

**MINISTÉRIO DA DEFESA  
EXÉRCITO BRASILEIRO  
DEPARTAMENTO DE CIÊNCIA E TECNOLOGIA  
INSTITUTO MILITAR DE ENGENHARIA  
CURSO DE GRADUAÇÃO EM ENGENHARIA DE COMPUTAÇÃO**

**1º Ten IGOR CESAR TORRES DE OLIVEIRA  
1º Ten LUCAS MENDES SANTOS SILVA**

**UTILIZAÇÃO DE MICROSERVIÇOS NA ARQUITETURA DE  
SISTEMAS DE ALTA ESCALABILIDADE E CRITICIDADE**

**Rio de Janeiro  
2016**

**INSTITUTO MILITAR DE ENGENHARIA**

**1º Ten IGOR CESAR TORRES DE OLIVEIRA**

**1º Ten LUCAS MENDES SANTOS SILVA**

**UTILIZAÇÃO DE MICROSERVIÇOS NA ARQUITETURA  
DE SISTEMAS DE ALTA ESCALABILIDADE E  
CRITICIDADE**

Projeto de Fim de Curso apresentado ao Curso de Graduação em Engenharia de Computação do Instituto Militar de Engenharia, como requisito parcial para a obtenção do título de Engenheiro de Computação.

Orientador: Prof. Clayton Escouper das Chagas - M.Sc.

Rio de Janeiro  
2016

c2016

INSTITUTO MILITAR DE ENGENHARIA

Praça General Tibúrcio, 80 – Praia Vermelha

Rio de Janeiro – RJ CEP: 22290-270

Este exemplar é de propriedade do Instituto Militar de Engenharia, que poderá incluí-lo em base de dados, armazenar em computador, microfilmar ou adotar qualquer forma de arquivamento.

É permitida a menção, reprodução parcial ou integral e a transmissão entre bibliotecas deste trabalho, sem modificação de seu texto, em qualquer meio que esteja ou venha a ser fixado, para pesquisa acadêmica, comentários e citações, desde que sem finalidade comercial e que seja feita a referência bibliográfica completa.

Os conceitos expressos neste trabalho são de responsabilidade do(s) autor(es) e do(s) orientador(es).

004.22	Oliveira, Igor Cesar Torres de
O48u	Utilização de microsserviços na arquitetura de sistemas de alta escalabilidade e criticidade / Igor Cesar Torres de Oliveira, Lucas Mendes Santos Silva; orientados por Clayton Escouper das Chagas – Rio de Janeiro: Instituto Militar de Engenharia, 2016.  90p. : il.  Projeto de Fim de Curso (PROFIC) – Instituto Militar de Engenharia, Rio de Janeiro, 2016.  1. Curso de Engenharia de Computação – Projeto de Fim de Curso. 2. Arquitetura de computadores. 3. Sistemas. I. Silva, Lucas Mendes Santos. II. Chagas, Clayton Escouper das. III. Título. IV. Instituto Militar de Engenharia.

**INSTITUTO MILITAR DE ENGENHARIA**

**1º Ten IGOR CESAR TORRES DE OLIVEIRA**  
**1º Ten LUCAS MENDES SANTOS SILVA**

**UTILIZAÇÃO DE MICROSERVIÇOS NA ARQUITETURA  
DE SISTEMAS DE ALTA ESCALABILIDADE E  
CRITICIDADE**


Projeto de Fim de Curso apresentado ao Curso de Graduação em Engenharia de Computação do Instituto Militar de Engenharia, como requisito parcial para a obtenção do título de Engenheiro de Computação.

Orientador: Prof. Clayton Escouper das Chagas - M.Sc.

Aprovado em 26 de Setembro de 2016 pela seguinte Banca Examinadora:

  
\_\_\_\_\_  
Prof. Clayton Escouper das Chagas - M.Sc. do IME - Presidente

  
\_\_\_\_\_  
Prof. Ricardo Choren Noya - D.Sc. do IME

  
\_\_\_\_\_  
Prof.ª Raquel Coelho Gomes Pinto - D.Sc. do IME

Rio de Janeiro  
2016

Aos nossos familiares, que deram todo o apoio e suporte no nosso processo de formação.

“Não é o conhecimento, mas o ato de aprender, não a posse mas o ato de chegar lá, que concede a maior satisfação. ”

CARL FRIEDRICH GAUSS

## SUMÁRIO

LISTA DE ILUSTRAÇÕES .....	7
LISTA DE SIGLAS .....	8
LISTA DE ABREVIATURAS .....	9
<b>1 INTRODUÇÃO .....</b>	<b>12</b>
1.1 Motivação .....	12
1.2 Objetivo .....	13
1.3 Justificativa .....	13
1.4 Metodologia .....	14
1.5 Estrutura do Texto .....	15
<b>2 ARQUITETURA DE MICROSERVIÇOS .....</b>	<b>16</b>
2.1 SOA - Arquitetura orientada a serviços .....	16
2.2 Microserviços .....	17
2.2.1 Principais Benefícios .....	18
2.2.2 Integração .....	21
2.2.3 Divisão de escopo .....	22
2.2.4 Complexidade .....	24
2.2.5 Tecnologias .....	24
<b>3 FERRAMENTAS .....</b>	<b>26</b>
3.1 Spring Framework .....	26
3.1.1 Spring Boot .....	27
3.1.2 Spring Cloud .....	29
3.2 Netflix OSS .....	29
<b>4 PROJETO .....</b>	<b>32</b>
4.1 Descrição textual .....	32
4.2 Projeto dos microserviços .....	33
<b>5 IMPLEMENTAÇÃO DO SGF .....</b>	<b>37</b>
5.1 Implementação do <i>auxiliar-service</i> .....	37
5.2 Implementação do <i>aluno-service</i> .....	37

5.3	Implementação do <i>web-service</i> .....	39
5.4	Implementação do <i>login-service</i> .....	41
5.5	Ciclo de uso de uma ata.....	42
<b>6</b>	<b>AMBIENTES DE FUNCIONAMENTO</b> .....	49
6.1	Servidor único .....	49
6.2	Máquinas Virtuais.....	50
6.3	Amazon Web Services .....	51
<b>7</b>	<b>CONCLUSÃO</b> .....	53
<b>8</b>	<b>REFERÊNCIAS BIBLIOGRÁFICAS</b> .....	55
<b>9</b>	<b>APÊNDICES</b> .....	57
9.1	APÊNDICE 1: Template dos arquivos JSON das entidades .....	58
9.2	APÊNDICE 2: Código fonte de sgf-core/AlunoRepository.java .....	63
9.3	APÊNDICE 3: Código fonte de sgf-core/AlunoController.java .....	65
9.4	APÊNDICE 4: Código fonte de sgf-web/AlunoService.java .....	67
9.5	APÊNDICE 5: Código fonte de sgf-web/AlunoController.java .....	69
9.6	APÊNDICE 6: Código fonte de sgf-web/AtaController.java .....	74
9.7	APÊNDICE 7: Passos para montar o Sistema em máquinas virtuais .....	86



## LISTA DE ILUSTRAÇÕES

FIG.2.1	Exemplo de sistema com heterogeneidade de armazenamento .....	19
FIG.2.2	Facilidade em escalar banco de dados com microserviços ao con- trário de um sistema monolítico .....	20
FIG.2.3	Comparação das divisões de equipes em sistemas monolíticos e em microserviços .....	23
FIG.2.4	Divisão no acesso ao banco de dados .....	23
FIG.2.5	Queda da produção com aumento da produtividade em sistemas monolíticos e em microserviços .....	25
FIG.3.1	Ribbon dentro de um microserviço .....	31
FIG.4.1	Projeto das entidades do sistema .....	34
FIG.4.2	Modelo da arquitetura dos microserviços .....	35
FIG.5.1	Sequência de telas onde coordenador cria um novo aluno .....	41
FIG.5.2	Tela de preenchimento da data .....	43
FIG.5.3	Tela de edição da ata de faltas .....	46
FIG.5.4	Sequência de telas de assinatura do professor .....	48
FIG.6.1	Tela de monitoramento do Eureka da máquina local .....	49
FIG.6.2	Tela de monitoramento do Eureka das máquinas virtuais .....	50
FIG.6.3	Sequência de telas para criar um professor com sistema no AWS .....	52
FIG.9.1	Configuração de rede .....	86
FIG.9.2	Seleção da imagem do sistema a ser instalado .....	87
FIG.9.3	Seleção da imagem do sistema a ser instalado .....	88
FIG.9.4	Telas de instalação .....	88
FIG.9.5	Programas a serem instalados .....	89
FIG.9.6	Informações de rede .....	90
FIG.9.7	Cópia de arquivos remoto .....	90
FIG.9.8	Inicialização do microserviço auxiliar .....	90

## LISTA DE SIGLAS

API	Application Programming Interface
AWS	Amazon Web Services
CRUD	Create, Read, Update and Delete
DNS	Domain Name Server
EJB	Enterprise JavaBeans
HTTP	Hypertext Transfer Protocol
IP	Internet Protocol
JAR	Java Archive
JEE	Java Platform, Enterprise Edition
JSON	JavaScript Object Notation
LTS	Long-Term Support
MVC	Model, Viewer, Controller
POJO	Plain Old Java Object
RAM	Random Access Memory
REST	Representational State Transfer
RPC	Remote Procedure Call
SOAP	Simple Object Access Protocol
SSH	Secure Shell
TCP	Transmission Control Protocol
TI	Tecnologia da Informação
UDP	User Datagram Protocol
XML	Extensible Markup Language
YAML	YAML Ain't Markup Language

## LISTA DE ABREVIATURAS

### ABREVIATURAS

DevOps - Development and Operations

MB - Megabyte

## RESUMO

O objetivo deste projeto de fim de curso é projetar, implementar e implantar um sistema de gerenciamento de faltas acadêmicas utilizando a arquitetura de microsserviços.

Para isso, inicialmente foi feito um estudo bibliográfico sobre a fundamentação teórica da arquitetura, como a filosofia de pensamento, as terminologias, os impactos na equipe e os conceitos que embasam e diferenciam essa arquitetura das demais, bem como as tecnologias atualmente empregadas pelas organizações.

Após o estudo, um sistema foi proposto, afim de colocar em prática os conceitos estudados. O sistema consiste em tornar digital do processo de lançamento de faltas do Instituto Militar de Engenharia. Os requisitos do sistema foram especificados e o projeto foi feito, segundo a arquitetura de microsserviços. Findo o projeto, os microsserviços foram implementados e o sistema foi concluído.

Por fim, o *software* foi implantado em três ambientes diferentes, sendo eles, uma máquina local, máquinas virtuais e um servidor AWS.

## ABSTRACT

This final project's goal is project, implement and deploy a system of academic absence management using the microservice architecture.

In order to do so, we initially made a bibliographic study on the architecture theory, as thought philosophy, terminology, impacts on the team and the concepts that support and differentiate this architecture from the others, as well as the technologies currently employed by organizations.

After the study phase, a system have been proposed, in order to practice the concepts that had been studied. The system consists on turn digital the Instituto Militar de Engenharia's absence release process. System requirement have been specified and the project was created, following microservice architecture. Ended the project, the microservices were implemented and the system was concluded.

Finally, the software have been deployed on three different environments, being a local machine, virtual machines and a server on AWS.

# 1 INTRODUÇÃO

O desenvolvimento de sistemas passou por diversas mudanças de metodologias e estruturas ao longo do tempo. Cada vez mais se vê a necessidade de encapsular partes do sistema para modularizá-lo, e técnicas para fazer tal encapsulamento vêm sendo aperfeiçoadas.

Quando se trabalha com sistemas muito grandes, a preocupação com manutenibilidade aumenta, pois torna-se muito difícil compreender todas as funcionalidades e principalmente a comunicação entre unidades. Outro fator que diminui a qualidade de grandes sistemas é o acoplamento entre unidades, algo que tenta-se evitar, mas é quase impossível em sistemas muito grandes.

A arquitetura de microsserviços surge para resolver esse problema, pois leva a modularização de unidades ao menor nível possível. A arquitetura tem por princípio básico dividir a aplicação em partes de funcionalidade completa, isto é, uma parte do sistema que tenha uma funcionalidade própria, chamadas de módulos. Um módulo deve ser o mais independente possível, para que a manutenção seja pontual e simples de ser feita (FOWLER, 2014b).

Diversas empresas têm aperfeiçoado essa arquitetura que ainda é incipiente e a cada dia mais empresas começam a migrar seus códigos para essa abordagem. Hoje em dia, a principal empresa no ramo é a Netflix, que desenvolveu diversas ferramentas e pacotes para facilitar a implementação de microsserviços e disponibilizou-os como código aberto. A empresa resolveu e otimizou diversos problemas referentes a implementação da arquitetura, possibilitando que outras organizações dessem prosseguimento e desenvolvessem ainda mais essas tecnologias (COCKCROFT, 2014).

## 1.1 MOTIVAÇÃO

Observa-se que diversas empresas, dentre as maiores na área de tecnologia da informação, têm migrado para a arquitetura de microsserviços. Isso se deve às melhorias que essa arquitetura traz ao processo de desenvolvimento de software.

Essa metodologia traz consigo vários benefícios ao desenvolvimento, como:

- Facilidade de modularizar o produto, separando em pequenas partes funcionais. Esse aspecto é especialmente importante quando se trabalha com grandes grupos de desenvolvimento;

- Cada serviço pode ser implantado de forma independente no servidor, o que diminui a possibilidade de falhas no sistema e facilita a detecção de problemas;
- Possibilidade de utilizar tecnologias distintas para cada serviço, envolvendo a linguagem de programação, frameworks e tecnologias de armazenamento.

Os problemas de se trabalhar com essa metodologia são os mesmos problemas de sistemas distribuídos, pois como cada serviço é implantado de forma independente, são necessárias chamadas remotas para fazer a comunicação entre eles, e tais chamadas são lentas e custosas. Outro ponto é a dificuldade de administrar vários serviços constantemente alterados de forma independente, isso pode se tornar muito complexo.

Mesmo diante desses problemas de gerência, a metodologia se mostra válida a medida que o projeto vai se tornando mais complexo. Para projetos muito simples, a metodologia tradicional (monolítica) pode ser mais facilmente implementada e administrada, porém, quanto maior e mais complexo o projeto se torna, a perda de qualidade do software monolítico passa a ser muito grande. Utilizando microsserviços, essa perda de qualidade é bem mais atenuada, fazendo com que se torne mais eficiente que a tradicional.

## 1.2 OBJETIVO

O objetivo do presente trabalho é desenvolver um software de lançamento de faltas utilizando a arquitetura de microsserviços. A aplicação consiste em um sistema que torna digital o processo de tiragem de faltas, projetado para ser didático, ressaltando as características da arquitetura em questão.

Com isso, a finalidade da implementação é o aprendizado dos conceitos da arquitetura e o uso das ferramentas e tecnologias atuais disponíveis e não o resultado final do software. É importante ressaltar este ponto, pois por vezes os métodos utilizados podem não ser os mais eficientes para resolver o problema em questão, entretanto o que está sendo avaliado é a utilização das técnicas de microsserviços.

## 1.3 JUSTIFICATIVA

A arquitetura de microsserviços ainda é incipiente, porém já está sendo adotada por diversas empresas dentre as maiores da área de tecnologia da informação. Dentre elas, destacam-se a Netflix, Uber, Amazon, Ebay e Sound Cloud. Com isso, nota-se o potencial que a arquitetura tem de dominar uma grande parte da produção de softwares num futuro próximo.

A compreensão dos conceitos e o aprendizado do ferramental atual para desenvolvimento de sistemas com microsserviços poderá ser potencialmente utilizada na produção de softwares a posteriori. Assim, o desenvolvimento de um sistema básico como o deste trabalho possibilita o contato com as tecnologias atuais e a ampla consolidação dos conceitos.

## 1.4 METODOLOGIA

Foi adotada uma metodologia composta por uma fase de estudo teórico, uma fase de projeto e a fase de implementação. Para realização deste trabalho, os seguintes passos foram planejados:

- Estudo dos conceitos de arquitetura de microsserviços: Primeiramente, é necessária uma revisão bibliográfica, para estudar os conceitos que darão base para todo o estudo. Para isso, foram utilizados principalmente os livros Cordeiro (2012), Newman (2015) e Silveira et al. (2012) e os artigos do Martin Fowler, retirados da sua página na internet, Fowler (2014b) e Fowler (2014a).
- Estudo das ferramentas e tecnologias utilizadas pelas principais empresas e familiarização com as principais escolhidas: Após ter os fundamentos da arquitetura bem definidos, é preciso saber o ferramental que está disponível para implementar utilizando essa abordagem. Então foi estudado sobre as tecnologias mais empregadas pelas grandes empresas do ramo. Esse estudo teve como principal base os artigos e matérias da InfoQ, como em Longa (2015).
- Proposta de um projeto que utilize o ferramental escolhido e se encaixe na abordagem de microsserviço: Para consolidar os aprendizados teóricos, propõe-se a elaboração de um projeto que utilize a arquitetura de microsserviços. Isto é, um software que será implementado utilizando os conceitos e ferramentas anteriormente estudados.
- Projeto e documentação do sistema: Como qualquer projeto de software, inicialmente será feito o planejamento do sistema, documentando-o para futura comprovação dos requisitos.
- Implementação do sistema localmente, isto é, numa única máquina: Um sistema na arquitetura de microsserviços é feito através da modularização, onde cada módulo é



implantado independentemente. Como primeiro passo de implementação, todos os módulos serão implantados numa mesma máquina de forma local.

- Divisão dos serviços em máquinas distintas: Posteriormente, quando os serviços estiverem com funcionamento pleno, serão separados em computadores distintos na mesma rede.
- Implantação de serviços na nuvem: Como passo final, os serviços serão implantados na nuvem utilizando um repositório remoto.

## 1.5 ESTRUTURA DO TEXTO

Essa dissertação encontra-se dividida da seguinte forma: o capítulo 2, é apresentado o embasamento teórico de arquiteturas orientadas a serviços e arquiteturas de microsserviços.

O capítulo 3 exhibe as principais ferramentas utilizadas na implementação de microsserviços e escolhidas para compor o sistema a ser desenvolvido no projeto, sendo elas o Spring Boot, Spring Cloud e o Netflix OSS.

O capítulo 4 é composto pela documentação do projeto do sistema, com apresentação da metodologia usada, a descrição textual e o diagrama esquemático do sistema.

O capítulo 5 é dedicado a uma explicação completa do sistema implementado, mostrando cada classe dos microsserviços e seus detalhes de implementação. Os microsserviços relacionados com Aluno são tomados como exemplo, e por fim, mostra-se a implementação dos microsserviços e o uso de uma Ata.

Por fim, o capítulo 6 mostra a implantação do sistema em três ambientes, sendo eles: uma máquina local, diferentes máquinas virtuais e um servidor remoto.

## 2 ARQUITETURA DE MICROSERVIÇOS

Em desenvolvimento de softwares, existem diversas arquiteturas possíveis de seguir, isto é, diversas metodologias para organizar e estruturar um software. Uma delas é baseada em serviços. Assim, um serviço é uma função de um sistema que é disponibilizado para outros sistemas, onde a união de todos os serviços gera o sistema inteiro.

