UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Rhaniel Cristhian Borges Miranda

Arquitetura de referência para construção, validação e implantação de serviços de integração entre sistemas

Uberlândia, Brasil

2018

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Rhaniel Cristhian Borges Miranda

Arquitetura de referência para construção, validação e implantação de serviços de integração entre sistemas

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, Minas Gerais, como requisito exigido parcial à obtenção do grau de Bacharel em Sistemas de Informação.

Orientador: William Chaves de Souza Carvalho

Universidade Federal de Uberlândia – UFU

Faculdade de Ciência da Computação

Bacharelado em Sistemas de Informação

Uberlândia, Brasil

2018

Rhaniel Cristhian Borges Miranda

Arquitetura de referência para construção, validação e implantação de serviços de integração entre sistemas

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, Minas Gerais, como requisito exigido parcial à obtenção do grau de Bacharel em Sistemas de Informação.

William Chaves de Souza Carvalho
Orientador

Prof. Pedro Moisés de Souza
Universidade Federal de Viçosa

Lorena Melo Guimarães Rosa
Fundação Getulio Vargas

Uberlândia, Brasil
2018

Este trabalho é dedicado a todos os profissionais de tecnologia que já passaram por dificuldades durante o desenvolvimento de sistemas mal arquitetados

Agradecimentos

Agradeço primeiramente a Deus e à Bem-Aventurada Sempre Virgem Maria pelo cuidado e suporte que me foi dado durante o desenvolvimento do trabalho. Agradeço também a minha querida mãe e irmã pelo incentivo e apoio, à minha namorada pelo carinho e compreensão nas horas complicadas, aos amigos e familiares, e por fim, de maneira especial ao meu orientador Prof. Me. William Chaves de Souza Carvalho.

"Comece fazendo o que é necessário, depois o que é possível, e de repente você estará fazendo o impossível."

São Francisco de Assis

Resumo

Este documento tem como objetivo apresentar uma arquitetura de referência para construção de serviços que permitam a comunicação entre vários sistemas de forma transparente, avaliado por ferramentas de *code review* e implantados de forma automática, gerando assim um código eficiente e de fácil entendimento. Para demonstrar a eficiência do modelo apresentado, foi desenvolvido uma API REST utilizando a linguagem JAVA e o framework Spring Boot, além disso, o exemplo desenvolvido foi submetido a uma avaliação feita por uma ferramente de *code-review* e implantada em um servidor Linux de forma automática utilizando o Jenkins.

Palavras-chave: Arquitetura de Referência, APIs, REST, RESTful, Serviços de integração.

Lista de ilustrações

Trechos de Códigos

Lista de abreviaturas e siglas

AOP	Aspect Oriented Programming
API	Application Programming Interface
CSS	Cascading Style Sheets
DOM	Documents Objects Models
FTP	File Transfer Protocol
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
HTTPS	Hyper Text Transfer Protocol Secure
JPA	Java Persistence API
JSON	Javascript Object Notation
MVC	Model-View-Controller
ORM	Object-Relational Mapping
REST	Representational State Transfer
SMS	Short Message Service
SOAP	Simple Object Access Protocol
SOs	Sistemas Operacionais
SQL	Structured Query Language
TI	Tecnologia da Informação
URI	Uniform Resource Identifier
XML	Extensible Markup Language
YAML	Ain't Markup Language

Sumário

1 Introdução

Vive-se em um mundo amplamente conectado, dispositivos recebendo e enviando informações, empresas preocupadas em obter dados para seus mainframes, pessoas criando novos aplicativos o tempo todo. Tudo isso precisa se comunicar de alguma forma. Pensar em uma linguagem de programação comum se torna inviável dado a gama de possibilidades para desenvolvimento, sem contar que cada linguagem tem suas limitações e dependendo da finalidade, uma se torna melhor do que a outra.

Dessa forma, quando um desenvolvedor se depara com a necessidade de criar serviços de integração que viabilizem a troca de dados, ele, geralmente, não tem à sua disposição um guia ou mesmo um manual de boas práticas de como fazer todas as etapas de construção desses mecanismos. Ou seja, o programador precisa pesquisar como implementar esta troca de dados; e a situação fica ainda mais complexa quando processos de *code review* e de *deployment* também devam ser considerados, pois esses recursos quase sempre não são encontrados juntos.

1.1 Motivação

Deve-se lembrar que este problema não é enfrentado por uma parcela irrelevante da comunidade de profissionais de TI. Muito pelo contrário! Esta problemática é cotidianamente enfrentada por parcela significativa de desenvolvedores de software; que cada qual ao seu jeito, o resolve ou contorna de acordo com o caso. Disso decorre retrabalho, atrasos, perda de competitividade, queda na qualidade e, por conseguinte, diminuição das vantagens competitivas da empresa. Neste cenário, propor alternativas suficientemente genéricas para criar tais serviços de comunicação é condição determinante para que se invistam tempo e recursos para sua concepção.

1.2 Objetivos e Desafios da Pesquisa

O objetivo deste trabalho é documentar uma proposta de arquitetura de referência para a construção de serviços de integração, *deployment* automatizado e *code reviews* para softwares escritos em Java que visam atender aos vários canais de comunicação[1] de uma empresa do setor bancário, envolvendo tanto seus clientes externos quanto os internos.

Pensando nesse problema surgiu o conceito API (*Application Programming Interface*), esse conceito nada mais é do que padrões que um software cria para prover seus

[1] Exemplos de canais são: internet e *mobile banking*, atendimento a lojistas e portadores de cartão de débito e crédito.

serviços sem que o utilizador precise codificar todo o corpo da função desejada. Com o conceito de API introduzido, surgiram elementos complementares para que a comunicação global entre plataformas com SOs, arquiteturas e linguagens de programação diferentes pudessem se falar de maneira transparente, dentre eles o REST e o SOAP.

No ano de 1994 surgiu um mecanismo que inicialmente foi chamado de modelo de objetos HTTP, esse recurso tinha o objetivo de realizar a comunicação de conceitos da WEB quando se desenvolvia o HTTP/1.1, esse nome trazia consigo confusões com modelos de implementação para servidores HTTP então foi batizado de *Representational State Transfer* ou REST.

Com o surgimento de APIs REST se tornou possível a troca de informações entre esses vários sistemas totalmente diferentes com um esforço mínimo e sem que haja necessidade de um conhecimento profundo dos demais sistemas. A comunicação basicamente feita por protocolos HTTP e HTTPS (GET, POST, PUT, PATCH, etc). A informação trafega em formato JSON, XML, YAML ou texto puro, o que facilita para quem está recebendo os dados.

Durante o processo de desenvolvimento de qualquer sistema é necessário que se tenha um bom código-fonte, seguindo os padrões para cada linguagem, para facilitar o processo de manutenção, evitar bugs e vulnerabilidades. Para isso, existem várias ferramentas que fazem a inspeção contínua da qualidade do código-fonte. Neste trabalho será apresentado uma ferramenta openSource desenvolvida pela SonarSource, o SonarQube. Ele suporta várias linguagens de programação como JAVA, C#, PHP, JavaScript, etc. É possível configurar várias métricas e gráficos para validação e representação do código. Além disso, o SonarQube permite a integração com plataformas de integração contínua como Atlassian Bambo, Jenkins, etc.

Após toda a implementação e revisão do código, é preciso publicar o sistema em algum servidor para que os serviços fornecidos por ele, seja consumido por outros sistemas. Dependendo da quantidade de implantações necessárias durante o dia, essa tarefa se torna extremamente problemática, pois, se o processo for feito de maneira erronia o servidor pode ficar fora do ar e impactar outras serviços publicadas no mesmo lugar. Para resolver esse problema será apresentado uma ferramenta de *deploy* automatizado chamada Jenkins.

> *"Jenkins is an open source automation server which enables developers around the world to reliably build, test, and deploy their software."* (JENKINS, 2018).

Essa plataforma permite que builds e *deploys* sejam disparados automaticamente através de commits feitos no git ou de maneira manual. O Jenkins permite a criação de scripts shell e comandos de lote do Windows tudo com o intuito de fornecer uma maneira

para se fazer o build e o *deploy* automatizado dos fontes evitando assim um problemas de falhas humanas.

1.3 Contribuições

Com a utilização desse documento qualquer desenvolvedor com conhecimentos básicos em programação orientada a objetos será capaz de desenvolver serviços de comunicação, utilizar ferramentas de *code review* e por fim construir um fluxo de *deploy* automático criando assim um ambiente de integração contínua.

1.4 Estrutura da Monografia

Para cumprir seus objetivos, o trabalho será dividido da seguinte maneira:

- O primeiro capítulo se refere à Introdução do trabalho.
- O segundo capítulo trata do referencial teórico utilizado para embasar a pesquisa.
- O terceiro capítulo, por sua vez, apresenta trabalhos correlatos ao tema da pesquisa.
- O quarto capítulo descreverá a implementação da arquitetura de referência, considerando a utilização do Sonar e do Jenkins.
- O quinto capítulo apresenta as considerações finais, trabalhos futuros, desafios encontrados durante a pesquisa e conclusões.

