Projeto e Implementação de um Serviço Web RESTful com Técnicas de Segurança

Claudiomar Pereira de Araújo

CENTRO DE INFORMÁTICA
UNIVERSIDADE FEDERAL DA PARAÍBA

João Pessoa, 2018

Claudiomar Pereira de Araújo

Projeto e Implementação de um Serviço Web RESTful com Técnicas de Segurança

Monografia apresentada ao curso Ciência da Computação do Centro de Informática, da Universidade Federal da Paraíba, como requisito para a obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Raoni Kulesza

Novembro de 2018

Ficha catalográfica: elaborada pela biblioteca do CI.

Será impressa no verso da folha de rosto e não deverá ser contada.
<Se não houver biblioteca, deixar em branco>

CENTRO DE INFORMÁTICA
UNIVERSIDADE FEDERAL DA PARAÍBA

Trabalho de Conclusão de Curso de Ciência da Computação intitulado **Projeto e Implementação de um Serviço Web RESTful com Técnicas de Segurança** de autoria de Claudiomar Pereira de Araújo, aprovada pela banca examinadora constituída pelos seguintes professores:

_____

Prof. Dr. Raoni Kulesza
Universidade Federal da Paraíba

_____

Prof. Dr. Gustavo Henrique Matos Bezerra Motta
Universidade Federal da Paraíba

_____

Mestrando Lucas Oliveira Costa Aversari
Universidade Federal da Paraíba

João Pessoa, 12 de novembro de 2018

Centro de Informática, Universidade Federal da Paraíba
Rua dos Escoteiros, Mangabeira VII, João Pessoa, Paraíba, Brasil CEP: 58058-600
Fone: +55 (83) 3216 7093 / Fax: +55 (83) 3216 7117

RESUMO

Sistemas de *Software* tem sido alvo de ataques em diversas áreas e diferentes tipos de sistemas. Resultados de pesquisas mostram a importância da segurança na área de TI e da entrega de produtos a clientes que atendam a requisitos de segurança satisfatórios. Sistemas Web estão inclusos, pois conectam pessoas do mundo inteiro e fazem parte do nosso dia-a-dia. Não só especialistas de segurança, mas também desenvolvedores são responsáveis pelo uso de técnicas de segurança, adicionando essa prática ao ciclo de desenvolvimento de *software*. Nesse contexto, este trabalho apresenta, além de uma revisão do histórico de sistemas Web e conceitos de segurança, o projeto e implementação de um sistema Web RESTful real com técnicas de segurança para avaliar requisitos funcionais e não funcionais no lado do servidor, utilizando o Spring como o principal *framework* Web. Adicionalmente, o estudo apresenta uma avaliação do uso de controles proativos de segurança para o projeto Você Digital do Tribunal de Contas do Estado da Paraíba (TCE-PB).

Palavras-chave: sistemas Web, segurança de sistemas, Spring, TCE-PB.

ABSTRACT

Software systems have been target of attacks in several fields and different types of systems. Research results show the importance of security in IT field and the delivery of products to clients that attend to satisfactory security requirements. Web systems are included because they connect people all over the world and they are part of our daily life. Not only security specialists, but also developers are responsible for using security techniques, adding this practice to the software development lifecycle. In this context, this work presents, besides a review about Web systems history and security concepts, the project and implementation of a real RESTful Web system with security techniques to evaluate functional and non functional requirements on the server-side, using Spring as the main framework Web. Additionally, this study presents an evaluation of using security proactive controls on Você Digital project for Tribunal de Contas do Estado da Paraíba (TCE-PB).

Key-words: Web systems, systems security, Spring, TCE-PB.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS

HTML	–	Hypertext Markup Language
URL	–	Uniform Resource Locator
HTTP	–	Hypertext Transfer Protocol
DIP	–	Dependency Injection Principle
SGBD	–	Sistema Gerenciador de Banco de Dados
ACID	–	Atomicidade Consistência Isolamento Durabilidade
CRUD	–	Create Read Update Delete
CSS	–	Cascading Style Sheet
DOM	–	Document Object Model
CGI	–	Common Gateway Interface
URL	–	Uniform Resource Locator
SQL	–	Structured Query Language
NoSQL	–	Not Only SQL
SOA	–	Service-Oriented Architecture
AOP	–	spect Oriented Programming
DI	–	Dependency Injection
IOC	–	Inversion Of Control
POJO	–	Plain Old Java Object
MVC	–	Model View Controller
JSON	–	Javascript Object Notation
API	–	Application Programming Interface
REST	–	Representational State Transfer
RESTful	–	Adjetivo para substantivo que possui REST
JVM	–	Java Virtual Machine
JDBC	–	Java Database Connectivity
ORM	–	Object Relational Mapping
JPA	–	Java Persistence API
JMS	–	Java Message Service
TDD	–	Test Driven Development
XML	–	Extensible Markup Language
IDE	–	Integrated Development Environment
ADE	–	API Development Environment

RFC	–	Request For Comments
CID	–	Confidencidade Integridade Disponibilidade
CA	–	Certification Authority
PKI	–	Public Key Infrastructure
IETF	–	Internet Engineering Task Force
IPSec	–	Internet Protocol Security
SSL	–	Socket Security Layer
SET	–	Secure Eletronic Transaction
S/MIME	–	Secure/Multipurpose Internet Mail Extensions
TCP	–	Transmission Control Protocol
ASCII	–	American Standard Code for Information Interchange
HTTPS	–	Hypertext Transfer Protocol Security
TLS	–	Transport Layer Security
MAC	–	Message Authentication Code
OWASP	–	Open Web Application Security Project
LAViD	–	Laboratório de Vídeo e Aplicação Digital
UFPB	–	Universidade Federal da Paraíba
TCE-PB	–	Tribunal de Contas do Estado da Paraíba
POI	–	Point Of Interest
TI	–	Tecnologia da Informação

SUMÁRIO

1 INTRODUÇÃO

A necessidade de realizar comunicação para transferir informação alcançou o acesso à Internet. Foi por meio da Internet que surgiu a Rede Mundial de Computadores (*World Wide Web*), ou simplesmente *Web*. Ela afetou definitivamente o modo como lidamos com informação e como nos comunicamos; ela conecta pessoas do mundo inteiro, faz parte do nosso dia-a-dia e está presente em diversos setores da indústria, em organizações com ou sem fins lucrativos. Com a evolução do *hardware* e do s*oftware* em conjunto, temos eletrônicos de diversos tipos conectados à rede para lidar com informação.

Antes da Web, segundo Sommerville (2015), sistemas de *software* funcionavam apenas em computadores locais, não sendo possível acessá-los fora de uma organização. A Web se destacou entre os *softwares* por ser acessível em qualquer computador com um navegador compatível instalado. Sistemas Web dispensam instalação e atualização do sistema desenvolvido em cada computador, sendo necessário manter somente o servidor funcionando com o sistema em execução e esperando por conexões. Outra vantagem é que navegadores distribuídos por terceiros resolvem parte da complexidade de conectar-se ao servidor e exibir o programa. Inicialmente, o ramo era tão promissor que houve competição entre distribuidores de navegadores - fato que ficou marcado como a Guerra dos Navegadores, conforme Zalewski (2012). Os navegadores continuaram evoluindo com novas funcionalidades e promovendo também a evolução da Web.

Técnicas de engenharia de *software* têm colaborado para o desenvolvimento de sistemas Web. Atualmente, eles são projetados e desenvolvidos com uso de componentes de *software* preexistentes. Existem soluções prontas para uso ou facilmente extensíveis por aplicação servidor, persistência de dados e para interface de usuário, de acordo com Deitel (2012). Este trabalho aborda o conceito e reuso dessas tecnologias com técnicas de segurança para avaliar requisitos funcionais e não funcionais de um sistema Web. São considerados projeto e arquitetura de *software*, com ênfase no servidor e na persistência de dados.

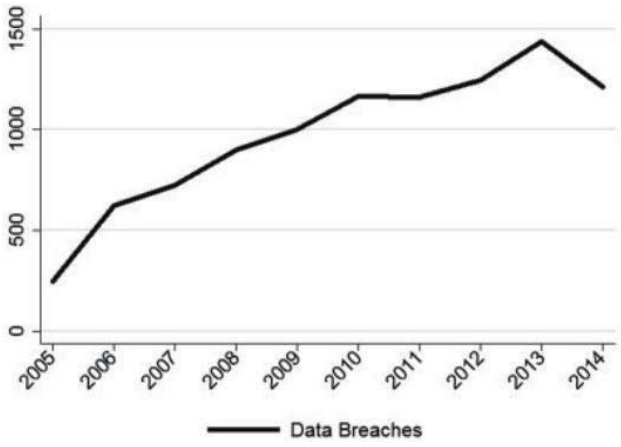
1.1 Tema

Software inseguro tem sido alvo de ataques nas áreas de finanças, saúde, energia e outros sistemas críticos, segundo OWASP (2017). Existem ações e dados de sistemas que uma vez comprometidos não existe mais retorno, podendo afetar definitivamente organizações e seus clientes. Conforme sistemas se tornam maiores, mais complexos e conectados a outros sistemas, alcançar níveis de segurança satisfatórios também se torna complexo.

Em 2013, o presidente dos Estados Unidos da América assinou uma ordem executiva para ajudar na segurança da infraestrutura da nação contra ataques cibernéticos (NIST, 2018). Sabendo disso, Romanosky (2016) pesquisou se existe incentivo para organizações aprimorarem suas práticas de segurança e reduzir riscos de ataques. Ele examinou a composição, riscos e incidentes de eventos cibernéticos unindo múltiplos conjuntos de dados divulgados publicamente. "Cibernética é a ciência que estuda como informação é comunicada em máquinas e dispositivos eletrônicos", (Cambridge, 2018). Para melhor análise, ele separou alguns tipos de eventos para as seguintes análises quantitativas.

A Figura 1 apresenta o número de vazamento de dados do ano de 2005 a 2014. Percebe-se que houve grande crescimento desde 2005, com tendência a redução a partir de 2013. Mas, ainda assim, 2014 teve aproximadamente quatro vezes mais eventos do que em 2005.

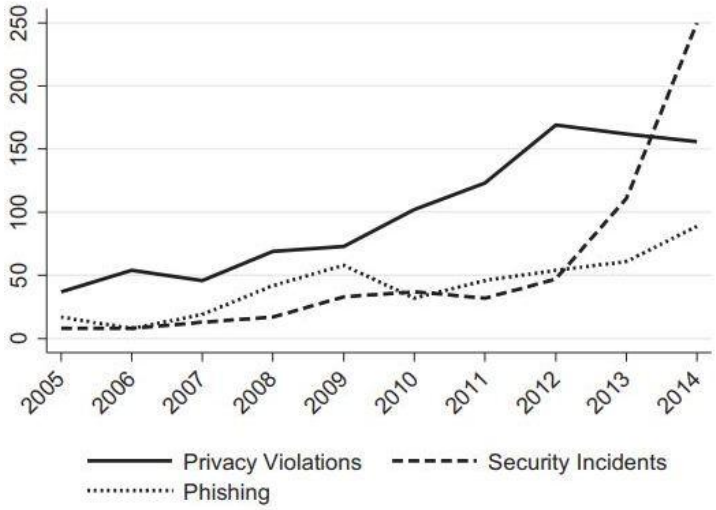
Figura 1: Vazamento de dados



Fonte: Romanosky (2016)

Outros três tipos de eventos são analisados: incidente de segurança, violação de privacidade e *phishing*. Um incidente de segurança envolve ou compromete um sistema corporativo de TI (a organização sofre a ação); violação de privacidade é a utilização indevida de informação pessoal (ação causada por uma organização), como obtenção não autorizada de informação pessoal ou envio de e-mails de spam; e *phishing* é quando indivíduos utilizam computadores ou eletrônicos diretamente contra outros indivíduos para obter informações pessoais. Na Figura 2, percebe-se que violação de privacidade tem reduzido de 2012 a 2014, mas ainda com um número três vezes maior que em 2005. Incidentes de *phishing* têm crescido levemente, enquanto incidentes de segurança tiveram aumentos significativos desde 2012.

Figura 2: Três tipos de eventos cibernéticos



Fonte: Romanosky (2016)

1.2 Problema

Em sistemas Web, existem questões de segurança que precisam ser tratadas tanto no lado do servidor quanto no lado do cliente. Segundo Groef (2016), há necessidade de desenvolvimento de tecnologias de segurança robustas ou contramedidas para aplicações Web. Entretanto, é preciso equilibrar redução de riscos com funcionalidades de aplicações para que elas possam alcançar seus objetivos sem restringi-las demais.

Requisitos de segurança podem ser atendidos de forma mais eficiente durante o projeto e implementação do *software*, sendo necessárias diretrizes de princípios de segurança. É importante notar que essa responsabilidade não é exclusiva para especialistas de segurança. Na verdade, desenvolvedores possuem grande responsabilidade na implementação dessas soluções, podendo aplicá-las diretamente no seu ciclo de desenvolvimento, conforme OWASP (2018). Porém, é preciso adquirir uma mentalidade de segurança para melhor percepção e compreensão, de acordo com Hibish et al (2016).

1.2.1 Objetivo geral

Utilizar técnicas de segurança para avaliar requisitos em sistemas Web do ponto de vista de projeto e implementação de software.

1.2.2 Objetivos específicos

- Pesquisar o histórico de sistemas Web para compreender sua evolução até a atualidade
- Estudar o *framework* Spring para utilizá-lo como principal tecnologia Web no desenvolvimento de um sistema
- Apresentar conceitos e a importância da segurança de sistemas computacionais, com foco em sistemas Web
- Projetar e implementar um sistema Web e avaliar seus requisitos com técnicas de segurança utilizando controles proativos

1.3 Estrutura

Esta monografia está estruturada em 5 capítulos. Este atual descreve a introdução, categorizado Capítulo 1. O Capítulo 2 apresenta os conceitos gerais e fundamentação teórica utilizada para produção deste trabalho. Nele, é apresentada uma visão geral da

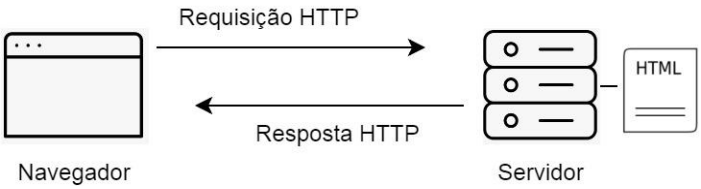
Web, uma introdução ao *framework* Spring, segurança de sistemas gerais e Web, e segurança para desenvolvedores. No Capítulo 3, é apresentado um estudo de caso de um projeto no qual sou responsável pela aplicação servidor e da modelagem de dados. No Capítulo 4 são apresentadas técnicas de segurança aplicadas ao servidor e ao acesso aos dados. Por fim, o Capítulo 5 apresenta os resultados obtidos neste trabalho e quais possíveis contribuições para trabalhos futuros ele oferece.

2 CONCEITOS GERAIS E REVISÃO DA LITERATURA

2.1 Histórico de Sistemas Web

A Rede Mundial de Computadores (*World Wide Web)*, ou *Web*, teve início com o objetivo de compartilhar informação através da computação. Isso tem sido feito através de documentos *hypertext* conectados por *hyperlink*, o que permite a navegação entre conteúdos que são disponibilizados através de páginas Web. Uma página é escrita em uma linguagem de marcação, normalmente *Hypertext Markup Language* (HTML), por ser padrão da Web. A página é interpretada e exibida por um navegador, que antes faz requisição ao servidor via *Hypertext Transfer Protocol* (HTTP). O servidor é identificado por um *Uniform Resource Locator* (URL), onde está hospedada a página desejada, conforme Deitel (2012). Essa é uma implementação do modelo cliente-servidor, um sistema distribuído a partir do qual a Web tem mudado o modo como lidamos com informação.

