



Departamento de Polícia Rodoviária Federal

Projeto: Servo2

Nota Técnica

DPRF	DPRF Segurança - Nota técnica	
-------------	--------------------------------------	--

Revisão	Descrição	Autor	Data
1.0	Construção do documento	Israel Branco	19/05/2020

1 Sumário

2 Considerações iniciais.....	4
3 Apresentação do cenário atual.....	5
3.1 Tecnologias utilizadas.....	8
4 Análise técnica.....	9
4.1 SonarQube.....	9
4.2 OWASP Dependency Check.....	10
4.3 OWASP ZAP.....	11
4.4 Estrutura do projeto.....	12
4.5 Manutenibilidade de código.....	13
4.6 Confiabilidade.....	14
4.7 Performance e estabilidade.....	14
5 Recomendações.....	16

2 Considerações iniciais

Este documento visa reportar o resultado da análise efetuada na aplicação DPRF Segurança. Para este estudo foram desconsiderados todo o contexto negocial ao qual a ferramenta está inserida, também foram desconsideradas o ambiente ao qual a ferramenta esta operando sendo analisado puramente questões que tangem a qualidade de código, padrões de codificação, vulnerabilidades de dependências, modelo relacional de banco de dados e concepção arquitetural.

Para a realização desta análise, gerou-se a tag *ctis-nota-tecnica* no repositório <https://git.prf/prf/servo2> com referência branch master na data de 13/05/2020.

3 Apresentação do cenário atual

Esta sessão ira descrever a arquitetura, tecnologias, frameworks e dependências que compõe a base da aplicação.

O sistema DPRF Segurança foi construído para funcionar em ambiente WEB, utiliza tecnologia Java e está estruturada arquiteturalmente como uma aplicação monolítica (entende-se por este termo quando o sistema é composto por camadas de interface com usuário, camada de aplicação de regras negociais e camada de acesso a dados combinadas em uma única aplicação), utiliza o banco de dados *PostgreSQL*.

Sua arquitetura é composta por um único artefato que contém classes controladoras, classes de persistencia, classes para mapeamento ORM, classes utilitárias, classes para expor API's Rest, arquivos xhtml, css e javascript.

O diagrama a seguir representa o modelo de componentes ao qual a aplicação está construída.

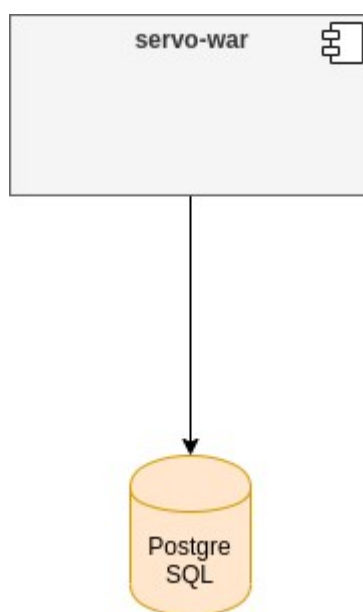


Figura 1: Diagrama de componentes



A aplicação utiliza o modelo MVC para a segregação de responsabilidades em camadas, as requisições http são oriundas das páginas JSF e endpoints Rest. O diagrama a seguir representa os fluxos encontrados durante o processo de análise da aplicação, o diagrama representa também a falta de padronização comportamental da mesma.