2 Referencial Teórico

2.1 Arquitetura de software

Antes de tudo é importante entender o que é uma arquitetura de software. O livro Software Architecture in Practice (BASS; CLEMENTS; KAZMAN, 2003) define arquitetura de software como sendo as estruturas do sistema que compreendem elementos de software, propriedades externamente visíveis desses elementos e as relações entre eles. David Garlan (GARLAN, 2000) diz ainda que arquitetura de software é a estrutura de componentes de um sistema, os relacionamentos entre esses componentes, e as diretrizes que governam os projetos e a evolução dos softwares. Mas por que é importante definir uma arquitetura de softwares?

De acordo com o livro (BASS; CLEMENTS; KAZMAN, 2003) existem três principais motivos para se definir arquiteturas. A primeira delas é a comunicação que uma arquitetura de software oferece, ou seja, cada parte de interesse de um software está preocupada com aspectos diferentes do mesmo, por exemplo o cliente está preocupado com o custo para implementar a arquitetura já o desenvolvedor está preocupado em como construir o sistema e sem essa arquitetura de software definida as diferentes partes de interesse encontram problemas se entenderem.

O segundo motivo é que com uma arquitetura de software definida é possível definir o design do sistema mais rapidamente, estas decisões inicias são difíceis de serem mudadas mais tarde. É com essa arquitetura de software que se define restrições de implementação, inibição ou permissão de atributos de qualidade no sistema como por exemplo: desempenho, midificabilidade, segurança, escalabilidade, etc.

E o terceiro motivo é a abstração do sistema que seja reutilizável. Isso significa que um modelo de arquitetura define como um sistema vai ser estruturado e como seus elementos trabalham juntos, isso pode ser transferível para um sistema com exigências parecidas, proporcionando assim uma vantagem imensa durante a construção de novos softwares.

Dessa forma, quando a organização usualmente constrói softwares do mesmo domínio de negócio ou com funcionalidades parecidas, é recomendável que o reúso arquitetural seja estimulado, e isso acontece, sobretudo, pelo uso de arquiteturas de referências, as quais serão descritas na próxima seção.

2.2 Arquitetura de Referência

Ainda sim existem várias diferenças entre arquitetura de software e arquitetura de referência, o livro (BASS; CLEMENTS; KAZMAN, 2003) define arquitetura de referência com sendo um modelo de referência mapeado em elementos de software que implementam de forma cooperativa as funcionalidades já definidas e os fluxos de dados entre eles. Uma arquitetura de referência se diferencia da arquitetura de software por ser mais ampla, ou seja, ela possui um nível alto de abstração e contempla mais atributos de qualidade. Além disso, ela pode envolver, também, a definição e pré-configuração das tecnologias que serão utilizadas nos projetos.

Outro ponto que deve ser abordado por uma arquitetura de referência são as melhores práticas de desenvolvimento e decisões arquiteturais. Logo o papel da arquitetura de referência, de acordo com (BASS; CLEMENTS; KAZMAN, 2003), é de proporcionar um ponto de partida para que as organizações consigam criar suas arquiteturas de software. O livro (BASS; CLEMENTS; KAZMAN, 2003) ainda diz que uma arquitetura de referência é uma maneira eficaz de se lidar com requisitos não-funcionais e por isso cabe ao arquiteto decidir qual se encaixa melhor no contexto do projeto desenvolvido.

2.3 Representational State Transfer

No ano 2000 o cientista Roy Fielding publicou sua tese de Ph.D (FIELDING; TAYLOR, 2000) com o objetivo de formalizar algumas melhores práticas denominadas*constraints*. Essas *constraints* determinavam como padrões HTTP e URI deveriam ser modelados para que se pudesse aproveitar ao máximo os recursos oferecidos.

1. Cliente-Servidor: a primeira *constraint* definada por Roy Fielding foi a de cliente-servidor, isso nada mais é do que separar responsabilidades de cada parte do sistema, ou seja, camada de interação com o usuário separada do *backend*, essa *constraint* garante a escalabilidade do sistema.

2. Stateless: significa que qualquer requisição feita ao servidor não pode depender de nenhuma outra feita anteriormente ou futuramente, logo essa caracteística afirma que todas as informações necessárias já devem estar contidas na requisição.

3. Cache: essa característica diz que a aplicação precisa ser passível de cache, isso permite que o servidor processe apenas tarefas realmente necessárias. Cada resposta deve, de alguam forma, conseguir informar para o cliente qual a política de cache mais adequada a ser usada.

4. Interface Uniforme: interface uniforme sem dúvida é uma das mais importantes. Deve ser mantido um padrão em termos de recursos, mensagens e corretude nos

códigos de erros retornados. Por exemplo no HTTP é importante se fazer a utilização correta dos métodos e códigos de retorno disponíveis.

5. Sistema em camadas: um sistema em camadas permite a escalabilidade do software. Isso permite que seja possível acrescentar elementos intermediários de forma transparente para os clientes, por exemplo, balanceadores.

6. Código sob demanda: código sob demanda prega a possibilidade de adaptação do cliente de acordo com novos requisitos e funcionalidades, algo parecido com extensibilidade.

Obedecendo todas essas *constraints* e mais algumas características, que serão abordadas posteriormente, Fielding (FIELDING; TAYLOR, 2000) diz que temos uma API REST.

Representational State Transfer foi definido por Fielding (FIELDING; TAYLOR, 2000) como sendo uma abstração dos elementos arquitetônicos dentro de um sistema de hipermídia distribuído. O REST ignora detalhes de implementação e de sintaxe do protocolo e se preocupa com as funções dos componentes, nas restrições das suas interações e na interpretação de elementos de dados significativos. Ele engloba as restrições sobre os componentes, conectores e dados que definem a base da arquitetura WEB.

Deste modo quando uma organização decide desenvolver mecanismos de comunicação, utilizando a liguagem JAVA e os conceitos propostos por Fielding (FIELDING; TAYLOR, 2000), ela se depara com uma quantidade imensa de pré-configurações que são necessárias para a construção, por isso, algumas optam por usar um *framework* chamado *Spring Boot* que será apresentado na próxima seção.

2.4 Spring Framework

O *Spring* é um *framework* criado por Rod Johnson e descrito no seu livro (JOHNSON, 2004), baseado nos padrões de projeto, inversão de controle[1], com o intuito de simplificar a programação em JAVA. O *Spring* facilita a injeção de dependências e a programação orientada a aspectos(AOP), sendo assim, o desenvolvedor apenas diz ao *framework* o que ele deseja usar. De acordo com a documentação do *Spring* (JOHNSON et al., 2004), o *framework* é modular, permitindo que se use apenas partes importantes para cada projeto. Ele oferece uma estrutura MVC com recursos completos que permitem a integração do seu software com a AOP.

Apesar de todos os benefícos que o *Spring Framework* oferece, uma grande dificuldade é encontrada pelos desenvolvedores ao utiliza-la em seus projetos. A ferramenta

[1] É um padrão de desenvolvimento de software onde a sequência de chamadas dos métodos fica invertida em relação a programação tradicional, logo, ela não é determinada diretamente pelo programador.(FOWLER, 2004)

tem uma complexidade consideravel grande de configurações iniciais, até que se possa desfrutar dos seus benefícios, o que, acaba afastando o interesse dos programadores por ela. Por isso a *Spring* desenvolveu um projeto chamado *Spring Boot* que veio, justamente, para fazer essas pré configurações durante o processo de desenvolvimento de aplicações utilizando o *Spring Framework*. Isso atraiu os olhos das organizações para a ferramente, pois possibilita um desenvolvimento rápido e robusto.

2.5 Code Review

De acordo com a patente criada pela Microsoft (RACHAMADUGU; BENDAPUDI; JAIN, 2008), *code review* é uma análise feita do código-fonte para evitar *bugs* de desenvolvimento, códigos mal escritos e fora do padrão definido pela organização ou mesmo a comunidade. Esse processo pode ser feito por membros da equipe ou por ferramentas de revisão de código, o que é mais adequado pois evita a falha humana.

Durante o processo de criação da arquitetura de referência que será apresentada por esse trabalho, será aplicada no código-fonte uma ferramenta de *code review* conhecida como Sonarqube. O sonar é uma plataforma de código aberto desenvolvido pela SonarSource para fazer a análise do código de forma automática(CAMPBELL; PAPAPETROU, 2013).

Além disso é importante que se tenha uma forma de implantação da aplicação de forma segura e automática, para evitar problemas com implantações de artefatos errônios e facilitar o *rollback* em caso de falhas. Por isso, será apresentado na próxima seção, uma ferramenta para *deploy* automático chamada Jenkins.

2.6 Automated Deployment

Antes de apresentar a ferramente é importante entender o que é um *deploy* automático. De acordo com (LIMA, 2015), o processo de implantação de uma aplicação é o conjunto de atividades necessárias para que o mesmo possa estar pronto para ser usado pelo cliente. Esse processo, muitas das vezes, é bem trabalhoso e problemático dentro das organizações, por isso, criaram-se ferramentas para automatizar esse processo.

