Proposta Arquitetural de Microsserviços no Contexto da Secretaria de Segurança Pública do Estado de Mato Grosso

Secretaria de Segurança Pública do Estado de Mato Grosso
Fundação de Amparo a Pesquisa do Estado de Mato Grosso
Instituto Federal de Mato Grosso - Campus Cuiabá Cel. Octayde Jorge da Silva
Edital 004/2020
Execução entre 01/08/2020 - 28/07/2021

Instituto Federal de Mato Grosso

Proposta Arquitetural de Microsserviços no Contexto da Secretaria de Segurança Pública do Estado de Mato Grosso

Pesquisadores:

Adriano Neres Araújo Souza

Evandro César Freiberger

João Paulo Delgado Preti

Tiago de Almeida Lacerda

Valtemir Emerêncio do Nascimento

Alunos:

Gabriel José Curvo Honda

Luís Fernando Vieira Sampaio

Membros Integrantes da SESP:

Willer Sondrei Oliveira Marques Silva

Lavídico Alves de Brito Junior

Gestão Operacional da SESP:

Marcos Roberto Weber Hubner

Walmir Akihiro Oribe

Diana Maria de Lima

Bruno de Oliveira Figueiredo

Instituto Federal de Mato Grosso

Dados Internacionais de Catalogação na Publicação (CIP) Câmara Brasileira do Livro, MT, Brasil

Preti, João Paulo D.; Souza, Adriano N. A.; Freiberger, Evandro César; Lacerda, Tiago A.

Proposta Arquitetural de Microsserviços no Contexto da Secretaria de Segurança Pública do Estado de Mato Grosso.

Cuiabá-MT: Instituto Federal de Mato GrossoLtda., 2021.

1. Programas de computador. 2. Arquitetura de Software. 3. Sistemas Distribuídos. 4. Microsserviços.

Lista de ilustrações

Lista de tabelas

Sumário

CAPÍTULO 1

Cenário

Aqui descrevemos o cenário do processo de negócio do protótipo funcional, onde optamos por tratar um caso particular de atendimento 190. É importante ressaltar que trata-se de um cenário hipotético, não envolvendo nenhum sistema da SESP-MT e que tem por finalidade apenas servir de base para validar a arquitetura e o ambiente de microsserviços.

Sabe-se que o atendimento 190 pode receber chamados para diversos problemas conforme ilustrado na figura 1:

Essa ilustração não é completa ou exaustiva, apenas cita como exemplo algumas ocorrências recebidas pelo atendimento 190. O processo de negócio escolhido para detalhamento e uso no projeto de microsserviços é o de Roubo de Veículo, conforme demarcado em vermelho na imagem.

Nesse cenário, vítimas de roubo devem entrar em contato com o 190 - Atendimento de Emergência da Polícia Militar, identificando-se e informando os dados relacionados ao fato, como local, descrição do carro e até dos suspeitos (em casos de roubo). Depois disso, deve-se ir à delegacia de polícia mais próxima e registrar um boletim de ocorrência. A imagem 2 descreve esse processo:

Figura 1 – Fluxo Atendimento 190

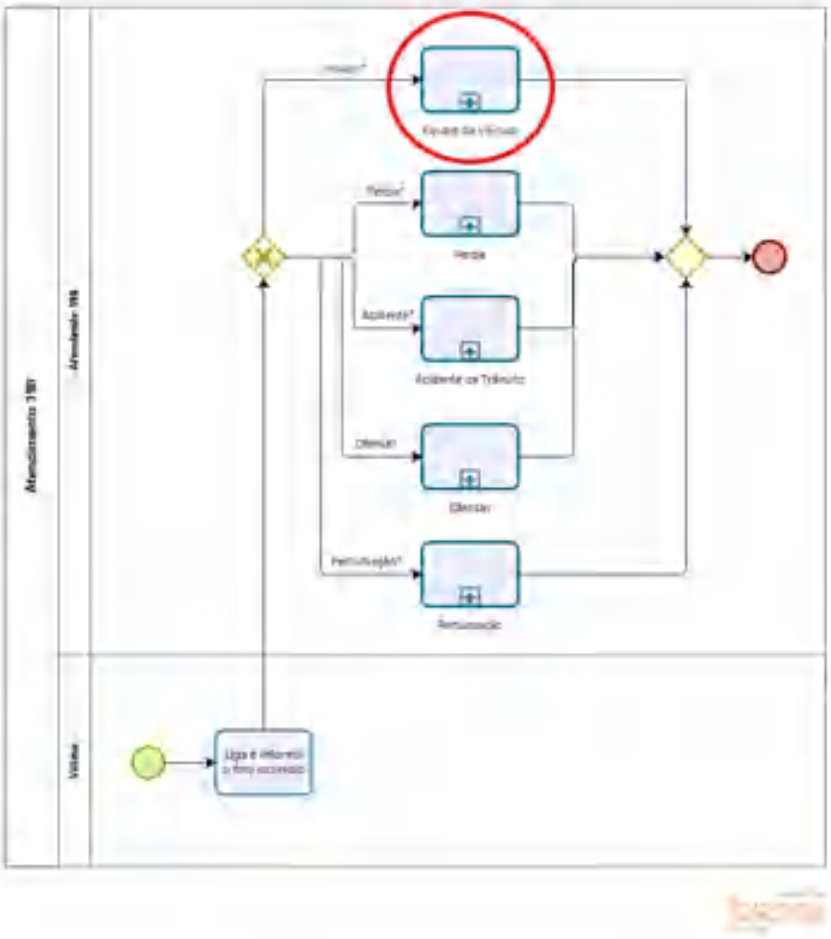
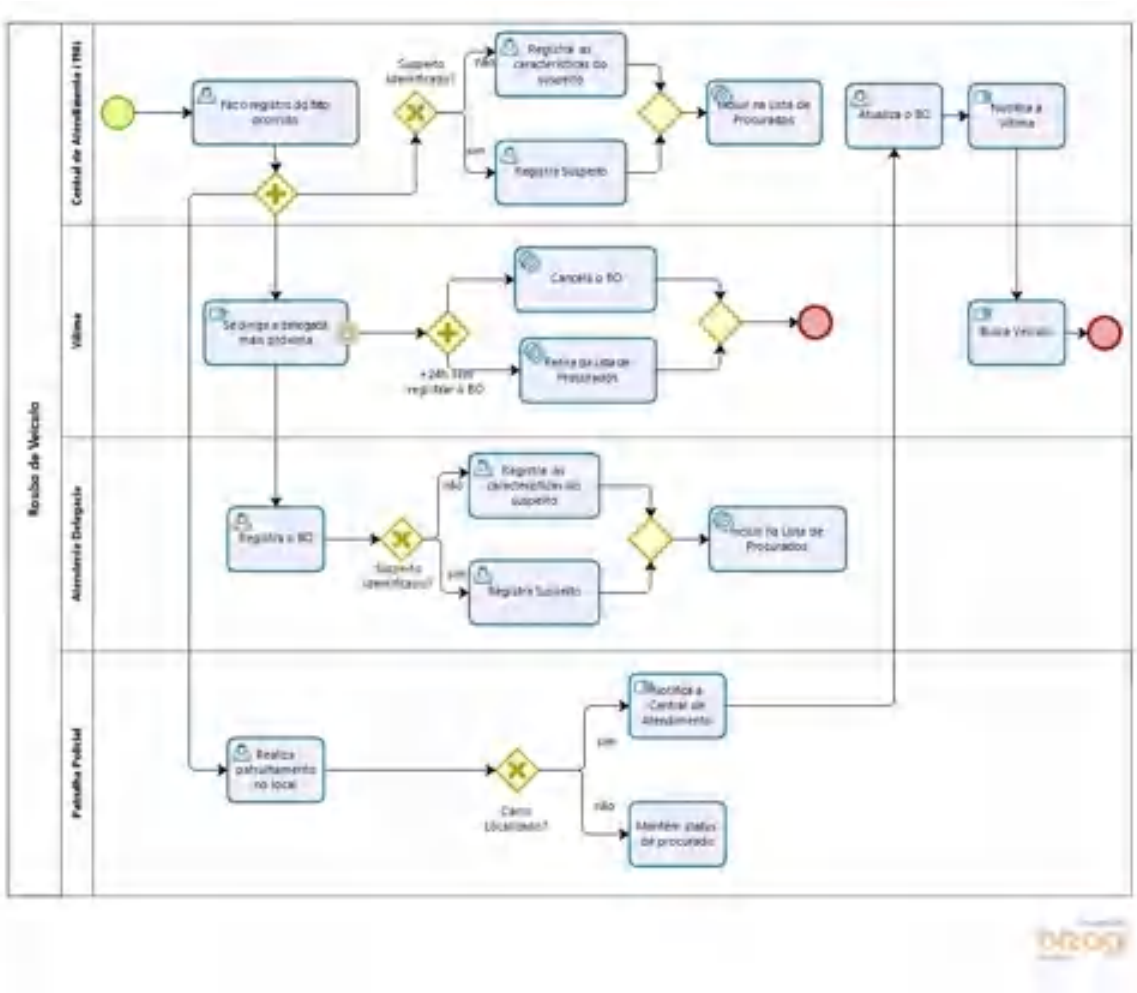


Figura 2 – Roubo de Veículos



CAPÍTULO 2

Projeto

Neste capítulo apresentamos as primeiras decisões de projeto, apresentando o domínio da aplicação e a organização básica.

2.1 Domínio

Tendo em vista o Cenário do atendimento 190 para Roubo de Veículos, suas ramificações de tarefas e envolvidos no processo é que se chegou ao domínio da figura abaixo, identificando os conceitos que devem ser tratados, apresentando seus relacionamentos e respectivos dados mínimos.

Lembrando que se trata de um cenário hipotético e que os elementos da figura abaixo servem apenas para dar sustentação ao protótipo funcional que será implementado para validar o ambiente e a arquitetura de microsserviços.

2.2 Microsserviços

Vislumbram-se cinco microsserviços que devem ser implementados cada qual em seu respectivo projeto, sendo estes:

Esses cinco microserviços podem estabelecer uma relação de comunicação conforme figura abaixo:

Quanto ao mapeamento dos microsserviços (endpoint), temos como base o Guia para Nomeação de Recursos REST, mapeando o microserviço e os recursos conforme: *http://domain-or-host:port/microservice/resource?*

Apresentamos em sequência as responsabilidades de cada microserviço e suas operações mínimas.

2.2.1 Registro Geral (/registro-geral)

Tem por objetivo representar um ponto de acesso para consulta aos dados básicos de qualquer pessoa física do Estado ou país. Podemos citar as operações mínimas desse serviço sendo:

Figura 3 – Modelo de Domínio

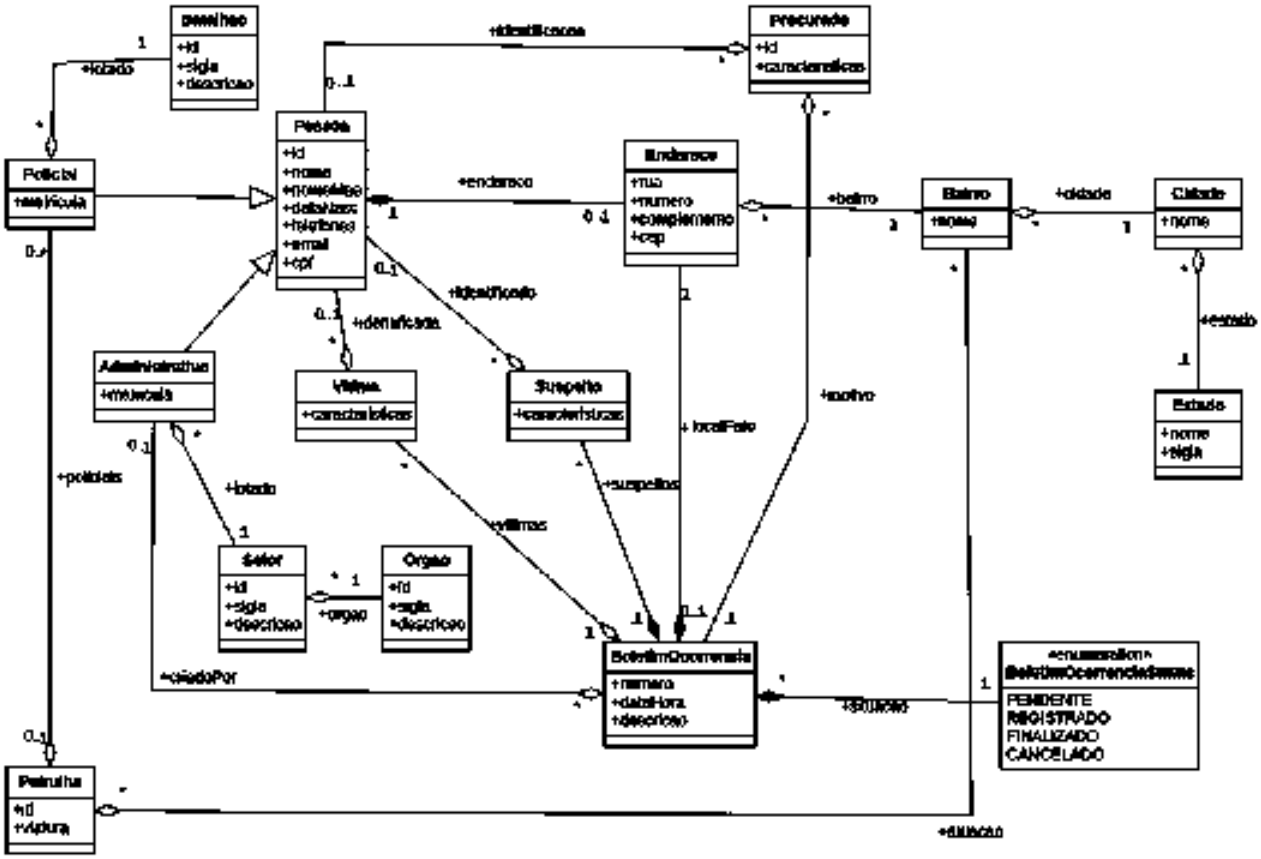


Figura 4 – Microsserviços

Figura 5 – Relações Entre Microsserviços

OPERAÇÃO	REQUEST	MAPEAMENTO
cadastrar pessoa	POST	/pessoas
alterar pessoa	PUT	/pessoas/{id}
buscar por identificador	GET	/pessoas/{id}
buscar pessoa por CPF	GET	/pessoas?cpf={cpf}
buscar pessoa por nome	GET	/pessoas?nome={nome}
buscar pessoa por nome da mãe	GET	/pessoas?nome-mae={nome}
buscar pessoa por data de nascimento	GET	/pessoas?nome-nascimento={data}
buscar cidade	GET	/cidades/{id}
buscar bairro por cidade	GET	/cidades/{id}/bairros/{id}
buscar bairro	GET	/bairros/{id}

Tabela 1 – Operações do Microsserviço Registro Geral

2.2.2 Carteira Funcional (/carteira-funcional)

Este serviço gerencia pessoas físicas que são servidores, funcionários, policiais e/ou colaboradores da SESP. Podemos citar as operações mínimas desse serviço sendo:

OPERAÇÃO	REQUEST	MAPEAMENTO
cadastrar administrativo	POST	/administrativos
cadastrar policial	POST	/policiais
alterar administrativo	PUT	/administrativos/{id}
alterar policial	PUT	/policiais/{id}
buscar por identificador	GET	/administrativos/{id} /policiais/{id}
buscar por matrícula	GET	/administrativos?matrícula={matrícula} /policiais?matrícula={matrícula}
buscar por nome	GET	/administrativos?nome={nome} /policiais?nome={nome}
buscar por CPF	GET	/administrativos?cpf={cpf} /policiais?cpf={cpf}
buscar por e-mail	GET	/administrativos?email={email} /policiais?email={email}

Tabela 2 – Operações do Microsserviço Carteira Funcional

2.2.3 Ocorrência (/gerenciamento-ocorrencia)

Gerencia os boletins de ocorrência, neste caso em particular para gerenciar as ocorrências de roubo de veículos do cenário estabelecido. Podemos citar as operações mínimas desse serviço sendo:

2.2.4 Procurado (/gerenciamento-procurado)

Tem por objetivo manter uma lista de procurados pela polícia, cujo motivo está relacionado a pelo menos um boletim de ocorrência. Podemos citar as operações mínimas desse serviço sendo:

2.2.5 Patrulha (/gerenciamento-patrulha)

Serviço que mantem o registro das viaturas e policiais que estão realizando patrulhamento em bairros da cidade. Podemos citar as operações mínimas desse serviço sendo:

2.3 Arquitetura

Cada microserviço tem ou deveria ter a independência técnica de outros microsserviços, podendo utilizar um stack de tecnologia mais apropriado para cada problema. Sob esse aspecto a arquitetura pode variar de um microserviço para outro.

OPERAÇÃO	REQUEST	MAPEAMENTO
registrar boletim de ocorrência	POST	/ocorrencias
alterar boletim de ocorrência	PUT	/ocorrencias/{id}
incluir suspeito	POST	/ocorrencias/{id}/suspeitos
incluir vítima	POST	/ocorrencias/{id}/vitimas
buscar boletim de ocorrência por número	GET	/ocorrencias?numero={numero}
buscar boletim de ocorrência por vítima	GET	/ocorrencias?id-vitima={id}
buscar boletim de ocorrência por suspeito	GET	/ocorrencias?id-suspeito={id}
buscar boletim de ocorrência por nome	GET	/ocorrencias/nome-vitima={nome} /ocorrencias/nome-suspeito={nome}
buscar boletim de ocorrência por características	GET	/ocorrencias?caracteristicas-vitima={v} /ocorrencias?caracteristicas-suspeito={s}
buscar boletim de ocorrência por situação	GET	/ocorrencias?situacao={situacao}

Tabela 3 – Operações do Microsserviço Ocorrências

OPERAÇÃO	REQUEST	MAPEAMENTO
inserir procurado	POST	/procurados
remover procurado	DELETE	/procurados/{id}
alterar procurado	PUT	/procurados/{id}
listar procurados	GET	/procurados
buscar procurado	GET	/procurados/{id}

Tabela 4 – Operações do Microsserviço Procurado

OPERAÇÃO	REQUEST	MAPEAMENTO
cadastrar patrulha	POST	/patrulhas
alterar patrulha	PUT	/patrulhas/{id}
buscar patrulha por viatura	GET	/patrulhas?viatura={placa}
buscar patrulha por policial	GET	/patrulhas?policial={id}
buscar patrulha por bairro	GET	/patrulhas?bairro={bairro}
listar patrulhas	GET	/patrulhas

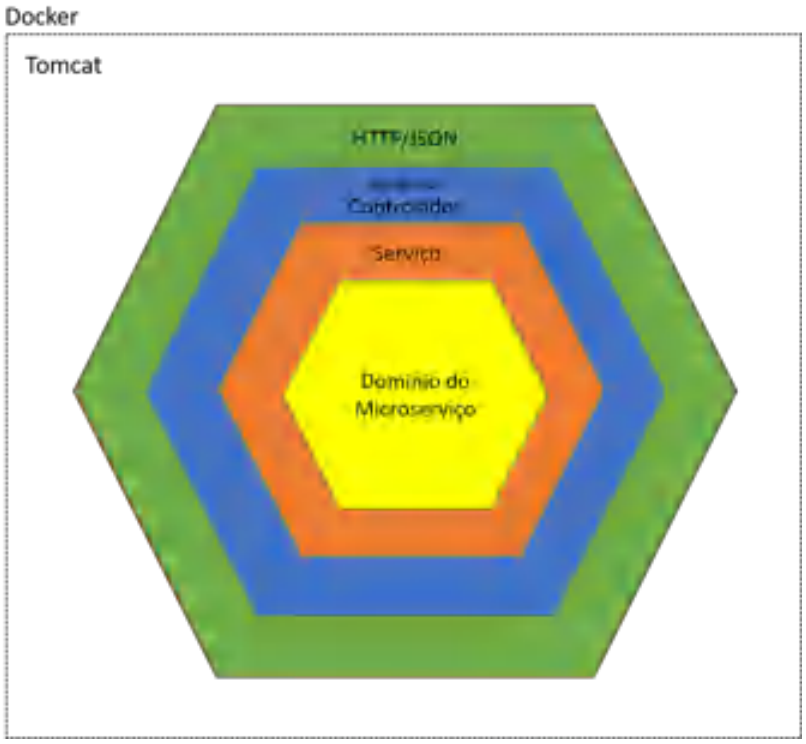
Tabela 5 – Operações do Microsserviço Patrulha

Apesar disso, iremos adotar um padrão arquitetural no protótipo funcional que será construído de forma gradual ao longo de todo projeto, podendo haver pequenas variações dependendo das necessidades de cada microserviço. Isso não tem nenhuma relação com o acoplamento entre os microsserviços, é apenas uma decisão que permite otimizar a curva de desenvolvimento.

2.3.1 Arquitetura de Base

Todos os 5 microsserviços tem como base o modelo abaixo:

Figura 6 – Arquitetura dos Microsserviços



Domínio do Microserviço: core do microserviço que representa classes de entidade e/ou value objects;

Serviço: contêm a lógica do negócio;

Controlador: disponibiliza uma interface de acesso ao microserviço, expondo suas operações visíveis, bem como estabelece um formato para os dados de entrada e saída;

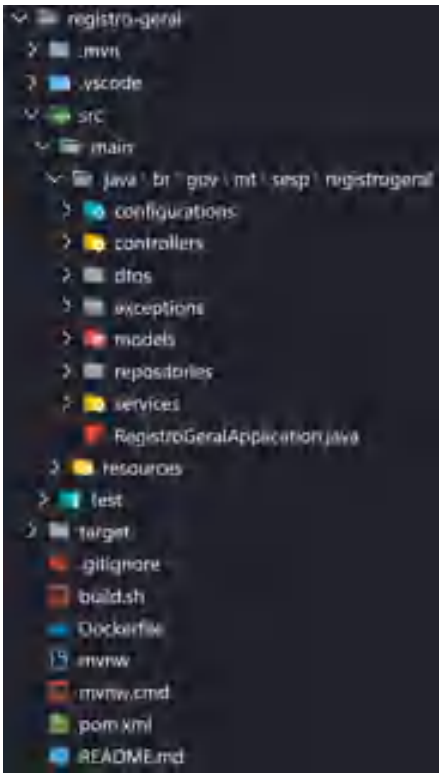
HTTP/JSON: protocolo de comunicação.

São microsserviços construídos utilizando Spring Boot, executados em um servidor Tomcat dentro de um cotainer Docker.

2.4 Estrutura do Projeto

Cada microserviço terá seu próprio projeto seguindo a estrutura de arquivos e diretórios padrão de um projeto Spring Boot web com Maven. Na figura abaixo temos um exemplo para o microserviço registro-geral.

Figura 7 – Estrutura do Projeto



A organização dos pacotes segue o domínio da SESP-MT seguido do nome do microserviço. Dentro dessa estrutura temos subpacotes que organizam as camadas do microserviço conforme segue:

2.4.1 configurations

Local para organização dos elementos de configuração de classes utilitárias, bibliotecas ou outros elementos utilizados no projeto que necessitem de uma classe de configuração. Temos como exemplos a configuração de:

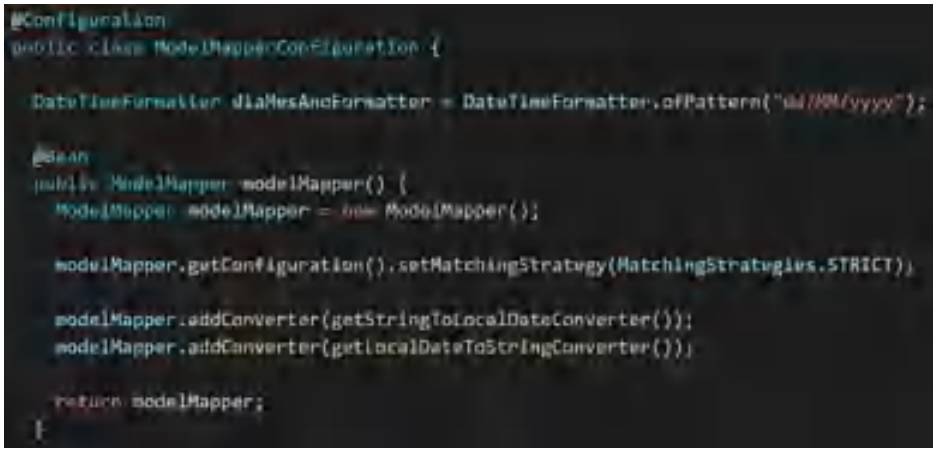
2.4.1.1 ModelMapperConfiguration

Tem por objetivo auxiliar no mapeamento entre elementos do domínio e os DTOs (Data Transfer Object), permitindo mapear propriedades entre dois objetos.

No exemplo acima estamos adicionando dois conversores para auxiliar no mapeamento (String → Date / Date → String) e definindo uma estratégia de mapeamento do tipo STRICT para que o algoritmo somente realize mapeamentos precisos, para não haver erros ou riscos de ambiguidade na operação.

Para maiores informações recomendamos a leitura de http://modelmapper.org/user-manual/.

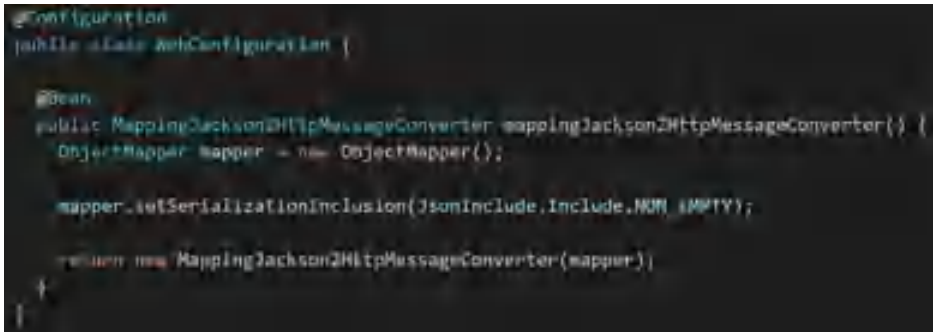
Figura 8 – ModelMapperConfiguration



2.4.1.2 WebConfiguration

Configuração do Jackson, um processador JSON de alto desempenho.

Figura 9 – WebConfiguration



Nesta configuração estamos estabelecendo que elementos vazios não podem ser incluídos ao gerar o JSON.

2.4.2 exceptions

Espaço reservado para as classes de exceções que podem ser lançadas pelo microserviço. Neste projeto temos a classe NegocioException que possui uma mensagem de erro padrão, mas que pode ser instanciada com uma mensagem personalizada:

As mensagens estão todas organizadas no arquivo messages.properties conforme exemplo abaixo:

Para um código mais limpo e seguro foi implementado um advice () que pemite interceptar exceções que forem lançadas sem a necessidade de um bloco try-catch. Dessa forma um simples throw interrompe o fluxo e formata uma respota JSON adequada para o requisitante.

No exemplo acima podemos ver a implementação do interceptador de exceções do tipo *NegocioException* que já define uma resposta de erro HTTP (422) do tipo UNPROCESSABLE_ENTITY.

Foram implementados no projeto outros interceptadores para exceções do tipo *MethodArgumentNotValidException* (400) e *RegistroNaoEncontradoException* (404).

Figura 10 – exceptions

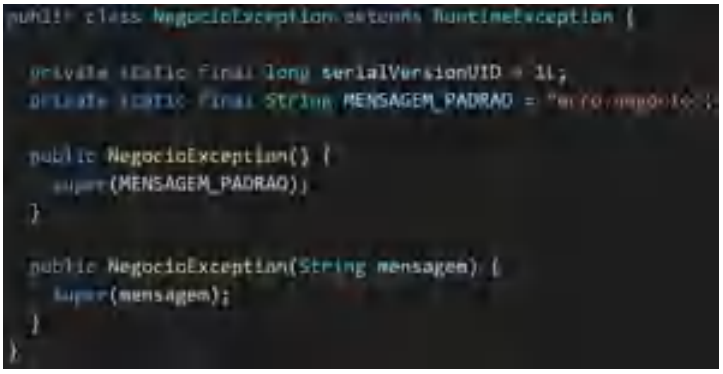
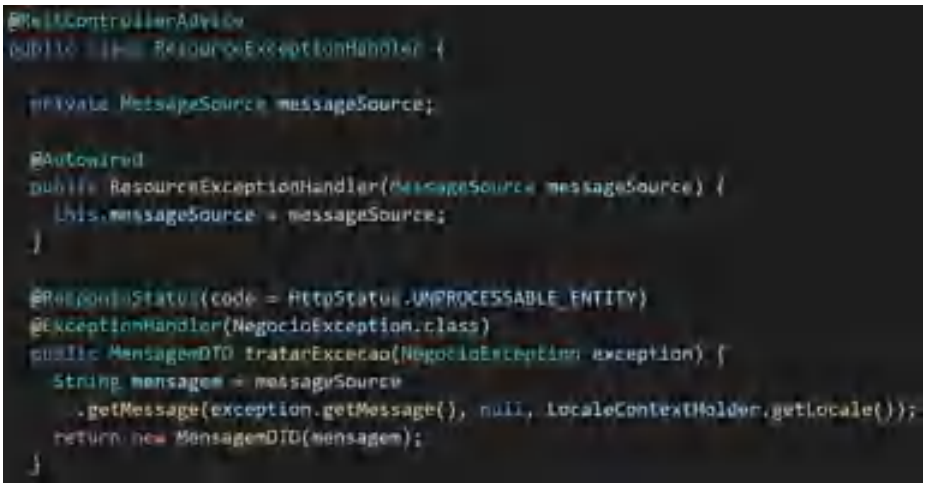


Figura 11 – Mensagens

Figura 12 – ResourceExceptionHandler



2.4.2.1 Códigos de Retorno HTTP

Na tabela abaixo apresentamos alguns códigos HTTP muito utilizados no universo REST para descrever o status das operações requisitadas.

2.4.3 models

Esta camada é o core do microserviço, representando seu domínio que é abstraído para o mundo externo por meio dos DTOs.

Nesta camada estão classes do tipo Entity, Enums e/ou outras que representem o domínio da aplicação.

Código	Status	Descrição
200	OK	O recurso solicitado foi processado e retornado com sucesso.
201	Created	O recurso foi criado comsucesso.
400	Bad Request	Não foi possível interpretar a requisição. Verifique a sintaxe das informações enviadas.
404	Not Found	O recurso solicitado ou o endpoint não foi encontrado.
406	Not Acceptable	O formato enviado não é aceito. O cabeçalho `Content-Type` da requisição deve conter obrigatoriamente o valor `application/json` para requisições do tipo POST e PUT.
409	Conflict	Um objeto já foi criado com as mesmas informações.
422	Unprocessable Entity	A requisição foi recebida com sucesso, porém contém parâmetros que não foram processados como válidos. Para mais detalhes verifique as mensagens contidas no retorno.
500	Internal Server Error	Ocorreu uma falha no microserviço.

Tabela 6 – Códigos de Erro HTTP

2.4.4 repositories

Camada responsável pela estratégia de persistência do domínio do microserviço. Classes como Data Access Object (DAO) estão neste espaço e representam as estratégias de persistência e recuperação de dados.

Sua nomenclatura utiliza o nome da classe de domínio seguido da palavra Repository, por exemplo, PessoaRepository.

2.4.5 dtos

Os Data Transfer Objects (DTO) representam objetos que organizam os dados trocados entre diferentes camadas da aplicação e também dados para o mundo externo. Esses objetos podem ocultar a estrutura interna do domínio do microserviço e facilitar a organização dos dados para os serviços que os solicitam.

Como podemos observar temos dois DTOs para o mesmo conceito Pessoa.

O primeiro (PessoaDTO) representa o conceito de Pessoa mantendo a organização original do domínio, mas sem o compromisso técnico de manter as configurações originais (por exemplo de JPA). Dessa forma podemos ter um DTO para trafegar entre as diversas camadas do microserviço sem a preocupação, por exemplo, de disparar uma chamada ao banco de dados. Esse mesmo DTO representa o modelo de dados que será exposto como resultado para o cliente do microserviço.

O segundo (PessoaSaveDTO) permite expor ao mundo externo um formato mais simples e menos burocrático para que outros serviços possam invocar operações de salvar e alterar, sem que precisem conhecer a relação de hierarquia e composição dos dados no domínio do microserviço.