Conforme os sistemas foram se tornando maiores e mais complexos, a necessidade de se pensar no sistema como uma divisão de serviços começou a crescer, pois já não era possível administrar grandes softwares com funcionalidades complexas.

Em 1996 o conceito de uma arquitetura de software orientada a serviços foi proposto pela primeira vez por Roy Schulte e Yefim Natis do Gartner Group, no artigo "*Service Oriented Architecture*". Na definição do Gartner Group: "SOA é uma abordagem arquitetural corporativa que permite a criação de serviços de negócio interoperáveis que podem facilmente ser reutilizados e compartilhados entre aplicações e empresas" (SILVEIRA et al., 2012).

### 2.1 SOA - ARQUITETURA ORIENTADA A SERVIÇOS

É possível encontrar diversas definições na literatura do que vem a ser SOA, entretanto é mais simples definir primeiramente o que SOA não é. SOA não é uma tecnologia, existem diversas tecnologias que viabilizam o trabalho com SOA, mas por si só, ele não é uma tecnologia. SOA também não é uma metodologia, há vários mecanismos para implantar SOA com sucesso, ele não define um método exclusivo a ser seguido.

SOA é uma filosofia arquitetural, isto é, uma linha de pensamento que permeia a implementação de necessidades de negócio, baseada no conceito de serviços atômicos, independentes e com baixo acoplamento, não se restringindo apenas à área de TI.

Essa linha de pensamento consiste em decompor um projeto em serviços independentes, de modo que a interoperabilidade dos serviços resulte no projeto total. Do ponto de vista do SOA, um serviço deve funcionar de forma independente do estado de outros serviços, exceto em casos de serviços compostos, e devem possuir uma interface bem definida. Esses serviços seguem algumas características, como: uso de padrões, incentivo a reutilização de componentes, abstração de tecnologias e infraestrutura, fraco acoplamento de serviços e composição de serviços. Dessas características, ressalta-se o fraco acoplamento,

isto é, tentar minimizar o impacto de alterações e falhas em um serviço no sistema como um todo.

Com isso, são esperados diversos benefícios no sistema. Utilizando serviços independentes, a manutenção de grandes sistemas fica mais fácil, pois diminui-se a complexidade da parte a ser mantida. Com a facilitação da manutenção, diminui-se grandes custos, visto que essa etapa é a maior geradora de gastos sem produção de funcionalidade.

Como os serviços tem tarefas específicas, possibilita-se o reuso dos mesmos, seja no mesmo sistema ou para projetos futuros. Estruturando o projeto em serviços independentes, separa-se nitidamente o back-end e a infraestrutura da interface com o usuário, assim, o serviço de back-end e infraestrutura podem ser facilmente substituídos com pouco impacto sobre a relação com o usuário, bem como novas interfaces podem ser adicionadas utilizando o mesmo servidor (SILVEIRA et al., 2012).

A partir dos anos 2000, diversas empresas começaram a desenvolver seus softwares utilizando uma filosofia baseada em serviços, mas ainda não era estruturado. No início, esse conceito era apenas informal, sem nenhum estudo profundo sobre técnicas e tecnologias para otimizar essa linha de desenvolvimento. A metodologia adotada consistia apenas em dividir o sistema em partes funcionais, chamadas de serviços, encapsulando-os em partes lógicas. Cada uma dessas partes devia ser logicamente independente, acessando o banco de dados e executando suas funções de forma autônoma. A Amazon foi a primeira empresa a adotar essa abordagem, porém sem nenhum conceito formal. Posteriormente, outras empresas adotaram métodos similares, que serviram de grande inspiração para o surgimento de uma arquitetura baseada em serviços (CORDEIRO, 2012).

## 2.2 MICROSERVIÇOS

Baseado na ideia de SOA, alguns desenvolvedores começaram a tentar transformar essa linha de pensamento em algo mais concreto, resultando numa metodologia. A ideia básica era formalizar conceitos em aberto, como a abrangência de um serviço, a comunicação entre serviços e principalmente o tamanho dele. Pensando nesses aspectos, a conclusão foi que o serviço deve ser o menor possível, desde que mantenha-se com uma funcionalidade completa, daí surge a ideia de um microserviço.

Atualmente, os principais nomes na área de microserviços são Martin Fowler e James Lewis. Para eles, a definição de Arquitetura de Microserviços é: “Abordagem para desenvolvimento de uma única aplicação como um conjunto de pequenos serviços, cada um rodando em um processo próprio e se comunicando por mecanismos leves, frequentemente

sendo uma API usando HTTP. Esses serviços são construídos com base nas capacidades de negócio e devem ser independentemente implantáveis por meios totalmente automáticos" (FOWLER, 2014b).

Com isso, podemos ver alguns pontos claramente comuns com o SOA, porém com um detalhamento maior. Por exemplo, a divisão do sistema em serviços é contemplada no SOA, mas em microserviços eles devem ter o menor tamanho possível mantendo uma funcionalidade completa. Um tópico extra é a necessidade dos serviços estarem rodando em processos próprios, tendo a capacidade de serem implantados e atualizados de modo independente e automático. A autonomia de um serviço é essencial para que se possa usufruir das vantagens prometidas pela arquitetura.

Quando se divide o projeto em microserviços e os implementa, deve-se perguntar: eu poderia fazer uma alteração neste serviço e reimplantá-lo sem precisar alterar nada mais no sistema? Se a resposta for não, será muito difícil obter as vantagens descritas (NEWMAN, 2015).

### 2.2.1 PRINCIPAIS BENEFÍCIOS

- Heterogeneidade tecnológica:

Como o sistema é composto de vários serviços colaborativos, podem-se escolher diferentes tecnologias para implementar cada um deles. Isso permite que se escolha a ferramenta ideal para cada necessidade específica da funcionalidade, ao invés da escolha geral que era feita em sistemas monolíticos, nivelando a performance por baixo.

Se uma funcionalidade necessita de uma melhor performance, pode-se utilizar uma linguagem que otimize o desempenho. Da mesma forma que o tipo de armazenamento pode ser ajustado às necessidades do serviço, como mostrado na figura 2.1. Por exemplo, um serviço de redes sociais poderia armazenar seus usuários num banco orientado a grafos, afim de simular a interação entre pessoas. Já os posts desses usuários podem ser armazenados em bancos de dados orientados a documento, gerando o diagrama da figura 2.1.

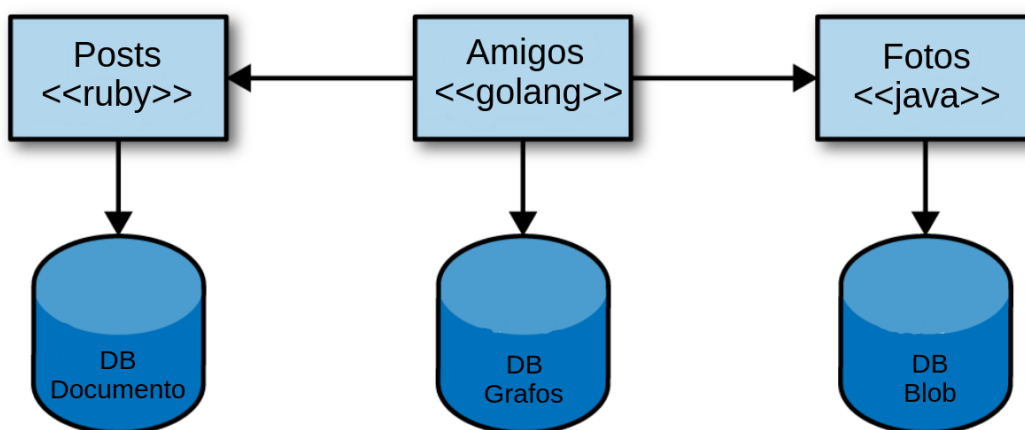


FIG. 2.1: Exemplo de sistema com heterogeneidade de armazenamento (NEWMAN, 2015)

Essa arquitetura também facilita a inserção de novas tecnologias, o que é uma grande barreira em sistemas monolíticos, pois a substituição de uma linguagem de programação, banco de dados ou um framework teria impacto sobre grande parte do sistema, exigindo muito custo e tempo. Com múltiplos serviços, há diversas possibilidades de tentar novas tecnologias e implantá-las de forma gradual, com pequeno impacto e custo, limitando potenciais perdas e falhas.

Muitas empresas costumam basear seus códigos em um pequeno conjunto de tecnologias e plataformas, pois já tem um alto grau de entendimento da tecnologia e produzem suas próprias ferramentas para essas plataformas. Sabe-se que a introdução de uma nova tecnologia não é feita de forma otimizada, mas se é possível reescrever todo um serviço em duas semanas, diminui-se o problema de utilizar a tecnologia de forma principiante.

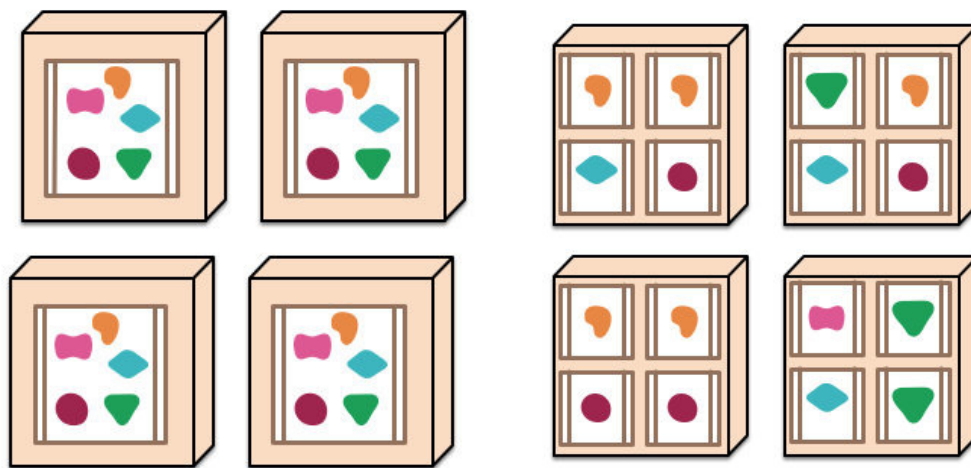
- Resiliência:

Sistemas monolíticos são em geral pouco tolerantes a falhas, pois um erro impacta grande parte da aplicação, o que torna a manutenção complexa. É possível, por exemplo, rodar um sistema monolítico em várias máquinas diferentes para reduzir a chance de falha. Entretanto, num sistema com diversos serviços independentes, pode-se isolar o componente restringindo o problema, desde que essa falha não atinja outros serviços em cascata. Com microsserviços, pode-se usar a mesma abordagem de sistemas distribuídos e construir sistemas que controlem as falhas.

- Escalabilidade:

Em um sistema monolítico, todas as funcionalidades devem crescer juntas. Se existe uma pequena funcionalidade acoplada a uma grande, o crescimento de uma implicaria no crescimento da outra, sem necessidade, dependendo tempo e trabalho para tal. Na figura 2.2a pode-se ver que a replicação de um sistema monolítico é feita sobre todo o sistema, assim, todas as unidades tem que ser escaladas juntas.

Com microsserviços, pode-se escalar cada módulo conforme for a sua necessidade. Assim, o nível de hardware e infraestrutura precisará acompanhar apenas a demanda real de cada serviço, podendo implantar um serviço menor e menos requisitado numa máquina mais simples e expandir apenas o hardware da parte necessária. Como cada unidade está dividida num ambiente independente, cada serviço pode escalar sem depender dos outros, conforme a divisão ilustrada na figura 2.2b, cada serviço pode ter mais ou menos replicações.



(a) Replicação de bancos de dados em sistemas monolíticos      (b) Replicação de bancos de dados em sistemas com microsserviços

FIG. 2.2: Facilidade em escalar banco de dados com microsserviços ao contrário de um sistema monolítico

(FOWLER, 2014a)

- Facilidade de implantação:

Uma pequena mudança implica em ter que implantar novamente toda aplicação, oferecendo um alto risco nessa tarefa em sistema monolíticos. Isso significa que as mudanças são lentas e carecem de muita precaução, pois qualquer erro pode degradar toda a aplicação.

Utilizando microsserviços, a implantação é pontual, tornando-a leve e diminuindo os riscos atrelados a mudança. Unindo esses conceitos com a metodologia DevOps,

consegue-se uma facilidade ainda maior nessa tarefa.

- Divisão de equipes:

As equipes podem ser menores em microsserviços, pois são necessários menos membros para cuidar de todas as partes de um único serviço. Assim, com um time menor trabalhando sobre um escopo menor, as soluções tendem a ser mais otimizadas e mais produtivas. A figura 2.3 também mostra as diferenças na divisão de equipes para sistemas monolíticos (divisão vertical) e com microsserviços (divisão horizontal).

- Reusabilidade:

Não apenas pode-se reaproveitar o código, como pode-se reutilizar um serviço implantado para atender a outros módulos. Como as funcionalidades contempladas em um serviço são muito pequenas, a chance de reuso é muito grande.

### 2.2.2 INTEGRAÇÃO

Existem diversas formas de fazer a comunicação entre os microsserviços, como SOAP, XML-RPC e REST.

Essa integração deve ser feita de forma que uma mudança num serviço não requira uma mudança nos serviços que fazem uso dele. Por exemplo, se um serviço adiciona um campo a um certo dado, usuários deste serviço não deveriam ser impactados.

Também não é desejável que os usuários saibam, através dos dados trocados, como é a implementação interna do serviço. Os dados devem seguir um padrão que não denuncie detalhes de representação que estão sendo usados no servidor. (NEWMAN, 2015)

Uma característica recomendada é que os serviços sejam *stateless* (sem estado). O principal benefício é a capacidade de resposta de eventos retirando ou adicionados serviços, sem a necessidade de mudar ou reconfigurar a aplicação. Por exemplo, caso a carga de um sistema aumente consideravelmente, mais serviços *stateless* poderão ser levantados, ou se algum serviço cair, ele pode ser simplesmente trocado por outro. Escalabilidade e resiliência podem ser alcançados mais facilmente utilizando serviços sem estado.

Com isso, o padrão mais aconselhado em microsserviços é o REST (*Representational State Transfer*), em português, Transferência de Estado Representacional. Esse padrão é largamente usado para gerar APIs, ou seja, serviços que apenas provêm e recebem dados. Chamados de RESTfull APIs, os serviços deste tipo mantêm modelos de dados, comunicando-se para enviar os dados deste modelo e recebê-los para mantê-los no banco.

Esse padrão é muito aplicável para serviços do tipo CRUD, onde um modelo é apenas criado e mantido. O envio dos dados é geralmente feito por JSON ou XML, dependendo da integração e tecnologia a ser utilizada. Assim, a comunicação fica muito leve, o que é de extrema importância em chamadas remotas.

Este é o principal problema na comunicação entre serviços independentes rodando em máquinas distintas, as chamadas remotas. Por se tratar de chamadas entre máquinas diferentes, por vezes distantes fisicamente, as chamadas podem ser lentas e custosas. Entretanto esse problema já é muito atenuado utilizando-se as tecnologias e padrões atuais. (NEWMAN, 2015)

### 2.2.3 DIVISÃO DE ESCOPO

A utilização de microsserviços possibilita a divisão do escopo de diversas tarefas, como acesso ao banco de dados, a equipe e a governança.

Em sistemas monolíticos, as equipes apresentam uma divisão horizontal, onde cada time é responsável por uma grande área do projeto. Por exemplo, como na figura 2.3a, muitas empresas dividem as equipes em especialistas em interface com usuário (front-end), especialistas em middlewares e especialistas em bancos de dados. O problema é que cada equipe precisa lidar com todos os serviços que o projeto tem. O time de banco de dados fica responsável por todo o banco, tendo que entender e gerenciar todos os dados da empresa. Consequentemente, um especialista de middleware não tem nenhum contato com o banco, tornando a comunicação entre as áreas muito difícil.

A abordagem para microsserviços é diferente, formam-se pequenas equipes responsáveis por um serviço, ou seja, são necessários especialistas de todas as áreas para compor a equipe, ilustrado na figura 2.3b. Com isso, a comunicação entre os membros da equipe é otimizada, fazendo com que melhores ideias sejam postas em prática e cada membro tenha conhecimento de todo o processo.

Da mesma forma que a equipe, os bancos de dados são organizados de modo diferente em microsserviços. Em aplicações tradicionais, os bancos de dados são acessados de por completo por cada usuário, ou seja, um grande banco de dados como na figura 2.4a contendo tudo que a aplicação completa precisa. Quando é necessário se fazer a replicação de dados, todo o banco é copiado para manter o sincronismo.

Quando se divide o sistema em pequenos serviços, cada um deles precisa acessar pequenas quantidades de dados, assim, é possível também dividir os bancos em pequenas partes de modo a atender os serviços de forma modularizada, formando um esquema como



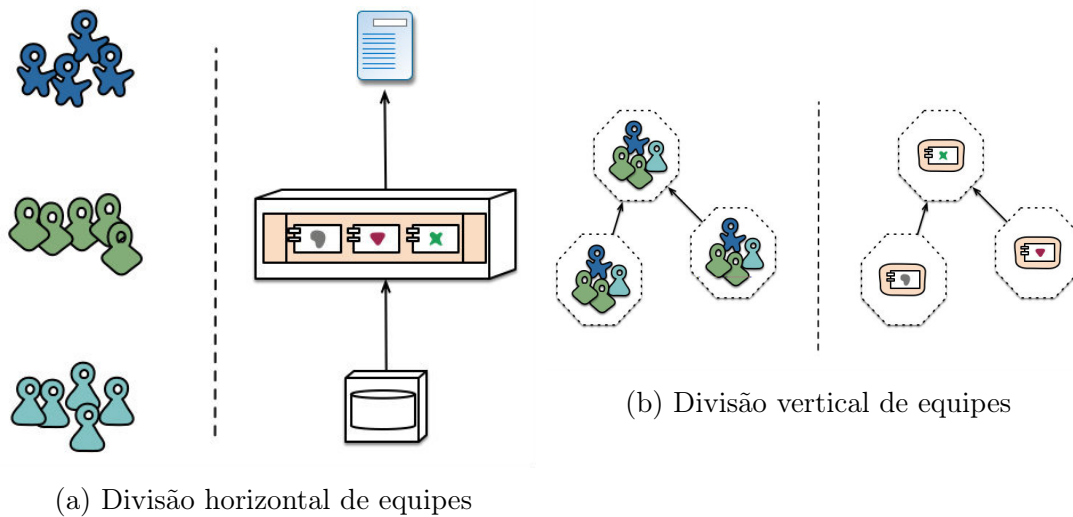


FIG. 2.3: Comparação das divisões de equipes em sistemas monolíticos e em microserviços

(FOWLER, 2014a)

o da figura 2.4b. Sendo assim, um banco pode ser acessado por diversos serviços e certos serviços podem acessar mais de um banco, de modo que os bancos possam ser alimentados de forma independente encapsulando dados logicamente separados.