Figura 3: Navegador solicita página HTML ao servidor via requisição HTTP



Fonte: Autor (2018)

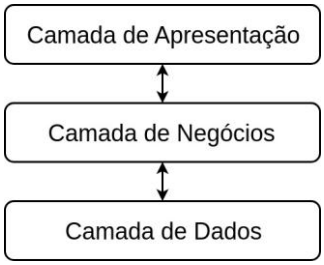
Inicialmente, a Web era mais simples, utilizava apenas conteúdo estático e o navegador só recebia informação. Passou a funcionar de modo dinâmico ao receber entrada de dados do usuário para ser processada no servidor e então realizar operações em bancos de dados. O usuário não mais apenas recebe informações, mas também é responsável por criá-las e organizá-las. Essa evolução aumentou a complexidade dos sistemas e exigiu soluções para melhor manipulação deles.

Uma solução que se tornou um padrão foi a arquitetura em camadas. Um padrão arquitetural pode ser visto como uma descrição abstrata de boas práticas que foram utilizadas e testadas em diferentes sistemas e plataformas com sucesso. Como o nome desse padrão sugere, ele separa o sistema em camadas com funcionalidades

relacionadas a cada uma delas e promove independência, concordando com Sommerville (2015). Ele permite separação física em hardware, ou apenas lógica em software. Para conectá-las, é uma boa prática fazer uso do *Dependency Inversion Principle* (DIP), o qual sugere que abstrações não devem depender de detalhes, e detalhes que devem depender de abstrações, conforme Hall (2017). Assim, as camadas podem ser modificadas ou até mesmo substituídas por outras sem causar danos na estrutura do projeto. Gamma et al (1996) apresentam alguns padrões estruturais que podem ser utilizados para praticar esse princípio na implementação, permitindo comunicação de modo desacoplado e facilitando a manipulação de sistemas separados em camadas.

Uma arquitetura comum na Web é a de três camadas, separadas em camada de apresentação, camada lógica de negócios e camada de dados, de acordo com Groef (2016, p. 14). A camada de apresentação fornece a interface do usuário onde são feitas entradas e saídas de dados. O usuário interage e envia requisições para serem processadas na camada de negócios, sem necessariamente ter conhecimento disso, podendo limitar-se ao uso do navegador e as funcionalidades oferecidas na página exibida. Na camada de dados, os dados modelados de acordo com as regras de negócios podem ser armazenados e recuperados. Eles são gerenciados através de processamento de arquivo tradicional ou bancos de dados, sendo a segunda opção a mais utilizada por alguns motivos como: possuir Sistema Gerenciador de Banco de Dados (SGBD) que permite restrição de acesso não autorizado, busca com processamento eficiente e por garantir a integridade dos dados através das propriedades de Atomicidade, Consistência, Isolamento e Durabilidade (ACID), de acordo com Elmasri e Navathe (2016). A camada de negócios faz o intermédio entre a camada de apresentação e a camada de dados. Ela recebe requisição do usuário, avalia e processa de acordo com as regras de negócios da aplicação. Isso envolve comunicação com a camada de apresentação para obter e exibir dados e enviar a página adequada ao usuário. Ela também faz comunicação com a camada de dados para realizar diversos tipos de operações de persistência como as famosas *Create*, *Read*, *Update* e *Delete* (CRUD*).* Essas três camadas são implementadas com tecnologias em particular que podem ser vistas como tecnologias *client-side* e *server-side*. A camada de apresentação é normalmente implementada com tecnologias *client-side,* e a camada de negócios e dados com tecnologias *server-side*.

Figura 4: Arquitetura Web três camadas



Fonte: Autor (2018)

No *client-side*, tem-se como principais linguagens HTML, *Cascading Style Sheet* (CSS) e *Javascript,* conforme Deitel (2012). CSS descreve o estilo de uma página HTML e como seus elementos devem ser exibidos com personalização. Javascript é a linguagem mais utilizada para atualizações dinâmicas, como tratamento de eventos, recepção de entrada de usuário e iteração de usabilidade através de manipulação do *Document Object Model* (DOM), que é uma *Application Programming Interface* (API) da Web com representação em forma de árvore. Javascript também permite enviar requisição da interface do usuário ao servidor sem carregar a página, o que valoriza a usabilidade do usuário. Por exemplo, um navegador pode enviar requisições com os métodos GET, POST, PUT e DELETE sem carregar a página; isso é feito com *Asynchronous Javascript And XML* (AJAX). Com a intenção de aprimorar a forma de desenvolvimento, melhorar o reuso de *software* e aumentar ainda mais a usabilidade do usuário, inclusive em dispositivos móveis, surgiram bibliotecas e *frameworks* que carregam todo o Javascript necessário no cliente para processamento dinâmico na primeira vez que a página é carregada. Isso significa que o DOM e regras de aplicação estão sendo implementadas no cliente, permitindo menor número de requisições no servidor e, consequentemente, possibilitando menor número de processamento. Atualmente, nessa categoria, estão em alta as tecnologias React e Angular, ambas baseadas em Javascript mas com suas particularidades.

No *server-side*, *Common Gateway Interface* (CGI) permitiu a criação de páginas Web dinâmicas, servindo como interface entre o navegador e programas

executados no servidor. A entrada do usuário é recebida, processada num desses programas e é gerada uma saída, conforme Fox e Hao (2018). Atualmente, é mais comum o uso de *frameworks* para desenvolvimento de sistemas Web, que dispensa implementação de *scripts* baseados em CGI. Os frameworks Web oferecem recursos para desenvolvimento da camada de negócios como a configuração de URL das páginas, envio e recebimento de dados entre o servidor e o cliente, configuração de segurança para autenticação, autorização e amenização de possíveis vulnerabilidades. Eles também permitem configuração para integração e acesso à banco de dados *Structured Query Language* (SQL) e *Not Only SQL* (NoSQL), inclusive através de frameworks de persistência que facilitam com operações prontas e possibilidade de personalização dos comandos de acesso. Algumas linguagens e *frameworks* para desenvolvimento Web são, respectivamente: Java e Spring, Javascript e Express, Ruby e Rails, PhP e Laravel, C# e ASP.NET, Python e Django.

Anteriormente, sistemas Web eram desenvolvidos para disponibilizar funcionalidades limitadas no navegador. Para resolver isso, foram desenvolvidos Web *services* que permitem definir interfaces para disponibilizar serviços para outros programas. Através dessas interfaces é possível determinar quais recursos podem ser acessados e alterados. *Web service* pode ser implementado com *Service Oriented Architecture* (SOA), que surgiu como uma abordagem para lidar com desafios de grandes sistemas monolíticos e para promover a reusabilidade de software, conforme Newman (2015). Múltiplos serviços podem ser disponibilizados como processos de diferentes sistemas operacionais e, consequentemente, podendo estar em diferentes máquinas. Com objetivos semelhantes, temos a arquitetura baseada em microserviços. Segundo Bonér (2016), um dos princípios de microserviços é Dividir e Conquistar: projetar um sistema a partir de subsistemas isolados e independentes que se comunicam através de protocolos leves, bem definidos e padronizados, para que cada um desses subsistemas faça uma coisa, e faça bem. Isso permite que cada serviço seja autônomo, ou seja, tenha sua própia camada de negócios e de dados, independentes dos outros subsistemas. Desse modo, microserviços facilitam manutenção e operações de *deploy e* promove escalabilidade. Colabora também na organização de equipes e suas responsabilidades, pois uma pessoa ou equipe pode ficar responsável por todas as etapas de um serviço, desde o desenvolvimento até o funcionamento dele em produção. Para seu desenvolvimento, pode-se utilizar tecnologias distintas, como

linguagens e bancos de dados, porque são serviços que funcionam de forma independente disponibilizados através de suas interfaces.

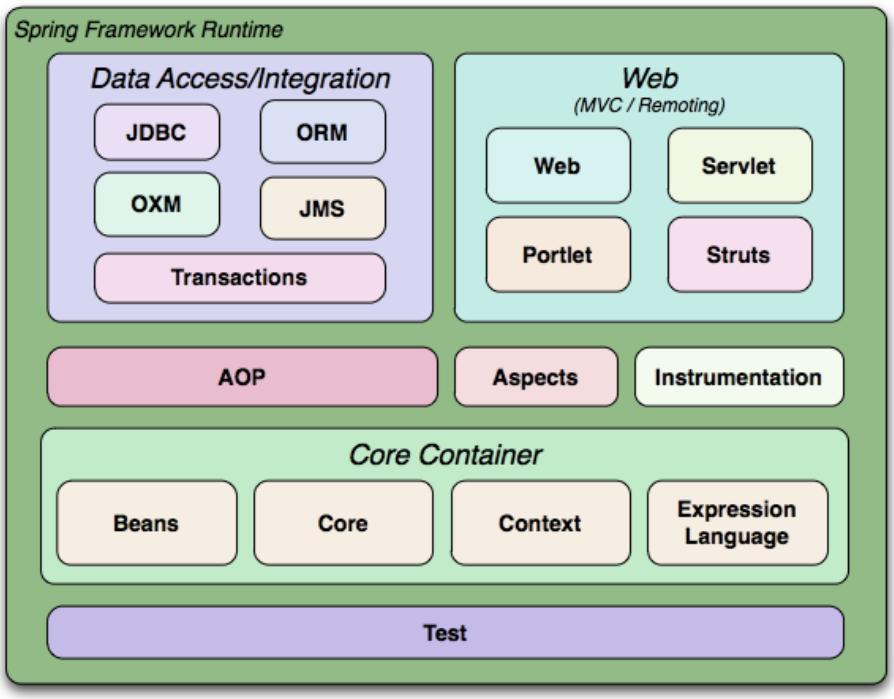
Nessa breve história e evolução da Web é possível perceber a importância da engenharia de software no desenvolvimento desses sistemas. A engenharia de *software* trata de todos os aspectos da produção de *software*, desde a especificação até a manutenção do sistema. Ela inclui atividades de desenvolvimento, aplicação e gerenciamento de teorias, métodos e ferramentas para apoiar o desenvolvimento de *software* quando apropriado, concordando com Sommerville (2015). Tudo isso tem sido aplicado ao desenvolvimento de sistemas Web como pode-se perceber nos exemplos anteriores. O reuso de *software* é a abordagem dominante para construção de sistemas Web. Quando os construímos, pensamos como montá-los a partir de elementos de software preexistentes, como APIs, *frameworks* e *Web servers*. Fazer o reuso deles é válido porque podem ser soluções muito utilizadas e testadas, que facilitam e agilizam todo o processo de construção e manutenção de sistemas. Isso ajuda atender a grande exigência do mercado em curto prazo e a adaptação dos métodos ágeis nas organizações para seguir o princípio de satisfazer o cliente através de entrega rápida e contínua de software valioso, de acordo com Rasmusson (2010).

A engenharia de *software* certamente garante vantagens, mas quando se desenvolve para reuso, existem repetições de código e o número de linhas de código é significativamente maior. E quando se desenvolve com reuso, aparecem os *boilerplates*: são repetições de códigos incluídos com pouca ou nenhuma alteração para serem usados como um todo ou personalizados, de acordo com Lämmel e Peyton (2003). Isso inclui importação de módulos ou bibliotecas no programa, métodos *set* e *get* para atributos de linguagens orientadas a objetos e configurações de *frameworks*. Com os frameworks, isso acontece frequentemente, pois dão suporte ao reuso de projeto e de classes específicas. São coleções de classes abstratas e concretas que são adaptadas e estendidas, como um esqueleto. As abstrações são implementadas com recursos como herança em classes de objetos que assumem o comportamento do *framework* e estendem suas funcionalidades. É quando os *boilerplates* de *frameworks* tendem a aparecer, inclusive para configuração personalizada do comportamento dos objetos. Apesar de repetitivos, eles ajudam muito o desenvolvimento e continuarão sendo abordados neste trabalho.

2.2 Framework Spring

Spring fornece modelos de programação e configuração para aplicações Java EE modernas em qualquer tipo de plataforma. Seu principal foco é infraestrutura para o nível de aplicação, permitindo as equipes se concentrarem nas regras de negócios. Ele é dividido em módulos, oferecendo suporte para desenvolvimento Web, *Aspect Oriented Programming* (AOP), processamento de dados, transações e integração com outras tecnologias; possui *Dependency Injection* (DI) por meio do seu *container* e suporte a testes de funcionalidades implementadas com seus módulos.

Figura 5 - Módulos do *framework* Spring



Fonte: Pivotal Software (2018)

2.2.1 Spring Core e Dependency Injection

O núcleo do Spring é um *container* que cria e gerencia *beans* automaticamente. *Beans* são instancias de objetos pertencentes ao *container Inversion of Control* (IoC). IoC é um princípio da engenharia de software que pode ser utilizado em conjunto com *Dependency Injection* (DI). A instância de um objeto é controlada pelo *container* sendo "injetada" em outro objeto que possui dependência com o primeiro. Isso

permite conexão desacoplada, aumenta a modularidade do programa e o torna mais extensível, como demonstra Hall (2017). Um cenário de uso real é quando o acesso a instância de um objeto é necessário em diferentes partes do programa ou quando um objeto acessa conexão entre camadas. Para realizar a injeção, ele procura por *beans* no contexto da aplicação por meio de *component scanning* e satisfaz suas dependências com *autowiring*. *Beans* possuem escopo padrão *singleton* por *container*, o que significa que é criada apenas uma instância por *container*, podendo inclusive ser configurado com outras opções de escopo. Essa implementação é um pouco diferente do padrão definido por Gamma et al (1996), onde o escopo é feito por classe. Uma *bean* deve ser definida em um contexto de configuração, e seu *Plain Old Java Object* (POJO) permanece o mesmo, sem invasão do framework.

```
class Feijao {}

class FeijaoPreto extends Feijao {}

class Refeicao {

    Feijao feijao;

    Refeicao(Feijao feijao) {
        this.feijao = feijao;
    }
}

@Configuration
public class RefeicaoConfig {

    @Bean
    Refeicao prepararRefeicao() {
        return new Refeicao(new FeijaoPreto());
    }
}
```

Nesse exemplo, com fins didáticos e sem qualquer lógica de negócios real, fica mais claro como funciona a injeção de dependência. *@Configuration* indica que a classe *RefeicaoConfig* declara *beans* através de métodos para serem processadas pelo *container* do Spring e disponibilizadas em tempo de execução; ela prepara sua refeição. Para dar ênfase no desacoplamento de tipos, foi utilizada herança para o contexto deste exemplo mas, para maior desacoplamento, poderia ser interface. Assim, *Refeicao* conhece apenas *Feijao*, sem conhecer a especialidade dele que, nesse caso, é *FeijaoPreto*. Poderia ser *FeijaoBranco*, outra especialidade de *Feijao* - depende do objetivo. E a classe que faz uso recebe a instância injetada de *Refeicao*.

```
@Component
public class MinhaRefeicao {

    Refeicao refeicao;

    @Autowired
    public MinhaRefeicao(Refeicao refeicao) {
        this.refeicao = refeicao;
    }
}
```

A classe *MinhaRefeicao* é anotada com *@Component*, indicando que ela será detectada e inicializada através de *component scanning*, que será executado automaticamente ao iniciar uma aplicação Spring Boot. Isso permite o uso de injeção com *@Autowired*, sendo aqui uma injeção de construtor, onde é obtida uma instância de *Refeicao* criada pelo método *prepararRefeicao* na classe *RefeicaoConfig*. Assim, Spring permite desacoplamento entre objetos no desenvolvimento de aplicação e sua lógica de negócios. Inclusive entre seus próprios módulos e a aplicação desenvolvida, como pôde ser visto na definição de *bean* e componente, e como será apresentado mais à frente.

2.2.2 Spring Aspect Oriented Programming

Spring também oferece desacoplamento por meio de AOP, de acordo com Wall (2015), que permite acesso a funcionalidades já desenvolvidas no seu programa por meio de componentes reutilizáveis. Ele permite definição dos seus próprios aspectos, bem como utilizar seus aspectos existentes, como no seu serviço de segurança Spring Security, que protege o sistema que o implementa realizando autorização antes de acessar suas funcionalidades. Pode ser utilizado também para gerenciamento de transações e *logging*.