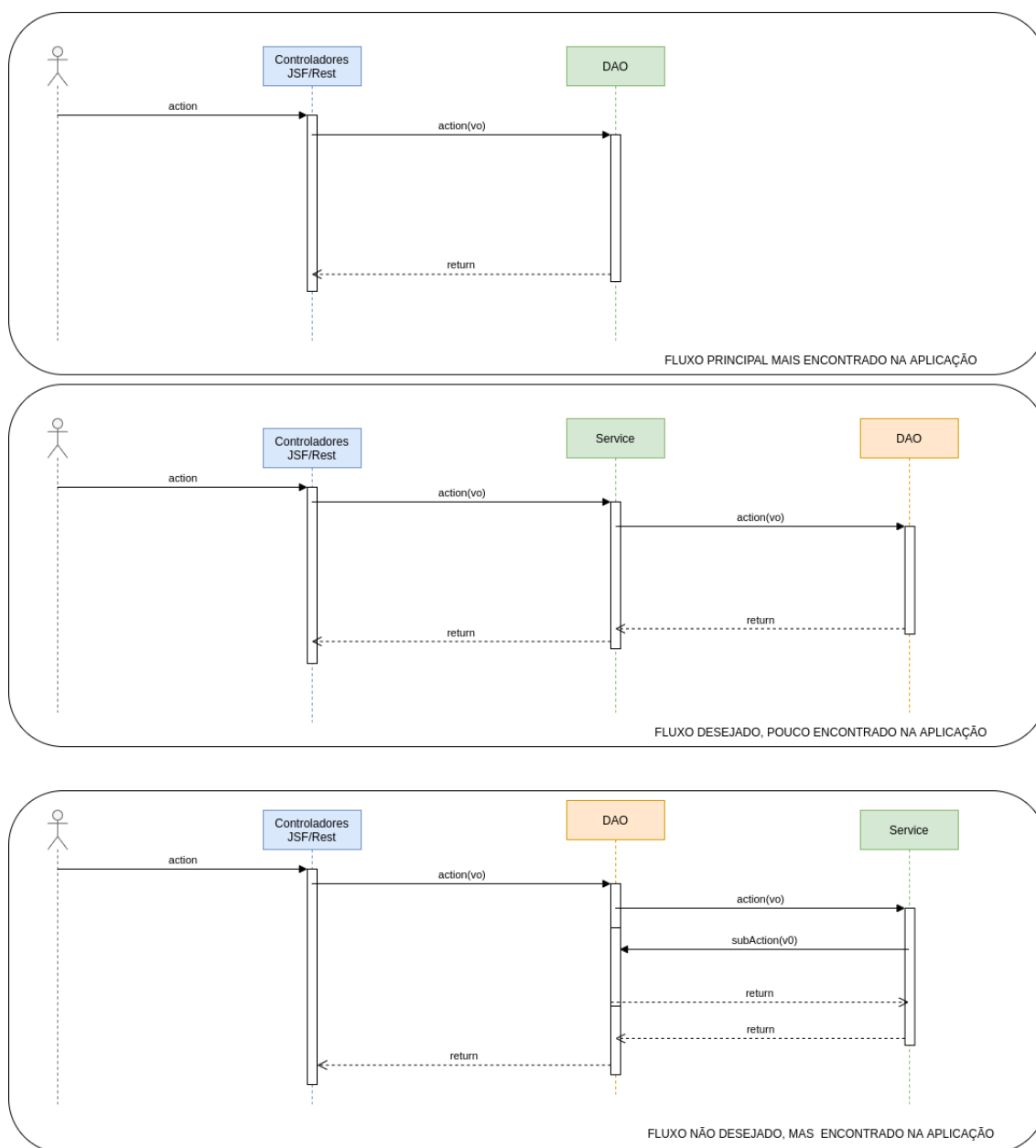


Figura 2: Diagrama de sequências



A solução utiliza um único schema com 59 tabelas.



Figura 3: MER - db sistemas comum H schema dbserve

3.1 Tecnologias utilizadas

Esta sessão descreve as tecnologias, frameworks e principais bibliotecas utilizadas na construção do projeto, descrevendo versões e propósitos de utilização.

Nome	Versão	Utilização	Observação
Java	1.8	Linguagem de programação.	
Hibernate	5.1.5	Framework ORM.	
Primefaces	6.2	Extensão de componentes JSF	
Apache POI	3.17	Biblioteca para trabalhar com documentos padrão Office.	
Wildfly	10.x	Servidor de aplicação JEE.	Utiliza container CDI e Servlet.
PostgreSQL		Banco de dados relacional	

4 Análise técnica

Este tópico descreve a ferramenta do ponto de vista técnico, tanto nos aspectos de codificação, análise estática de código, análise de vulnerabilidade de dependências e particularidades de implementação.

4.1 SonarQube

Ferramenta utilizada para verificação de estática de código. Para esta análise não foram utilizadas as métricas de qualidade implantadas no SonarQube da DPRF, contudo foram utilizadas as regras padrões de análise da ferramenta.

