura da arquitetura de microsserviços utilizada neste projeto faz uso das camadas e tecnologias básicas, por questão de simplificação, a mesma será composta basicamente por quatro camadas, são elas: *Gateway*, *Discovery*, Auth Server e serviços.

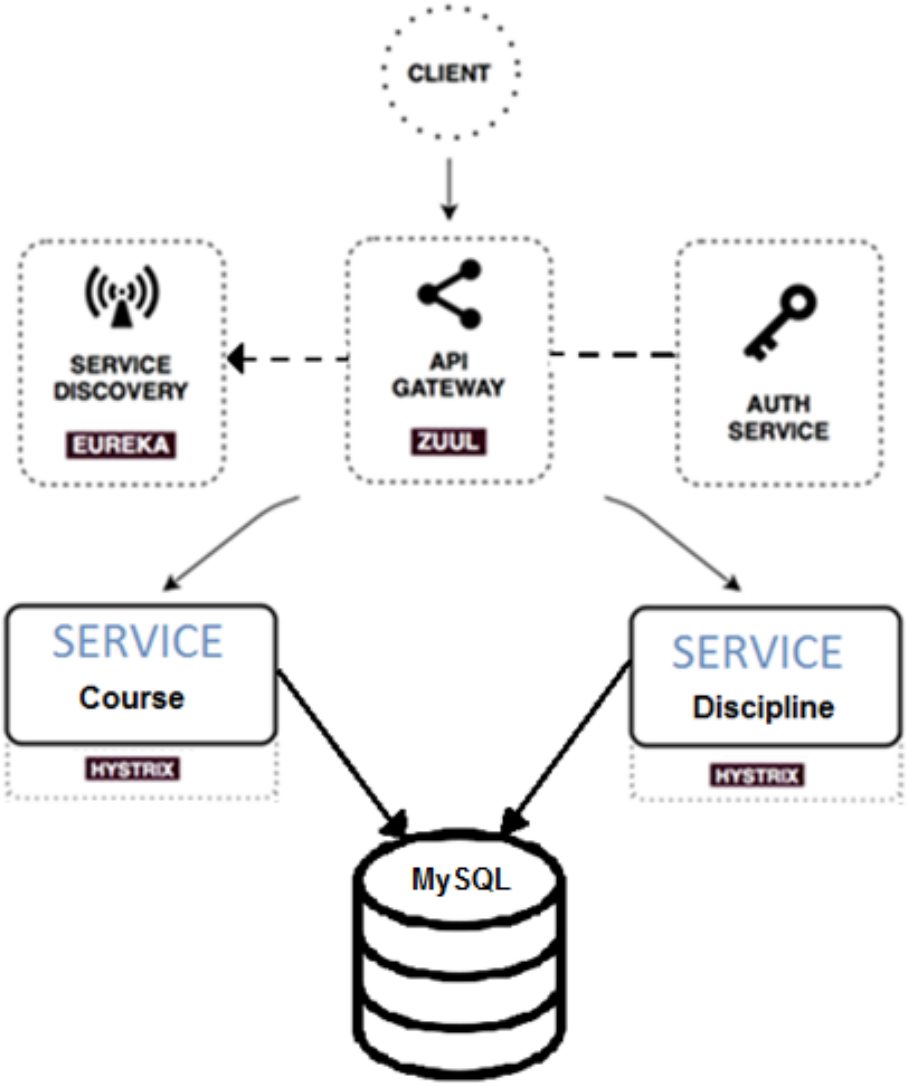
Para a implementação é utilizado à linguagem Java, por ela ser a linguagem de programação base dos cursos de bacharelado em computação na UniEVANGÉLICA, além da utilização do framework Spring por possuir varias ferramentas disponibilizadas por eles, estas resolvem vários problemas encontrados casualmente na hora do desenvolvimento, seja na questão de segurança utilizando o Spring Security, na inicialização rápida do projeto utilizando o Spring Boot ou utilizando o Spring Cloud que ajuda durante o desenvolvimento da arquitetura de microsserviços.

As tecnologias do *Framework Spring* utilizadas para o desenvolvimento das camadas são: O Zuul, Eureka, Hystrix e *Oauth2*. Além de outras tecnologias que são usadas para desenvolver a arquitetura de microsserviços, tais como o banco de dados MySQL, JWT, Maven, a ferramenta Docker, dentre outros. É utilizado o padrão MVC na construção dos componentes dessa arquitetura, por ele separar a aplicação em três camadas com suas responsabilidades.

Seguindo o exposto a figura 26 demonstra a estrutura da arquitetura de microsserviços que é utilizada no presente projeto. O fluxo de dados desta arquitetura começa no *Client* após ter feito a autenticação, por exemplo, o mesmo quer acessar o *Service* Course, para isso o

Client necessita acessar a camada Gateway, que verificará se existe tal serviço, se o serviço existir, o Gateway acessa a camada *Auth Server* e verifica se o *Client* possui permissão para acessar o *Service* Course, se possuir, então o *Gateway* acessará a camada *Discovery* para pegar o endereço do *Service* Course, depois disso a camada *Gateway* acessará o Service Course e retornando-o para o Client.

Figura 26 – Estrutura da arquitetura de microsserviços



FONTE: Produção Própria dos Autores

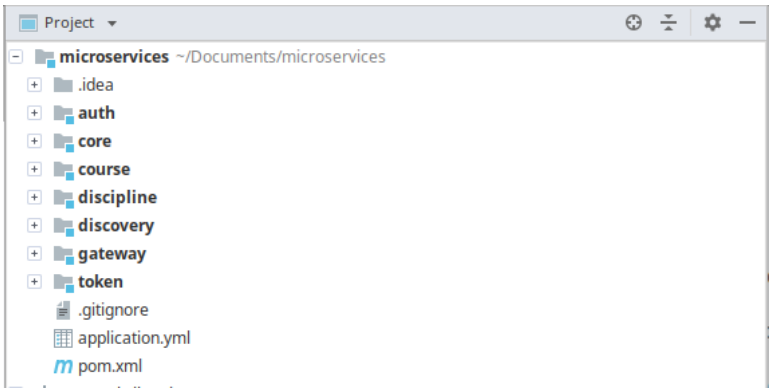
4.2 Desenvolvimento da arquitetura de microsserviços

Nesta sessão são expostas as estruturas de pastas e são explanadas as etapas do desenvolvimento de cada serviço (camada), tais como Gateway, Discovery e Auth Server, além da configuração da tecnologia Docker Compose.

Vale ressaltar que o desenvolvimento dessa arquitetura foi baseado nas vídeo aulas do autor Suane (2019).

A figura 27 mostra a estrutura da pasta raiz do projeto, onde é possível ver a localização de cada serviço contido nesta arquitetura, como o Discovery, o Gateway e os demais.

Figura 27 – Estrutura de pastas da arquitetura

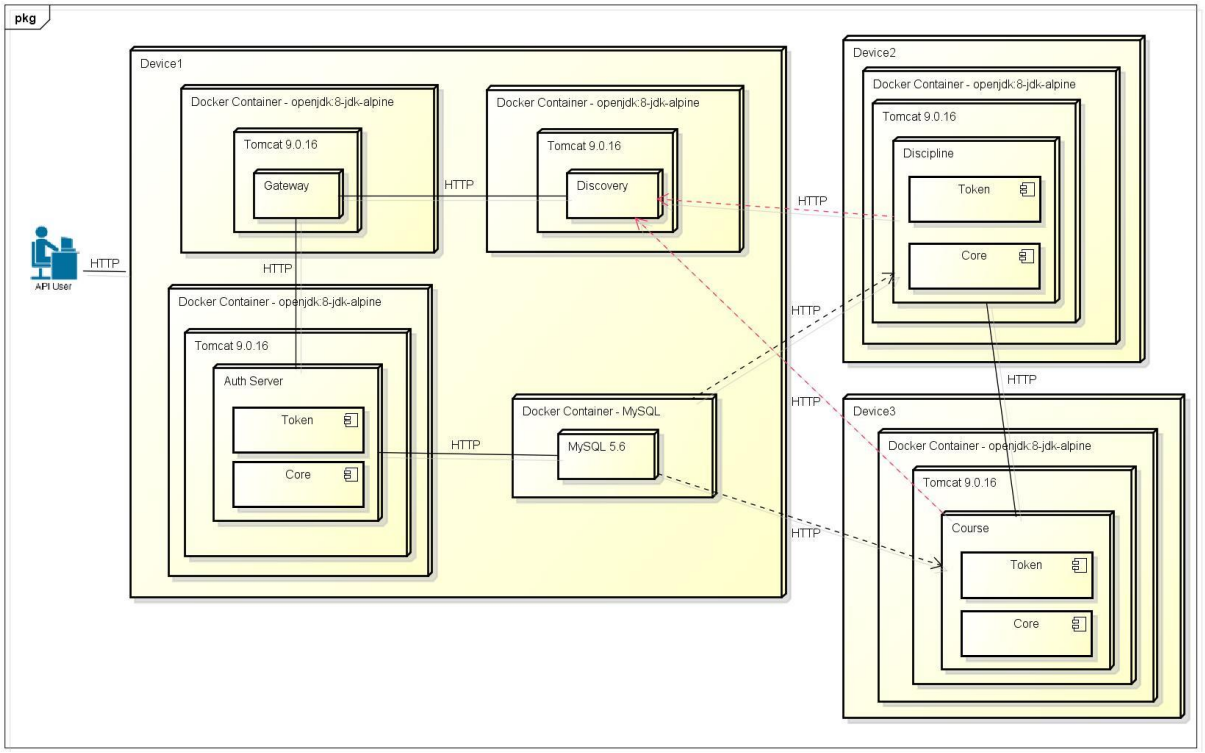


FONTE: Produção Própria dos Autores

4.2.1 Docker

O Docker é responsável por executar cada serviço em um container de forma isolada. A figura 28 demonstra todo o aspecto físico além da distribuição dos módulos da arquitetura utilizando o docker.

Figura 28 – Diagrama de implantação



FONTE**:** Produção Própria dos Autores

No diagrama acima temos os servidores, os dockers que criam os contêineres com a imagem *openjdk 8-jdk-alpine para rodar a aplicação java*, além da imagem MySQL para rodar o banco de dados MySQL 5.6. Para a comunicação desses serviços é utilizado o protocolo HTTP.

Nesta arquitetura a configuração de cada container é semelhante, diferenciando-se apenas o nome do arquivo .jar. É usado como exemplo o arquivo Dockerfile da camada Gateway conforme a figura 29. Vale ressaltar que para cada serviço adicionado a esta arquitetura é necessário adicionar também o arquivo Dockerfile deste serviço.

Figura 29 - Arquivo Dockerfile

Dockerfile

```
1  FROM openjdk:8-jdk-alpine
2  VOLUME /tmp
3  COPY target/gateway.jar app.jar
4  CMD ["java", "-Djava.security.egd=file:/dev/./urandom", "-Dspring.profiles.active=docker", "-jar", "/app.jar"]
5
```

FONTE**:** Produção Própria dos Autores

A fim de simplificar a configuração do Docker, foram utilizados apenas os comandos necessários para a montagem da imagem de cada container, os comandos são: a instrução FROM indica que iremos utilizar o opnejdk:8-jdk-alpine. Já a instrução VOLUME informa que deixaremos a pasta /tmp acessível pelo container. A instrução COPY está copiando o executável gateway.jar para a pasta /tmp e o renomeando para app.jar. A instrução CMD com o argumento passado executa o artefato adicionado dentro da imagem

4.2.2 Docker Compose

O Docker Compose tem a função de gerenciar e centralizar as configurações dos contêineres em um único arquivo. Conforme a figura 30 o arquivo application.yml contém a configuração de cada container da arquitetura de microsserviços. Vale ressaltar que ao adicionar um arquivo Dockerfile é necessário configurá-lo no Docker Compose, ou seja, no arquivo application.yml.

Figura 30 - Arquivo application.yml

application.yml

```yaml
# Especifica a versão do Docker Compose
version: '3.3'

# Nesta seção será especificada a configuração de cada container.
services:
  # Container do banco de dados.
  mysql:
    # Imagem utilizada no container.
    image: mysql:5.6
    # Nome do Container.
    container_name: mysql
    # Aqui são passados os parametros usuário, senha, nome do banco a ser criado e o host.
    environment:
      - MYSQL_USER=root
      - MYSQL_ROOT_PASSWORD=root
      - MYSQL_DATABASE=microservice
      - MYSQL_ROOT_HOST=%
    # Porta de acesso ao banco de dados.
    ports:
      - 3306:3306

  # Container do discovery.
  discovery:
    stdin_open: true
    tty: true
    build:
      context: discovery
      # Especificando o arquivo de configurações Dockerfile.
      dockerfile: Dockerfile
    container_name: discovery
    ports:
      - 8081:8081

  gateway:
    stdin_open: true
    tty: true
    build:
      context: gateway
      dockerfile: Dockerfile
    container_name: gateway
    ports:
      - 8080:8080
    # Indica as dependências para a criação desta imagem.
    depends_on:
      - discovery

  auth:
    stdin_open: true
    tty: true
    build:
      context: auth
      dockerfile: Dockerfile
    container_name: auth
    ports:
      - 8083:8083
    depends_on:
      - mysql
      - discovery
```

FONTE**:** Produção Própria dos Autores

Neste arquivo estão configurados os contêineres que compõem este sistema, são eles o mysql, Discovery, Gateway e Auth.