Caso as operações de salvar e alterar demandem DTOs diferentes, ou seja, apenas PessoaSaveDTO não é suficiente, pode-se criar DTOs Create e Update para suprir essa necessidade. Nesse caso teríamos PessoaCreateDTO e PessoaUpdateDTO, um para cada operação.

Tabela 7 – Classes DTOs

2.4.6 services

Camada que implementa as regras do negócio acerca do contexto tratado pelo microserviço. Esta camada é responsável por se comunicar com as camadas que representam recursos de interesses, como por exemplo repositories, manipular os objetos de domínio e transformá-los, se necessário, em DTOs para as camadas superiores.

O formato base é uma classe com o nome do domínio mais a palavra Service, por exemplo PessoaService, conforme exemplo abaixo:

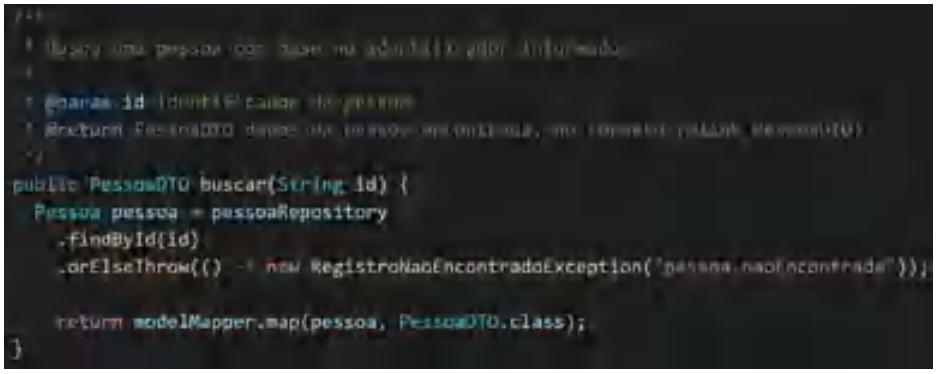
Figura 13 – PessoaService

```
@Service
public class PessoaService {

  private PessoaRepository pessoaRepository;
  private BairroRepository bairroRepository;
  private ModelMapper modelMapper;

  @Autowired
  public PessoaService(PessoaRepository pessoaRepository,
                       BairroRepository bairroRepository,
                       ModelMapper modelMapper) {
    this.pessoaRepository = pessoaRepository;
    this.bairroRepository = bairroRepository;
    this.modelMapper = modelMapper;
  }
```

Neste exemplo em particular PessoaService acessa dois repositórios (PessoaRepository e BairroRepositor) e precisa de ModelMapper para auxiliar no mapeamento das propriedades dos objetos de domínio (models) e DTOs.

Abaixo podemos ver no exemplo a busca de Pessoa no repositório e o uso de ModelMapper para mapear as propriedades de Pessoa para PessoaDTO.

Figura 14 – Método buscar

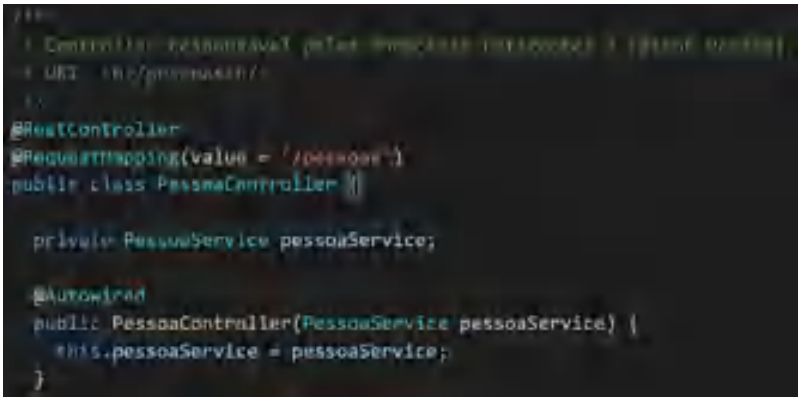


2.4.7 controllers

Esta camada é responsável por implementar e expor os endpoints servindo de porta de comunicação do microserviço com o mundo externo. Também é a camada que se comunica e organiza as chamadas a camada services e que pode estabelecer comunicação com endpoints de outros microsserviços.

O formato base é uma classe com o nome do recurso mais a palavra Controller, por exemplo PessoaController, conforme exemplo abaixo:

Figura 15 – PessoaController



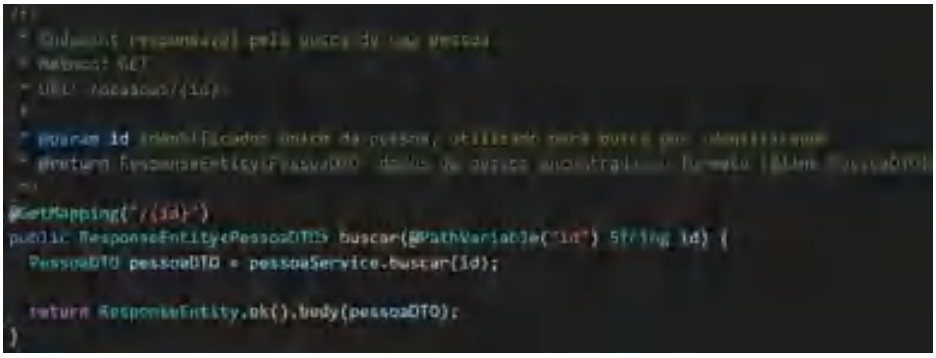
Cada controller estabelece pelo menos:

- documentação sobre sua finalidade;
- uma URI de acesso;
- referência a pelo menos uma classe de serviço;
- pelo menos uma operação (inserção, buscas, alteração, ...).

Às operações podem variar dependendo da finalidade do controlador, mas não é incomum termos para um recurso mapeado o suporte às requisições HTTP GET (busca), POST (inserção), PUT (alteração) e DELETE (exclusão).

Para cada operação temos uma documentação da finalidade, método HTTP suportado, URI de acesso, parâmetros e retorno.

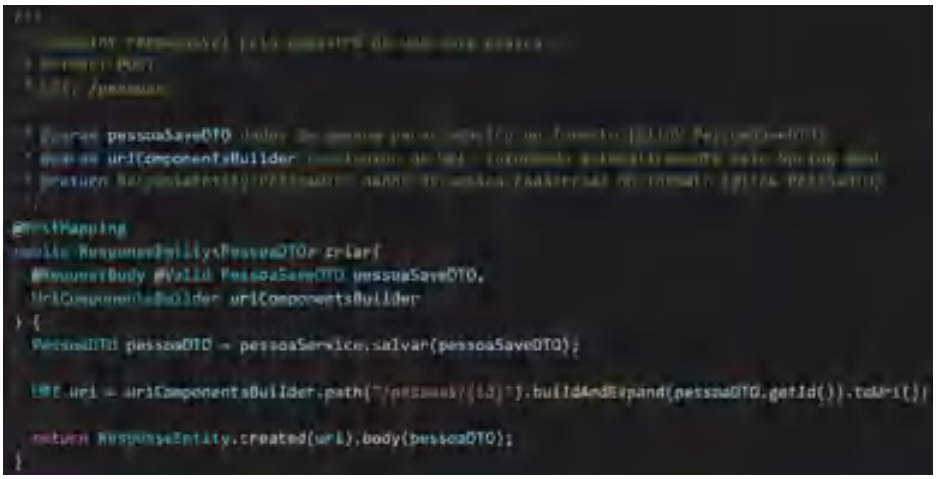
Figura 16 – Requisição GET



Abaixo temos o exemplo do suporte a uma requisição GET para acesso a uma instância do recurso por meio do seu identificador:

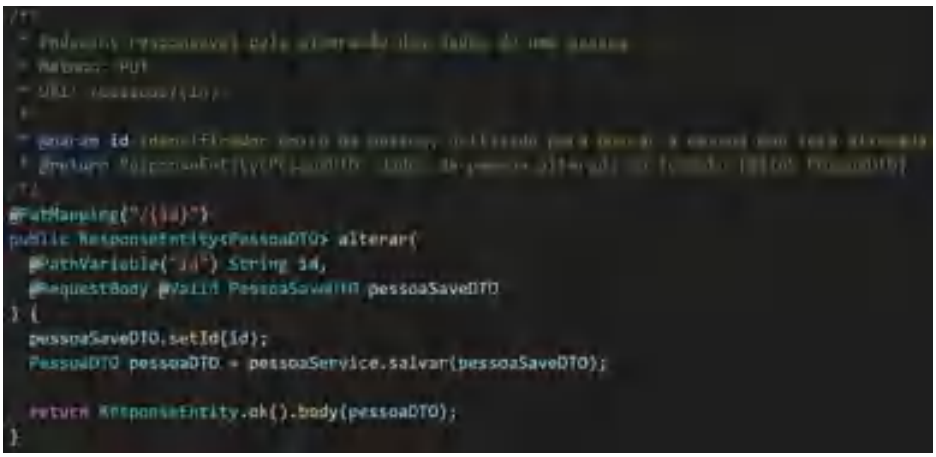
Neste outro exemplo a URI tem suporte a requisição POST para inserção:

Figura 17 – Requisição POST



E neste último o suporte a uma requisição PUT para alteração de dados:

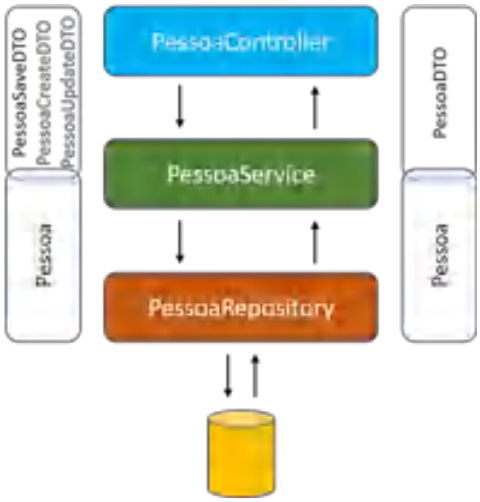
Figura 18 – Requisição PUT



Perceba que nestes dois últimos (criar e alterar) sempre retornamos o JSON de PessoaDTO como resultado de confirmação da inserção ou alteração e que o mesmo DTO (PessoaSaveDTO) é suficiente como representação de dados para as duas operações.

Podemos resumir essas camadas e responsabilidades acerca do conceito de Pessoa conforme ilustração abaixo:

Figura 19 – Camadas e Responsabilidades do Conceito Pessoa



CAPÍTULO 3

Ambiente de Desenvolvimento

Aqui descrevemos o ambiente de desenvolvimento que foi configurado e utilizado no projeto.

3.1 Configuração do Ambiente

3.1.1 Ferramentas e recursos

O ambiente de desenvolvimento é composto por ferramentas e recursos, alguns essenciais e outros opcionais. Novas ferramentas e atualizações ocorrerão ao longo de todo o projeto.

3.1.1.1 Ferramentas e recursos essenciais

As ferramentas essenciais, que exigem instalação no ambiente do desenvolvedor, estão listadas a seguir.

Ferramenta	Versão	Descrição
Docker Community	19.03.12	Conjunto de serviços utilizados para manutenção e gerenciamento de contêineres
Java	11.x	Linguagem de programação
Git	2.28.0	Sistema de Controle de Versionamento

Tabela 8 – Ferramentas Essenciais

3.1.1.2 Ferramentas e recursos opcionais

Estes recursos não são necessários e não influenciarão no funcionamento correto do ambiente caso não sejam instalados ou sejam substituídos por equivalentes. Seu uso é opcional e visa apenas facilitar ou melhorar o ambiente de desenvolvimento.

Ferramenta / Recurso	Descrição
Visual Studio Code	Editor de código para desenvolvimento
Insomnia Core	Cliente REST utilizado para testes dos endpoints das APIs
Mark Text	Ferramenta para edição de textos em formato Markdown, ideal para manutenção dos arquivos README.md dos projetos
Fira Code	Fonte mono-espaçada ideal para ambiente de desenvolvimento
JetBrains Mono	Fonte mono-espaçada ideal para ambiente de desenvolvimento

Tabela 9 – Ferramentas Opcionais

Para garantir uma melhor experiência de usuário no ambiente de desenvolvimento, faz-se necessário a instalação de algumas extensões do Visual Studio Code.

Este procedimento pode ser realizado dentro do próprio editor, acessando a aba “Extensions” no menu lateral da ferramenta.

A lista de extensões essenciais estão listadas a seguir.

Extensão	Descrição
Java Extension Pack	Pacote de extensões que adicionam a capacidade de interpretação e execução de código Java no Visual Studio Code
Spring Boot Extension Pack	Pacote de extensões que adicionam recursos para desenvolvimento utilizando Spring Boot no Visual Studio Code.
GitLens	Facilita a visualização de commits do Git dentro do código.
SonarLint	Ajuda a detectar problemas e melhorias no código.
Docker	Facilita a criação e gerenciamento de contêineres dentro do Visual Studio Code.
Remote - Containers	Facilita a manutenção de contêineres dentro do Visual Studio Code.
Material Icon Theme	Tema de ícones para o Visual Studio Code. Permite personalizar ícones para cada diretório ou arquivo nas configurações de usuário.
Dracula	Tema para o Visual Studio Code.
One Dark Pro	Tema para o Visual Studio Code

Tabela 10 – Extensões Essenciais do Visual Studio Code

3.1.1.3 Configuração do Visual Studio Code

O Visual Studio Code permite a configuração por meio de arquivo. Esta configuração é acessível por meio do menu Preferences -¿ Settings.

Na tela seguinte, o acesso às configurações por arquivo estão acessíveis no canto superior direito da ferramenta, conforme opção destacada:

Figura 20 – Preferências do Visual Studio Code

Uma configuração sugerida pode ser baixada no link a seguir.

```
{
  // Define o tema de icones do VSCode
  "workbench.iconTheme": "material-icon-theme",

  // Tamanhos e fonte
  "window.zoomLevel": 0,
  "editor.tabSize": 2,
  "editor.fontSize": 14,
  "editor.fontFamily": "JetBrains Mono",
  "editor.fontLigatures": true,

  // Terminal
  "terminal.integrated.fontSize": 14,

  // Comportamento padrao do editor
  "window.restoreWindows": "none",
  "breadcrumbs.enabled": true,
  "editor.renderLineHighlight": "gutter",
  "editor.parameterHints.enabled": false,
  "editor.rulers": [80, 120],
  "editor.suggestSelection": "first",
  "explorer.compactFolders": false,
  "explorer.confirmDragAndDrop": false,
  "explorer.confirmDelete": false,
  "workbench.startupEditor": "newUntitledFile",
  "workbench.editor.labelFormat": "short",
  "vsintellicode.modify.editor.suggestSelection": "automaticallyOverrodeDefaultValue",

  // Java
  "java.semanticHighlighting.enabled": true,
  "java.refactor.renameFromFileExplorer": "preview",
  "java.project.importOnFirstTimeStartup": "automatic",
  "files.exclude": {
      "**/.classpath": true,
      "**/.project": true,
      "**/.settings": true,
      "**/.factorypath": true
  },

  // Material Icons
  "material-icon-theme.activeIconPack": "nest",
    "material-icon-theme.folders.associations": {
        "dtos": "client",
        "exceptions": "error",
        "models": "class",
        "repositories": "database",
        "services": "resource"
    }
}
```

Código 3.1 – settings.json

3.1.2 Tecnologias

Além das ferramentas de desenvolvimento, os projetos utilizam algumas tecnologias específicas que não exigem a instalação no ambiente do desenvolvedor, mas importa ressaltar.

Tecnologia	Versão	Descrição
Spring Boot	2.3.3	Framework para desenvolvimento Java
Maven	3.>	Ferramenta para gerenciamento de pacotes e execução de scripts

Tabela 11 – Tecnologias de Apoio ao Desenvolvimento

3.1.3 Repositórios

Todos os projetos estão versionados utilizando Git e disponibilizados em plataforma GitLab própria da Fábrica de Software do IFMT:

Projeto	Repositório
Registro Geral	https://gitlab.com/sesp-mt/registro-geral - Connect to preview
Carteira Funcional	https://gitlab.com/sesp-mt/carteira-funcional - Connect to preview
Ocorrência	https://gitlab.com/sesp-mt/gerenciamento-ocorrencia - Connect to preview
Procurado	https://gitlab.com/sesp-mt/gerenciamento-procurado - Connect to preview
Patrulha	https://gitlab.com/sesp-mt/gerenciamento-patrulha - Connect to preview
GUI	https://gitlab.com/sesp-mt/gerenciamento-roubo-web - Connect to preview
Provider Keycloak	https://gitlab.com/sesp-mt/sespmt-spi - Connect to preview
Autenticação	https://gitlab.com/sesp-mt/autenticacao - Connect to preview
setup	https://gitlab.com/sesp-mt/setup - Connect to preview
Sprig Boot Admin	https://gitlab.com/sesp-mt/spring-boot-admin - Connect to preview

Tabela 12 – Repositórios dos Projetos

Para acesso aos repositórios apenas para leitura utilize:

Usuário: fabricadesoftwareifmt@gmail.com
Senha: sespmt2020&

Na raiz de cada projeto o arquivo README.md fornece instruções acerca da construção e execução do projeto.

Temos ao total 8 projetos sendo 5 microsserviços, 1 de interface gráfica para o protótipo funcional, 1 Service Provider Interface (SPI) que é um adaptador para o keycloak acessar o banco de dados de usuários e 1 setup.

O projeto **setup** contêm scripts que permitem a construção e execução dos 5 microsserviços utilizando Docker Compose.

3.2 Organização das Branches

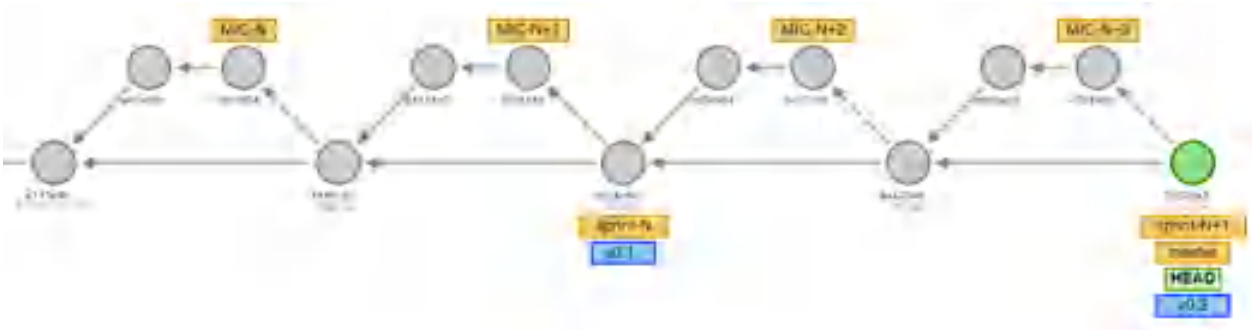
3.2.1 Até a versão 0.2 (Sprint 2)

Até a versão 0.2 (sprint 2), foi adotada a organização abaixo:

Conforme podemos observar na figura abaixo temos três branches no projeto:

1. **master:** principal e representa uma versão estável do sistema;
2. **sprint:** recebe as modificações finalizadas de cada tarefa daquela sprint;
3. **MIC:** contém código que está sendo desenvolvido para atender uma tarefa específica da ferramenta de issue tracking.

Figura 21 – Fluxo das Branchs



Criamos uma *branch* para cada *sprint* em andamento e a partir dessa nova branch criamos branches MIC-N que representam as tarefas que estão sendo desenvolvidas (na ferramenta de issue tracking as tasks tem id com a sigla MIC de microserviço). Ao término da tarefa a *branch* MIC-N é mesclada à sprint na qual está associada.

Quando ocorre o término da sprint, fazemos a apresentação da demonstração para o cliente, para então mesclarmos a sprint na branch master e demarcá-la com uma tag de versionamento. A sprint 1 é demarcada como v0.1, a sprint 2 como v0.2 e na **entrega final** do projeto temos então a **v1.0**.

A partir da versão 0.3 (Sprint 3) adotamos o processo definido e documentado em Processo de Ramificações e Versionamento.

3.3 Build

3.3.1 Build Individual

O processo de build pode ocorrer de forma isolada e individual e já está documentado no README.md de cada projeto, sendo possível utilizar duas estratégias: build local no ambiente de desenvolvimento ou utilizando Docker.

3.3.2 Build local

Para efetuar o build local é necessário ter o Java instalado na versão correspondente.

Execute a build do projeto utilizando o wrapper do Maven incorporado no projeto:

```
./mvnw clean package
```

Execute o comando abaixo no arquivo gerado no diretório target, substituindo o nome do arquivo pelo nome correspondente gerado pela build:

```
java -jar target/registro-geral.jar
```

3.3.3 Build com Docker

Execute o arquivo build.sh para criar a imagem da aplicação:

```
sh build.sh
```

Inicie o container da aplicação:

```
docker run -d -p 8005:8080 --name registro-geral br.gov.mt.sesp/regis
```

3.3.4 Build Geral

O processo de build também pode ocorrer de forma geral, aplicado sobre os 5 microserviços e instanciando os contêineres por meio do Docker Compose.

O projeto setup contêm os scripts necessários para a realização dessas tarefas. O arquivo README.md descreve o processo abaixo:

3.3.5 Pré-requisito

Para utilizar o projeto setup, é necessário que os serviços estejam previamente organizados em um diretório comum, com os nomes padrões dos projetos, conforme a representação abaixo:

```
.
+-- autenticacao
+-- carteira-funcional
+-- gerenciamento-ocorrencia
+-- gerenciamento-patrulha
+-- gerenciamento-procurado
+-- registro-geral
+-- sespmt-spi
+-- spring-boot-admin
+-- gerenciamento-roubo-web
+-- setup
```

3.3.6 Build dos projetos

Faz-se necessária, apenas na primeira execução, a criação de dois volumes para o docker:

```
docker volume create db-keycloak-volume
docker volume create db-apps-volume
```

3.3.6.1 Até a versão 0.5.X

Para efetuar a build dos projetos, pode-se utilizar o script localizado na raiz do pojeto setup, informando como argumento do comando a branch específica para a qual as imagens serão geradas:

```
sh setup.sh master
```

Este procedimento irá efetuar o checkout do projeto na branch informada, empacotar o projeto utilizando o maven e efetuar o build da imagem correspondente:

Figura 22 – Processo de Build

3.3.6.2 A partir da versão 0.6.X

Tendo em vista o crescimento do ambiente e a quantidade de contêineres com serviços de base para suporte aos microsserviços é que o processo de construção e execução foi alterado para facilitar a administração e execução do projeto.

Para efetuar a build e já realizar a execução dos projetos, utiliza-se a partir da versão 0.6.X o script localizado na raiz do pojeto setup, informando como argumento do comando a branch específica para a qual as imagens serão geradas e executadas:

```
sh start.sh master
```

Este procedimento irá efetuar o checkout do projeto na branch informada, empacotar o projeto utilizando o maven, efetuar a build da imagem correspondente, executar primeiro os contêineres de base verificando se todos os serviços estão online (*docker-compose-tools.yml*) para só então executar os contêineres dos microsserviços (*docker-compose-apps.yml*).

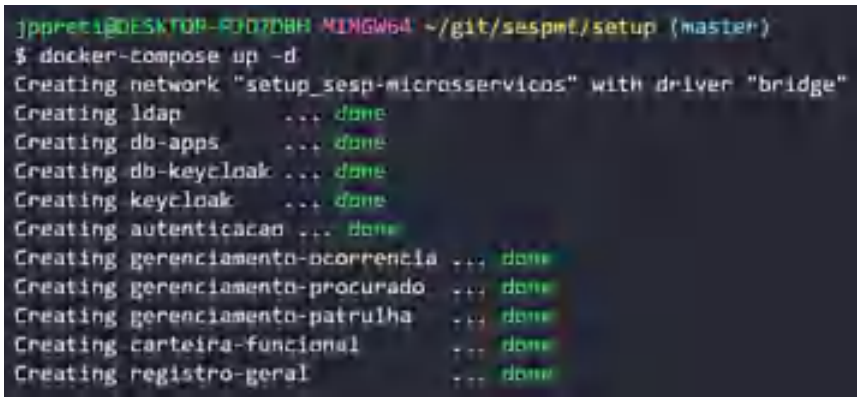
3.3.7 Execução dos projetos

3.3.7.1 Até a versão 0.5.X

Para executar os projetos, pode-se utilizar o docker-compose.yml localizado na raiz do projeto setup:

```
docker-compose up -d
```

Figura 23 – Execução dos Projetos



Para verificar a situação de cada serviço, pode-se utiliar o comando ps:

```
docker ps
```

Para interromper a execução dos serviços, o mesmo docker-compose.yml pode ser utilizado:

```
docker-compose down
```

Figura 24 – Situação dos Serviços

3.3.7.2 A partir da versão 0.6.X

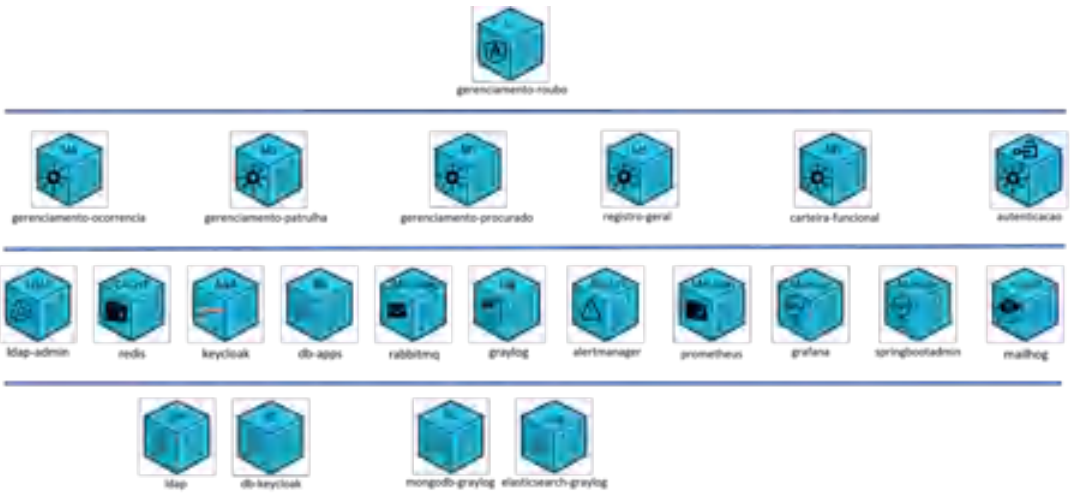
O próprio script start.sh descrito na seção anterior já realiza o build e a execução do ambiente e dos microsserviços, cabendo apenas mencionar a existência do script stop.sh para encerrar a execução do ambiente como um todo.

3.4 Portas dos Serviços

Ao inicializar os containers utilizando o docker-compose.yml cada serviço será vinculado a uma porta específica:

Nas figuras abaixo podemos ver os 22 containers e as dependências:

Figura 25 – Dependências entre os Containers



Para simplificar e facilitar o entendimento apresenta-se o modelo simplificado abaixo:

O gráfico acima foi gerado utilizando a ferramenta https://yuml.me/diagram/scruffy/class/draw, com o script abaixo:

```
[gerenciamento-roubo]->[gerenciamento-ocorrencia]
[gerenciamento-roubo]->[carteira-funcional]
[gerenciamento-roubo]->[registro-geral]
[gerenciamento-roubo]->[autenticacao]
[gerenciamento-ocorrencia]->[autenticacao]
[gerenciamento-procurado]->[autenticacao]
[gerenciamento-patrulha]->[autenticacao]
[carteira-funcional]->[autenticacao]
[registro-geral]->[autenticacao]
[autenticacao]->[keycloak]
[keycloak]->[db-keycloak]
[keycloak]->[ldap]
```

Nome do container	Descrição	Porta
autenticacao	Wrapper para o Keycloak	8000
carteira-funcional	Microserviço Carteira Funcional	8001
gerenciamento-ocorrencia	Microserviço Gerenciamento de Ocorrências	8002
gerenciamento-patrulha	Microserviço Gerenciamento de Patrulhas	8003
gerenciamento-procurado	Microserviço Gerenciamento de Procurados	8000
registro-geral	Microserviço Registro Geral	8003
keycloak	Serviço de AEoA (Keycloak)	38080
db-keycloak	Serviço de BD do Keycloak	35432
db-apps	Banco de dados de aplicação	35432
ldap	LDAP utilizado pelo Keycloak	3389
ldap admin	LDAP Client	3389
redis	Serviço de cache distribuído	36379
rabbitmq	Message Broker	15672
logstash	Ingestão de dados de Log	12201
elasticsearch	Indexador de logs	9200
graylog	Visualização de Logs	8000
prometheus	Indexador de Métricas	8000
grafana	Visualização de Métricas	8000
mailhog	Fake SMTP Server Interface Gráfica do Mailhog	1025 8025
alertmanager	Serviço de notificação dos alertas de Prometheus	8003
springbootadmin	Ferramenta de monitoramento e administração dos microserviços Spring Boot.	8002

Tabela 13 – Portas dos Serviços

Figura 26 – Dependências entre os Containers

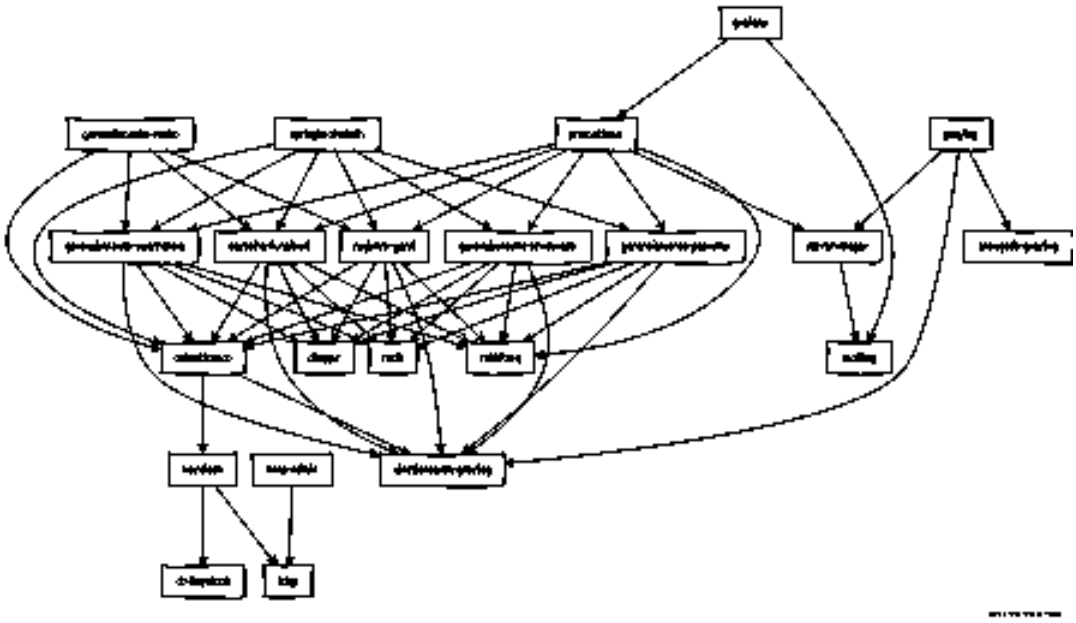
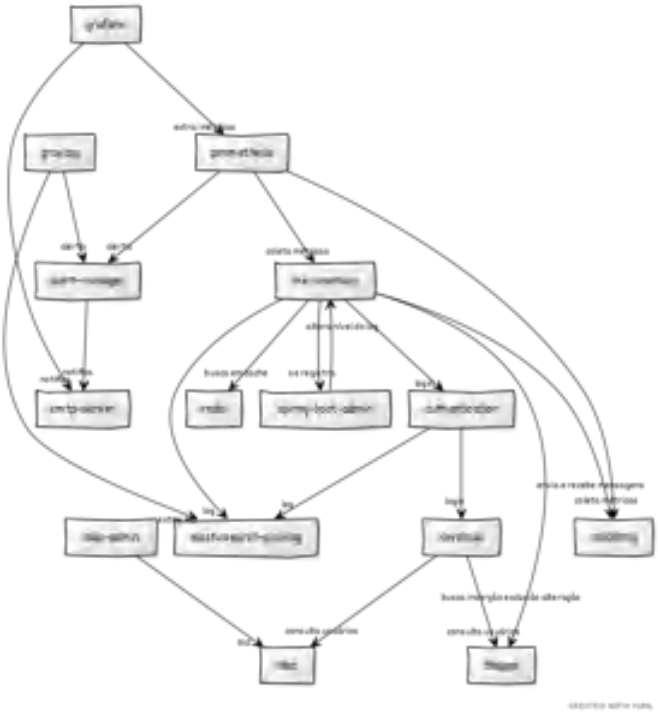


Figura 27 – Dependências entre os Containers



```
[ldap-admin]->[ldap]
[gerenciamento-ocorrencia]->[redis]
[gerenciamento-procurado]->[redis]
[gerenciamento-patrulha]->[redis]
[carteira-funcional]->[redis]
[registro-geral]->[redis]
[gerenciamento-ocorrencia]->[rabbitmq]
[gerenciamento-procurado]->[rabbitmq]
[gerenciamento-patrulha]->[rabbitmq]
[carteira-funcional]->[rabbitmq]
[registro-geral]->[rabbitmq]
[gerenciamento-ocorrencia]->[dbapps]
[gerenciamento-procurado]->[dbapps]
[gerenciamento-patrulha]->[dbapps]
[carteira-funcional]->[dbapps]
[registro-geral]->[dbapps]
[autenticacao]->[elasticsearch-graylog]
[gerenciamento-ocorrencia]->[elasticsearch-graylog]
[gerenciamento-procurado]->[elasticsearch-graylog]
[gerenciamento-patrulha]->[elasticsearch-graylog]
[carteira-funcional]->[elasticsearch-graylog]
[registro-geral]->[elasticsearch-graylog]
[graylog]->[elasticsearch-graylog]
[graylog]->[mongodb-graylog]
[alertmanager]->[mailhog]
[prometheus]->[alertmanager]
[grafana]->[prometheus]
[grafana]->[mailhog]
[springbootadmin]->[autenticacao]
[springbootadmin]->[gerenciamento-ocorrencia]
[springbootadmin]->[gerenciamento-procurado]
[springbootadmin]->[gerenciamento-patrulha]
[springbootadmin]->[carteira-funcional]
[springbootadmin]->[registro-geral]
[prometheus]->[rabbitmq]
[graylog]->[alertmanager]
[prometheus]->[carteira-funcional]
[prometheus]->[gerenciamento-ocorrencia]
[prometheus]->[gerenciamento-patrulha]
[prometheus]->[gerenciamento-procurado]
[prometheus]->[registro-geral]
```

Código 3.2 – Dependências

3.5 Testes

Recomendamos o uso da ferramenta *Insomnia Core* para interagir com os microsserviços e testar cada *endpoint*.