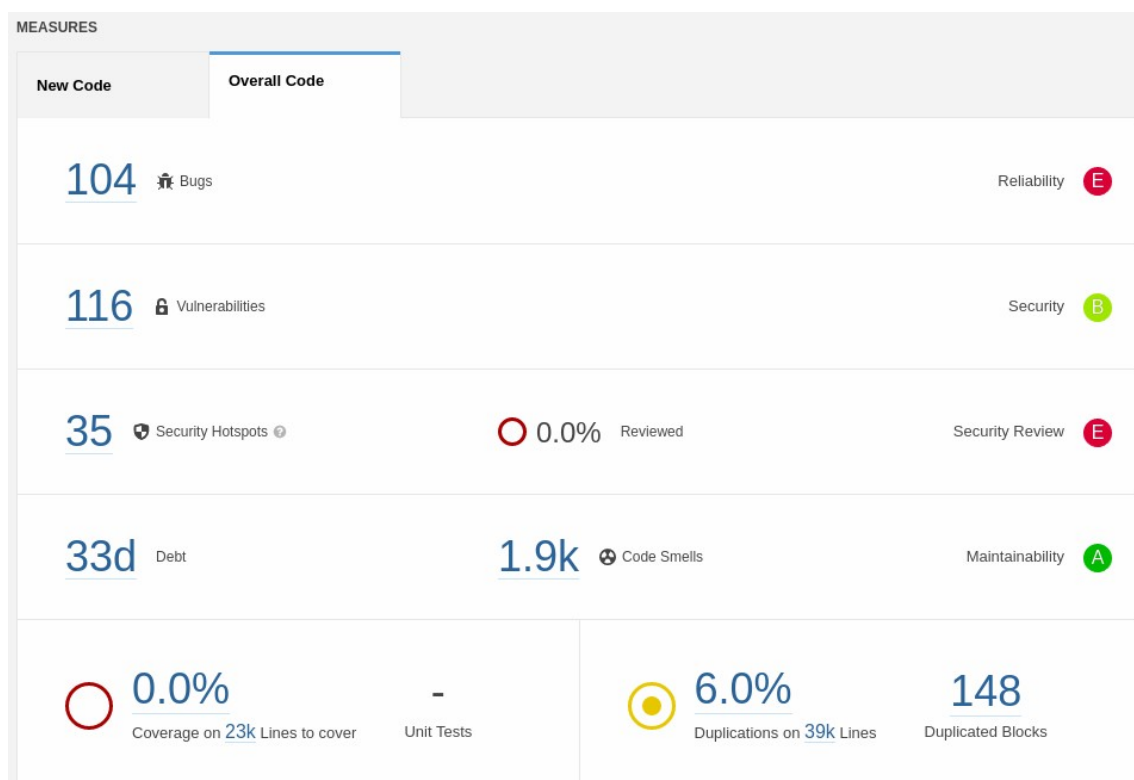


Figura 4: Sonarqube - análise estática de código

DPRF	DPRF Segurança - Nota técnica	
-------------	--------------------------------------	--

- 104 bugs;
- 116 vulnerabilidades de código;
- 35 violações de segurança;
- 1900 violações de código ruim (complexidade cognitiva , complexidade ciclomática e débito técnico);
- 6% de duplicidade de código

4.2 OWASP Dependency Check

A utilização de bibliotecas de terceiros aumenta substancialmente a produtividade na construção de um software, contudo estas podem trazer consigo vulnerabilidades que afetam diretamente a segurança da aplicação. A ferramenta Dependency Check tem como propósito efetuar análise de vulnerabilidade de dependências utilizadas na construção deste projeto, a seguir temos as principais informações extraídas desta análise, a relação completa desta análise está disponível no Anexo I deste documento.