2.2.3 Spring Web

Nosso principal objeto aqui é o desenvolvimento Web com o módulo Spring Web MVC, sendo *Movel-View-Controller*, de acordo com Sommerville (2015), uma

abordagem reconhecidamente utilizada para construção de sistemas, onde a visão do cliente da aplicação é desacoplada da sua lógica de negócios. Com esse módulo, podemos definir métodos que recebem requisições HTTP e devolvem respostas em JSON através de API RESTful ou uma página HTML. Faremos uso também do módulo de acesso a dados Spring Data.

Spring roda na *Java Virtual Machine* (JVM) e possui duas arquiteturas de pilha: a tradicional síncrona e a mais recente nos últimos poucos anos, assíncrona. A arquitetura síncrona funciona com entrada e saída (E/S) bloqueante e atende uma requisição por *thread*. Ela é construída sobre a API *Servlet*, conhecida como *Servlet Stack,* dando origem ao módulo *Spring Web MVC,* que está presente no seu *framework* desde o início. A arquitetura assíncrona foi construída para aproveitar os processadores *multicore,* funcionar com E/S não bloqueante e atender muitas requisições concorrentes. Ela é conhecida por *Reactive Stack* e, em Spring, seu módulo é o *Spring WebFlux*. A arquitetura a ser utilizada depende do caso de uso, sendo ela uma decisão de projeto.

2.2.4 Spring Data

Persistência de dados tende a gerar código *boilerplate* para criar a conexão ao banco de dados, realizar consulta, processar o resultado e finalizar a conexão, além de *queries* repetidas para cada tipo de consulta e tabela modelada. Spring Data abstrai o código *boilerplate* necessário para procedimentos como esses e ainda oferece métodos prontos para persistência quando implementados em conjunto com seus objetos. Isso evita erros lógicos de implementação de *query,* fechamento de recursos e tratamento de erros. Ele tem suporte a *Java Database Connectivity* (JDBC) e *Object Relational Mapping* (ORM) com Hibernate e *Java Persistence API* (JPA). Funciona com bancos SQL e NoSQL. Ainda, possui suporte à transações com abstração para *Java Message Service* (JMS) para integração assíncrona com outras aplicações através de mensagens e faz uso de AOP para gerenciamento de transações.

2.2.5 Spring Test

Um processo fundamental na construção de sistemas é o *Test-Driven Development* (TDD), conforme Langr (2015), onde requisitos de software são tratados de forma específica. Testes podem ser feitos em funcionalidades pequenas e até em funcionalidades integradas, onde são tratadas as partes do sistema em conjunto. Spring permite testes com JUnit, TestNG e com seu próprio módulo de teste. Por exemplo, requisições HTTP, configuração de segurança e persistência de dados. Ele disponibiliza objetos *mock* configuráveis*,* que são implementações de abstrações de contexto e ambiente de execução, inclusive em conjunto com injeção de dependência.

2.2.6 Configuração Inicial Spring Boot

Spring Boot é o ponto de partida para aplicações Spring. Ele foi projetado para facilitar a criação de aplicações prontas para produção onde você pode rapidamente iniciar o desenvolvimento com pouca configuração, incluindo integração com outras tecnologias. Spring faz uso de um modelo de programação baseado em anotações nas classes Java, permitindo substituição da programação baseada em XML, para o qual também possui suporte. Spring Boot levou isso para um outro nível dando suporte à autoconfiguração para códigos *boilerplates* de configuração e desenvolvimento.

A partir daqui, segue um fluxo de desenvolvimento com Spring Boot para conhecê-lo melhor, na prática. Uma interface para criação do projeto inicial é disponibilizada como uma página Web em *https://start.spring.io/*. Através dela, é possível escolher a ferramenta para *build* e gerenciamento de dependências, a linguagem de desenvolvimento, o tipo de *packaging*, a versão do Spring Boot e as dependências necessárias. Aqui, será utilizado projeto Maven, linguagem Java com *packaging* Jar, que é o mais moderno, Java 8, Spring Boot 2.0.4 e as dependências Web, Spring MVC e arquitetura *Servlet Stack*, JPA (Java Persistence API*),* banco de dados H2 e Thymeleaf. Por padrão, Spring utiliza o *servlet Apache Tomcat* embutido, mas pode ser configurado para outros *containers* compatíveis com a versão *servlet* mínima exigida. Após *download* do projeto, no diretório principal está o arquivo *pom.xml* com toda a configuração do projeto Maven. Em

src/main/java/nome/do/pacote existe uma classe que inicia a aplicação Spring Boot no famoso método *main* Java com o método estático *run* da classe *SpringApplication*. Ela também possui uma anotação *@SpringBootApplication que* habilita autoconfiguração e busca por todas as *Spring beans* existentes no projeto ao ser executado.

Antes de começar o desenvolvimento, é bom executar o projeto para garantir que está tudo funcionando. Pode-se fazer isso a partir de um *Integrated Development Environment* (IDE) com suporte ao gerenciador de projeto Apache Maven ou diretamente com o Maven a partir de um terminal. A partir de um IDE é simples construir e executar o projeto com sua interface. No terminal, pode-se utilizar os arquivos *mvnw*, escrito em *shell script*, para sistemas Unix e *mvnw.cmd,* escrito em *batch*, para Windows. Basta executar um dos seguintes comandos: *./mvnw spring-boot:run* ou *mvnw.cmd spring-boot:run*. Também é possível gerenciar o projeto diretamente pelo comando *mvn*, com Maven configurado no sistema. Ao inserir o comando de execução, todas as classes são compiladas e os arquivos *.class* resultantes são armazenados no diretório *target* criado automaticamente.

2.2.7 Serviço Web RESTful com Spring MVC

É hora de começar o desenvolvimento. Será utilizado o módulo Spring MVC para mapear requisições com anotações. Até aqui, tem-se um projeto criado pelo inicializador do Spring, que vem pronto para execução. O endereço padrão de execução do servidor é *localhost:8080*, e pode ser acessado diretamente no navegador ou com outro cliente HTTP. A seguir, cria-se uma API com *Representational State Transfer* (REST) para simplesmente retornar uma mensagem. Ela pode ser visualizada em *localhost:8080/mensagens*.

```
@RestController
@RequestMapping("/mensagens")
class ControladorMensagem {

    @GetMapping
    String encontrarMensagem() {
        return "Opa!";
    }
}
```

Pode-se ter ideia do que está acontecendo apenas pelo código e sem conhecer Spring. O resultado é um serviço REST que mapeia requisições HTTP do tipo GET no caminho */mensagens*. Para entender como isso funciona, é necessário conhecer os estereótipos *@Componente* e *@Controller* fornecidos pelo Spring. Anotações de estereótipos declaram componentes que serão identificados e registrados como *beans* gerenciadas pelo *container* IoC e iniciadas junto com a aplicação. *@Component* é um tipo genérico para componentes. *@Controller* é uma especialização de *Component* utilizada para representar controladores Web que recebem requisições. Ela costuma ser usada com *@RequestMapping* para mapear requisições em nível de classe ou de método. Em nível de método, normalmente utiliza-se especializações para identificar métodos HTTP como *@GetMapping* e *@PostMapping*. Por fim, *@RestController* é uma especialização de *Controller* com adição de *@ResponseBody,* que escreve diretamente no corpo da resposta HTTP.

Classes com papel de controlador e seus métodos que recebem requisição são normalmente públicos, mas foram omitidos aqui. Agora, será retornado um objeto Java no formato *Javascript Object Notation* (JSON). Para isso, será simulado um sistema de avaliação. Define-se uma classe Avaliacao com métodos *gets* e um construtor para seus atributos, com algumas omissões aqui.

```
class Avaliacao {

    int classificacao;
    String comentario;
}
```

O controlador agora retorna um objeto *Avaliacao* no formato JSON, que requer obrigatoriamente métodos *get* para os atributos da classe.

```
@RestController
@RequestMapping("/avalicoes")
class ControladorAvaliacao {

    @GetMapping
    Avaliacao encontrarAvaliacao() {
        return new Avaliacao(5, "Ótimo");
    }
}
```

Um modo comum de acessar recursos em *Web services* é através de identificadores. Para isso, adiciona-se um parâmetro na API. Mas antes, é necessário incluir o novo campo *id* na classe *Avaliacao* e seu método *get*. O recurso pode ser

acessado em *http://localhost:8080/avaliacoes/100*, onde 100 é um argumento para o ID do objeto procurado. Variáveis *Uniform Resource Identifier* (URI) podem ser declaradas entre {} e acessadas com *@PathVariable* no parâmetro do método. Elas são convertidas automaticamente com suporte padrão a tipos como *long*, *double* e *String*.

```
@GetMapping("/{id}")
Avaliacao encontrarAvaliacao(@PathVariable long id) {
    return new Avaliacao(id, 5, "Ótimo");
}
```

Até aqui, tem-se uma API que permite consulta, mas receber dados também é necessário. Para receber requisições POST, basta utilizar *@PostMapping* e exigir um corpo na requisição com *@RequestBody* seguido do tipo do objeto esperado no parâmetro do método. Além disso, a classe *Avaliacao* precisa de um construtor padrão para instância do objeto. Construtor padrão não possui parâmetros e Java define um de forma implícita para cada classe. Se for definido algum outro, o padrão é sobrescrito e deve ser declarado de forma explícita, se desejado.

```
@PostMapping
void criarAvaliacao(@RequestBody Avaliacao avaliacao) {

}
```

Para os métodos PUT ou PATCH, pode-se usar dois parâmetros: um identificador com *PathVariable* e *RequestBody* para o conteúdo que pretende-se atualizar. Para o método DELETE, basta receber um identificador com *PathVariable* para procurar o recurso alvo. Abaixo segue um exemplo JSON para a classe *Avaliacao*, enviado no corpo da requisição de métodos que exigem um, como o método POST anterior. Assim, tem-se uma API RESTful que permite comunicação entre cliente e servidor com transmissão de dados no formato JSON.

```
{
    "id": 100,
    "classificacao": 5,
    "comentario": "Ótimo!"
}
```

Vale destacar que podemos utilizar o Spring MVC Test para testar a API ou um cliente HTTP como a aplicação Postman, um *API Development Environment* (ADE) gratuito.

2.2.8 Persistência de Dados com Spring Data JPA

Spring Data reduz significativamente a quantidade de código *boilerplate* necessária para implementar a camada de acesso a dados. O acesso da camada de dados será implementado com Spring Data JPA. Ele funciona com uma das mais famosas implementações da JPA: Hibernate. Spring Boot fornece o arquivo *application.properties* para configuração do projeto em *src/main/resources*. Por conveniência, pode-se usar um arquivo YAML, pois ele possui um formato de configuração em hierarquia. Assim, criamos o arquivo *application.yml* no mesmo local e dispensamos o outro. O banco de dados utilizado será o H2, um banco relacional em memória que funciona apenas durante a execução da aplicação. Spring Boot fornece integração embutida para ele e não exige instalação, tornando-o conveniente para testes e para este tutorial. Para utilizar outro banco relacional, inclusive para persistência em memória não volátil, como MySQL, seria necessário apenas incluir seu conector em *pom.xml* e a configuração adequada em *application.yml*; o código Java da entidade permaneceria o mesmo!

Para acessar o banco de dados H2, podemos usar a configuração padrão do Spring Boot. Aqui também será habilitada a visualização das *queries* realizadas pela JPA. Isso é feito adicionando-se o trecho de código abaixo no arquivo *application.yml.*

```
spring:
  h2:
    console:
      enabled: true
  jpa:
    show-sql: true
```

A aplicação pode ser executada e o painel H2 visualizado em *localhost:8080/h2-console* em um navegador. São utilizados os mesmos dados no painel conforme Figura 3. Ao testar a conexão deve ser visualizada uma mensagem de sucesso.

Figura 6 - Painel de acesso ao banco H2 em navegador *Web* com Spring

Fonte: Autor (2018)

Após configuração, vamos ao desenvolvimento. JPA faz uso de anotações pertencentes ao pacote *javax.persistence* para mapear objetos para tabelas relacionais (ORM). A classe *Avaliacao* é atualizada conforme abaixo, e o restante dela pode permanecer o mesmo. *@Entity* indica que *Avaliacao* é uma entidade JPA mapeada para uma tabela relacional com todos seus atributos como colunas. *@Id* especifica a chave primária da tabela e *@GeneratedValue* indica que o ID deve ser gerado automaticamente. Um construtor padrão é necessário porque JPA exige. A tabela e suas colunas podem ter seus nomes personalizados com as anotações *@Table(name = "nometabela")* no topo da classe e *@Column(name = "nomecoluna")* nos atributos. Caso não sejam especificados, são utilizados os mesmos nomes da classe e dos atributos. Existem outras personalizações normalmente utilizadas, como validação de dados dos atributos das classes com anotações contidas no pacote *javax.validation.constraints*.

```
@Entity
class Avaliacao {
```

```
@Entity
class Avaliacao {

    @Id
    @GeneratedValue
    private long id;

    public Avaliacao() {}
}
```

Alguns tipos de operações em bancos de dados são comuns em muitos sistemas, como as operações CRUD, por exemplo. A características mais brilhante do Spring Data é a capacidade de criar repositórios de forma automática. Repositórios do Spring são interfaces que podem ser definidas para acessar dados. Eles disponibilizam métodos comuns para persistência e podem criar *queries* a partir do nome de métodos personalizados em tempo de execução. Para *queries* mais complexas, pode-se também defini-las utilizando *Java Persistence Query Language* (JPQL) ou SQL nativa com *@Query* acima do método. Segue abaixo a classe que representa um repositório para a classe *Avaliacao*.

```
@Repository
interface RepositorioAvaliacao extends
CrudRepository<Avaliacao, Long> {

}
```

É só isso. Já é possível realizar operações no banco. *@Repository* indica que uma classe possui papel *Data Access Object* (DAO). *CrudRepository*, como o nome sugere, fornece funcionalidades CRUD para a entidade especificada no tipo genérico esperado, seguida pelo tipo da sua chave primária. Agora, o repositório será acessado no *ControladorAvaliacao* para aprimorar a API.

```
class ControladorAvaliacao {

    RepositorioAvaliacao repositorioAvaliacao;

    @Autowired
    ControladorAvaliacao(RepositorioAvaliacao r) {
        this.repositorioAvaliacao = r;
    }
}
```

É definido um atributo do tipo *RepositorioAvaliacao* que é iniciado no consturtor da classe. *@Autowired* solicita a instância de uma *bean* criada anteriormente, nesse caso, do *RepositorioAvaliacao*. O restante do código anterior da classe pode permanecer o mesmo. O *Autowired* pode ser removido do construtor e Spring vai reconhecer a solicitação da *bean* do mesmo modo. O principal objetivo de utilizá-la aqui foi mostrar o seu funcionamento. Com acesso à instância do repositório, serão utilizados os métodos de criar e encontrar avaliação.

```
@PostMapping
void criarAvaliacao(@RequestBody Avaliacao avaliacao) {
    repositorioAvaliacao.save(avaliacao);
}


 @GetMapping("/{id}")
 Avaliacao encontrarAvaliacao(@PathVariable long id) {
     return repositorioAvaliacao.findById(id)
     .orElse(null);
 }
```

Aproveitando ainda mais do que Spring Data pode oferecer, será criada uma *query* a partir do nome de um método personalizado com palavras-chave fornecidas pelo Spring. Em *RepositorioAvaliacao*, é adicionado o método abaixo para encontrar todas as avaliações por valor de classificação. *find* indica o inicio de uma busca; *Classificacao* corresponde ao atributo da classe *Avaliacao*; e a palavra-chave *Equals* é utilizada para comparação; Em seguida, compare com a SQL nativa que tem a mesma semântica. Nela, *classificacao* corresponde à uma coluna da tabela de *Avaliacao* e *valor* corresponde ao parâmetro *int classificacao* do método *findByClassificacaoEquals*.

```
List<Avaliacao> findByClassificacaoEquals(int classificacao);
```

```
"select * from avaliacao where classificacao = valor;"
```

A nova busca é adicionada no *ControladorAvaliacao* com o método abaixo. Ele exige um parâmetro para filtrar avaliações de acordo com o valor de classificação. *@RequestParam* é utilizada para parâmetros de busca ou dados de formulário. Essa funcionalidade pode ser acessada através de uma requisição GET em *localhost:8080/avaliacoes?classificacao=5*.

```
@GetMapping(params = "classificacao")
Iterable<Avaliacao> encontrarPorClassificacao(
@RequestParam int classificacao) {

    return repositorioAvaliacao
    .findByClassificacaoEquals(classificacao);
}
```

É possível também buscar avaliações por palavras contidas na *string* comentário. O método abaixo é adicionado em *RepositorioAvaliacao*. Nele, as palavras-chave utilizadas são: *Containing* para encontrar avaliações que possuam uma *substring* no seu comentário e *IgnoreCase* para não diferenciar letras maiúsculas e minúsculas.

```
List<Avaliacao> findByComentarioContainingIgnoreCase(
String comentario);
```

Com isso, tem-se um RESTful *Web Service* integrado com persistência de dados. As *queries* realizadas podem ser acompanhadas no *console* e seus dados verificados no banco através do painel *Web* do H2 em *localhost:8080/h2-console*.