Conforme pode ser observado na figura abaixo, organizamos às chamadas aos diversos endpoints em pastas representando cada um dos cinco microsserviços, contendo inclusive estruturas JSON preenchidas com dados de exemplo.

É possível verificar também que estamos utilizando alias para representações da URI de identificação do recurso, como forma de flexibilizar a mudança do host sem ter que alterar individualmente cada chamada.

Para isso criamos uma configuração de ambiente chamada dev de acordo com o modelo abaixo:

Figura 28 – Testes dos endpoint

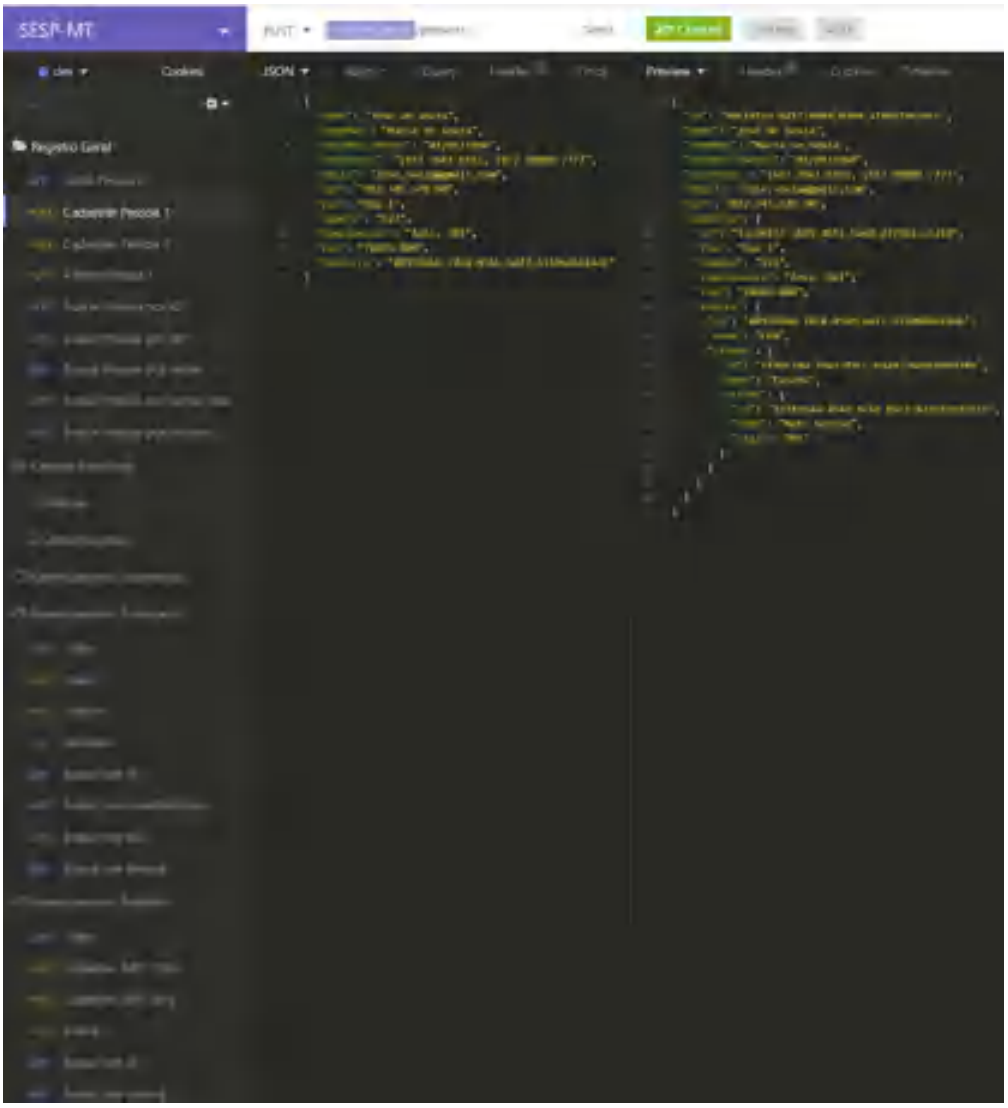


Figura 29 – Configuração do Ambiente

Disponibilizamos abaixo um arquivo para importação do workspace SESP-MT que contêm a lista de endpoints e a configuração do ambiente dev para teste.

Para realizar a importação vá até Application/Preferences/Data/Import Data.

3.6 Geração de Release

Há necessidade de geração de uma nova versão do microserviço sempre que ocorrerem manutenções evolutivas, corretivas ou adaptativas. Esse processo está definido em Processo de Ramificações e Versionamento.

Na branch release faz-se necessário atualizar o caminho do microserviço para que possamos identificar a versão em execução. Essa atividade também é importante para que se possa implantar e iniciar mais de uma instância do mesmo microserviço em versões diferentes. Essa alteração é realizada em cada microserviço em seu arquivo *application.propertie*s, conforme linha abaixo:

```
server.servlet.context-path=/v0-3/carteira-funcional
```

Com o objetivo de facilitar a administração e recuperação de versões anteriores dos microsserviços é que adotamos a política de evoluir o versionamento sempre em conjunto. Podemos exemplificar que mesmo que tenha ocorrido evolução no versionamento de um único microserviço, todos os demais microsserviços também deverão evoluir suas versões de forma a estabelecer uma mesma versão.

Essa política dispensa a necessidade de manter e adminstrar uma matriz de compatibilidade entre microsserviços e facilita a recuperação de versões anteriores. Ao recuperar um microserviço de versão 0.6, não é necessário investigar quais são os microsserviços compatíveis com essa versão, basta restaurar todos os artefatos na mesma numeração.

CAPÍTULO 4

Spring Boot Style Guide

Este capítulo aborda as convenções de nomenclatura utilizadas de acordo com o Spring Boot Style Guide.

4.1 Estrutura do projeto

Os projetos desenvolvidos em Java com Spring Boot seguem uma estrutura padrão de pacotes. Abaixo do pacote principal da aplicação são criados os seguintes pacotes:

configurations: Classes de configurações.

controllers: Classes de controle da API.

dtos: Data Transfer Objects.

exceptions: Exceções personalizadas e classe de tratamento de erros.

models: Modelos da aplicação.

repositories: Interfaces de repositórios da aplicação.

services: Classes de serviço. Regras negociais da aplicação.

4.2 Convenções de nomenclatura

Recomenda-se seguir a convenção padrão de nomenclatura . As definições deste documento complementam e indicam alguns padrões utilizados para estes projetos em específico.

4.2.1 Pacotes

Se o pacote possuir nome composto não utilize maiúsculas, hífens ou qualquer outra estratégia para indicar a separação dos nomes. Eles devem ser escritos todos em caixa baixa, conforme o exemplo abaixo.

```
br.gov.mt.sesp.gerenciamentoocorrencia
```

4.2.2 Classes

Classes que implementem padrões relacionadas a outras classes devem trazer o nome desse padrão como sufixo de seu próprio nome.

Por exemplo, em uma API de cadastro de Pessoa, as classes de controle, serviço e DAO devem seguir o seguinte padrão, respectivamente:

```
public class PessoaController {}
public class PessoaService {}
public class PessoaRepository {}
```

4.2.3 Interface

Interfaces costumam adicionar capacidades à classes. Por este motivo, para tornar o código mais semântico, quando possível adjetivar o nome da interface utilizando o prefixo -vel, conforme exemplos abaixo:

```
public interface Enumeravel {}
public interface Expansivel {}
public interface Gerenciavel {}
```

4.2.4 Métodos

Quando possível utilizar verbos no infinitivo para nomear métodos de negócio, conforme exemplos abaixo:

```
public void salvar() {}
public void registrarPagamento() {}
public void incluirItem() {}
```

Defina um padrão de nomenclatura para ser utilizado em seu projeto, evite utilizar nomenclaturas diferentes para métodos que executam ações equivalentes em diferentes lugares.

Um exemplo de mau uso seria criar o método incluir() na classe PessoaService para cadastrar pessoas no banco de dados, enquanto na classe ViaturaService o nome do método que cadastra produtos no banco de dados é cadastrar().

Como sugestão, utilize os seguintes nomes para métodos cujo resultado deve realizar alguma ação no banco de dados:

Salvar: Inclui ou altera um registro do banco de dados.

Cadastrar: Cria um novo registro no banco de dados.

Buscar: Busca um registro do banco de dados.

Alterar: Altera um registro do banco de dados.

Excluir: Exclui um registro do banco de dados.

Quando a ação que deseja realizar não executa diretamente uma ação no banco de dados, prefira uma nomenclatura diferente. Por exemplo:

Incluir: Inclui um item em uma lista, mas sem incluir no banco de dados diretamente. Neste caso, prefira incluirItem em vez de salvarItem ou cadastrarItem, por exemplo.

Remover: Remove o item de uma lista, mas sem removê-lo do banco de dados diretamente. Equivalente à situação acima, prefira removerItem em vez de excluirItem.

4.2.5 Variáveis

Utilize nomes descritivos. Não há problema em nomes extensos, desde que eles sejam descritivos o suficiente.

Exemplos de mau uso seriam:

```
1 private p = new Pessoa();
2 private vt = new Viatura();
3 private cf = new CarteiraFuncional();
```

Código 4.1 – Mau Uso de Nome de Variáveis

Exemplos melhorados dessas variáveis seriam:

```
1 private pessoa = new Pessoa(); // Ou: private novaPessoa = new Pessoa();
2 private produto = new Produto();
3 private carteiraFuncional = new CarteiraFuncional();
```

Código 4.2 – Bom Uso de Nome de Variáveis

4.3 Outros padrões

Existem outros padrões, não diretamente ligados à nomenclatura, que adotamos.

4.3.1 Controllers

Padrões aplicados à camada de controle.

4.3.1.1 URIs

URIs devem utilizar a separação de palavras com hífen. Portanto, um bom exemplo de URI para o controller responsável pela manutenção da entidade *BoletimOcorrencia.java* seria */boletins-ocorrencia* ou apenas *ocorrencias* em vez de */boletimOcorrencia*.

Adote a separação com hífens também para os parâmetros das URIs.

Controllers devem possuir URIs preferencialmente no plural, quando isso não afetar a semântica. Sendo assim, um controller responsável pela manutenção da entidade *Pessoa.java* dentro da API, por exemplo, seguiria o seguinte padrão:

```
GET       /pessoas         Listar pessoas
POST      /pessoas         Cadastrar pessoa
GET       /pessoas/–id"    Buscar pessoa pelo identificador
PUT       /pessoas/–id"    Alteração de todos os dados da pessoa relacionada ao identificador
PATCH     /pessoas/–id"    Alteração parcial dos dados da pessoa relacionada ao identificador
DELETE    /pessoas/–id"    Exclusão da pessoa relacionada ao identificador informado
```

4.3.1.2 URIs aninhadas

Em situações onde seja necessário criar endpoints para URIs aninhadas, utilizar o caminho inteiro das entidades na URI.

Exemplo: A aplicação já possui um endpoint de ocorrencias, mas surge a necessidade de um endpoint específico que retorne somente a lista de vitimas de uma ocorrência.

Pode ser resolvido com a criação do Controller VitimaController.java, com a seguinte URI:

```
/ocorrencias/–idBoletimOcorrencia"/vitimas
```

4.3.1.3 Métodos

Preferencialmente, Controllers devem ter no máximo 6 métodos básicos:

- GET - listar
- POST - criar
- GET - buscar
- PUT - alterar
- PATCH - alterar
- DELETE - excluir

Se não houver necessidade de todos eles é comum haver menos, mas não mais que isso. Se houver necessidade de mais algum método, provavelmente isso pode ser separado em outro Controller, mais específico.

CAPÍTULO 5

Projeto Interface Gráfica do Protótipo

Neste capítulo apresentamos o projeto Angular para compor a interface gráfica do sistema, facilitando a interação com os microsserviços e a apresentação do domínio de exemplo que foi implementado.

5.1 Estrutura do projeto

Toda a organização da estrutura de pastas do projeto é gerada pelo CLI (Command-line interface) do angular, seguindo portanto a convenção de estrutura de arquivos [1] do angular.

O núcleo do projeto está contido dentro da pasta src/app, e nesta o projeto segue estrutura de pastas abaixo:

- **models:** Contém as entidades da aplicação;
- **models/dtos:** Data Transfer Object;
- **services:** Classes de serviço, representam a camada de comunicação com a API;
- **views:** Classes e componentes da interface gráfica.

Cada feature desenvolvida, com exceção das classes contidas na pasta models e service , são formadas por componentes e módulos. Um módulo isola a regra de negócio de vários outros componentes enquanto cada componente é composto pelos seguintes arquivos:

- **nome-do-arquivo.component.ts**: um componente que controla as ações do usuário;
- **nome-do-arquivo.spec.ts:** teste unitário do componente;

[1] https://angular.io/guide/styleguide#file-structure-conventions

- **nome-do-arquivo.html:** tela controlada pelo componente;
- **nome-do-arquivo.css:** definições de estilo personalizadas para o componente.

5.2 Convenções de nomenclatura

A seguir, apresentamos as recomendações de nomenclatura dos elementos contidos no projeto.

5.2.1 Classes

É recomendado que as classes no Angular sigam o padrão de nomenclatura fornecido pelo framework. Sendo assim, em exemplos, os nomes atribuídos devem ser:

- **Pessoa:** Classe de modelo da entidade Pessoa;
- **PessoaService:** Uma classe de serviço que possui regras de negócio relacionadas à entidade Pessoa;
- **ListarPessoaComponent:** No caso de um componente, o padrão deve seguir o modelo TarefaComponent e recomenda-se utilizar verbos no infinitivo. Ex: Listar, cadastrar, excluir, buscar etc.

5.2.2 Arquivos

Os arquivos no projeto devem possuir a nomenclatura que respeite o seguinte padrão: *feature.type.ts*

Sendo assim, seguindo os mesmos exemplos exibidos na seção anterior, os arquivos seriam:

- pessoa.model.ts
- pessoa.service.ts
- listar-pessoa.component.ts

Existem outros nomes convencionais como *.module*, para arquivos de módulos, e *.spec*, para arquivos contento testes. Dessa maneira, recomenda-se seguir o mesmo padrão aplicado nos exemplos da lista acima.

5.2.3 Variáveis

Recomenda-se a utilização de nomenclatura descritiva que permita rápido entendimento acerca do porquê da existência da variável.

```
1 let nome: string;
2 let idade: number;
3 let listaNotas: Array<number>;
```

Código 5.1 – Nomenclatura de Variáveis

Preferencialmente, recomenda-se utilizar *const* sempre que possível.

5.2.4 Métodos

É recomendado utilização de nomes curtos e concisos como:

```
public buscaPessoa(id: number) : Pessoa {}
```

Código 5.2 – Nomenclatura de Métodos

Deve-se também utilizar o padrão verboRecurso, resultando em buscaPessoa, cadastraPessoa, apagaPessoa etc como títulos recomendados. Lembrando que o verbo deve estar na terceira pessoa do singular, no presente.

Há uma exceção para a chamada de recursos do microsserviço, nesse caso, deverá ser utilizado a convenção definida em Spring Boot Style Guide cujo os controllers, que correspondem ao services no angular, possuem no máximo 6 métodos básicos:

- GET - listar
- POST - criar
- GET - buscar
- PUT - alterar
- PATCH - alterar
- DELETE - excluir

5.3 Versionamento

O projeto gerenciamento roubo web está disponível no Gitlab e para versionamento da aplicação serão adotados o processo de versionamento definido em Processo de Ramificações e Versionamento e Padrão de Commits.

5.4 Ambiente de desenvolvimento

Tabela 14 – Ambiente de Desenvolvimento

Dependência	Versão
Node.js	14.x.x
Angular	10.1.x
NPM (Node Package Manager)	6.14.x
Typescript	4.0.3

Ao clonar o projeto deve-se executar o comando *npm install* para instalar todas as dependências do projeto.

5.5 Build e Execução

Para realizar o build e execução local é necessário que o usuário possua os serviços Gerenciamento de Ocorrência , Gerenciamento de Procurado, Registro Geral e o projeto Gerenciamento Roubo Web clonados.

Primeiro, é necessário inicializar os serviços do qual o projeto depende e, para isso, recomenda-se a utilização do processo automatizado de construção e build dos serviços disponível no projeto Setup e documentado em Ambiente de Desenvolvimento.

O build local do projeto Gerenciamento Roubo Web utiliza o **npm** combinado com Angular CLI e ocorre no servidor de desenvolvimento embutido no próprio Angular. O script definido no arquivo **package.json** gera um proxy reverso para realizar a comunicação com os serviços nos contêineres. Para realizar a execução do projeto basta utilizar o comando **npm run start** que assim o script **start**, definido no **package.json**, inicializará o servidor de desenvolvimento tornando o projeto acessível em *http://localhost:4200*.

Ainda, é possível realizar o build e deploy do contêiner do projeto utilizando apenas o Setup, para isso basta executar o comando **./setup BRANCH** onde a versão do build será baseada na branch passada como argumento para o script. Após isso, basta utilizar o comando **docker-compose up -d** para iniciar o projeto Gerenciamento Roubo Web em http://localhost:8006.

5.6 Navegação e Comunicação com os Microsserviços

Temos as seguintes páginas que compõem a interface gráfica do protótipo para gerenciamento de roubo:

- login: para autenticação dos usuários;
- ocorrências: para listar as últimas ocorrências cadastradas e permitir direcionar o usuário para o cadastro de uma nova ocorrência ou alteração de uma existente;
- cadastro: para inserção de uma nova ocorrência.

A figura 30 abaixo apresenta a página de login que realiza autenticação por meio de um serviço de autenticação que interage com o keycloak, este por sua vez obtêm a lista de usuários de duas fontes: LDAP e Banco de Dados.

Na figura 31 temos as duas páginas (ocorrências e cadastro) e os microsserviços envolvidos:

Como podemos observar a interface de cadastro realiza buscas junto ao microsserviço **Registro Geral** para vítimas e suspeitos identificados e comunicação direta com o microsserviço **Gerenciamento Ocorrência** para inserção, que por sua vez tem por responsabilidade se comunicar com o microsserviço **Gerenciamento Procurado**.

De acordo com a matriz de permissões em Autorização, o usuário pode ter acesso negado a alguns recursos a depender do seu perfil. Sempre que o usuário tenta acessar uma operação fora do seu perfil ele é direcionado para a página abaixo:

Toda e qualquer restrição de acesso ocorre somente quando o microserviço retorna um código 403, portanto, não há lógica implementada no frontend com relação ao que o usuário pode ou não

Figura 30 – Processo de Login

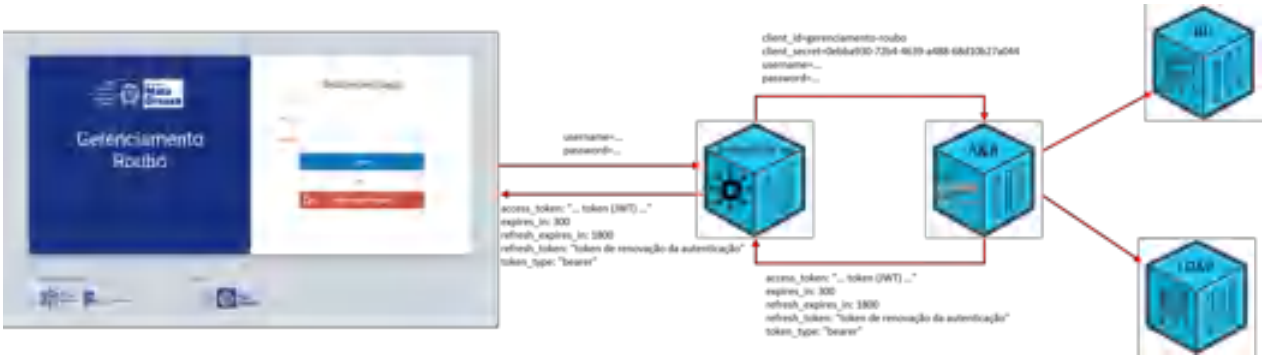


Figura 31 – Páginas ocorrências e cadastro

Figura 32 – Página de Acesso Negado

realizar ou visualizar. Essa estratégia tem como benefício a independência do projeto GUI com os microsserviços, ou seja, alterações de perfis de usuários nos microsserviços não demandam alteração no projeto GUI.

Do ponto de vista prático, um usuário com perfil de acesso de Viatura (VT), por exemplo, poderá visualizar os boletins de ocorrência (BO) e até acessar a página de cadastro de BO, mas quando tentar executar a operação de cadastro terá sua solicitação negada.

5.7 Integração com o Login Social

Para que o mecanismo de login utilizando uma rede social do usuário funcione de maneira correta é necessário que sejam realizadas algumas configurações específicas na aplicação. Para este protótipo funcional, optou-se por contemplar a autenticação utilizando uma conta do Google.

Primeiramente é necessário criar uma aplicação na página do Google Developer, disponível em *https://console.developers.google.com*, e adicionar um cliente para utilizar a API do Google e ter acesso às funcionalidades de autenticação social. Mais detalhes e instruções podem ser encontrados em Autenticação — 4.1-Criação-da-aplicação-no-Google-Developer .

A criação do cliente é realizada acessando o menu Credentials e adicionando credenciais do tipo OAuth, para permitir que a aplicação tenha acesso aos dados do usuário com seu consentimento. Para isso, basta clicar em Create Credentials e depois em "OAuth client ID".

Figura 33 – Criação do Cliente

Feito isso, obtém-se o *Client ID*, identificador do cliente criado que deve ser fornecido pela aplicação para validar as configurações e informações juntamente ao Google.

Para este protótipo funcional, foi gerado um cliente com o nome de **gerenciamento-bo-sesp**, que possui as configurações exibidas na imagem a seguir.

Com o Client ID é possível realizar a autenticação a partir do front-end. A página *https://developers.google.com/identity/sign-in/web/sign-in* apresenta um exemplo de implementação para obtenção de um token com os dados do usuário.

Foram feitas algumas adaptações no exemplo fornecido no último parágrafo para melhorar o funcionamento com o Angular. Com isso foram implementados os métodos a seguir:

```
public googleInit() {
    gapi.load('auth2', () => {
      this.auth2 = gapi.auth2.init({
                                    client_id: '758457946572-
                                        snen85ij17l1r6vfqso430d1ujefjode.
                                    apps.googleusercontent.com',
                                    cookiepolicy: 'single_host_origin',
```

Figura 34 – Criação do Cliente para o Protótipo


```
7                                   scope: 'profile email'
8                                 });
9       this.criaBotaoLoginGoogle(document.getElementById('botaoGoogle'));
10    });
11  }
12
13 public criaBotaoLoginGoogle(element) {
14    this.auth2.attachClickHandler(element, {},
15                                  (googleUser) => {
16                                   this.authService.
17                                   autenticarGoogle(googleUser.getAuthResponse().
                                         id_token)
18                                       .subscribe( async (response) => {
19
20                                         this.authService.setToken(JSON.stringify(
                                               response));
21
22                                         await this.zone.run(  () => {
23                                            this.router.navigate(['boletim-ocorrencia
                                                  ']);
24                                         });
25                                       }, error => {
26                                         this.adicionaMensagemErro();
27                                       });
28                                  },
29                                 (error) => alert(JSON.stringify(error, undefined, 2)
                                       )
30    );
31  }
```

Código 5.3 – Cliente Angular

O método **googleInit()** é responsável por realizar a comunicação com a API do Google, passando na linha 4 o Client ID criado. Na linha 9, é invocado o método **criaBotaoLoginGoogle()**. Esse método cria um botão para autenticação social que obtém o token fornecido pelo Google com as informações do usuário em seu *payload*.

CAPÍTULO 6

Autenticação

Neste capítulo damos início às questões de segurança, mais especificamente os aspectos de identificação dos usuários.

6.1 Introdução

A etapa de **autenticação** consiste em **identificar** usuários e sistemas que desejam acessar os microsserviços e proibir ou redirecionar o acesso a requisições não identificadas. Já a **autorização** é uma etapa posterior e está relacionada aos **privilégios** que são concedidos a determinado usuário, podendo restringir suas ações e acesso aos microsserviços ou parte de seus recursos e operações.

Para prover à arquitetura do protótipo funcional essa capacidade, vamos obervar algumas abordagens possíveis e as características de cada uma, sendo estas:

- Autenticação e Autorização local;
- Autenticação e Autorização global;
- Autenticação global e Autorização como parte do microserviço;

6.1.1 Autenticação e Autorização Local

Neste modelo cada microserviço é responsável tanto pela autenticação quanto pela autorização e estabelece uma forte relação com o conceito de que cada microserviço deve ser autônomo e independente. Apesar de fornecer a forma mais fracamente acoplada, precisamos observar de forma mais próxima as consequências dessa decisão.

6.1.1.1 Aspectos positivos

- Podemos ter implementações completamente diferentes de autenticação e autorização para cada microserviço, reforçando a independência técnica entre os diferentes microsserviços;
- A autorização pode ser realizada de forma mais granular e independente.

6.1.1.2 Aspectos negativos

- Temos um código mais volumoso, já que haverá replicação dos aspectos de autenticação e autorização em cada microserviço;
- A replicação e o código mais volumoso leva a um aumento na chance de erros de implementação e de publicação em ambiente de produção;
- A probabilidade de cada microserviço precisar de mecanismos diferentes de autenticação é muito baixa, o que acarretaria na multiplicidade de um mesmo código em diferentes projetos de microsserviços e elevaria significativamente o esforço de manutenção corretiva e evolutiva desse requisito nos diversos projetos;
- Essa independência entre os microsserviços acarreta em maior complexidade no gerenciamento de uma matriz de permissões.

6.1.2 Autenticação e Autorização Global

Neste modelo temos uma abordagem do tipo tudo ou nada, ou seja, se existe o registro do microserviço e o usuário está identificado então ele está autorizado.

6.1.2.1 Aspectos positivos

- Por ser global não precisamos replicar a implementação desse requisito não funcional;
- A manutenção corretiva e/ou evolutiva se torna mais fácil por conta da centralização;
- A implementação do microserviço permanece focada nas regras de negócio.

6.1.2.2 Aspectos negativos

- Neste modelo o microserviço não tem controle sobre o que o usuário pode ou não realizar, ou seja, uma granularidade mais fina de permissões não é realizada;
- A disponibilidade é prejudicada, tendo em vista que por ser centralizado, uma falha pode causar interrupção do sistema como um todo;
- Ocorrências indesejadas deverão ser averiguadas por meio de auditoria em log, por exemplo.

6.1.3 Autenticação Global e Autorização Local

Neste modelo temos uma abordagem híbrida, a autenticação é global mas a autorização é gerenciada de forma local por cada microserviço.

6.1.3.1 Aspectos positivos

- Granularidade mais fina dos aspectos de permissão;
- O gerenciamento centralizado se torna mais fácil, visto que tem menos responsabilidades;

- Não há latência de rede quanto aos aspectos de autorização;
- Falhas de autorização ficam limitadas ao microserviço em que ocorreu a falha.

6.1.3.2 Aspectos negativos

- Mais código para o desenvolvedor, já que deverá implementar esses aspectos para cada microserviço;
- Matriz de permissões necessária para entender o contexto de permissões do usuário em cada microserviço.

Apesar de entendermos que uma estratégia de autenticação e autorização implementada por cada microserviço é 100% aderente ao fraco acoplamento, autonomia e idependência entre os microsserviços ela é extremamente custosa, trazendo aspectos negativos muito relevantes e benefícios práticos pouco justificáveis, podendo por exemplo, utilizar de redundância para aumentar a disponibilidade de uma autenticação global.

6.2 Ferramentas

Na Tabela 15 apresentamos algumas ferramentas que auxiliam no gerenciamento de identidades e acessos com algumas características mapeadas:

Tabela 15 – Tabela de Ferramentas

	Keycloak	Identity Server	Gluu	CAS	OpenAM	Shibboleth IdP	Lemonl. DAP	Gluu
OpenID Connect /OAuth support	Sim	Sim	Sim	Sim	Sim	Plugin	Sim	Sim
Multi-factor authentication	Sim	Sim	Sim	Sim	Sim	Sim	Sim	Sim
Admin UI	Sim	Sim	Sim	Sim	Sim	Não	Sim	Sim
Identity federation	Sim	Sim	Sim	Sim	-	Sim	Sim	Sim
Open source	Sim	Sim	Sim	Sim	Sim	Sim	Sim	Não
Suporte comercial	Sim	Sim	Sim	Terceiros	Sim	Terceiros	Terceiros	Sim
GitHub Stars	7.2k	7.1k	-	3.3k	266	-	28	-

Este mapeamento foi um exercício inicial para verificar e documentar algumas ferramentas mais utilizadas no mercado e verificar as características do **keycloak** bem como sua atividade na comunidade que se mostra bastante atuante.

6.3 Keycloak

Tendo em vista que a SESP-MT já adota o Keycloak e que está em vias de migrar para um modelo de autenticação e autorização fornecido pela MTI, esta será a ferramenta utilizada no projeto para que o protótipo funcional esteja o mais alinhado possível com a realidade onde será implantado.

6.3.1 Sessão vs Token

No modelo tradicional utilizamos sessão no lado servidor para salvar o estado do usuário autenticado, mas tendo em vista o impacto negativo dessa estratégia no escalonamento horizontal (scale out/in) é que recomenda-se o uso de Token para registrar o status de login de um usuário em uma arquitetura de microsserviços.

A principal diferença está no local do armazenamento, sessões são armazenadas de forma centralizada em um servidor, tokens são mantidos pelos próprios clientes. O token não é apenas um identificador único, ele contêm informações sobre a identidade do usuário e de sua autenticação. Toda vez que uma requisição é enviada, o servidor pode verificar essa identidade e determinar se o requisitante possui ou não acesso ao recurso solicitado.

Contudo, o conteúdo do token precisa estar cifrado para evitar falsificações e por isso utilizaremos JWT (Json Web Token), um padrão aberto (RFC 7519) que define o formato e o conteúdo do token de forma cifrada. Esse padrão possui apoio em diversas bibliotecas nas mais diversas linguagens.