No final do desenvolvimento da arquitetura de referência para criação de mecanimos de comunição, será apresentado o Jenkins. Ele é uma ferramenta criada apartir do Hudson[2] que permite que uma aplicação seja compilada, testada e implantada em um servidor de forma automática, possibilitando assim uma velocidade e acertividade maior durante o processo de *deployment* de uma aplicação.

[2] Ferramenta de integração contínua.

3 Trabalhos Correlatos

Está seção apresenta trabalhos já publicados sobre arquiteturas de referências. Visto a vantagem de se ter arquiteturas de referências, diversos trabalhos já foram publicados abordando os mais variados temas dentro da computação. Ao final será apresentado as diferenças dos trabalhos já publicados para este trabalho em questão.

3.1 A Reference Architecture for Web Servers

No ano 2000 foi publicado pela Uiversidade de Waterloo o trabalho com título "*A Reference Architecture for Web Servers*"(HASSAN; HOLT, 2000) que apresenta uma arquitetura de referência para servidores WEB. O trabalho publicado define uma arquitetura de refência para três servidores web, são eles: Apache, AOL-Server e Jigsaw. De acordo com o artigo (HASSAN; HOLT, 2000), uma arquitetura de refência permite a reutilização de softwares e reduz os esforços para o desenvolvimento.

Um servidor web fornece acesso a recursos estáticos, como HTML e texto comum, ou até recursos dinâmicos, como *servelets* JAVA. Cada servidor web pode ter vários recursos extras que os diferenciam, porém, todos os servidores web devem prover arquivos de texto simples, o que gera uma semelhança entre todos, podendo assim, se construir uma arquitetura de referência. Então, a partir do código fonte dos três servidores de web já mencionados os pesquisadores criaram uma arquitetura de referência e publicaram no artigo (HASSAN; HOLT, 2000).

Na seção 3 do artigo(HASSAN; HOLT, 2000) é mostrado os passos usados para se construir a arquitetura de referência proposta. O primeiro passo foi derivar uma arquitetura conceitual para cada aplicação, esse passo foi sub-divido em outros dois.

1. Propor uma arquitetura conceitual para cada aplicação, utilizando o conhecimento e as documentações disponíveis.

2. refinar a arquitetura conceitual utilizando uma arquitetura concreta

E o segundo passo foi criar uma arquitetura de referência baseada nos modelos conceituais adquiridos nos passos anteriores. O segundo passo foi utilizado as mesmas técnicas dos sub-passos da etapa um, porém utilizando como base não mas o conhecimento e os documentos disponíveis e sim as arquiteturas conceituais já extraídas. Com isso foi possível criar uma arquitetura de referência para o desenvolvimento de servidores web.

O modelo de arquitetura de referência proposto foi demonstrado na seção 4, onde cada componente foi explicado detalhadamente. Ele ajuda na reutilização de um sistema web e na construção de novos servidores de web. A figura 1 representa o modelo sugerido pelo trabalho.

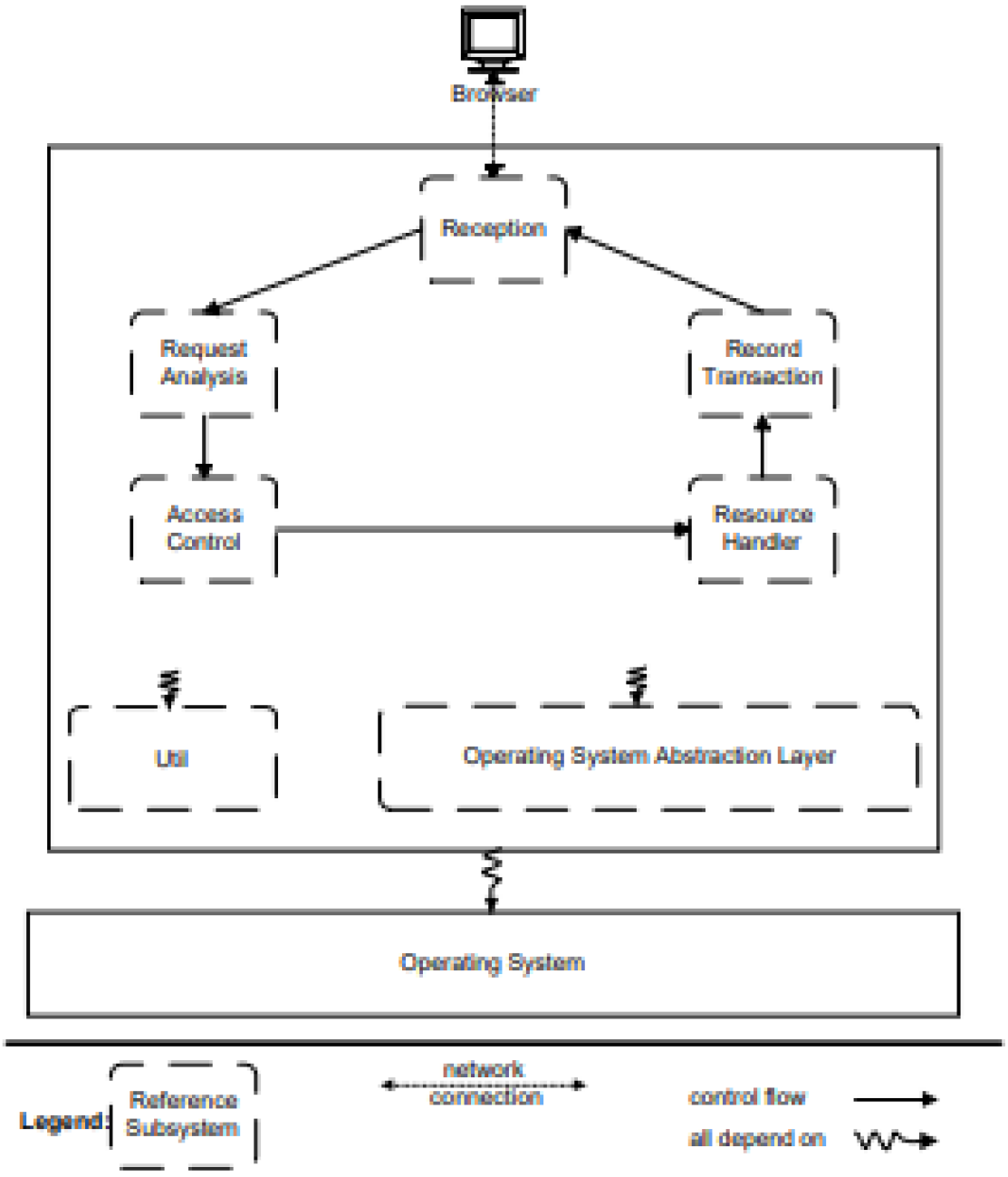

Figura 1 – Arquitetura de referência para *Web Server*. Extraída de (HASSAN; HOLT, 2000).

3.2 A reference architecture for web browsers

No ano de 2005 mais um trabalho sobre arquiteturas de referência foi publicado pela Universidade de Waterloo, dessa vez o tema abordado foi arquitetura de referência para *web browsers* com o título "*A reference architecture for web browsers*"(GROSSKURTH; GODFREY, 2005). O trabalho se baseou em dois sistemas de código aberto e após a definição da arquitetura de referência eles validaram os achados com cinco sistemas adicionais.

Os pesquisadores usaram o código fonte do Mozilla e do Konqueror para extrair semelhanças e desenvolver uma arquitetura de referência. A arquitetura de referência proposta é mostrada na figura 2, que foi extraida do trabalho (GROSSKURTH; GODFREY, 2005), ela possui oito subsistemas são eles:

1. Interface de Usuário: Essa é a camada que fica entre o utilizador e os mecanismos do navegador.

2. Motor de navegação: Esse mecanismo é um componente incorporado responsável por realizar tarefas primitivas como os botões de "voltar","avançar"e "recarregar".

3. Motor de renderização: Este recurso é responsável por fazer a representação visual do conteúdo, ou seja, exibir documentos HTML, CSS e imagens.

4. Subsistema de *Networking*: Responsável por implementar protocolos de transferência como HTTP e FTP.

5. Interpretador de JavaScript: Este recurso é responsável por interpretar códigos javaScripts que geralmente são usados para exibir janelas *pop-ups* entre outras funcionalidades.

6. Analisador XML: Esse subsistema faz a conversão de arquivos XML em Documents Objects Models (DOM).

7. *Display Backend*: Esse componente fornece desenhos e janelas primitivas para interação com a interface de usuário.

8. *Data Persistence*: É o subsistema responsável pelas barras de ferramentas, *cookies*, certificados de segurança ou *cache*.