Logo no início do arquivo application.yml, deve ser informada a versão utilizada do Docker Compose, está sendo utilizada a versão 3.3 como demonstra a figura acima. Em seguida são configurados cada container na sessão services. O primeiro container a ser configurado é o MySQL responsável pelo banco de dados. Na linha 9 é informado o mysql:5.6, essa é a imagem base utilizada para sua construção. Em seguida, na linha 11 temos o nome do container mysql. Na linha 13 começa a sessão environment, nesta são passados parâmetros utilizados para a configuração do banco de dados, como o nome do banco,

usuário, senha e o host, por último é informada a porta onde o banco estará disponível, a porta 3306 localizada na linha 20. A partir da linha 23 temos as configurações dos contêineres discovery, gateway e auth. Os demais serviços que venham a ser adicionados a esta arquitetura deverão ter suas configurações adicionadas neste arquivo.

Estes contêineres citados possuem configurações semelhantes, pois em todos eles são configurados as portas onde ficarão disponíveis, os nomes de cada container, suas dependências como ocorre com os contêineres gateway e auth, estes possuem como dependências o discovery e o mysql, isso significa que os contêineres gateway e auth serão construídos após a construção de suas dependências. Além destas configurações, para cada um é informado que algumas configurações estarão disponíveis nos respectivos arquivos DockerFile, localizados na raiz de cada projeto. Nestes arquivos estão informadas as imagens base utilizadas para suas construções e os comandos para a construção de cada aplicação.

4.2.3 Spring Boot

O Spring facilita o processo de configuração e publicação de aplicações, cuja intenção é rodar o projeto o mais rápido possível e sem complicações, no tópico Spring boot do referencial teórico é mostrado à tela inicial do Spring Initializr, este foi utilizado para criar cada serviço do projeto.

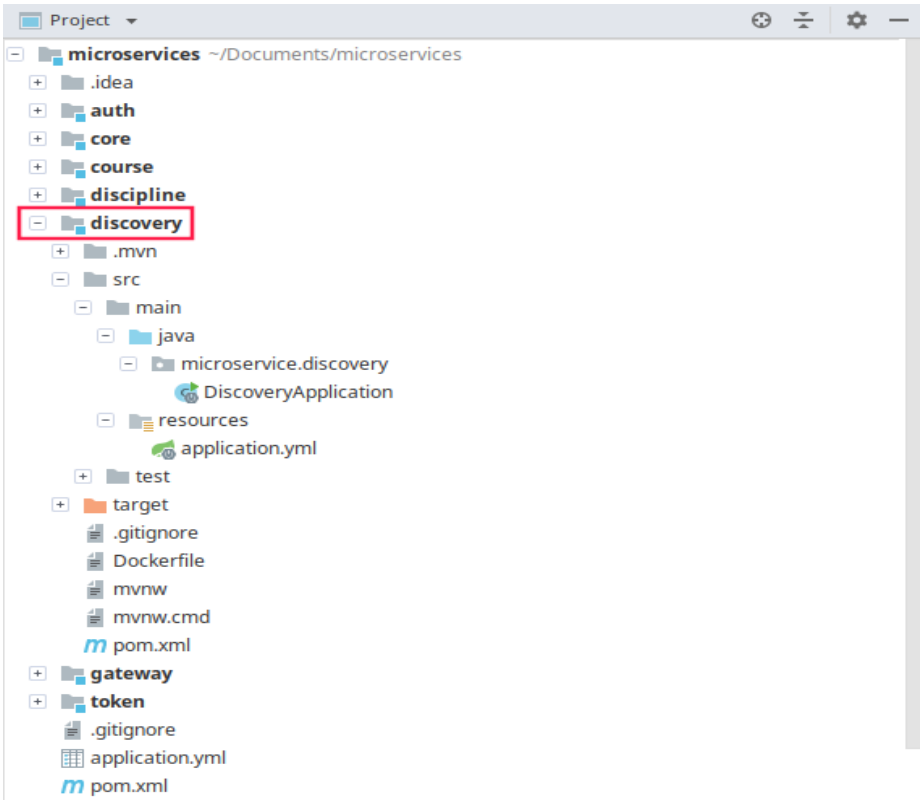
No Spring Initializr foi selecionada a versão do Spring Boot 2.1.3, além das seguintes dependências: MySQL, Cloud Config, Cloud Zuul, Cloud Hystrix, Cloud Eureka, Cloud Circuit Breaker, Cloud Auth Server entre outros.

4.2.4 Discovery

Este serviço faz uma tarefa essencial numa arquitetura baseada em microsserviços, registrando cada serviço disponível no sistema. Através dele, cada serviço sabe o endereço para encontrar outro serviço necessário para o seu funcionamento, possibilitando a comunicação entre eles.

Para sua implementação foi utilizada a tecnologia *Eureka*, com ela foi possível criar e configurar o Discovery de forma rápida e simples, bastando apenas algumas linhas de código para que ele ficasse disponível. É possível observar na figura 31 que o Discovery contém apenas a classe principal e seu arquivo de configuração.

Figura 31 - Estrutura de pasta da camada discovery



FONTE: Produção Própria dos Autores

A figura 32 mostra o arquivo de configuração application.yml do Discovery com suas devidas configurações.

Figura 32 - Arquivo de configurações do Discovery

application.yml

```yaml
spring:
  application:
    name: discovery

server:
  port: 8081

eureka:
  client:
    register-with-eureka: false
    fetch-registry: false
    service-url:
      defaultZone: http://discovery:${server.port}/eureka/
```

FONTE: Produção Própria dos Autores

Neste arquivo é configurado somente o nome da aplicação, a porta que ele estará disponível para as demais aplicações e ao contrário dos demais serviços, o Eureka deve ser configurado para não se registrar, pois por padrão ele tentará se auto registrar.

A figura 33 mostra a classe DiscoveryApplication do Discovery.

Figura 33 - Classe principal do Discovery

DiscoveryApplication.java

```
1  package microservice.discovery;
2
3  import ...
6
7  @SpringBootApplication
8  @EnableEurekaServer
9  public class DiscoveryApplication {
10
11     public static void main(String[] args) { SpringApplication.run(DiscoveryApplication.class, args); }
14
15 }
16
```

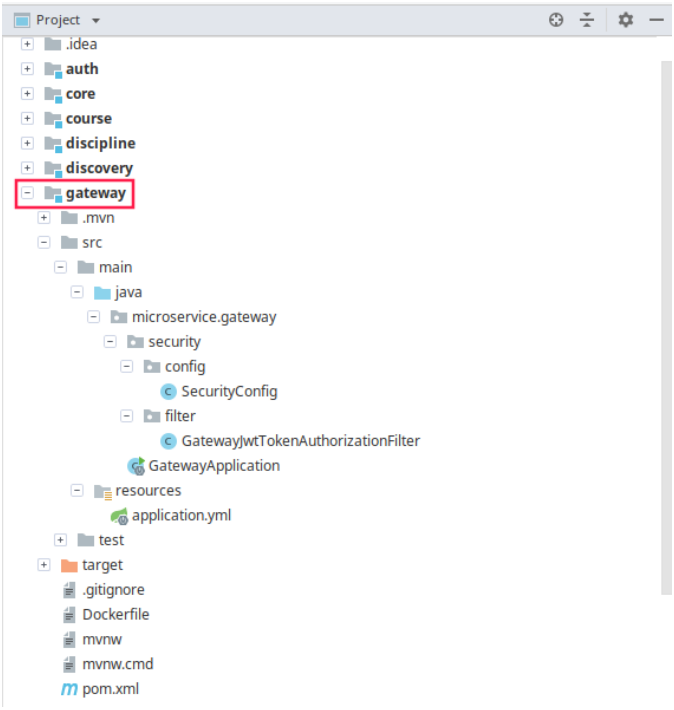
FONTE: Produção Própria dos Autores

Esta classe habilita o serviço de registros com a notação @EnableEurekaServer, o que a torna um servidor de registros Eureka, onde os demais serviços irão se registrar para ficarem disponíveis para outros serviços.

4.2.5 Gateway

Este serviço é fundamental para a disposição do sistema a acessos externos, pois todas as requisições ao sistema chegam através dele, este é a porta de entrada para os demais sistemas internos contidos nesta arquitetura. Para o seu desenvolvimento foi utilizada a tecnologia Zuul, essa é responsável por fazer o roteamento de todo o sistema. Na figura 34 é demonstrada a estrutura de pastas desse serviço.

Figura 34 - Estrutura de pastas da camada Gateway



FONTE: Produção Própria dos Autores

Na figura 36 é possível observar a classe principal do gateway, o arquivo de configuração da aplicação, estes serão descritos a seguir, além das configurações de segurança implementadas nas classes SecurityConfig e GatewayJwtTokenAutorizationFilter.

A figura 35 mostra o arquivo application.yml do *Gateway* com suas configurações necessárias.

Figura 35 - Arquivo de configurações do *Gateway*

application.yml

```yaml
spring:
  application:
    name: gateway

server:
  port: 8080
  servlet:
    context-path: /gateway

eureka:
  instance:
    prefer-ip-address: true
  client:
    service-url:
      defaultZone: http://discovery:8081/eureka/
    fetch-registry: true
    register-with-eureka: true

zuul:
  sensitive-headers: Cookie

jwt:
  config:
    login-url: /auth/login

ribbon:
  ReadTimeout: 10000
  ConnectTimeout: 10000
```

FONTE**:** Produção Própria dos Autores

Neste arquivo foi configurado o nome da aplicação, a porta na qual o serviço estará disponível e a configuração para que ele se auto registre no *Discovery*. Além das configurações de autenticação e de roteamento dinâmico.

A figura 36 mostra a classe GatewayApplication do *Gateway*.

Figura 36 - Classe principal do Gateway

GatewayApplication.java

```java
package microservice.gateway;

import microservice.core.property.JwtConfiguration;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.context.properties.EnableConfigurationProperties;
import org.springframework.cloud.netflix.eureka.EnableEurekaClient;
import org.springframework.cloud.netflix.zuul.EnableZuulProxy;
import org.springframework.context.annotation.ComponentScan;

@SpringBootApplication
@EnableZuulProxy
@EnableEurekaClient
@EnableConfigurationProperties(value = JwtConfiguration.class)
@ComponentScan("microservice")
public class GatewayApplication {

    public static void main(String[] args) { SpringApplication.run(GatewayApplication.class, args); }

}

```

FONTE**:** Produção Própria dos Autores

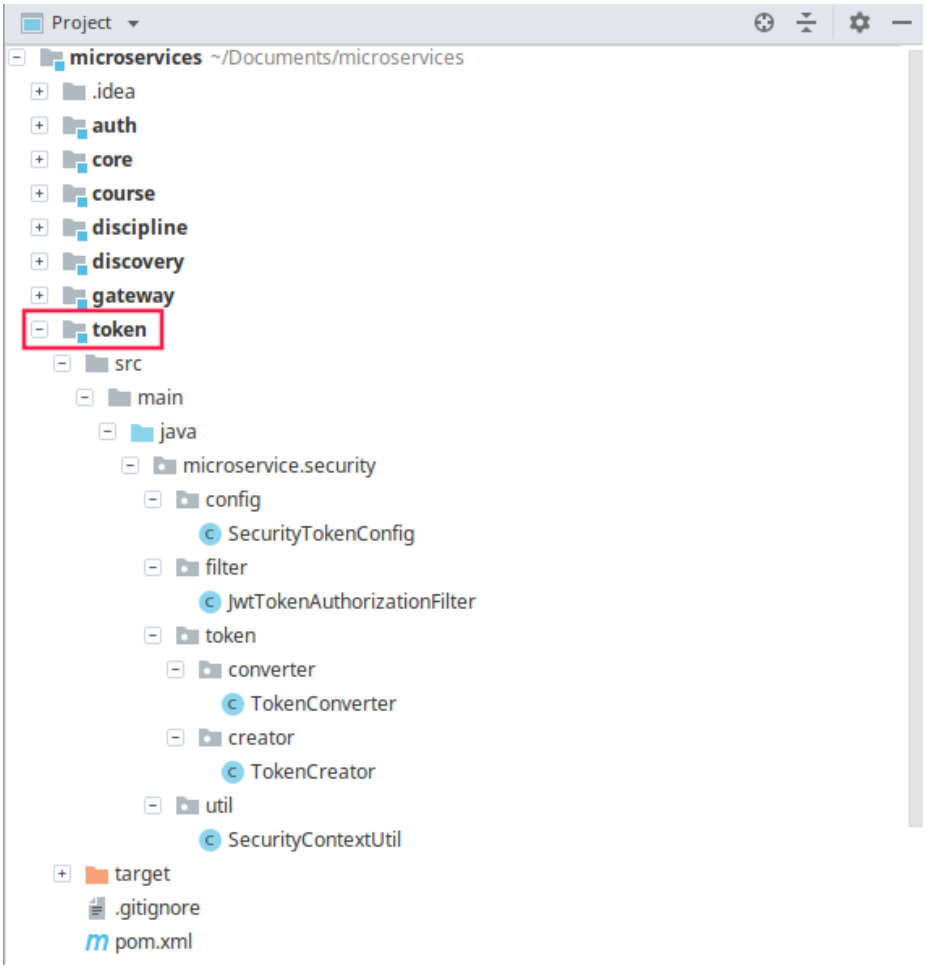
Nesta classe, o serviço de *proxy* do *Zuul* é habilitado com a notação @EnableZuulProxy, esta transformará o serviço Gateway em um proxy responsável por encaminhar as chamadas relevantes para outros serviços, tais como, Auth Server, Discovery e demais serviços que venham ser implementado na arquitetura. Já a notação @EnableEurekaClient, faz com que o Gateway se instancie no *Eureka*, ou seja ele se registra na camada Discovery (SPRING, 2014).