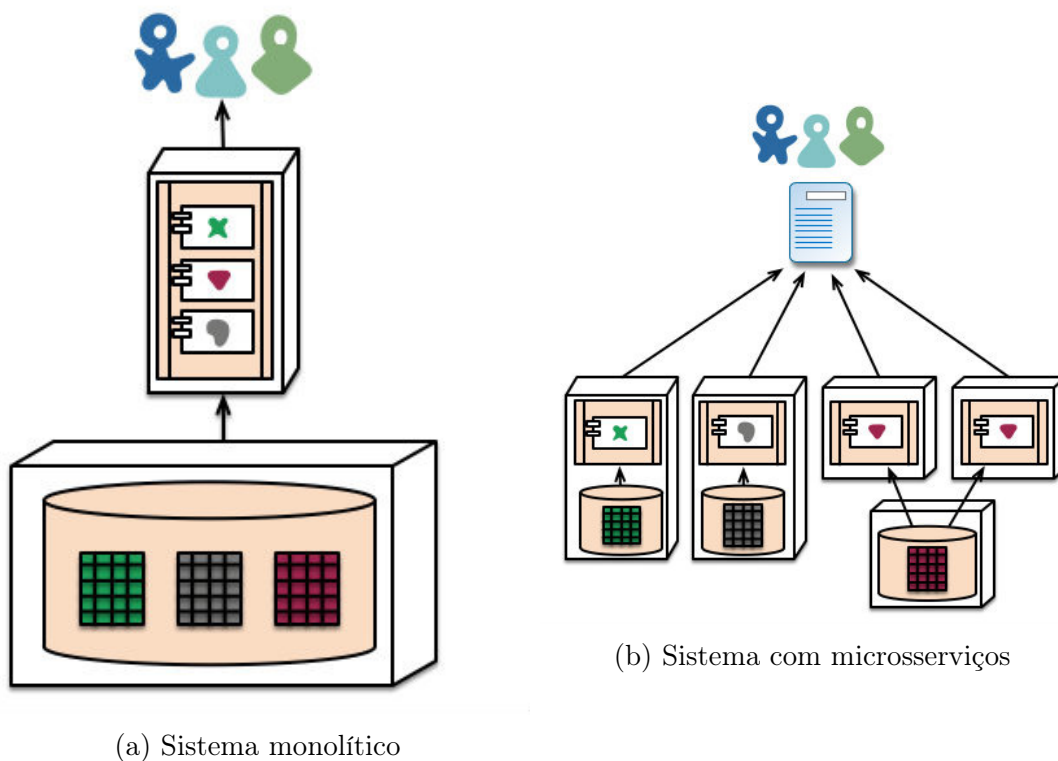


FIG. 2.4: Divisão no acesso ao banco de dados

(FOWLER, 2014a)

## 2.2.4 COMPLEXIDADE

A eficiência do uso da arquitetura de microsserviços não é sempre maior que a arquitetura tradicional. Na verdade, não é aconselhado começar uma aplicação com a arquitetura de microsserviços e sim iniciar um sistema monolítico, migrando-o para microsserviços quando for necessário. Martin Fowler diz que quase todos os casos de empresas que tentaram iniciar já implementando sistemas completos com microsserviços não deram certo. Isso se deve a grande complexidade de implementação no início do projeto e principalmente a falta de informação sobre a utilização dos serviços. Isto é, deve-se perceber as necessidades específicas de cada processo e quais deles valem a pena para então implementar os microsserviços da melhor forma.

Inicialmente, um projeto monolítico tende a ser mais barato e menos complexo, pois as interações entre unidades não são tão grandes. A medida que o projeto cresce e torna-se mais complexo a produtividade cai em ambas as arquiteturas, porém esta queda é mais atenuada quando se usa microsserviços. O gráfico 2.5 mostra essa relação.

## 2.2.5 TECNOLOGIAS

Para viabilizar o uso de microsserviços, são necessárias algumas tecnologias para resolver problemas relativos a implantação de tantos serviços independentes e principalmente a comunicação entre eles. Além disso são necessárias metodologias de recursos humanos para se adequar a essa nova abordagem.

Muitos dos problemas relativos a implementação foram resolvidos e otimizados por empresas como a Netflix, que contribuiu para a formação de ferramentas que hoje são fundamentais para o uso da arquitetura. A abordagem DevOps também é fundamental para uma equipe que se disponha a trabalhar com microsserviços, pois exige-se que as alterações sejam implantadas de forma automatizada, o que rompe com a ideia monolítica de ter uma equipe só para manter o código em produção.

Hoje, as empresas utilizam as mais diversas tecnologias para cada serviço juntamente com implementação da arquitetura de microsserviços. Destacam-se as tecnologias que serão utilizadas nessa dissertação:

- Spring Boot;
- Spring Cloud; e
- Netflix OSS.

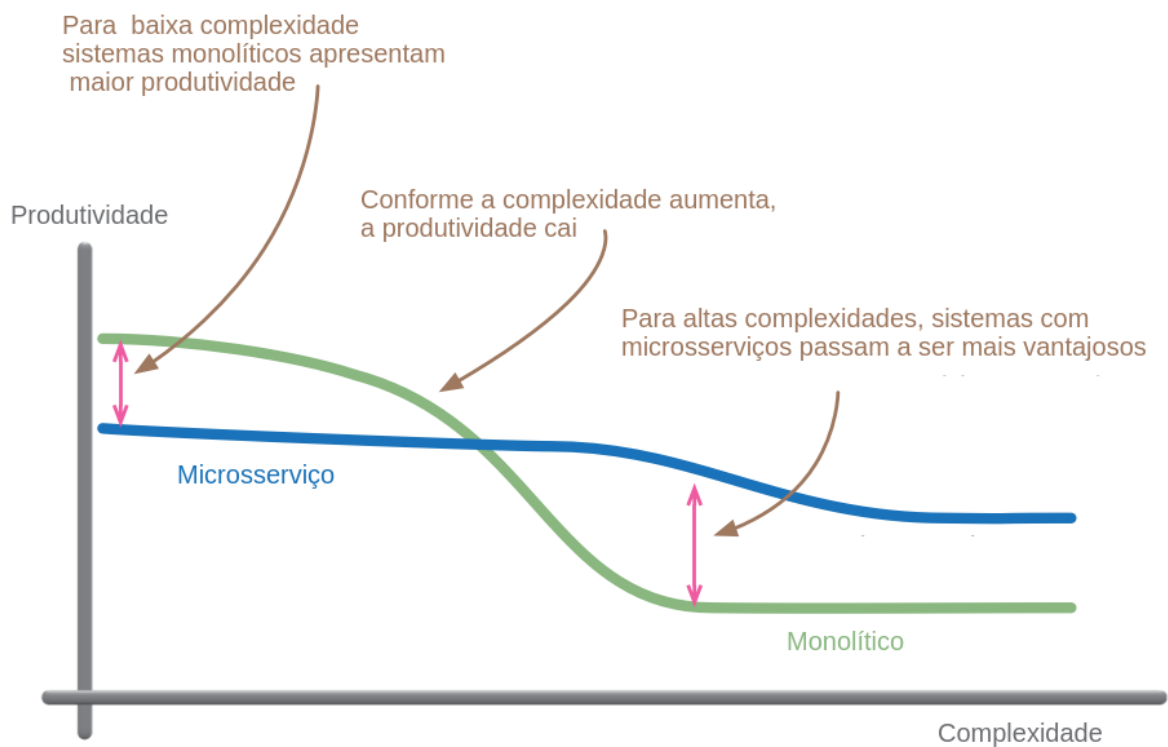


FIG. 2.5: Queda da produção com aumento da produtividade em sistemas monolíticos e em microsserviços

(FOWLER, 2014b)

O Spring é feito sobre Java, uma linguagem que permite a implementação de microsserviços e está sendo muito usada devido a grande contribuição oferecida pela equipe do Spring.

### 3 FERRAMENTAS

Para o desenvolvimento de microsserviços, é importante a utilização de frameworks e softwares que dão suporte às características necessárias para alcançar os princípios almejados por essa arquitetura. Esses frameworks são necessários para que o desenvolvedor não tenha que se preocupar com problemas prévios das características de microsserviços que já foram amplamente solucionados.

A Netflix, com seu Centro de Softwares com Códigos Abertos, resolveu diversos desses problemas e disponibilizou-os no Netflix OSS. Com isso, o Spring Framework agregou os códigos disponibilizados criando uma rica plataforma para desenvolvimento em microsserviços. Como o Spring é um Framework de Java, a linguagem tornou-se uma referência para desenvolvedores de microsserviços.

#### 3.1 SPRING FRAMEWORK

O Spring Framework já possui mais de uma década de existência e já se encontra como framework padrão em diversas aplicações Java. Ele começou como uma alternativa leve ao padrão JEE. Ao invés do desenvolvimento dos pesados Enterprise EJBs, o Spring ofereceu uma abordagem similar, utilizando injeção de dependências e programação orientada a aspectos como forma de alcançar as capacidades similares ao EJB com POJOs.

No que se refere a simplicidade de codificação, o Spring possuía uma pesada configuração. Inicialmente era baseado em arquivos de configuração em XML. A versão 2.5 introduziu escaneamento de componentes baseado em anotações, o que eliminou boa parte das configurações explícitas em XML para os componentes da aplicação. O Spring 3.0 apresentou configurações baseadas em Java como uma alternativa segura ao XML.

Todas essas configurações representam um atrito no desenvolvimento. Todo o tempo gasto em definição de configuração é um tempo em que não é desenvolvida a lógica da aplicação. O desgaste para a configuração do Spring é uma desatração para a resolução de problemas de negócio. Como todo framework, o Spring faz muito para o desenvolvedor, entretanto ele também exige bastante em retorno.

Além de tudo, a gestão de dependências é uma tarefa custosa. Decidir quais as bibliotecas devem fazer parte pode ser difícil, mas é ainda mais desafiador saber quais as versões que se comportam bem com as outras. Mesmo sendo importante, a gestão de

dependência é outro atrito ao desenvolvimento. Quando se está adicionando dependências, é gasto um tempo em que não se escreve código. Qualquer incompatibilidade que ocorra enquanto se seleciona versões erradas de dependências se apresentam como grande causa de improdutividade.

### 3.1.1 SPRING BOOT

O Spring Boot vem como forma de se amenizar todos os problemas da decisão de se utilizar do Spring para o desenvolvimento de serviços web.

Para o desenvolvimento de uma aplicação web simples utilizando o Spring clássico é necessário: uma estrutura de projeto definida com o Maven, um arquivo web.xml que declara o DispatcherServlet do Spring, a configuração do Spring para habilitar o Spring MVC, uma classe controller que irá responder às requisições HTTP e um servidor de aplicação web, como o Tomcat, para a realização do deploy. Com o uso do Spring Boot, tudo isso se resume a uma estrutura de projeto com o Maven, uma classe Java usada para dar início à aplicação e uma classe controller que responderá às requisições HTTP. Todo o resto necessário é resolvido pelo Spring Boot.

O Spring Boot realiza duas operações bases para alavancar o desenvolvimento:

- Configuração automática - A plataforma consegue prover as configurações para as funcionalidades comuns a várias aplicações Spring.

Em qualquer aplicação Spring comum, a configuração é baseada em XML ou em classes Java que habilitam suporte para certas funcionalidades. Por exemplo, caso seja necessário o uso de um banco de dados embutido como o H2 é necessário uma definição de um bean como o abaixo, que retorna uma referência ao banco de dados embutido.

```
@Bean public DataSource dataSource() {  
    return new EmbeddedDatabaseBuilder()  
        .setType(EmbeddedDatabaseType.H2)  
        .addScripts('schema.sql', 'data.sql')  
        .build();  
}
```

Mesmo não sendo complexo a configuração desse bean, ele representa somente uma fração de uma configuração típica de uma aplicação Spring. Além do mais, existirão diversas aplicações que possuirão os mesmos métodos, então qual a necessidade

de se reescrever? O Spring Boot consegue configurar automaticamente esse cenário, somente detectando que o projeto possui uma biblioteca do banco de dados embutido H2 no class-path, que poderá ser utilizado através de injeção de dependências nos beans que o utilizarão.

- Dependências iniciais - Definindo os tipos de funcionalidades desejadas, o Spring Boot irá garantir que as bibliotecas necessárias sejam adicionadas ao projeto. Dependências iniciais são um tipo especial de dependência do Maven (e o Gradle) que tira vantagem da resolução transitiva de dependências. Por exemplo, para a criação de uma API REST com Spring MVC que utiliza representação JSON e a aplicação utiliza um servidor Tomcat embutido é necessário declarar as seguintes dependências:

- org.springframework:spring-core
- org.springframework:spring-web
- org.springframework:spring-webmvc
- com.fasterxml.jackson.core:jackson-databind
- org.hibernate:hibernate-validator
- org.apache.tomcat.embed:tomcat-embed-core

Utilizando a vantagem do Spring Boot, basta adicionar o *web stater* como dependência que teria o mesmo efeito. Entretanto, muito mais do que a redução na contagem de dependências o Spring Boot trás o conceito de declaração de funcionalidades ao invés do de dependências. Para a aplicação web, adiciona-se o *web starter*. Se é necessário segurança, colocasse o *security starter*. Para persistência JP, *jps starter*. Portanto ao invés de pensar em que bibliotecas suportam certa funcionalidade, basta pedir a tal funcionalidade.

Devido às funcionalidades do Spring Boot muitos conceitos errados são inferidos. Primeiramente, Spring Boot não é um servidor de aplicação. Esse equívoco se dá pelo fato da possibilidade da criação de aplicações web auto executadas por meio de arquivos JAR, que são capazes de rodar via linha de comando sem a necessidade de servidores de aplicações convencionais. Spring Boot consegue esse feito embutindo um container *servlet* dentro da aplicação, mas esse servidor embutido provê uma funcionalidade, não sendo o próprio Spring Boot.

Similarmente, o Spring Boot não implementa nenhuma especificação Java enterprise, como o JPA. Ele suporta diversas especificações, mas através de configuração automática de beans no Spring que suportam essas funcionalidades. Por exemplo, o Spring Boot não implementa JPA, mas suporta o JPA pela autoconfiguração do bean apropriado para a implementação do JPA, como o Hibernate.

Finalmente, o Spring Boot não prega nenhuma forma de geração de código para cumprir suas funcionalidades. Ao invés, ele se utiliza das características de configuração condicional do Spring 4, junto com a resolução de dependência transitiva, oferecida pelo Maven e Gradle, para automaticamente configurar beans no contexto das aplicações Spring. (WALLS, 2015)

### 3.1.2 SPRING CLOUD

O Spring Boot é voltado para o desenvolvimento de aplicações web. Entretanto diversos sistemas crescem o bastante para ter que mudar sua arquitetura tradicional para uma voltada em escalabilidade e disponibilidade. Portanto, uma arquitetura de sistemas distribuídos é desejada e o tema de microserviços se torna sinônimo para o desenvolvimento de sistemas distribuídos.

Os engenheiros do Spring criaram o projeto chamado Spring Cloud visando essa necessidade. Ele provê ferramentas para desenvolvedores construírem rapidamente alguns padrões comuns em sistemas distribuídos, como gerenciamento de configurações, descobrimento de serviços, quebra de circuitos, balanceamento de carga, entre outros.

O Spring Cloud se baseia no Spring Boot, fornecendo diversas bibliotecas que melhoram o comportamento das aplicações adicionadas no *classpath*. Ele fornece uma camada para outras pilhas de implementações, como o Netflix OSS, que são consumidas por meio de configurações baseadas em anotações, familiares para desenvolvedores acostumados com o Spring (SPR) (STINE, 2015).

## 3.2 NETFLIX OSS

Quando uma aplicação faz sucesso, é comum a necessidade de mais desenvolvedores, e com mais clientes existem mais requisições para serem atendidas. Também é necessário aumentar a disponibilidade e distribuir a aplicação globalmente para garantir baixa latência, como visto por (FARIAS, 2014).

Para resolver esses problemas, o Netflix faz uso da plataforma AWS como plataforma de *cloud computing*. A empresa mudou de uma arquitetura monolítica para uma baseada

em centenas de microsserviços.

O Netflix é uma companhia que é referencia quando se trata de arquitetura em microsserviços. Um dos princípios fundamentais sobre o qual o Netflix se baseia é que qualquer componente deve ser capaz de sofrer uma falha, mas o sistema ainda deve continuar funcional. Quando se tenta construir um sistema dessa forma, existem diversos desafios como identificados por (STINE, 2015):

- Como decompor o sistema em diversos microsserviços?
- Processos em um ambiente em *cloud computing* são criados e terminados, e ainda mudam sua localização. Como determinar a localização dos processos que os serviços utilizam?
- Como podemos manter a visibilidade para o comportamento do sistema que surgiu a partir da topologia evolução dos serviços autônomos, para que se possa fazer ajustes?

A equipe do Netflix desenvolveu uma série de componentes, que resolvem boa parte desses problemas. Esses componentes foram testados em produção e se tornaram *open source*. Grande parte deles estão presentes no Spring Cloud para serem utilizados nos mais diversos sistemas. Os componentes que serão utilizados no projeto serão o Eureka e o Ribbon.

O Projeto Eureka vem da necessidade de se descobrir quais são os serviços existentes e onde eles estão localizados. Ele atua como um registro de serviços, ele conhece todos os serviços e sabe o status de cada um (iniciando, em execução, fora do ar, etc.) e em que máquina, zona, região e IP que estão sendo executados. O servidor Eureka pode ser implementado utilizando uma aplicação simples com o Spring Cloud sendo necessário apenas que a classe de configuração contenha a *annotation* `@EnableEurekaServer`. As configurações de funcionamento são feitas ou por propriedades em Java ou por arquivos YAML. A comunicação com esse serviço ocorre via REST, portanto uma aplicação não precisa ser escrita em Java e não é obrigada a utilizar o Spring Cloud para se registrar no Eureka, ela somente deve implementar a forma de comunicação que o Eureka utiliza.

O Ribbon realiza o balanceamento de cargas dos serviços. Ele é responsável por realizar a comunicação entre os diversos microsserviços e fazer o balanceamento de carga, suportando diversos protocolos (HTTP, TCP, UDP) inclusive para chamadas assíncronas e arquiteturas reativas. Ele funciona como uma parte interna da aplicação como explica a figura 3.1, sendo que quando é feito uma chamada a um serviço externo o Ribbon



é acionado. Primeiramente ele verifica as localizações do serviço desejado no registro do Eureka, com essa informação ele realiza o balanceamento de carga, sendo que ele já possui diferentes implementações e ainda suporta que seja utilizado uma forma própria se desejado, mas por padrão ele utiliza o algoritmo *round robin*, como visto na documentação oficial.