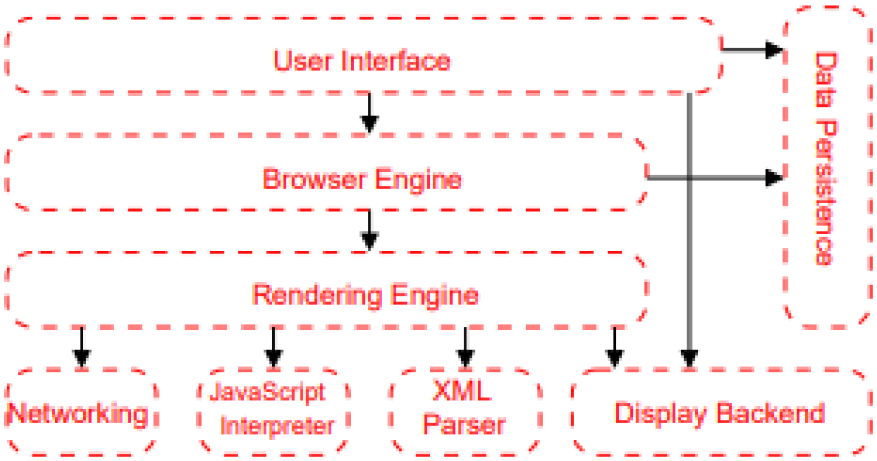

Figura 2 – Arquitetura de referência para navegadores web. Extraída de (GROSSKURTH; GODFREY, 2005).

De acordo com o trabalho(GROSSKURTH; GODFREY, 2005), a arquitetura de referência proposta pode servir como guia para manuntenção e para o desenvolvimento de novos sistemas. Durante as comparações dos cinco sistemas adicionais pode-se perceber que a arquitetura de referência proposta precisaria apenas de alguns ajustes para se encaixar com os sistemas usados, isso mostra que o modelo proposto pode ser usado como forma de documento para atuais e novos navegadores web.

3.3 Conclusão sobre trabalhos correlatos

Apesar de já terem sido publicados vários trabalhos sugerindo arquiteturas de referências, pouco foi feito sobre serviços de comunicação REST. Esse tipo de serviço pode ser considerado recente dentro das empresas o que acaba gerando grandes diferenças de uma pra outra. Então este trabalho irá apresentar uma arquitetura de referência para que se possa seguir um padrão dentro das organizações, facilitando assim a troca de serviços entre essas e também a manutenção dos mecanismo criados pelas mesmas.

Os trabalhos desenvolvidos e apresentados nessa seção, (HASSAN; HOLT, 2000) e (GROSSKURTH; GODFREY, 2005), se diferenciam deste pois não se trata de mecanismos de comunicação desenvolvidos em JAVA. Porém os objetivos são parecidos, todos buscam facilitar a manutenção e criação de sistemas utilizando dos benefícios, já mencionados, sobre possuir uma arquitetura de referência.

4 Desenvolvimento

O capítulo a seguir expõe na seção 4.1 à arquitetura de referência para construção de serviços de integração, seguido pelas subseções 4.1.1, 4.1.2, 4.1.3, 4.1.4, 4.1.5, 4.1.6 e 4.1.7, onde cada componente é apresentado de forma mais específica e com exemplos de implementação. Os exemplos serão desenvolvidos seguindo o padrão REST, utilizando a linguagem de programação JAVA e o *framework* Spring Boot, para facilitar o processo de configuração dos recursos. Logo após, na seção 4.2, é demonstrado o uso do SonarQube para validar o código fonte. Por fim, na seção 4.3 é apresentado o Jenkins como ferramenta para realizar *deploys* automatizados.

4.1 Modelo de Arquitetura de Referência

Uma arquitetura de referência é de extrema importância para o desenvolvimento de sistemas de alta performance, de fácil manuntenção e com alta capacidade de reúso. Dessa forma, o modelo de arquitetura de referência para construção de serviços de integração sugerido nesse trabalho, foi baseado em estudos feitos pela equipe de tecnologia e desenvolvimento do Banco Triângulo. A escolha dos componentes foi fundamentada em uma variação do padrão de projeto Model-View-Controller (MVC) que é difundido em todo o mundo. Além disso, o modelo final só foi definido após um longo período de experimentos e testes que avaliaram sua efetividade no processo de contrução, manutenção e reúso de recursos desse tipo.

A figura 3 representa o diagrama de componentes da arquitetura de referência sugerida por esse trabalho. O modelo é divido em 7 principais componentes, dessa forma, a estrutura permite o desenvolvimento seguindo padrões como SOAP, REST, WEB API, Socket, etc.

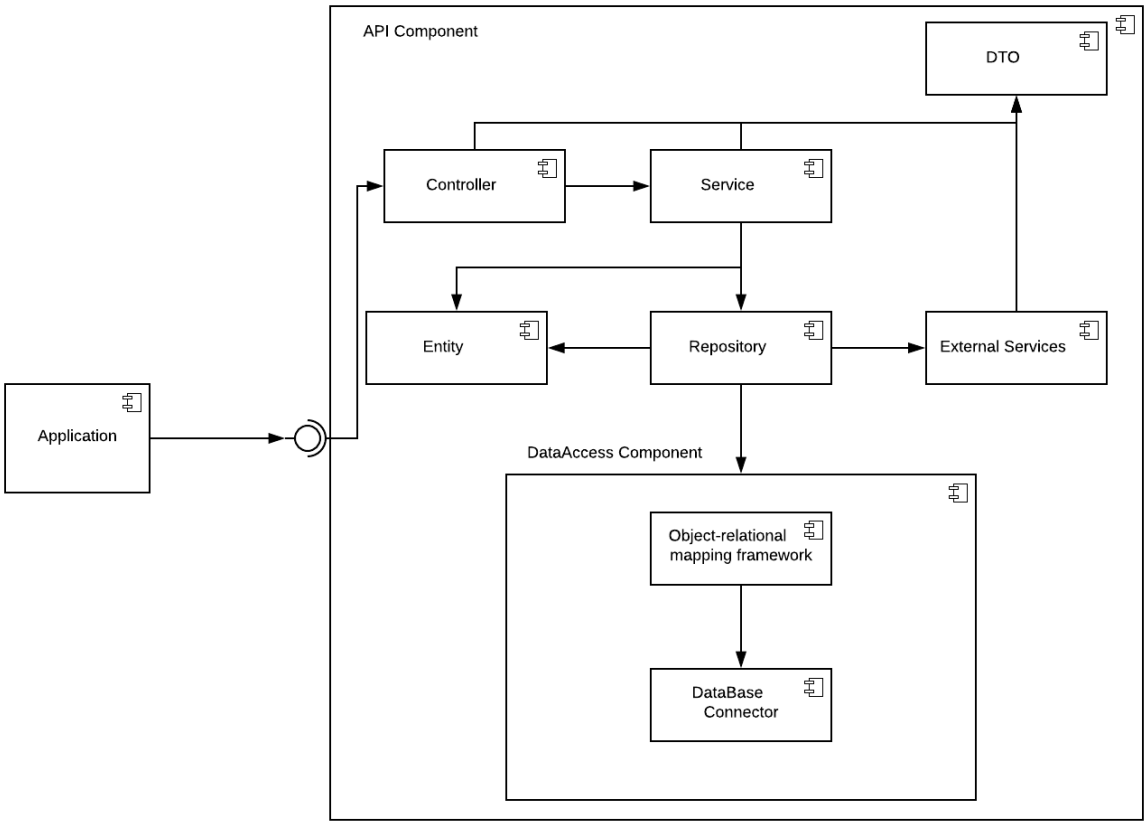

Figura 3 – Modelo arquitetura de referência para APIs.

4.1.1 Controller

O primeiro componente é o *controller*, ele é responsável por fazer a interface com o usuário, ou seja, é ele quem recebe os parâmetros de entrada, faz as devidas validações de dados, repassa para os próximas componentes e aguarda o processamento para fazer o retorno ao usuário. Esse componente não deve conter regras de negócio ou acesso a base de dados, ele deve conhecer somente os *Services* e os *DTOs*. O *controller*, se necessário, cria uma instância de DTO baseado nos parâmetros de entradas e repassa o objeto/parâmetros para a camada de serviço.

No *controller* podem ser expostos básicamente 8 métodos HTTP, são eles:

- GET
- POST
- PUT
- PATCH
- DELETE

- TRACE
- OPTIONS
- HEAD

O código representado no trecho de código 4.1 é uma exemplificação de implementação de *controller* com os métodos GET e POST implementados.