4.2.6 Módulo Token

Este módulo foi criado a fim de reduzir a repetição de código, pois as configurações e métodos contidos neste deverão ser usados por todos os microsserviços dessa arquitetura para validações e segurança.

Na figura 37 são mostradas as subpastas deste módulo.

Figura 37 - Estrutura de pastas do módulo Token



FONTE: Produção Própria dos Autores

Dentre essas subpastas serão detalhadas as pastas config, filter, token e util. Na pasta config temos a classe SecurityTokenConfig demonstrada na figura 38.

Figura 38 - Classe SecurityTokenConfig

SecurityTokenConfig.java

```
1  package microservice.security.config;
2
3  import lombok.RequiredArgsConstructor;
4  import microservice.core.property.JwtConfiguration;
5  import org.springframework.beans.factory.annotation.Autowired;
6  import org.springframework.http.HttpMethod;
7  import org.springframework.security.config.annotation.web.builders.HttpSecurity;
8  import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter
9  import org.springframework.web.cors.CorsConfiguration;
10
11 import javax.servlet.http.HttpServletResponse;
12
13 import static org.springframework.security.config.http.SessionCreationPolicy.STATELESS;
14
15 @RequiredArgsConstructor(onConstructor = @__(@Autowired))
16 public class SecurityTokenConfig extends WebSecurityConfigurerAdapter {
17     protected final JwtConfiguration jwtConfiguration;
18
19     @Override
20     protected void configure(HttpSecurity http) throws Exception {
21         http
22                 .csrf().disable()
23                 .cors().configurationSource
24                         (request -> new CorsConfiguration().applyPermitDefaultValues())
25                 .and()
26                 .sessionManagement().sessionCreationPolicy(STATELESS)
27                 .and()
28                 .exceptionHandling()
29                     .authenticationEntryPoint(
30                         (req, resp, e) -> resp.sendError(HttpServletResponse.SC_UNAUTHORIZED))
31                 .and()
32                 .authorizeRequests()
33                 .antMatchers( ...antPatterns: jwtConfiguration.getLoginUrl(),
34                         "/**/swagger-ui.html").permitAll()
35                 .antMatchers(HttpMethod.GET,  ...antPatterns: "/**/swagger-resources/**",
36                         "/**/webjars/springfox-swagger-ui/**", "/**/v2/api-docs/**").permitAll()
37                 .antMatchers( ...antPatterns: "/course/v1/admin/**").hasRole("ADMIN")
38                 .antMatchers( ...antPatterns: "/auth/user/**").hasAnyRole( ...roles: "ADMIN", "USER")
39                 .antMatchers( ...antPatterns: "/discipline/v1/**").hasAnyRole( ...roles: "ADMIN", "USER")
40                 .anyRequest().authenticated();
41     }
42
43
44 }
45
```

FONTE**:** Produção Própria dos Autores

Toda requisição realizada no sistema passará pelo método configure, pois nele são definidas as configurações HTTP utilizadas em cada requisição feita ao sistema. Algumas destas configurações serão descritas a seguir, como: a configuração de CORS, localizada na linha 23, nela é colocada uma configuração padrão, que atende aos requisitos do projeto. Na linha 25 está configurado que todas as sessões serão do tipo STATELESS, pois não é guardada nenhuma sessão de usuário. Já na linha 27 é configurado para ser lançada a exceção UNAUTHORIZED para qualquer exceção lançada.

Nesta classe também são configuradas as rotas que precisam e as que não precisão de autenticação para serem acessadas, como as páginas na linha 30 que mapeiam a API da aplicação através do Swagger. Logo abaixo, na linha 32 são definidos quais perfis de usuários

terão acesso a cada rota do sistema, nela podemos ver que somente os usuários com o perfil ADMIN poderão acessar as rotas referentes ao serviço Course.

A figura 39 mostra a classe JwtTokenAuthorizationFilter ela se encontra dentro da pasta filter, esta é executada em toda requisição feita neste sistema que contenha o módulo token adicionado em suas dependências.

Figura 39 - Classe JwtTokenAuthorizationFilter

JwtTokenAuthorizationFilter.java

```java
package microservice.security.filter;

import com.nimbusds.jwt.SignedJWT;
import lombok.RequiredArgsConstructor;
import lombok.SneakyThrows;
import lombok.extern.slf4j.Slf4j;
import microservice.core.property.JwtConfiguration;
import microservice.security.token.converter.TokenConverter;
import microservice.security.util.SecurityContextUtil;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.lang.NonNull;
import org.springframework.web.filter.OncePerRequestFilter;

import javax.servlet.FilterChain;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

import static org.apache.commons.lang3.StringUtils.equalsIgnoreCase;

@Slf4j
@RequiredArgsConstructor(onConstructor = @__(@Autowired))
public class JwtTokenAuthorizationFilter extends OncePerRequestFilter {
    protected final JwtConfiguration jwtConfiguration;
    protected final TokenConverter tokenConverter;

    @Override
    /Duplicates/
    protected void doFilterInternal(@NonNull HttpServletRequest request,
                                    @NonNull HttpServletResponse response,
                                    @NonNull FilterChain chain)
            throws ServletException, IOException {

        String header = request.getHeader(jwtConfiguration.getHeader().getName());

        if (header == null || !header.startsWith(jwtConfiguration.getHeader().getPrefix())) {
            chain.doFilter(request, response);
            return;
        }

        String token = header.replace(jwtConfiguration.getHeader()
                .getPrefix(), replacement: "").trim();

        SecurityContextUtil.setSecurityContext(equalsIgnoreCase("signed",
                jwtConfiguration.getType()) ? validate(token) : decryptValidating(token));

        chain.doFilter(request, response);
    }

    @SneakyThrows
    private SignedJWT decryptValidating(String encryptedToken) {
        String signedToken = tokenConverter.decryptToken(encryptedToken);
        tokenConverter.validateTokenSignature(signedToken);
        return SignedJWT.parse(signedToken);
    }

    @SneakyThrows
    private SignedJWT validate(String signedToken) {
        tokenConverter.validateTokenSignature(signedToken);
        return SignedJWT.parse(signedToken);
    }
}
```

FONTE**:** Produção Própria dos Autores

Na figura acima mostra a classe JwtTokenAuthorizationFilter ela estende a classe OncePerRequestFilter para garantir que a classe JwtTokenAuthorizationFilter seja executada apenas uma vez por requisição, caso contrário, esta classe pode acabar sendo executada mais de uma vez em cada requisição.

Na linha 34 temos uma validação para averiguar a necessidade do Header, pois caso a requisição não necessite de autenticação o Header não será necessário. Na linha 39 ocorre à abstração do token do header, nesta linha é removido seu prefixo e atribuído o token a uma variável para posteriormente ser descriptografado e validado.

Caso o token chegue criptografado será chamado o método decryptValidating, na linha 47 para descriptografar e validar o token, caso contrário, será chamado o método validate da linha 54 para a validação do token.

A pasta token contém duas subpastas, a pasta converter e a pasta creator. Nelas temos as classes responsáveis por criar, criptografar e descriptografar os tokens, que são demonstrados a seguir.

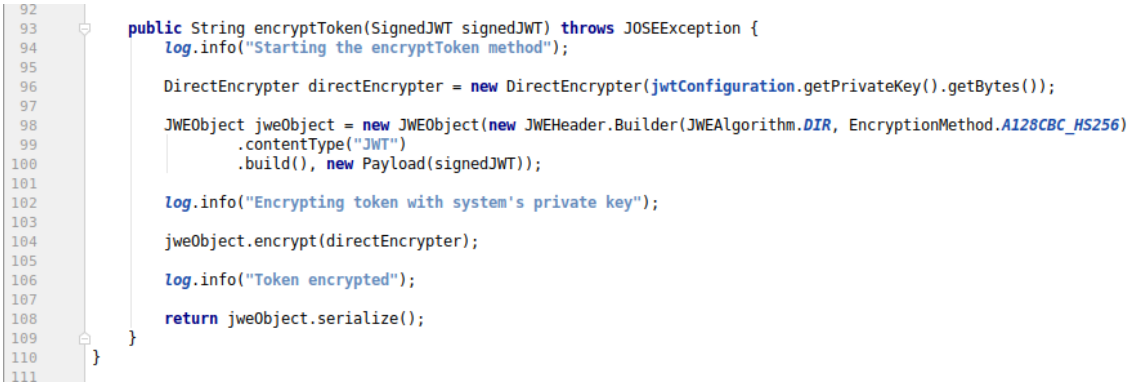
Figura 40 - Método createSignedJWT

```
34      @SneakyThrows
35      public SignedJWT createSignedJWT(Authentication auth) {
36          log.info("Starting to create the signed JWT");
37
38          ApplicationUser applicationUser = (ApplicationUser) auth.getPrincipal();
39
40          JWTClaimsSet jwtClaimSet = createJWTClaimSet(auth, applicationUser);
41
42          KeyPair rsaKeys = generateKeyPair();
43
44          log.info("Building JWK from the RSA Keys");
45
46          JWK jwk = new RSAKey.Builder((RSAPublicKey) rsaKeys.getPublic()).keyID(UUID.randomUUID().toString()).build();
47
48          SignedJWT signedJWT = new SignedJWT(new JWSHeader.Builder(JWSAlgorithm.RS256)
49                  .jwk(jwk)
50                  .type(JOSEObjectType.JWT)
51                  .build(), jwtClaimSet);
52
53          log.info("Signing the token with the private RSA Key");
54
55          RSASSASigner signer = new RSASSASigner(rsaKeys.getPrivate());
56
57          signedJWT.sign(signer);
58
59          log.info("Serialized token '{}'", signedJWT.serialize());
60
61          return signedJWT;
62
63      }
64
```

FONTE**:** Produção Própria dos Autores

A figura 40 mostra o método createSignedJWT, contido na classe TokenCreator é o responsável pela criação do token já assinado. Ainda nesta classe temos o método encryptToken mostrado na figura 41.

Figura 41 - Método encryptToken



```
92
93      public String encryptToken(SignedJWT signedJWT) throws JOSEException {
94          log.info("Starting the encryptToken method");
95
96          DirectEncrypter directEncrypter = new DirectEncrypter(jwtConfiguration.getPrivateKey().getBytes());
97
98          JWEObject jweObject = new JWEObject(new JWEHeader.Builder(JWEAlgorithm.DIR, EncryptionMethod.A128CBC_HS256)
99                  .contentType("JWT")
100                 .build(), new Payload(signedJWT));
101
102         log.info("Encrypting token with system's private key");
103
104         jweObject.encrypt(directEncrypter);
105
106         log.info("Token encrypted");
107
108         return jweObject.serialize();
109     }
110 }
111
```

FONTE: Produção Própria dos Autores

O método acima recebe um token já assinado e o criptografa através de uma criptografia direta usando uma chave privada, o algoritmo JWEAlgorithm.DIR e o método de encriptação EncryptionMethod.A128CBC_HS256.

Na classe TokenConverter estão as classes responsáveis por descriptografar e validar os tokens.

Figura 42 - Método decryptToken

```
22      @SneakyThrows
23      public String decryptToken(String encryptedToken) {
24          log.info("Decrypting token");
25
26          JWEObject jweObject = JWEObject.parse(encryptedToken);
27
28          DirectDecrypter directDecrypter = new DirectDecrypter(jwtConfiguration.getPrivateKey().getBytes());
29
30          jweObject.decrypt(directDecrypter);
31
32          log.info("Token decrypted, returning signed token . . . ");
33
34          return jweObject.getPayload().toSignedJWT().serialize();
35      }
36
```

FONTE: Produção Própria dos Autores

Este método recebe uma string com o token criptografado e o retorna descriptografado.