A estrutura de um token JWT consiste de 3 partes:

1. Header: é um cabeçalho com um tipo de valor fixo “JWT” e o algoritmo de hash utilizado;
2. Payload: uma carga contendo um conjunto de informações sobre o usuário e a autenticação como o id do usuário, nome, data/hora em que o token expira, etc;
3. Signature: a assinatura do token que é utilizada para verificar a identidade do token e também se a mensagem não foi modificada ao longo do caminho que ela percorreu.

Essas 3 partes são combinadas utilizando codificação Base64 e se torna a string do token (separada por ponto “.”) que é retornada para o cliente.

O fluxo básico de autenticação utilizando token é apresentado abaixo:

Para mostrar o token JWT e sua composição em 3 partes fizemos um teste com o Insomnia em nosso container keycloak.

Observe na figura acima o token JWT demarcado em vermelho, sendo este:

```
eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6ICJxSzdjZnVHaW
JFNi1PVHM1dHlBdnZGNWJ3aHp6Z05MSFJCU0NrQ3FvaWN3In0.eyJleHAiOjE2
MDMwMjk3NjcsImlhdCI6MTYwMzAyOTQ2NywianRpIjoiMTJkNmVlZTktNTE5MC
00ZmU2LTk4MTUtOWJjMjk3ZDJkODU1IiwiaXNzIjoiaHR0cDovL2xvY2FsaG9z
dDozODA4MC9hdXRoL3JlYWxtcy9kZXYiLCJhdWQiOiJhY2NvdW50Iiwic3ViIj
oiYzEyZGM2ZjEtYjM0Yy00YmQ1LTkyOWUtMjI2NGIwNTI0ZjEyIiwidHlwIjoi
QmVhcmVyIiwiYXpwIjoicmVnaXN0cm8tZ2VyYWwiLCJzZXNzaW9uX3N0YXRlIj
oiNjA4MjA1OTgtYmI0OS00ZTNjLTkwYzUtYWNkMGRhOWVkMzM5IiwiYWNyIjoi
MSIsInJlYWxtX2FjY2VzcyI6eyJyb2xlcyI6WyJvZmZsaW5lX2FjY2VzcyIsIn
VtYV9hdXRob3JpemF0aW9uIl19LCJyZXNvdXJjZV9hY2Nlc3MiOnsiYWNjb3Vu
dCI6eyJyb2xlcyI6WyJtYW5hZ2UtYWNjb3VudCIsIm1hbmFnZS1hY2NvdW50LW
```

Figura 35 – Fluxo básico de autenticação

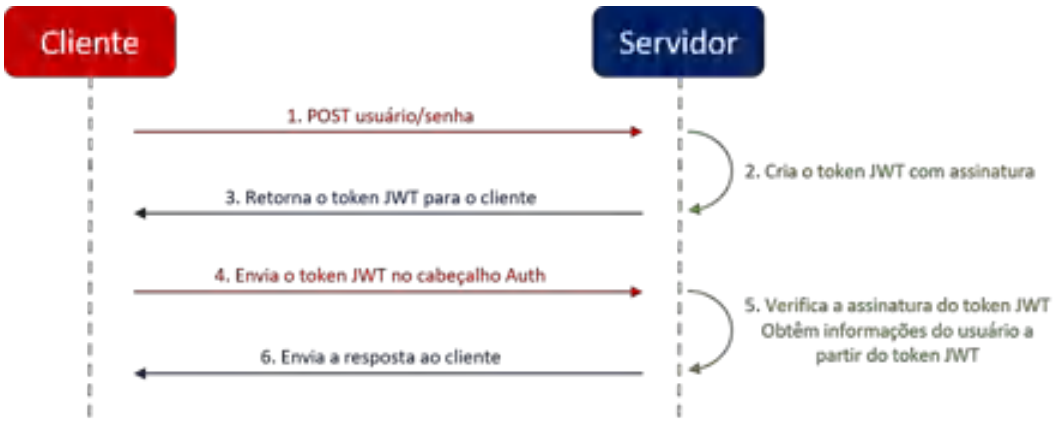
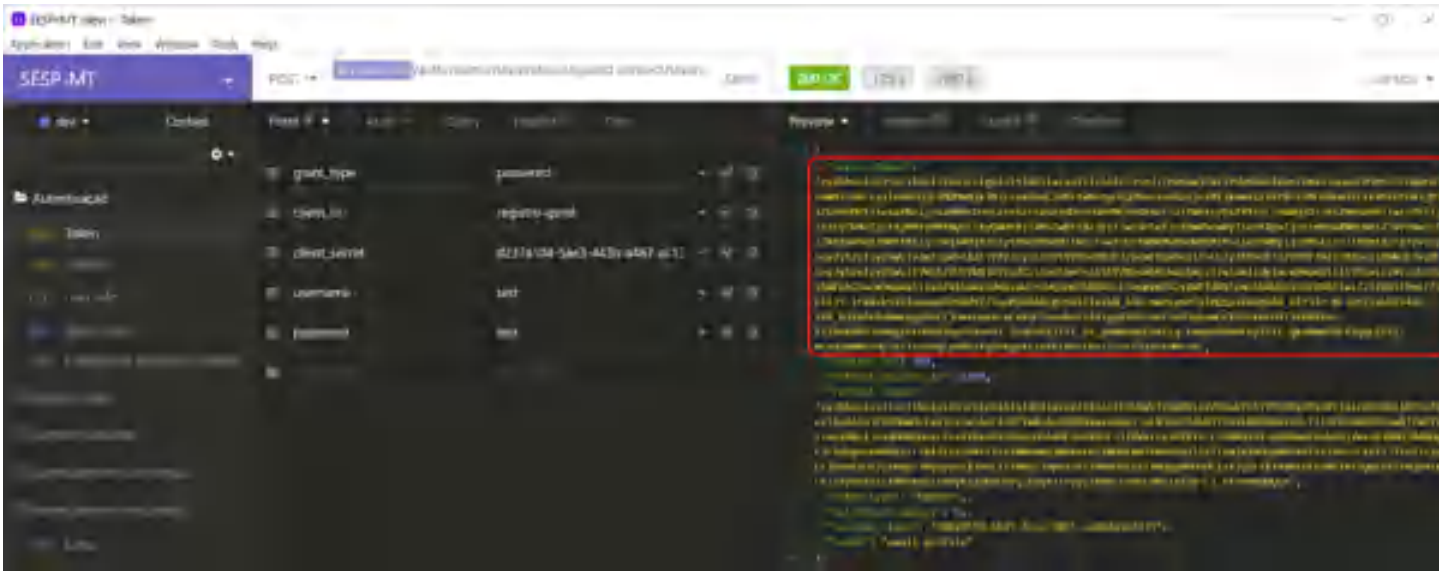


Figura 36 – Token JWT




```
xpbmtzIiwidmlldy1wcm9maWxlIl19fSwic2NvcGUiOiJlbWFpbCBwcm9maWxl
IiwiZW1haWxfdmVyaWZpZWQiOmZhbHNlLCJwcmVmZXJyZWRfdXNlcm5hbWUiOi
J0ZXN0IiwiZ2l2ZW5fbmFtZSI6IiJ9.lrkWsk5ikToaqqKPO50dYZ75qcM70BA
BFgO9o1P1xtbN_h9h-NwesymP5xT0Qhu1kGdgH4B_AIY3Je-
NF3Hf21abP2948R-
ATR_BjXUAFRaNmCqgtHbFZjOwxcugXrvCxkQPluaubvt6f4iggB2UScOeTdsD5
qhvwOLYBJrkHrO9TJKRB91ns-ylIb6XDRP36wRg1KthDHlS4pfGYeM9J_
31qO3UIjiZf_et_ghNDn2wFp4EIq-
EwwpGh9bHEky2TfF_QknMpmFhFY5QqCZfrI-
mCvHzwMMzvHyJ6Z7leOoqLym9bzfq2HtgvXEzdfGI88tLbeZIe-
rfJ1utxDMrvw
```

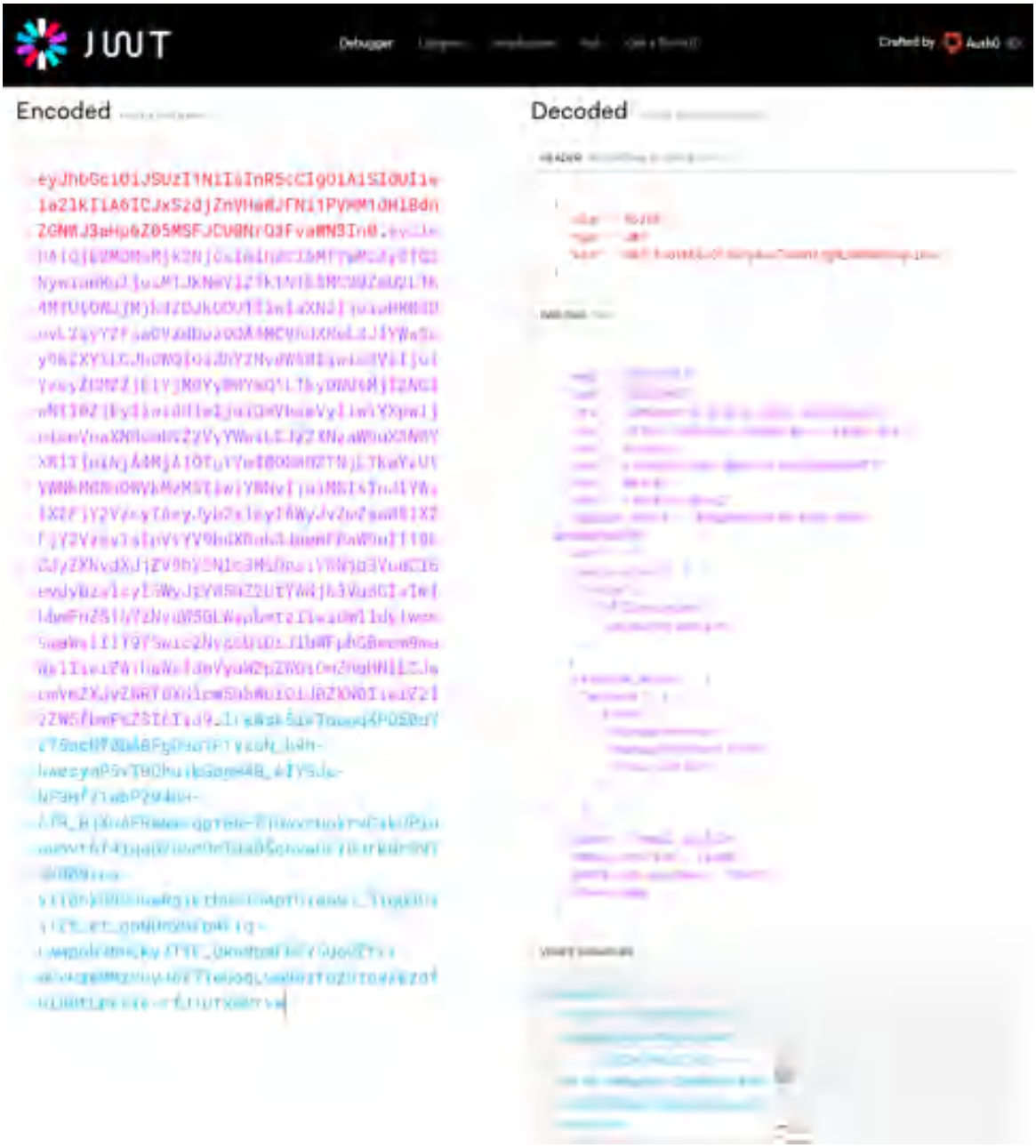
Código 6.1 – Token JWT

Ao postarmos esse token no http://jwt.io obtemos o resultado abaixo:

Perceba que é possível visualizar claramente pela separação de cores as **3 partes** que compõem o token JWT e o conteúdo que ali está representado. Dentro do token existem diversas informações que podem ser inseridas (nome de usuário, perfil, ...), mas nunca devemos inserir dados sensíveis tal como senha.

É importante ressaltar que existem nomenclaturas adotadas pela comunidade para identificação

Figura 37 – Token JWT



dos campos (*reserved claims*) e que estas devem ser mantidas. Podemos elencar alguns atributos não obrigatórios (mas recomendados) que são usados na validação do token pelos protocolos de segurança das APIs.

- **exp** (expiration): Timestamp de quando o token irá expirar;
- **iat** (issued at): Timestamp de quando o token foi criado;
- **iss** (issuer): Emissor do token;
- **aud** (audience): Destinatário do token, representa a aplicação que irá usá-lo;
- **sub** (subject): Entidade à qual o token pertence, normalmente o ID do usuário.

6.3.2 Refresh Token

Vamos observar mais de perto o retorno de nossa autenticação no keycloak:

Figura 38 – Retorno da Autenticação keycloak

Perceba que temos um segundo token (refresh_token) utilizado para renovar a autenticação. Pela configuração de nosso servidor keycloak o token JWT vence após 5 minutos ou 300 segundos (expires_in), mas o refresh token tem um período mais longo de 30 minutos ou 1800 segundos (refresh_expires_in). Renovar uma autenticação é computacionalmente menos custoso do que realizar uma nova autenticação.

A estratégia para renovar a autenticação do usuário podem ser as mais variadas:

- **A cada requisição:** o que gera um número de requisições muito grande e desnecessário tráfego de rede e processamento;
- **Quando receber um erro de autenticação:** o que envolve ter que enviar uma requisição de refresh token e ao receber a atualização reenviar a requisição original que ocasionou o erro;
- **Observar o vencimento do token no cliente antes de enviar uma requisição:** requer que o horário de servidores e clientes estejam em sintonia, se o cliente perceber que o token expirou pode se antecipar e enviar uma requisição de refresh para evitar a resposta de erro;

A observação do *timestamp* de vencimento do token é uma estratégia interessante, tendo em vista a economia dos recursos de rede. Uma alternativa para não se preocupar com a sincronização do relógio entre cliente e servidor está em calcular o tempo de vencimento do token e ao recebê-lo, utilizar o *timestamp* local como sendo o *timestamp* de emissão. A mesma estratégia vale para o timestamp do refresh token.

Erros do tipo **401** demandam uma renovação de token utilizando o *refresh token*, caso este também tenha vencido faz-se necessário uma nova autenticação.

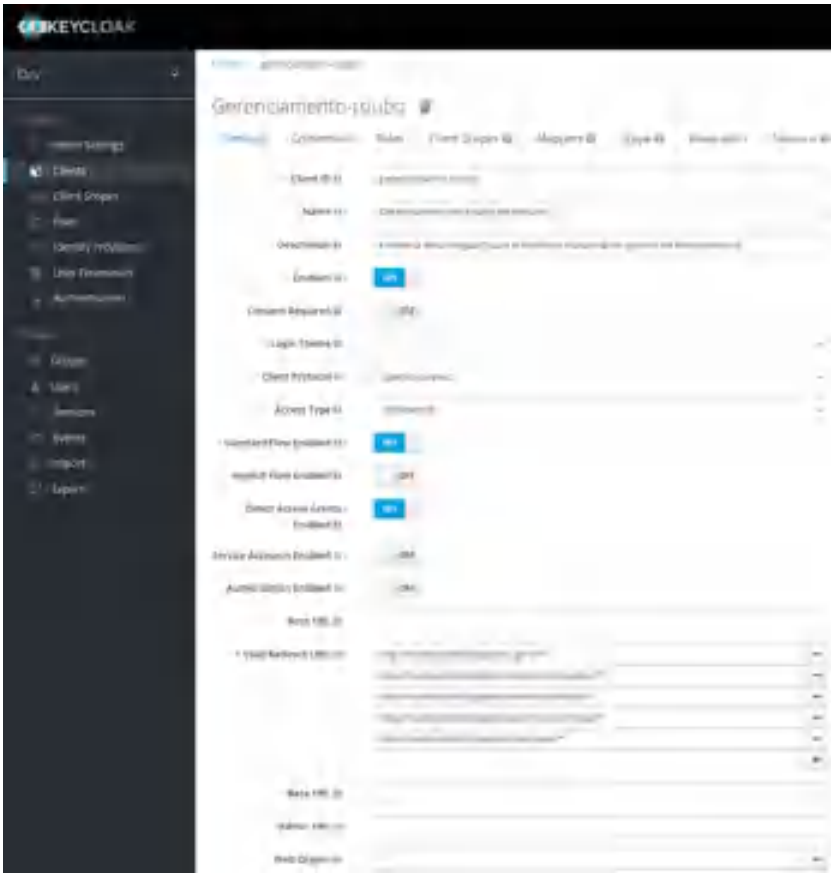
6.3.3 Criando um novo client no Keycloak

Para conectar uma nova aplicação ao Keycloak é necessário, antes, criar um client para esta aplicação nas configurações do Keycloak. Para isso, siga os passos a seguir.

- Acessar o Keycloak em *http://localhost:38080/* com as credenciais *admin/admin*.
- Clicar sobre o nome do realm, por padrão **Master** e criar um novo realm com o nome dev e o display name **Desenvolvimento**.
- Acessar o menu **Clients**, clicar no botão Create e criar um novo cliente para a aplicação que irá ser integrada, com atenção aos seguintes detalhes:
  - Client ID sendo o nome da aplicação, todo em minúsculo, separando os nomes com hífen (Por exemplo gerenciamento-roubo);
  - Client Protocol sendo openid-connect;
  - Root URL em branco.
- Após salvar a informação, uma tela com todas as informações do cliente será exibida. Procure pela opção Access Type e altere o valor para confidential.
- Em Valid Redirect URIs insira a URI local da aplicação, prestando atenção na porta específica. Em nosso caso vamos utilizar um Client ID para os 5 microsserviços, portanto, adicionamos 5 URIs:
  - http://localhost:8001/carteira-funcional/*
  - http://localhost:8002/gerenciamento-ocorrencia/*
  - http://localhost:8003/gerenciamento-patrulha/*
  - http://localhost:8004/gerenciamento-procurado/*
  - http://localhost:8005/registro-geral/*
- Em Web Origins insira a URI "*" para permitir o compartilhamento de recursos de origem cruzada (CORS). CORS é uma especificação que define meios para um servidor permitir que seus recursos sejam acessados por uma página web de um domínio diferente, como estamos tratando o Realm Dev não há problemas em habilitar para qualquer origem.
- Salve as configurações.
- Com isso, a aba Credentials será liberada. Navegue para esta aba para recuperar a informação do Secret desse cliente.

Ao final a configuração deve ser similar a da figura 39 abaixo:

Figura 39 – Configuração keycloak



6.3.3.1 Um Client ID para Múltiplos Microsserviços

No exemplo de configuração do Keycloak para cadastro dos clientes perceba que utilizamos um único Client ID para todos os microsserviços envolvidos no produto de software do Protótipo Funcional.

Essa solução torna mais fácil a administração e implementação dos tokens pela aplicação cliente, caso contrário, cada microserviço teria seu próprio Client ID e a aplicação cliente teria que implementar e manter uma matriz de tokens para os serviços utilizados por ela.

Neste formato, acessos indesejados a um microserviço serão controlados pela autorização e não pela autenticação.

6.3.4 Configurando uma aplicação para conexão ao Keycloak

Abaixo estão os passos para configurar uma aplicação para conexão à uma instância de Keycloak onde o client já foi criado.

- Adicione as dependências de segurança e a dependência específica do Keycloak no pom.xml da aplicação:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
</dependency>

```

```
<dependency>
  <groupId>org.keycloak</groupId>
  <artifactId>keycloak-spring-boot-starter</artifactId>
  <version>11.0.2</version>
</dependency>
```

Código 6.2 – Arquivo pom.xml

- Adicione os dados do Keycloak no arquivo application.properties da aplicação conforme o exemplo a seguir:

```
keycloak.realm=dev
keycloak.auth-server-url=http://keycloak:8080/auth
keycloak.resource=gerenciamento-roubo
keycloak.credentials.secret=0ebba930-72b4-4639-a488-68d10b27a044
keycloak.bearer-only=true
keycloak.use-resource-role-mappings=true
keycloak.ssl-required=none
keycloak.principal-attribute=preferred_username
```

Código 6.3 – Arquivo pom.xml

- Configure a classe principal da aplicação para informar ao Spring Boot que o gerenciamento de segurança não será feito pelas classes internas do framework, e sim por uma ferramenta externa. Para isso, na anotação **SpringBootApplication**, adicione as exclusões conforme o exemplo abaixo:

```
@SpringBootApplication(exclude = {
    SecurityAutoConfiguration.class,
    UserDetailsServiceAutoConfiguration.class
})
public class RegistroGeralApplication {

  // Outras implementacoes

}
```

Código 6.4 – Configuração da Classe Principal

- Crie uma classe para centralizar as configurações do Keycloak. A classe abaixo apresenta uma configuração básica inicial e pode ser criada no pacote **configurations**:

```
package br.gov.mt.sesp.registrogeral.configurations;

import org.keycloak.adapters.KeycloakConfigResolver;
import org.keycloak.adapters.springboot.KeycloakSpringBootConfigResolver;
import org.keycloak.adapters.springsecurity.authentication.
    KeycloakAuthenticationProvider;
import org.keycloak.adapters.springsecurity.config.
    KeycloakWebSecurityConfigurerAdapter;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
```

```
10 import org.springframework.security.config.annotation.authentication.builders.
       AuthenticationManagerBuilder;
11 import org.springframework.security.config.annotation.method.configuration.
       EnableGlobalMethodSecurity;
12 import org.springframework.security.config.annotation.web.builders.HttpSecurity;
13 import org.springframework.security.config.annotation.web.configuration.
       EnableWebSecurity;
14 import org.springframework.security.core.authority.mapping.SimpleAuthorityMapper;
15 import org.springframework.security.core.session.SessionRegistryImpl;
16 import org.springframework.security.web.authentication.session.
       RegisterSessionAuthenticationStrategy;
17 import org.springframework.security.web.authentication.session.
       SessionAuthenticationStrategy;
18
19 @Configuration
20 @EnableWebSecurity
21 @EnableGlobalMethodSecurity(jsr250Enabled = true)
22 class SecurityConfig extends KeycloakWebSecurityConfigurerAdapter {
23
24     /**
25      * Registra o KeycloakAuthenticationProvider como gerenciador de autenticacao.
26      */
27     @Autowired
28     public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
29         KeycloakAuthenticationProvider keycloakAuthenticationProvider =
               keycloakAuthenticationProvider();
30         keycloakAuthenticationProvider.setGrantedAuthoritiesMapper(new
               SimpleAuthorityMapper());
31         auth.authenticationProvider(keycloakAuthenticationProvider);
32     }
33
34     @Bean
35     public KeycloakConfigResolver getKeycloakConfigResolver() {
36         return new KeycloakSpringBootConfigResolver();
37     }
38
39     /**
40      * Define a estrategia de autenticacao
41      */
42     @Bean
43     @Override
44     protected SessionAuthenticationStrategy sessionAuthenticationStrategy() {
45         return new RegisterSessionAuthenticationStrategy(new SessionRegistryImpl());
46     }
47
48     /**
49      * Define as configuracoes padroes de seguranca
50      */
51     @Override
52     protected void configure(HttpSecurity http) throws Exception {
53         super.configure(http);
54         http.csrf().disable();
55         http.cors();
56         http.authorizeRequests().anyRequest().authenticated();
57     }
58 }
```

Código 6.5 – Configurações do Keycloak

6.3.5 Configurando as User Federation do Keycloak

O Keycloak possui a capacidade de se integrar com diferentes estratégias de descoberta de usuários, conhecidas como **User Federations**. Neste projeto foram integradas duas estratégias diferentes:

- Integração com LDAP;
- Integração com base de dados utilizando SPI (Service Provider Interface).

Estas estratégias são tratadas com detalhes nas próximas sessões.

6.3.5.1 Integração com LDAP

O Keycloak permite a integração com um ou mais servidores de LDAP para descoberta de usuários. Esta configuração pode ser alcançada com os seguintes passos:

1. Dentro do realm desejado, acesse o menu User Federation.
2. Na caixa de seleção apresentada, selecione a opção ldap.
3. Uma tela de configuração será apresentada, possibilitando a entrada de dados específicos do servidor do LDAP, conforme apresentado na figura 40.

4. Após a entrada dos dados, sincronize os usuários clicando no botão **Synchronize all users**. Com isso, os usuários importados estarão prontos para serem utilizados para autenticação.

3.4.2. Integração com base de dados utilizando SPI Caso haja a necessidade de se integrar com uma base de dados específica, nas situações onde as credenciais dos usuários são salvas no banco de dados, o Keycloak permite ainda o desenvolvimento de uma **SPI (Service Provider Interface)**.

Uma SPI consiste em um módulo desenvolvido em Java que deve ser publicado no container do Keycloak. Este módulo deve ser desenvolvido com base nas especificações do Keycloak SPI e adiciona a capacidade de interpretação de uma base de dados específica pelo Keycloak.

Neste projeto, o módulo foi desenvolvido como uma aplicação Java, utilizando as seguintes tecnologias.

Tabela 16 – Tecnologias Utilizadas

Tecnologia	Versão
Java	8
Maven	3.x
Hibernate	5.4.12

O código completo para este projeto pode ser encontrado no repositório **sespmt-spi** no GitLab.

A configuração do SPI no Keycloak se resume a adicioná-lo como **User Federation**, visto que todas as configurações de acesso estão presentes no código da aplicação. Para isso, siga os seguintes passos:

Figura 40 – Integração keycloak com LDAP

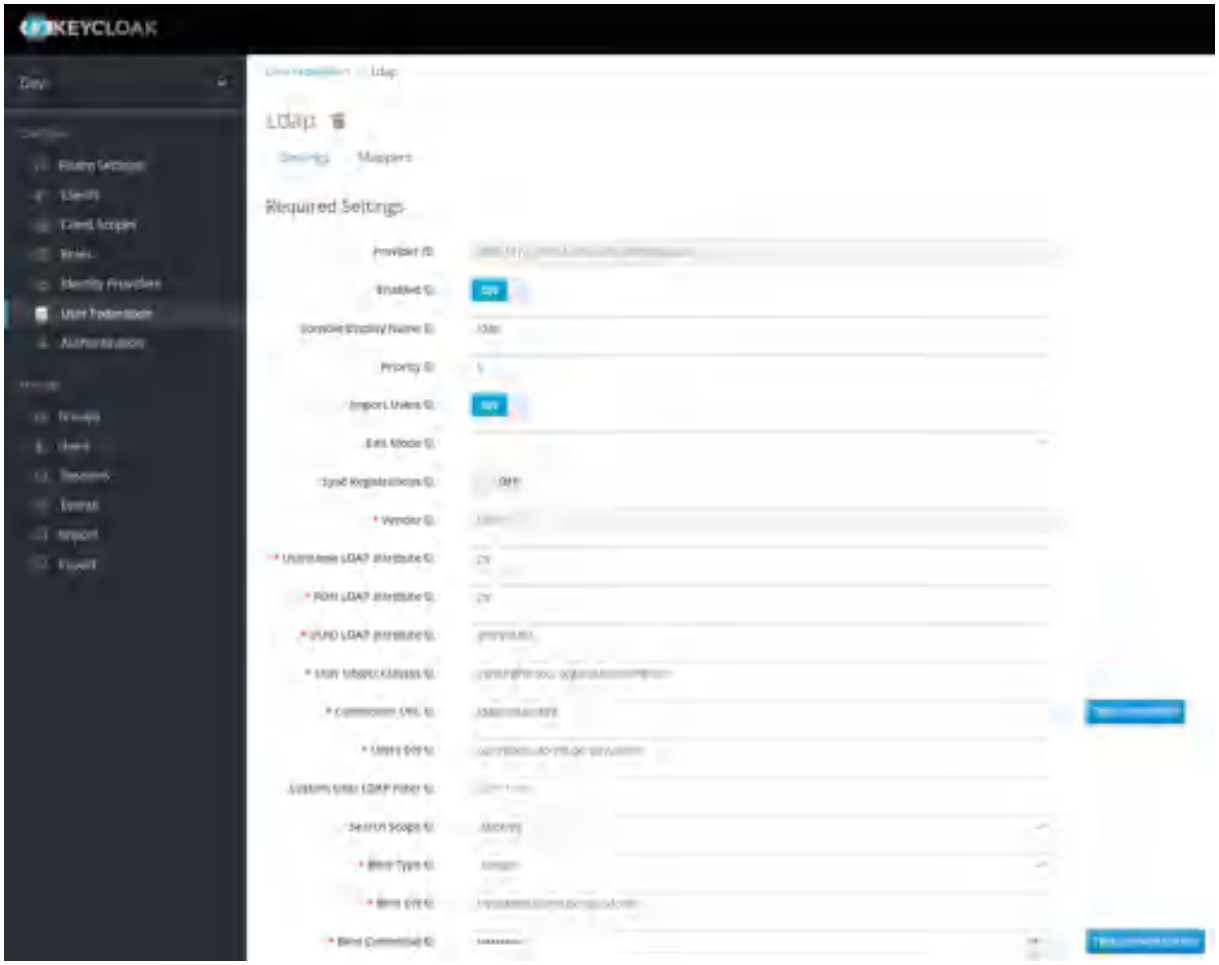


Figura 41 – Sincronização de Usuários

1. Dentro do realm desejado, acesse o menu **User Federation**.
2. Na caixa de seleção apresentada na figura 42, selecione a opção com o nome do SPI desenvolvido e publicado, neste caso sespmt-spi.
3. Na tela seguinte, apresentada na figura 43, garanta que a opção Enabled esteja selecionada e a configuração de cache esteja com o valor NO_CACHE inicialmente. Aconselha-se que este último seja otimizado posteriormente com base na configuração do módulo infinispan, responsável pelo cache do container do Keycloak.
4. Salve as configurações.

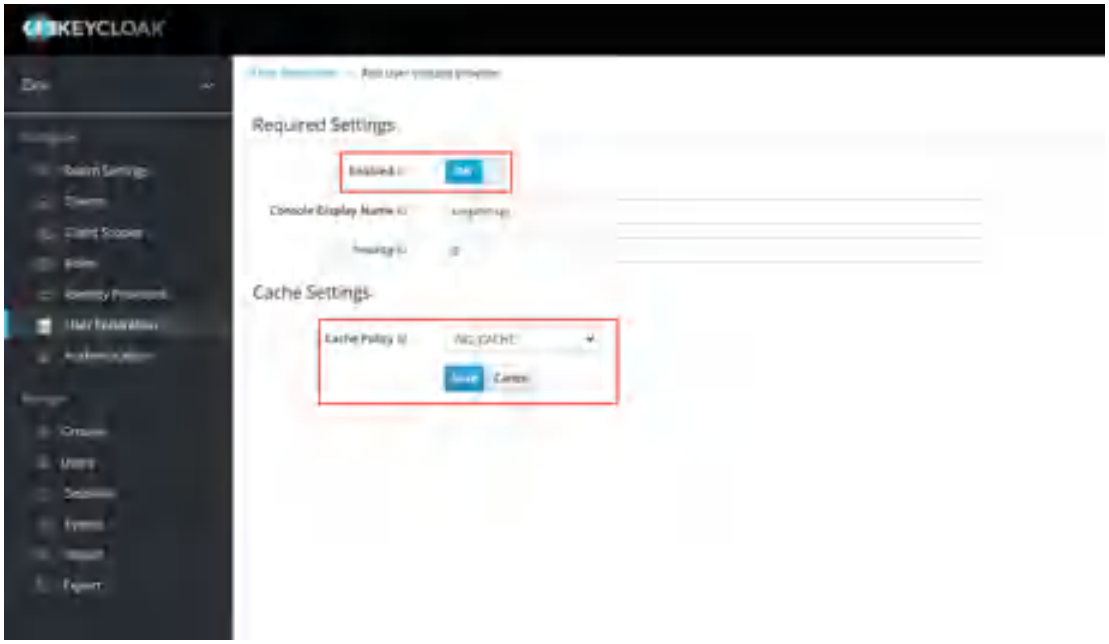
Com isso os usuários já estarão disponíveis para autenticação.

6.3.6 Salvando Alterações

Tendo em vista que a ferramenta de exportação do keycloak é muito precária, não permitindo exportar as User Federation e nem as credenciais dos clientes cadastrados é que optamos por

Figura 42 – Configuração de User Federation

Figura 43 – Configuração de User Federation



realizar o versionamento do volume gerenciado pelo docker.

Nesta solução cada host deve criar o volume **db-keycloak-volume** que será utilizado ao executar o container, sendo este volume versionado no repositório git em **setup/db-keycloak/data**.

Sempre que uma modificação realizada no keycloak quiser ser persistida, deverá realizar **commit** e **push** dos arquivos e pastas alterados nessa estrutura. Vale ressaltar que esse procedimento faz-se necessário apenas para a execução local de todo ambiente dos microsserviços e que na prática, em homologação ou produção, haverá um serviço do keycloak ativo nos respectivos ambientes de homologação/produção.

6.4 Microserviço Autenticação

Tendo em vista o problema enfrentado e descrito na seção 5.2, criou-se um microserviço chamado autenticacao com o papel de Wrapper para os serviços do Keycloak. Essa abordagem traz os seguintes benefícios:

- Reduz a exposição do Keycloak para clientes externos, permitindo limitar a exposição de alguns dados sensíveis de configuração do Keycloak, como o CLIENT ID;
- Permite centralizar configurações e modificações relacionadas a autenticação no microserviço, reduzindo o repasse dessas modificações para os clientes;
- Permite resolver problemas de emissor do token, quando o keycloak é acessado por uma URL diferente da sua rede interna.

O projeto encontra-se em *https://gitlab.com/sesp-mt/autenticacao* - **Connect to preview** e possui as seguintes operações implementadas:

- /keycloak/login: realiza o login com usuário e senha, retornando um JWT;
- /keycloak/refresh: renova o token recebendo o refresh token e retornando um novo JWT;
- /keycloak/info: retorna informações sobre usuário autenticado (necessário informar o token);
- /google/login: a partir de uma autenticação realizada pelo Google, cria um usuário no Keycloak e faz nova autenticação via Keycloak para recebimento de JWT válido.

Tendo em vista a problemática apresentada na seção 5.3, a solução adotada para a realização de Login Social integrado ao Keycloak envolve duas estratégias:

- Utilizar os recursos do Spring Boot para autenticação social, e
- Utilizar a API do Keycloak para poder criar esse usuário e autenticá-lo a fim de poder obter um JWT.

Esse usuário é criado (ou não caso já exista) no Keycloak utilizando os próprios dados obtidos da autenticação na rede social, sendo que a senha definida para esse usuário passa a ser seu ID alfanumérico (sub) gerado pelo próprio keycloak.

Para a implementação desta estratégia de login, foram implementadas as seguintes etapas:

- Criação da aplicação no Google Developer;
- Implementação da chamada à API do Google Developer no frontend;
- Implementação da comunicação do backend com o Keycloak via API.

6.4.1 Criação da aplicação no Google Developer

Para efetuar o login social pelo Google é necessário acessar o painel do Google Developer e criar uma nova aplicação. Com isso serão geradas as credenciais necessárias para integrar a API do Google com o sistema desenvolvido.

O painel do Google Developer está acessível pelo link *https://console.developers.google.com* e, neste caso, deve ser acessado utilizando as seguintes credenciais:

Login: fabsoftsesp@gmail.com

Senha: OLu!g1!o

A configuração da aplicação pode ser feita seguindo as etapas da própria documentação do Google Developer, disponível em *https://developers.google.com/identity/sign-in/web/sign-in*.

6.4.2 Implementação da chamada à API do Google Developer no frontend

Com a aplicação criada, pode-se implementar o acesso pelo frontend conforme especificado na API do Google Developers, disponível em *https://developers.google.com/identity/sign-in/web/sign-in*.

Um exemplo completo da implementação sugerida pelo Google, bem como um exemplo de consumo do *token* de resposta (linhas 19 e 20), pode ser visualizado a seguir.