Trecho de Código 4.1 – Código JAVA para exemplificar implementação do *Controller*

```
1
2  @RestController
3  @RequestMapping("/cliente")
4  @Api(tags = { "API de Exemplo Trabalho de conclusão de curso" })
5  public class ExampleController {
6
7      //Instacia da classe de serviço
8      @Autowired
9      private ClienteService service;
10
11     @Autowired
12     public ExampleController() {
13         // Construtor padrao
14     }
15     /**
16      * Metodo GET que busca um cliente no banco baseado em seu CPF
17      *
18      * @param cep
19      * @return
20      */
21     @RequestMapping(method = RequestMethod.GET, value = "/{cpf}")
22     @ApiOperation(value = "Busca o cliente no banco")
23     public ResponseEntity<ConsultaClienteResponseDTO> findByCpf(
24             @ApiParam(value = "CPF do cliente") @PathVariable String cpf)
                {
25         return new ResponseEntity<>(service.consultaCliente(cpf),
               HttpStatus.OK);
26     }
27
28     /**
29     * Metodo POST responsavel por inserir um cliente
```

```
30      *
31      * @param cliente
32      * @return
33      **/
34      @RequestMapping(method = RequestMethod.POST)
35      @ApiOperation(value = "Insere o cliente no banco")
36      public ResponseEntity<Map<String,Object>> insereCliente(
37              @ApiParam(value = "Cliente a ser inserido") @RequestBody
                    ClienteDTO cliente) {
38
39          Map<String,Object> retorno = new HashMap<>();
40          if(service.insereCliente(cliente)){
41              retorno.put("sucesso",true);
42              retorno.put("mensagem", "Cliente inserido com sucesso");
43              return new ResponseEntity<>(retorno, HttpStatus.OK);
44          }
45          retorno.put("sucesso",false);
46          retorno.put("mensagem", "Cliente não foi inserido corretamente");
47          return new ResponseEntity<>(retorno, HttpStatus.
                INTERNAL_SERVER_ERROR);
48      }
49  }
```

4.1.2 DTO

O componente DTO vem do padrão de projeto *"Data Transfer Object"*, essa camada recebe os atributos passados pelo cliente ou os atributos que serão retornados para o cliente. Os DTOs são objetos simples com apenas os atributos e os devidos recursos de get e set.

Abaixo segue um exemplo de objeto JSON (4.2) enviado pelo cliente e o DTO (4.3) no qual esse objeto é desserializado, assim, as informações fornecidas pelo cliente podem ser manipuladas de forma mais eficaz.

Trecho de Código 4.2 – Objeto Json fornecido pelo cliente

```
1
2   {
3       "nrCpf": "00000000000",
4       "rg": "mg00000000",
5       "nmCompleto": "Rhaniel Cristhian Borges Miranda",
```

```
    "nmPai": "Odevnio Lemos Miranda",
    "nmMae": "Enilda Borges Ferreira",
    "dsEmail": "rhaniel@ufu.br",
    "nrTelefone": "34999999999",
    "sexo": "M",
    "idade": 21
}
```

Trecho de Código 4.3 – Código JAVA para exemplificar implementação de DTO

```
@Data
@AllArgsConstructor
public class ClienteDTO{

    private String nrCpf;
    private String rg;
    private String nmCompleto;
    private String nmPai;
    private String nmMae;
    private String dsEmail;
    private String nrTelefone;
    private String sexo;
    private int idade;
}
```

4.1.3 Service

O componente *service* é reponsável por encapsular as regras de negócio e fazer o orquestramento entre *controllers* e *repositories*. É importante lembrar, que os *services* não devem conter acesso a base de dados ou a recursos externos, essas atividades são de responsabilidade dos *repositories*, que serão explicados posteriormente. Esse tipo de separação permite que o as regras de negócio sejam testadas de forma mais abrangente.

A camada de serviço pode utilizar as *entities* para preparar um objeto para ser persistido na base ou para receber um objeto já persistido. O trecho de código 4.4 representa um exemplo de implementação de *service* no qual é solicitado a camada de *repository* um objeto persistido e um segundo recurso para persistir um objeto.

Trecho de Código 4.4 – Código JAVA para exemplificar implementação do *Service*

```
@Log4j2
@Service
```

```
3  public class ClienteService {
4
5      @Autowired
6      private ClienteApiRepository repository;
7
8      @Autowired
9      public ClienteService() {
10         // Construtor padrao
11     }
12
13     public ClienteEntity findByCpf(String cpf) {
14         if (!StringUtils.isNumeric(cpf)) {
15             throw new BusinessException("CPF deve ser numerico!");
16         }
17
18         log.debug("Consultando CPF: {}", cpf);
19         return repository.findByCpf(cpf);
20     }
21
22     public boolean insereCliente(ClienteDTO cliente) {
23         try {
24             ClienteEntity clienteEntity = new ClienteEntity(cliente.
                   getNrCpf(),
25                 cliente.getRg(),
26                 cliente.getNmCompleto(),
27                 cliente.getNmPai(),
28                 cliente.getNmMae(),
29                 cliente.getDsEmail(),
30                 cliente.getNrTelefone(),
31                 cliente.getSexo(),
32                 cliente.getIdade()
33             );
34             repository.save(clienteEntity);
35             return true;
36         } catch (Exception e) {
37             log.catching(e);
38             return false;
39         }
40     }
```

```
41 }
```

4.1.4 Repository

O componente *repository* é responsável por fazer a abstração para o acesso ao banco de dados, é nessa camada onde as operações necessárias a base são feitas. O repository faz o mapeamento da base para as entidades e a persistências das entidades para o banco de dados, conforme representado no trecho de código 4.5.

Além disso, é nos repositórios que devem ser declaradas as assinaturas dos métodos que consomem algum recurso externo, por exemplo, um serviço de geolocalização ou envio de SMS. Esse tipo de cenário está exemplificado no trecho 4.6.

Trecho de Código 4.5 – Código JAVA para exemplificar implementação do *Repository*

```
1 @Repository
2 public interface ClienteRepository extends CrudRepository<ClienteEntity,
     String> {
3
4 }
```

Trecho de Código 4.6 – Código JAVA para exemplificar implementação de *Repository* que consome recursos externos

```
1 @Repository
2 @FunctionalInterface
3 public interface EnvioSmsRepository {
4     public SmsRetorno enviaSms(SmsDTO sms);
5 }
```

4.1.5 Entity

A *entity* representa os objetos do banco de dados mapeados em classes, ou seja, funcionam como espécies de espelhos da base. Essa camada, possui apenas os atributos e seus devidos recursos de *get/set* com as referências para as colunas das consultas feitas na base.

Trecho de Código 4.7 – Código JAVA para exemplificar implementação de uma *Entity*

```
1 @Entity
2 @Table(name = "TBL_CLIENTE")
3 @Data
4 @AllArgsConstructor
```

```
5  public class ClienteEntity implements Serializable{
6
7      private static final long serialVersionUID = 1L;
8
9      @Id
10     @Column(name = "NR_CPF")
11     private String nrCpf;
12
13     @Column(name = "RG")
14     private String rg;
15
16     @Column(name = "NM_COMPLETO")
17     private String nmCompleto;
18
19     @Column(name = "NM_PAI")
20     private String nmPai;
21
22     @Column(name = "NM_MAE")
23     private String nmMae;
24
25     @Column(name = "DS_EMAIL")
26     private String dsEmail;
27
28     @Column(name = "NR_TELEFONE")
29     private String nrTelefone;
30
31     @Column(name = "SEXO")
32     private String sexo;
33
34     @Column(name = "IDADE")
35     private int idade;
36 }
```

4.1.6 External Services

O componente *external services* encapsula as chamadas feitas a serviços externos a aplicação, ou seja, chamadas REST, SOAP ou outras integrações. Essa camada deve ser conhecida apenas pela camada de repositório explicada na seção 4.1.4.

O trecho de código 4.8 implementa o repositório definido no trecho 4.6, já que a

camada de serviço não conhece este componente.

Trecho de Código 4.8 – Código JAVA para exemplificar implementação de um *External Service*

```
1  @Repository
2  public class EnvioSmsRepositoryImpl implements EnvioSmsRepository{
3
4      private static final Logger LOG = LogManager.getLogger();
5
6      private static final String CONTENT_TYPE = "Content-Type";
7      private static final String TOKEN_ID = "token";
8      private static final String APPLICATION_JSON = "application/json";
9
10     @Value("${api.envioSms.url}")
11     private String url;
12
13     @Value("${api.envioSms.token}")
14     private String token;
15
16     @Override
17     public SmsRetorno enviaSms(SmsDTO sms) {
18         try {
19             RestTemplate restTemplate = new RestTemplate();
20             HttpHeaders headers = new HttpHeaders();
21             headers.set(CONTENT_TYPE, APPLICATION_JSON);
22             headers.set(TOKEN_ID, token);
23
24             Map<String, String> requestBody = new HashMap<>();
25
26             requestBody.put("msg", sms.getMsg());
27             requestBody.put("destinatario", sms.getDestinatario());
28             requestBody.put("from", sms.getFrom());
29             requestBody.put("dtEnvio", new SimpleDateFormat("yyyy-MM-dd HH
                   :mm:ss")
30                     .format(new Date()));
31
32             HttpEntity<Map<String, String>> request = new HttpEntity<>(
                   requestBody, headers);
33
34             return restTemplate.exchange(url, HttpMethod.POST, request,
```

```
                SmsRetorno.class)
35                  .getBody();
36          } catch (Exception e) {
37              LOG.error(e);
38              return null;
39          }
40      }
41  }
```

4.1.7 Data Access Component

Este componente é responsável por fazer o acesso a base de dados, é através dele que a aplicação se conecta no banco e realiza as operações necessárias. Nessa arquitetura de referência o componente é formado por dois sub-componentes que serão explicados nas sub-seções 4.1.7.1 e 4.1.7.2.