Figura 43 - Método validateTokenSignature

```
37      @SneakyThrows
38      public void validateTokenSignature(String signedToken) {
39          log.info("Starting method to validate token signature...");
40
41          SignedJWT signedJWT = SignedJWT.parse(signedToken);
42
43          log.info("Token Parsed! Retrieving public key from signed token");
44
45          RSAKey publicKey = RSAKey.parse(signedJWT.getHeader().getJWK().toJSONObject());
46
47          log.info("Public key retrieved, validating signature. . . ");
48
49          if (!signedJWT.verify(new RSASSAVerifier(publicKey)))
50              throw new AccessDeniedException("Invalid token signature!");
51
52          log.info("The token has a valid signature");
53      }
54  }
```

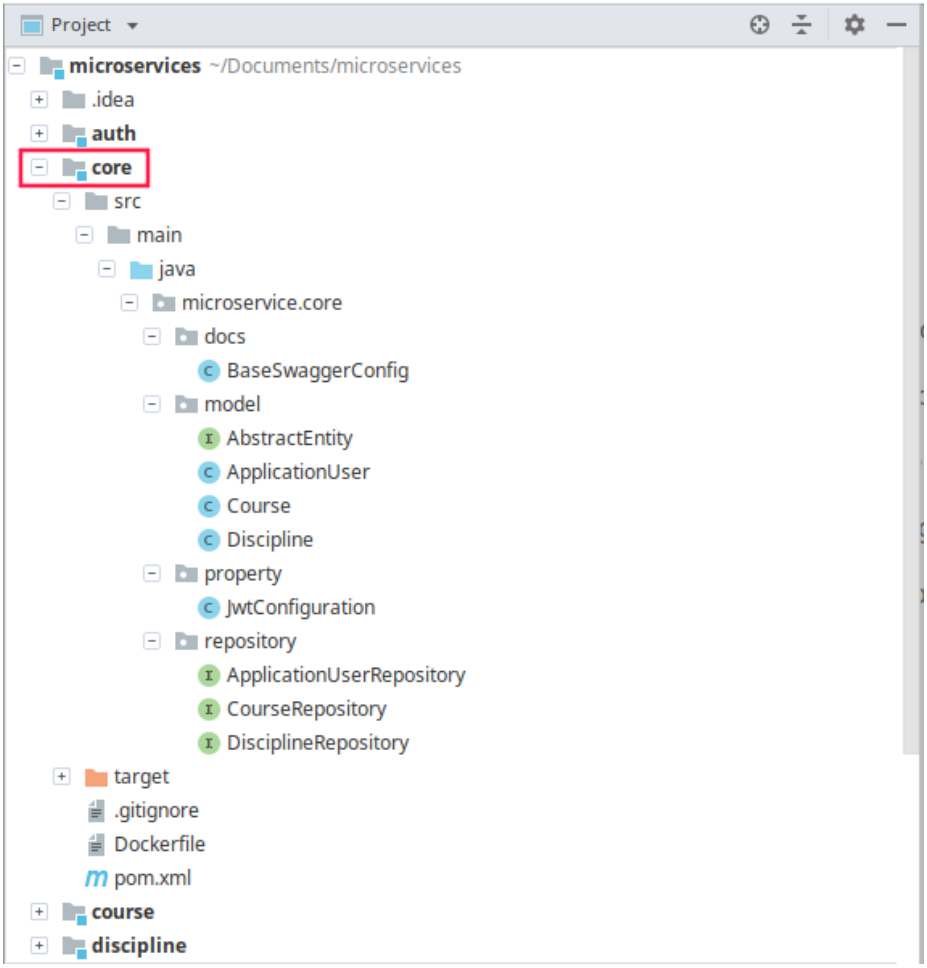
FONTE: Produção Própria dos Autores

O método validateTokenSignature mostrado na figura 43 é responsável por validar o token que é recebido já descriptografado para que seja validada sua assinatura. Na linha 45 é recuperada a chave pública do token e na linha 49 é verificado se a chave é válida ou não, caso seja inválida é lançada uma exceção na linha 50, como este método só valida o token, não é necessário que ele retorne algo.

4.2.7 Módulo Core

O módulo Core foi desenvolvido a fim de reunir as entidades e repositórios em um único lugar, pois estes podem ser utilizados por outros serviços, fazendo com que a modularização simplifique a utilização destas entidades e repositórios. Na figura 44 mostra a estrutura de pastas do módulo Core, nela se destacam as pastas docs, model, property e repositor.

Figura 44 - Estrutura de pastas do módulo Core



FONTE: Produção Própria dos Autores

Na imagem acima temos a pasta docs, nela se encontra a classe BaseSwaggerConfig que contém as configurações básicas do Swagger, as demais configurações sobre cada endpoint do sistema estarão localizadas em cada microsserviço. Assim, cada serviço ficará responsável por documentar sua própria API.

Na pasta model estão localizadas as entidades e na pasta repository os repositórios utilizados nesta arquitetura. Vale ressaltar que para novas entidades e repositórios que venham a ser adicionados nesta arquitetura através de novos serviços, não serão obrigatórios sua adição neste módulo, caso seja necessário, as novas entidades e repositórios poderão estar contidos nos microsserviços adicionados a esta arquitetura.

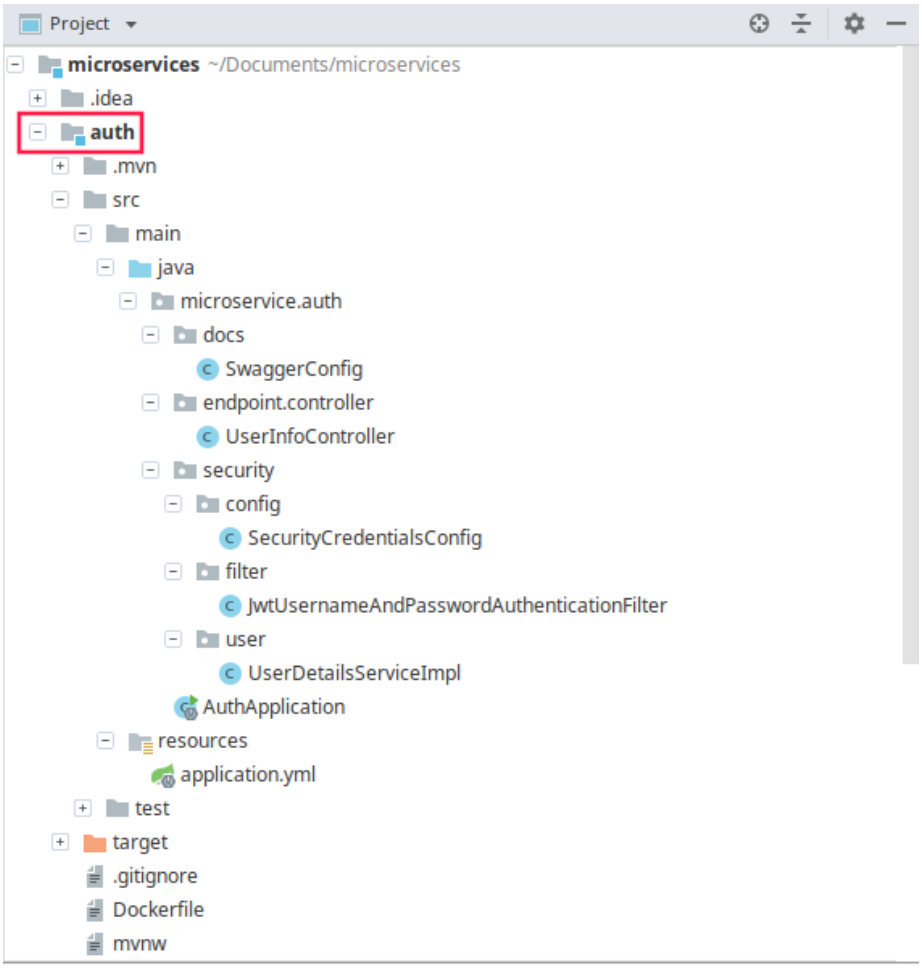
Na pasta property está localizada a classe JwtConfiguration, nesta estão as configurações do JWT, como a chave privada utilizada para criptografia, o tempo de expiração dos tokens gerados pelo sistema, cujo objetivo é que a partir de determinado tempo os tokens gerados sejam inválidos, obrigando assim a solicitação de um novo token, além das configurações do Header como nome e prefixo e outras mais configurações que juntas compõem o JWT.

4.2.8 Auth

Foi utilizado o *Spring Security* para o desenvolvimento da camada *Auth Server*, esta foi modularizada a fim de reduzir a duplicidade de código e dividir certas responsabilidades em módulos, pois algumas configurações como as de token devem ser mantidas em cada microsserviço.

A figura 45 mostra a estrutura de pastas do serviço Auth responsável pela a autenticação de toda a arquitetura.

Figura 45 - Estrutura de pastas da camada Auth



FONTE: Produção Própria dos Autores

Na imagem acima temos a pasta docs com a classe SwaggerConfig, que contém as configurações da API deste serviço.

Dentro da pasta endpoint está a subpasta controller, nela se localiza a classe UserInfoController, esta classe está reservada para os endpoints de usuário relacionados à autenticação como mostra a figura 46 a seguir.

Figura 46 - Classe UserInfoController

UserInfoController.java

```java
package microservice.auth.endpoint.controller;

import io.swagger.annotations.Api;
import io.swagger.annotations.ApiOperation;
import microservice.core.model.ApplicationUser;
import org.springframework.http.HttpStatus;
import org.springframework.http.MediaType;
import org.springframework.http.ResponseEntity;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import java.security.Principal;

@RestController
@RequestMapping("user")
@Api(value = "Endpoints to manage User's information")
public class UserInfoController {

    @GetMapping(path = "info", produces = MediaType.APPLICATION_JSON_UTF8_VALUE)
    @ApiOperation(value = "Recuperará as informações do usuário disponíveis no token.", response = ApplicationUser.class)
    public ResponseEntity<ApplicationUser> getUserInfo(Principal principal) {
        ApplicationUser applicationUser = (ApplicationUser) ((UsernamePasswordAuthenticationToken) principal).getPrincipal();
        return new ResponseEntity<>(applicationUser, HttpStatus.OK);
    }
}
```

FONTE: Produção Própria dos Autores

A classe UserInfoController mostrada na figura acima, exibe o método getUserInfo utilizado na autenticação dos usuários, este quando chamado recupera o usuário a partir de informações contidas no token.

A pasta security contém três subpastas, são elas a config, filter e user. Na pasta config se localiza a classe SecurityCredentialsConfig como ilustrada na figura 47.

Figura 47 - Classe SecurityCredentialsConfig

SecurityCredentialsConfig.java

```
1  package microservice.auth.security.config;
2
3  import microservice.auth.security.filter.JwtUsernameAndPasswordAuthenticationFilter;
4  import microservice.core.property.JwtConfiguration;
5  import microservice.security.config.SecurityTokenConfig;
6  import microservice.security.filter.JwtTokenAuthorizationFilter;
7  import microservice.security.token.converter.TokenConverter;
8  import microservice.security.token.creator.TokenCreator;
9  import org.springframework.beans.factory.annotation.Qualifier;
10 import org.springframework.context.annotation.Bean;
11 import org.springframework.security.config.annotation.authentication.builders
12         .AuthenticationManagerBuilder;
13 import org.springframework.security.config.annotation.web.builders.HttpSecurity;
14 import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
15 import org.springframework.security.core.userdetails.UserDetailsService;
16 import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
17 import org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;
18
19 @EnableWebSecurity
20 public class SecurityCredentialsConfig extends SecurityTokenConfig {
21     private final UserDetailsService userDetailsService;
22     private final TokenCreator tokenCreator;
23     private final TokenConverter tokenConverter;
24
25     public SecurityCredentialsConfig(JwtConfiguration jwtConfiguration,
26                                      @Qualifier("userDetailsServiceImpl")
27                                              UserDetailsService userDetailsService,
28                                      TokenCreator tokenCreator, TokenConverter tokenConverter) {
29         super(jwtConfiguration);
30         this.userDetailsService = userDetailsService;
31         this.tokenCreator = tokenCreator;
32         this.tokenConverter = tokenConverter;
33     }
34
35     @Override
36     protected void configure(HttpSecurity http) throws Exception {
37         http
38                 .addFilter(new JwtUsernameAndPasswordAuthenticationFilter(
39                         authenticationManager(), jwtConfiguration, tokenCreator))
40                 .addFilterAfter(new JwtTokenAuthorizationFilter(
41                         jwtConfiguration, tokenConverter),
42                         UsernamePasswordAuthenticationFilter.class);
43
44         super.configure(http);
45     }
46
47     @Override
48     protected void configure(AuthenticationManagerBuilder auth) throws Exception {
49         auth.userDetailsService(userDetailsService).passwordEncoder(passwordEncoder());
50     }
51
52     @Bean
53     public BCryptPasswordEncoder passwordEncoder() { return new BCryptPasswordEncoder(); }
56 }
57
```

FONTE**:** Produção Própria dos Autores

Na figura acima mostra a classe SecurityCredentialsConfig, ela estende a classe SecurityTokenConfig localizada no módulo Token. O método SecurityCredentialsConfig é o construtor desta classe, ele sempre é executado quando esta classe é instanciada.

Nas linhas 34 e 42 na imagem acima demonstra dois métodos que utilizam a anotação @Override, esta informa que estes métodos estão sobrescrevendo um método localizado na classe estendida desta classe SecurityTokenConfig. O método da linha 34 adiciona filtros ao HTTP, já o método da linha 42 criptografa a senha do usuário utilizando o método passwordEncoder da linha 47.

A figura 48 e 49 mostram a classe *JwtUsernameAndPasswordAuthenticationFilter*, responsável por gerar, assinar e criptografar o token. Nesta classe só é feita a manipulação dos dados necessários para que o Spring faça a validação do usuário.