2.2.9 Lógica de Negócios como Serviço

Spring disponibiliza também um estereótipo *@Service* que é especialização de *@Component*. Ele é baseado no *Domain-Driven Design*, proposto por Evans (2003), onde podem ser oferecidas operações isoladas através de interfaces. Semelhantemente, pode também indicar uma fachada para a lógica de negócios. *@Service* é um estereótipo de propósito geral e cabe ao desenvolvedor definir bem a sua semântica. Nesse projeto, podemos abstrair o acesso aos dados como regra de negócios. Nossos controladores não vão conhecer o repositório. E, caso exista algum processamento antes ou depois da consulta no repositório, pode ser realizado no componente de serviço e nossos controladores - ou outras classes clientes - não precisam implementar nem conhecer seus detalhes.

Duas classes de serviço são adicionadas conforme abaixo, sendo obtida uma instância do repositório na implementação do serviço através de inicialização no construtor, como anteriormente. No *ControladorAvaliacao*, é obitida uma instância de *ServicoAvaliacaoImpl* através da interface *ServicoAvaliacao*. Spring reconhece

automaticamente a classe que implementa a interface pelo tipo de dado, desde que ela tenha anotação de componente, nesse caso, com *@Service*. Todos os acessos anteriores ao repositório apresentados aqui também podem ser substituídos por acessos ao serviço.

```
interface ServicoAvaliacao {

    Optional<Avaliacao> encontrarPorId(long id);
}


@Service
class ServicoAvaliacaoImpl implements ServicoAvaliacao {

    RepositorioAvaliacao repositorioAvaliacao;

    ServicoAvaliacaoImpl(RepositorioAvaliacao r) {
        this.repositorioAvaliacao = r;
    }

    @Override
    public Optional<Avaliacao> encontrarPorId(long id) {
        return repositorioAvaliacao.findById(id);
    }
}


class ControladorAvaliacao {

   ServicoAvaliacao servicoAvaliacao;

   ControladorAvaliacao(ServicoAvaliacao sa) {
       this.servicoAvaliacao = sa;
   }
}
```

2.2.10 Página Web com Spring MVC

Spring permite a criação de páginas Web no *server-side* com diferentes tecnologias. Uma delas é Thymeleaf, um modelo Java que funciona com HTML5 e possui autoconfiguração com Spring Boot. Será criada uma simples página Web para demonstrar basicamente sua funcionalidade. Com configuração padrão, os arquivos são procurados em *src/main/resources/templates*. Lá, é criado um arquivo *inicial.html* e incluído o código abaixo. Nele, é feita a importação do Thymeleaf, o que nos permite utilizar a palavra-chave *th* para acessar dados recebidos do *server-side* como

tipos primitivos ou não primitivos, como *mensagem*, definida entre ${} para indicar uma variável.

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<h1 th:text="${mensagem}"></h1>
</html>
```

Agora, é preciso tratar requisições para esta página. A classe *ControladorInicial.java* é criada em *src/main/java/nome/do/pacote* com o código abaixo. *@Controller* indica que essa classe é um componente que trata requisições, de forma semelhante ao que vimos antes, mas não implementa *@ResponseBody* como em *@RestController*. O método *inicial* recebe um *Model* e retorna uma *view*, nesse caso, *inicial.html*. Através do *model*, podemos mapear dados entre controlador e visão com chave-valor.

```
@Controller
@RequestMapping("/inicial")
class ControladorInicial {

    @GetMapping
    String inicial(Model model) {
        model.addAttribute("mensagem", "Avaliações");
        return "inicial";
    }
}
```

Requisições e seus parâmetros podem ser tratadas com anotações como visto antes em *RestController*. Para exibir dados de um objeto, os trechos de código abaixo são inseridos nos arquivos HTML e Java. Para mais funcionalidades, a documentação Thymeleaf está disponível em *https://www.thymeleaf.org/*.

```
<div th:object="${avaliacao}">
    <td th:text="${avaliacao.classificacao}"></td>
    <td th:text="${avaliacao.comentario}"></td>
</div>
```