```
1  <html lang="en">
2    <head>
3      <meta name="google-signin-scope" content="profile email">
4      <meta name="google-signin-client_id" content="758457946572-
           snen85ij17l1r6vfqso430d1ujefjode.apps.googleusercontent.com">
5      <script src="https://apis.google.com/js/platform.js" async defer></script>
6    </head>
7    <body>
8      <div class="g-signin2" data-onsuccess="onSignIn" data-theme="dark"></div>
9      <script>
10       function onSignIn(googleUser) {
11         var profile = googleUser.getBasicProfile();
12         console.log("ID: " + profile.getId());
13         console.log('Nome Completo: ' + profile.getName());
14         console.log('Nome: ' + profile.getGivenName());
15         console.log('Sobrenome: ' + profile.getFamilyName());
16         console.log("Foto: " + profile.getImageUrl());
17         console.log("Email: " + profile.getEmail());
18
19         var id_token = googleUser.getAuthResponse().id_token;
20         console.log("ID Token: " + id_token);
21       }
22     </script>
23   </body>
24 </html>
```

Código 6.6 – Consumo do Token

Neste projeto a estratégia de consumo dos dados do usuário é feita por meio do token de resposta (linhas 19 e 20 do código acima). Este token contém, em seu payload, as informações necessárias do usuário para seu cadastro e validação, conforme demonstrado na imagem abaixo.

Figura 44 – Consumo do Token

Este token, por sua vez, é enviado ao *backend* pelo endpoint **/autenticacao/google/login** para que seja processado e o login do usuário seja validado.

6.4.3 Implementação da comunicação do backend com o Keycloak

O *backend*, por sua vez, recebe o token e verifica se o usuário já está cadastrado no Keycloak. Caso não esteja, um novo usuário será criado com a role **CD** (Cidadão) do tipo client vinculada a ele, utilizando os dados do *token* conforme especificados na tabela abaixo.

Tabela 17 – Dados do Token

Dado do Token	Descrição
email	Utilizado como login e email no Keycloak
sub	Utilizado como senha do usuário no Keycloak
given_name	Utilizado como primeiro nome no Keycloak
family_name	Utilizado como sobrenome no Keycloak

Para cadastro do usuário e vínculo de role, são feitos acessos ao Keycloak via API REST[1]. Os endpoints utilizados da API do Keycloak são descritos na tabela a seguir, na ordem em que são executados.

Após o cadastro do usuário, um novo token de login para este usuário é gerado e disponibilizado como resposta da requisição.

[1] https://www.keycloak.org/docs-api/5.0/rest-api/index.html

Tabela 18 – Endpoints da API Keycloak

Método	Endpoint	Descrição
POST	/{realm}/users	Cria um novo usuário
GET	/auth/realms/master/protocol/openid-connect/userinfo	Obtém os dados do usuário correspondente ao token enviado no cabeçalho Authorization. Necessário para obter o identificador do usuário.
GET	/auth/admin/realms/{realm}/clients	Obtém a lista de clients do realm especificado. Necessário para obter o identificador do client ao qual a role do usuário será vinculada
GET	/auth/admin/realms/{realm}/users/{idUser}/role-mappings/clients/{idClient}/available	Obtém a lista de roles disponíveis no client especificado que podem ser aplicadas ao usuário especificado. Necessário para obter o identificador da role que será vinculada ao usuário.
POST	/auth/admin/realms/{realm}/users/{idUser}/role-mappings/clients/{idClient}	Vincula roles ao usuário especificado. O corpo da requisição deve conter a lista de roles a serem adicionadas, contendo o identificador e nome de cada uma, conforme exemplo: [{"id": "9c3d0f83-a431-42d5-83a3-98b3cf7b9bAd", "name": "CCC"}]

6.5 Observações Importantes

Neste tópico apresentamos alguns desafios encontrados e estratégias aplicadas para vencê-los.

6.5.1 Replicar Configuração do Keycloak

6.5.1.1 Problemática:

A ferramenta de exportação do keycloak é muito precária, não permitindo exportar as User Federation e nem as credenciais dos clientes cadastrados. Essa limitação gera os seguintes problemas:

- Cada desenvolvedor deve realizar uma nova configuração do servidor keycloak, cadastrando as federações e gerando novas credenciais para cada aplicação cliente;
- Tendo em vista que cada desenvolvedor terá credenciais diferentes para a mesma aplicação, cada qual deverá manter os arquivos de configuração da aplicação personalizados para executar de forma exclusiva em sua estação de trabalho.
  - Neste caso, no arquivo application.properties, a linha abaixo ficaria diferente para cada desenvolvedor: **keycloak.credentials.secret=0ebba930-72b4-4639-a488-68d10b27a044**

6.5.1.2 Soluções testadas:

- Mapeamento e versionamento do diretório de dados do banco de dados do keycloak: essa solução encontrou problemas quando o host é windows. Um problema de permissão de pastas do container impedia a cópia dos arquivos. Encontramos diversas recomendações contrárias a essa prática.

- Exportação do script sql do banco de dados do keycloak: essa solução é interessante por conta do tamanho do conteúdo a ser versionado, mas o container do banco de dados do keycloak impedia a inserção de alguns registros por questões de segurança. Quando utilizada a imagem oficial do PostgreSQL, no momento da importação do script completo contendo estrutura e dados do Keycloak, algumas destas estruturas não puderam ser criadas. Foram apresentadas erros de permissão para execução de algumas partes do script. Verificou-se que alguns destes problemas puderam ser tratados com a adequação do script gerado, alterando algumas ordens de execução, no entanto esta solução se mostrou pouco viável, já que o script havia sido gerado utilizando o psql e ainda necessitou de intervenção manual.

- Versionamento do volume gerenciado pelo docker: nesta solução cada host deve criar um volume que será utilizado ao executar o container, sendo este volume versionado no repositório git.

6.5.1.3 Solução adotada:

Versionar o volume (**db-keycloak-volume**) do container **db-keycloak**, que representa a base de dados utilizada pelo keycloak.

6.5.2 Issuer do Token JWT

6.5.2.1 Problemática:

O token JWT recebido, quando solicitado por uma aplicação externa (insomnia), não é aceito pelo próprio emissor (keycloak).

Para testar os microsserviços por meio do insomnia a requisição é feita externa ao contexto dos containers (http://localhost:38080/auth), mas internamente os serviços que estão em execução nos containers utilizam o endereçamento http://keycloak:8080/auth. Essa diferença no host faz com que o emissor do token JWT sejam diferentes, por consequência o keycloak rejeita tokens emitidos por localhost.

6.5.2.2 Soluções testadas:

- Requisitar o token dentro do container: neste caso o emissor do token é keycloak e não localhost e portanto aceito para teste no insomnia:

  - docker exec -it registro-geral /bin/bash
  - curl -d 'client_id=gerenciamento-roubo' -d 'username=jose@email.com' -d 'password=123456' -d 'grant_type=password' -d 'client_secret=0ebba930-72b4-4639-a488-68d10b27a044'

'http://keycloak:8080/auth/realms/dev/protocol/openid-connect/token' — python -m json.tool

- Criação de um serviço utilitário, que será publicado como um container na mesma rede dos outros serviços. Este serviço deverá receber as requisições de criação do token de autenticação e de refresh token e enviá-las internamente para o serviço do Keycloak. Dessa forma será garantido que o emissor do JWT gerado será compatível com os serviços.

6.5.2.3 Solução adotada:

- Criação do serviço utilitário autenticacao, que expõe dois endpoints:
  - /autenticacao/keycloak/login para criação do token de autenticação
  - /autenticacao/keycloak/refresh para atualização do token de autenticação

6.5.3 Autenticação por Login Social

6.5.3.1 Problemática:

Não há integração entre o Spring Boot e o Keycloak quanto ao login social. Identificamos e testamos as seguintes integrações:

- Spring Boot com Keycloak
- Spring Boot com Login Social
- Keycloak com Login Social

O problema surge da necessidade de integrar as três soluções simultaneamente utilizando as soluções pré-definidas de cada uma. Ao fazê-lo identificamos os seguintes problemas:

- Ao habilitar o Keycloak para realizar login social, todas as requisições de autenticação são válidas apenas quando realizadas pela interface de autenticação do Keycloak (inclusive as autenticações que não são por meio de rede social), inviabilizando o serviço de autentição criado no projeto;
- Ao habilitar o Spring Boot para realizar o login social, obtêm-se uma autenticação sem um token fornecido pelo Keycloak, o que inviabiliza a autenticação e a autorização configurada em cada microserviço integrado ao Keycloak.

6.5.3.2 Soluções testadas:

- A não utilização de um microserviço de autenticação, fazendo com que toda e qualquer autenticação seja realizada diretamente pelo Keycloak. Essa solução não permite o uso de uma interface de login personalizada para cada um dos diferentes clientes com acesso ao mesmo Client ID.

- Utilizar recursos do Spring Boot para login social. Essa solução demanda uma implementação da geração e gestão do JWT no microserviço de autenticação e elimina a necessidade do Keycloak. Entendemos que essa solução não é favorável em função de peder recursos altamente personalizáveis, seguros, testados e mantidos pela comunidade.
- Utilizar recursos do Spring Boot para login social e utilizar API do Keycloak no microserviço autenticação para criar dinamicamente esse usuário autenticado no Keycloak, autenticá-lo e receber o JWT assinado, permitindo o acesso aos recursos autorizados a esse usuário.

6.5.3.3 Solução adotada:

- Utilizar recursos do Spring Boot para login social utilizando API do Keycloak.

CAPÍTULO 7

Autorização

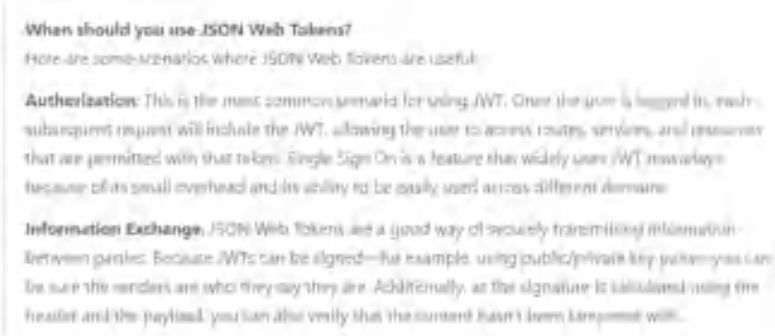
Este capítulo é uma continuação dos aspectos de segurança, mas observando os requisitos de permissão e controle do que pode ser realizado por cada usuário identificado no sistema.

7.1 Introdução

A autorização é uma etapa posterior a autenticação e está relacionada aos privilégios que são concedidos a determinado usuário, podendo restringir suas ações e acesso aos microsserviços ou parte de seus recursos e operações.

Se observarmos o texto extraído na íntegra de *https://jwt.io/introduction/* na data de 05/11/2020:

Figura 45 – Texto Extraído


When should you use JSON Web Tokens?

Here are some scenarios where JSON Web Tokens are useful:

Authorization: This is the most common scenario for using JWT. Once the user is logged in, each subsequent request will include the JWT, allowing the user to access routes, services, and resources that are permitted with that token. Single Sign On is a feature that widely uses JWT nowadays, because of its small overhead and its ability to be easily used across different domains.

Information Exchange. JSON Web Tokens are a good way of securely transmitting information between parties. Because JWTs can be signed—for example, using public/private key pairs—you can be sure the senders are who they say they are. Additionally, as the signature is calculated using the header and the payload, you can also verify that the content hasn't been tampered with.

Perceba que no site oficial a menção é explícita sobre autorização ao contrário de autenticação para o uso de JWT. Isso porque permite ao cliente acessar rotas, serviços e recursos sem a necessidade de verificar constantemente uma entidade centralizada.

Essa característica vai ao encontro da realidade atual dos sistemas que tem diferentes requisitos de arquitetura, aplicativos híbridos, APIs,... que podem depender de vários back-ends, servidores web, de aplicação, bancos de dados, de processamento estatístico, de imagens, etc. Nesses tipos de cenários, o uso de sessão pode ser uma decisão ruim, tendo em vista que a sessão obtida de um servidor não corresponde a de outro e a depender das estratégias de centralização ou replicação e

sincronismo, cria-se um problema de escalabilidade. Esses são alguns apontamentos a favor do uso de JWTs em uma arquitetura de microsserviços, já que não usam sessões e são independentes.

Apesar dessas características relevantes acerca do uso de JWT cabe apresentar alguns pontos de observação e atenção quanto a escolha desse modelo.

Uma vez criado o token você está garantindo essas permissões até que o token perca sua validade, isso quer dizer que alterações de pemissões, de um determinado usuário ou grupo de usuários (ativos), somente serão aplicadas quando seus tokens (que foram emitidos antes da alteração) perderem a validade.

Estratégias que buscam sanar esse problema podem acabar em uma solução centralizada que levantam outros problemas relacionados a disponibilidade e escalabilidade. Podemos citar como exemplo o uso de uma *blacklist* para *tokens* (o que permite invalidar um token antes de seu vencimento). Outra estratégia está em não enviar essas permissões no JWT, o que também acaba por obrigar em cada requisição ter que resgatar essas permissões de outro local, e ao invés de ter que consultar uma *blacklist* terá que consultar uma base de dados de permissões.

Com relação ao overhead para a infraestrutura de comunicação reitera-se que não configura-se problemática relevante, tendo em vista que o token é compacto e o custo para realização de uma nova requisição de checagem de permissão para cada solicitação teria impacto maior.

7.2 Definição de Roles e Autorizações

Recomendamos relembrar o cenário hipotético do protótipo funcional para entender entidades e tipos de usuários participantes do processo de roubo de veículo.

Um Role é um papel, uma função, um cargo, algo que define acesso a um ou um grupo de usuários em partes da aplicação.

Apesar dessa definição, o projeto e a especificação de roles podem variar a depender da estratégia selecionada. Comumente encontramos roles baseadas no que se pode realizar observando os dados, por exemplo roles CREATE PESSOA, UPDATE PESSOA, SEARCH PESSOA, DELETE PESSOA, ... ou roles baseados no perfil do usuário, por exemplo ATENDENTE CAIXA, GERENTE ESTOQUE, VENDEDOR, etc.

Cada uma dessas estratégias de mapeamento de roles possui vantagens e desvantagens e vamos entender as consequências dessas escolhas.

7.2.1 Roles por Operação vs Roles por Perfil de Usuário

Tendo em vista que se trata de uma arquitetura de microsserviços, cujos contextos são menores e divididos em diversos projetos bem delimitados e contextualizados (bounded contexts), cujo perfil de acesso dos usuários é bem conhecido e estabelecido, é que adota-se a estratégia de role por perfil de usuário, permitindo uma leitura semântica no código fonte de cada microserviço quanto aos aspectos de autorização. Esse tipo de estratégia também beneficia o tamanho do token (JWT) que permanece inalterado independentemente da quantidade operações disponibilizadas pelos diversos microsserviços e deixa livre espaço relevante que pode ser utilizado para demandas futuras.

Tabela 19 – Roles por Operação vs Roles por Perfil de Usuário

Características	Role	
	Por Operação	Por Perfil de Usuário
Quantidade de Roles	Cresce acompanhando novas funcionalidades incorporadas ao sistema.	Cresce somente quando novos perfis de usuários são demandados.
Legibilidade do código	Dificultada, visto que no código fonte sabe-se que a operação i.e. `excluirProcurado()` está autorizada para aquele que possui *role* i.e. `DELETE_PROCURADO`, mas só se sabe quais perfil de usuário possuem essa autorização.	Facilitada, visto que o próprio *role* denota o perfil do usuário que terá acesso a operação, por exemplo a operação `excluirProcurado()` possui *roles* `ATENDIMENTO_CIOSP` e `ATENDIMENTO_DELEGACIA`.
Mapeamento	1:1 (Um para Um) Basicamente um role por operação.	1:N (Um para Muitos) Uma operação pode ser acessada por diversos perfis (roles).
Mudança em Autorizações de um Perfil de Usuário	Não necessita alteração no código fonte do sistema.	Necessita alteração no código fonte do sistema.
JWT	*Payload* cresce quanto maior o nível de autorização do usuário.	Praticamente inalterado, visto que o token só precisa carregar o perfil do usuário e não todas as operações a que ele pode ter acesso.

É importante ressaltar também que projetos de microsserviços normalmente estão inseridos em uma cultura de desenvolvimento ágil, em um processo de integração contínua, liberando frequentemente novas versões em produção que não necessariamente possuem novas funcionalidades (otimizações, adequações e/ou manutenções corretivas).

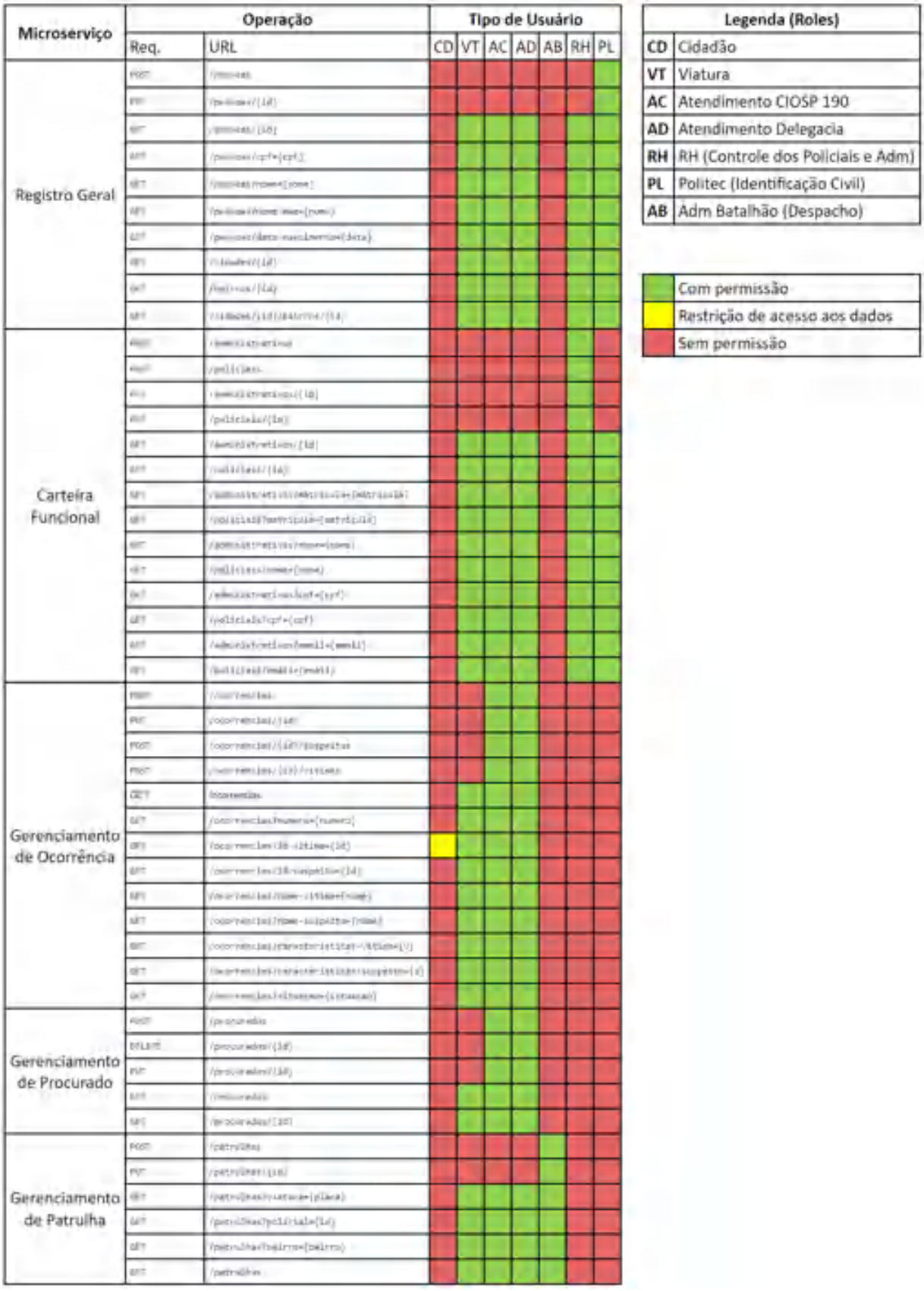
7.2.2 Roles por Perfil de Usuário

Com relação a configuração de roles e pemissões ficou definida conforme:

Para um melhor esclarecimento acerca dos roles estabelecidos, podemos descrever conforme segue:

- **Cidadão (CD):** tem permissão de obter informações acerca das ocorrências, mas apenas daquelas em que ele é vítima;
- **Viatura (VT):** possui apenas permissão para consulta aos dados gerenciados pelos diversos microsserviços;
- **Atendimento 190 (AT):** representa aqueles com permissão para o registro de ocorrências por meio do atendimento 190;
- **Atendimento Delegacia (AD):** representa aqueles com permissão para o registro de ocorrências por meio do atendimento presencial em delegacia;

Tabela 20 – Definições de roles e permissões



Microserviço	Operação		Tipo de Usuário						
	Req.	URL	CD	VT	AC	AD	AB	RH	PL
Registro Geral	POST	/pessoas							
	PUT	/pessoas/{id}							
	GET	/pessoas/{id}							
	GET	/pessoas/cpf={cpf}							
	GET	/pessoas/nome={nome}							
	GET	/pessoas/nome-mae={nome}							
	GET	/pessoas/data-nascimento={data}							
	GET	/biometria/{id}							
	GET	/midias/{id}							
	GET	/midias/{id}/datahora/{id}							
Carteira Funcional	POST	/emissao-orgao							
	POST	/policias							
	PUT	/emissao-orgao/{id}							
	GET	/policiais/{id}							
	GET	/emissao-orgao/{id}							
	GET	/cnh/{id}							
	GET	/administrativo/matricula={matricula}							
	GET	/policial?matricula={matricula}							
	GET	/administrativo/nome={nome}							
	GET	/policiais/nome={nome}							
	GET	/administrativo/cpf={cpf}							
	GET	/policiais?cpf={cpf}							
	GET	/administrativo/email={email}							
	GET	/policial/email={email}							
Gerenciamento de Ocorrência	POST	/ocorrencias							
	PUT	/ocorrencias/{id}							
	POST	/ocorrencias/{id}/suspeitos							
	POST	/ocorrencias/{id}/vitimas							
	GET	/ocorrencias							
	GET	/ocorrencias?numero={numero}							
	GET	/ocorrencias?id-vitima={id}							
	GET	/ocorrencias?id-suspeito={id}							
	GET	/ocorrencias?nome-vitima={nome}							
	GET	/ocorrencias?nome-suspeito={nome}							
	GET	/ocorrencias/caracteristicas/{id}							
	GET	/ocorrencias?caracteristicas-suspeito={id}							
	GET	/ocorrencias?situacao={situacao}							
Gerenciamento de Procurado	POST	/procurados							
	DELETE	/procurados/{id}							
	PUT	/procurados/{id}							
	GET	/procurados							
	GET	/procurados/{id}							
Gerenciamento de Patrulha	POST	/patrulhas							
	PUT	/patrulhas/{id}							
	GET	/patrulhas?placa-viatura={placa}							
	GET	/patrulhas?policiais={id}							
	GET	/patrulhas?bairro={bairro}							
	GET	/patrulhas							

Legenda (Roles)	
CD	Cidadão
VT	Viatura
AC	Atendimento CIOSP 190
AD	Atendimento Delegacia
RH	RH (Controle dos Policiais e Adm)
PL	Politec (Identificação Civil)
AB	Adm Batalhão (Despacho)

	Com permissão
	Restrição de acesso aos dados
	Sem permissão

- **RH:** tem por objetivo representar aqueles com permissão para realizar manutenções nos dados de policiais e servidores administrativos presentes no contexto da carteira funcional;
- **Politec:** está representada apenas para o processo de identificação civil, responsável pela manutenção dos dados do registro geral de pessoas;
- **Adm Batalhão:** representa um direito administrativo sobre o gerenciamento das patrulhas, responsável pelo despacho, mas no caso do cenário do protótipo funcional está envolvido na organização das viaturas e atribuição dos policiais e bairros.

7.2.3 Observações Importantes

Tendo em vista a matriz de operações e autorizações, caso optássemos por uma estratégia de role por operação, poderíamos ter um total de até 48 roles ao invés de 7 roles por perfil de usuário e o JWT do usuário de "perfil atendimento CIOSP" teria um total de 40 roles em seu payload ao invés de um único role.

Quanto ao tamanho do payload é importante não ultrapassar 8kB no token. Apesar de não haver um limite definido na especificação (*https://www.rfc-editor.org/rfc/rfc7519.txt*), o JWT é incluído no cabeçalho HTTP o que limita a 8kB na maioria dos servidores, alguns poucos possuem restrição de 4kB e por isso algumas recomendações de manter o tamanho do token até esse valor.

Na tabela 21 apresentamos as limitações de alguns servidores quanto ao tamanho do cabeçalho:

Tabela 21 – Limitações de Tamanho de Cabeçalho por Servidor

Servidor Web	Limite (kB)
Apache	8
Nginx	4–8
IIS	8–16
Tomcat	8–48

Caso o tamanho do cabeçalho ultrapasse esse limite, o servidor retorna um erro de código **413 Entity Too Large**.

Um JWT padrão já ocupa facilmente mais de 1kB, como em nosso token de exemplo da versão 0.2 que tratava apenas de autenticação, sem autorização. Em nosso exemplo, 48 roles representariam um máximo de 960 bytes (roles com palavras de no máximo 20 caracteres). A depender do tamanho do sistema, quantidade de microsserviços e roles por usuário isso pode representar um consumo expressivo desse espaço restante, restringindo possibilidades para tráfego de outros tipos de dados no token.

Outro ponto de observação é que apesar do JWT ser compacto, se houver a necessidade de trafegá-lo na URL é importante mantê-lo menor do que 2000 caracteres. A RFC 7230 [1] que é uma atualização da especificação HTTP/1.1 recomenda que:

Various ad hoc limitations on request-line length are found in practice. It is RECOMMENDED that all HTTP senders and recipients support, at a minimum, request-line lengths of 8000 octets.

[1] http://tools.ietf.org/html/rfc7230#section-3.1.1

Mas na realidade alguns navegadores, como o Microsoft IE, possuem limite de 2048 caracteres.

Vale lembrar também que um aumento no tamanho do token incrementa levemente seu tempo de verificação, o que é insignificante do ponto de vista de uma única requisição, mas que pode representar algo substancial quando processando milhares de requisições.

Tendo em vista os pontos aqui levantados é que ressaltamos a importância de não carregar dados no token a revelia, mantendo o tamanho do token o menor possível.

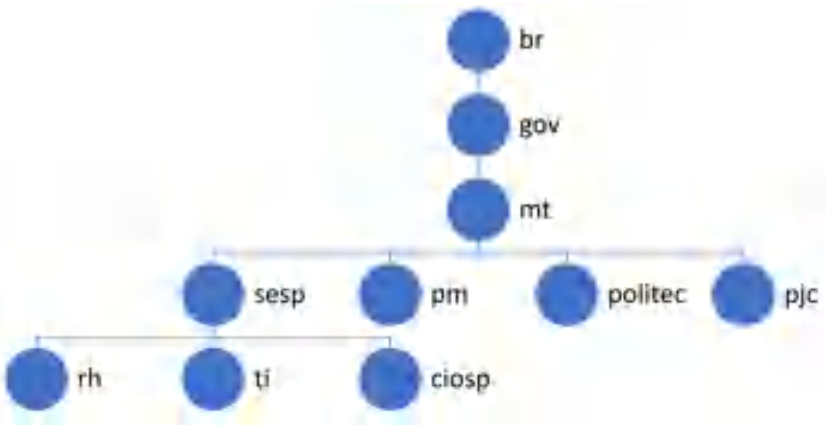
7.3 Configuração das Autorizações

Foi apresentada na autenticação a configuração e obtenção da lista de usuários provenientes de duas fontes de dados, por meio de um banco de dados acessado via SPI e por LDAP.

7.3.1 LDAP

A organização dos usuários no LDAP está representada na árvore ilustrada na figura 46:

Figura 46 – Organização dos Usuários no LDAP



Essa organização da árvore com usuários ancorados em diferentes níveis foi intencional para ilustrar no keycloak como realizar a configuração para busca desses usuários nesses diferentes níveis.

A organização da árvore segue o padrão DNS sendo o login por Common Name (CN) ou também chamado de *Fully Qualified Domain Name* (FQDN). O *Domain Component* (DC) ficou estabelecido conforme:

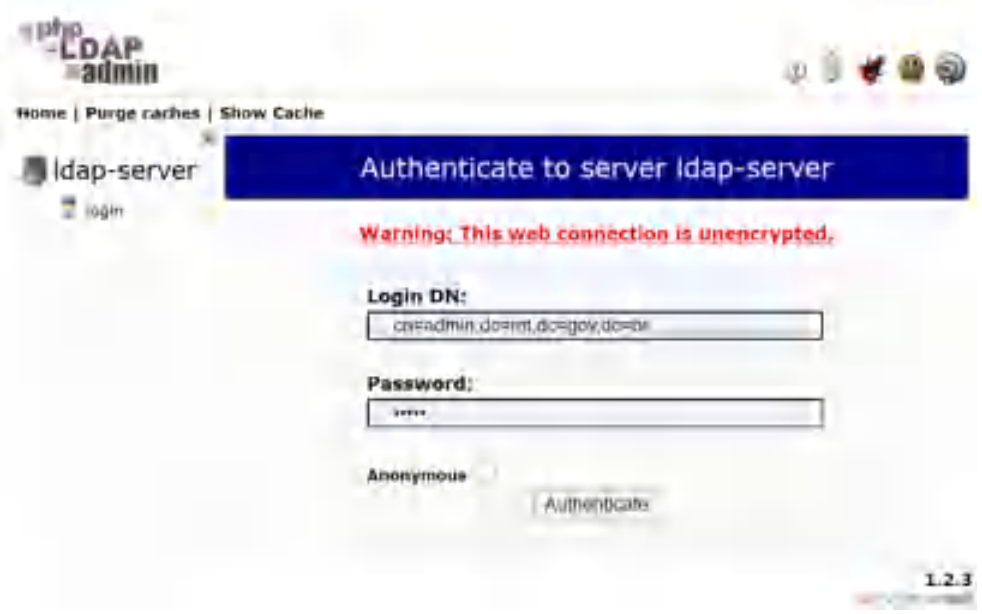
dc=mt,dc=gov,dc=br

Os demais ramos da árvore ficaram representados como Organizational Units (OU) aonde os usuário são inseridos.

7.3.1.1 Acessando o LDAP

Para explorar e navegar na árvore LDAP é possível fazê-lo acessando o PHP LDAP Admin em *http://localhost:3390*, sendo o usuário de login **cn=admin,dc=mt,dc=gov,dc=br** e a senha **admin**, conforme figura 47.

Figura 47 – Autenticação no LDAP



Efetuado o login podemos ver na figura 48 a hierarquia de grupos e usuários configurados para o exemplo do protótipo funcional, bem como roles estabelecidos.

É importante, ao criar o objeto Role, que seu tipo seja groupOfNames, conforme ilustrado na figura 49.

Isso permite ao keycloak atribuir o role ao client roles conforme apresentado na próxima seção.

7.3.1.2 Configurando LDAP no Keycloak

A configuração inicial no Keycloak permanece a mesma realizada na Autenticação, conforme figura 50.

Será necessário complementar essa configuração acessando a aba lateral "Mappers" e criar um novo mapeamento que nós chamamos de **role** sendo do tipo **role-ldap-mapper**.

Importante ressaltar que por padrão, o vínculo de roles é do tipo realm - o valor de "*Use Realm Roles Mapping*". A implicância técnica de se manter o role como realm são basicamente duas:

1. No JWT, a regra é apresentada em uma seção diferente, exclusiva para roles do tipo realm;

2. Por causa disso, no projeto Spring Boot a leitura desses roles não pode ser feita via anotação em cada endpoint. Ao menos não por padrão.

Outro aspecto a ser condiderado é que, do ponto de vista negocial, é coeso que roles sejam do tipo client.

A figura 51 apresenta a configuração realizada para o mapper:

Perceba que estamos informando ao keycloak aonde encontram-se os roles na árvore LDAP (**ou=Roles,dc=mt,dc=gov,dc=br**), os tipo de objetos que deverão ser selecionados **groupOfNames** e que o mapeamento desses roles deve estar associado ao "Client ID" **gerenciamento-roubo**. Com as devidas configurações basta sincronizar "Sync LDAP Roles To Keycloak".

Figura 48 – Hierarquia de Grupos e Usuários

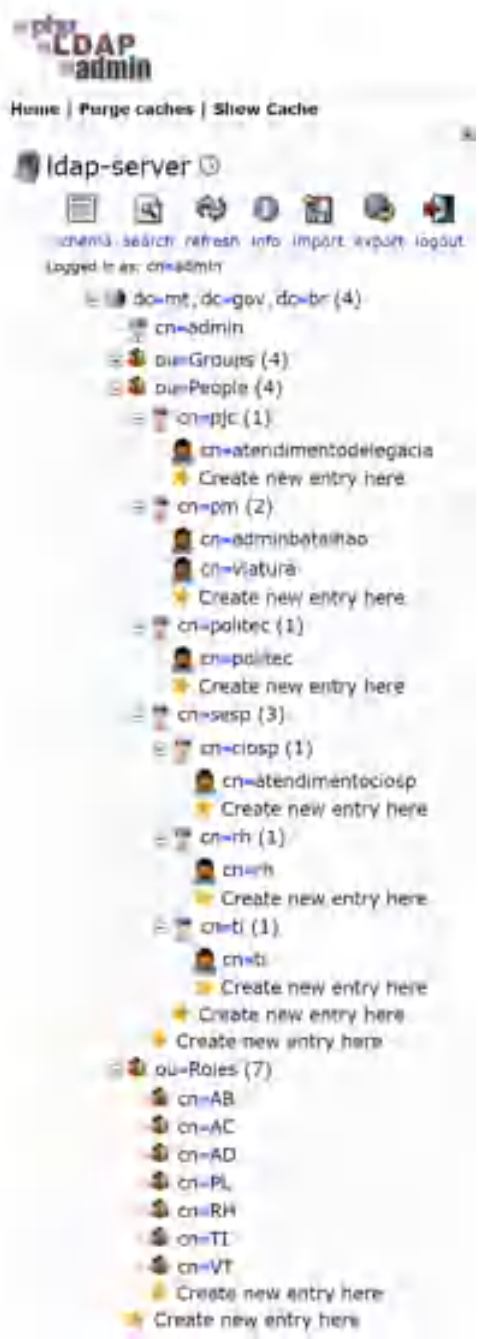
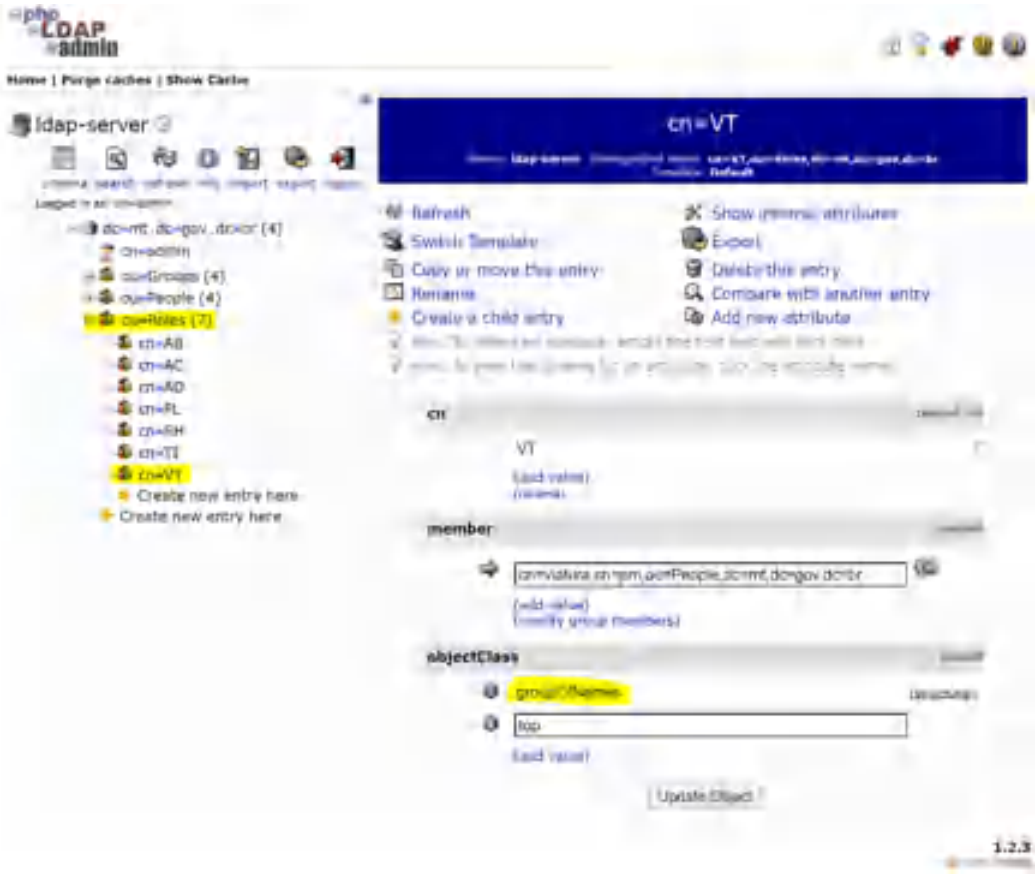


Figura 49 – Configuração de Rules



Após a sincronização podemos acessar o menu “Roles” do Keycloak e observar que o “Client Roles” **gerenciamento-roubo** importou os roles do LDAP e os associou ao Client conforme figura 52:

É importante estar atento para que os roles sejam atribuídos como “Client Roles”, tendo em vista que a autenticação segue o modelo de token para o client **gerenciamento-roubo** que possui um “Client ID” configurado em cada microserviço.

7.3.2 Banco de Dados

O processo de integração do Keycloak com Banco de Dados já está descrito na Autenticação. Descrevemos aqui adequações necessárias no esquema do banco e complementos de configuração para habilitar o recurso de autorização por meio de roles.

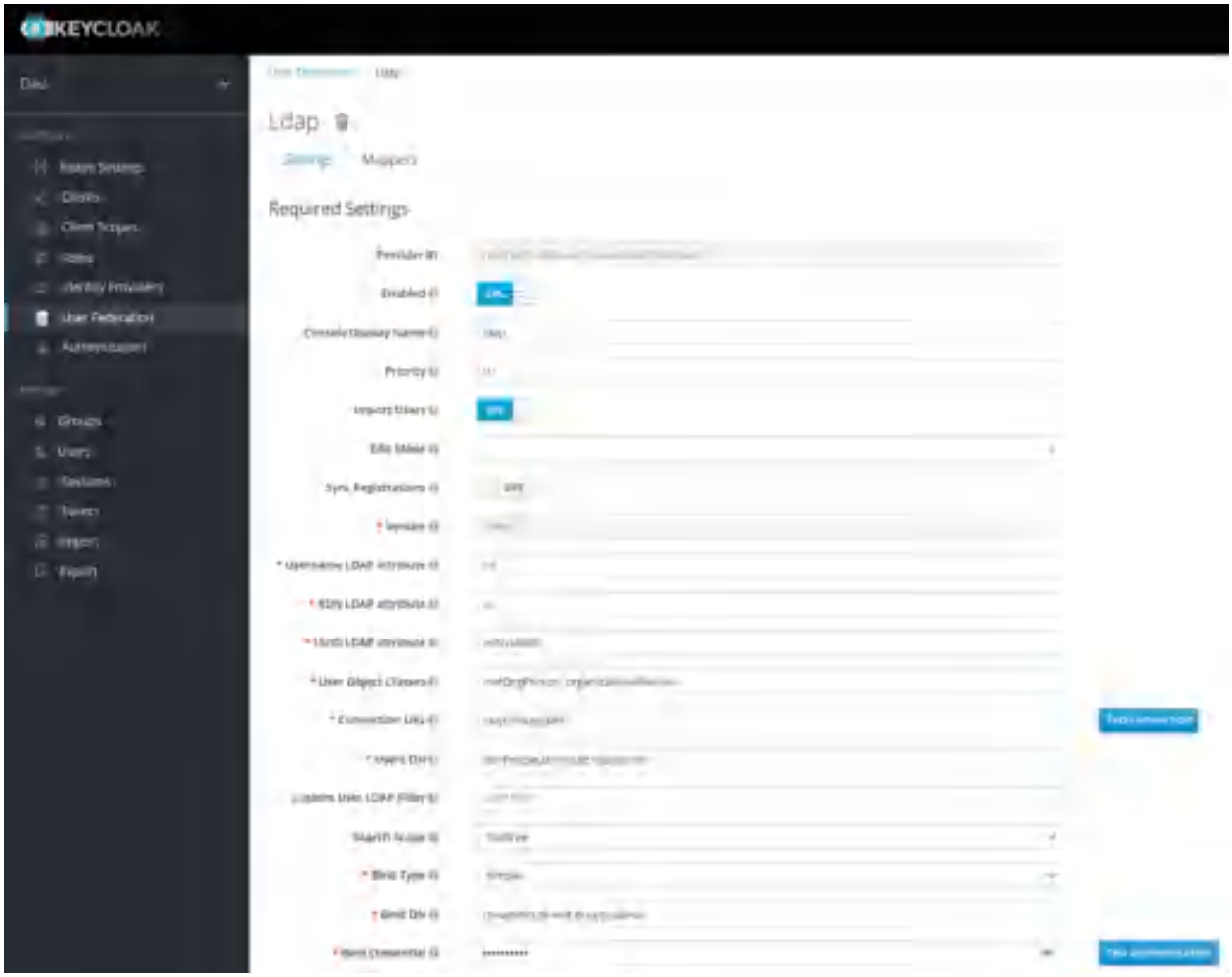
7.3.2.1 Schema utilizado

Basicamente alteramos o esquema do banco de dados para registrar os papéis (roles) que podem ser atribuídos a esses usuários no banco, conforme figura 53:

7.3.2.1.1 Configuração no Keycloak

Para que o SPI possa vincular os roles provenientes do banco de dados, faz-se necessário que estes estejam definidos previamente no keycloak. Mais especificamente é preciso no Client

Figura 50 – Configuração do Keycloak



"gerenciamento-roubo" adicionar roles existentes no sistema legado, conforme ilustrado na figura 54.

Como no banco de dados do protótipo funcional temos apenas usuários com perfil de cidadão cadastrados, apenas o role **CD** (Cidadão) foi adicionado manualmente, os demais vieram automaticamente do LDAP. Isso ocorre porque o SPI não permite criar roles, apenas associá-los.

Entendemos que a forma correta é essa, adicionando manualmente via keycloak ao client, roles que sejam provenientes do banco de dados de um sistema legado. Não é coeso adicionar roles de um banco legado ao LDAP pela conveniência de que o mapper LDAP do keycloak permite importar automaticamente esses roles.

7.3.2.2 Service Provider Interface

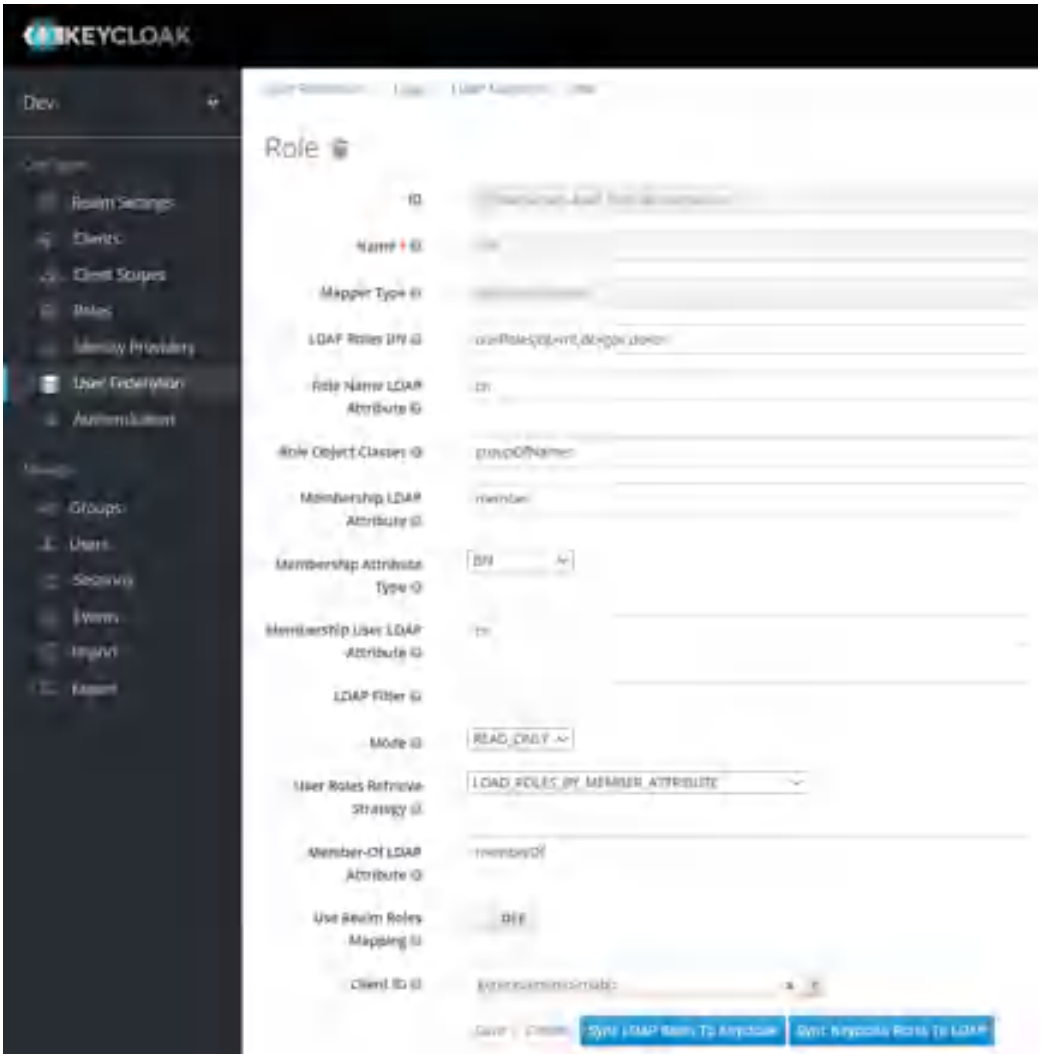
Sobre o SPI a documentação encontra-se na página de Autenticação e o código completo para este projeto pode ser encontrado no repositório sespmt-spi[2] no GitLab.

7.3.3 Microserviço

Quanto a configuração nos microsserviços precisamos apenas importar:

[2] https://gitlab.com/sesp-mt/sespmt-spi.git

Figura 51 – Configuração do Keycloak para mapper



import javax.annotation.security.RolesAllowed;

Agora podemos anotar as operações do microserviço com roles autorizados, por exemplo a operação criar de **PessoaController** do microserviço Registro Geral:

```
...
  @RolesAllowed("PL")
  @PostMapping
  public ResponseEntity<PessoaDTO> criar(
    @RequestBody @Valid PessoaSaveDTO pessoaSaveDTO,
    UriComponentsBuilder uriComponentsBuilder
  ) {
      ...
  }
...
```

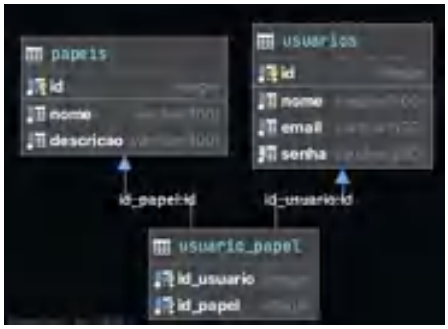
Código 7.1 – Anotação de Roles

Perceba que apenas usuários da Politec (PL) podem cadastrar novas pessoas. Abaixo temos outro exemplo para autorização de múltiplos roles:

```
...
  @RolesAllowed({"VT", "AC", "AD", "RH", "PL"})
  @GetMapping
```

Figura 52 – Acesso ao Client Roles

Figura 53 – Esquema do banco de dados para registrar as Roles

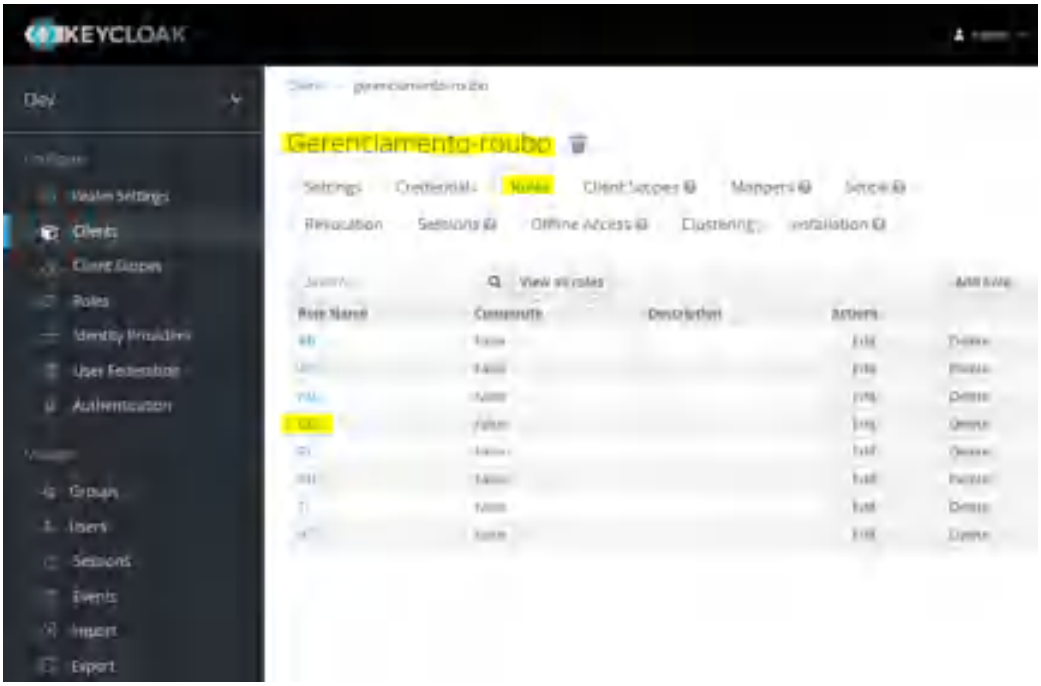