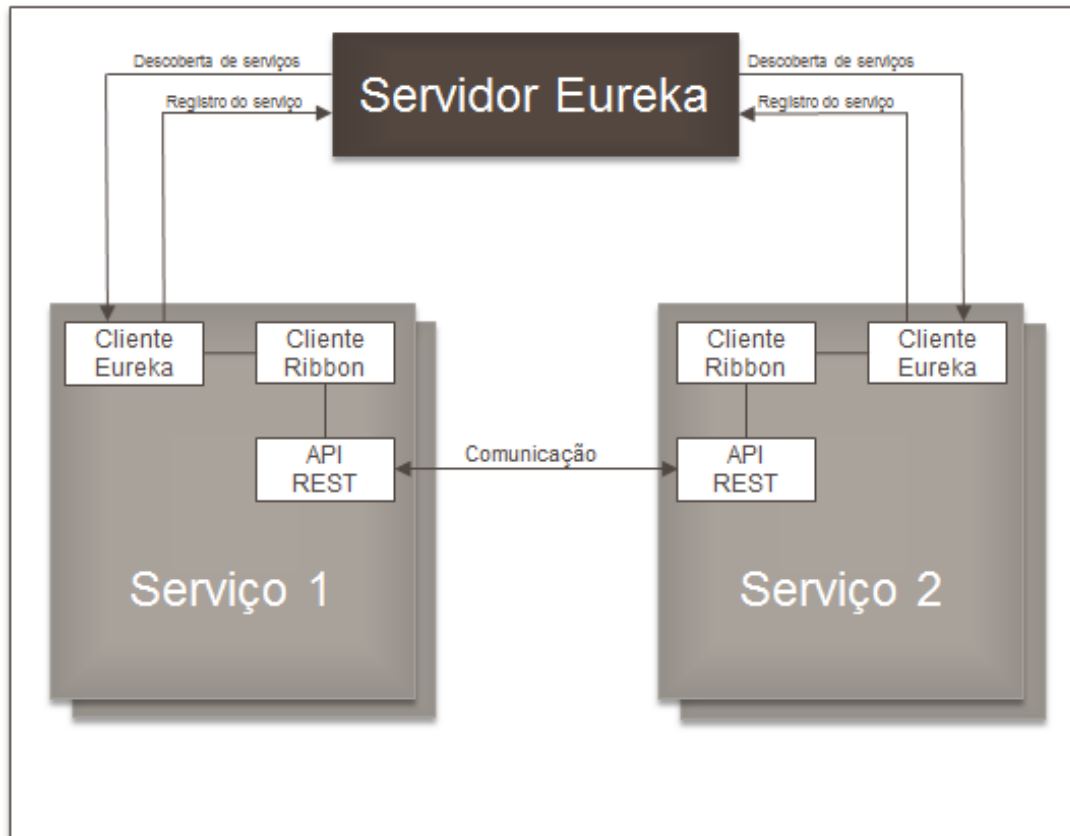


FIG. 3.1: Ribbon dentro de um microserviço

## 4 PROJETO

Para ratificar os conceitos estudados e ter contato com as principais tecnologias de microserviços, foi estruturado um projeto de sistema que se encaixe na arquitetura de microserviços. Entretanto, o projeto visa colocar em prática a arquitetura de microserviços, e por isso o projeto terá um caráter didático, cujo fim é utilizar os conceitos de microserviços mesmo que em uma situação real fosse mais conveniente o uso de outra abordagem.

### 4.1 DESCRIÇÃO TEXTUAL

#### Sistema de Gerenciamento de Faltas (SGF)

O sistema visa digitalizar o processo de tiragem de faltas realizado pelos alunos do Instituto Militar de Engenharia. O processo manual segue os seguintes passos:

- a) Um aluno (chefe de turma) anota na ata de faltas a matéria lecionada e os alunos faltosos para cada tempo de aula;
- b) O professor assina cada tempo de aula;
- c) O aluno (chefe de turma) assina a ata de faltas;
- d) O aluno (chefe de turma) entrega a ata de faltas para a coordenação.

É necessário que o sistema também implemente as funcionalidades auxiliares que viabilizem o processo de tiragem de faltas, como cadastro de todos os integrantes do processo e autenticação.

Assim, as funcionalidades necessárias são:

- a) Cadastro de alunos. O coordenador pode fazer CRUD de um aluno, especificando o seu nome, número, senha e email. O aluno autenticado pode alterar sua senha;
- b) Cadastro de turmas. O coordenador pode fazer CRUD de uma turma, especificando o seu ano e engenharia, pode também associar uma lista de alunos e disciplinas a uma turma;

- c) Cadastro de professores. O coordenador pode fazer CRUD de um professor, especificando o seu nome, email, senha, se é um coordenador e as engenharias em que leciona.
- d) Cadastro de disciplinas. O coordenador pode fazer CRUD de uma disciplina, especificando o seu nome e professor.
- e) Lançamento de faltas. Um aluno autenticado pode lançar as faltas de um determinado dia para sua turma. Isto é, o aluno cria uma ata de faltas inserindo a data e para cada tempo de aula ele marca a disciplina lecionada e os alunos da turma que estão na falta, com a respectiva justificativa.
- f) Assinatura. O professor assina o seu próprio tempo de aula. Quando todos os tempos estão assinados, o aluno que criou a ata pode então assiná-la. Com todas as assinaturas prontas o coordenador pode assinar a ata e ela fica completa.
- g) Registro da ata de faltas. Quando o coordenador assina a ata de faltas, ela é arquivada. Uma vez arquivada, a ata só pode ser lida e excluída pelo coordenador.

Um aluno pode pertencer a mais de uma turma. Para alterar a senha de qualquer usuário, é requisitada a senha atual. Um professor só pode assinar se os campos do tempo de aula estiverem preenchidos e após a assinatura, os campos do tempo de aula não podem mais ser alterados. O aluno (chefe de turma) só pode assinar a ata de faltas se todos os tempos de aula estiverem assinados, então, o aluno pode submeter a ata para o coordenador. O coordenador não pode alterar nada na ata de faltas assinada pelo aluno.

## 4.2 PROJETO DOS MICROSERVIÇOS

Pela descrição textual, há cinco entidades no sistema:

- Aluno
- Professor
- Disciplina
- Turma
- Ata

A figura 4.1 mostra cada entidade e seus atributos, extraídos da especificação do sistema.

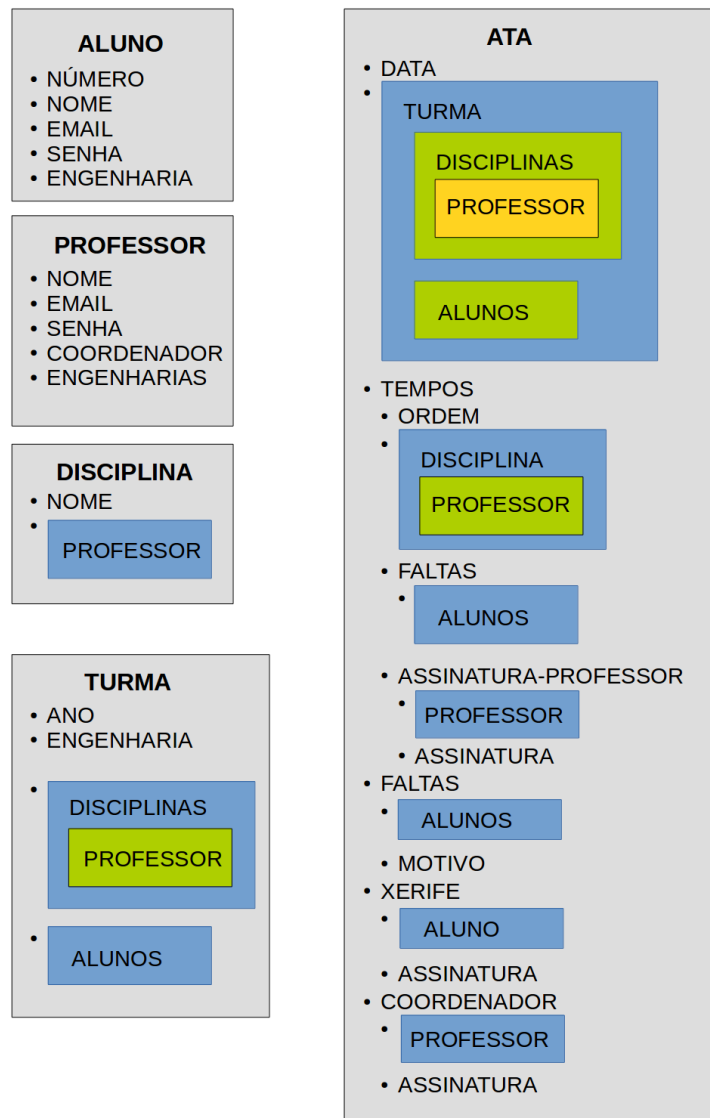


FIG. 4.1: Projeto das entidades do sistema

Isso leva a necessidade de um microserviço para manter cada uma dessas entidades, já que os bancos de dados devem ser independentes. Esses microserviços serão chamados de *Core* e realizarão apenas CRUD das respectivas entidades.

Além dos serviços do Core, para atender aos requisitos precisa-se de um serviço de login e um serviço de interface com o usuário. Portanto, serão usados cinco microserviços para manter as entidades no banco de dados (*aluno-service*, *professor-service*, *disciplina-service*, *turma-service* e *ata-service*), um microserviço de interface com o usuário (*web-service*), um microserviço de autenticação (*login-service*) e o Eureka. O *web-service*

atende às requisições do usuário, e portanto terá que se relacionar com cada um dos outros cinco microsserviços do Core para atendê-las. Além disso, ele terá que usar o *login-service* para fazer a autenticação dos seus usuários, que por sua vez, precisa se comunicar com *aluno-service* e *professor-service*. Estes sete microsserviços se relacionam conforme o diagrama apresentado na figura 4.2.

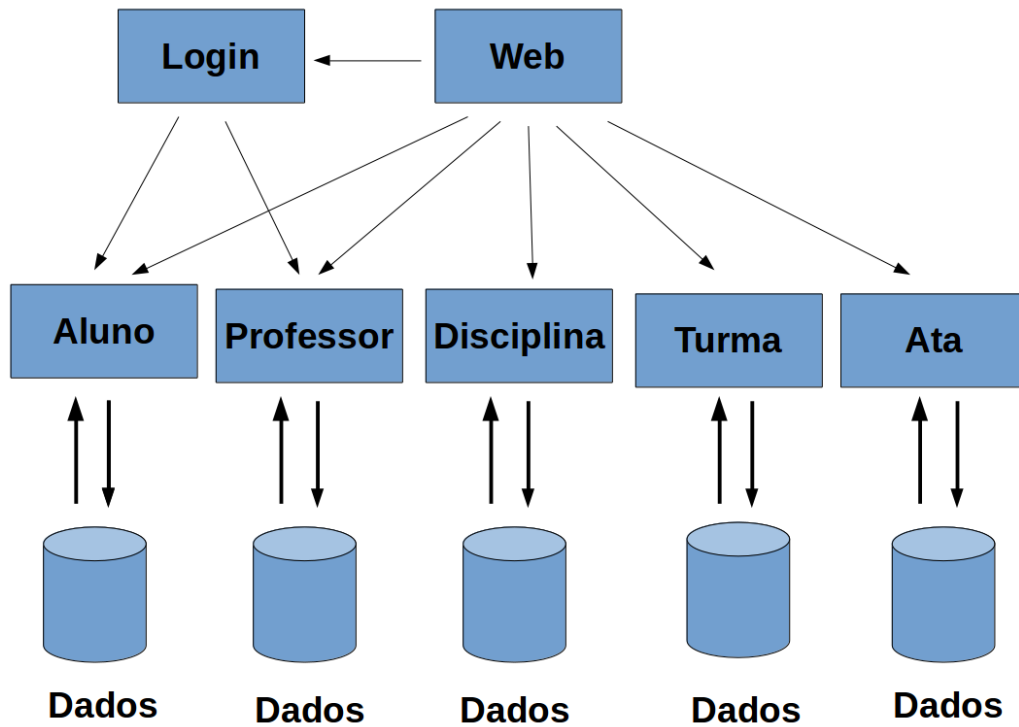


FIG. 4.2: Modelo da arquitetura dos microsserviços

Vale lembrar que todos esses relacionamentos, na verdade são chamadas remotas entre os microsserviços, sendo que a comunicação é feita por chamadas RESTFull e utilizando JSON como forma de transmissão de entidades, e portanto são custosos. Visto isso, optou-se por utilizar um banco de dados não relacional orientado a documento, que evita junções para leituras, já que os objetos são salvos diretamente com as agregações necessárias, exatamente como mostrado na figura 4.1 e que pode ser melhor visto nos arquivos de modelos em JSON no apêndice 9.1. A comunicação ainda se faz necessária para escritas e alterações, mas como as entidades serão mais lidas do que escritas, a opção de banco não relacional adapta-se melhor no sistema.

A figura 4.2 explicita todas as chamadas feitas pelos microsserviços, onde observa-se que os serviços do Core não se relacionam diretamente, pois não precisam fazer chamadas uns aos outros para obter dados. O *web-service* faz esse serviço, obtendo os dados necessários e usando cada microsserviço do Core. Por exemplo, mesmo que *turma-service*

contenha uma lista de disciplinas, o *web-service* é quem faz uma chamada ao *disciplina-service* para buscar essa lista, e então enviá-la pronta ao *turma-service*, não havendo portanto, relação direta entre os microsserviços do Core.

## 5 IMPLEMENTAÇÃO DO SGF

### 5.1 IMPLEMENTAÇÃO DO *AUXILIAR-SERVICE*

Primeiramente, para que um sistema com microsserviços seja possível, é necessário um serviço que autentique os demais, ou seja, um serviço no qual todos os microsserviços ativos se registram e podem saber o endereço dos outros. A ferramenta escolhida foi o Eureka, descrito na seção 3.2. A sua implementação se dá de forma simplificada sendo somente necessário uma classe e um arquivo de configuração.

A única classe desse serviço é o `RegistrationServer`:

```
@SpringBootApplication
@EnableEurekaServer
public class RegistrationServer {
    public static void main(String[] args) {
        SpringApplication.run(RegistrationServer.class, args);
    }
}
```

A *annotation* `@SpringBootApplication` é a responsável por toda a configuração automática necessária para o funcionamento do serviço. A *annotation* `@EnableEurekaServer` inicializa um servidor Eureka que será utilizado para o registro dos microsserviços. A configuração fica no arquivo `application.yml`, onde fica especificado a porta de funcionamento e, se necessário, outros parâmetros que podem ser necessários.

### 5.2 IMPLEMENTAÇÃO DO *ALUNO-SERVICE*

O objetivo deste microsserviço é fazer um CRUD completo de um Aluno, comunicando-se com o banco de dados.

Portanto, a primeira classe necessária é `AlunoRepository` (completa no apêndice 9.2, que será responsável pela interface do serviço com o MongoDB. Nessa classe, utiliza-se o componente `MongoOperations` para fazer inserções, buscas, atualizações e exclusões no `MongoDb`. Foram implementados métodos para interfacear essas operações diretas no banco, como por exemplo, criar um novo aluno:

```

public Aluno create(Aluno aluno) {
    mongo.save(aluno);
    return aluno;
}

```

A *annotation* `@Component` indica que a classe é um componente genérico e o torna apto a ser automaticamente instanciado pelo Spring se a *annotation* `@ComponentScan` estiver presente na classe de configuração. Com o `@Autowired` indica-se que um componente ou funcionalidade da classe X depende de uma classe Y, o que o próprio Framework tenta resolver atribuindo essa dependência a algum componente já instanciado que seja parente da classe Y.

A segunda classe, `AlunoController` (completa no apêndice 9.3), recebe as requisições e então utiliza o `AlunoRepository` para atendê-las. Como o desenvolvimento de microserviços prega o uso do padrão REST API para a comunicação dos serviços, utiliza-se a *annotation* `@RestController`. As funções dessa classe recebem uma requisição HTTP, acessam o `AlunoRepository` e retornam os dados JSON, como por exemplo, para criar um aluno:

```

@RequestMapping(method = RequestMethod.POST)
public Aluno create(@RequestBody Aluno aluno){
    return repo.create(aluno);
}

```

Define-se a URL que cada método atenderá através do *@RequestMapping*, e se a requisição é do tipo POST, GET, UPDATE ou DELETE usando o *RequestMethod*. Os parâmetros podem ser extraídos da URL, através do *@PathVariable* ou extraídos do corpo da requisição, usando *@RequestBody*. Após isso, o método pode usar a classe já criada `AlunoRepository` (repo) para acessar o banco e então, atender à requisição.

Por fim, esse serviço precisa estar disponível para que outros serviços possam usá-lo e para isso a classe `AlunoServer` é responsável por ativar o microserviço, tomando como base o arquivo `aluno-server.yml`. O Spring Framework trata do registro do serviço no Eureka utilizando a *annotation* `@EnableDiscoveryClient`, que permite que o Eureka reconheça o `aluno-server` como um microserviço ativo. A *annotation* `@SpringBootApplication` é equivalente a usar `@Configuration`, `@EnableAutoConfiguration` e `@ComponentScan`. A primeira indica a utilização da classe como forma de configuração, em detrimento as baseadas em arquivos XML. A segunda permite a realização das operações base já explicadas na seção 3.1.1. E a última informa que o Spring Framework deve procurar dentro do



*classpath* os componentes que serão instanciados para a utilização da aplicação.

Com isso, tem-se o primeiro microserviço ativo. Assim como *aluno-service*, todos os serviços do Core seguem a mesma implementação, apenas variando as rotas no *controller* e as funções no *repository* para se adequarem às necessidades de cada entidade.

### 5.3 IMPLEMENTAÇÃO DO *WEB-SERVICE*

Este é o maior e mais complexo de todos os serviços, pois possui uma interface com o usuário e uma com cada microserviço do Core. O objetivo é ser acessado por usuários através de um navegador e atender às requisições utilizando os serviços do Core.

É necessária uma classe para obter os dados de cada entidade, comunicando-se com os microserviços do Core. *AlunoService* (completa no apêndice 9.4) é uma destas classes, responsável por buscar os dados necessários para atender às requisições utilizando as rotas de *aluno-service*. Para criar um aluno usa-se a função abaixo:

```
@LoadBalanced
private RestTemplate restTemplate;

public Aluno create(Aluno aluno){
    return restTemplate.postForObject(
        ServiceName.aluno , aluno , Aluno.class );
}
```

Usa-se um atributo do tipo *RestTemplate*, que envia e recebe objetos abstraindo as chamadas remotas além de realizar o balanceamento de carga dos microserviços operantes. Dentro das dependências iniciais está o *spring-cloud-starter-ribbon*, que é o balanceador de carga utilizado, além desse atributo possuir a *annotation* *@LoadBalanced* que indica que o Cliente Ribbon será utilizado para a realização do balanceamento de carga, como explicado na seção 3.2. Para encontrar o microserviço desejado, usa-se a classe *ServiceName*, que mapeia os nomes dos serviços nos endereços, porém utilizando os nomes registrados no Eureka. Por exemplo, *ServiceName.aluno* é mapeado no endereço *http://aluno-service*, sendo o Eureka o responsável por fornecer os endereços corretos desse caminho, ficando a cargo do *restTemplate* escolher qual deles utilizar.