Figura 48 - Classe JwtUsernameAndPasswordAuthenticationFilter

JwtUsernameAndPasswordAuthenticationFilter.java

```java
package microservice.auth.security.filter;

import com.fasterxml.jackson.databind.ObjectMapper;
import com.nimbusds.jwt.SignedJWT;
import lombok.RequiredArgsConstructor;
import lombok.SneakyThrows;
import lombok.extern.slf4j.Slf4j;
import microservice.core.model.ApplicationUser;
import microservice.core.property.JwtConfiguration;
import microservice.security.token.creator.TokenCreator;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.Authentication;
import org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;

import javax.servlet.FilterChain;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import static java.util.Collections.emptyList;

@RequiredArgsConstructor(onConstructor = @__(@Autowired))
@Slf4j
public class JwtUsernameAndPasswordAuthenticationFilter
        extends UsernamePasswordAuthenticationFilter {
    private final AuthenticationManager authenticationManager;
    private final JwtConfiguration jwtConfiguration;
    private final TokenCreator tokenCreator;

    @Override
    @SneakyThrows
    public Authentication attemptAuthentication(HttpServletRequest request,
                                                HttpServletResponse response) {
        log.info("Attempting authentication. . .");
        ApplicationUser applicationUser = new ObjectMapper()
                .readValue(request.getInputStream(), ApplicationUser.class);

        if (applicationUser == null)
            throw new UsernameNotFoundException("Unable to retrieve the username or password");

        log.info("Creating the authentication object for the user '{}' and calling" +
                " UserDetailServiceImpl loadUserByUsername", applicationUser.getUsername());

        UsernamePasswordAuthenticationToken usernamePasswordAuthenticationToken =
                new UsernamePasswordAuthenticationToken(applicationUser.getUsername(),
                        applicationUser.getPassword(), emptyList());

        usernamePasswordAuthenticationToken.setDetails(applicationUser);

        return authenticationManager.authenticate(usernamePasswordAuthenticationToken);
    }

    @Override
    @SneakyThrows
    protected void successfulAuthentication(HttpServletRequest request,
                                            HttpServletResponse response,
                                            FilterChain chain, Authentication auth) {
        log.info("Authentication was successful for the user '{}'," +
                " generating JWE token", auth.getName());

        SignedJWT signedJWT = tokenCreator.createSignedJWT(auth);
```

FONTE**:** Produção Própria dos Autores

Figura 49 – Segunda parte da Classe JwtUsernameAndPasswordAuthenticationFilter

```
64
65          String encryptedToken = tokenCreator.encryptToken(signedJWT);
66
67          log.info("Token generated successfully, adding it to the response header");
68
69          response.addHeader( s: "Access-Control-Expose-Headers", s1: "XSRF-TOKEN, "
70                  + jwtConfiguration.getHeader().getName());
71
72          response.addHeader(jwtConfiguration.getHeader().getName(),
73                  s1: jwtConfiguration.getHeader().getPrefix() + encryptedToken);
74      }
75
76  }
```

FONTE**:** Produção Própria dos Autores

O método attemptAuthentication é responsável por tentar autenticar o usuário, delegando a responsabilidade para a classe UserDetailsServiceImpl, caso a autenticação seja bem-sucedida, passamos para o método successfulAuthentication, na linha 57 é chamado o método que cria o token assinado, logo após na linha 59 envia este token para ser criptografado.

Dentro da pasta security está a subpasta user, nela esta localizada a classe UserDetailsServiceImpl mostrada na figura 50.

Figura 50 - Classe UserDetailsServiceImpl

UserDetailsServiceImpl.java

```
1  package microservice.auth.security.user;
2
3  import lombok.RequiredArgsConstructor;
4  import lombok.extern.slf4j.Slf4j;
5  import microservice.core.model.ApplicationUser;
6  import microservice.core.repository.ApplicationUserRepository;
7  import org.springframework.beans.factory.annotation.Autowired;
8  import org.springframework.security.core.GrantedAuthority;
9  import org.springframework.security.core.userdetails.UserDetails;
10 import org.springframework.security.core.userdetails.UserDetailsService;
11 import org.springframework.security.core.userdetails.UsernameNotFoundException;
12 import org.springframework.stereotype.Service;
13
14 import javax.validation.constraints.NotNull;
15 import java.util.Collection;
16
17 import static org.springframework.security.core.authority
18         .AuthorityUtils.commaSeparatedStringToAuthorityList;
19
20 @Service
21 @Slf4j
22 @RequiredArgsConstructor(onConstructor = @__(@Autowired))
23 public class UserDetailsServiceImpl implements UserDetailsService {
24     private final ApplicationUserRepository applicationUserRepository;
25
26     @Override
27     public UserDetails loadUserByUsername(String username) {
28         log.info("Searching in the DB the user by username '{}'", username);
29
30         ApplicationUser applicationUser = applicationUserRepository.findByUsername(username);
31
32         log.info("ApplicationUser found '{}'", applicationUser);
33
34         if (applicationUser == null)
35             throw new UsernameNotFoundException(
36                     String.format("Application user '%s' not found", username));
37
38         return new CustomUserDetails(applicationUser);
39     }
40
41     private static final class CustomUserDetails
42             extends ApplicationUser implements UserDetails {
43         CustomUserDetails(@NotNull ApplicationUser applicationUser) {
44             super(applicationUser);
45         }
46
47         @Override
48         public Collection<? extends GrantedAuthority> getAuthorities() {
49             return commaSeparatedStringToAuthorityList( authorityString: "ROLE_" + this.getRole());
50         }
51
52         @Override
53         public boolean isAccountNonExpired() { return true; }
56
57         @Override
58         public boolean isAccountNonLocked() { return true; }
61
62         @Override
63         public boolean isCredentialsNonExpired() { return true; }
66
67         @Override
68         public boolean isEnabled() { return true; }
71     }
72 }
```

FONTE**:** Produção Própria dos Autores

Na figura acima é possível observar o método loadUserByUsename, este recebe um nome de usuário como parâmetro, busca esse usuário no banco a partir de seu nome e caso não o encontre, retorna uma exceção dizendo que o usuário informado não foi encontrado, caso contrário, é retornado o usuário, porém, neste caso será retornado um CustomUserDetails para que haja mais flexibilidade.

No CustomUserDetails localizado na linha 39 temos o método getAuthorities, este simplesmente retorna os perfis do usuário, caso o usuário tenha mais de um perfil, estes perfis

deverão ser guardados separadamente por uma vírgula para que este método retorne-os de forma correta quando solicitado.

A figura 51 mostra o arquivo application.yml do Auth com suas configurações necessárias.

Figura 51 - Arquivo de configurações do Auth

application.yml

```yaml
server:
  port: 8083

eureka:
  instance:
    prefer-ip-address: true
  client:
    service-url:
      defaultZone: http://discovery:8081/eureka/
    register-with-eureka: true


spring:
  application:
    name: auth
  jpa:
    show-sql: false
    hibernate:
      ddl-auto: update
    properties:
      hibernate:
        dialect: org.hibernate.dialect.MySQL8Dialect
  jmx:
    enabled: false
  datasource:
    url: jdbc:mysql://mysql:3306/microservice?allowPublicKeyRetrieval=true&sslMode=DISABLED
    username: root
    password: root

jwt:
  config:
    type: signed

```

FONTE**:** Produção Própria dos Autores

Como já descrito anteriormente, neste arquivo são configurados a porta, seu auto registro, o nome da aplicação, as configurações necessárias para a conexão com o banco de dados. A figura 52 mostra a classe AuthApplication, esta é a classe principal do serviço Auth.

Figura 52 - Classe AuthApplication

AuthApplication.java

```java
package microservice.auth;

import microservice.core.property.JwtConfiguration;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.autoconfigure.domain.EntityScan;
import org.springframework.boot.context.properties.EnableConfigurationProperties;
import org.springframework.cloud.netflix.eureka.EnableEurekaClient;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.data.jpa.repository.config.EnableJpaRepositories;

@SpringBootApplication
@EntityScan({"microservice.core.model"})
@EnableJpaRepositories({"microservice.core.repository"})
@EnableEurekaClient
@EnableConfigurationProperties(value = JwtConfiguration.class)
@ComponentScan("microservice")
public class AuthApplication {

    public static void main(String[] args) { SpringApplication.run(AuthApplication.class, args); }

}

```

FONTE: Produção Própria dos Autores

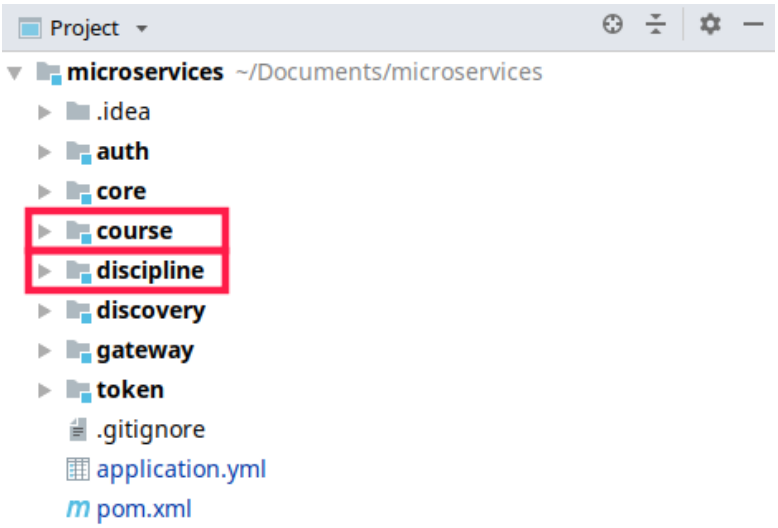
A classe mostrada na figura acima faz uso de algumas anotações do Spring, tais como: @EntityScan, esta mostra onde este serviço irá procurar suas entidades, é necessário informar dentro do parênteses o caminho que as entidades estão localizadas. A anotação @EnableJpaRepositories indica onde se localiza os repositórios utilizados por teste serviço. A anotação @EnableEurekaClient habilita este serviço para que ele se registre no serviço Discovery. A anotação @EnableConfigurationProperties informa para que este serviço utilize as configurações contidas na classe JwtConfiguration.

4.3 Desenvolvimento dos microsserviços para validação da arquitetura

Foram criados dois microsserviços para validação dessa arquitetura, estes serviram para exemplificar a integração entre os microsserviços, além de validar o uso do servidor de autenticação.

A figura 53 mostra a localização dos dois microsserviços na arquitetura, estes são descritos abaixo.

Figura 53 - Localização dos microsserviços Course e Discipline



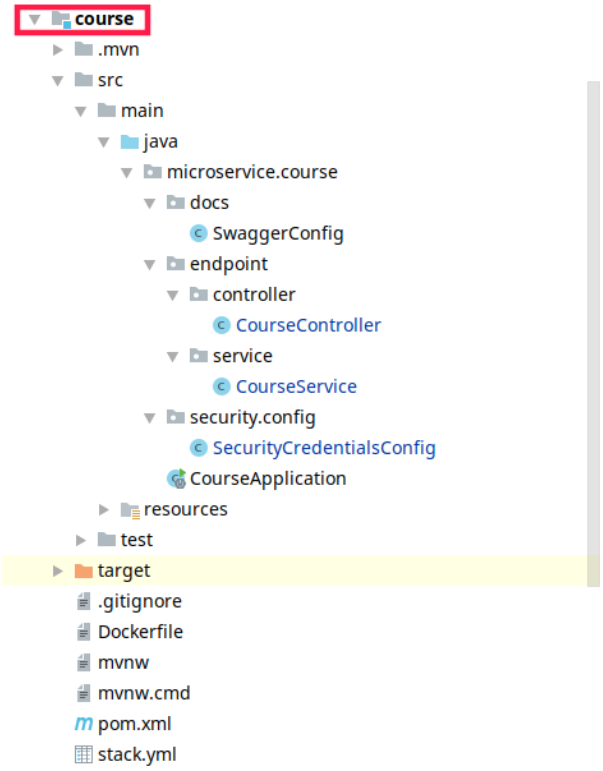
FONTE: Produção Própria dos Autores

Os serviços Course e Discipline foram criados com o intuito de exemplificar a interação entre os microsserviços, além de validar as camadas de segurança da aplicação, tanto como as de acesso exterior, ou seja, requisições recebidas pelo Gateway quanto à comunicação entre os serviços internos do sistema. Estes serviços realizam tarefas básicas como cadastrar, alterar e listar cursos e disciplinas.

4.3.1 Microsservice Course

A figura 54 mostra a estrutura de pastas do serviço Course, este contem as classes SwaggerConfig, CourseController, CourseService e SecurityCredentialsConfig, que serão descritas em seguida.

Figura 54 – Estrutura de pastas do serviço Course



FONTE: Produção Própria dos Autores

A classe SwaggerConfig localizada na pasta docs contem configurações referentes a documentação da API, conforme a figura 55 esta classe estende a classe BaseSwaggerConfig, nela estão a maioria das configurações da API, restando para a classe SwaggerConfig apenas passar o pacote onde se encontram os controladores do serviço Course.

Figura 55 – Classe SwaggerConfig

SwaggerConfig.java

```
1  package microservice.course.docs;
2
3  import microservice.core.docs.BaseSwaggerConfig;
4  import org.springframework.context.annotation.Configuration;
5  import springfox.documentation.swagger2.annotations.EnableSwagger2;
6
7  @Configuration
8  @EnableSwagger2
9  public class SwaggerConfig extends BaseSwaggerConfig {
10     public SwaggerConfig() {
11         super( basePackage: "microservice.course.endpoint.controller");
12     }
13 }
```

FONTE: Produção Própria dos Autores

A pasta endpoint é composta por duas subpastas, as pastas controller e service, na pasta controller se encontra a classe CourseController, a mesma é ilustrada na figura 56.