```
@GetMapping
String inicial(Model model) {
    Avaliacao avaliacao = new Avaliacao(5, "Muito bom");
    model.addAttribute("avaliacao", avaliacao);
    return "inicial";
}
```

2.2.11 Conclusão e Direcionamento

Observe que esse tutorial sobre Spring é descrito com uma abordagem bem direta para ajudar o leitor a compreender o framework de forma rápida e simples. Porém, por esse motivo, existem boas práticas que não foram adotadas aqui. Exemplos são: modificadores de acesso a classes e atributos de classes, bem como a palavra-chave *final* em instância de *beans* e boas práticas para desenvolvimento de API RESTful. Testes unitários e de integração não foram abordados, mas se pretende-se desenvolver com Spring, é essencial conhecer seu módulo de testes e adicionar essa prática ao seu ciclo de desenvolvimento. Inclusive, Spring REST Docs gera a documentação automática da sua API após testes unitários. Outro módulo de extrema importância para a maioria das aplicações dinâmicas é o Spring Security, que protege sua aplicação com autenticação e autorização de modo extensível.

Por fim, o código fonte desse tutorial está disponível no Github em *https://github.com/claudiomarpda/spring-boot-tutorial*. Spring disponibiliza uma rica documentação em sua página oficial *https://spring.io/*. Lá, pode-se conhecer melhor os projetos fornecidos. E quando pretende-se colocar o sistema em produção, é importante ler a documentação do Spring Boot sobre boas práticas para isso.

2.3 Segurança de Sistemas

2.3.1 Conceitos Básicos

Preocupamo-nos com segurança quando temos algo de valor e existe alguma possibilidade disso estar em risco. Adiante, são definidas algumas terminologias para este trabalho conforme RFC 4949, Internet Security Glossary (2007).

- **Risco:** A expectativa de perda pela possibilidade de um ataque causar algum resultado indesejado;
- **Ataque:** A utilização de uma ameaça para violar a segurança de um sistema;

- **Ameaça:** Uma possível ação que pode explorar uma vulnerabilidade e causar dano;
- **Vulnerabilidade:** Um erro ou uma fraqueza na implementação de algum projeto de sistema que pode ser explorada para violar sua segurança.

Se não existe vulnerabilidade, também não há possibilidade de ameaça, ataque nem risco. Mas sistemas computacionais são complexos para provar que fazem somente o que devem fazer - e nada mais - através de requisitos formais de segurança para sistemas, de acordo com Zalewski (2012).

2.3.2 Propriedades de Segurança

Sistemas computacionais são projetados para manipulação de informação, desde um bit à conjuntos de dados dos quais uma organização depende. De acordo com United States Code (2011), "segurança da informação" significa proteger informações e sistemas de informação das seguintes atividades não autorizadas: acesso, uso, divulgação, interrupção ou modificação, de modo a fornecer a tríade Confidencialidade, Integridade e Disponibilidade (CID).

- **Confidencialidade:** Restrição autorizada sobre acesso e divulgação de informação, incluindo meios para proteger privacidade pessoal e informação proprietária. "Uma perda de confidencialidade é a divulgação não autorizada de informação" (FIPS, 2004, p. 2);
- **Integridade:** Proteção contra modificação ou destruição inapropriada de informação, e inclui não repúdio de informação e autenticidade. "Uma perda de integridade é a modificação ou destruição não autorizada de informação" (FIPS, 2004, p. 2);
- **Disponibilidade:** Acesso e uso de informação confiável de modo oportuno. "Uma perda de disponibilidade é a interrupção de acesso ou uso de informação ou de um sistema de informação" (FIPS, 2004, p. 2);

O conceito de integridade mencionado anteriormente faz uso de dois temos que são detalhados abaixo:

- **Autenticidade:** “propriedade que uma entidade é o que diz ser.” (ISO/IEC 27000, 2018, p. 2);
- **Não repúdio:** “habilidade de provar a ocorrência de um evento reivindicado ou ação e suas entidades originarias.” (ISO/IEC 27000, 2018, p. 6).

Como sistemas não são completamente seguros, uma estratégia de defesa é rastrear possíveis brechas de segurança. Exemplos são armazenamento de registros ou exibição de dados analíticos em tempo real. Assim, para maior controle sobre sistemas, Stallings e Brown (2015) descrevem também essa estratégia para termos uma visão mais completa sobre segurança:

- **Determinação de responsabilidade (*accountability)*:** As ações de uma entidade devem ser rastreadas e atribuídas unicamente à ela. Os sistemas devem manter registros de suas atividades para análise. Isso oferece suporte a não repúdio, detecção e prevenção de intrusões e recuperação pós ação.

2.3.3 Criptografia

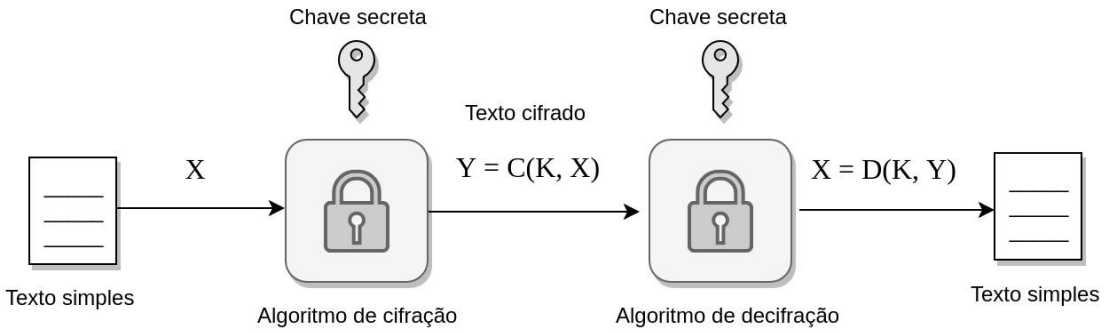
Um modo de proteger dados confidenciais é transformá-los em algo incompreensível. Algoritmos de cifração são frequentemente utilizados em sistemas que pretendem oferecer segurança, pois são fundamentais para fazer esse tipo de transformação. Eles fazem um processo de conversão de texto simples em uma forma incompreensível para acesso não autorizado. Mesmo que o texto seja acessado, sua semântica é protegida. São utilizados para armazenamento ou transferência de dados de forma segura. Com acesso permitido, é possível fazer o processo inverso com o texto cifrado e obter o texto simples com o algoritmo de decifração correspondente. A criptografia é a área que estuda métodos para cifração.

A criptografia simétrica ou criptografia de chave única têm sido amplamente usada para comunicação secreta, afirma Stallings (2017), e possui cinco componentes:

- **Texto simples:** O dado original e inteligível, utilizado como entrada para o algoritmo de cifração;
- **Algoritmo de cifração:** Realiza transformações no texto simples;
- **Chave secreta:** Utilizada como entrada para o algoritmo de cifração. As transformações exatas realizadas pelo algoritmo dependem da chave;
- **Texto cifrado:** A mensagem ininteligível produzida como saída. Para um mesmo texto simples, duas chaves produzem dois textos cifrados diferentes;
- **Algoritmo de decifração:** Recebe o texto cifrado e a chave correspondente para produzir o texto simples de volta.

Ela faz uso da criptografia de chave secreta, que usa a mesma chave para cifrar e decifrar o texto. Isso significa que numa troca de mensagem entre dois pontos, o remetente e o destinatário devem usar a mesma chave. A conversão para texto cifrado e a volta para texto simples são feitas matematicamente e são garantidas pelos algoritmos.

Figura 7 - Processo de criptografia simétrica



Fonte: Stallings (2017)

Outro tipo é a criptografia assimétrica ou de chave pública, a qual faz uso de uma chave pública e uma privada. Segundo Stallings (2017), ela possui os mesmos cinco componentes da criptografia simétrica, com adição da chave pública. Na assimétrica, porém, a chave secreta será chamada de chave privada para melhor compreensão. A chave pública pode ser conhecida por outras pessoas e é usada em

combinação com a privada. Para enviar uma mensagem secreta, a chave pública é usada para cifração e a chave privada é usada para decifração. Por exemplo, Alice pode usar a chave pública de Bob para cifrar uma mensagem, e somente Bob poderá decifrá-la, porque só ele possui a chave privada correspondente. Nota que esse esquema favorece a confidencialidade da mensagem. A criptografia de chave pública também pode ser usada para assinar e verificar a autenticidade de uma mensagem. A mensagem é assinada com a chave privada e pode ser verificada com a chave pública. Por exemplo, considerando que somente Alice pode fazer uma assinatura como a sua, ela pode enviar uma mensagem assinada com sua chave privada, e qualquer pessoa com sua chave pública pode decifrar a mensagem. Como a mensagem é decifrada com sua chave pública, logo sabe-se que foi ela que assinou, o que favorece a verificação da autenticidade da mensagem. Este método é conhecido como assinatura digital. O par de chave pública e privada são combinadas matematicamente. Se uma das chaves é utilizada para cifrar uma mensagem, somente a chave pareada correspondente pode decifrá-la.

É comum protocolos de segurança fazerem uso da criptografia simétrica e assimétrica em conjunto. Primeiramente, pode-se usar o cifrador assimétrico para autenticar ambos os lados envolvidos na comunicação. Depois, definir uma chave de cifração secreta para ambos a utilizarem no processo de cifrar e decifrar os dados com um cifrador simétrico, favorecendo a confidencialidade entre eles. Ainda, pode-se usar um cifrador assimétrico para assinar o dado cifrado digitalmente, garantindo sua autenticidade e aumentando a segurança. Um sistema de cifração deve ser seguro mesmo se o atacante souber de todos os detalhes do sistema, com exceção da chave secreta. Isso permite que o algoritmo seja livre e possa ser aprimorado pela comunidade. O ideal é que se utilize somente cifradores amplamente conhecidos e que tenham sido estudados há anos, tornando-se padrões estabelecidos com alto índice de confiabilidade. É possível alcançar tal confiança porque esses cifradores conseguem garantir ser computacionalmente inviável descobrir a chave secreta através do algoritmo de cifração e em posse da chave pública. Mas a chave pública ainda precisa de mais cuidados.

Uma pessoa pode enviar sua chave pública para outra pessoa ou disponibilizá-la em um repositório de acesso público, o que é comum. Como a chave pública é de livre acesso à qualquer pessoa, existe uma vulnerabilidade. Um atacante pode afirmar,

falsamente, ser dono de uma chave pública e enviar a sua própria. Seu objetivo, aqui, é ter acesso à dados restritos ao verdadeiro dono da chave pública. A solução para este problema é o certificado de chave pública, conforme Stallings e Brown (2015). Um certificado possui três componentes principais: a chave pública de um usuário, seu *ID*, e um bloco assinado por uma terceira entidade confiável. Além disso, pode conter informação sobre a terceira entidade e um período de validade para o certificado. Geralmente, a terceira unidade é uma autoridade certificadora (*Certification Authority* - CA), confiável por todos seus usuários. É ela que assina um certificado e garante a veracidade de uma chave pública pertencente a um usuário.

Para aquisição de chaves públicas de modo seguro e eficiente, é definida a infraestrutura de chave pública (*Public Key Infrastructure* - PKI) na RFC 4949, Internet Security Glossary (2007). PKI é um sistema de autoridade certificadora que realiza gerenciamento de certificados, arquivos, chaves e *tokens* para usuários de uma aplicação de criptografia assimétrica. As principais funções da PKI são: registrar usuários e seus certificados de chave pública, revogar certificados quando requerido e arquivar dados necessários para validar certificados futuramente. O grupo *Public Key Infrastructure X.509* (PKIX) da *Internet Engineering Task Force* (IETF) tem sido responsável por definir uma arquitetura e um conjunto de protocolos para fornecer serviços para a Internet baseados em X.509, com o objetivo de promover interoperabilidade entre diferentes implementações que usam esse certificado. Segundo Stallings (2017), X.509 é um modelo de serviço de diretório para conter informações de usuários que pode servir como um repositório de certificados de chaves públicas. É um padrão importante porque sua estrutura de certificado e protocolos de autenticação são utilizados em importantes aplicações de segurança de redes como *IP Security* (IPSec), *Socket Security Layer* (SSL), *Secure Eletronic Transaction* (SET) e *Secure/Multipurpose Internet Mail Extensions* (S/MIME).

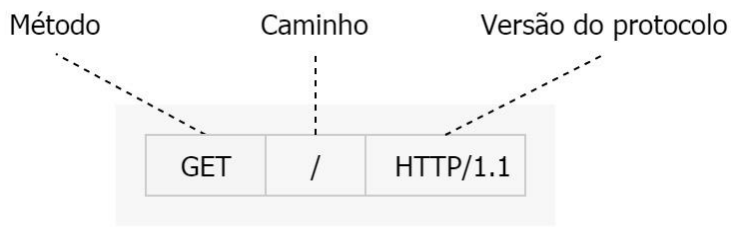
2.4 Protocolo de Comunicação

O principal meio de comunicação da Web é o *Hypertext Transfer Protocol* (HTTP). Ele segue o modelo cliente-servidor e funciona na camada de aplicação sobre o *Transmission Control Protocol* (TCP). O cliente inicia uma requisição através de um URL, que deve localizar um recurso de forma única em um servidor remoto e

espera uma resposta. No HTTP/1.1 e versões anteriores, as mensagens de requisição e resposta são enviadas no formato *American Standard Code for Information Interchange* (ASCII), feitas para serem simples e compreensíveis por humanos. No HTTP/2, segundo Mozilla (2018), essas mensagens são transmitidas em uma nova estrutura binária, permitindo otimizações, mas suas semânticas permanecem as mesmas. Os conteúdos são identificados pelo tipo *Multipurpose Internet Mail Extension* (MIME), que identifica a classe e o formato de um arquivo. Entre a requisição e a resposta existem outras entidades chamadas de *proxies*, as quais realizam diferentes operações e funcionam como *gateway, cache* ou filtro, por exemplo. HTTP é um protocolo sem estado, o que significa que cada requisição é uma nova conexão, com exceção para os casos de otimização, onde a conexão é mantida por algum tempo.

O fluxo de comunicação HTTP acontece do seguinte modo quando um cliente quer se conectar ao servidor: abre uma conexão TCP, envia uma mensagem HTTP, lê a resposta enviada pelo servidor e fecha ou reusa a conexão para requisições futuras. Além de solicitar uma página da Web, HTTP permite a escolha de métodos em cada requisição. Um método indica uma ação a ser realizada em um determinado recurso. O nome de um método é *case sensitive* e deve ser em letras maiúsculas. De acordo com Mozilla (2018), seguem alguns dos métodos HTTP mais utilizados. O método GET solicita uma representação de um determinado recurso e deve ser usado somente para obter dados; é através dele que um navegador obtém uma página para exibi-la, sendo o método mais utilizado. O método POST envia dados ao servidor, normalmente para criação. O tipo do corpo da requisição, isto é, o tipo de mídia a ser enviado, é indicado pelo cabeçalho *Content-Type*, que contém um tipo MIME. O método PUT substitui todas as representações do recurso alvo com o corpo da requisição. Ele é ideal para enviar sucessivas cargas de dados ao servidor, isto é, para atualização do recurso alvo. O método DELETE faz o que seu nome propõe ao recurso alvo. Utilizando o método GET como exemplo, a sintaxe de um comando de requisição, sem cabeçalhos, tem forma conforme Figura 5.

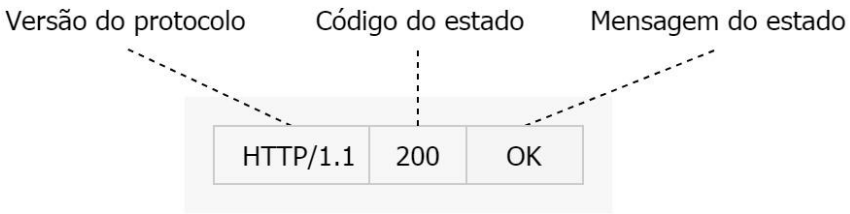
Figura 8 - Exemplo de comando para requisição HTTP sem cabeçalho



Fonte: Mozilla (2018)

Uma requisição possui inicialmente um método que indica seu propósito; um caminho pertencente ao atual contexto da conexão TCP, onde o contexto pode ser o endereço inicial de uma página; uma versão do protocolo; cabeçalhos opcionais que carregam informações adicionais ao servidor; e um corpo contendo recursos para métodos como POST e PUT. Uma resposta possui a versão do protocolo; um código de estado que informa o resultado da requisição; uma curta mensagem de estado relativa ao código de estado; cabeçalhos HTTP, como os de requisição; e um corpo opcional contendo o recurso solicitado ou uma resposta personalizada. Segue abaixo, na Figura 6, um exemplo de uma resposta HTTP contendo a linha de estado, mas sem cabeçalhos e corpo.

Figura 9 - Exemplo de linha de estado de uma resposta HTTP



Fonte: Mozilla (2018)

A linha de estado contém um código de estado formado por três dígitos, que divide as respostas em cinco grandes grupos, Conforme Taenebaum e Wetherall (2011). Os códigos 1xx servem apenas para informação e dificilmente são usados na prática. Os códigos 2xx significam que a requisição foi bem sucedida e que seu conteúdo está sendo retornado. Os códigos 3xx informam para procurar o conteúdo em outro lugar utilizando outro URL. Os códigos 4xx significam que a requisição falhou devido a falhas do cliente, como página inexistente. E os códigos 5xx

significam que houve um problema interno no servidor, que pode ter origem no código fonte da aplicação.

Tabela 1 - Grupos de códigos de respostas HTTP

Código	Significado	Exemplo
1xx	Informação	100 - requisição correta e o cliente pode continuar
2xx	Sucesso	200 - requisição realizada com sucesso
3xx	Redirecionamento	301 - recurso movido de localização
4xx	Erro no cliente	404 - recurso não encontrado
5xx	Erro no servidor	500 - erro interno do servidor

Fonte: Tanenbaum (2011)

HTTP foi desenvolvido para ser extensível, permitindo evolução com mais controle e adição de funcionalidades na Web. Seguem adiante algumas funcionalidades de acordo com Mozilla (2018). HTTP permite o controle de *cache* de documentos. O servidor pode informar a *proxies* e clientes do que fazer *cache* e por quanto tempo. HTTP também permite autenticação de usuários com cabeçalhos ou sessões com *cookies*. Embora HTTP seja um protocolo sem estado, ele permite criar sessões com uso de *cookies*. Um c*ookie* é formado por pequenos dados que o servidor envia ao navegador do usuário para que numa próxima requisição ele seja reconhecido com seus estados anteriores. Outro exemplo de utilização de *cookie* é um carrinho de compras de *e-commerce*. Por questões de segurança, navegadores forçam a separação entre sites, permitindo somente páginas de mesma origem acessarem informações de outra. Cabeçalhos HTTP podem relaxar essa restrição, permitindo compartilhamento de informação entre páginas. Assim, o HTTP tem mostrado ser capaz de evoluir com as necessidades da Web.

Para proteger a comunicação na Web entre cliente e servidor, é utilizado HTTP sobre *Transport Layer Security* (TLS), conhecido como HTTPS. TLS é um serviço de propósito geral implementado como um conjunto de protocolos que funcionam com TCP, segundo Stallings (2017). Um dos protocolos que fazem parte do TLS é o *Hand Shake Protocol*, responsável por criar uma sessão entre o cliente e o

servidor de forma segura antes de que qualquer dado seja trocado entre eles. Para estabelecer uma sessão, eles combinam a comunicação através de parâmetros que incluem algoritmo de cifração, chaves de cifração simétrica do cliente e do servidor e certificado de chave pública para autenticação das partes envolvidas. Ele também define uma chave compartilhada e secreta para formar um código de autenticação de mensagem ou *Message Authentication Code* (MAC), que será utilizada pelo destinatário para verificar se houve alteração no dado que foi inicialmente enviado. A comunicação é protegida pelo *TLS Record Protocol*, que fragmenta os dados em blocos manuseáveis, opcionalmente os comprime, aplica o código MAC, criptografa os blocos, adiciona um cabeçalho e envia o resultado pelo TCP. Com sessão ativa, a comunicação com HTTPS é feita com os seguintes componentes criptografados: URL do conteúdo e conteúdo requisitado, formulários, *cookies* enviados do servidor para o cliente e do cliente para o servidor e conteúdos de cabeçalho HTTP. Portanto, podemos ver que TLS favorece a confidencialidade, integridade e autenticidade para proteger a comunicação de ponta a ponta, sendo um dos serviços de segurança mais amplamente utilizados.