Feita a classe que recupera os dados, precisa-se da classe que receberá as requisições e utilizará *AlunoService* para atendê-las. *AlunoController* (completa no apêndice 9.5), assim como a classe da seção 5.2, utiliza o padrão REST para receber requisições, porém retornando páginas HTML, pois se trata do usuário final.

Agora, para a criação de um aluno, é preciso primeiro obter uma página para preencher os dados do novo aluno. Este método é apresentado como `newAluno`, o qual apenas verifica se o usuário autenticado é um coordenador e retorna a página de criação.

```
@Autowired
private AlunoService service;

@RequestMapping(value = "/new", method = RequestMethod.GET)
public String newAluno(Model model, Principal u){
    User user = new User((UsernamePasswordAuthenticationToken) u);

    if(!user.isCoordenador())
        return "redirect:/403";

    model.addAttribute("title", "Novo_Aluno");
    model.addAttribute("user", user);
    return "aluno/new";
}
```

Após o preenchimento, o método `createAluno` é chamado, pois é um POST, com os dados do novo aluno preenchidos no formulário. Nesse método têm-se os parâmetros sendo passados com a annotation `@RequestParam`, que serão usados para gerar o novo aluno. Como feito no Core, o método usa o `AlunoService` (`service`) para chamar o método já criado. Após a criação, retorna para a página que mostra o aluno criado.

```
@RequestMapping(value = "/new", method = RequestMethod.POST)
public String createAluno(@RequestParam("nome") String nome,
    @RequestParam("email") String email, @RequestParam("numero")
    String numero, @RequestParam("password") String password,
    Principal u){
    User user = new User((UsernamePasswordAuthenticationToken) u);

    if(!user.isCoordenador())
        return "redirect:/403";

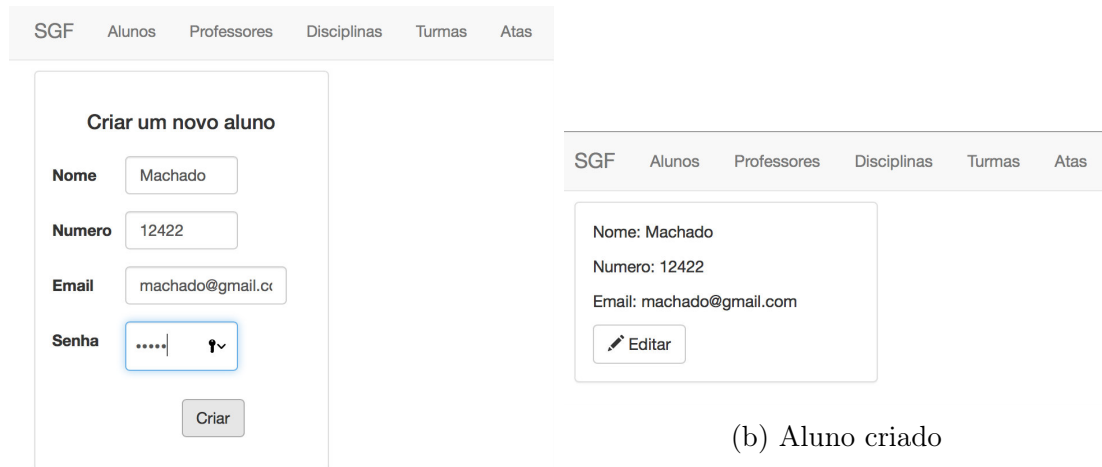
    Aluno aluno = new Aluno(numero, nome, email, password);
    service.create(aluno);
}
```

```

    return "redirect:" + "/aluno/numero/" + numero;
}

```

Dessa forma, um coordenador pode criar um aluno acessando este serviço, fazendo como mostrado na figura 5.1.



(a) Tela onde coordenador pode criar um novo aluno

(b) Aluno criado

FIG. 5.1: Sequência de telas onde coordenador cria um novo aluno

O mesmo é feito para interfaciar Professores, Turmas, Disciplinas e Atas de faltas, apenas com uma diferença de complexidade, pois quanto mais relacionamentos uma entidade tem, mais services o controller precisará usar para atender uma requisição. Na seção 5.5, será apresentada a implementação da interface do WebService com Ata para explicitar esse aumento de complexidade.

## 5.4 IMPLEMENTAÇÃO DO *LOGIN-SERVICE*

No sistema, certas funcionalidades são diferentes para alunos, professores e coordenadores e por isso é necessário diferenciar os usuários no sistema. Para isso, foi feito o serviço de login. O login no sistema é feito com a utilização do sistema fornecido pelo Spring Security. O *web-service* é o microserviço que utiliza essa ferramenta oferecida pelo Spring Framework, quando é informado a dependência do *spring-boot-starter-security*.

O sistema possui dois tipos de usuários distintos, aluno e professor, que por sua vez pode também ser coordenador, portanto o *login-service* é utilizado para juntar os clientes que estão em bases distintas para um único tipo que será usado pelo Spring Security para ser autenticado.

LoginController: Classe do *login-service* que recebe requisições de clientes através de seu *id*. Ela identifica se o cliente é professor, se a identificação informada for um email, ou aluno, se for um número. Em seguida faz a busca do cliente pelo microserviço de aluno ou de professor, e retorna um objeto do tipo Client que é a abstração do usuário do sistema.

LoginService: Classe do *web-service* responsável por comunicar-se com o *login-service* e obter um objeto do tipo Client para ser utilizado pelo componente de autenticação. A forma de comunicação com o microserviço se dá de forma bem similar ao WebService explicado na seção 5.3 e as demais classes do tipo *Service*.

SGFUserDetailsService: Classe que implementa a interface UserDetailsService que faz uso do LoginService para a forma de se encontrar usuários cadastrados no sistema.

SecurityConfig: Classe de configuração do *web-service* que permite o Spring Security operar como forma de autenticação do sistema e onde é indicado quais páginas devem ser acessadas por usuários autenticados e onde não é exigido autenticação. O atributo do tipo UserDetailsService é instanciada com o SGFUserDetailsService automaticamente pelo Spring através da *annotation* *@Autowired* e ele é passado como forma de se obter detalhes dos usuários ao se sobrescrever o método *configure(AuthenticationManagerBuilder auth)*.

## 5.5 CICLO DE USO DE UMA ATA

A última parte do *web-service* a ser feita foi a interface com Ata, pois se relaciona com todos os serviços anteriores. O AtaService é implementado de forma muito similar ao AlunoService já apresentado na seção 5.3, apenas usando as rotas de ata-service no Core. A diferença de complexidade está no controller, que precisará acessar não só o AtaService, mas também os demais AlunoService, TurmaService, DisciplinaService e ProfessorService, como declarado abaixo, no início de AtaController (completa no apêndice 2):

```
@Autowired
private AlunoService alunoService;
@Autowired
private ProfessorService professorService;
@Autowired
private DisciplinaService disciplinaService;
@Autowired
private TurmaService turmaService;
@Autowired
```

```
private AtaService service;
```

Para criar uma ata, a primeira tela vista é a de preenchimento da data, mostrada na figura 5.2.

Data	Ano	Engenharia	Status	
02/09/2016	1	Computação	Assinada	
31/08/2016	1	Computação	Pendente	
03/09/2016	1	Computação	Pendente	

Nova Ata

14/09/2016

September 2016

Su	Mo	Tu	We	Th	Fr	Sa
28	29	30	31	1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	1
2	3	4	5	6	7	8

FIG. 5.2: Tela de preenchimento da data

A data é passada como parâmetro para o método newAta:

```
@RequestMapping(value = "/new", method = RequestMethod.POST)
public String newAta(Model model, Principal u, @RequestParam("data")
String data) {
    User user = new User((UsernamePasswordAuthenticationToken) u);

    if (!user.isAluno())
        return "redirect:/403";

    String numero = user.getName();
    Aluno xerife = alunoService.getByNumero(numero);
```

```

Turma turmaDoXerife = turmaService.getByAluno(xerife);

if (turmaDoXerife == null)
    return "redirect:/403";

Ata ata = new Ata();
ata.setData(data);
ata.setXerife(new Xerife(xerife));
ata.setTurma(turmaDoXerife.getId());
ata.setCoordenador(new Coordenador(
    turmaService.getCoordenador(turmaDoXerife), null));
ata.getTempos().add(new Tempo("1"));
ata = service.create(ata);

return "redirect:" + "/ata/edit/" + ata.getId();
}

```

O método verifica se o usuário é um aluno, pois apenas alunos podem criar atas. Para recuperar o aluno chefe de turma, o método precisa usar o `alunoService`, que acessará o *aluno-service* do Core para buscar a informação no banco, como já apresentado. O mesmo é feito para buscar a turma do aluno chefe de turma. Uma vez que estes dados foram buscados, uma nova ata é criada e o sistema redireciona para a edição de ata, chamando o método `editAta`:

```

@RequestMapping(value = "/edit/{id}", method = RequestMethod.GET)
public String editAta(@PathVariable("id") String id, Model model,
Principal u) {
    (...)
    Ata ata = service.getById(id);
    (...)
    if (Utils.recebeuAssinaturaXerife(ata.getXerife())) {
        model.addAttribute("title", "Ata_de_Faltas");
        model.addAttribute("ata", ata);
        model.addAttribute("user", user);
        return "ata/assinadaXerife";
    }
}

```

```

Turma turmaDoXerife = turmaService.getById(ata.getIdTurma());
(...)
model.addAttribute("ata", ata);
model.addAttribute("disciplinas", turmaDoXerife.getDisciplinas());
model.addAttribute("title", "Editar_Ata_de_Faltas");
model.addAttribute("user", user);
return "ata/edit";
}

```

Através do *id* o método recupera a ata, usando o AtaService. Caso a ata já esteja assinada pelo chefe de turma, ele não pode mais editar e portanto é redirecionado para a página que apenas visualiza a ata. Usando o TurmaService o método recupera a turma do chefe de turma, e então pode-se acessar os alunos da turma e as disciplinas que a turma possui. Diante desses dados, a página de edição da ata pode ser apresentada, como na figura 5.3.

SGF
Perfil
Atas

Ata de Faltas

Data
14/09/2016

Ordem
1

Disciplina
APS

Faltas

André
Luiz
Santos

Salvar

Salvo

Aluno	Numero	Justificativa

Assinar ata

FIG. 5.3: Tela de edição da ata de faltas

Nessa tela, o usuário pode preencher os tempos de aula com a disciplina e as faltas do tempo. Na figura 5.3 observa-se que o campo de Faltas é preenchido selecionando-se um aluno dentre todos da turma. Por isso era necessário acessar o *turma-service* e buscar a turma do chefe de turma, assim é possível exibir todos os alunos da mesma. O mesmo é feito com as disciplinas da turma.

Após o preenchimento, o chefe de turma pode salvar o tempo de aula, o que faz um post para o método `saveAta`:

```

@RequestMapping(value = "/edit/{id}", method = RequestMethod.POST)
public String saveAta(@PathVariable("id") String id,
    @RequestParam("ordem") String ordem, Model model,
    @RequestParam(value = "alunos", required = false,
        defaultValue = "") String[] alunos,
    @RequestParam("disciplina") String idDisciplina, Principal u) {
    (...)
}

```



```

Ata ata = service.getById(id);
(...)
Turma turmaDoXerife = turmaService.getById(ata.getIdTurma());
Tempo tempo = new Tempo();
tempo.setOrdem(ordem);

for (String numero : alunos) {
    Aluno actual = alunoService.getByNumero(numero);
    if (actual == null)
        continue;
    tempo.getFaltas().add(actual);
    int i;
    for (i = 0; i < ata.getFaltas().size(); i++) {
        if (ata.getFaltas().get(i).getAluno().equals(actual))
            break;
    }
    if (i >= ata.getFaltas().size()) {
        ata.getFaltas().add(new Falta(actual, "Destino_Indeterminado"));
    }
}

Disciplina disciplina = disciplinaService.getById(idDisciplina);
tempo.setDisciplina(disciplina);
(...)
service.update(ata);
(...)
model.addAttribute("ata", ata);
model.addAttribute("user", user);
model.addAttribute("title", "Editar_AtadeFaltas");
return "/ata/edit";
}

```

Esse método usa o AtaService e TurmaService assim como o outro, cria um novo tempo de aula, adiciona o número de ordem e os alunos faltosos, recupera a disciplina do tempo usando DisciplinaService e adiciona-a ao tempo, e então redireciona para a página

de edição novamente.

Na página de edição o chefe de turma pode também clicar para justificar a falta de um aluno, quando alguma falta já tiver sido lançada, levando assim para a página de justificativa. O método de justificar falta apenas adiciona a justificativa e redireciona para a tela de edição novamente.

O professor de cada disciplina, ao entrar no sistema, pode visualizar as atas de falta que possuem tempos lecionados por ele, e então assinar os tempos. A figura 5.4 mostra a sequência de ações de um professor para assinar seus tempos. O método de assinar atas professorAssinar recupera o professor usando o ProfessorService e verifica se o professor autenticado é o mesmo do tempo de aula e então realiza a assinatura, conforme o trecho de código abaixo:

```
if (!ata.getTempos().get(ind).getDisciplina()  
.getProfessor().getEmail().equals(user.getName()))  
    return "redirect:/403";  
Professor professor = professorService  
.getByEmail(user.getName());  
ata = Assinatura.assinarProfessor(professor, ata, ind);
```

The figure consists of two side-by-side screenshots of a web application interface titled 'Ata de Faltas'. Both screenshots have a top navigation bar with links: SGF, Alunos, Professores, Disciplinas, Turmas, and Atas. The main content area shows a form for signing attendance. In both, the 'Data' is 14/09/2016, 'Ordem' is 1, and 'Disciplina' is APS. In (a), the 'Faltas' dropdown menu is open, showing 'André' as the selected student, and there is an 'Assinar Tempo' button at the bottom. In (b), the 'Assinado' button is highlighted, indicating the action has been completed.

(a) Tela onde professor pode assinar seus tempos (b) Tela com o tempo assinado pelo professor

FIG. 5.4: Sequência de telas de assinatura do professor

Após todos os tempos estarem assinados pelos respectivos professores, o aluno chefe de turma pode assinar a ata, clicando em "Assinar ata" e então finalizar sua edição. A sequência de ações e o método de assinatura são similares as do professor. Por fim, o coordenador pode visualizar as atas de faltas assinadas pelo chefe de turma e então, assiná-las para finalmente serem arquivadas.

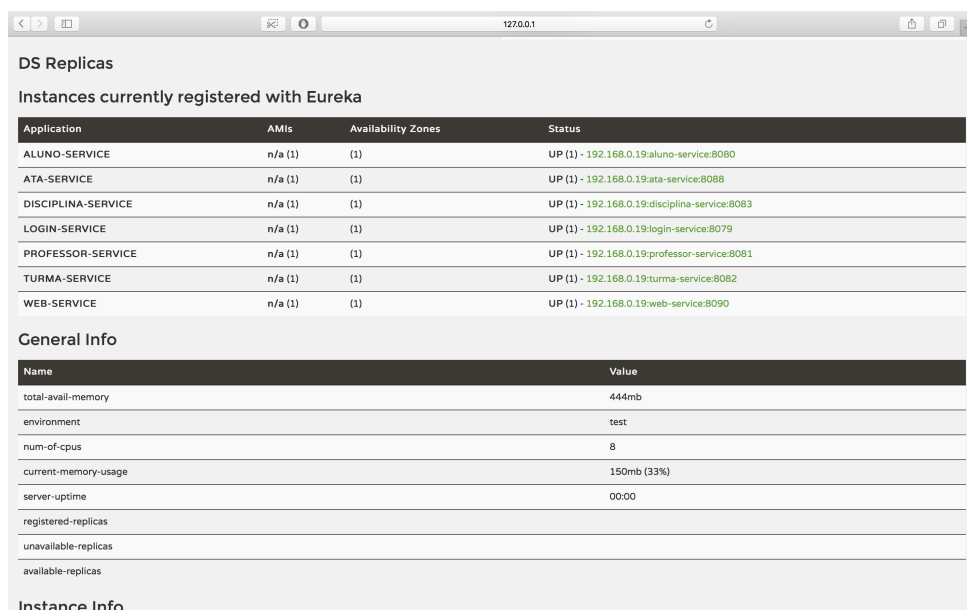
## 6 AMBIENTES DE FUNCIONAMENTO

A grande vantagem da utilização da arquitetura de microsserviços é a possibilidade da execução de partes dos sistemas em mais de um servidor de forma rápida e dinâmica. Para o sistema SGF foram testados três tipos de ambiente: execução do sistema completo em um único servidor; utilização de máquinas virtuais para a execução dos diversos microsserviços e no AWS.

### 6.1 SERVIDOR ÚNICO

A utilização de um único servidor para a execução do sistema assemelha-se à arquitetura monolítica, o ponto de falha e o gargalo se concentram em uma única máquina. Mesmo com suas desvantagens esse ambiente é excelente para produção, porque ele é fácil de se montar, fazer testes e descobrir possíveis falhas. Enquanto o SGF era desenvolvido, esse ambiente foi escolhido para a realização dos testes.

Como o sistema foi construído com dois desenvolvedores, as configurações da máquina local divergem um pouco, mas em todos os testes foi utilizado o ambiente Java 8 desenvolvido pela Oracle e o MongoDB 3.2. Pode-se verificar na figura 6.1, a tela de monitoramento do Eureka, todos os serviços e suas respectivas portas.



The screenshot shows the Eureka monitoring interface in a web browser. The address bar shows '127.0.0.1'. The page title is 'DS Replicas'. Below the title, it says 'Instances currently registered with Eureka'. There is a table with four columns: 'Application', 'AMIs', 'Availability Zones', and 'Status'. The table lists several services: ALUNO-SERVICE, ATA-SERVICE, DISCIPLINA-SERVICE, LOGIN-SERVICE, PROFESSOR-SERVICE, TURMA-SERVICE, and WEB-SERVICE. Each service has a status of 'UP (1)' and a specific port number. Below the table, there is a section titled 'General Info' with a table of system metrics. The metrics include total-avail-memory (444mb), environment (test), num-of-cpus (8), current-memory-usage (150mb (33%)), server-up-time (00:00), registered-replicas, unavailable-replicas, and available-replicas. At the bottom, there is a section titled 'Instance Info'.