Figura 56 – Classe CourseController

CourseController.java

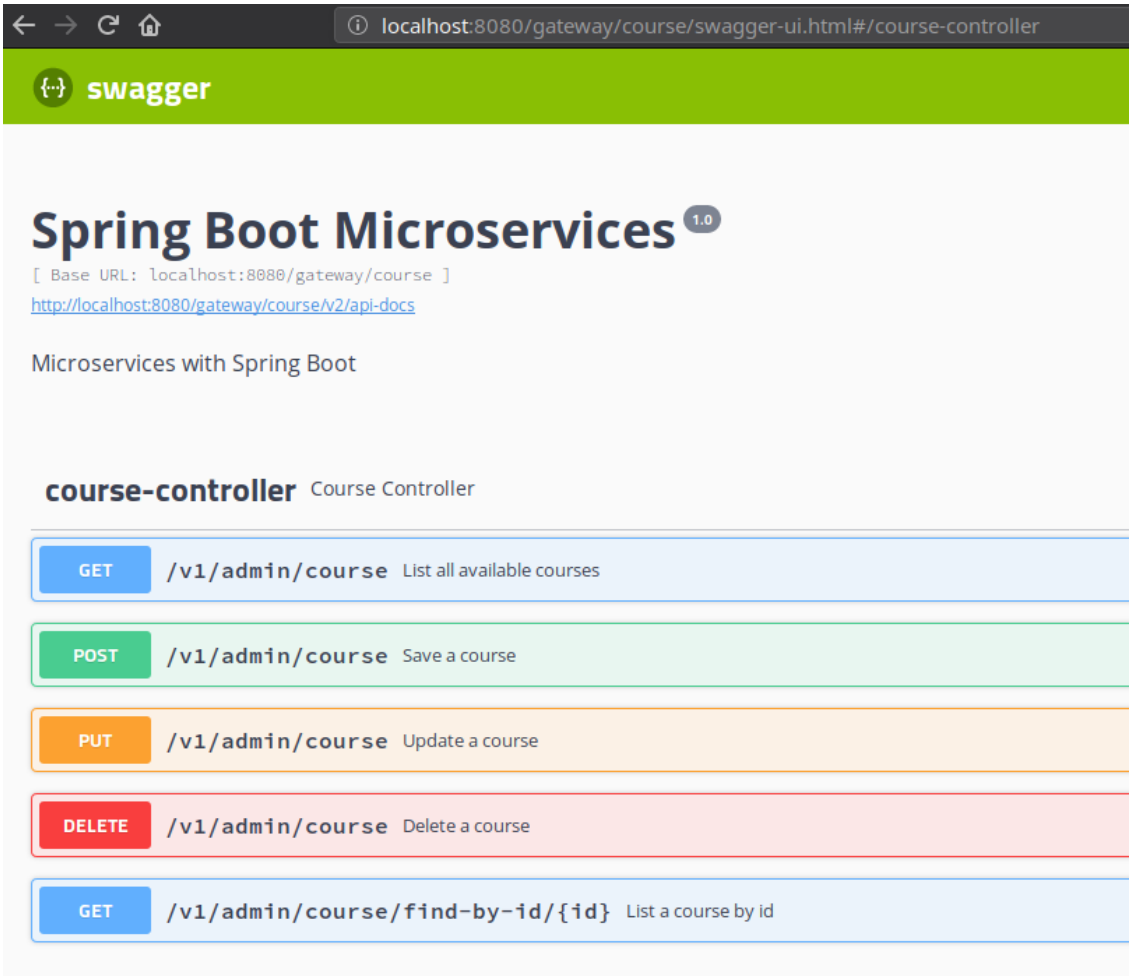
```java
package microservice.course.endpoint.controller;

import io.swagger.annotations.Api;
import io.swagger.annotations.ApiOperation;
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import microservice.core.model.Course;
import microservice.course.endpoint.service.CourseService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.domain.Pageable;
import org.springframework.http.HttpStatus;
import org.springframework.http.MediaType;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.Optional;

@RestController
@RequestMapping("v1/admin/course")
@Slf4j
@RequiredArgsConstructor(onConstructor = @__(@Autowired))
@Api(value = "Endpoints to manage course")
public class CourseController {
    private final CourseService courseService;

    @GetMapping(produces = MediaType.APPLICATION_JSON_UTF8_VALUE)
    @ApiOperation(value = "List all available courses", response = Course[].class)
    public ResponseEntity<Iterable<Course>> list(Pageable pageable) {
        return new ResponseEntity<>(courseService.list(pageable), HttpStatus.OK);
    }

    @CrossOrigin(origins = "*")
    @GetMapping(value = "/find-by-id/{id}", produces = MediaType.APPLICATION_JSON_UTF8_VALUE)
    @ApiOperation(value = "List a course by id", response = Course[].class)
    public ResponseEntity<Optional<Course>> findById(@PathVariable("id") Long id) {
        return new ResponseEntity<>(courseService.findById(id), HttpStatus.OK);
    }

    @PostMapping(produces = MediaType.APPLICATION_JSON_UTF8_VALUE)
    @ApiOperation(value = "Save a course", response = Course[].class)
    public ResponseEntity<Optional<Course>> save(@RequestBody Course course) {
        return new ResponseEntity(courseService.save(course), HttpStatus.OK);
    }

    @PutMapping(produces = MediaType.APPLICATION_JSON_UTF8_VALUE)
    @ApiOperation(value = "Update a course", response = Course[].class)
    public ResponseEntity<Optional<Course>> update(@RequestBody Course course) {
        return new ResponseEntity(courseService.update(course), HttpStatus.OK);
    }

    @DeleteMapping(produces = MediaType.APPLICATION_JSON_UTF8_VALUE)
    @ApiOperation(value = "Delete a course", response = Course[].class)
    public void deleteById(@RequestParam("id") Long id) { courseService.deleteById(id); }
}
```

FONTE: Produção Própria dos Autores

A figura acima mostra todos os endpoints referentes ao serviço Course, todos eles possuem uma URL base, ela é definida com a anotação @RequestMapping localizada na linha 19. As anotações @PostMapping, @GetMapping, @PutMapping e @DeleteMapping são referentes aos tipos de requisições HTTP, nestas anotações são definidas as URLs a partir da URL base e outras mais configurações. A anotação @ApiOperation faz parte da configuração do Swagger, nela é passada uma breve descrição do endpoint que será exibido na página de documentação da API.

A figura 57 ilustra como é documentada a API do serviço Course, este tipo de documentação facilita o entendimento dos endpoints que esta API disponibiliza e seus parâmetros necessários.

Figura 57 – Documentação da API Course



FONTE**:** Produção Própria dos Autores

O método list da figura 56 traz todos os cursos do banco de dados, este não recebe nenhum parâmetro. Já o método findById busca um único curso a partir do parâmetro id recebido durante a requisição. Os métodos save e update são semelhantes, ambos utilizam o mesmo endereço de URL e recebem um curso como parâmetro, porém, se diferenciam através dos tipos de requisição HTTP, sendo um do tipo POST e o outro do tipo PUT. O primeiro é indicado para enviar objetos para serem salvos, já o outro é recomendado para atualizar um objeto já existente na base de dados. Por fim o método delete recebe um id para que seja feita sua remoção do banco de dados.

A próxima subpasta da pasta endpoint é a pasta service, nela se localiza a classe CourseService, essa é responsável por conter as regras de negócio referentes ao serviço Course, como demonstra a figura 58.

Figura 58 – Classe CourseService

CourseService.java

```java
package microservice.course.endpoint.service;

import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import microservice.core.model.Course;
import microservice.core.repository.CourseRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.domain.Pageable;
import org.springframework.stereotype.Service;

import java.util.Optional;

@Service
@Slf4j
@RequiredArgsConstructor(onConstructor = @__(@Autowired))
public class CourseService {
    private final CourseRepository courseRepository;

    public Iterable<Course> list(Pageable pageable) {
        log.info("Listing all courses");
        return courseRepository.findAll(pageable);
    }

    public Optional<Course> findById(Long id) {
        log.info("Listing a course by id");
        return courseRepository.findById(id);
    }

    public Course save(Course course) {
        log.info("Saving a course...");
        return courseRepository.save(course);
    }

    public Course update(Course course) {
        log.info("Update a course...");
        return courseRepository.save(course);
    }

    public void deleteById(Long id) {
        log.info("Deleting a course...");
        courseRepository.deleteById(id);
    }

}
```

FONTE: Produção Própria dos Autores

A figura 58 demostra os métodos contidos na classe CourseService. Todos os seus métodos recebem um parâmetro e os enviam para a próxima camada, a CourseRepository. Isto ocorre devido ao baixo nível de complexidade deste serviço, já que o seu intuito é apenas validar a arquitetura. Entretanto, caso houvesse alguma validação ou regra de negócio a ser implementada, estas seriam feitas nesta classe.

A figura 59 apresenta a classe SecurityCredentialsConfig, esta estende a classe SecurityTokenConfig, que foi descrita no tópico Token. A classe SecurityCredentialsConfig contem apenas dois métodos, o primeiro é o construtor desta classe. A anotação @Override indica que o segundo método sobrescreve o método configure contido na classe SecurityTokenConfig, este método apenas adiciona um filtro ao método descrito anteriormente.

Figura 59 – Classe SecurityCredentialsConfig

SecurityCredentialsConfig.java

```java
package microservice.course.security.config;

import microservice.core.property.JwtConfiguration;
import microservice.security.config.SecurityTokenConfig;
import microservice.security.filter.JwtTokenAuthorizationFilter;
import microservice.security.token.converter.TokenConverter;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;

@EnableWebSecurity
public class SecurityCredentialsConfig extends SecurityTokenConfig {
    private final TokenConverter tokenConverter;

    public SecurityCredentialsConfig(JwtConfiguration jwtConfiguration,
                                     TokenConverter tokenConverter) {
        super(jwtConfiguration);
        this.tokenConverter = tokenConverter;
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.addFilterAfter(new JwtTokenAuthorizationFilter(jwtConfiguration, tokenConverter),
                UsernamePasswordAuthenticationFilter.class);
        super.configure(http);
    }

}
```

FONTE: Produção Própria dos Autores

A figura 60 apresenta a classe principal CourseApplication, esta possui anotações que já foram descritas anteriormente na descrição da figura 51 do tópico Auth. Esta nesta classe está contido o método main, este é o primeiro método a ser executado em uma aplicação java, em razão disto é chamada de classe principal.

Figura 60 – Classe CourseApplication

CourseApplication.java

```java
package microservice.course;

import microservice.core.property.JwtConfiguration;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.autoconfigure.domain.EntityScan;
import org.springframework.boot.context.properties.EnableConfigurationProperties;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.data.jpa.repository.config.EnableJpaRepositories;

@SpringBootApplication
@EntityScan({"microservice.core.model"})
@EnableJpaRepositories({"microservice.core.repository"})
@EnableConfigurationProperties(value = JwtConfiguration.class)
@ComponentScan("microservice")
public class CourseApplication {

    public static void main(String[] args) { SpringApplication.run(CourseApplication.class, args); }

}
```

FONTE: Produção Própria dos Autores

A figura 61 apresenta o arquivo de configurações do serviço Course, um arquivo igual a este já foi descrito anteriormente na descrição da imagem 49 no tópico Auth. Por eles se registrarem na mesma aplicação de descoberta de registros e utilizarem o mesmo banco de

dados, as configurações deste arquivo são praticamente as mesmas, diferenciando-se apenas o nome da aplicação e a porta a ser utilizada.

Figura 61 – Arquivo de configurações do serviço Course

application.yml

```yaml
server:
  port: 8082

eureka:
  instance:
    prefer-ip-address: true
  client:
    service-url:
      defaultZone: http://discovery:8081/eureka/
    register-with-eureka: true


spring:
  application:
    name: course
  jpa:
    show-sql: false
    hibernate:
      ddl-auto: update
    properties:
      hibernate:
        dialect: org.hibernate.dialect.MySQL8Dialect
  jmx:
    enabled: false
  datasource:
    url: jdbc:mysql://mysql:3306/microservice?allowPublicKeyRetrieval=true&sslMode=DISABLED
    username: root
    password: 'root'

```

FONTE: Produção Própria dos Autores

Na figura 62 é apresentada a interface CourseRepository, que estende a interface PagingAndSortingRepository e ela a interface CrudRepository, nesta estão os métodos deste serviço que acessam o banco de dados, como os métodos save, findAll, findById e outros. Caso seja necessário realizar alguma consulta mais específica, esta deverá ser implementada na interface CourseRepository.

Figura 62 – Interface CourseRepository

CourseRepository.java

```java
package microservice.core.repository;


import microservice.core.model.Course;
import org.springframework.data.repository.PagingAndSortingRepository;

public interface CourseRepository extends PagingAndSortingRepository<Course, Long> {
}
```

FONTE: Produção Própria dos Autores

A classe Course é apresentada na figura 63, esta representa a entidade course, cujo serviço Course tem a responsabilidade de cadastrar, alterar e listar.

Figura 63 – Classe Course

Course.java

```java
package microservice.core.model;

import lombok.*;

import javax.persistence.*;
import javax.validation.constraints.NotNull;

@Entity
@Getter
@Setter
@Builder
@NoArgsConstructor
@AllArgsConstructor
@ToString
@EqualsAndHashCode(onlyExplicitlyIncluded = true)
public class Course implements AbstractEntity {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @EqualsAndHashCode.Include
    private Long id;
    @NotNull(message = "The field 'title' is mandatory")
    @Column(nullable = false)
    private String title;
}
```

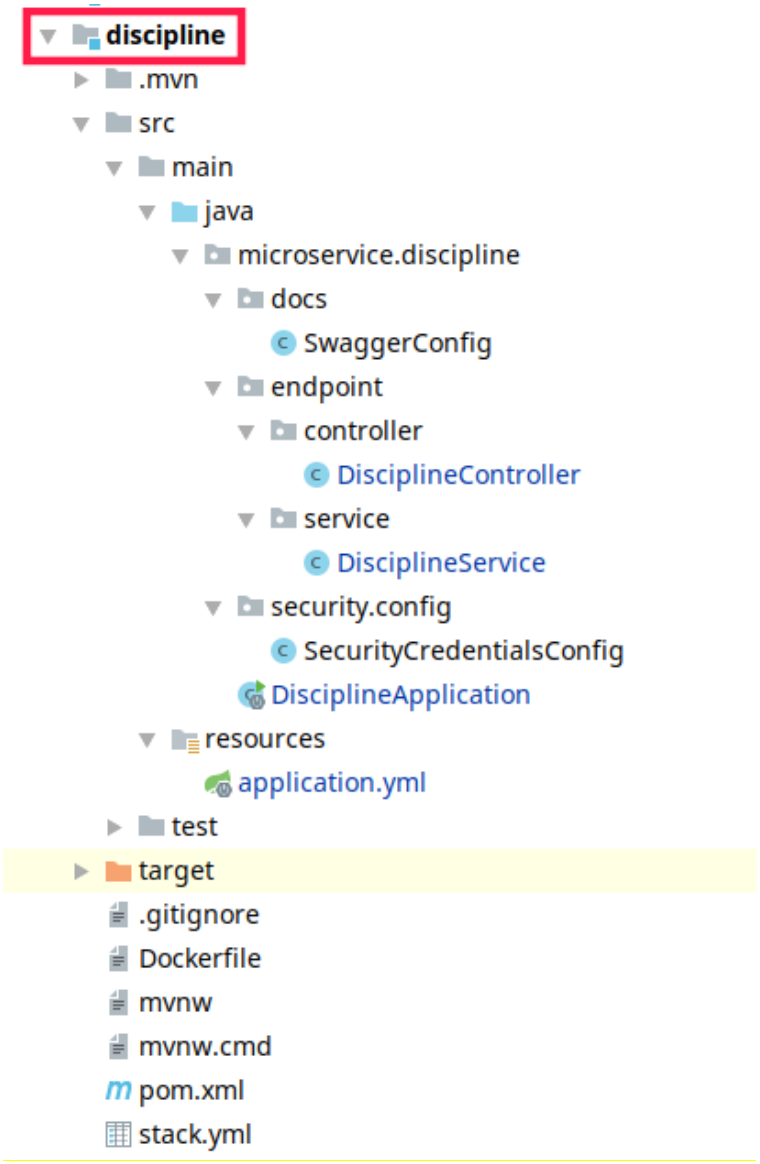
FONTE: Produção Própria dos Autores

A figura 62 apresenta a entidade Course e seus atributos id e title. Quatro anotações que merecem destaque são as anotações @Getter, @Setter, @EqualsAndHashCode e @ToString, elas simplificam a classe deixando-a com menos código, já que estas anotações geram automaticamente seus respectivos códigos no momento da compilação da classe.