```
4   public ResponseEntity<Set<PessoaDTO>> listar(
5       @RequestParam(name = "cpf", required = false) String cpf,
6       @RequestParam(name = "nome", required = false) String nome,
7       @RequestParam(name = "nome-mae", required = false) String nomeMae,
8       @RequestParam(name = "data-nascimento", required = false) String dataNascimento
9     ) {
10          ...
11  }
12 ...
```

Código 7.2 – Múltiplas Autorizações

Nesse caso a consulta aos dados de pessoas está autorizada para diversos perfis de usuário: Viatura (**VT**), Atendente CIOSP (**AC**), Atendente Delegacia (**AD**), Recursos Humanos ou Gestão de Pessoas da SESP-MT (**RH**) e a Politec (**PL**).

Figura 54 – Roles para Gerenciamento-roubo



CAPÍTULO 8

Persistência

Neste capítulo abordamos como os aspectos de armazenamento e recuperação de dados foram tratados observando as restrições definidas pela SESP-MT.

8.1 Introdução

Antes de discorrermos sobre persistência dos dados é importante deixar claro que o design de microsserviços é realizado por meio da identificação de suas fronteiras, ou seja, um escopo bem definido e delineado. Para identificar esses limites de escopo para cada microserviço recomenda-se o uso de DDD (Domain Driven Design), seu conjunto de princípios com foco em domínio e exploração de modelos ajuda na delimitação de contexto.

No entanto, dada a realidade apresentada pela SESP-MT, partimos de um modelo bottom-up, em que temos uma base de dados já definida, consolidada e utilizada por várias aplicações. Neste caso não temos de fato uma arquitetura idiomática de microserviço, em que respeitamos e seguimos os diversos padrões estabelecidos. Temos um ponto de partida para auxiliar no processo de conhecimento, adaptação e migração, trata-se de uma jornada a ser percorrida.

Por definição microsserviços devem ser fracamente acoplados, o objetivo é que sejam independentes e bem delimitados. Conceber diversos microsserviços que possuem direito de escrita sobre o mesmo banco de dados fere esse princípio, já que os dados podem ser modificados por diferentes atores de forma concomitante, além da estrutura compartilhada, que pode ser modificada ao longo do tempo por demandas de outros serviços.

Quando falamos de uma arquitetura de microserviço há um padrão de projeto chamado **Database Per Service** que estabelece a política de **um serviço por banco de dados**. Isso significa que cada microserviço opera como intermediário entre o cliente e seus dados e estes somente podem ser modificados pela interface desse serviço.

Sobre a política de um serviço por banco de dados, é importante ressaltar que o inverso não se aplica, ou seja, um microserviço pode ter mais de um banco de dados associado. Existem microsserviços que podem adotar a estrutura abaixo:

Figura 55 – Um Banco de Dados por Serviço

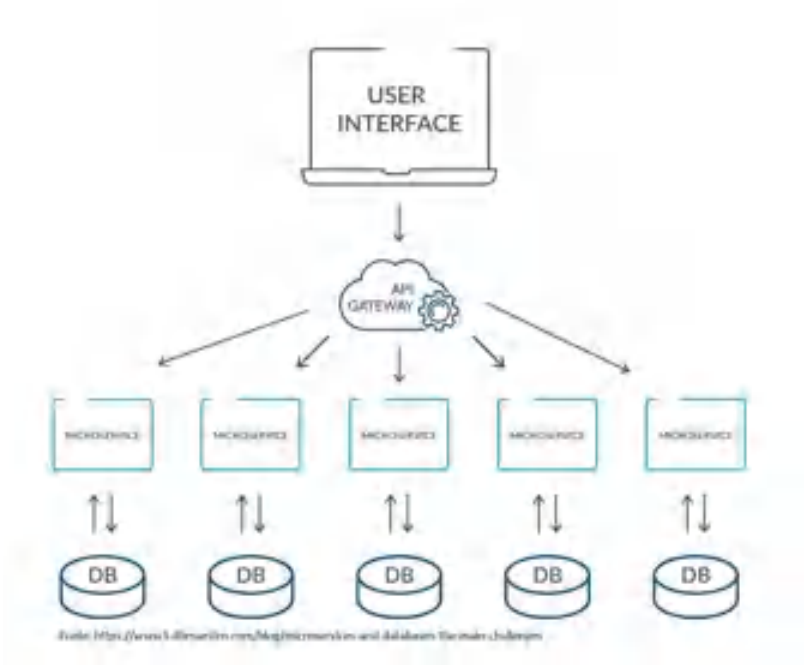
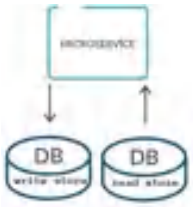


Figura 56 – Mais de um Banco de Dados por Serviço



No entanto, no caso da SESP-MT temos diversas aplicações e bancos de dados já em uso com o desejo de migrar gradualmente os serviços oferecidos para uma arquitetura de microsserviços. A própria SESP-MT apontou a impossibilidade de adotar de forma imediata uma arquitetura de microsserviços em que cada microserviço possui seu próprio banco de dados.

O desafio está portanto, em:

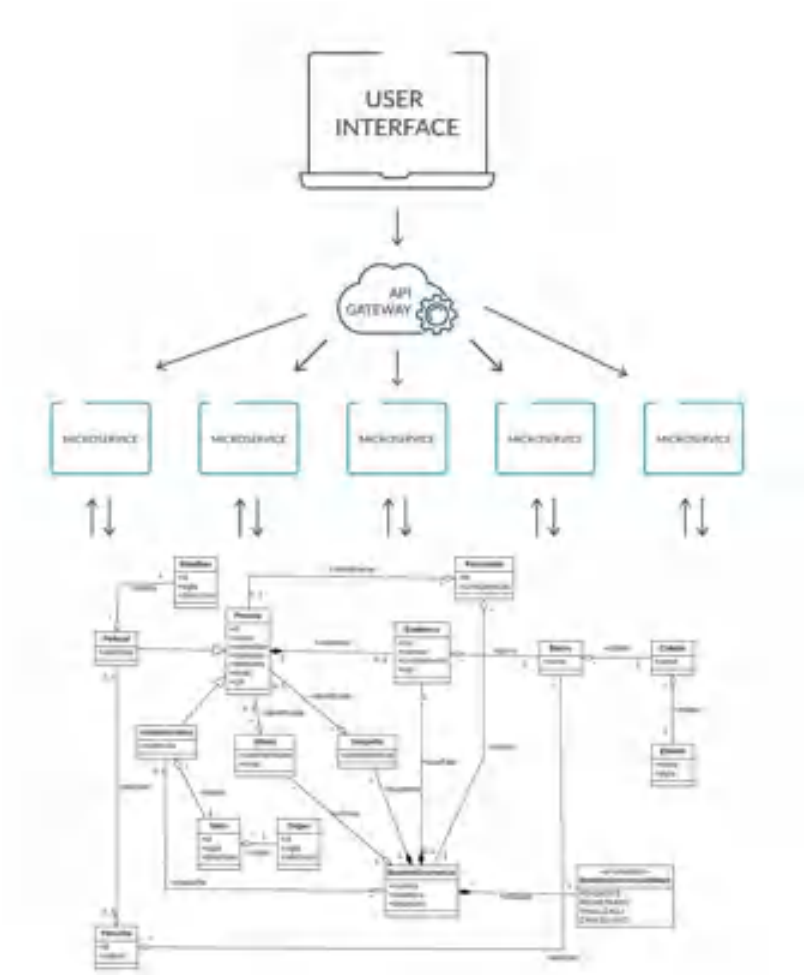
- compartilhar uma base dados para múltiplos microsserviços
  - de forma a minimizar os problemas de consistência e impactos de manutenção ocorridos pela modificação na estrutura dos dados

  e definir uma estratégia de migração
  - que possa ser realizada de forma gradual para os sistemas legados ou monolitos.

A figura 57 representa a estrutura dos microsserviços do protótipo funcional em que todos tem acesso ao mesmo database.

Apesar de não encontrarmos na literatura uma arquitetura de microserviço com base de dados compartilhada e da dificuldade de encontrar experiências documentadas sobre a transformação gradual de um monolito em microsserviços, vislumbra-se que uma alternativa viável está em restringir a visibilidade de cada microserviço aos dados pertinentes ao seu contexto (*bounded*

Figura 57 – Estrutura de Acesso ao Banco do Protótipo Funcional

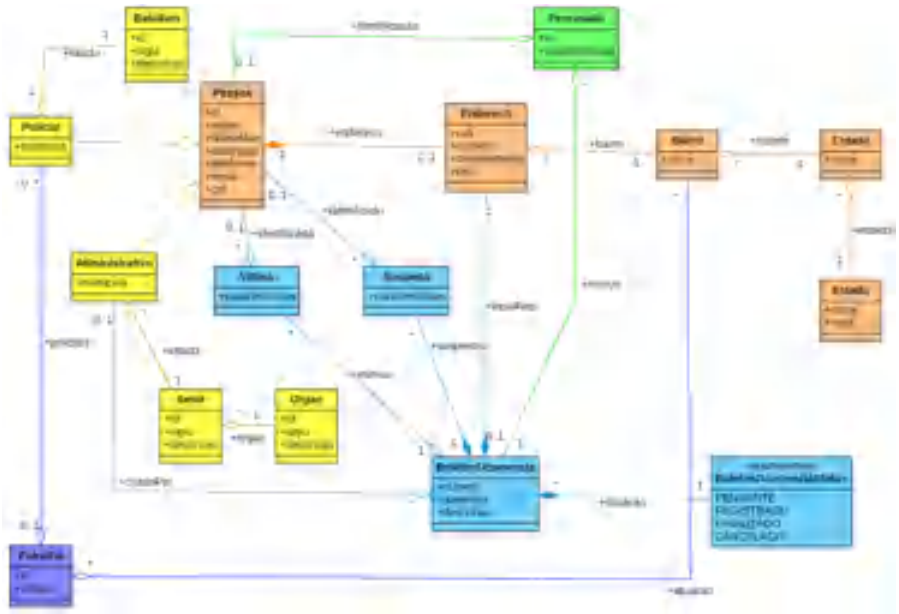


context), realizando dessa forma uma separação lógica. Nessa estratégia cada microserviço terá um usuário próprio no banco de dados com roles que limitem sua atuação dentro do modelo global do repositório.

Nas figuras 58 e 59 apresentamos uma separação por cores do contexto de cada microserviço sobre o domínio global.

Figura 58 – Microsserviços do Protótipo Funcional

Figura 59 – Modelo de Domínio do Protótipo Funcional



8.2 Domínio Global

O objetivo está em representar o contexto de um sistema legado que deseja migrar de forma gradual para uma arquitetura de microserviço. Neste cenário não serão realizadas alterações na estrutura global para atender demandas específicas de cada microserviço. Cada microserviço deverá ter direitos exclusivos sobre a parte do banco de dados pertinente ao seu contexto.

Quanto a conversão desse modelo orientado a objetos para o modelo entidade-relacionamento, adotamos as decisões abaixo:

- em relacionamentos de herança são mantidas todas as entidades, sendo os atributos replicados aos descendentes;
- em relacionamentos de composição (1:1) a entidade a ser composta deixa de existir e seus atributos são absorvidos pela entidade que agrega.

8.2.1 Estratégia de Compartilhamento

É importante destacar que duas estratégias podem ser adotadas para microsserviços que compartilham uma mesma base de dados:

1. **Permitir** a visibilidade e a leitura de entidades que estejam fora de seu contexto;
2. **Bloquear** a leitura de entidades que estejam fora de seu contexto, exigindo que seja realizada uma consulta ao microserviço responsável.

A primeira estratégia economiza recursos de rede e computação, mas tem potencial maior de sofrer impactos devido a mudanças na estrutura dos dados demandados por outro microserviço, além de demandar uma maior gestão pela equipe de banco de dados quanto às autorizações de cada microserviço.

A segunda estratégia tem a seu favor a menor dependência, visto que a visibilidade da estrutura do banco de dados é menor e portanto sofre menos impacto com alterações no schema. Ela também evita o acoplamento não intencional entre serviços, que poderá ocorrer se os serviços compartilharem os mesmos esquemas de dados subjacentes. No entanto, demanda mais recursos de processamento e comunicação, visto que precisa estar sempre solicitando dados aos microsserviços responsáveis. Essa estratégia também facilita a administração do banco de dados quanto a gestão de permissões.

Na tabela 22 apresentamos um resumo dessas características:

Tabela 22 – Resumo de Características das Estratégias de Persistência

Características	Estratégia	
	Permissiva	Restritiva
Performance	melhor	pior
Impactos de estrutura	maior	menor
Migração	dificultada	facilitada
Administração BD	dificultada	facilitada

Pode parecer que a segunda estratégia é natural e ocorreria da mesma forma em uma arquitetura idiomática de microserviço, onde cada qual possui seu próprio banco de dados. Na verdade, quando cada microserviço possui sua própria base, a acomodação de modificações na estrutura é facilitada, sendo possível também realizar tunning e configurações de SGBD de forma personalizada, analisando a realidade de cada bounded context.

A escolha da estratégia **não é mutuamente exclusiva**, mas tendo como objetivo facilitar um futuro processo de migração para uma arquitetura de microserviço, entendemos que a segunda estratégia é mais adequada já que minimiza a dependência com o schema global da base de dados e facilita o processo de separação física do SGBD.

8.3 Domínios por Microserviço

Vamos apresentar e elaborar a restrição de visibilidade para cada microserviço. Em cada microserviço iremos apresentar 2 (duas) figuras: a primeira representa um recorte acerca das entidades/dados de interesse de acordo com o modelo de domínio global (estratégia 1 - permissiva) e a segunda figura o recorte de acordo com a estratégia 2 (restritiva), limitando a visibilidade do microserviço às entidades que ele possui permissão de escrita.

De acordo com a estratégia 2 temos 5 usuários:

- registro_geral
- carteira_funcional
- gerenciamento_patrulha
- gerenciamento_procurado

- gerenciamento_ocorrencia

Por padrão todos esses usuários deverão ter suas permissões revogadas para todas as tabelas do banco de dados. Por exemplo:

REVOKE ALL ON ALL TABLES IN SCHEMA "..."FROM registro_geral;

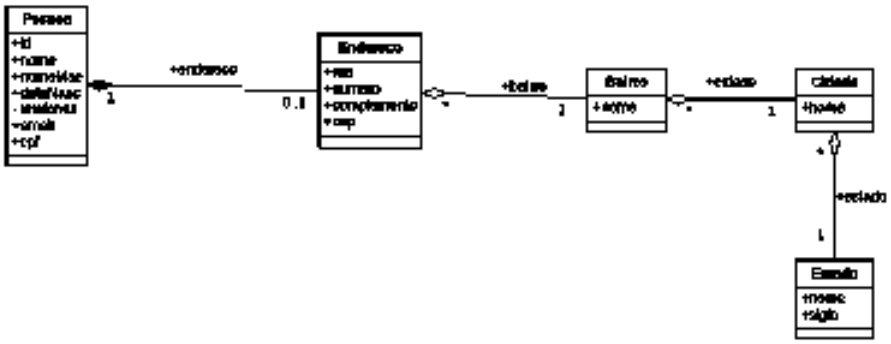
Em seguida iremos atribuir os privilégios de SELECT, INSERT, UPDATE e DELETE para o usuário de cada microserviço em cada tabela do recorte realizado de acordo com a estratégia 2.

GRANT SELECT, UPDATE, INSERT, DELETE ON ¡table¿ TO ¡user¿;

8.3.1 Registro Geral

Este microserviço é responsável pela manutenção das entidades da figura 60.

Figura 60 – Modelo Registro Geral



Este serviço em particular não precisa de um recorte adicional, todas as entidades presentes na figura são de inteira responsabilidade do microserviço Registro Geral e não demanda dados de outros contextos. Dessa forma o usuário registro_geral possui privilégios **SELECT**, **INSERT**, **UPDATE** e **DELETE** para tabelas referentes a essas entidades.

8.3.2 Gerenciamento Patrulha

As entidades ilustrada na figura 61 representam todos os dados de interesse do microserviço Gerenciamento Patrulha.

Perceba que o recorte ilustrado na figura 62 somente permite visibilidade às entidades geradas pelo relacionamento (N:M), obrigando o microserviço Gerenciamento Patrulha a consultar o microserviço Carteira Funcional para obtenção dos dados dos policiais e o microserviço Registro Geral para informações acerca de endereço.

Para este microserviço temos o usuário **gerenciamento_patrulha** que possui privilégios **SELECT**, **INSERT**, **UPDATE** e **DELETE** para a entidade Patrulha e apenas as tabelas de relacionamentos.

Figura 61 – Modelo Patrulha

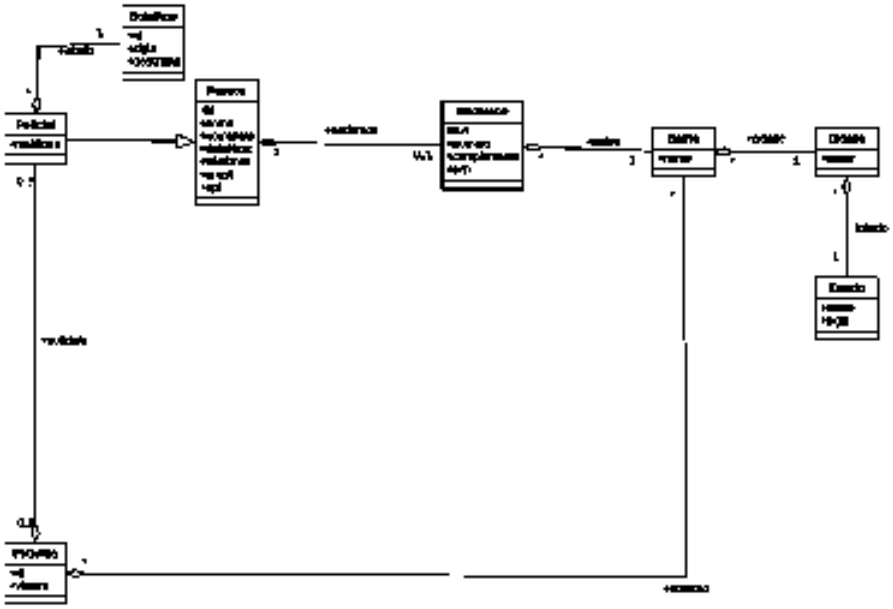


Figura 62 – Recorte do Relacionamento (N:M)

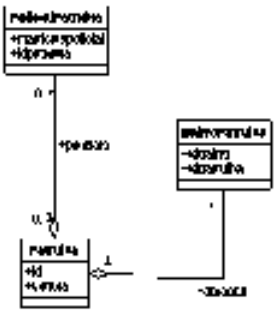
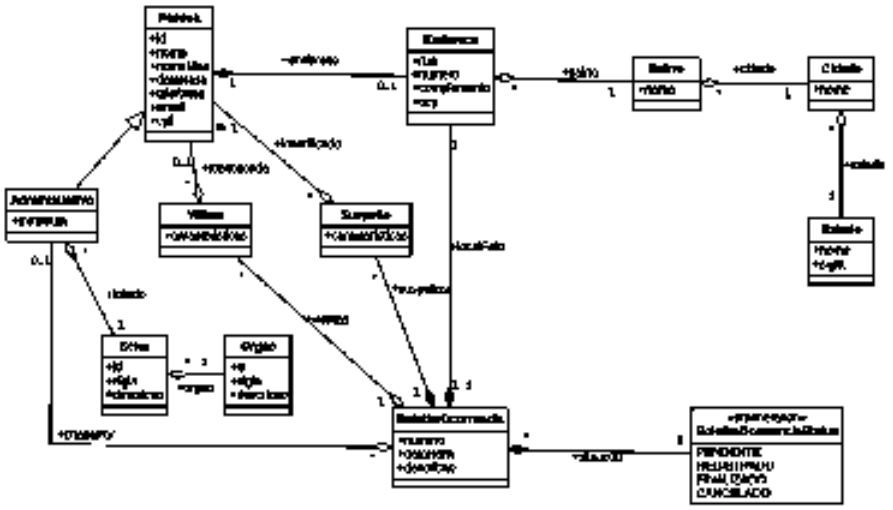


Figura 63 – Modelo Ocorrência

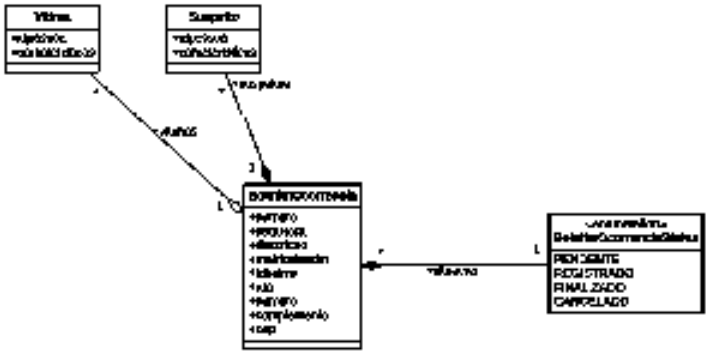


8.3.3 Gerenciamento Ocorrência

As entidades ilustradas na figura 63 representam todos os dados de interesse do microserviço Gerenciamento Ocorrência. Este é o microserviço que abrange o maior número de entidades.

Com o recorte da figura 64, este microserviço precisa se comunicar com os microserviços Registro Geral e Carteira Funcional para obter informações completas acerca do boletim de ocorrência.

Figura 64 – Recorte da Dependência entre Registro Geral e Carteira Funcional

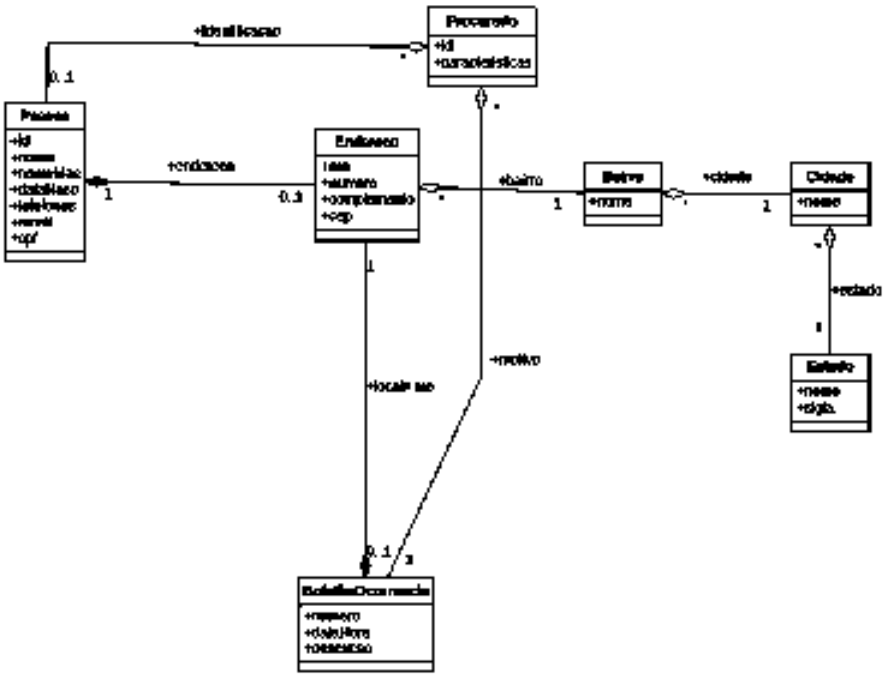


Este utiliza o usuário **gerenciamento_ocorrencia** que possui privilégios **SELECT**, **INSERT**, **UPDATE** e **DELETE** para tabelas referentes a essas entidades.

8.3.4 Gerenciamento Procurado

As entidades ilustradas na figura 65 representam todos os dados de interesse do microserviço Gerenciamento Procurado.

Figura 65 – Modelo Procurado



Este microserviço basicamente possui visibilidade de uma única entidade, ilustrada na figura 66, e precisa consultar os microsserviços Registro Geral e Gerenciamento de Ocorrência para maiores informações.

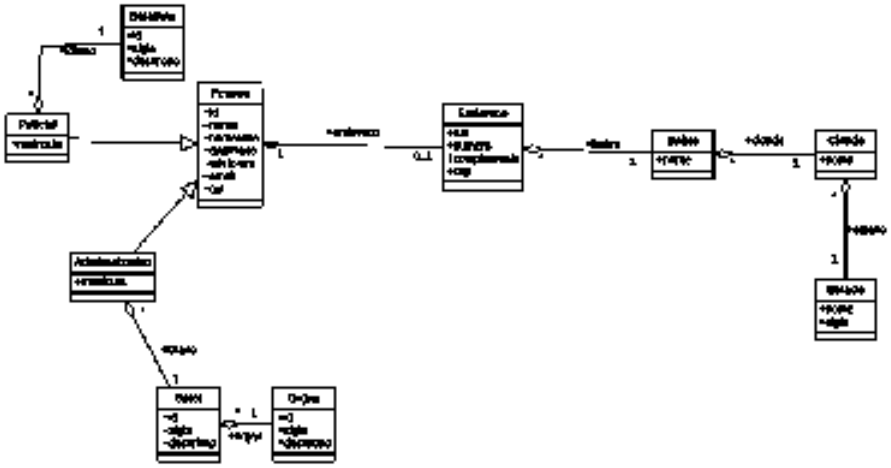
Figura 66 – Entidade Procurado

Este microserviço realiza acesso ao banco de dados por meio do usuário gerenciamento_procurado que posui privilégios **SELECT**, **INSERT**, **UPDATE** e **DELETE** para uma única entidade.

8.3.5 Carteira Funcional

As entidades ilustradas na figura 67 representam todos os dados de interesse do microserviço Carteira Funcional.

Figura 67 – Modelo Carteira Funcional



De acordo com a estratégia adotada, o recorte ilustrado na figura 68 permite minizar impactos de manutenção na estrutura da base dados, visto que dados relacionados à pessoa e respectivo endereço devem ser solicitados ao microserviço Registro Geral, que é o responsável por esse contexto.

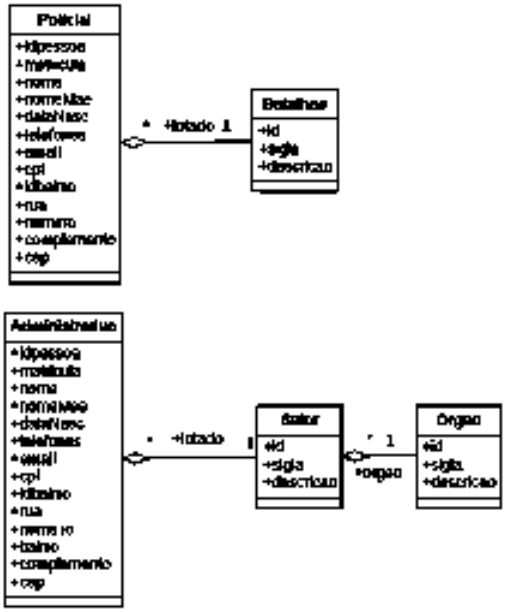
Neste último microserviço o usuário carteira_funcional possui privilégios SELECT, INSERT, UPDATE e DELETE para essas entidades.

8.4 Criação do Banco

Por se tratar de um protótipo funcional o schema apresentado a seguir buscou ser o mais fiel ao modelo para facilitar a leitura e associação dos conceitos.

8.4.1 Tabelas

Figura 68 – Recorte Modelo Carteira Funcional



```sql
CREATE TABLE cidade (
  id SERIAL PRIMARY KEY,
  nome VARCHAR(40) NOT NULL,
  uf CHAR(2) NOT NULL
);

CREATE TABLE bairro (
  id SERIAL PRIMARY KEY,
  nome VARCHAR(60) NOT NULL,
  idcidade INT NOT NULL,
  FOREIGN KEY(idcidade) REFERENCES cidade(id)
);

CREATE TABLE pessoa (
  id SERIAL PRIMARY KEY,
  nome VARCHAR(50) NOT NULL,
  nomemae VARCHAR(50) NOT NULL,
  datanasc DATE NOT NULL,
  telefones VARCHAR(100) NOT NULL,
  email VARCHAR(30) NOT NULL,
  cpf CHAR(14) NOT NULL,
  idbairro INT,
  rua VARCHAR(60) NOT NULL,
  numero VARCHAR(15) NOT NULL,
  complemento VARCHAR(40) NOT NULL,
  cep CHAR(9) NOT NULL,
  FOREIGN KEY(idbairro) REFERENCES bairro(id)
);

CREATE TABLE batalhao (
  id SERIAL PRIMARY KEY,
  sigla VARCHAR(15),
  descricao VARCHAR(50) NOT NULL
);

CREATE TABLE policial (
  id SERIAL PRIMARY KEY,
```

```
38   idpessoa INT,
39   matricula VARCHAR(20),
40   nome VARCHAR(50) NOT NULL,
41   nomemae VARCHAR(50) NOT NULL,
42   datanasc DATE NOT NULL,
43   telefones VARCHAR(30) NOT NULL,
44   email VARCHAR(30) NOT NULL,
45   cpf CHAR(14) NOT NULL,
46   idbatalhao INT,
47   idbairro INT,
48   rua VARCHAR(60) NOT NULL,
49   numero VARCHAR(15) NOT NULL,
50   complemento VARCHAR(40) NOT NULL,
51   cep CHAR(9) NOT NULL,
52   FOREIGN KEY (idbatalhao) REFERENCES batalhao (id),
53   FOREIGN KEY (idbairro) REFERENCES bairro (id)
54 );
55
56 CREATE TABLE orgao (
57   id SERIAL PRIMARY KEY,
58   sigla VARCHAR(15),
59   descricao VARCHAR(50) NOT NULL
60 );
61
62 CREATE TABLE setor (
63   id SERIAL PRIMARY KEY,
64   sigla VARCHAR(15),
65   descricao VARCHAR(50) NOT NULL,
66   idorgao INT NOT NULL,
67   FOREIGN KEY (idorgao) REFERENCES orgao (id)
68 );
69
70 CREATE TABLE administrativo (
71   id SERIAL PRIMARY KEY,
72   idpessoa INT,
73   matricula VARCHAR(20),
74   nome VARCHAR(50) NOT NULL,
75   nomemae VARCHAR(50) NOT NULL,
76   datanasc DATE NOT NULL,
77   telefones VARCHAR(30) NOT NULL,
78   email VARCHAR(30) NOT NULL,
79   cpf CHAR(14) NOT NULL,
80   idsetor INT,
81   idbairro INT,
82   rua VARCHAR(60) NOT NULL,
83   numero VARCHAR(15) NOT NULL,
84   complemento VARCHAR(40) NOT NULL,
85   cep CHAR(9) NOT NULL,
86   FOREIGN KEY (idsetor) REFERENCES setor (id),
87   FOREIGN KEY (idbairro) REFERENCES bairro (id)
88 );
89
90 CREATE TABLE patrulha (
91   id SERIAL PRIMARY KEY,
92   viatura VARCHAR(10) NOT NULL
93 );
94
95 CREATE TABLE patrulha_policial (
96   id SERIAL PRIMARY KEY,
97   idpatrulha INT NOT NULL,
98   idpolicial INT NOT NULL,
```

```
  FOREIGN KEY (idpatrulha) REFERENCES patrulha (id),
  FOREIGN KEY (idpolicial) REFERENCES policial (id),
  UNIQUE (idpatrulha,idpolicial)
);

CREATE TABLE bairro_patrulha (
  id SERIAL PRIMARY KEY,
  idbairro INT NOT NULL,
  idpatrulha INT NOT NULL,
  FOREIGN KEY (idbairro) REFERENCES bairro (id),
  FOREIGN KEY (idpatrulha) REFERENCES patrulha (id),
  UNIQUE (idbairro,idpatrulha)
);

CREATE TABLE boletim_ocorrencia (
  id SERIAL PRIMARY KEY,
  numero VARCHAR(20) NOT NULL,
  datahora TIMESTAMP NOT NULL,
  idbairro INT,
  rua VARCHAR(60) NOT NULL,
  complemento VARCHAR(40) NOT NULL,
  cep CHAR(9) NOT NULL,
  descricao VARCHAR(400) NOT NULL,
  situacao VARCHAR(30) NOT NULL,
  idadministrativo INT NOT NULL,
  FOREIGN KEY (idbairro) REFERENCES bairro (id),
  FOREIGN KEY (idadministrativo) REFERENCES administrativo (id)
);

CREATE TABLE vitima (
  id SERIAL PRIMARY KEY,
  caracteristicas VARCHAR(100),
  idpessoa INT,
  idboletimocorrencia INT NOT NULL,
  FOREIGN KEY (idpessoa) REFERENCES pessoa (id),
  FOREIGN KEY (idboletimocorrencia) REFERENCES boletim_ocorrencia (id)
);

CREATE TABLE suspeito (
  id SERIAL PRIMARY KEY,
  caracteristicas VARCHAR(100),
  idpessoa INT,
  idboletimocorrencia INT NOT NULL,
  FOREIGN KEY (idpessoa) REFERENCES pessoa (id),
  FOREIGN KEY (idboletimocorrencia) REFERENCES boletim_ocorrencia (id)
);

CREATE TABLE procurado (
  id SERIAL PRIMARY KEY,
  caracteristicas VARCHAR(100),
  idpessoa INT,
  idboletimocorrencia INT NOT NULL,
  FOREIGN KEY (idpessoa) REFERENCES pessoa (id),
  FOREIGN KEY (idboletimocorrencia) REFERENCES boletim_ocorrencia (id)
);
```

Código 8.1 – Tabelas

8.4.2 Permissões

```sql
CREATE USER registro_geral;
CREATE USER carteira_funcional;
CREATE USER gerenciamento_patrulha;
CREATE USER gerenciamento_procurado;
CREATE USER gerenciamento_ocorrencia;

ALTER USER registro_geral WITH PASSWORD 'registro_geral';
ALTER USER carteira_funcional WITH PASSWORD 'carteira_funcional';
ALTER USER gerenciamento_patrulha WITH PASSWORD 'gerenciamento_patrulha';
ALTER USER gerenciamento_procurado WITH PASSWORD 'gerenciamento_procurado';
ALTER USER gerenciamento_ocorrencia WITH PASSWORD 'gerenciamento_ocorrencia';

REVOKE ALL ON ALL TABLES IN SCHEMA "public" FROM registro_geral;
REVOKE ALL ON ALL TABLES IN SCHEMA "public" FROM carteira_funcional;
REVOKE ALL ON ALL TABLES IN SCHEMA "public" FROM gerenciamento_patrulha;
REVOKE ALL ON ALL TABLES IN SCHEMA "public" FROM gerenciamento_procurado;
REVOKE ALL ON ALL TABLES IN SCHEMA "public" FROM gerenciamento_ocorrencia;

GRANT SELECT, UPDATE, INSERT, DELETE ON cidade TO registro_geral;
GRANT SELECT, UPDATE, INSERT, DELETE ON bairro TO registro_geral;
GRANT SELECT, UPDATE, INSERT, DELETE ON pessoa TO registro_geral;

GRANT SELECT, UPDATE, INSERT, DELETE ON batalhao TO carteira_funcional;
GRANT SELECT, UPDATE, INSERT, DELETE ON policial TO carteira_funcional;
GRANT SELECT, UPDATE, INSERT, DELETE ON orgao TO carteira_funcional;
GRANT SELECT, UPDATE, INSERT, DELETE ON setor TO carteira_funcional;
GRANT SELECT, UPDATE, INSERT, DELETE ON administrativo TO carteira_funcional;

GRANT SELECT, UPDATE, INSERT, DELETE ON patrulha_policial TO gerenciamento_patrulha;
GRANT SELECT, UPDATE, INSERT, DELETE ON bairro_patrulha TO gerenciamento_patrulha;
GRANT SELECT, UPDATE, INSERT, DELETE ON patrulha TO gerenciamento_patrulha;

GRANT SELECT, UPDATE, INSERT, DELETE ON procurado TO gerenciamento_procurado;

GRANT SELECT, UPDATE, INSERT, DELETE ON vitima TO gerenciamento_ocorrencia;
GRANT SELECT, UPDATE, INSERT, DELETE ON suspeito TO gerenciamento_ocorrencia;
GRANT SELECT, UPDATE, INSERT, DELETE ON boletim_ocorrencia TO gerenciamento_ocorrencia
    ;

GRANT USAGE, SELECT ON SEQUENCE cidade_id_seq TO registro_geral;
GRANT USAGE, SELECT ON SEQUENCE bairro_id_seq TO registro_geral;
GRANT USAGE, SELECT ON SEQUENCE pessoa_id_seq TO registro_geral;

GRANT USAGE, SELECT ON SEQUENCE batalhao_id_seq TO carteira_funcional;
GRANT USAGE, SELECT ON SEQUENCE policial_id_seq TO carteira_funcional;
GRANT USAGE, SELECT ON SEQUENCE orgao_id_seq TO carteira_funcional;
GRANT USAGE, SELECT ON SEQUENCE setor_id_seq TO carteira_funcional;
GRANT USAGE, SELECT ON SEQUENCE administrativo_id_seq TO carteira_funcional;

GRANT USAGE, SELECT ON SEQUENCE patrulha_policial_id_seq TO gerenciamento_patrulha;
GRANT USAGE, SELECT ON SEQUENCE bairro_patrulha_id_seq TO gerenciamento_patrulha;
GRANT USAGE, SELECT ON SEQUENCE patrulha_id_seq TO gerenciamento_patrulha;

GRANT USAGE, SELECT ON procurado_id_seq TO gerenciamento_procurado;

GRANT USAGE, SELECT ON vitima_id_seq TO gerenciamento_ocorrencia;
GRANT USAGE, SELECT ON suspeito_id_seq TO gerenciamento_ocorrencia;
GRANT USAGE, SELECT ON boletim_ocorrencia_id_seq TO gerenciamento_ocorrencia;
```

Código 8.2 – Tabelas

8.4.3 Geração de Amostra de Dados

Para realizar a geração de dados iniciais a serem inseridos na base de dados do projeto, optou-se por criar um programa para automatizar essa tarefa. O projeto do aplicativo pode ser baixado neste link. Após execução do programa, um arquivo SQL contendo dados aleatórios é exportado e pode ser integrado à base de dados da aplicação.

O conteúdo do arquivo é integrado ao script SQL (init.sql) presente no container db-apps, localizado no repositório *https://gitlab.com/sesp-mt/setup* - Connect to preview.

8.4.4 Configuração do Microserviço

A camada de persistência, representada pelo pacote repository, utiliza JPA.

Adicionamos as dependências necessárias:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
  <groupId>org.postgresql</groupId>
  <artifactId>postgresql</artifactId>
  <version>42.2.18</version>
</dependency>
```

Código 8.3 – Dependências

E configuramos o datasource em application.properties:

```
# Datasource
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
spring.jpa.hibernate.naming.physical-strategy=org.hibernate.boot.model.naming.
    PhysicalNamingStrategyStandardImpl
spring.datasource.driverClassName=org.postgresql.Driver
spring.datasource.url=jdbc:postgresql://db-apps:5432/sespmt
spring.datasource.username=registro_geral
spring.datasource.password=registro_geral
```

Código 8.4 – Arquivo de Propriedades

Conforme pode ser observado na configuração do datasource temos um contêiner (db-apps) para a instância do SGBD que está cofigurado no projeto *https://gitlab.com/sesp-mt/setup* - Connect to preview.

Para as classes de repositório utilizamos o recurso de paginação (**Pageable**) disponível do spring data:

```
public interface CidadeRepository extends JpaRepository<Cidade, Long> {

    @Query("SELECT DISTINCT c FROM Cidade c "
            + " WHERE LOWER(c.uf) LIKE LOWER(CONCAT('%', :uf, '%')) "
            + " AND LOWER(c.nome) LIKE LOWER(CONCAT('%', :nome, '%')) ")
    Page<Cidade> findAll(Pageable pageable,
                         @Param("uf") String uf,
                         @Param("nome") String nome);
}
```

Código 8.5 – Arquivo de Propriedades

8.5 Observações Importantes

Nos tópicos a seguir deixamos algumas observações importantes a serem consideradas em projetos futuros de migração.

8.5.1 Identificador Universal

Por se tratar de uma base de dados nos moldes tradicionais, em que ocorrerá um processo de migração como já explanado, é que utilizamos como identificador das tuplas um número inteiro sequencial. No entanto, recomendamos o uso de **UUID** para novos sistemas e novas bases de dados que venham a ser compostas na arquitetura de microsserviços.

Trata-se de um identificador de 128 bits representado por 32 dígitos hexadecimais (36 caracteres com o hifen), que tem como objetivo indentificar unicamente uma informação em sistemas distribuídos, **sem uma coordenação central**. Um dos motivos de uso está em proteger-se de um **vazamento acidental de informação** (como por exemplo dar indícios de quantidade) e também proteger-se da facilidade em **manusear e estimar identificadores** baseados apenas em incremento e decremento (realizados de forma massiva por robôs, por exemplo). Um exemplo é o que aconteceu com vazamento acidental do Twitter em 2015 e fez suas ações caírem 18

Outro aspecto positivo está na facilidade de **sincronização de dados**, caso sua aplicação precise trabalhar offline é possível armazenar os registros localmente sem se preocupar com duplicidade de identificadores, já que os UUIDs são descentralizados e únicos.

Embora o UUID possua diversas implementações, a versão mais útilizada é a v4 que gera o identificador utilizando um número aleatório ou pseudo-aleatório.

8.5.2 Caching

Tendo em vista a característica de sistema distribuído e a granularidade dos microsserviços, o uso de cache a nível do serviço pode ser um grande aliado para melhorar o tempo de resposta.

A organização do cache na arquitetura do sistema pode se estabelecer de diferentes formas a depender do projeto e dos requisitos não funcionais considerados mais importantes, mas comumente encontramos 2 (duas) abordagens sendo utilizadas: **cache embutido e cliente-servidor**. Pra uma reflexão sobre algumas abordagens possíveis em cache recomendamos a leitura desse texto.

No cache embutido temos a vantagem de não nos preocuparmos com a latência, mas o cache cliente-servidor tem a seu favor a escalabilidade.

Caso comece a se preocupar com escalabilidade dos microsserviços, consequentemente em aumentar o número de instâncias simultâneas destes, faz-se necessário pensar em provedores que forneçam a possibilidade de cache distribuído, de forma que as informações armazenadas em cache possam ser compartilhadas entre as instâncias.

Podemos consolidar essas características na tabela 23.

Tomemos como exemplo o microserviço /registro-geral, que é central e demandado por diversos microsserviços. Também é um serviço que gerencia dados que sofrem pouca ou quase nenhuma modificação, como bairro, cidade e estado, excelentes candidatos para cache.

Tabela 23 – Consolidação de Características

Cache	Vantagem	Desvantagem
Embutido	Fácil de configurar e implantar. Baixa latência de acesso aos dados. Sem necessidade de manutenção pela equipe de operação. Impacto de disponibilidade local.	Gerenciamento não é flexível (escalonamento, backup). Limitado a tecnologia utilizada em cada microserviço (Java, python, ...). Dados armazenados e gerenciados junto da aplicação.
Cliente-Servidor	Dados armazenados e gerenciados separados da aplicação. Gerenciamento flexível (escalonamento, backup). Independe da linguagem de programação.	Demanda esforço de equipe de operação. Ajustes de configuração de rede necessários. Maior latência. Impacto de disponibilidade global.