Application	AMIs	Availability Zones	Status
ALUNO-SERVICE	n/a (1)	(1)	UP (1) - 192.168.0.19:aluno-service:8080
ATA-SERVICE	n/a (1)	(1)	UP (1) - 192.168.0.19:ata-service:8088
DISCIPLINA-SERVICE	n/a (1)	(1)	UP (1) - 192.168.0.19:disciplina-service:8083
LOGIN-SERVICE	n/a (1)	(1)	UP (1) - 192.168.0.19:login-service:8079
PROFESSOR-SERVICE	n/a (1)	(1)	UP (1) - 192.168.0.19:professor-service:8081
TURMA-SERVICE	n/a (1)	(1)	UP (1) - 192.168.0.19:turma-service:8082
WEB-SERVICE	n/a (1)	(1)	UP (1) - 192.168.0.19:web-service:8090

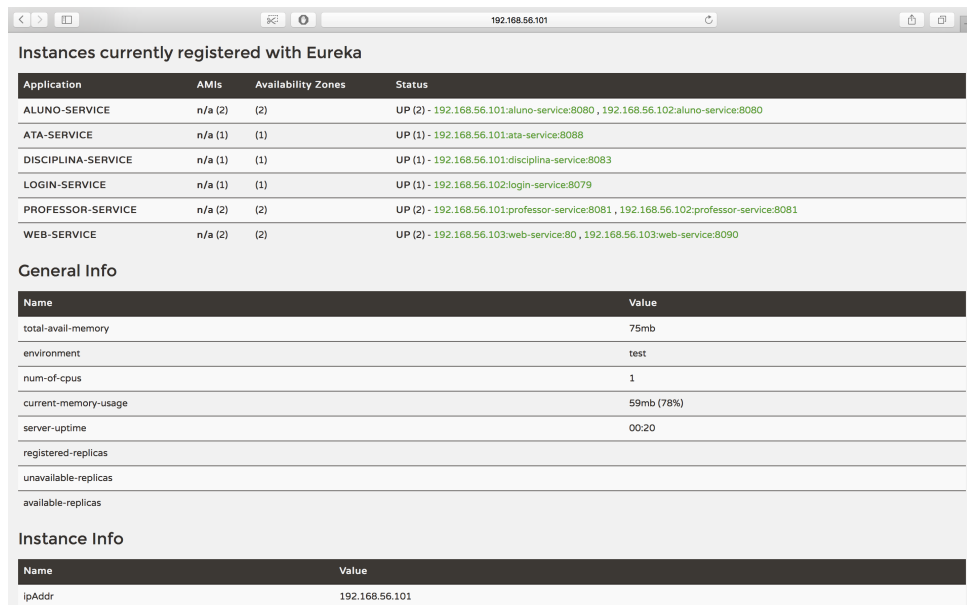
Name	Value
total-avail-memory	444mb
environment	test
num-of-cpus	8
current-memory-usage	150mb (33%)
server-up-time	00:00
registered-replicas	
unavailable-replicas	
available-replicas	

FIG. 6.1: Tela de monitoramento do Eureka da máquina local

## 6.2 MÁQUINAS VIRTUAIS

A utilização de diversas máquinas virtuais aproveita melhor os benefícios da arquitetura estudada. Pode-se subir os diversos microsserviços com o sistema em funcionamento, sem atrapalhar a execução dos usuários. Portanto, caso algum ponto do sistema esteja sobrecarregado ou algum servidor sair do ar, o administrador do sistema ou um programa de monitoração podem levantar outra máquina virtual com os microsserviços necessários, sem ser necessário a parada do sistema.

Para o SGF, foi testado a execução em três máquinas virtuais distintas utilizando o VirtualBox. Cada servidor possui 768 MB de memória RAM e possui um *core* virtual, todos rodando o Sistema Operacional Ubuntu Server 16.04.1 LTS e estando em uma mesma rede local. Todos os sistemas possuem o ambiente Java versão 8 desenvolvida pela Oracle instalado, sendo que um deles possui o MongoDB versão 3.2. No primeiro sistema, com o mongo instalado, foi executado o servidor Eureka e os microsserviços core: Aluno, Professor, Disciplina, Turma e Ata. Na segunda máquina virtual foram executados os microsserviços Login, Aluno e Professor, se conectando ao banco de dados da primeira máquina. Por fim, no terceiro e último servidor virtual o microsserviço Web. Pode-se verificar na figura 6.2, a tela de monitoramento do Eureka, todos os serviços em execução.



The screenshot shows the Eureka monitoring interface in a web browser. The address bar shows the URL 192.168.56.101. The page title is "Instances currently registered with Eureka". Below the title is a table with columns: Application, AMIs, Availability Zones, and Status. The table lists several services: ALUNO-SERVICE, ATA-SERVICE, DISCIPLINA-SERVICE, LOGIN-SERVICE, PROFESSOR-SERVICE, and WEB-SERVICE. Each service has a status of "UP" and a list of instances. Below the table is a section titled "General Info" with a table of system metrics. The metrics include total-avail-memory (75mb), environment (test), num-of-cpus (1), current-memory-usage (59mb (78%)), server-uptime (00:20), registered-replicas, unavailable-replicas, and available-replicas. At the bottom is a section titled "Instance Info" with a table of instance details, including ipAddr (192.168.56.101).

Application	AMIs	Availability Zones	Status
ALUNO-SERVICE	n/a (2)	(2)	UP (2) - 192.168.56.101:aluno-service:8080 , 192.168.56.102:aluno-service:8080
ATA-SERVICE	n/a (1)	(1)	UP (1) - 192.168.56.101:ata-service:8088
DISCIPLINA-SERVICE	n/a (1)	(1)	UP (1) - 192.168.56.101:disciplina-service:8083
LOGIN-SERVICE	n/a (1)	(1)	UP (1) - 192.168.56.102:login-service:8079
PROFESSOR-SERVICE	n/a (2)	(2)	UP (2) - 192.168.56.101:professor-service:8081 , 192.168.56.102:professor-service:8081
WEB-SERVICE	n/a (2)	(2)	UP (2) - 192.168.56.103:web-service:80 , 192.168.56.103:web-service:8090

Name	Value
total-avail-memory	75mb
environment	test
num-of-cpus	1
current-memory-usage	59mb (78%)
server-uptime	00:20
registered-replicas	
unavailable-replicas	
available-replicas	

Name	Value
ipAddr	192.168.56.101

FIG. 6.2: Tela de monitoramento do Eureka das máquinas virtuais

Nesse ambiente verificou-se a estabilidade do sistema, porque os microsserviços Aluno e Professor da primeira máquina virtual foram encerrados propositalmente, enquanto um usuário utilizava o sistema. Como esperado, o sistema continuou o seu funcionamento

normalmente, porque o segundo servidor ainda possuía os microsserviços em funcionamento.

### 6.3 AMAZON WEB SERVICES

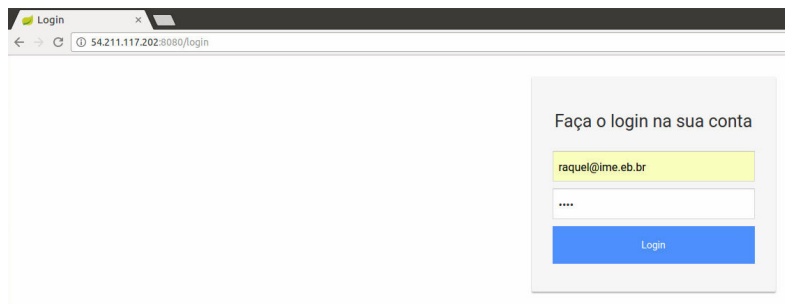
Implantar sistemas num servidor em nuvem é uma forte tendência atual, então a simulação feita no AWS é a mais real em se tratando de um sistema em produção.

Foi escolhido o AWS por ser um ambiente *cloud* amplamente utilizado e pela possibilidade de execução de um micro servidor de forma gratuita para testes. Foi utilizado o Ubuntu Server 14.04 LTS como o sistema operacional, com o MongoDB versão 3.2 e o ambiente Java 8 desenvolvido pela Oracle. Os testes não foram realizados de forma completa, porque o servidor gratuito fornecido pelo AWS não possui memória suficiente para rodar todos os microsserviços do SGF.

A simulação mais real do ambiente em produção seria colocar cada microsserviço em uma instância diferente, o que não foi possível pois a máquina gratuita do AWS dá direito a apenas uma instância.

Portanto, foram iniciados somente o Eureka, *professor-service*, *login-service* e *web-service*, que é o mínimo para um usuário fazer *login* no sistema e realizar um CRUD. Pode ser visto na figura 6.3, o acesso ao endereço do servidor AWS com parte do SGF em funcionamento. A possibilidade de se rodar apenas alguns microsserviços no AWS e o restante em máquinas locais foi levantada, mas seria necessário um IP real para a máquina local, para que os serviços pudessem se comunicar.

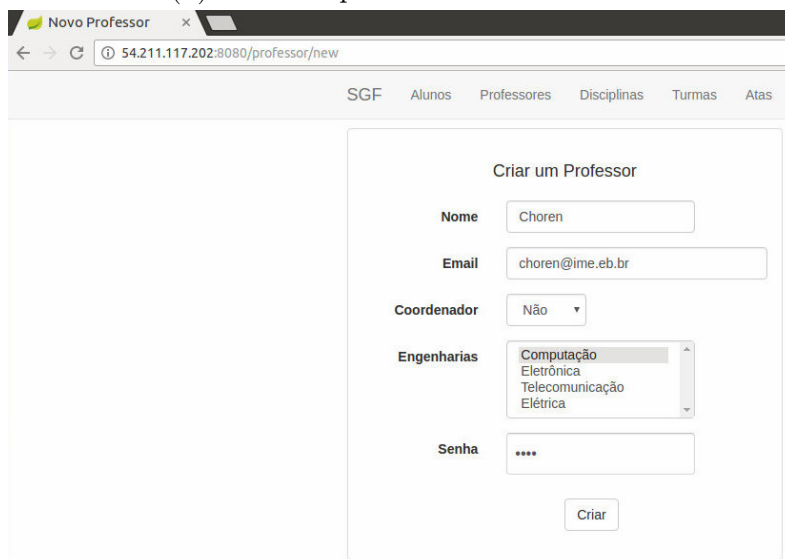
Entretanto, o conjunto de serviços *eureka*, *professor*, *login* e *web* é suficiente para mostrar uma funcionalidade sendo executada. Na figura 6.3, temos um CRUD de professor sendo feito.



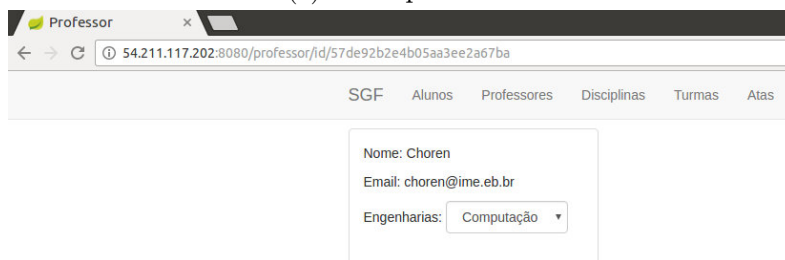
(a) Login de um coordenador



(b) Todos os professores cadastrados



(c) Novo professor



(d) Professor criado

FIG. 6.3: Sequência de telas para criar um professor com sistema no AWS

## 7 CONCLUSÃO

Após o estudo dos conceitos da arquitetura de microsserviços, foi proposto um sistema que pudesse demonstrar as peculiaridades desta arquitetura de desenvolvimento, o Sistema de Gerenciamento de Faltas (SGF). O trabalho apresentou, no capítulo 4, o projeto do SGF segundo a ótica de microsserviços, especificando os módulos necessários e relacionando-os.

Finda a fase de projeto, os seguintes microsserviços foram implementados:

- Aluno-service
- Professor-service
- Disciplina-service
- Turma-service
- Ata-service
- Login-service
- Web-service

O capítulo 5 foi dedicado à implementação do SGF. O desenvolvimento dos primeiros microsserviços foram os mais trabalhosos, porque existiu o tempo de adaptação à essa arquitetura, mas conforme o sistema foi sendo construído, mais fluído se tornou a criação dos microsserviços. O grande problema encontrado em todo o projeto foi como identificar quais funcionalidades poderiam ser separadas em microsserviços e quais deveriam ser agregados a outros, algumas vezes o planejamento do sistema inicial teve que ser alterado para se adaptar à arquitetura utilizada.

Por fim, no capítulo 6 se encontram os resultados do SGF em funcionamento nos três ambientes propostos: local, máquinas virtuais e na nuvem. O ambiente local assemelha-se com um sistema monolítico sendo recomendado para o desenvolvimento e a realização de testes. A tolerância a falha dessa arquitetura pode ser verificada com os testes em servidores virtuais. Dois microsserviços foram encerrados durante o uso do sistema, mas como existiam mais instâncias deles em execução, o usuário não foi afetado. O ambiente na nuvem não possibilitou o pleno funcionamento do SGF, porque o AWS fornece uma

máquina com recursos limitados para testes, portanto foi somente executado uma pequena parte do SGF para a realização de um CRUD da entidade professor.

Após o estudo e a implementação de um sistema com a arquitetura de microsserviços fica evidente a resiliência dessa arquitetura. Existe contudo, um tempo de adaptação para os desenvolvedores se adequarem as peculiaridades dos microsserviços, tanto na parte de projeto quanto na codificação e utilização das ferramentas.

O trabalho foi focado apenas no projeto e desenvolvimento de um sistema, as falhas foram tratadas de forma manual através de um administrador. Fica aberta a possibilidade de projetos futuros implementarem um ambiente onde sistemas baseados em microsserviços sejam administrados de forma automática, com o levantamento de microsserviços e a criação de máquinas virtuais de forma dinâmica e sem interferência humana.



## 8 REFERÊNCIAS BIBLIOGRÁFICAS

- Spring Cloud. Disponível em: <<http://projects.spring.io/spring-cloud/>>. Acesso em: 06/05/2016.
- ADRIAN COCKCROFT. Migrating to Microservices. Disponível em: <<https://www.infoq.com/presentations/migration-cloud-native>>. Acesso em: 07/05/2016.
- EDUARDO BARRA CORDEIRO. SOA - Arquitetura Orientada a Serviços. Disponível em: <<http://blog.iprocess.com.br/2012/10/soa-arquitetura-orientada-a-servicos/>>. Acesso em: 06/05/2016.
- ANDRÉ FARIAS. Microservices com Netflix OSS. Disponível em: <<http://blog.andrefaria.com/monolitico-aos-microservices-com-netflix-oss>>. Acesso em: 06/05/2016.
- MARTIN FOWLER. Microservices. Disponível em: <<http://martinfowler.com/articles/microservices.html>>. Acesso em: 06/05/2016.
- MARTIN FOWLER. Microservices Guide. Disponível em: <<http://martinfowler.com/microservices/>>. Acesso em: 06/05/2016.
- LONGA, RICARDO. Um caso real: Microservices com Dropwizard, SparkJava, SpringBoot e VertX. Disponível em: <<http://www.infoq.com/br/presentations/microservices-com-dropwizard-sparkjava-springboot-vertx>>. Acesso em: 06/05/2016.
- NEWMAN, S. **Building Microservices: Designing Fine-Grained Systems**. 1st. ed. [S.l.]: O'Reilly Media, 2015. 280 p. ISBN 978-1491950357.
- SILVEIRA, P.; SILVEIRA, G.; LOPES, S.; MOREIRA, G.; STEPPAT, N. ; KUNG, F. **Introdução à Arquitetura e Design de Softwares**. 1. ed. [S.l.]: Casa do Código, 2012.
- MATT STINE. Build self-healing distributed systems with Spring Cloud. Disponível em: <<http://www.javaworld.com/article/2927920/cloud-computing/build-self-healing-distributed-systems-with-spring-cloud.html>>. Acesso em: 06/05/2016.

WALLS, C. Bootstarting spring. In: \_\_\_\_\_. **Spring Boot in Action**. Shelter Island, NY: Manning Publications, 2015. p. 1–8. ISBN 9781617292545.

## 9 APÊNDICES

## APÊNDICE 1: TEMPLATE DOS ARQUIVOS JSON DAS ENTIDADES

```
"aluno": {  
  "id": "_",  
  "nome": "_",  
  "email": "_",  
  "senha": "_",  
  "engenharia": "_"  
}
```

```
"professor": {  
  "id": "_",  
  "nome": "_",  
  "email": "_",  
  "coordenador": "_",  
  "senha": "_",  
  "engenharias": "_"  
}
```

```
"disciplina": {  
  "id": "_",  
  "nome": "_",  
  "professor": {  
    "id": "_",  
    "nome": "_",  
    "email": "_",  
    "coordenador": "_",  
    "senha": "_",  
    "engenharias": "_"  
  }  
}
```

```

"turma": {
  "id": "_",
  "ano": "_",
  "engenharia": "_",
  "disciplinas": [
    {
      "id": "_",
      "nome": "_",
      "professor": {
        "id": "_",
        "nome": "_",
        "email": "_",
        "coordenador": "_",
        "senha": "_",
        "engenharias": "_"
      }
    }
  ],
  "alunos": [
    {
      "id": "_",
      "nome": "_",
      "email": "_",
      "senha": "_",
      "engenharia": "_"
    }
  ]
}

```

```

"ata": {
  "id": "_",
  "data": "",
  "turma": {
    "id": "_",

```

```

"ano": "_",
"engenharia": "_",
"disciplinas": [
    {
        "id": "_",
        "nome": "_",
        "professor": {
            "id": "_",
            "nome": "_",
            "email": "_",
            "coordenador": "_",
            "senha": "_",
            "engenharias": "_"
        }
    }
],
"alunos": [
    {
        "id": "_",
        "nome": "_",
        "email": "_",
        "senha": "_",
        "engenharia": "_"
    }
]
},
"tempos": [
    {
        "ordem": "_",
        "disciplina": {
            "id": "_",
            "nome": "_",
            "professor": {
                "id": "_",

```

```

        "nome": "_",
        "email": "_",
        "coordenador": "_",
        "senha": "_",
        "engenharias": "_"
    }
},
"faltas": [
    {
        "id": "_",
        "nome": "_",
        "email": "_",
        "senha": "_",
        "engenharia": "_"
    }
],
"assinatura": {
    "idProfessor": "_",
    "assinatura": "_"
}
}
],
"faltas": [
    {
        "aluno": {
            "id": "_",
            "nome": "_",
            "email": "_",
            "senha": "_",
            "engenharia": "_"
        },
        "motivo": "_"
    }
],

```

```

"xerife": {
  "xerife": {
    "id": "_",
    "nome": "_",
    "email": "_",
    "senha": "_",
    "engenharia": "_"
  },
  "assinatura": "_"
},
"coordenador": {
  "coordenador": {
    "id": "_",
    "nome": "_",
    "email": "_",
    "coordenador": "_",
    "senha": "_",
    "engenharias": "_"
  },
  "assinatura": "_"
}
}