4.3.2 Microsservice Discipline

A figura 64 demonstra a estrutura de pastas do serviço Discipline, este contem as classes SwaggerConfig, DisciplineController, DisciplineService e SecurityCredentialsConfig. Devido a sua semelhança com o serviço Course, serão descritos somente as classes DisciplineService e Discipline, pois este serviço Discipline contém regras de negócio específicas.

Figura 64 – Estrutura de pastas do serviço Discipline



FONTE: Produção Própria dos Autores

A classe DisciplineService é responsável por conter as regras de negócio referentes ao serviço Discipline, como demonstram as figuras 65 e 66.

Figura 65 – Classe DisciplineService

DisciplineService.java

```java
package microservice.discipline.endpoint.service;

import com.netflix.appinfo.InstanceInfo;
import com.netflix.discovery.EurekaClient;
import com.netflix.hystrix.contrib.javanica.annotation.HystrixCommand;
import javassist.NotFoundException;
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import microservice.core.model.Course;
import microservice.core.model.Discipline;
import microservice.core.repository.DisciplineRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.domain.Pageable;
import org.springframework.http.HttpEntity;
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpMethod;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Service;
import org.springframework.web.client.RestTemplate;

import java.util.Collections;
import java.util.Optional;

@Service
@Slf4j
@RequiredArgsConstructor(onConstructor = @__(@Autowired))
public class DisciplineService {

    private final DisciplineRepository disciplineRepository;
    private final RestTemplate restTemplate = new RestTemplate();
    private final EurekaClient eurekaClient;

    public Iterable<Discipline> list(Pageable pageable) {
        log.info("Listing all disciplines");
        return disciplineRepository.findAll(pageable);
    }

    public Optional<Discipline> findById(Long id) {
        Optional<Discipline> discipline = disciplineRepository.findById(id);
        log.info("Listing a discipline by id");
        return discipline;
    }

    public Discipline save(Discipline discipline, String authHeader) throws NotFoundException {
        if (!validateCourse(discipline, authHeader)) {
            log.info("Invalid course id");
            throw new NotFoundException("Invalid course id");
        } else {
            log.info("Saving a discipline...");
            return disciplineRepository.save(discipline);
        }
    }

    public Discipline update(Discipline discipline, String authHeader) throws NotFoundException {
        if (!validateCourse(discipline, authHeader)) {
            log.info("Invalid course id");
            throw new NotFoundException("Invalid course id");
        } else {
            log.info("Update a discipline...");
            return disciplineRepository.save(discipline);
        }
    }
```

FONTE: Produção Própria dos Autores

Figura 66 – Segunda parte da Classe DisciplineService

```
63
64      public void deleteById(Long id) {
65          log.info("Deleting a discipline...");
66          disciplineRepository.deleteById(id);
67      }
68
69      private String getUrlServicoCurso() {
70          InstanceInfo info = eurekaClient
71                  .getApplication( appName: "course").getInstances().iterator().next();
72          if (info == null) {
73              log.info("Unavailable service");
74          }
75          return String.format("http://%s:%d/v1/admin/course/", info.getHostName(), info.getPort());
76      }
77
78      @HystrixCommand(fallbackMethod = "unavailableService")
79      private boolean validateCourse(Discipline discipline, String authHeader) {
80          HttpHeaders headers = new HttpHeaders();
81          headers.set("Authorization", authHeader);
82
83          HttpEntity entity = new HttpEntity(headers);
84
85          String url = getUrlServicoCurso().concat(Long.toString(discipline.getCourseId()));
86
87          ResponseEntity<Course> response = restTemplate
88                  .exchange(url, HttpMethod.GET, entity, Course.class, Collections.emptyMap());
89          if (!response.getStatusCode().is2xxSuccessful() || null == response.getBody())
90              return false;
91          return true;
92      }
93
94      public void unavailableService() throws NotFoundException {
95          throw new NotFoundException("Unavailable service!");
96      }
```

FONTE: Produção Própria dos Autores

As figuras 65 e 66 demostram os métodos contidos na classe DisciplineService. Alguns métodos são semelhantes aos já descritos na classe CourseService, como os métodos list, findById e delete. Os métodos save e update possuem uma regra de negócio que exige a validação do identificador do Course, este só é permitido cadastrar ou alterar uma disciplina que tenha um identificador de Course válido.

A fim de garantir esta consistência de dados, antes de cadastrar ou alterar uma disciplina é chamado o método booleano validateCourse mostrado na figura 66, que retornará true caso o identificador do curso seja válido e false caso contrário.

Para verificar a existência do curso, o método validateCourse consulta o serviço Course enviando um identificador como parâmetro, retornando um objeto curso caso exista. Entretanto, para que esta consulta seja realizada deve-se saber qual URL utilizar para a consulta, para isso o método validateCourse chama o método getUrlServicoCurso, este irá consultar o serviço Discovery, o host e a porta onde serviço Course se encontra disponível. Em seguida é retornada a URL e é feita uma requisição do tipo GET para o serviço Course enviando o identificador do curso como parâmetro. Caso seja encontrado algum curso, o método getUrlServicoCurso retorna true e a disciplina é salva na base de dados. Entretanto, caso ocorra algum erro durante a requisição que solicita um curso, seja por indisponibilidade

do serviço ou qualquer outro erro, a anotação @HystrixCommand gera um fallBack chamando o método unavailableService que lança uma excessão.

A figura 67 apresenta a entidade Discipline, que assim como a entidade Course faz uso de anotações para a redução de código. Esta entidade possui os atributos id, title e courseId.

Figura 67 – Classe Discipline

Discipline.java

```
1  package microservice.core.model;
2
3  import lombok.*;
4
5  import javax.persistence.*;
6  import javax.validation.constraints.NotNull;
7
8  @Entity
9  @Getter
10 @Setter
11 @Builder
12 @NoArgsConstructor
13 @AllArgsConstructor
14 @ToString
15 @EqualsAndHashCode(onlyExplicitlyIncluded = true)
16 public class Discipline implements AbstractEntity {
17     @Id
18     @GeneratedValue(strategy = GenerationType.IDENTITY)
19     @EqualsAndHashCode.Include
20     private Long id;
21     @NotNull(message = "The field 'title' is mandatory")
22     @Column(nullable = false)
23     private String title;
24     @NotNull(message = "The field 'courseId' is mandatory")
25     @Column(nullable = false)
26     private Long courseId;
27 }
28
```

FONTE: Produção Própria dos Autores

CAPÍTULO 5 - CONSIDERAÇÕES FINAIS

A realização desse projeto constituiu em uma experiência fundamental para o crescimento profissional, acadêmico e pessoal. Com base no presente estudo foi definida e implementada a arquitetura de software baseada em microsserviços. As etapas alcançadas para este feito foram concluídas, sendo elas a criação do documento arquitetural e do desenvolvimento da arquitetura.

Como resultado desse trabalho foi gerado o documento arquitetural, este contem o escopo da arquitetura, os componentes e as visões arquiteturais, além da descrição do passo a passo para integração de novos sistemas a esta arquitetura.

Foi criada a arquitetura de microsserviços que tem por objetivo a padronização e a integração dos novos sistemas desenvolvidos pelos acadêmicos. Tendo como parte dela dois serviços utilizados em sua validação, na qual foi possível observar a comunicação entre eles, além da possibilidade de acessar qualquer serviço dentro dessa arquitetura, utilizando um login único.

Através deste estudo e da arquitetura desenvolvida, espera-se contribuir efetivamente para a comunidade acadêmica compartilhando o conhecimento obtido pelos autores. Dentre essas contribuições podemos citar a definição de uma arquitetura de software baseado em Microsserviços para padronização de novos sistemas para o curso de bacharelado em computação do centro universitário UniEvangélica. Além do documento arquitetural que fornece visões da arquitetura para representar vários aspectos do sistema.

Durante o desenvolvimento desse projeto foi encontrado dificuldades e limitações, porem esses não comprometeram o projeto em si. A principal dificuldade foi entender o fluxo de dados e a responsabilidade de cada componente, postergando o desenvolvimento dos artefatos.

É considerada limitação deste projeto a dificuldade que a estrutura do trabalho impõe em relação ao prazo e ao tamanho do estudo. Isso limitou a utilização da arquitetura proposta em apenas um caso, tendo sido implementado apenas dois protótipo para validação da mesma, cujo objetivo foi apenas para demonstrar a utilização da arquitetura.

O fato de ter usado protótipos gerou outra limitação para o projeto, não sendo viável a implementação do componente Ribbon, porque ele é utilizado em um cenário de alta concorrência comprovada pelo usuário, pois ele trata da escalabilidade horizontal, ou seja, ter várias instancia do mesmo serviço.

Para facilitar a infraestrutura e a utilização do trabalho proposto, foram utilizados contêineres Docker, devido a sua portabilidade e da desnecessidade de configurar todo um ambiente para a execução do projeto, contribuindo para um desenvolvimento ágil.

Para a implementação da arquitetura foi utilizado o framework Spring Boot, tornando possível a obtenção de uma maior produtividade durante o desenvolvimento da arquitetura, além da utilização dos componentes que fazem parte do Spring Cloud Netflix OSS, estes possuem diferentes tipos de soluções distribuídas que facilitou o desenvolvimento.

Vantagens: as melhores práticas de baixo acoplamento são levados a níveis arquiteturais, assim como algumas QoS como resiliência/disponibilidade e escalabilidade horizontal. Além disso, quando calibrada a granularidade dos serviços fica mais fácil à gestão dos produtos e times de software, aumentando a eficiência das metodologias administrativas.

Entretanto, os serviços por si necessitam de orquestração e operação qualificada, aumentando a pilha de tecnologias envolvidas para criar o ambiente mais próximo do ideal para produção.

A coesão dos serviços e a sua descartabilidade são muito aderentes a modelos evolutivos e ágeis de produção de software. Em contrapartida, a pluralidade e multidisciplinaridade exigida aumenta a curva de aprendizado. Se os domínios dos modelos não estão em estágio de maturidade que permita dividir em pacotes dentro de uma mesma aplicação, a arquitetura se torna um risco.

Tendo tudo isso em vista, microsserviços são o padrão de facto da indústria atual de software como serviço (SaaS) pois as qualidades inerentes à melhoria contínua pagam com tranquilidade os requisitos de qualificação técnica para o melhor encapsulamento (BHOJWANI, 2018).

Pretende-se para trabalhos futuros a implementação do componente Ribbon na arquitetura de microsserviços, este permite distribuir requisições uniformemente entre as instancias de serviços. Além da realização de testes de qualidade na arquitetura projetada.

REFERÊNCIAS BIBLIOGRÁFICAS

ALBUQUERQUE, Diego Macêdo. **Arquitetura de aplicações em 2, 3, 4 ou N camadas.** 2012. Disponível em: <https://www.diegomacedo.com.br/arquitetura-de-aplicacoes-em-2-3-4-ou-n-camadas/>. Acesso em: 10/11/2018.

ARMIGLIATTO, Gilherme Moraes. **REST usa JSON e SOAP usa XML, certo?.** 2017. Disponível em: <http://www.matera.com/blog/post/rest-usa-json-e-soap-usa-xml-certo>. Acesso em: 04/05/2019.

ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS. **NBR ISO/IEC 9126-1:** Engenharia de software - Qualidade de produto. Rio de Janeiro, 2003.

BACK, Renato Pereira. **Análise Comparativa de Técnicas de Integração entre Microsserviços.** Santa Catarina, 2016. Disponível em: <https://repositorio.ufsc.br/bitstream/handle/123456789/171437/Relat%C3%B3rio%20Final%20TCC%20com%20Artigo%20SBC%20-%20Renato%20Back%20-%20SIN%202016-2.pdf?sequence=1&isAllowed=y>. Acesso em: 10/05/2018.