2.5 Segurança de Sistemas Web

Sistemas Web estão em alto nível de abstração quando todo o sistema computacional necessário para seu funcionamento é observado. Eles funcionam em uma ou mais máquinas com função de servidor, que rodam em sistemas operacionais instalados em hardwares, fazem uso de redes para comunicação e, ainda, fazem reuso de serviços e componentes de software de terceiros para seu funcionamento. Todas essas partes possuem suas complexidades individuais. Os sistemas operacionais, por exemplo, podem ter milhões de linhas de código, de acordo com Silberschatz (2018). Além disso, novas vulnerabilidades podem ser descobertas no futuro, e aplicações de software tendem a crescer em complexidade conforme evoluem. Por esses motivos, afirmar que um sistema Web funciona completamente de modo seguro é uma tarefa impraticável. Apesar disso, podemos atender a requisitos de segurança que amenizam ricos e eliminam vulnerabilidades para alcançar níveis de segurança satisfatórios, de acordo com o escopo de cada aplicação. Diferentes tipos de aplicações possuem

diferentes informações e requisitos de software, criando seus valores que motivam diferentes tipos de ameaças.

No desenvolvimento de sistemas Web, devem ser avaliados critérios de segurança para o *client-side*, o s*erver-side* e o meio de comunicação entre eles. No *client-side*, é preciso tratar as entradas do usuário para restringir suas ações no sistema, como de costume em diversos outros tipos de sistema, sendo elas originadas em navegadores ou por outros meios de realizar requisições. O meio de comunicação deve ser protegido com criptografia e protocolos seguros, tanto para acesso à camada de negócios quanto à de dados. E, no s*erver-side,* também deve-se tomar cuidado com os dados recebidos e qualquer outra forma de interação com o sistema, para que a camada de negócios funcione de acordo com as regras de negócios, evitando qualquer funcionamento indesejado e acesso não autorizado, incluindo a proteção do acesso à camada de dados e o que é apresentado ao usuário. Em todos os casos, precisamos considerar a seção sobre propriedades de segurança apresentada anteriormente.

2.6 Segurança para Desenvolvedores

É importante considerar a segurança desde as primeiras fases do desenvolvimento de software, adicionando essa prática no seu ciclo de desenvolvimento. A OWASP disponibiliza técnicas de segurança para desenvolvedores em seu projeto *Pro Active Controls For Developers 2018 V 3.0*, onde são listados dez tipos de controles. *Open Web Application Security Project* é uma organização mundial sem fins lucrativos com foco na segurança de *software*. Sua missão é tornar a segurança de *software* visível para indivíduos e organizações no mundo inteiro. Seus projetos são abertos à comunidade onde todos podem participar e ter acesso aos seus materiais de forma gratuita. A seguir, são apresentados cada um dos seus controles proativos em ordem de importância, sendo o primeiro o mais importante.

2.6.1 Definir Requisitos de Segurança

Requisitos de segurança adicionam novas características ou complementam outras existentes para resolver problemas de segurança específicos ou eliminar potenciais vulnerabilidades. Eles são provenientes de padrões industriais, leis e histórico de vulnerabilidades, que inclusive podem ter sido alvo no passado. Isso evita criar novas soluções para cada aplicação, permitindo reuso de padrões de requisitos de segurança e suas melhores práticas.

Implementar requisitos com sucesso envolve quatro etapas, que são: seleção, investigação, implementação e avaliação da implementação. Na seleção é onde são descobertos requisitos e selecionados para serem implementados de forma iterativa. Na investigação, a aplicação é revisada com base nos novos requisitos de segurança. Na implementação, a aplicação é modelada para ser adequada ao requisito e o desenvolvedor atualiza o código. Por fim, casos de testes devem ser criados para confirmar se o novo requisito foi cumprido.

2.6.2 Selecionar Frameworks e Bibliotecas de Segurança

Frameworks e bibliotecas com segurança embutidas ajudam desenvolvedores de software a protegerem suas aplicações. Implementar soluções do zero pode ser inviável devido a falta de conhecimento, tempo e fundos financeiros. Por isso, utilizá-los ajuda a alcançar objetivos de segurança de modo mais efetivo e preciso. Porém, é importante que componentes de terceiros sejam confiáveis, ativamente mantidos e usados por muitas aplicações. Além disso, deve-se manter suas versões atualizadas, sendo útil o uso de ferramentas de gerenciamento de dependências.

2.6.3 Assegurar Acesso ao Banco de Dados

Bancos de dados devem ser assegurados nas consultas, configuração, autenticação e comunicação. Injeção SQL é um dos riscos mais perigosos, segundo

OWASP (2017), sendo o número um do seu Top 10. Ele é fácil de ser disparado e pode comprometer a confidencialidade, integridade e disponibilidade do banco de dados e, consequentemente, a funcionalidade de toda a aplicação. Por exemplo, os dados podem ser copiados, alterados ou apagados, permitindo o atacante de controle sobre eles, até mesmo sem ser percebido.

Também, é importante considerar as configurações de segurança necessárias para levar bancos de dados em produção, e isso deve ser feito especificamente para cada distribuidor de banco. Os acessos ao banco devem ser autenticados e o canal de comunicação protegido.

2.6.4 Codificar e Escapar Dados

Codificação e escape são técnicas para impedir ataques de injeção. Codificação (ou codificação de saída) é traduzir caracteres especiais em algo diferente mas equivalente, para que não seja interpretado um código malicioso. Escape envolve adicionar um caractere especial antes de um caractere ou string para evitar que ele seja mal interpretado, por exemplo, adicionar ″\″ (barra para trás) antes de ″″″ (aspa dupla) para que seja interpretado como uma string e não um comando de fechar string.

2.6.5 Validar Entradas

Validação de entrada é uma técnica de software que garante que somente dados propriamente formatados sejam aceitos no sistema. Dados devem ser verificados sintaticamente e semanticamente, nessa ordem, antes de utilizá-los. Existem duas formas de validação: *blacklisting*, que verifica se a entrada não é desejada, e *whitelisting*, que verifica se a entrada é desejada. *Whitelisting* é a abordagem mínima recomendada, enquanto *blacklisting* tende a ser facilmente evitada por atacantes, mas pode ser útil para identificar ataques óbvios.

2.6.6 Implementar Identidade Digital

Identidade digital é a representação única de um usuário em uma transação. Ela pode ser verificada com autenticação, para certificar que um usuário é quem ele diz ser. Para manter o estado de autenticidade de um usuário, permitindo seu acesso sem precisar autenticar novamente, utiliza-se gerenciamento de sessão. Assim, o usuário não precisa enviar suas credenciais sempre que quiser acessar um recurso.

2.6.7 Aplicar Controle de Acesso

Controle de acesso (ou autorização) é o processo de aceitar ou rejeitar requisições específicas de um usuário ou programa. Todos os acessos devem ser verificados, seguindo o princípio do menor privilégio para os clientes. E, por padrão, se um tipo de requisição não é especificamente permitido, ele deve ser negado. Um modo de fazer isso é com Controle de Acesso Baseado em Papel. Esse é um modelo para controlar acesso a recursos onde ações são identificadas com papéis de usuários. Por exemplo, um usuário pode ter papel de usuário final, enquanto outro pode ter papel de administrador do sistema, que possui acesso a recursos restritos.

2.6.8 Proteger Dados

Dados podem ser classificados como públicos e privados. Dados sensíveis precisam de proteção extra, tais como senhas, informação pessoal, registros de saúde e segredos de negócios. Para armazenamento de informação sensível, uma regra é evitar sempre que possível e, quando necessário, fazer armazenamento com criptografia. Para transmitir dados sensíveis através da rede, segurança para comunicação fim-a-fim deve ser considerada. Na Web, TLS é o protocolo criptográfico mais utilizado para este propósito, como descrito anteriormente na seção sobre protocolo de comunicação.

2.6.9 Implementar Registro de Segurança

Registrar informações é uma prática normalmente conhecida por desenvolvedores, também conhecida como *logging,* pois muitos já utilizaram para *debug*. Na segurança, registra-se informação durante a execução de uma aplicação com uma perspectiva defensiva para identificar possíveis ataques e permitir sua análise. Registros podem ser usados para realizar monitoramento de segurança com apresentação em tempo real.

2.6.10 Tratar Erros e Exceções

Tratamento de exceção é um conceito de programação que permite aplicações identificarem e, possivelmente, responderem a erros em tempo de execução. Essa prática é importante para desenvolver código confiável e tolerante a falhas. Não tratar erros pode fazer a aplicação parar de funcionar completamente. Erros também podem ser disparados por ataques, sendo possível identificá-los tratando erros e exceções.

3 ESTUDO DE CASO: VOCÊ DIGITAL

3.1 Visão Geral

Você Digital é um projeto de pesquisa e desenvolvimento entre o Laboratório de Vídeo e Aplicação Digital (LAViD) com base na Universidade Federal da Paraíba (UFPB) e o Tribunal de Contas do Estado da Paraíba (TCE-PB) para modelagem e desenvolvimento de uma plataforma computacional colaborativa de governo eletrônico. O principal objetivo é permitir a automação de diagnósticos e escutar populares, para possibilitar melhor interação e comunicação entre a sociedade e os gestores públicos.

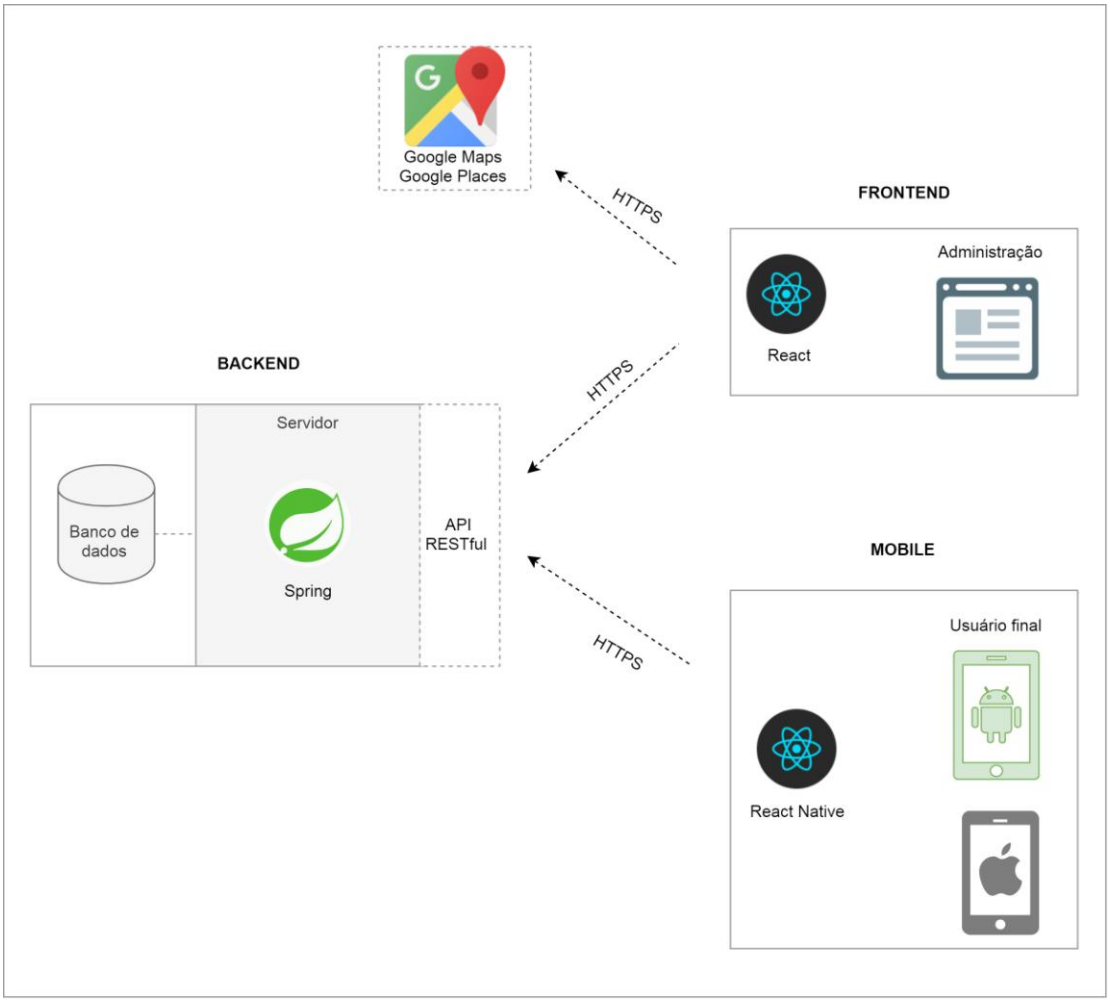
A intenção é explorar a inteligência coletiva presente nas redes, incentivando a participação cidadã, o que pode ajudar na redução de custos no processo de geração desse plano, e também na promoção da transparência e confiabilidade. Além disso, pretende-se aproveitar das vantagens computacionais para atender a demanda da sociedade de forma democrática e fazer bom uso do tempo necessário para atender sua colaboração por meio de visualização virtual. Como ferramenta de gestão pública digital, a ideia é avaliar serviços públicos tendo como auxílio a participação popular no processo de tomada de decisões de auditores e gestores públicos. O sistema está em desenvolvimento e, quando estiver em produção, espera-se identificar problemas em diversas áreas de gestão pública como, por exemplo, educação, saúde e segurança.

3.2 Arquitetura de Alto Nível

O sistema Você Digital é composto por três subsistemas interconectados: *backend* (aplicação no servidor e banco de dados), *frontend* (página de administração) e *mobile* (aplicação móvel). O *frontend* faz uso das APIs Google Maps e Google Places para obter Pontos de Interesse (ou *Point of Interest* - POI). Os POIs são enviados para o *backend* através da API RESTful, onde são inseridos no banco de dados do sistema. No *backend*, são adicionadas regras de negócios e, então, são disponibilizadas funcionalidades na API para o *frontend* e o *mobile.* Esses POIs são

órgãos públicos que estarão disponíveis nas plataformas Android e IOS para usuários avaliarem seus serviços.

Figura 10 - Arquitetura de alto nível Você Digital



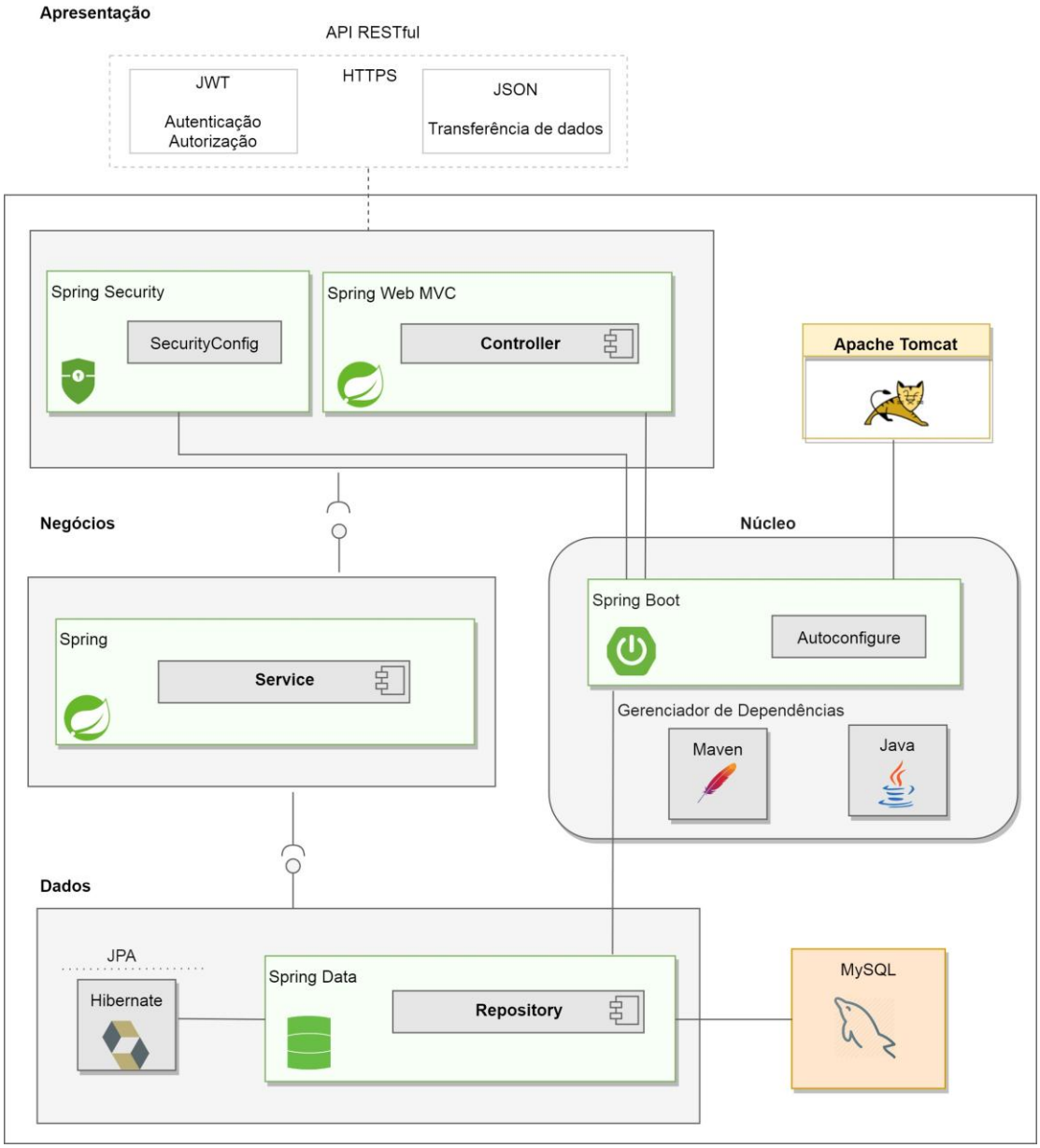
Fonte: Autor (2018)

3.3 Arquitetura *Backend*

A figura a seguir ilustra a arquitetura do *backend* com suas principais tecnologias. Se trata de um sistema Web RESTful desenvolvido com o *framework* Spring, com o qual são aproveitadas autoconfigurações do Spring Boot. Iniciando pelo núcleo da aplicação, é utilizada a linguagem Java, fazendo uso do seu paradigma de programação orientada a objetos. O servidor utilizado é o Tomcat, que implementa a API Servlet de Java. Todas as dependências necessárias são gerenciadas pelo Maven.

É utilizada a arquitetura de pilha síncrona com Spring Web MVC. Os componentes controladores recebem requisições dos clientes e processam as lógicas de negócios nos componentes de serviços. Os componentes de serviços acessam os componentes de repositório, a partir dos quais é possível realizar consultas no banco de dados. O acesso a dados é implementado com Spring Data e as consultas são feitas com o *framework* Hibernate e seu mapeamento de objeto relacional. Logo, é utilizado o banco relacional MySQL. A API é implementada com REST e o formato dos dados transferidos em JSON. A comunicação é realizada e protegida através do protocolo HTTPS.

Figura 11 - Arquitetura *backend* Você Digital

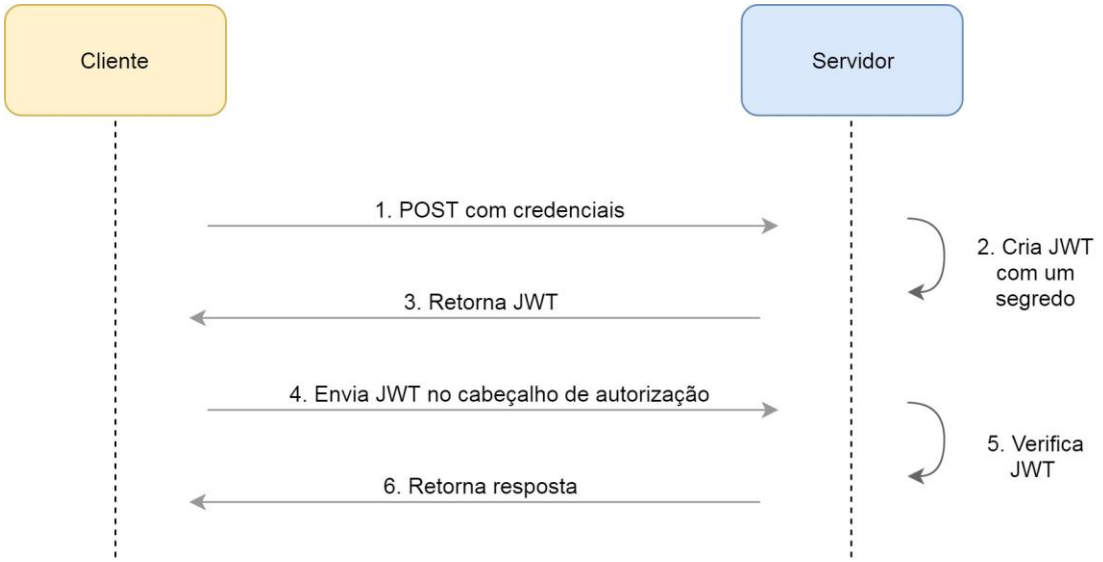


Fonte: Autor (2018)

Requisitos provenientes do cliente requerem registro de usuários com diferentes níveis de acesso ao sistema. É necessário verificar quem é o usuário nas funcionalidades protegidas da API e restringi-las à papéis de usuários específicos. Com este propósito, é configurado o padrão *JSON Web Token* (JWT) com Spring Security para autenticação e autorização no sistema. Após autenticar o usuário, o ideal é permitir seu acesso ao sistema sem exigir suas credenciais em todas as requisições. Para isso, o gerenciamento de sessão é feito também com JWT.

JWT é um padrão aberto, suportado por Auth0 (2018), definido na RFC 7519 para transmitir informação de modo seguro no formato JSON. Ele é confiável por ser assinado digitalmente. Ele pode ser assinado com uma sequencia de caracteres secreta ou com uma chave pública ou privada. Sua sequência de utilização é ilustrada na Figura 9, que é a solução implementada para o sistema proposto.

Figura 12 - Sequência de utilização do JWT



Fonte: Autor (2018)

Um JWT consiste de três partes: cabeçalho, carga útil e assinatura. O cabeçalho possui o tipo do *token* e o tipo do algoritmo *hash*. A segunda parte é a carga útil, geralmente informação para identificar o cliente e tempo de expiração do *token*. Dados sensíveis não devem ser enviados junto com a carga, a menos que sejam criptografados. O cabeçalho e a carga são codificados separadamente com Base64Url (RFC 4648) e, então, são utilizados para criar a terceira parte, a assinatura. Para ser

criada, aplica-se o algoritmo escolhido no cabeçalho e na carga. A assinatura é utilizada para verificar se a mensagem não foi alterada. A saída são três *strings* resultantes das três partes anteriores, concatenadas e separadas por pontos na forma *xxxxx.yyyyy.zzzzz*.

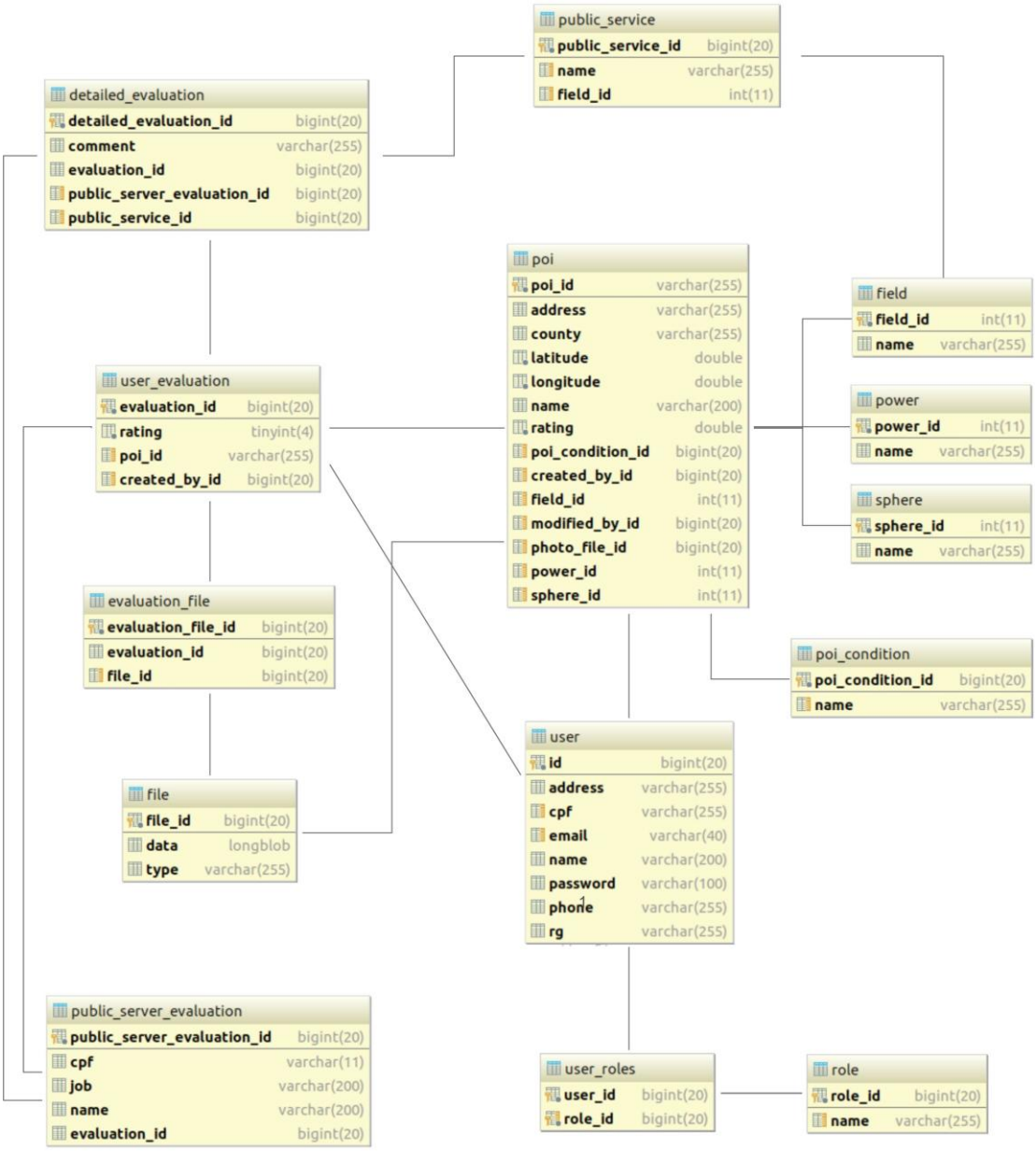
3.4 Arquitetura de Baixo Nível

O modelo de dados do sistema é ilustrado na figura a seguir. Ele é feito no modelo relacional com uso de tabelas. O projeto está em desenvolvimento e esse modelo de dados ainda passará por modificações. Ele tem sido modelado em sincroniza com os clientes *frontend* e *mobile*. A seguir, são descritas cada uma das tabelas.

- *poi* representa órgãos públicos; ele possui dados básicos de identificação como nome e localização por latitude e longitude; faz parte de alguma divisão pública; possui foto, identificação do usuário que o criou, modificou e uma condição atual;
- *poi_condition* representa uma condição ou estado de um POI; por exemplo: POI candidato a ser adicionado no sistema, registrado ou removido;
- *user* representa usuários do sistema com dados pessoais e suas credenciais;
- *user_evaluation* representa a avaliação de um POI feita por um usuário; é uma avaliação quantitativa;
- *public_service_evaluation* relaciona uma avaliação de POI com um serviço;
- *detailed_evaluation* possui detalhes de uma avaliação existente; pode estar relacionada a um serviço e um servidor público e possui um comentário;
- *evaluation_file* relaciona uma avaliação de POI com um arquivo;

- *public_service* representa serviços públicos, os quais são relacionados com áreas públicas;
- *file* representa um arquivo genérico, podendo ser utilizado para foto de POI e para arquivos de avaliação;
- *role* representa papéis de usuários;
- *user_roles* relaciona um usuário com um papel de usuário;
- *power* representa poder público; por exemplo: poder executivo ou judiciário;
- *sphere* representa esferas públicas; por exemplo: esfera estadual ou municipal;
- *field* representa áreas públicas; por exemplo: segurança ou educação.

Figura 13 - Modelo de dados Você Digital



Fonte: Autor (2018)

4 AVALIAÇÃO

Neste capítulo, são apresentadas as avaliações feitas para implementação dos requisitos do estudo de caso apresentado, o projeto Você Digital. Mais especificamente, se trata da segurança aplicada em todo o *backend*, isto é, a aplicação servidor e acesso à camada de dados, do ponto de vista do desenvolvedor. Foi adotada uma abordagem de medidas proativas, ou seja, criando controles como prevenção antes de algo negativo acontecer. No Capítulo 2.6, foi apresentado o projeto da OWASP *Pro Active Controls For Developers* com um resumo sobre técnicas de segurança para desenvolvedores. Nesta seção, as técnicas selecionadas para o *backend* são exploradas mais detalhadamente por meio de técnicas e aplicação no sistema desenvolvido.

4.1 Definir Requisitos de Segurança

O projeto proposto é abordado com metodologia ágil, realizando entrega de valor rápida, contínua e iterativa. Conforme requisitos funcionais de usuários são solicitados, são pensados também nos requisitos não funcionais, inclusive de segurança. Os requisitos de segurança serão abordados adiante.

4.2 Selecionar Frameworks e Bibliotecas de Segurança

O *framework* utilizado é o Spring, pois possui soluções modernas prontas para uso de desenvolvimento de aplicações, incluindo integração e segurança. Além disso, o projeto tem código aberto no Github, está disponível para a comunidade colaborar e solicitar suporte, e está em desenvolvimento, sendo atualizado frequentemente.

4.3 Assegurar Acesso ao Banco de Dados

Credenciais de configuração padrão após instalação do banco não devem existir, assim como outras credenciais fracas normalmente usadas durante o

desenvolvimento. Aplicações usuárias do banco devem ter nível de acesso limitado à sua utilização.

A comunicação com o banco de dados deve ser feita somente através do protocolo HTTPS, fazendo uso do protocolo TLS/SSL para prover autenticidade, confidencialidade e integridade.

Todos os acessos ao banco devem ser propriamente autenticados. A aplicação que conecta ao banco deve usar variáveis de ambiente para suas credenciais. Elas não devem estar no código fonte. O acesso ao banco para consultas é feito somente por meio da aplicação Spring, a qual possui autenticação e autorização para suas funcionalidades.

Quando entrada de usuário dinâmica é adiciona à uma consulta SQL de modo inseguro, pode ocorrer uma injeção SQL, geralmente por concatenação de strings. Para amenizar esse risco, deve-se evitar que entradas não confiáveis sejam interpretadas diretamente como parte de um comando SQL.

Exemplo de vulnerabilidade em Java, conforme OWASP (2017):