```



## APÊNDICE 2: CÓDIGO FONTE DE SGF-CORE/ALUNOREPOSITORY.JAVA

```
package br.eb.ime.comp.pfc.sgf.core.aluno;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.mongodb.core.MongoOperations;
import org.springframework.data.mongodb.core.query.Criteria;
import org.springframework.data.mongodb.core.query.Query;
import org.springframework.data.mongodb.core.query.Update;
import org.springframework.stereotype.Component;

import br.eb.ime.comp.pfc.sgf.models.Aluno;

@Component
public class AlunoRepository {

    @Autowired
    private MongoOperations mongo;

    public Aluno create(Aluno aluno) {
        mongo.save(aluno);
        return aluno;
    }

    public List<Aluno> getAll() {
        return mongo.findAll(Aluno.class);
    }

    public Aluno getByNumero(String numero) {
        Query searchAlunoQuery = new Query(Criteria.where("numero").is(numero))
    }
```

```

        return mongo.findOne(searchAlunoQuery , Aluno.class);
    }

    public Aluno getByEmail(String email) {
        Query searchAlunoQuery = new Query( Criteria.where("email").is(email));
        return mongo.findOne(searchAlunoQuery , Aluno.class);
    }

    public Aluno save(Aluno aluno) {
        Query searchAlunoQuery = new Query( Criteria.where("numero").is(aluno.getNumero()));

        Update update = new Update();
        update.set("nome", aluno.getNome());
        update.set("email", aluno.getEmail());

        mongo.upsert(searchAlunoQuery , update , Aluno.class);
        return this.getByNumero(aluno.getNumero());
    }
}

```

### APÊNDICE 3: CÓDIGO FONTE DE SGF-CORE/ALUNOCONTROLLER.JAVA

```
package br.eb.ime.comp.pfc.sgf.core.aluno;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestBody;

import br.eb.ime.comp.pfc.sgf.models.Aluno;

@RestController
@RequestMapping("/")
public class AlunoController {

    @Autowired
    private AlunoRepository repo;

    @RequestMapping(method = RequestMethod.POST)
    public Aluno create(@RequestBody Aluno aluno){
        return repo.create(aluno);
    }

    @RequestMapping(method = RequestMethod.PUT)
    public Aluno update(@RequestBody Aluno aluno){

        Aluno old = repo.getByNumero(aluno.getNumero());
```

```

        if (old.equals(null)){
            return repo.create(aluno);
        }

        return repo.save(aluno);
    }

    @RequestMapping(method = RequestMethod.GET)
    public List<Aluno> getAll(){
        return repo.getAll();
    }

    @RequestMapping(value =("/{numero}", method = RequestMethod.GET)
    public Aluno getByNumero(@PathVariable("numero") String numero){
        return repo.getByNumero(numero);
    }

    @RequestMapping(value = "/email/{email}", method = RequestMethod.GET)
    public Aluno getByEmail(@PathVariable("email") String email){
        return repo.getByEmail(email);
    }
}

```

#### APÊNDICE 4: CÓDIGO FONTE DE SGF-WEB/ALUNOSERVICE.JAVA

```
package br.eb.ime.comp.pfc.sgf.web.service;

import java.util.Arrays;
import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cloud.client.loadbalancer.LoadBalanced;
import org.springframework.http.HttpEntity;
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpMethod;
import org.springframework.http.HttpStatus;
import org.springframework.http.MediaType;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Service;
import org.springframework.web.client.RestTemplate;

import br.eb.ime.comp.pfc.sgf.models.Aluno;

@Service
public class AlunoService {

    @Autowired
    @LoadBalanced
    private RestTemplate restTemplate;

    public Aluno getByNumero(String numero){
        return restTemplate.getForObject(ServiceName.aluno + "/" + numero,
            Aluno.class, numero);
    }
}
```

```

public Aluno create(Aluno aluno){
    return restTemplate.postForObject(ServiceName.aluno , aluno ,
    Aluno.class );
}

public Aluno update(Aluno aluno){
    HttpHeaders headers = new HttpHeaders();
    headers.setContentType(MediaType.APPLICATION_JSON);

    HttpEntity<Aluno> entity = new HttpEntity<Aluno>(aluno , headers);

    ResponseEntity<Aluno> response = restTemplate.exchange(
    ServiceName.aluno , HttpMethod.PUT, entity , Aluno.class );

    if(response.getStatusCode() == HttpStatus.OK)
        return response.getBody();
    return null;
}

public List<Aluno> getAll(){
    ResponseEntity<Aluno[]> response =
    restTemplate.getForEntity(ServiceName.aluno , Aluno[].class );
    return Arrays.asList(response.getBody());
}
}

```

## APÊNDICE 5: CÓDIGO FONTE DE SGF-WEB/ALUNOCONTROLLER.JAVA

```
package br.eb.ime.comp.pfc.sgf.web.controller;

import java.security.Principal;
import java.util.Comparator;
import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.security.authentication.
UsernamePasswordAuthenticationToken;

import br.eb.ime.comp.pfc.sgf.models.Aluno;
import br.eb.ime.comp.pfc.sgf.web.User;
import br.eb.ime.comp.pfc.sgf.web.service.AlunoService;

@Controller
@RequestMapping("/aluno")
public class AlunoController {

    @Autowired
    private AlunoService service;

    @RequestMapping(value = "/", method = RequestMethod.GET)
    public String index(Model model, Principal u){
        User user = new User((UsernamePasswordAuthenticationToken) u);
```

```

if (!user.isProfessor())
    return "redirect:/403";

List<Aluno> alunos = service.getAll();
//ordenar por numero
alunos.sort(new Comparator<Aluno>() {

    @Override
    public int compare(Aluno o1, Aluno o2) {
        return o1.getNumero().compareTo(o2.getNumero());
    }
});

model.addAttribute("title", "Alunos");
model.addAttribute("alunos", alunos);
model.addAttribute("user", user);
return "aluno/index";
}

@RequestMapping(value = "/numero/{numero}", method = RequestMethod.GET)
public String getByNumero(@PathVariable("numero") String numero,
Model model, Principal u){
    User user = new User((UsernamePasswordAuthenticationToken) u);

    if (user.isAluno()){
        if (!user.getName().equals(numero))
            return "redirect:/403";
    }

    Aluno aluno = service.getByNumero(numero);
    model.addAttribute("aluno", aluno);
    model.addAttribute("title", "Aluno_" + aluno.getNome());
}

```



```

        model.addAttribute("user", user);    return "aluno/aluno";
    }

@RequestMapping(value = "/todos", method = RequestMethod.GET)
public String all(Model model){
    return "redirect:/aluno";
}

@RequestMapping(value = "/new", method = RequestMethod.GET)
public String newAluno(Model model, Principal u){
    User user = new User((UsernamePasswordAuthenticationToken) u);

    if (!user.isCoordenador())
        return "redirect:/403";

    model.addAttribute("title", "Novo_Aluno");

    model.addAttribute("user", user);
    return "aluno/new";
}

@RequestMapping(value = "/new", method = RequestMethod.POST)
public String createAluno(@RequestParam("nome") String nome,
    @RequestParam("email") String email, @RequestParam("numero")
    String numero, @RequestParam("password") String password,
    Principal u){
    User user = new User((UsernamePasswordAuthenticationToken) u);

    if (!user.isCoordenador())
        return "redirect:/403";

    Aluno aluno = new Aluno(numero, nome, email, password);
    service.create(aluno);
    return "redirect:" + "/aluno/numero/" + numero;
}

```

```

}

@RequestMapping(value = "/edit/{numero}", method = RequestMethod.GET)
public String editAluno(@PathVariable("numero") String numero,
Model model, Principal u){
    User user = new User((UsernamePasswordAuthenticationToken) u);

    if(!user.isCoordenador())
        if(!(user.isAluno() && user.getName().equals(numero)))
            return "redirect:/403";

    Aluno aluno = service.getByNumero(numero);
    model.addAttribute("aluno", aluno);
    model.addAttribute("title", "Editar_Aluno");

    model.addAttribute("user", user);
    return "aluno/edit";
}

@RequestMapping(value = "/edit/{numero}", method = RequestMethod.POST)
public String saveAluno(@PathVariable("numero") String numero,
@RequestParam("nome") String nome, @RequestParam("email") String
email, @RequestParam("password") String password, Principal u){
    User user = new User((UsernamePasswordAuthenticationToken) u);

    if(!user.isCoordenador())
        if(!(user.isAluno() && user.getName().equals(numero)))
            return "redirect:/403";

    Aluno aluno = service.getByNumero(numero);
    aluno.setEmail(email);
    aluno.setNome(nome);
    if(user.isAluno() && !password.equals(""))
        aluno.setPassword(password);

```

```
    service.update(aluno);  
    return "redirect:" + "/aluno/numero/" + numero;  
}  
}
```

## APÊNDICE 6: CÓDIGO FONTE DE SGF-WEB/ATACONTROLLER.JAVA

```
package br.eb.ime.comp.pfc.sgf.web.controller;

import java.security.Principal;
import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RequestMethod;

import br.eb.ime.comp.pfc.sgf.models.Aluno;
import br.eb.ime.comp.pfc.sgf.models.Professor;
import br.eb.ime.comp.pfc.sgf.models.Tempo;
import br.eb.ime.comp.pfc.sgf.models.Disciplina;
import br.eb.ime.comp.pfc.sgf.models.Falta;
import br.eb.ime.comp.pfc.sgf.models.Turma;
import br.eb.ime.comp.pfc.sgf.models.Xerife;
import br.eb.ime.comp.pfc.sgf.models.Ata;
import br.eb.ime.comp.pfc.sgf.models.Coordenador;
import br.eb.ime.comp.pfc.sgf.web.Assinatura;
import br.eb.ime.comp.pfc.sgf.web.User;
import br.eb.ime.comp.pfc.sgf.web.Utills;
import br.eb.ime.comp.pfc.sgf.web.service.AlunoService;
import br.eb.ime.comp.pfc.sgf.web.service.ProfessorService;
import br.eb.ime.comp.pfc.sgf.web.service.DisciplinaService;
import br.eb.ime.comp.pfc.sgf.web.service.TurmaService;
```

```

import br.eb.ime.comp.pfc.sgf.web.service.AtaService;

@Controller
@RequestMapping("/ata")
public class AtaController {

    @Autowired
    private AlunoService alunoService;
    @Autowired
    private ProfessorService professorService;
    @Autowired
    private DisciplinaService disciplinaService;
    @Autowired
    private TurmaService turmaService;
    @Autowired
    private AtaService service;

    @RequestMapping(value = "/", method = RequestMethod.GET)
    public String index(Model model, Principal u) {
        User user = new User((UsernamePasswordAuthenticationToken) u);
        List<Ata> atas;

        if (user.isCoordenador()) {
            String email = user.getName();
            Professor coordenador = professorService.getByEmail(email);
            atas = service.getByCoordenador(coordenador);

            List<Ata> atasProfessor = service.getByProfessor(coordenador);
            for(Ata ata : atasProfessor){
                if(!atas.contains(ata))
                    atas.add(ata);
            }
        } else if (user.isAluno()) {
            String numero = user.getName();

```

```

        Aluno xerife = alunoService.getByNumero(numero);
        atas = service.getByXerife(xerife);
    } else {
        String email = user.getName();
        Professor professor = professorService.getByEmail(email);
        atas = service.getByProfessor(professor);
    }

    for (int i = 0; i < atas.size(); i++) {
        atas.get(i).setTurma(turmaService.getById(atas.get(i)
            .getIdTurma()));
    }

    for(int i = 0; i < atas.size(); i++){
        atas.get(i).getCoordenador().setAssinado(
            Utils.recebeuAssinaturaCoordenador(atas.get(i)
                .getCoordenador()));
    }

    model.addAttribute("title", "Atas");
    model.addAttribute("atas", atas);
    model.addAttribute("user", user);

    return "ata/index";
}

@RequestMapping(value = "/id/{id}", method = RequestMethod.GET)
public String getById(@PathVariable("id") String id, Model model,
    Principal u) {
    User user = new User((UsernamePasswordAuthenticationToken) u);

    Ata ata = service.getById(id);

    if (user.isProfessor() || ata.getXerife().getXerife().getNumero())

```

```

.equals(user.getName())) {
    for (int i = 0; i < ata.getTempos().size(); i++) {
        ata.getTempos().get(i).setSaved(Utils.
            recebuAssinaturaProfessor(ata.getTempos().get(i)));

        if (user.isProfessor()) {
            if(ata.getTempos().get(i).getDisciplina().getProfessor().
                getEmail().equals(user.getName())){
                ata.getTempos().get(i).setProfessorDoTempo(
                    !ata.getTempos().get(i).isSaved());
            }
        }
    }
}

if(Utils.recebuAssinaturaCoordenador(ata.getCoordenador()))
    ata.getCoordenador().setAssinado(true);

ata.getXerife().setAssinado(Utils.
    recebuAssinaturaXerife(ata.getXerife()));

model.addAttribute("ata", ata);
model.addAttribute("title", "Ata");
model.addAttribute("user", user);

return "ata/ata";
}
return "redirect:/403";
}

@RequestMapping(value = "/todos", method = RequestMethod.GET)
public String all(Model model) {
    return "redirect:/ata/";
}

```

```

@RequestMapping(value = "/new", method = RequestMethod.POST)
public String newAta(Model model, Principal u, @RequestParam("data")
String data) {
    User user = new User((UsernamePasswordAuthenticationToken) u);

    if (!user.isAluno())
        return "redirect:/403";

    String numero = user.getName();
    Aluno xerife = alunoService.getByNumero(numero);
    Turma turmaDoXerife = turmaService.getByAluno(xerife);

    if (turmaDoXerife == null)
        return "redirect:/403";

    Ata ata = new Ata();
    ata.setData(data);
    ata.setXerife(new Xerife(xerife));
    ata.setTurma(turmaDoXerife.getId());
    ata.setCoordenador(new Coordenador(
turmaService.getCoordenador(turmaDoXerife), null));
    ata.getTempos().add(new Tempo("1"));
    ata = service.create(ata);

    return "redirect:" + "/ata/edit/" + ata.getId();
}

@RequestMapping(value = "/edit/{id}", method = RequestMethod.GET)
public String editAta(@PathVariable("id") String id, Model model,
Principal u) {
    User user = new User((UsernamePasswordAuthenticationToken) u);

    if (!user.isAluno())
        return "redirect:/403";

```



```

Ata ata = service.getById(id);

if (!ata.getXerife().getXerife().getNumero().equals(user.getName()))
    return "redirect:/403";

if (Utils.recebeuAssinaturaXerife(ata.getXerife())) {
    model.addAttribute("title", "Ata_de_Faltas");
    model.addAttribute("ata", ata);
    model.addAttribute("user", user);
    return "ata/assinadaXerife";
}

Turma turmaDoXerife = turmaService.getById(ata.getIdTurma());

for (int i = 0; i < ata.getTempos().size(); i++) {
    for (Aluno aluno : turmaDoXerife.getAlunos()) {
        if (!ata.getTempos().get(i).getFaltas().contains(aluno)) {
            ata.getTempos().get(i).getAlunosAvulsos().add(aluno);
        }
    }
}

ata.getTempos().get(i).setSaved(true);
ata.getTempos().get(i).setDisciplinasAvulsas(
turmaDoXerife.getDisciplinas());
ata.getTempos().get(i).getDisciplinasAvulsas().remove(
ata.getTempos().get(i).getDisciplina());
}

model.addAttribute("ata", ata);
model.addAttribute("disciplinas", turmaDoXerife.getDisciplinas());
model.addAttribute("title", "Editar_Atade_Faltas");
model.addAttribute("user", user);
return "ata/edit";

```

```

}

@RequestMapping(value = "/edit/{id}", method = RequestMethod.POST)
public String saveAta(@PathVariable("id") String id,
@RequestParam("ordem") String ordem, Model model,
@RequestParam(value = "alunos", required = false, defaultValue = "")
String[] alunos, @RequestParam("disciplina") String idDisciplina,
Principal u) {
    User user = new User((UsernamePasswordAuthenticationToken) u);

    if (!user.isAluno())
        return "redirect:/403";

    Ata ata = service.getById(id);
    if (!ata.getXerife().getXerife().getNumero().equals(user.getName()))
        return "redirect:/403";

    if (Utils.recebeuAssinaturaXerife(ata.getXerife())) {
        model.addAttribute("title", "Ata_de_Faltas");
        model.addAttribute("ata", ata);
        model.addAttribute("user", user);
        return "ata/assinadaXerife";
    }

    Turma turmaDoXerife = turmaService.getById(ata.getIdTurma());
    Tempo tempo = new Tempo();
    tempo.setOrdem(ordem);

    for (String numero : alunos) {
        Aluno actual = alunoService.getByNumero(numero);
        if (actual == null)
            continue;
        tempo.getFaltas().add(actual);
        int i;
    }
}

```

```

    for (i = 0; i < ata.getFaltas().size(); i++) {
        if (ata.getFaltas().get(i).getAluno().equals(actual))
            break;
    }
    if (i >= ata.getFaltas().size()) {
        ata.getFaltas().add(new Faltas(actual, "Destino_Indeterminado"));
    }
}

```

```

Disciplina disciplina = disciplinaService.getById(idDisciplina);
tempo.setDisciplina(disciplina);

```

```

int count;
for (count = 0; count < ata.getTempos().size(); count++) {
    if (ata.getTempos().get(count).getOrdem().equals(ordem)) {
        ata.getTempos().set(count, tempo);
        break;
    }
}
if (count >= ata.getTempos().size()) {
    ata.getTempos().add(tempo);
}

```

```

service.update(ata);

```

```

for (int i = 0; i < ata.getTempos().size(); i++) {
    for (Aluno aluno : turmaDoXerife.getAlunos()) {
        if (!ata.getTempos().get(i).getFaltas().contains(aluno)) {
            ata.getTempos().get(i).getAlunosAvulsos().add(aluno);
        }
    }
}

```

```

ata.getTempos().get(i).setSaved(true);
ata.getTempos().get(i).setDisciplinasAvulsas(turmaDoXerife

```

```

        .getDisciplinas());
        ata.getTempos().get(i).getDisciplinasAvulsas().remove(ata
        .getTempos().get(i).getDisciplina());
    }

    Integer newOrdem = Integer.parseInt(ordem) + 1;
    ata.getTempos().add(new Tempo(newOrdem.toString(),
    turmaDoXerife.getDisciplinas(), turmaDoXerife.getAlunos()));

    model.addAttribute("ata", ata);
    model.addAttribute("user", user);
    model.addAttribute("title", "Editar_Atade_Faltas");
    return "/ata/edit";
}

@RequestMapping(value = "/justificar/{id}/{numero}", method =
RequestMethod.GET)
public String justificarAta(@PathVariable("id") String id,
@PathVariable("numero") String numero, Model model,
Principal u) {
    User user = new User((UsernamePasswordAuthenticationToken) u);
    if (!user.isAluno())
        return "redirect:/403";

    Ata ata = service.getById(id);
    Faltas faltas = null;
    for (Faltas f : ata.getFaltas()) {
        if (f.getAluno().getNumero().equals(numero)) {
            faltas = f;
            break;
        }
    }

    model.addAttribute("title", "Justificar_Faltas");
    model.addAttribute("ata", ata);
}