Dependency	Highest Severity	CVE Count	Confidence	Evidence Count
commons-beanutils-1.8.3.jar	HIGH	2	Highest	35
bcprov-jdk14-138.jar	HIGH	16	Highest	23
hibernate-validator-5.1.0.Final.jar	MEDIUM	1	Highest	33
postgresql-9.4.1207.jar	HIGH	3	Highest	48
dom4j-1.6.1-jboss.jar	CRITICAL	2	Highest	25
poi-3.17.jar	MEDIUM	1	Highest	29
primefaces-6.2.RC2.jar	MEDIUM	1	Highest	27
cdi-api-1.2.jar	MEDIUM	1	Low	36
javax.faces-api-2.2.jar	MEDIUM	1	Highest	43
shiro-core-1.2.3.jar	CRITICAL	3	Highest	32
httpclient-4.2.6.jar	MEDIUM	2	Highest	34
commons-fileupload-1.3.jar	CRITICAL	4	Highest	36
jquery.js	medium	3		3
primefaces-6.2.RC2.jar: jquery.js	MEDIUM	2		3

4.3 OWASP ZAP

Ferramenta funciona como scanner de segurança, utilizada para realização de testes de vulnerabilidade de aplicações WEB. Atualmente trata-se de um dos projetos mais ativos na comunidade de software livre.

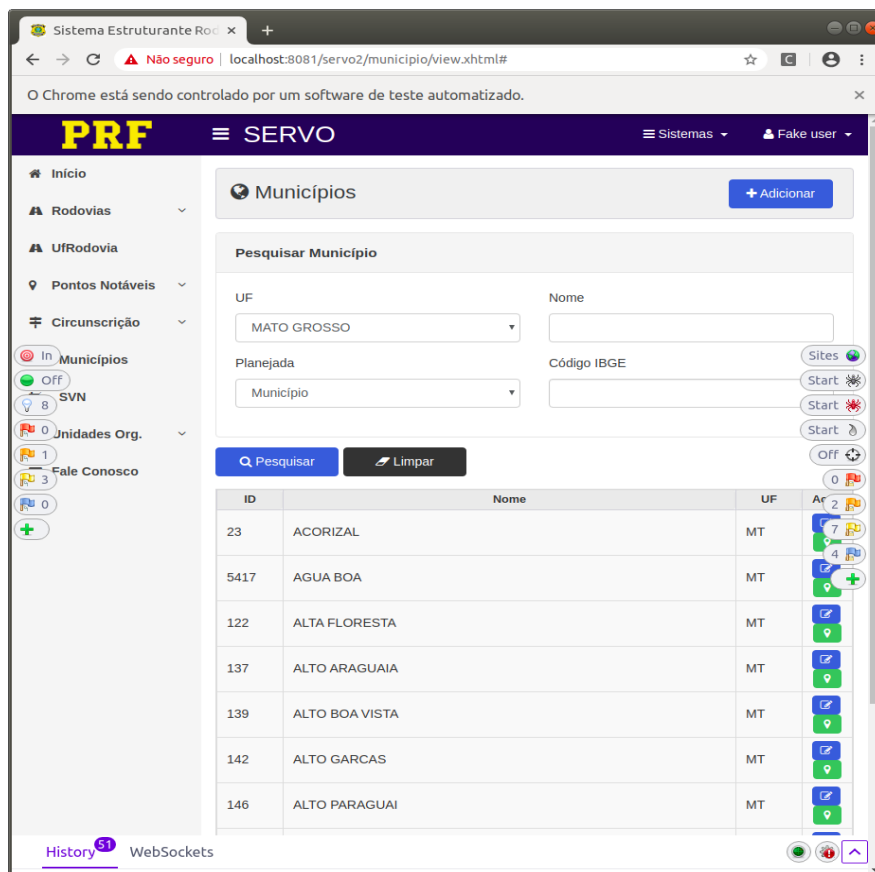


Figura 5: OWASP ZAP - análise de intrusão - Servo2

- 0 vulnerabilidade de severidade alta;
- 4 vulnerabilidade de severidade média;
- 20 vulnerabilidades de baixa média;
- 23 vulnerabilidades a nível informativo;

O relatório completo dos testes aplicados estão disponíveis no anexo I deste documento.

4.4 Estrutura do projeto

O projeto possui organização razoavelmente adequada, há segregação tanto por contexto comercial quando segregação por contexto funcional. Existem classes duplicidade de pacotes para organização dos Managed Beans, DAO e classes comerciais (VO/DTO).