```
String query = "SELECT * FROM conta
WHERE id_cliente='" + request.getParameter("id") + "'";
```

O atacante poderia usar valores na variável *id* para realizar consultas maliciosas, diferentes da regra de negócios da aplicação. Segue um exemplo abaixo:

```
 'or '1'='1
```

Essa expressão adiciona uma condição que o resultado é sempre verdadeiro, retornando todos os registros de *conta*. A consulta completa seria:

```
String query = "SELECT * FROM conta
WHERE id_cliente='' or '1'='1'";
```

Para prevenir isso, parametrização de consultas (OWASP Cheat Sheet, 2018) é a melhor opção. Java oferece a classe *PreparedStatement* da API JDBC (Oracle, 2018), que separa a entrada do contexto de comando SQL.

```
PreparedStatement ps = connection.preparedStatement(
"SELECT * FROM conta WHERE id_client = ?");
ps.setString(1, id);
```

Apesar de ser possível cometer o mesmo erro com Hibernate, Spring Boot permite utilizá-lo com entrada do usuário parametrizada, conforme seção sobre o *framework* Spring. Assim, ao realizar consultas, entradas com possíveis comandos maliciosos não serão interpretadas como comandos SQL. Mais detalhes podem ser consultados na documentação oficial Hibernate (2018).

O mesmo cuidado deve ser tomado com procedimento armazenado SQL (*Stored Procedure)*. Se construído dinamicamente, existe o risco de injeção SQL. Por isso, deve-se parametrizar variáveis semelhantemente ao exemplo de consulta anterior. Mesmo com esses cuidados, ainda deve-se validar a entrada do usuário antes de utilizá-la no programa. Essa técnica é abordada na seção Validar Entradas deste capítulo.

4.4 Validar Entradas

Em muitos casos, entrada de usuários não deve ser usada diretamente na lógica de negócios da aplicação. No estudo de caso, por exemplo, temos um serviço que permite o usuário buscar entidades próximas de sua localização. Essa busca é feita com sua latitude e longitude atual e um valor de raio desejado. Para evitar que o usuário obtenha todas as entidades do sistema, o raio é limitado em um valor máximo. Caso a aplicação receba um valor superior, o valor máximo é utilizado na busca.

Em Spring, é possível validar dados diretamente no parâmetro do método que recebe a requisição HTTP. Isso pode ser feito definindo um tipo e validar cada um de

seus atributos com anotações de Java do pacote *javax.validation.constraints* (Oracle, 2018).

```
class CadastroUsuario {

    @Email
    String email;

    @Size(min = 10)
    String senha;
}
```

A classe *CadastroUsuario* possui dois atributos com seus métodos *set* e *get* omitidos. O atributo *email* deve ter formato de email conforme a expressão regular da anotação *@Email*. O atributo *senha* deve ter comprimento mínimo 10, pois é um dos requisitos para criação de senha forte da OWASP quando autenticação multi fator não é utilizada. Assim, o formato dos dados de uma requisição é estruturado como objeto e validado no método que recebe a requisição com *@Valid*. Se a entrada for inválida, é lançada uma exceção com uma mensagem para o cliente e código de requisição mal formada (HTTP 400 *Bad Request*). É possível também personalizar a resposta, porém omitida aqui.

```
@PostMapping("/cadastro")
void cadastrar(@Valid @RequestBody
CadastroUsuario cadastroUsuario) {

}
```

4.5 Implementar Identidade Digital

Como requisitos do sistema proposto, é preciso identificar usuários nas aplicações de administração e usuário final. Na página de administração, só é permitido acesso autenticado. Um usuário administrador tem acessos restritos somente a ele. Só ele pode gerenciar e criar e outros usuários com qualquer nível de acesso disponível. Usuários finais podem se registrar e têm nível de acesso conforme seu papel. Isto é, usuários possuem papéis com diferentes permissões no sistema, e como esse controle é feito é apresentado na sessão Aplicar Controle de Acesso.

Como apresentado no Estudo de Caso, o sistema proposto implementa autenticação com *JSON Web Token*. As aplicações clientes precisam autenticar-se

para obter acesso informando as credenciais do usuário. Deve-se, por exemplo, realizar um método POST no caminho */login* com as credenciais no formato JSON.

```
POST http://localhost:8080/login
Content-Type: application/json;charset=UTF-8

{
  "email": "email@email.com",
  "senha": "0000000000"
}
```

As credenciais são verificadas no banco de dados e, se estiverem corretas, a autenticação é bem sucedida e é retornado o *token* de acesso e seu tipo no corpo da resposta da requisição, também no formato JSON.

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
Content-Length: 221

{
  "accessToken" :
"eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiIxMzMiLCJpYXQiOjE1NDA5MjYz
MTIsImV4cCI6MTU0MTUzMTExMn0.k4lFKgA5Nk9-
jHAk4qyPx2JZfVeGumeCrPXdohA30gxT0J1RfC45LF5dSm8ihbjYVTBpY8
cQ7Y72qNxXGuqYnQ",
  "tokenType" : "Bearer"
}
```

A partir de então, a aplicação cliente pode enviar o *token* no cabeçalho de autorização do método HTTP nas requisições seguintes para ter acesso ao sistema de forma autenticada. A sintaxe é descrita abaixo.

```
Authorization: Bearer <token>
```

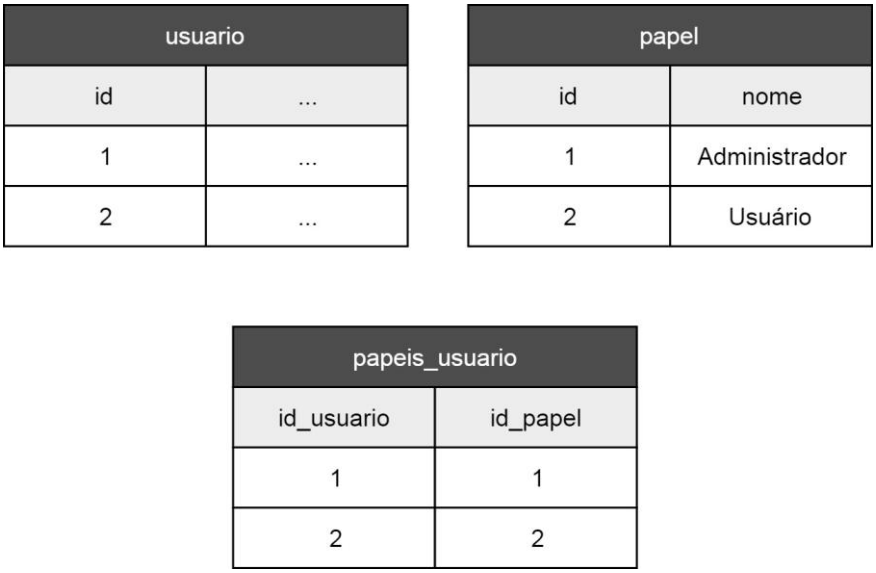
4.6 Aplicar Controle de Acesso

Spring Security permite realizar controle de acesso em nível de método, classe e endereço (URL). No sistema proposto, o controle é feito em nível de método nos componentes de serviço, pois existem funcionalidades que servem para todos os

papéis de usuário, dificultando assegurar por classe ou por URL. Quando as regras de negócios estiverem mais bem definidas, será possível realizar controle por classe.