```

```

        model.addAttribute("falta", falta);
        model.addAttribute("user", user);
        return "/ata/justificar";
    }

@RequestMapping(value = "/justificar/{id}", method =
RequestMethod.POST)
public String salvarJustificativa(@PathVariable("id") String id,
@RequestMapping("numero") String numero, @RequestParam("motivo")
String motivo, Model model, Principal u) {
    User user = new User((UsernamePasswordAuthenticationToken) u);
    if (!user.isAluno())
        return "redirect:/403";

    Ata ata = service.getById(id);
    for (int i = 0; i < ata.getFaltas().size(); i++) {
        if (ata.getFaltas().get(i).getAluno().getNumero()
            .equals(numero)) {
            ata.getFaltas().get(i).setMotivo(motivo);
            break;
        }
    }
}

service.update(ata);
return "redirect:" + "/ata/edit/" + ata.getId();
}

@RequestMapping(value = "/assinar/aluno/{id}", method =
RequestMethod.GET)
public String alunoAssinar(@PathVariable("id") String id,
Principal u, Model model) {
    User user = new User((UsernamePasswordAuthenticationToken) u);
    if (!user.isAluno())
        return "redirect:/403";
}

```

```

Ata ata = service.getById(id);
if (!ata.getXerife().getXerife().getNumero().equals(user.getName()))
    return "redirect:/403";

Aluno xerife = alunoService.getByNumero(user.getName());
ata = Assinatura.assinar(xerife, ata);
if(ata == null){
    model.addAttribute("user", user);
    model.addAttribute("title", "Erro");
    return "/ata/erro";
}
service.update(ata);
return "redirect:" + "/ata/id/" + ata.getId();
}

```

```

@RequestMapping(value = "/assinar/coordenador/{id}",
method = RequestMethod.GET)
public String coordenadorAssinar(@PathVariable("id") String id,
Principal u, Model model) {
    User user = new User((UsernamePasswordAuthenticationToken) u);
    if (!user.isCoordenador())
        return "redirect:/403";
    Ata ata = service.getById(id);
    if (!ata.getCoordenador().getCoordenador().getEmail()
.equals(user.getName()))
        return "redirect:/403";

    Professor coordenador = professorService.getByEmail(user.getName());

    ata = Assinatura.assinarCoordenador(coordenador, ata);
    if(ata == null){
        model.addAttribute("user", user);
        model.addAttribute("title", "Erro");
    }
}

```

```

        return "/ata/erro";
    }
    service.update(ata);

    return "redirect:" + "/ata/id/" + ata.getId();
}

@RequestMapping(value = "/assinar/professor/{id}/tempo/{ordem}",
method = RequestMethod.GET)
public String professorAssinar(@PathVariable("id") String id,
@PathVariable("ordem") String ordem, Principal u, Model model) {
    User user = new User((UsernamePasswordAuthenticationToken) u);
    if (!user.isProfessor())
        return "redirect:/403";
    Ata ata = service.getById(id);
    Integer ind = Integer.parseInt(ordem) - 1;
    if (!ata.getTempos().get(ind).getDisciplina().getProfessor().getEmail().equals(user.getName()))
        return "redirect:/403";

    Professor professor = professorService.getByEmail(user.getName());
    ata = Assinatura.assinarProfessor(professor, ata, ind);
    if(ata == null){
        model.addAttribute("user", user);
        model.addAttribute("title", "Erro");
        return "/ata/erro";
    }

    service.update(ata);

    return "redirect:" + "/ata/id/" + ata.getId();
}
}

```

## APÊNDICE 7: PASSOS PARA MONTAR O SISTEMA EM MÁQUINAS VIRTUAIS

Primeiramente deve-se executar o *software* VirtualBox e selecionar criar uma nova máquina virtual. Indica-se a quantidade de memória necessária e avança para a criação de um disco virtual.

Criada a máquina virtual deve-se alterar as configurações de rede, indo na opção de configurações e selecionando a aba Rede. A opção utilizada foi a de placa em modo *Bridge* como na figura 9.1, que torna uma interface de rede da dispositivo hospedeiro como um da máquina virtual.

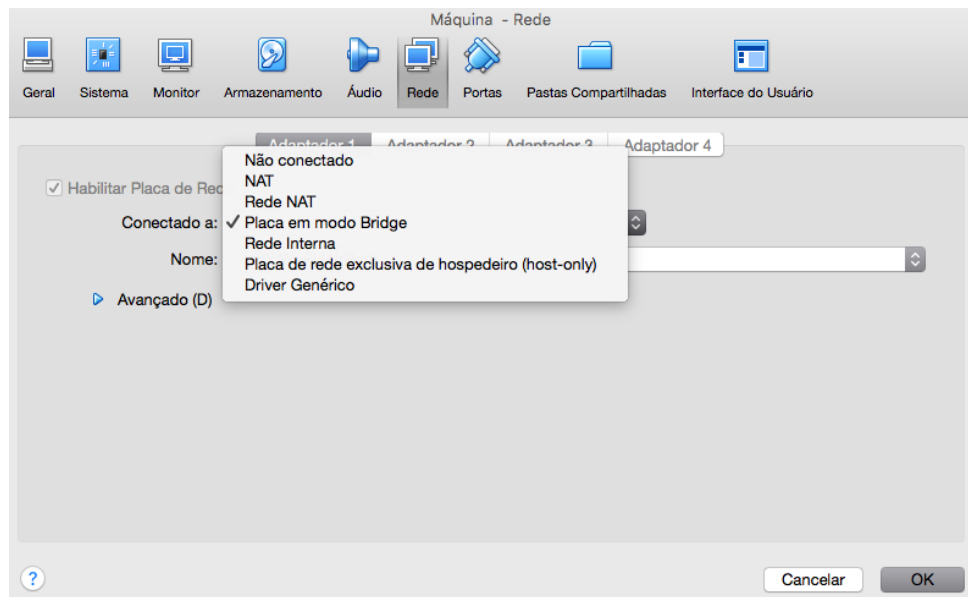


FIG. 9.1: Configuração de rede

Agora deve-se instalar o sistema operacional. Iniciando pela primeira vez a máquina virtual surge a opção de seleção de uma imagem de um sistema a ser instalado como na figura 9.2. Seleciona-se a localização da imagem e pode-se prosseguir com a instalação, que no caso foi do Ubuntu 16.04.1 Server.



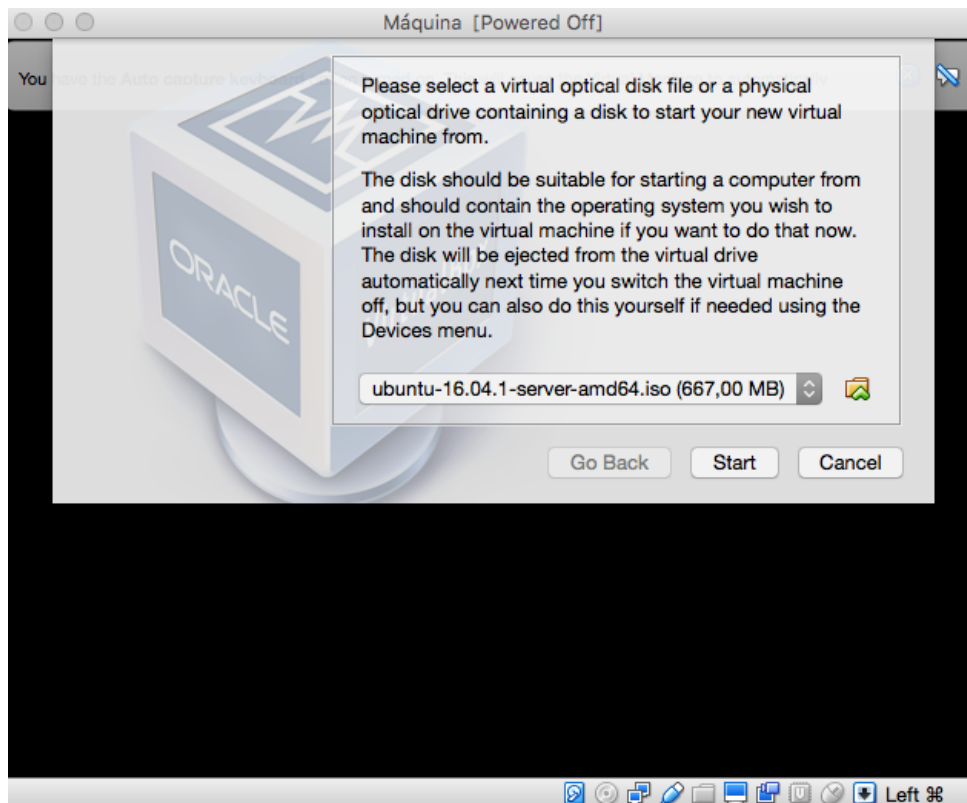


FIG. 9.2: Seleção da imagem do sistema a ser instalado

A inicialização leva a uma tela de seleção de idiomas e depois a um menu onde a opção por instalar o Ubuntu Server irá aparecer a qual deve ser selecionada. O assistente de instalação irá ser executado onde inicialmente será selecionado o tipo de teclado. Passa-se para a fase do particionamento do disco, como na figura 9.3, selecionando o método guiado levará a um assistente que particionará de forma automática o disco.

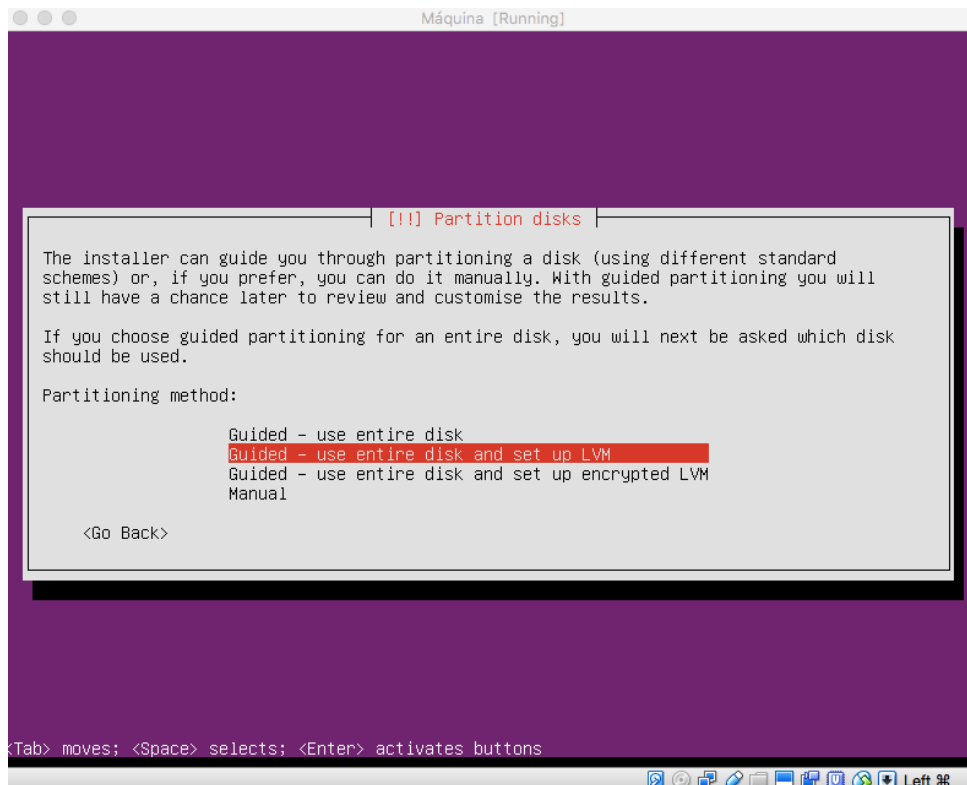
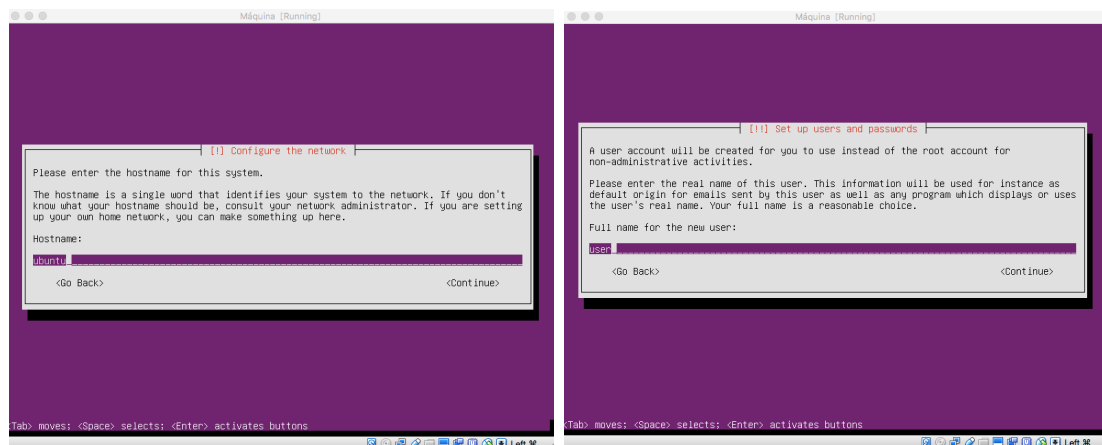


FIG. 9.3: Seleção da imagem do sistema a ser instalado

Após deve ser informado o nome de *host*, como na figura 9.4a e passará para a fase de criação de um usuário, como na figura 9.4b, e será perguntado se deseja criptografar a pasta pessoal do usuário, recomenda-se em não optar pela criptografia. Então o instalador tentará identificar automaticamente o fuso horário.



(a) Nome de *host*

(b) Criação de usuário

FIG. 9.4: Telas de instalação

Será pedido informações sobre o *proxie* da rede, informe se a rede a qual a máquina fará parte utilizar um. Após será perguntado qual será a política de atualização do sistema.

Em um ambiente de produção é recomendado não optar pelas atualizações automáticas, porque novas versões podem entrar em conflitos com sistemas em funcionamento, logo as atualizações devem ser feitas com cautela. Finalmente será perguntado os *softwares* a serem instalados, como na figura 9.5. É importante a seleção dos programas padrões do sistema e o OpenSSH é recomendado para o acesso remoto à máquina virtual.

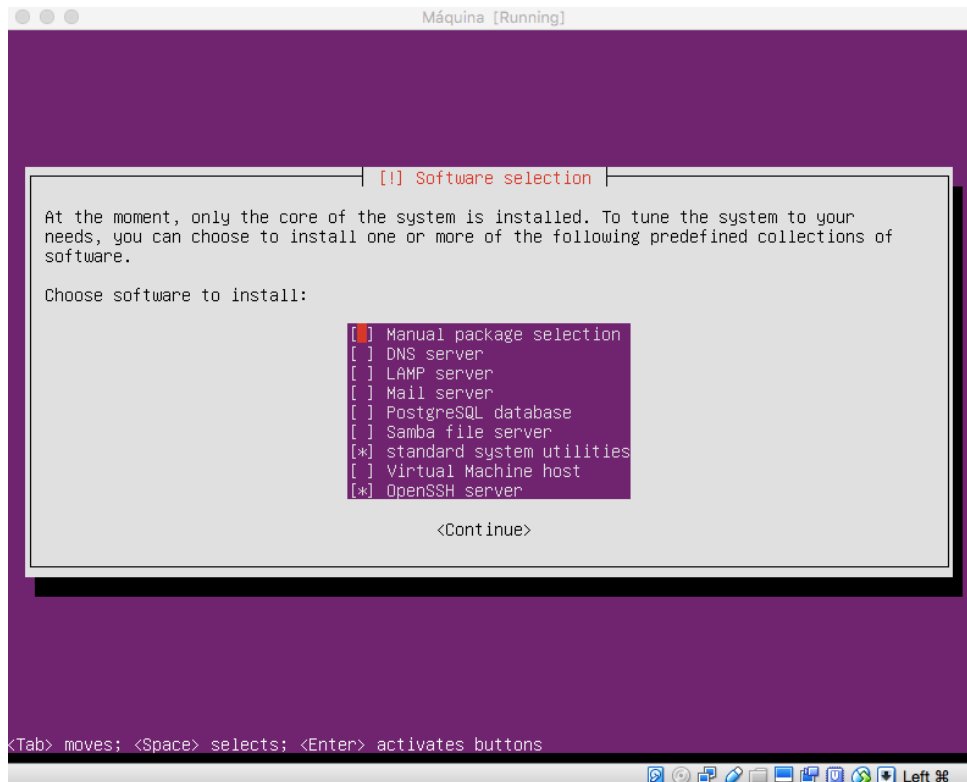


FIG. 9.5: Programas a serem instalados

Após a instalação dos programas será instalado o gerenciador de *boot*, onde será finalizada a instalação do sistema operacional. A máquina virtual será reiniciada podendo-se logar com o usuário criado na fase de instalação. Se a máquina for conter o MongoDB, esse é o momento da instalação com o comando `sudo apt-get install mongodb`.

Para se descobrir o IP da máquina pode-se utilizar o comando `ip addr` que irá informar as informações de todas as interfaces de rede da máquina visto na figura 9.6, que conterá a informação desejada.

```

user@ubuntu:~$ ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen 1000
    link/ether 08:00:27:ac:bf:30 brd ff:ff:ff:ff:ff:ff
    inet 192.168.0.30/24 brd 192.168.0.255 scope global enp0s3
        valid_lft forever preferred_lft forever
    inet6 fe80::a00:27ff:feac:bf30/64 scope link
        valid_lft forever preferred_lft forever

```

FIG. 9.6: Informações de rede

Para instalar o Java 8 da Oracle, deve-se usar o comando `sudo add-apt-repository ppa:webupd8team/java` para adicionar o repositório onde contém os instaladores do Java e o comando `sudo apt-get update` para atualizar o banco de dados dos programas do gerenciador de pacotes do Ubuntu. O comando `sudo apt-get install oracle-java8-installer` inicia a instalação do Java 8.

Os arquivos JAR do sistema SGF devem estar nas máquinas virtuais para o sistema ser iniciado. Uma forma de se passar os arquivos é com o comando `scp`, que é uma forma de se copiar arquivos via SSH. Para o funcionamento desse comando, o computador que se deseja copiar um arquivo deve ter um serviço de SSH rodando que seja acessível pela máquina virtual. A forma de se usar o comando é `scp user@ip-remote:local/file`, onde *user* é um usuário da máquina remota que possui permissão de leitura do arquivo desejado, *ip-remote* é o endereço da máquina que contém os arquivos JAR, *local* é o caminho onde estão os arquivos e *file* é o nome do JAR desejado. A figura 9.7 exemplifica o uso desse comando.

```

user@ubuntu:~$ scp user@192.168.0.10:Local/sgf-auxiliar.jar_

```

FIG. 9.7: Cópia de arquivos remoto

Com os JARs contendo os micros serviços pode-se inicializar o SGF. Cada micro serviço deve ser executado separadamente não importando a ordem de execução, contando que todos estejam sendo executados no final. Os micros serviços são iniciados via terminal com o comando `java -jar microservice.jar params`, onde *microservice.jar* é o arquivo JAR do micro serviço desejado e *params* são os parâmetros necessários para a inicialização. A figura 9.8 mostra a inicialização de um micro serviço.

```

user@ubuntu:~$ java -jar sgf-auxiliar.jar_

```

FIG. 9.8: Inicialização do micro serviço auxiliar