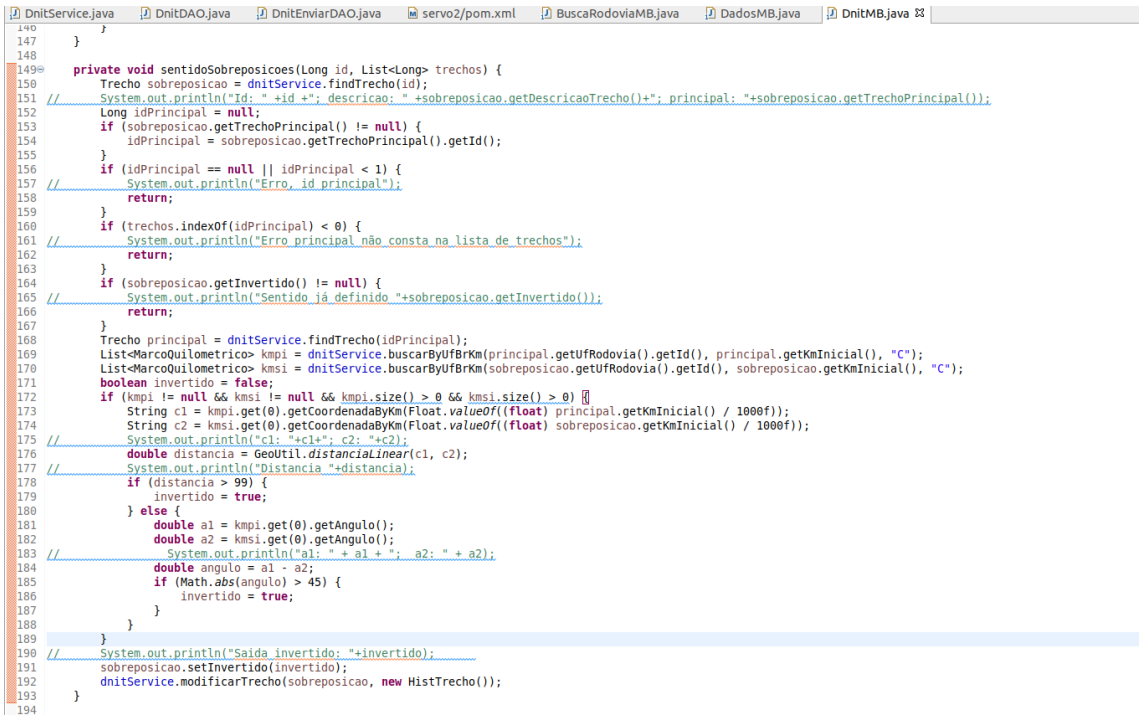


Figura 6: Organização do projeto

4.5 Manutenibilidade de código

Os relatórios apresentados pela ferramenta SonarQube demonstram uma série de vícios adotados durante o processo de construção do software e alinhado a estes vícios, a inexistência de cobertura de testes de unidade que trazem a dificuldade no processo de refactoring da aplicação, uma vez que não há condições de mensurar impactos durante o processo de manutenção corretiva/adaptativa.