O trecho de código 4.9 representa uma configuração feita do componente abrangendo seus dois sub-componentes. Nesse exemplo, muitas configurações são feitas de forma automática através do *framework* utilizado para desenvolver os exemplos (Spring Boot).

4.1.7.1 Object-Relational Mapping Framework

O componente *Object-Relational Mapping* também conhecido como ORM é usado para fazer a conversão das tabelas e objetos do banco de dados para as entidades nas linguagens orientadas a objetos. Além disso, esse componente geralmente encapsula as configurações do componente *database connector* que será explicado na seção 4.1.7.2. Nos exemplos fornecidos nesse trabalho foi usado o *Spring Data JPA* que utiliza o ORM chamado de Hibernate e *Java Database Connectivity* como conector.

A principal vantagem de se usar um ORM é a redução da quantidade e complexidade do código usado para fazer as consultas a base. A figura 4 mostra o funcionamento de um ORM.

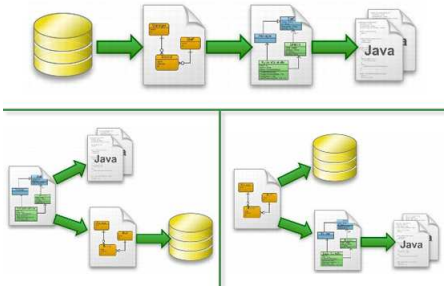

Figura 4 – Funcionamento ORM (DEVMEDIA, 2011)

4.1.7.2 Database Connector

O *database connector* é uma camada que encapsula o *driver* de conexão do banco de dados. Esse componente deve possuir recursos para realizar a conexão, criar e executar instruções SQLs. Se tratando da linguagem de programação JAVA o *database connector* utilizado é o JDBC, porém cada linguagem possuí o seu próprio conector.

A figura 5 foi extraída do site da Oracle e representa a arquitetura definida para o JDBC.

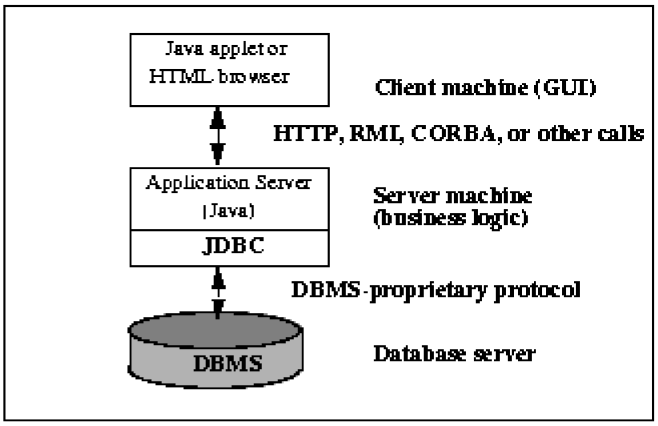

Figura 5 – Arquitetura JDBC (CORPORATION,)

Trecho de Código 4.9 – Código JAVA para exemplificar implementação do *Data access component*

```
1  @Profile("!test")
2  @Configuration
3  @EnableTransactionManagement
4  @EnableJpaRepositories(//
5          basePackages = "br.com.sample.spring.db.sql.repository", //
6          entityManagerFactoryRef = "sql-em", //
7          transactionManagerRef = "sql-tm")
8  public class SqlTrc01Config {
9
10     @Value("${datasource.sql.url}")
11     private String url;
12     @Value("${datasource.sql.username}")
13     private String username;
14     @Value("${datasource.sql.password}")
15     private String password;
16     @Value("${datasource.sql.class-name}")
17     private String driverClassName;
```

```
18      /**
19       * Factory para criação do Data Source
20       *
21       * @return
22       */
23      @Bean(name = "sql-ds")
24      public DataSource dataSourceFactory() {
25          return DataSourceBuilder.create() //
26                  .driverClassName(driverClassName) //
27                  .username(username) //
28                  .password(password) //
29                  .url(url) //
30                  .build();
31      }
32      /**
33       * Factory para criação do Entity Manager
34       *
35       * @param builder
36       * @return
37       */
38      @PersistenceContext(unitName = "TRC01")
39      @Bean(name = "sql-em")
40      public LocalContainerEntityManagerFactoryBean entityManagerFactory(
41              EntityManagerFactoryBuilder builder) {
42          return builder.dataSource(dataSourceFactory()).persistenceUnit("
            TRC01").properties(jpaProperties())
43                  .packages("br.com.sample.spring.db.sql.entity").build();
44      }
45      /**
46       * Factory para criação do Transaction Manager
47       *
48       * @param em
49       * @return
50       */
51      @Bean(name = "sql-tm")
52      public PlatformTransactionManager transactionManagerFactory(
53              @Qualifier("sql-em") EntityManagerFactory em) {
54          return new JpaTransactionManager(em);
55      }
```

```
56      /**
57       * Propriedades do Persistence Unity
58       *
59       * @return
60       */
61      private Map<String, Object> jpaProperties() {
62          Map<String, Object> props = new HashMap<>();
63          props.put("hibernate.dialect", "org.hibernate.dialect.
                SQLServer2005Dialect");
64          return props;
65      }
66  }
```

4.2 SonarQube

A qualidade de um código fonte é baseada em um certo padrão definido, previamente, pela comunidade e/ou organização responsável pelo desenvolvimento. Apesar de uma avaliação de código ser subjetiva existem varias métricas que são aceitas universalmente pela comunidade como um padrão de desenvolvimento. Essa tarefa pode ser feita manualmente por um individuo responsável por checar linha por linha do código fonte, porém não é a forma mais indicada e nem mesmo mais eficaz.

Dito isso, para resolver o problema esse trabalho sugere que durante o desenvolvimento de uma API baseada na arquitetura de referência sugerida na seção 4.1, através da figura 3, seja utilizada uma ferramente de verificação do código fonte chamada SonarQube. Essa ferramente faz toda a validação de forma automática e com métricas pré-estabelecidas que seguem um padrão global, ainda assim, ela permite que essas métricas sejam alteradas e desabilitadas além de possibilitar a criação novos indicadores de acordo com a necessidade do usuário.

Antes fazer a verificação do código fonte, é necessário instalar e configurar o SonarQube. Os passos estão descritos abaixo:

1. Primeiro é necessário fazer o download do SonnarQube através do link e o *unzip* do arquivo para o diretório desejado.

2. Definir *<directorySonnarQube>*/bin para variável de ambiente PATH conforme demonstrado na figura 6.

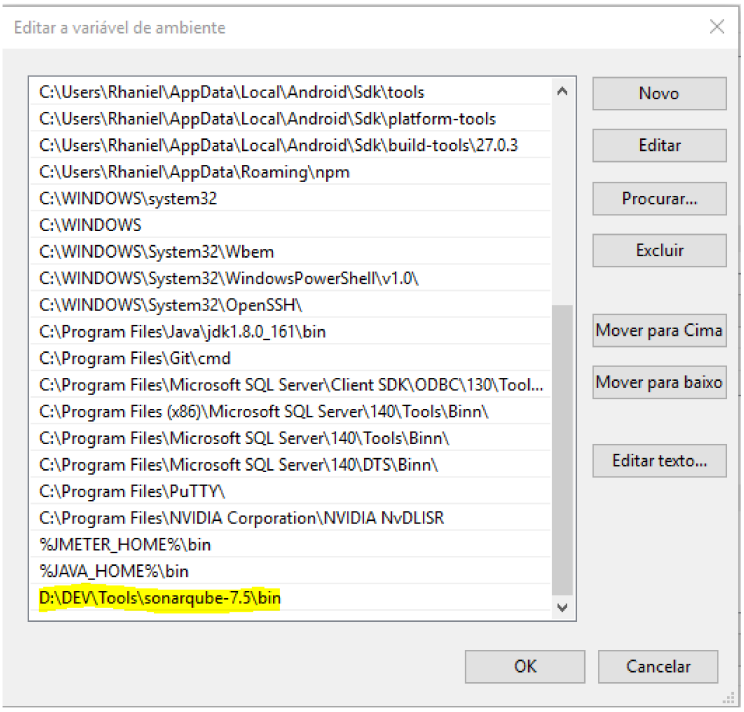

Figura 6 – Variáveis de Ambiente

3. Iniciar o servidor do SonarQube através do executável disponível em *<directorySonarQube>*/bin/(Diretório que corresponde ao seu SO)/StartSonar.bat. Para verificar se o servidor do sonar iniciou com sucesso, basta acessar o endereço http://localhost:9000, e uma página igual a figura 7 deve ser aberta.