A API bloqueia todas as requisições não autorizadas e retorna o código de proibição HTTP 403 (*Forbidden*). Para implementar papéis de usuário, foi utilizado o modelo relacional com três tabelas. Uma para a entidade *usuário*, outra para a entidade *papel*, e a terceira para o relacionamento entre usuário e seus possíveis papéis, *papéis_usuário*. Isto é, um usuário pode ter mais de um papel.

Figura 14 - Papéis de usuários em modelo relacional



Fonte: Autor (2018)

Para verificar os papéis de usuário, após a requisição de um usuário ser autenticada, seus dados são carregados no sistema e Spring mantém a instância do usuário durante toda a sessão. Seus papéis são verificados com AOP pelo *framework* Spring Security, que é implementado de forma extensível. Sempre que o usuário realizar uma requisição, verifica-se se ele possui o papel compatível com o papel exigido no método que implementa a lógica de negócios da aplicação.

```
@Secured(ADMIN)
void fazerAlgo () {
}
```

No código, basta adicionar uma anotação no topo do método indicando os papéis permitidos. Nesse caso, foi utilizada a anotação *@Secured* do Spring para permitir somente acesso administrador no método *fazerAlgo,* onde *ADMIN* é uma

constante com o nome *Administrador* da tabela papel na Figura 11. No sistema proposto, são utilizados cinco papéis diferentes de usuário.

4.7 Proteger Dados

É comum em diversos sistemas que exigem autenticação de usuário possuir um tipo de senha. Elas precisam ser armazenadas no lado do servidor. No caso estudado, usuários também precisam se identificar com suas credenciais, incluindo sua senha. Senha é um dado sensível, logo não pode ser armazenada em texto simples, pois se a confidencialidade do banco é violada, todas as senhas armazenadas são comprometidas. Por isso, a senha é armazenada com criptografia *hash*.

O algoritmo utilizado para proteger a senha é BCrypt, proposto por Provos (1999), um algoritmo de criptografia *hash*. Ele usa a técnica do uso de *sal* (no inglês, *salt). Sal* auxilia na geração de diferentes *hashes* para senhas iguais e controla o fator de trabalho. O fator e o *sal* são utilizados juntamente com a senha original para gerar o *hash*. É aconselhável que o *sal* seja sempre aleatório. Para adequar o algoritmo à evolução do processamento computacional, ele possui um parâmetro para personalizar o fator de trabalho, podendo tornar seu processamento mais custoso. Essa mesma técnica é utilizada para reduzir o número de tentativas de um possível atacante, sendo considerado lento em processamento, e evita reaproveitamento de *hash*, amenizando *ataques de dicionário.*

O resultado em *hash* possui quatro informações: a versão do algoritmo, o fator de trabalho, o valor *sal* e a senha ilegível, conforme Arias (2018). Ele é implementado pelo Spring Security (2018), conhecido e de código aberto. Assim, a senha do usuário não é armazenada em texto simples. Antes, ela é codificada com *sal* aleatório e só então é armazenada em forma *hash*. Na Tabela 2, são apresentados resultados de *hashes* diferentes utilizando duas senhas iguais para codificação.

Tabela 2 - Resultado de *hashes* gerados com BCrypt

Senha	Hash
pass123456	`$2a$10$LqbDQfYjMBNc.lEGSouwV.p0acvxlQoUB2UVDju.cvxnLlYl2Yw9K`
pass123456	`$2a$10$9IqBzNUSSL6UkwrVQiFcW.liSl9mmMWzreYP8j2yRNFaJ3woTXFHK`

Fonte: Autor (2018)

Por fim, no canal de comunicação entre cliente e servidor, os dados devem ser protegidos com HTTPS. Caso contrário, o tráfego pode ser comprometido.

4.8 Tratar Erros e Exceções

No sistema estudado, tratamento de exceção é utilizado para evitar que a aplicação quebre e enviar respostas adequadas para clientes do serviço. Se uma requisição é feita e um recurso que não é encontrado, é lançada uma exceção que emite uma resposta HTTP de código 404 (*Not Found*). O mesmo é feito para entradas inválidas, como tentativa de cadastro de usuário com mesmo email e CPF (Cadastro de Pessoa Física), sendo retornada uma resposta HTTP de código 409 (*Conflict*). O mesmo serve para recebimento de arquivo. Se ele não possuir extensão válida, é lançada uma exceção que emite resposta HTTP de código 400 (*Bad Request*).

4.9 Conclusão das avaliações

Neste capítulo, foram apresentadas soluções de segurança existentes e utilizadas pela comunidade, aplicando especificamente para os requisitos do sistema estudado. Alguns exemplos foram apresentados e, a partir deles, foi possível perceber a importância da seleção de tecnologias durante a fase de projeto de *software*. Foi feito reuso das funcionalidades disponibilizadas pelo *framework* Spring, incluindo seu módulo de segurança Spring Security. Além disso, percebe-se como a segurança se relaciona com diferentes partes de *software,* colaborando com sua segurança como um todo.

5 CONCLUSÕES E TRABALHOS FUTUROS

Este trabalho apresentou histórico e conceitos relacionados ao aspecto de segurança do projeto e implementação de sistemas Web. A principal contribuição foi o desenvolvimento de um serviço Web no contexto de um projeto do TCE-PB e que considerou os seguintes mecanismos de segurança

1. apresentação da importância de canal de comunicação seguro entre as aplicações e onde deve ser utilizado;
2. implementação da identificação de identidade digital para autenticação no sistema, onde tem-se dois tipos diferentes de clientes, com flexibilidade para outros novos;
3. funcionalidades do sistema são protegidas com controle de acesso por meio de papéis de usuários, onde os que são registrados possuem seus papéis devidos;
4. acesso ao banco de dados é realizado por intermédio da aplicação servidor autenticada, a qual toma contramedida a ataques de injeção.
5. senhas de usuários são armazenadas com solução de criptografia atual e utilizadas por diversas outras aplicações; e
6. entradas dos usuários são validadas de acordo com os requisitos do sistema e os possíveis erros e exceções são tratados, procurando devolver respostas adequadas ao usuário.

Adicionalmente, são proposta as seguintes melhorias:

- registros de segurança podem ser implementados de acordo com os riscos do sistema, inclusive com monitoramento em tempo real;
- autenticação multifator pode ser considerada para maior segurança;
- testes de segurança automatizados de fora da aplicação podem ser realizados para melhor validação do sistema, inclusive com ferramentas existentes próprias para isso.

Para trabalhos futuros, esta pesquisa pode ser utilizada para compreensão de sistemas Web atuais e suas técnicas de segurança, mesmo para outras arquiteturas e tecnologias. O modelo de autenticação apresentado pode ser utilizado como base para outros protocolos entre aplicações, inclusive comunicação em arquitetura de microserviços. Além disso, são sugeridos novos estudos para definir uma metodologia, com o apoio de ferramentas de automação, para avaliar o nível de segurança de sistemas Web.

REFERÊNCIAS

SOMMERVILLE, Ian. **Software Engineering**. 10. ed. London: Pearson Education, Inc, 2015.

ZALEWSKI, Michal. **The Tangled Web**. San Francisco: No Starch Press, Inc, 2012.

DEITEL, Paul J et al. **Internet & World Wide Web**. Boston: Pearson Education, Inc, 2012.

Cybersecurity Framework. National Institute of Standards and Techonology, 2018. Disponível em: https://www.nist.gov/cyberframework. Acesso em: 5/11/2018.

ROMANOSKY, Sasha. Examining the costs and causes of cyber incidents. **Journal of Cybersecurity**, 20 páginas, junho, 2016.

HIBSHI, Hanan et al. A grounded analysis of experts' decision-making during security assessments. **Journal of Cybersecurity**, 17 páginas, agosto, 2016.

GAMMA, Erich. HELM, Richard. JOHNSON, Ralph. VLISSIDES, John. **Design Patterns**. Boston: Pearson Education Corporate Sales Division, 1996.

HALL, Gary. **Adaptive Code**. Redmond: Online Training Solutions, Inc, 2017.

GROEF, Willem De. **Client- and Server-Side Security Technologies for JavaScript Web Applications**. 2016. Doctor of Engineering Science (PhD): Computer Science. Faculty of Engineering Science, Ku Leuven, Leuven, 2016.

ELMASRI, Ramez e NAVATHE, Shamkant B. **Fundamentals of Database Systems**. 7. ed. Haboken: Pearson Education, Inc, 2016.

NEWMAN, Sam. **Building Microservices**. Sebastopol: Pearson Education, Inc, 2015.

BONÉR, Jonas. **Reactive Microservices Architecture**. Sebastopol: Pearson Education, Inc, 2016.

FOX, Richard e HAO, Wei. **Internet Infrastructure**. New York: Taylor & Francis Group, LLC, 2018.

RASMUSSON, Jonathan. **The Agile Samurai**. Raleigh: The Pragmatic Bookshelf, 2010.

LÄMMEL, Ralf e PEYTON, Simon. Scrap You Boirlerplate: A Practical Design Pattern for Generic Programming. **ACM SIGPLAN**, 13 páginas, março, 2003.

Spring Framework. Pivotal Software, Inc, 2018. Disponível em: https://spring.io/. Acesso em: 18/8/2018.

Módulos Spring Framework. Pivotal Software, Inc, 2018. Disponível em: https://docs.spring.io/spring/docs/3.0.0.M4/reference/html/ch01s02.html. Acesso em: 18/8/2018.

Java Servlet Specification, 2018. Disponível em: https://javaee.github.io/servlet-spec/. Acesso em: 18/09/2018.

WALL, Craig. **Spring In Action**. 2. ed. Shelter Island: Manning Publication Co, 2015.

Spring Initializr. Pivotal Software, Inc, 2018. Disponível em: https://start.spring.io/. Acesso em: 18/8/2018.

Maven. The Apache Software Foundation, 2018. Disponível em: https://maven.apache.org/. Acesso em: 18/8/2018.

Postman., 2018. Postdot Technologies, Inc, 2018. Disponível em: https://www.getpostman.com/. Acesso em: 18/8/2018.

Hibernate ORM. Red Hat Developers, 2018. Disponível em: http://hibernate.org/orm/. Acesso em: 18/8/2018.

LANGR, Jeff. **Pragmatic Unit Testing in Java 8 with JUnit**. Raleigh: The Pragmatic Bookshelf, 2015.

Thymeleaf Template. The Thymeleaf Team, 2018. Disponível em: https://www.thymeleaf.org/. Acesso em: 18/8/2018.

ARAÚJO, Claudiomar. spring-boot-tutorial, 2018. Disponível em: https://github.com/claudiomarpda/spring-boot-tutorial. Acesso em: 18/8/2018.

Request For Comments 4949. **Internet Security Glossary**. 2. v. Internet Engineering Task Force, 2007.

United States Code. **Title 44- Public Printing and Documents, Subchapter III – Information Security.** Seção 3542, U.S Government Publishing Office, 2011.

FIPS PUB 199. **Standards for Security Categorization of Federal Information and Information Systems.** Gaithersburg: National Institute of Standards and Technology, 2004.

ISO/IEC 27000. **Information technology – Security techniques – Information security management systems – Overview and vocabulary**. 5. ed, 2018.

STALLINGS, William e BROWN, Lawrie. **Computer Security**. 3. ed. Boston: Pearson Education, Inc, 2015.

STALLINGS, William. **Cryptography and Network Security**. 7. ed. Boston: Pearson Education, Inc, 2017.

Mozilla. HTTP. MDN Web docs, 2018. Disponível em: https://developer.mozilla.org/en-US/docs/Web/HTTP. Acesso em: 5/7/2018.

TANENBAUM, Andrew S. e WETHERALL, David J. **Computer Networks**. 5. ed. Boston: Pearson Education, Inc, 2011.

OWASP. OWASP Pro Active Controls - 2018. **Open Web Application Security Policy**. 40 páginas, 2018.

OWASP. OWASP Top 10 - 2017. **Open Web Application Security Policy**. 24 páginas, 2017.

JSON Web Token. Auth0, Inc, 2018. Disponível em: https://jwt.io/introduction/. Acesso em: 29/10/2018.

IETF. Request For Comments 7519, 2018. Disponível em: https://tools.ietf.org/html/rfc7519. Acesso em: 30/10/2018.

IETF. Request For Comments 4648, 2018. Disponível em: https://tools.ietf.org/html/rfc4648. Acesso em: 30/10/2018.

Spring Projects. Pivotal Software, Inc, 2018. Disponível em: https://github.com/spring-projects. Acesso em: 1/11/2018.

OWASP. Query Parametrization Cheat Sheet. Disponível em: https://www.owasp.org/index.php/Query_Parameterization_Cheat_Sheet. Acesso em: 2/11/2018.

Oracle. Java PreparedStatement. Disponível em: https://docs.oracle.com/javase/tutorial/jdbc/basics/prepared.html. Acesso em: 2/11/2018.

Hibernate. Red Hat, Inc, 2018. Database access. Disponível em: http://docs.jboss.org/hibernate/orm/5.3/userguide/html_single/Hibernate_User_Guide.html. Acesso em: 2/11/2018.

Javax Validation Constraints. Oracle, 2018. Disponível em: https://javaee.github.io/javaee-spec/javadocs/javax/validation/constraints/package-summary.html. Acesso em: 2/11/2018.

PROVOS, Niels e MAZIÈRES, David, Simon. A Future-Adaptable Password Scheme. **USENIX**, 13 páginas, junho, 1999.

ARIAS, Dan. Hashing in Action: Understanding bcrypt. Auth0, Inc, 2018. Disponível em: https://auth0.com/blog/hashing-in-action-understanding-bcrypt/. Acesso em 3/11/2018

BCrypt. Pivotal Software, Inc, 2018. Disponível em: https://docs.spring.io/spring-security/site/docs/4.2.5.RELEASE/apidocs/org/springframework/security/crypto/bcrypt/BCrypt.html. Acesso em: 3/11/2018.

Cybernetics. Cambridge University Press, 2018. Disponível em: https://dictionary.cambridge.org/pt/dicionario/ingles-portugues/cybernetics. Acesso em: 11/9/2018.

APÊNDICE A – DOCUMENTAÇÃO DE API

A documentação de API do serviço Web Você Digital, na versão em que este trabalho é escrito, encontra-se disponível em: https://github.com/claudiomarpda/tcc. Acesso em: 9/11/2018.

APÊNDICE B – DEPENDÊNCIAS

As dependências utilizadas na aplicação servidor do projeto Você Digital são detalhadas abaixo. O trecho de código apresentado faz parte do arquivo *pom.xml* gerenciado pelo Maven.

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.0.5.RELEASE</version>
    <relativePath/>
</parent>
<properties>
    <project.build.sourceEncoding>
        UTF-8
    </project.build.sourceEncoding>
    <project.reporting.outputEncoding>
        UTF-8
    </project.reporting.outputEncoding>
    <java.version>1.8</java.version>
    <jwt.version>0.9.1</jwt.version>
</properties>
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <scope>runtime</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework.restdocs</groupId>
        <artifactId>spring-restdocs-mockmvc</artifactId>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-thymeleaf</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
```

```
            <artifactId>spring-boot-starter-hateoas</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-security</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.security</groupId>
            <artifactId>spring-security-test</artifactId>
        </dependency>
        <dependency>
            <groupId>io.jsonwebtoken</groupId>
            <artifactId>jjwt</artifactId>
            <version>${jwt.version}</version>
        </dependency>
    </dependencies>
```