A alta complexidade ciclométrica e a falta de artefatos de testes de unidade dificultam o processo de refactoring, a ilustração que seguem demonstram o cenário apontado (OBS: a característica apresentada é utilizada de forma recorrente em diversos momentos do código).



```

149 private void sentidoSobreposicoes(Long id, List<Long> trechos) {
150     Trecho sobreposicao = dnitService.findTrecho(id);
151     // System.out.println("Id: " + id + "; descricao: " + sobreposicao.getDescricaoTrecho() + "; principal: " + sobreposicao.getTrechoPrincipal());
152     Long idPrincipal = null;
153     if (sobreposicao.getTrechoPrincipal() != null) {
154         idPrincipal = sobreposicao.getTrechoPrincipal().getId();
155     }
156     if (idPrincipal == null || idPrincipal < 1) {
157         System.out.println("Erro, id principal");
158         return;
159     }
160     if (trechos.indexOf(idPrincipal) < 0) {
161         // System.out.println("Erro principal não consta na lista de trechos");
162         return;
163     }
164     if (sobreposicao.getInvertido() != null) {
165         // System.out.println("Sentido já definido " + sobreposicao.getInvertido());
166         return;
167     }
168     Trecho principal = dnitService.findTrecho(idPrincipal);
169     List<MarcoQuilometrico> kmpi = dnitService.buscarByUfBrKm(principal.getUfRodovia().getId(), principal.getKmInicial(), "C");
170     List<MarcoQuilometrico> kmsi = dnitService.buscarByUfBrKm(sobreposicao.getUfRodovia().getId(), sobreposicao.getKmInicial(), "C");
171     boolean invertido = false;
172     if (kmpi != null && kmsi != null && kmpi.size() > 0 && kmsi.size() > 0) {
173         String c1 = kmpi.get(0).getCoordenadaByKm(Float.valueOf((float) principal.getKmInicial() / 1000f));
174         String c2 = kmsi.get(0).getCoordenadaByKm(Float.valueOf((float) sobreposicao.getKmInicial() / 1000f));
175         // System.out.println("c1: " + c1 + "; c2: " + c2);
176         double distancia = GeoUtil.distanciaLinear(c1, c2);
177         // System.out.println("Distancia " + distancia);
178         if (distancia > 99) {
179             invertido = true;
180         } else {
181             double a1 = kmpi.get(0).getAngulo();
182             double a2 = kmsi.get(0).getAngulo();
183             // System.out.println("a1: " + a1 + "; a2: " + a2);
184             double angulo = a1 - a2;
185             if (Math.abs(angulo) > 45) {
186                 invertido = true;
187             }
188         }
189     }
190     // System.out.println("Saída invertido: " + invertido);
191     sobreposicao.setInvertido(invertido);
192     dnitService.modificarTrecho(sobreposicao, new HistTrecho());
193 }
194

```

Figura 7: Complexidade ciclométrica – Classe DnitMB

4.6 Confiabilidade

Há controle de transação com as operações em banco de dados na camada de acesso a dados (DAO – Data Access Object). Embora esta esteja sendo utilizado, esta prática é melhor utilizada quando aplicada a camada de serviço, uma vez que ali é possível garantir as propriedades ACID da transação como um todo. Esta prática é altamente recomendada principalmente quando há utilização de várias fontes de dados.

A manutenção da consistência de dados é algo fortemente desejado, contudo esta não garante toda a confiabilidade da solução. A quantidade elevada de bugs, vulnerabilidades no código e nas bibliotecas de terceiros encontradas nos relatórios apresentados trazem riscos a confiabilidade da ferramenta.

4.7 Performance e estabilidade

Não foi analisado o funcionamento da aplicação para avaliar demais requisitos não funcionais, recomenda-se a utilização de ferramentas de APM para mensurar performance e recursos de máquina utilizados.

A arquitetura monolítica citada no tópico 3 deste documento prejudica a escalabilidade da ferramenta, os recursos empreendidos para a escalabilidade vertical (aumento de recursos de processamento, disco, memória e demais) são limitados e onerosos.

A escalabilidade vertical do monólito é possível levando em consideração o aumento de nós no cluster, contudo esta escalabilidade é prejudicada tendo em vista que temos que escalar a aplicação como um todo, necessitando assim da mesma quantidade de recursos empreendidas nos demais nós existentes. Nesta arquitetura não há a possibilidade de escalar somente as

funcionalidades/módulos que mais são demandados.