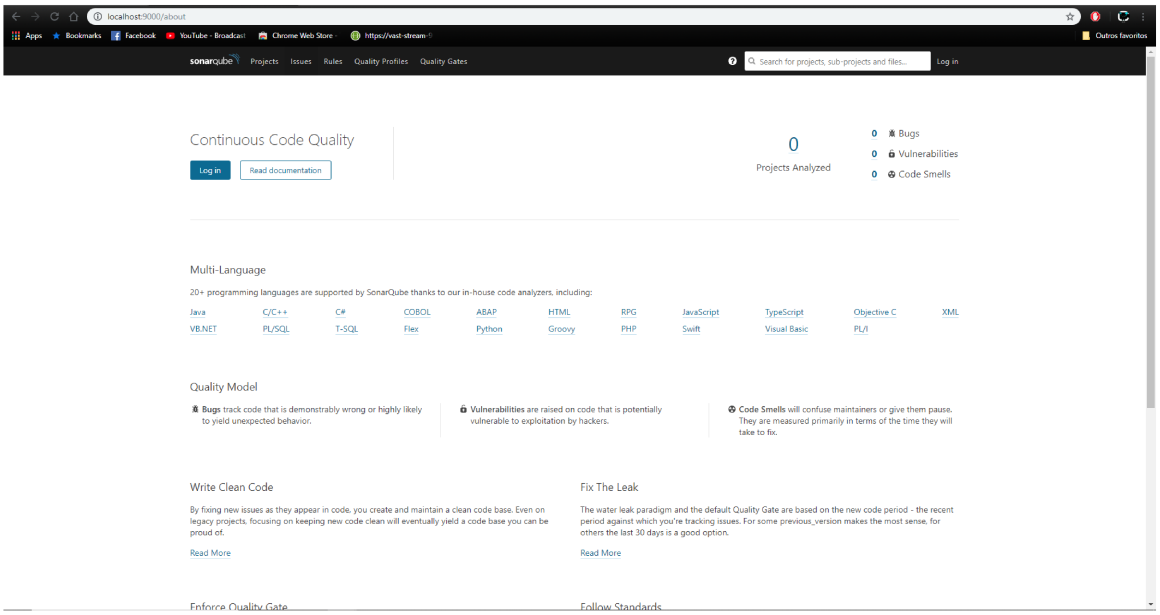

Figura 7 – Página WEB SonarQube

4. Entrar na raiz do projeto e criar um arquivo com o nome "sonar-project.properties"semelhante ao da figura 8. Nesse arquivo podem ser definidas mais propriedades de acordo com

a documentação disponível no site SonarQube.

```
# metadados requeridos
sonar.projectKey=exemplo-sonar-tcc-rhaniel
sonar.projectName=tcc\\exemplo-sonar-tcc-rhaniel
sonar.projectVersion=0.1

# caminhos podem ser separados por vírgula
sonar.sources=src/main/java
sonar.tests=src/test/java

# A linguagem do projeto
sonar.language=java

# Formato
sonar.sourceEncoding=UTF-8
```

Figura 8 – Arquivo de propriedades projeto sonar

5. Abrir o terminal, navegar até a raiz do código fonte que se deseja fazer a validação e executar o comando **mvn sonar:sonar -Dsonar.host.url=http://localhost:9000**. Se tudo ocorrer bem, a saída do terminal deve ser parecida com a da figura 9

```
C:\WINDOWS\system32\cmd.exe
[INFO] 1/1 source files have been analyzed
[INFO] Sensor XML Sensor [xml]
[INFO] Sensor XML Sensor [xml] (done) | time=79ms
[INFO] Sensor Analyzer for "php.ini" files [php]
[INFO] Sensor Analyzer for "php.ini" files [php] (done) | time=3ms
[INFO] Sensor Zero Coverage Sensor
[INFO] Sensor Zero Coverage Sensor (done) | time=21ms
[INFO] Sensor CPD Block Indexer
[INFO] Sensor CPD Block Indexer (done) | time=25ms
[INFO] No SCM system was detected. You can use the 'sonar.scm.provider' property to explicitly specify it.
[INFO] 11 files had no CPD blocks
[INFO] Calculating CPD for 10 files
[INFO] CPD calculation finished
[INFO] Analysis report generated in 239ms, dir size=97 KB
[INFO] Analysis reports compressed in 75ms, zip size=59 KB
[INFO] Analysis report uploaded in 630ms
[INFO] ANALYSIS SUCCESSFUL, you can browse http://localhost:9000/dashboard/index/br.com.arquitetura:sample-spring-busine
ss-api
[INFO] Note that you will be able to access the updated dashboard once the server has processed the submitted analysis r
eport
[INFO] More about the report processing at http://localhost:9000/api/ce/task?id=AWgb8J2eVINnquLw23zQ
[INFO] Task total time: 4.749 s
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 7.071 s
[INFO] Finished at: 2019-01-05T00:56:20-02:00
[INFO] ------------------------------------------------------------------------

D:\DEV\GIT\sample-spring-business-api-master-2ec7c1d54d95f31cbb630420b5f10d31838aba21>
```

Figura 9 – Saída console sucesso ao executar validações

6. Ao voltar para o site http://localhost:9000, perceba que o componente "*Projects Analyzed* "agora está definido para 1. Ao clicar no componente uma nova janela é aberta com as informações dos projetos analisados semelhante a figura 10, para obter mais informações dos problemas encontrados, basta clicar no projeto que uma nova página é aberta com mais detalhes da análise conforme indicado na figura 11.

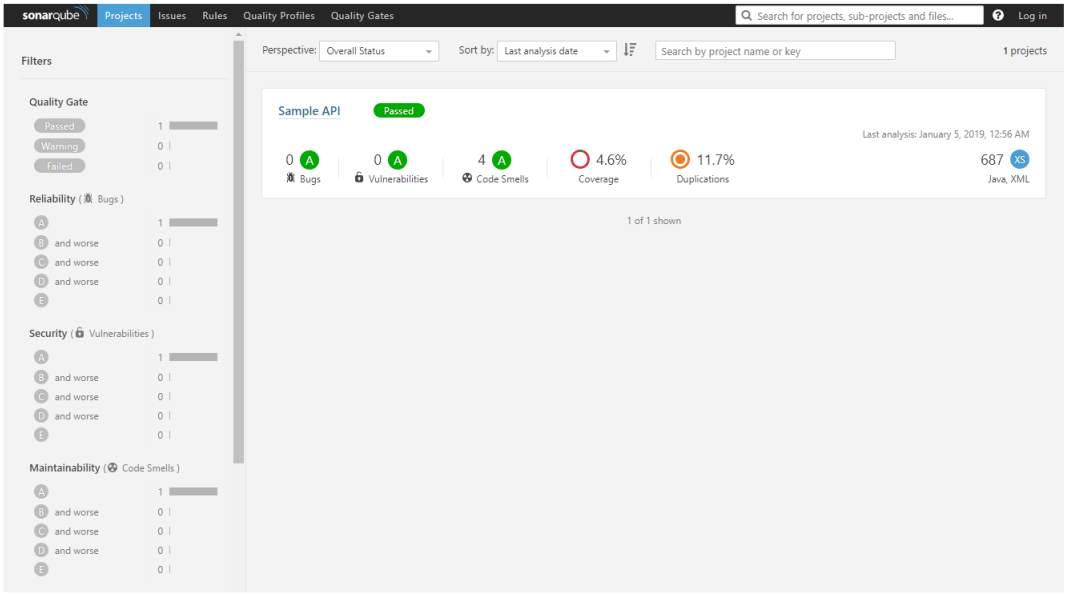

Figura 10 – Página projetos analisados

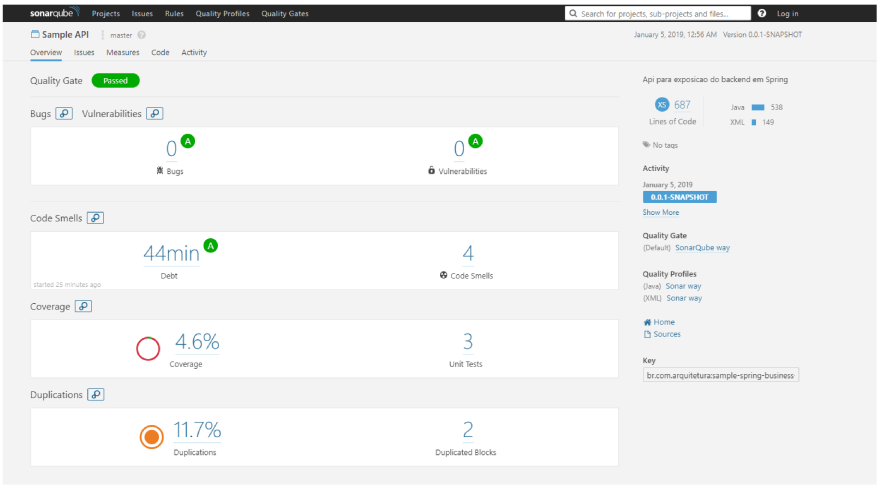

Figura 11 – Página com detalhes dos projetos analisados

4.3 Jenkins

Após todo o processo de construção dos recursos, respeitando o modelo de arquitetura de referência proposta e utilização do sonar para avaliar o código fonte escrito, ainda assim é preciso implantar o pacote gerado em algum servidor e iniciar sua execução. Essa tarefa pode ser feita de forma manual ou utilizando ferramentas específicas que fazem a construção e a implantação do pacote.