BHOJWANI, Rajesh**. Design Patterns for Microservices.** Disponível em: <https://dzone.com/articles/design-patterns-for-microservices> Acesso em: 09/05/2019.

BUSCHMANN, Frank. **Pattern-Oriented Software Architecture, A System Of Patterns.** Vol. 1. West Sussex: Wiley, 2001. Disponível em:<http://disi.unal.edu.co/dacursci/sistemasycomputacion/docs/SWEBOK/John%20Wiley%20&%20Sons%20-%20Pattern-Oriented%20Software%20Architecture,%20A%20System%20Of%20Patterns,%20Volume%201.pdf> Acesso em 05/11/18.

CARVALHO, Lucas Silva Pedatela; ANJOS, Matheus César Lopes dos. **Impacto dos padrões arquiteturais de Micro Serviço e Monolítico no desenvolvimento de softwares.** Anápolis, 2017. Disponível em: <http://repositorio.aee.edu.br/handle/aee/50> Acesso em: 08/11/18

CARVALHO, Luis Heustákio Lima. **Uma Abordagem De Migração Para Arquitetura Microservices A Partir De Aplicações Monolíticas Em Produção**. Fortaleza, 2017. Dissertação de mestrado: Ciência da Computação. Disponível em: <https://uol.unifor.br/oul/ObraBdtdSiteTrazer.do?method=trazer&ns=true&obraCodigo=104388>. Acesso em: 23/10/2018.

CHAND, Swatee**. Spring Tutorial – A Java Framework Providing Efficiency.** Disponível em: <https://www.edureka.co/blog/spring-tutorial/> Acesso em: 09/05/2019.

COSTA, Monise. **Swagger: Como gerar uma documentação interativa para API REST.** 2016. Disponível em: <http://www.matera.com/blog/post/swagger-como-gerar-uma-documentacao-interativa-para-api-rest> Acesso em: 08/05/19

DIEDRICH, Cristiano. **O que é dockerfile.** 2015. Disponível em: <https://www.mundodocker.com.br/o-que-e-dockerfile/>. Acesso em: 01/05/2019.

DOCKER. **Recursos do container**. 2013. Disponível em: <https://www.docker.com/resources/what-container >. Acesso em: 01/05/2019.

ELLIOTT, Eric. **10 Interview Questions Every JavaScript Developer Should Know.** 2015. Disponível em: <https://medium.com/javascript-scene/10-interview-questions-every-javascript-developer-should-know-6fa6bdf5ad95>. Acesso em: 30/04/2018.

FALBO, Ricardo de Almeida, BARCELLOS, Monalessa Perine. **Engenharia de software: Notas de aula PARTE II**, 2011. Disponível em: <https://www.inf.ufes.br/~monalessa/PaginaMonalessa-NEMO/ES/NotasDeAula-EngSoftware-EngComp-Parte-II.pdf>. Acesso em 24/04/2018

FERNÁNDEZ, José, M. **Java. Parte I.** 1998. Disponível em: <http://www.linuxfocus.org/Portugues/July1998/article57.html>. Acesso em: 01/05/2019.

FERREIRA, Cleber de F. MOTA, Roberto Dias. **COMPARANDO APLICAÇÃO WEB SERVICE REST E SOAP**, 2014. Disponível em: <http://web.unipar.br/~seinpar/2014/artigos/pos/Cleber_de_F_Ferreira_Roberto_Dias_Mota%20(1).pdf>. Acesso em 24/04/2019

FERREIRA, Rodrigo. **REST: Princípios e boas práticas**, 2017. Disponível em: <https://blog.caelum.com.br/rest-principios-e-boas-praticas/>. Acesso em 24/04/2019

FILHO, Gerson Luis Ferreira Da Silva**. Desenvolvimento De Aplicativo Para Adoção de Animais Abandonados Utilizando a Linguagem de Programação Kotlin e Programação Reativa.** Curitiba, 2017. Disponível em: <http://repositorio.roca.utfpr.edu.br/jspui/bitstream/1/8462/1/CT_COENC_2017_1_4.pdf> Acesso em: 23/10/18

FOWLER, M. **Microservices, a definition of this new architectural term**, 2014. Disponível em: <https://martinfowler.com/articles/microservices.html> Acesso em 24/04/2018.

GOPAL, Senthilkumar. **Application Resiliency Using Netflix Hystrix.** 2015. Disponível em: <https://www.ebayinc.com/stories/blogs/tech/application-resiliency-using-netflix-hystrix/>. Acesso em: 10/05/2019.

GRANATYR, Jones. **Introdução ao modelo multicamadas.** 2007. Disponível em: <https://www.devmedia.com.br/introducao-ao-modelo-multicamadas/5541>. Acesso em: 10/11/2018.

JAMES, Geordy. **REST API Architecture**. 2017. Disponível em: <https://shareurcodes.com/blog/creating%20a%20simple%20rest%20api%20in%20php> Acesso em: 09/05/2019

JWT. **Introdução aos Tokens Web JSON.** 2014. Disponível em: <https://jwt.io/introduction/>. Acesso em: 01/05/2019.

LACERDA, Henrique. **Quais os beneficios da arquitetura REST?**, 2014. Disponível em: <http://www.matera.com/blog/post/quais-os-beneficios-da-arquitetura-rest.> Acesso em: 01/05/2019.

LARSSON, Magnus. **Building Microservices with Spring Cloud and Netflix OSS, part 1.** 2015. Disponível em: <http://callistaenterprise.se/blogg/teknik/2015/04/10/building-microservices-with-spring-cloud-and-netflix-oss-part-1/> Acesso em 07/11/2018.

LUKYANCHIKOV, Alexander. **Microservice Architectures With Spring Cloud and Docker.** 2016. Disponível em: <https://dzone.com/articles/microservice-architecture-with-spring-cloud-and-do> Acesso em: 15/10/18

MACORATTI, José Carlos. **MVC 3 - Ordenação, Filtragem e Paginação com EF.** 2011. Disponível em: <http://www.macoratti.net/12/01/aspn_mvc32.htm> Acesso em: 07/06/19

MATIOLI, Danilo Cavassini. **Importância da segurança em banco de dados.** 2010. Disponível em: < https://cepein.femanet.com.br/BDigital/arqTccs/0811060524.pdf> Acesso em: 15/04/19

MAVEN. **Bem vindo ao Apache Maven**. 2011. Disponível em: <https://maven.apache.org/>. Acesso em 18/03/2019.

MICROSOFT. **The Model-View-ViewModel Pattern.** 2017. Disponível em: <https://docs.microsoft.com/pt-br/xamarin/xamarin-forms/enterprise-application-patterns/mvvm> Acesso em: 06/11/18

MITRA, Shamik. **Microservices Tutorial: Ribbon as a Load Balancer.** 2017. Disponível em: <https://dzone.com/articles/microservices-tutorial-ribbon-as-a-load-balancer-1> Acesso em: 07/11/18

NASCIMENTO. Wellington. **Entendendo tokens JWT (Json Web Token)**. 2018. Disponível em: <https://medium.com/tableless/entendendo-tokens-jwt-json-web-token-413c6d1397f6> Acesso em: 18/05/2019

NETFLIX. **Zuul**. 2018. Disponível em: <https://github.com/Netflix/zuul/wiki> Acesso em: 08/05/2019

NEWMAN, Sam. **Building microservices: Designing fine-grained systems**. Sebastopol: Editora O'Reilly media, 2015.

OLIBONI, Daniel. **O que é um SGBD?** 2016. Disponível em: <https://www.oficinadanet.com.br/post/16631-o-que-e-um-sgbd>. Acesso em: 14/05/2019.

OLIVEIRA, Edson Mendes. **Vantagens e desvantagens de SOA.** 2013. Disponível em: <https://www.devmedia.com.br/vantagens-e-desvantagens-de-soa/27437>. Acesso em: 10/11/2018.

OLIVEIRA, Thiago Barros. **Estudo e Avaliação de Boas Práticas de Engenharia de Software para o Desenvolvimento de um Sistema Web Visando a Melhoria da Produtividade e Manutenabilidade.** Recife, 2016. Disponível em: <http://thiagoti.com/tcc_public.pdf>. Acesso em 19/06/2018.

PIZA, Pedro. **O que é e como usar o MySQL?** 2012. Disponível em: <https://www.techtudo.com.br/artigos/noticia/2012/04/o-que-e-e-como-usar-o-mysql.html>. Acesso em 14/05/2019.

РУСИН, Марія. **Maven pom.xml.** 2015. Disponível em: <https://www.slideshare.net/ssuser220b38/maven-54639726>. Acesso em 08/05/2019.

RISHABH. **Understanding The Difference Between MVC, MVP and MVVM Design Patterns.** 2016. Disponível em: <https://www.linkedin.com/pulse/understanding-difference-between-mvc-mvp-mvvm-design-rishabh-software/>. Acesso em 23/10/2018.

SALERNO, Rafael. **Desmistificando o Spring Cloud Netflix**. 2017. Disponível em: <https://www.infoq.com/br/articles/desmistificando-spring-cloud-netflix>. Acesso em 08/05/2019.

SANDOVAL, Kristopher. **Why Can't I Just Send JWTs Without OAuth?**. 2017. Disponível em: < https://nordicapis.com/why-cant-i-just-send-jwts-without-oauth/>. Acesso em 01/05/2019.

SANTANA, Eduardo. **Conheça o fascinante conceito de Enterprise Service Bus**. 2015. Disponível em: < http://bufallos.com.br/bg_br/conheca-o-fascinante-conceito-de-enterprise-service-bus/>. Acesso em 07/06/2019.

SANT'ANNA, Mauro. **SOAP e WebServices**. 2015. Disponível em: <http://www.linhadecodigo.com.br/artigo/38/soap-e-webservices.aspx>. Acesso em 01/05/2019.

SOMMERVILLE, Ian. **Engenharia de Software.** Tradução: BOSNIC, Ivan; GONÇALVES, Kalinka G. de O. 9 ed. São Paulo: Pearson Prentice Hall, 2013. Disponível em: <ftp://ftpaluno.umc.br/Aluno/Daisy/Engenharia%20de%20Software/Engenharia_Software_3Edicao%20sommerville.pdf> Acesso em 20/10/18.

SOUZA, Jamila Peripolli. Desenvolvendo microsserviços com Spring Cloud Netflix. 2018. Disponível em: < http://www.matera.com/blog/post/desenvolvendo-microsservicos-spring-cloud-netflix>. Acesso em 09/05/2019.

SOUZA, Marcio. Como começar com Spring?. Disponível em: <https://www.devmedia.com.br/exemplo/como-comecar-com-spring/73>. Acesso em 01/05/2019.

SPRING. Docs. Disponível em: <http://spring.io/docs/reference>. Acesso em 18/ 03/2019.

SPRING. Spring initializr. Disponível em: <https://start.spring.io/>. Acesso em 08/05/2019.

SPRING. Spring **Cloud Security.** 2014. Disponível em: <http://spring.io/projects/spring-cloud-security>. Acesso em 09/11/2018.

STENBERG, Jan. **Microserviços, Containers e o Docker.** 2015. Disponível em: <https://www.infoq.com/br/news/2015/02/microservices-docker>. Acesso em: 10/04/2019.

STOIBER, M. **Build your first Node.js microservice.** 2017. Disponível em: <https://mxstbr.blog/2017/01/your-first-node-microservice/>. Acesso em 29/03/2018.

SUANE, Willian. **Spring Boot Microservices.** 2019. Disponível em: <https://www.youtube.com/watch?v=vxeMnM15gsI&list=PL62G310vn6nH_iMQoPMhIlK_ey1npyUUl>. Acesso em: 02/01/2019.

SWAGGER. **Sobre o Swagger.** 2018. Disponível em: <https://swagger.io/about/>. Acesso em 01/05/2019.

SWAGGER. **Editor Swagger.** 2019. Disponível em: <https://editor.swagger.io/?_ga=2.226731486.212367077.1557079840-996798822.1556723757>. Acesso em 09/05/2019.

TILKOV, Stefan. **Uma rápida Introdução ao REST.** 2008. Disponível em: <https://www.infoq.com/br/articles/rest-introduction/>. Acesso em: 10/04/2019.

TRIPOLI, Crislaine da silva; CARVALHO, Rodrigo Pimenta. **Micros-serviços:** características, benefícios e desvantagens em relação à arquitetura monolítica que impactam na decisão do uso desta arquitetura**.** Minas Gerais, 2016. Disponível em: http://docplayer.com.br/54537593-Micros-servicos-caracteristicas-beneficios-e-desvantagens-em-relacao-a-arquitetura-monolitica-que-impactam-na-decisao-do-uso-desta-arquitetura.html> Acesso em 06/11/18.

TRUCCO, Cristian. **DOCKER COMPOSER: O QUE É? PARA QUE SERVE? O QUE COME.** 2017. Disponível em: < https://www.concrete.com.br/2017/12/11/docker-compose-o-que-e-para-que-serve-o-que-come/>. Acesso em: 10/04/2019.

VAROTO, Ane Cristina. **Visões em arquitetura de software.** Dissertação: Mestre em Ciências da Computação. São Paulo, 2002. Disponível em: <http://bdpi.usp.br/item/001247269>. Acesso em: 10/05/2018.

APÊNDICES

APÊNDICE 1 – DOCUMENTO ARQUITETURAL

DOCUMENTO ARQUITETURAL

ARQUITETURA DE MICROSSERVIÇO

VERSÃO: 01

Autores:
JOSÉ FRANCISCO DE OLIVEIRA JÚNIOR
MAYCON DA SILVA MOREIRA

ANÁPOLIS
2019