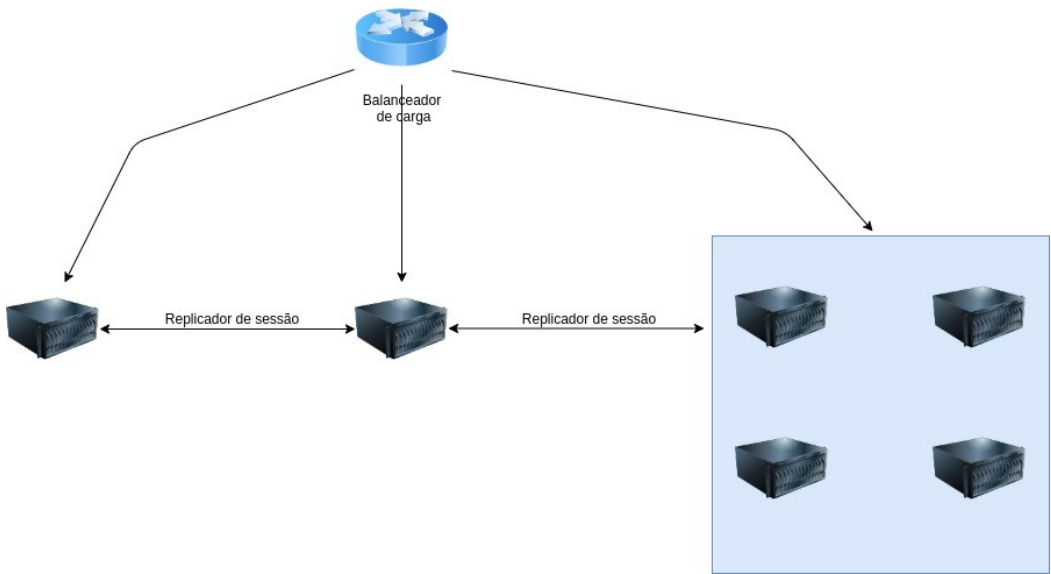


Figura 8: Escalabilidade do monólito

5 Recomendações

É altamente recomendado que seja efetuado refactoring de código dos bugs e vulnerabilidades de código apontadas pelo SonarQube , estas atividades certamente trarão maior confiabilidade a ferramenta e estabilidade em seu uso. Para os demais itens apontados pela ferramenta SonarQube durante o processo de análise de código são altamente desejáveis, contudo este processo de ajuste de código é moroso e trás consigo risco em potencial e está diretamente aliado a falta de cobertura de testes de unidade.

Ajustar as dependências que trazem maior risco para a aplicação é altamente recomendável, contudo este trabalho deve ser feito de forma analítica e cautelosa afim de não prejudicar a estabilidade da ferramenta. Sugere-se a associação dos relatórios de análise de dependências com os relatórios de análise de intrusão para que sejam analisados as principais vulnerabilidades da aplicação e associá-las as dependências que oferecem tais riscos para os devidos ajustes. Esta recomendação esta embasada na interseção de resultados das ferramentas utilizadas e na otimização e na assertividade do trabalho de refactoring.

Recomenda-se a implantação de ferramentas de APM para que sejam criadas métricas e alarmes que auxiliem na continuidade do serviço em ambiente produtivo(monitoramento de processamento e memória por exemplo), tendo em vista que este tipo de ferramenta fornece mecanismos para determinarmos o comportamento da solução (auxiliam no refactoring de código) e também subsidia para o correto dimensionamento da infraestrutura.

Recomenda-se o desacoplamento das API's Rest do cor da aplicação, uma vez que esta prática promove a concorrência de recursos da aplicação principal e dificulta a escalabilidade horizontal. Vale ressaltar que ao segregar as API's Rest do core da aplicação,

DPRF	DPRF Segurança - Nota técnica	
-------------	--------------------------------------	--

recomenda-se segregar os demais componentes (classes negociais, serço, persistência e etc) para que se promova o reuso dos componentes.

Recomenda-se também o ajuste dos fluxos de execução da aplicação, uma vez que não um comportamento uniforme, temos classes de serviço acessando controladores e classes de persistência acessando classes de serviço. Esta prática não promove o princípio da abstração da orientação a objetos aplicado ao modelo MVC.

Para fins de uma melhor organização do projeto, recomenda-se que o mesmo seja segredado por contexto funcional (controladores, utilitários, serviço, persistência, negocial e etc) e em cada empacotamento seja criado a segregação por funcionalidade negocial. Esta prática ajuda a manter coesão entre os pacotes da aplicação e facilita o entendimento do programador ao promover manutenções corretivas/evolutivas.

Para fins de organização e padronização, recomenda-se que seja mantido de forma permanente 3 branchs no repositório GIT e que estas representem especificamente os ambientes ao qual estão implantadas, sendo elas master, homologação e desenvolvimento.