Fazer essa tarefa de forma manual deixa o fluxo com vulnerabilidades e suscetivo falhas, por isso, este trabalho sugere o uso do Jenkins como ferramenta auxiliar para automação de *builds* e *deploys*. O Jenkins, permite a integração com ferramentas de ar-

mazenamento de código, como por exemplo o GIT, possibilitando que a cada *commit*[1] feito um *build* ou mesmo um *deploy* seja feito de forma automática. Outra funcionalidade da ferramente é a possibilidade de fazer integrações com o SonarQube explicado na seção 4.2.

O diagrama representado na figura 12, ilustra o funcionamento do Jenkins e suas integrações para uma possível rotina sugerida por este trabalho.

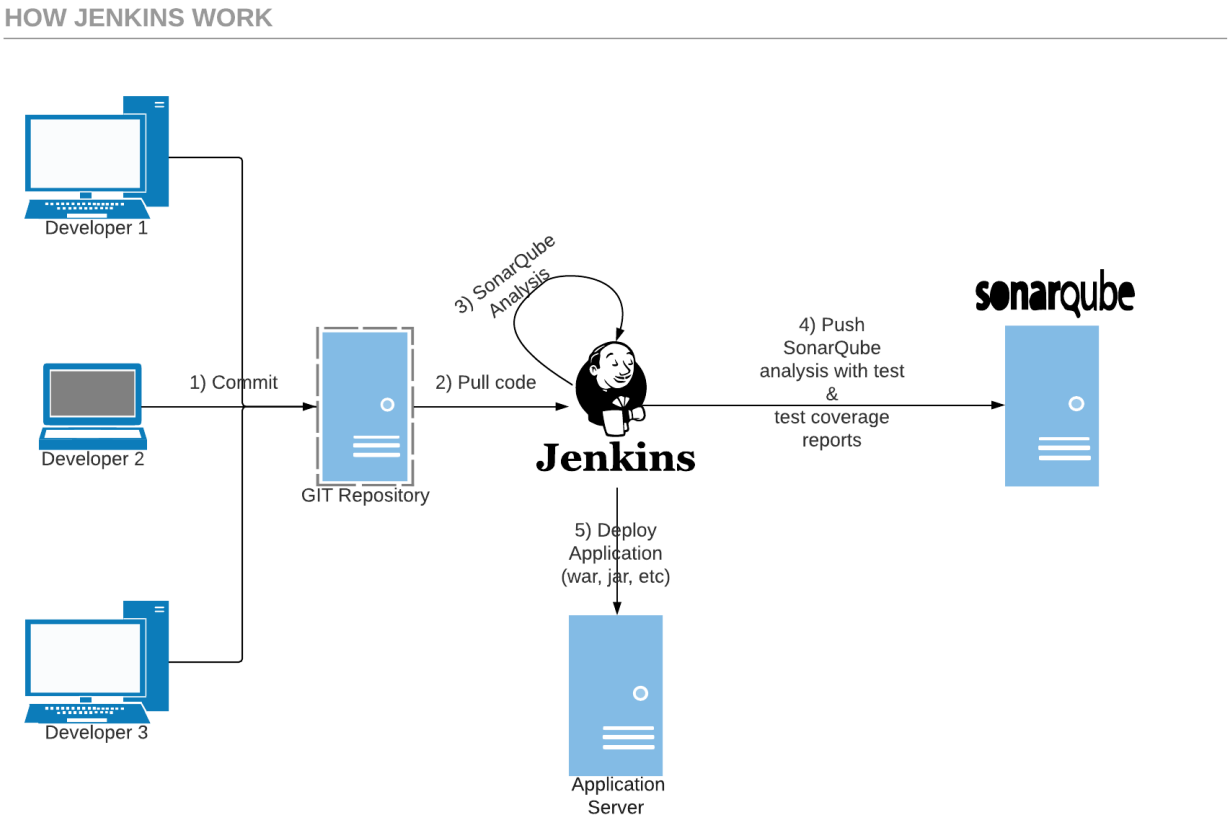

Figura 12 – Representação de possível rotina no Jenkins

[1] Confirmação de um conjunto de alterações feitas no código fonte

5 Conclusão

Atualmente pessoas e organizações necessitam trocar informações há todo tempo, e por isso, surge a necessidade de construir mecanismos que facilitam a integração de sistemas diferentes, e algumas vezes até desconhecidos. Dessa forma, os desenvolvedores responsáveis por construir esses serviços, em sua grande maioria, não possuem uma arquitetura de referência para que seus serviços possam ser de fácil entendimento e com grande taxa de reuso. Por isso, o intuito deste trabalho foi disponibilizar uma arquitetura de referência para construção de serviços que viabilizem a comunicação entre sistemas distintos, validadas por ferramentas de *code review* e implantadas de forma automatizada.

Durante o desenvolvimento deste trabalho muitos desafios foram encontrados. Primeiro, foi definir a arquitetura de referência representada na figura 3, para isso, foi necessário acompanhar durante 6 meses o desenvolvimento de APIs em um banco na cidade de Uberlândia, até chegar em um modelo simples e completo.

Após a definição do modelo de arquitetura de referência, a falta de experiência com *shell script*, gerou dificuldades para instalar e iniciar o SonarQube de forma correta.

Todos esses desafios encontrados foram oportunidades para estudo e aprimoramento do conhecimento. A persistência permitiu a criação de um modelo de arquitetura simples e robusta que viabiliza a construção de serviços bem arquitetados, com códigos bem escritos e implantados de forma automática, evitando erros e gerando agilidade. Tudo isso possibilita uma comunidade de tecnologia da informação mais equalizada e sistemas que possam ser entendidos globalmente.

5.1 Trabalhos Futuros

Após a definição conceitual do modelo de arquitetura de referência, surge a possibilidade de desenvolver um trabalho que implementa de forma detalhada o modelo sugerido. Além disso, esse trabalho não contempla, especificamente, a configuração de métricas no SonarQube nem configurações de rotinas no Jenkins, isso possibilita o desenvolvimento de trabalhos que abranjam esses temas.

Referências

BASS, L.; CLEMENTS, P.; KAZMAN, R. *Software architecture in practice*. [S.l.]: Addison-Wesley Professional, 2003. Citado 2 vezes nas páginas 15 e 16.

CAMPBELL, G.; PAPAPETROU, P. P. *SonarQube in action*. [S.l.]: Manning Publications Co., 2013. Citado na página 18.

CORPORATION, O. *Three-tier Architecture for Data Access*. Disponível em: <https://docs.oracle.com/javase/tutorial/jdbc/overview/architecture.html>. Citado 2 vezes nas páginas 7 e 33.

DEVMEDIA. *Object Relational Mapper work*. 2011. Disponível em: <https://www.devmedia.com.br/orm-object-relational-mapper/19056>. Citado 2 vezes nas páginas 7 e 32.

FIELDING, R. T.; TAYLOR, R. N. *Architectural styles and the design of network-based software architectures*. [S.l.]: University of California, Irvine Doctoral dissertation, 2000. v. 7. Citado 2 vezes nas páginas 16 e 17.

FOWLER, M. Inversion of control containers and the dependency injection pattern. 2004. Citado na página 17.

GARLAN, D. Software architecture: a roadmap. In: ACM. *Proceedings of the Conference on the Future of Software Engineering*. [S.l.], 2000. p. 91–101. Citado na página 15.

GROSSKURTH, A.; GODFREY, M. W. A reference architecture for web browsers. In: IEEE. *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*. [S.l.], 2005. p. 661–664. Citado 3 vezes nas páginas 7, 21 e 22.

HASSAN, A. E.; HOLT, R. C. A reference architecture for web servers. In: IEEE. *Reverse Engineering, 2000. Proceedings. Seventh Working Conference on*. [S.l.], 2000. p. 150–159. Citado 4 vezes nas páginas 7, 19, 20 e 22.

JENKINS. *Jenkins Press Information*. 2018. Disponível em: <https://jenkins.io/press/#about-jenkins>. Citado na página 13.

JOHNSON, R. *Expert one-on-one J2EE design and development*. [S.l.]: John Wiley & Sons, 2004. Citado na página 17.

JOHNSON, R. et al. The spring framework–reference documentation. *Interface*, v. 21, p. 27, 2004. Citado na página 17.

LIMA, R. C. *Automatizando a geração de pacotes com o Jenkins*. 2015. Disponível em: <https://www.devmedia.com.br/automatizando-a-geracao-de-pacotes-com-o-jenkins/33655>. Citado na página 18.

RACHAMADUGU, R.; BENDAPUDI, P.; JAIN, M. *Integrated code review tool*. [S.l.]: Google Patents, 2008. US Patent App. 11/754,231. Citado na página 18.