8.5.2.1 Embutido

Primeiro registramos a dependência:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-cache</artifactId>
</dependency>
```

Código 8.6 – Configuração de Dependências

Depois habilitamos no microserviço com **@EnableCaching**:

```
@EnableSpringDataWebSupport
@EnableCaching
@SpringBootApplication(exclude = {
  SecurityAutoConfiguration.class,
  UserDetailsServiceAutoConfiguration.class
})
public class RegistroGeralApplication {
  ...
}
```

Código 8.7 – Configuração EnableCaching

Agora podemos anotar os serviços que se deseja guardar no cache com **@Cacheable**:

```
@Cacheable("cidades")
public CidadeDTO buscar(Long id) {
  ...
}
```

Código 8.8 – Configuração Cacheable

No caso de operações que podem indicar que o cache está desatualizado faz-se necessário indicar a necessidade de atualização com **@CacheEvict**. Neste caso o cache é limpo e reconstruído na próxima cosulta:

```
@CacheEvict(cacheNames="cidades")
public CidadeDTO inserir(CidadeSaveDTO cidadeSaveDTO) {
  ...
```

```
}
```

Código 8.9 – Configuração CacheEvict

No caso da atualização de um objeto é possível atualizar o cache com @CachePut:

```
@CachePut(value = "cidades", key = "#cidadeSaveDTO.id")
public CidadeDTO alterar(CidadeSaveDTO cidadeSaveDTO) {
    ...
}
```

Código 8.10 – Configuração CachePut

8.5.2.2 Cliente-Servidor

Tendo em vista o tamanho do escopo de atuação da SESP-MT e da importância da escalabilidade, recomendamos e disponibilizamos esse modelo no ambiente elaborado do protótipo funcional.

Na busca por um produto que atenda esse modelo de cache distribuído realizamos uma pesquisa sucinta. Observa-se que Redis e Memcached são grandes candidados, sendos estes inclusive disponibilizados pela Elastic Cache da Amazon. Verifica-se que Memcached é direcionado a simplicidade enquanto Redis oferece um conjunto mais avançado de recursos tornando-o flexível para uma variedade maior de contextos. A tabela abaixo apresenta um comparativo extraído da própria amazon:

Tabela 24 – Comparativo Memcached vx Redis

Característica	Memcached	Redis
Baixa latência (sub-millisecond latency)	Sim	Sim
Facilidade de uso	Sim	Sim
Particionamento de Dados	Sim	Sim
Suporte a extensa variedade de linguagens de programação	Sim	Sim
Estruturas de Dados Avançadas	-	Sim
Arquitetura Multithreaded	Sim	-
Snapshots	-	Sim
Replicação	-	Sim
Transações	-	Sim
Publish/Subscribe	-	Sim
Script Lua	-	Sim
Suporte Geospacial	-	Sim

Dado o conjunto mais avançado de recursos selecionamos Redis, um produto open source que permite estruturar dados em memória principal e que pode ser utilizado para cache. Foi disponibilizado mais um contêiner para esse serviço.

Uma vez que o contêiner do Redis esteja disponível, a configuração do cache com Redis no serviço se resumirá em duas etapas: adicionar a dependência e configurar as propriedades.

1 - No arquivo **pom.xml** deve-se adicionar a dependência **spring-boot-starter-data-redis**:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
```

Código 8.11 – Configuração do pom.xml

2. No arquivo **application.properties** deve-se adicionar as propriedades do redis, com os valores correspondentes:

```
spring.cache.type=redis
spring.redis.host=redis
spring.redis.port=6379
spring.redis.timeout=5000
```

Código 8.12 – Propriedades do Redis

5.3. Redundância de Dados

Redundância de dados não necessariamente é algo ruim, ainda mais em sistemas distribuídos em que a autonomia de cada nó é importante. Neste caso a redundância não tem por objetivo melhorar a performance e reduzir a latência (o cache tem esse propósito), mas sim garantir a autonomia do microserviço, garantir que a indisponibilidade de um serviço não se propague completamente para outros serviços.

Este é um tópico muito sensível e deve ser utilizado com muita criticidade, a duplicidade de um dado pode vir a ser cogitada em serviços extremamente críticos, cuja indisponibilidade é altamente prejudicial ao negócio. Pode ocorrer quando o dado tem interpretação diferente a depender do contexto, por exemplo o endereço no registro geral ser relativo a residência e na carteira funcional a um endereço comercial ou de contato (que pode ser diferente da residência).

Caso a duplicidade venha a ocorrer por questões de disponibilidade é importante se ater ao príncipio "Uma única **fonte de confiança**", ou seja, busque sempre o dado consistente e atualizado da fonte de confiança.

Tomemos como exemplo o microserviço Gerenciamento Patrulha e vamos supor que o nome do bairro é informação crítica e essencial no gerenciamento e notificação das atividades de patrulha para ocorrências detectadas e/ou notificadas pela central de emergência.

O nome do bairro é um dado que dificilmente sofrerá modificação, ele poderia ser persistido pelo microserviço Gerenciamento Patrulha, garantindo que mesmo na indisponibilidade do microserviço Registro Geral seja possível obter as patrulhas por bairro.

Isso não quer dizer que o acesso ao microserviço Registro Geral (fonte de cofiança) não deva ser realizado após a persistência desse dado, mas caso haja indisponibilidade o dado local pode ser recuperado para manutenção das atividades de patrulhamento.

8.6 Processo de Migração

Migrar um monolito para uma arquitetura de microsserviços não é uma atividade trivial, é complexa e demanda grande esforço dada a necessidade de coexistirem por um período de tempo. O processo apresentado na figura 69 não foi extraído de nenhuma literatura e representa uma

possível abordagem elaborado pela equipe deste projeto para auxiliar no processo de separação da base de dados.

Figura 69 – Processo de Migração

Transformar: representa a primeira fase do processo (bottom-up), esta etapa consiste:

- da identificação de um módulo, componente ou conjunto de funcionalidades correlacionadas;
- da identificação e separação lógica dos dados pertinentes a esse contexto e;
- da implementação do microserviço.

Coexistir: nesta fase devemos garantir que o recorte realizado na etapa anterior possa coexistir com o monolito, permitindo duas portas de entrada para a realização dos mesmos serviços.

Estrangular: consiste na refatoração gradual dos módulos, componentes ou conjunto de funcionalidades presentes no monolito que dependem do recorte selecionado na primeira etapa;

Separar: nesta etapa devemos ter garantido de que toda a manipulação de dados (da segmentação realizada na etapa de transformação) ocorre apenas por meio do microserviço e não mais pelo monolito, permitindo a separação física dos dados para um banco de dados exclusivo para o microserviço.

Eliminar: após a separação física dos dados é possível remover a estrutura e os dados do banco global (monolito) e desativar/remover o componente, módulo ou conjunto de funcionalidades não mais utilizadas.

Reprojetar: a partir deste momento a equipe tem liberdade para repensar todo o contexto do microserviço, rever suas fronteiras, desmembrá-lo se necessário, repensar a estrutura dos dados,... Recomenda-se refletir sobre o domínio observando os princíptios DDD para então projetar como representar esses conceitos fisicamente no banco de dados, ou seja, ser guiado pelo modelo de domínio e não o contrário (top-down).

Para eleger qual componente do monolito começar a fase de transformação elencamos algumas características que podem auxiliar nessa escolha:

- Para uma primeira escolha selecione um **componente simples**, isso irá ajudar a adquirir prática e conhecimento do processo, dos desafios e de melhores práticas;
- Um componente com **boa cobertura de teste e baixo débito técnico** pode ajudar a equipe a ter mais confiança no processo de migração;
- Componentes com requisitos de escalabilidade também são boas escolhas;

- Componentes que são **modificados frequentemente**, que absorvem novos requisitos ou que precisam ser implantados com mais frequência podem ajudar no processo de migração, visto que reduzem a necessidade de redeploy frequente de todo o monolito;
- **Novos requisitos** que demandam independência do stack de tecnologia do sistema legado e que precisam utilizar **tecnologias incompatíveis** com a arquitetura do sistema legado.

Todo o processo descrito nas seções deste documento refere-se apenas a primeira fase (**Transformar**), o planejamento e a execução das demais fases demanda uma análise do produto de software e deve ser realizado em outro projeto cujo escopo seja de migração.

Não podemos portanto classificar o serviço como microserviço até alcançar pelo menos a fase 4 (**Separar**), sendo que uma visão mais alinhada desse conceito obtemos apenas na última fase (**Reprojetar**).

CAPÍTULO 9

Mensageria

Neste capítulo abordamos como foram tratados os aspectos de comunicação entre os microsserviços no protótipo funcional.

9.1 Introdução

O objetivo da **mensageria** é permitir que cada microserviço trate, troque mensagens e colabore com outros microserviços para atender a requisição de um cliente.

Essa comunicação entre os microsserviços pode se encaixar em duas categorias:

- Síncrona
- Assíncrona

Essa comunicação pode ser gerenciada diretamente por cada microserviço ou por meio de um **broker/bus**, sendo que este último absorve a responsabilidade de gerenciar o recebimento e a entrega das mensagens e fornece recursos comuns que deveriam ser implementados por cada microserviço.

A **vantagem** da utilização de um intermediário é o uso de um conjunto avançado de recursos, personalizável, independente de linguagem, mantido e testado por uma comunidade ou organização com versões atualizadas frequentemente provendo maior segurança e robustez. A desvantagem é que temos mais um elemento centralizador na arquitetura em que uma falha no serviço de mensageria pode comprometer a disponibilidade de diversos serviços.

9.1.1 Padrões Arquiteturais

Os termos Message Broker e Message Bus são usados em padrões de arquitetura para sistemas de mensagens, também conhecidos como topologias de mensagens.

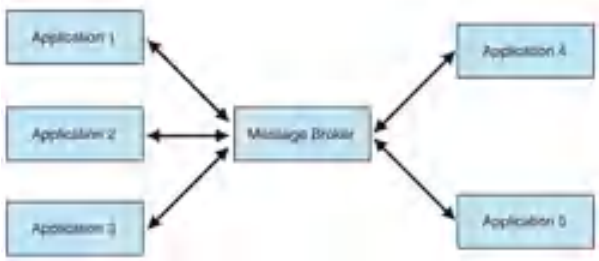
Embora um Message Bus seja uma dessas topologias, um Message Broker é um componente em uma topologia alternativa conhecida como Hub and Spoke.

Essas topologias descrevem maneiras diferentes de integrar aplicativos usando mensagens.

9.1.1.1 Message Broker

Na topologia Hub e Spoke (message broker), os aplicativos são integrados por meio de um componente de middleware de mensagem central. Os aplicativos enviam e recebem mensagens de e para o hub apenas. Os remetentes não têm conhecimento de outros aplicativos ou de suas preocupações específicas. O hub **intermedia ativamente** (broker) todas as comunicações entre os aplicativos, para fornecer integração adicional e serviços de mensagens. O hub é, portanto, comumente referido como Message Broker.

Figura 70 – Mensagem Broker



Fonte: https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ff648849(v=pandp.10)?redirectedfrom=MSDN

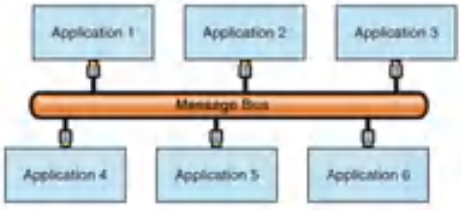
O hub visa desacoplar totalmente os aplicativos uns dos outros, de modo que um aplicativo de envio não tenha conhecimento do número ou destino / localização dos aplicativos de consumo.

9.1.1.2 Message Bus

É um padrão de arquitetura que oferece suporte à integração de aplicativos usando um conjunto compartilhado (comum) de interfaces. Consiste nos seguintes elementos-chave:

- Uma infraestrutura compartilhada (rede e canais de mensagens) para enviar mensagens aos destinatários;
- Um conjunto de esquemas de mensagens acordados (cabeçalhos de mensagens e um modelo de dados comum em termos de recursos e representações nas mensagens);
- Um conjunto de mensagens de comando comuns.

Figura 71 – Mensagem Bus



Fonte: https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ff647328(v=pandp.10)?redirectedfrom=MSDN

Quando um aplicativo deseja enviar uma mensagem, ele a passa para o barramento de mensagem, que é responsável por entregar a mensagem a todos os outros aplicativos que escutam mensagens na infraestrutura compartilhada do barramento.

São os aplicativos os responsáveis por implementar a interface de barramento comum. Um aplicativo que envia mensagens pelo barramento deve preparar a mensagem de forma que ela esteja de acordo com o tipo de mensagem que o restante dos aplicativos conectados ao barramento espera. Da mesma forma, um aplicativo que recebe mensagens deve ser capaz de compreender (sintaticamente, embora não necessariamente semanticamente) os tipos de mensagem.

Basicamente um message bus provê pouco ou nenhum serviço de centralização das mensagens. Mensagens enviadas podem ser que sejam apenas entregues no formato broadcast para todos os sistemas a ele conectados, apesar de que recursos como publish-subscribe podem ser suportados.

Uma arquitetura hub and spoke integra aplicativos sem impor uma API comum (esquema de mensagem) e oferece suporte para conectá-los usando protocolos distintos. Isso é possível por conta de adaptadores específicos (por exemplo, de protocolo) e um componente de middleware de mensagem central ativo. Este hub, ou message broker, entende as mensagens enviadas a ele, sabe para quais aplicativos a mensagem precisa ser enviada e pode transformar uma mensagem para atender ao esquema esperado por cada aplicativo.

9.1.2 Modelos de Comunicação

Além do padrão arquitetural do sistema de mensagens, outro aspecto está nas diversas formas em que a comunicação pode se estabelecer:

- **Notificações:** o emissário envia uma mensagem para um destino mas não espera nenhuma resposta;
- **Request/response:** o emissário envia uma mensagem para um destino e aguarda por uma resposta;
- **Request/asynchronous response:** o emissário envia uma mensagem para um destino e tem a expectativa de receber uma resposta, eventualmente;
- **Publish/subscribe:** o serviço publica uma mensagem para zero ou mais recipientes, ou seja, para quem se interessar;
- **Publish/asynchronous response:** o serviço publica uma requisição para um ou mais recipientes, em que alguns podem responder e outros não.

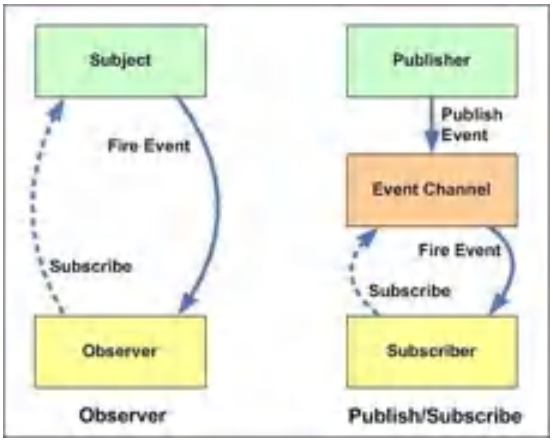
9.1.3 Modelos Tradicionais de Mensageria

Fila (Queuing): neste modelo um conjunto de consumidores pode ler do servidor e cada mensagem é entregue para um deles, ou seja, se a mensagem foi entregue para o consumidor A ela não será entregue para mais nenhum consumidor e o consumidor B deverá pegar outra mensagem na fila. A vantagem desse modelo é a facilidade de escalar o processamento, já que é possível dividir o processamento para múltiplas instâncias de consumidores.

Publish-subscribe: neste modelo a mensagem é enviada para todos os cosumidores interessados e por isso não há como escalar o processamento, visto que cada mensagem é entregue para todos os interessados.

Importante ressaltar que o padrão pub/sub não é como o padrão observer, essa relação pode ser mal compreendida. No padrão observer o sujeito (subject) conhece os interessados (observers), bem como os interessados também conhecem o sujeito, já no padrão pub/sub, publisher e subscribers não se conhecem e são auxiliados por um terceiro componente que é o broker.

Figura 72 – Padrão pub/sub, publisher e subscribers



Fonte: https://medium.com/@huytrongnguyen1985/from-pub-sub-pattern-to-observer-pattern-f4ae1e425cc9

Outra diferença reside no fato de que o padrão observer normalmente é implementado utilizando comunicação síncrona enquanto no padrão pub/sub a comunicação ocorre de forma assíncrona utilizando filas de mensagens.

9.2 Produtos

Entre os podutos mais conhecidos no mercado temos o **RabbitMQ** (message broker) e **Apache Kafka** (message bus), apesar do Redis poder ser utilizado como message broker seu foco de uso é o de caching, sendo este já utilizado na arquitetura do protótipo funcional para esse fim.

Kafka e RabbitMQ são ambos open-source e com suporte ao modelo publish-subscribe.

9.2.1 RabbitMQ

RabbitMQ é um broker message e tem mais tempo no mercado, lançado em 2007 e utilizado inicialmente em SOA. Atualmente também suporta streaming. Trata-se de um broker de **propósito geral**, de uso mais amplo e com suporte a um maior número de cenários de uso. Pode ser utilizado em cenários que demandam alto throughtput, que demandam processamento em segundo plano ou como message broker entre microsserviços.

Utiliza modelo **push** e permite controlar a carga dos consumidores por meio de um limite de pré-busca definido no consumidor. Isso pode ser usado para mensagens de baixa latência (não sobrecarregar o consumidor).

O objetivo do modelo push é distribuir mensagens de forma individual e rápida, para garantir que o trabalho seja paralelizado de maneira uniforme e que as mensagens sejam processadas aproximadamente na ordem em que chegaram na fila.

RabbitMQ suporta diversos protocolos que são padrões de mercado como AMQP, MQTT, STOMP, etc, o que permite seu uso em diversos cenários. O uso de protocolos padronizados também permite a troca de RabbitMQ por qualquer outro AMQP broker.

9.2.2 Kafka

Kafka é um message bus lançado em 2011 com foco em **streaming e data replay** (reprocessamento de dados). Trata-se de um bus que permite processar, persistir e reprocessar streamed data.

Utiliza modelo **pull**, os consumidores solicitam lotes de mensagens de um intervalo específico. Kafka utiliza conceito de partições para um melhor desempenho e fornece as mensagens em ordem.

Kafka utiliza um protocolo próprio sobre TCP/IP para comunicação entre aplicações e o cluster, ou seja, Kafka não pode simplesmente ser trocado já que é o único a implementar o protocolo.

Ambos suportam o consumo de milhões de mensagens por sergundo, mas como Kafka utiliza um modelo de partições com foco na leitura sequencial em disco, a necessidade de hardware é menor.

9.2.3 Tabela Comparativa

Tabela 25 – Comparativo entre RabbitMQ e Apache Kafka

	RabbitMQ	Apache Kafka
O que é?	É um *message broker* sólido, maduro e de propósito geral.	É um *message bus* otimizado para data stream e replay.
Uso principal	Message queue para comunicação e integração dentro e entre aplicações. Para tarefas de processamento longo ou processamento confiável em segundo plano.	Como framework para armazenamento, leitura (reteitura) e análise de fluxo de dados.
Licença	Open Source: Mozilla Public License	Open Source: Apache License 2.0
Escrito em:	Erlang	Scala (JVM)
Primeira versão	2007	2011
Persistência	Persiste as mensagens até que receba uma confirmação de reconhecimento.	Persiste as mensagens (recorde) com opção de exclusão após período ou período de retenção.
Replay	Não	Sim
Roteamento	Suporta roteamento flexível que pode retornar informações para um nó consumidor.	Não suporta roteamento flexível, precisa ser realizado por meio da separação por tópicos.
Prioridade nas mensagens	Suportado	Não suportado
Monitoramento	Disponível por meio de interface própria.	Disponível por meio de produtos de terceiros, tais como os disponíveis quando implantado em CloudKarafka ou Confluent.
Suporte de linguagens	Maioria das linguagens suportadas.	Maioria das linguagens suportadas.
Autenticação segura	Suporta autenticação padrão e OAuth2.	Suporta autenticação padrão, Oauth2 e Kerberos.

Fonte: https://www.cloudamqp.com/blog/2019-12-12-when-to-use-rabbitmq-or-apache-kafka.html#: :text=RabbitMQ%20can%20handle%20high%20throughput,ingress%20data%20streams%20and%20replay.

Tendo em vista que a arquitetura e o ambiente estabelecido para comunicação entre microsserviços não é destinado a um produto específico, mas sim uma arquitetura e ambiente flexível para surportar um processo de migração gradual de diversos produtos de software da SESP-MT, entende-se que o RabbitMQ é um produto mais adequado para essa finalidade, tendo em vista o suporte a diversos protocolos padronizados e suportar uma variedade maior de modelos de comunicação entre os microsserviços.

9.3 RabbitMQ

Um protocolo muito utilizado pelo broker é o Advanced Message Queue Protocol (AMQP), mais especificamente vamos abordar a versão 0-9-1, um dos protocolos implementados pelo RabbitMQ.

Um guia completo da referência pode ser encontrado neste link, contudo vamos apresentar uma **tradução adaptada** deste documento para o cenário do protótipo funcional.

9.3.1 Visão Geral do Modelo AMQP

9.3.1.1 Brokers e os Papéis envolvidos

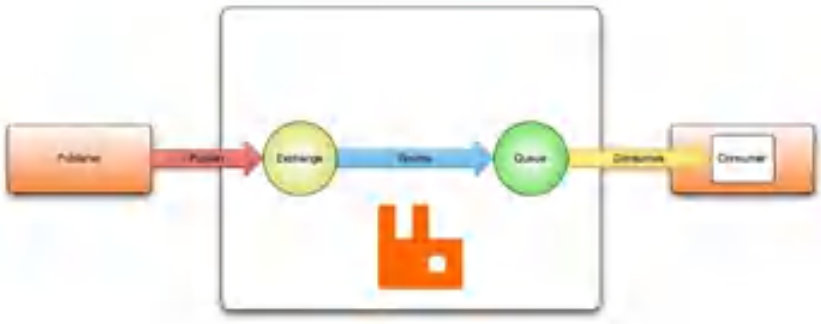
Um **broker** de mensageria recebe mensagens de **publishers**, também conhecidos como **producers** (produtores) e tem por responsabilidade rotear ou encaminhar essas mensagens para os **consumers** (consumidores).

Tendo em vista que se trata de um protocolo de rede, publishers, consumers e o broker podem estar cada qual em uma máquina distinta na rede.

9.3.1.2 Modelo AMQP 0-9-1 Resumido

Nesse modelo mensagens são enviadas para os **exchanges**, normalmente comparados ao serviço de correio. Os exchanges tem por responsabilidade distribuir cópias da mensagem para **queues** (filas) por meio de regras chamadas de **bindings**. O broker pode realizar a entrega de mensagens para consumidores inscritos nas filas ou os consumidores podem solicitar mensagens das filas sob demanda (fetch/pull).

Figura 73 – Modelo AMQP Resumido



Fonte: https://www.rabbitmq.com/tutorials/amqp-concepts.html

O produtor (publisher) pode adicionar diversos atributos acerca da mensagem que está sendo publicada, atuando como metadados da mensagem. Alguns desses metadados podem ser utiliza-

dos pelo broker, mas outros podem ser completamente ignorados, sendo utilizados apenas pelo consumidor.

Com relação a confiabilidade na entrega da mensagem, o modelo AMQP 0-9-1 utiliza o conceito de confirmação de mensagem (message acknowledgement): quando a mensagem é entregue ao consumidor, este notifica o broker automaticamente ou quando a aplicação consumidora decide realizá-la. Quando esse recurso está em uso, o broker somente remove a mensagem da fila quando a notificação é recebida.

Em algumas situações, por exemplo quando uma mensagem não pode ser roteada, esta pode retornar para o produtor, excluída, ou posta em uma "dead letter queue" (somente caso o broker implemente uma extensão). Produtores definem como tratar essas situações por meio da passagem de parâmetros ao publicar a mensagem.

Queues, exchanges e bindings, esses 3 elementos em conjunto são referenciados como **AMQP entities** (entidades AMQP).

9.3.1.3 AMQP 0-9-1 é um Protocolo Programável

AMQP 0-9-1 é um protocolo programável no sentido de que as entidades AMQP e a organização de roteamento são definidos pela própria aplicação e não por um administrador do broker.

Essa característica dá aos desenvolvedores uma grande liberdade e autonomia, mas requer atenção para não ocorrer em conflitos. As aplicações declaram as entidades AMQP necessárias, definem o esquema das rotas e podem excluir essas entidades quando não são mais necessárias.

9.3.1.4 Exchanges e Tipos de Exchange

As exchanges são responsáveis por receber uma mensagem e roteá-la para zero ou mais filas. O algoritmo de roteamento que será utilizado depende do tipo de exchange e das regras (bindings). Brokers AMQP 0-9-1 fornecem 4 tipos de exchange:

Tabela 26 – Tipos de Exchange

Tipo de Exchange	Nomenclatura padrão
Direct exchange	(String vazia) e amq.direct
Fanout exchange	amq.fanout
Topic exchange	amq.topic
Headers exchange	amq.match (e amq.headers no RabbitMQ)

Uma exchange, além do tipo, também possui diversos atributos, sendo os mais importantes:

- Name
- Durability (A exchange persiste mesmo após reinício do broker?)
- Auto-delete (A exchange deve ser excluída quando a última fila for desvinculada?)
- Arguments (opcional, utilizado por plugins ou recursos específicos do broker)

Direct Exchange

Entrega a mensagem para as filas baseado na chave de roteamento (routing key), sendo ideal para um roteamento do tipo **unicast** (apesar de poder ser utilizada para multicast também). Detalhamos abaixo como funciona:

- uma fila se liga a exchange com uma chave K (routing key);
- quando uma nova mensagem com a chave de roteamento R chega, a exchange encaminha a mensagem para a fila cuja condição K = R seja verdadeira.

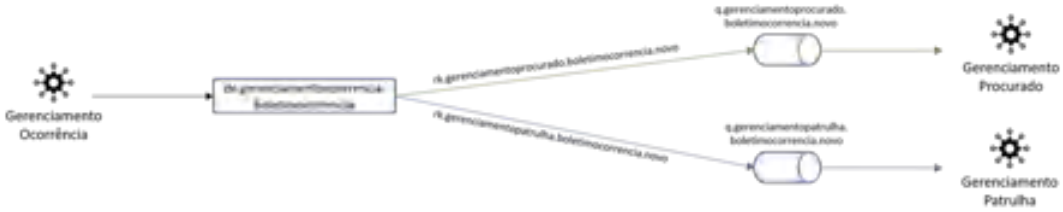
Este tipo de exchange normalmente é utilizada para distribuir tarefas entre múltiplos consumidores (instâncias da mesma aplicação) por meio de algoritmo do tipo round robin. É importante entender que há um balanceamento de carga entre as mensagens e os consumidores mas não com as filas.

Podemos representar conforme ilustração da figura 74:

Figura 74 – Exchange Registro Geral

Outro exemplo ilustrado na figura 75:

Figura 75 – Exchange Gerenciamento Ocorrência



Fanout Exchange

Entrega a mensagem para todas as filas vinculadas a ela e a chave de roteamento (routing key) é ignorada. Se N filas estão vinculadas a uma fanout exchange, quando uma nova mensagem é publicada na exchange uma cópia da mensagem é entregue para todas as N filas. Este tipo é ideal para mensagens do tipo broadcast.

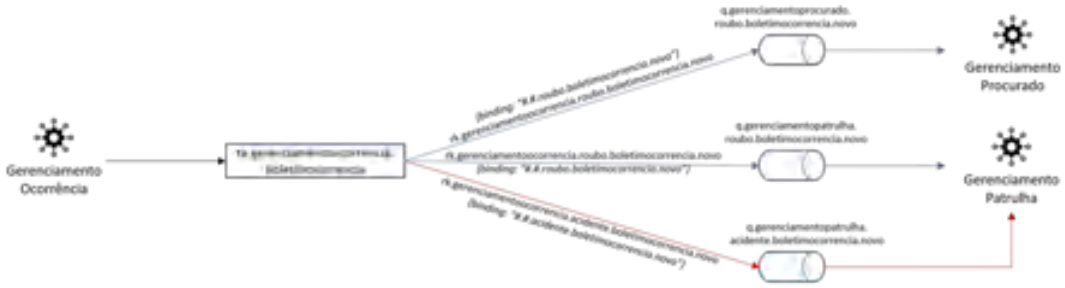
Podemos representar conforme ilustração da figura 76:

Topic Exchange

Entrega a mensagem para uma ou mais filas baseado na correspondência entre a chave de roteamento da mensagem e o padrão utilizado para vincular a fila a exchange. Normalmente utilizado para implementar o padrão publish/subscribe e para rotear mensagens de forma **multicast**.

Figura 76 – Fanout Exchange Gerenciamento Ocorrência

Figura 77 – Topic Exchange Gerenciamento Ocorrência



Seu uso deve ser aplicado quando um problema envolve múltiplos consumidores que podem de forma seletiva escolher que tipo de mensagem querem receber.

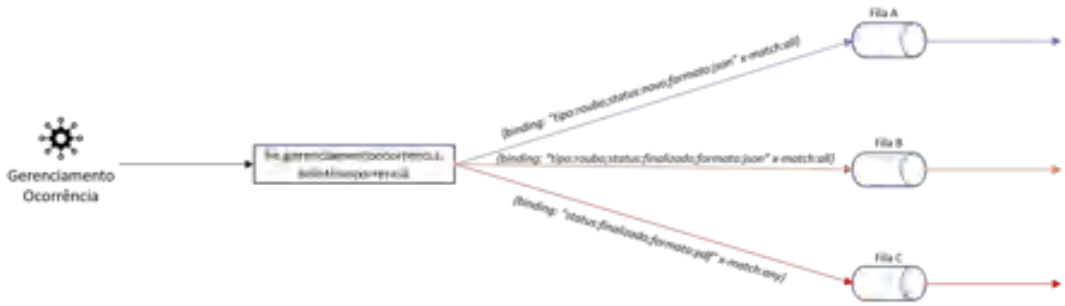
Podemos representar conforme ilustração da figura 77:

Header Exchange

Este tipo permite rotear as mensagens por meio de múltiplos atributos que são mais facilmente expressos na forma de cabeçalhos do que utilizando chaves de roteamento. Neste tipo de exchange a chave de roteamento é ignorada e os valores utilizados no roteamento são obtidos do header. Uma mensagem é considerada adequada para a fila se o valor do header for igual ao valor especificado quando a fila foi vinculada.

É possível vincular uma fila utilizando mais de um valor de header. Neste caso o broker precisa de mais uma informação, já que a mensagem deve ser entregue quando uma combinação ocorrer ou somente quando todas ocorrerem? Ao vincular a fila o argumento *x-match* tem essa finalidade, quando seu valor é *any*, apenas uma combinação é suficiente, contudo se seu valor for *all* então todos os valores devem combinar para que o roteamento ocorra. Este também pode ser utilizado para rotear mensagens de forma **multicast**. A figura 78 ilustra esse tipo de *exchange*.

Figura 78 – Header Exchange Gerenciamento Ocorrência



9.4 Implementação

9.4.1 Padrão de nomenclatura de recursos

Diferente dos padrões de nomenclatura adotados na linguagem Java, não há uma convenção geral aplicada aos recursos de mensageria. Por este motivo cada instituição pode adotar um padrão que melhor se adeque às necessidades do time. Sendo assim, o padrão adotado na implementação é uma proposta que visa facilitar a identificação dos recursos pela interpretação de seu nome.

Este padrão divide o nome dos recurso em **níveis**, separados por ponto, conforme exemplo abaixo:

tipo_recurso.publisher—consumer.entidade[.ação — tipo]*

Optamos por **4 níveis** para o protótipo funcional que se aplica de forma geral a qualquer cenário, mas esse estudo também deve levar em cosideração o domínio do problema e deve ser adequado em projetos novos ou de migração.

Nível 1: Identificação do **tipo do recurso**, podendo conter um dos seguintes valores:

Tabela 27 – Tipos de Recursos

Valor	Descrição
de	*Direct Exchange*
fe	*Fanout Exchange*
he	*Header Exchange*
te	*Topic Exchange*
q	*Queue*
rk	*Routing Key*

Apesar do nível 1 não ser necessário, tendo em vista que a associação da string ao tipo de objeto já o classifica, entendemos que essa informação presente na string é mais didática para fins de um modelo/protótipo, facilitando o entendimento.

Nível 2: Identificação do **serviço** responsável pelo recurso. Optamos pelo uso do publisher na composição do nome.

Nível 3: Identificação da **entidade** referente ao recurso.

Nível 4: Identificação da **ação** referente ao recurso ou uma separação por **tipo** das instâncias da entidade. Este nível será utilizado exclusivamente por recursos do tipo **Queue** e **Routing Key**.

Abaixo temos o exemplo de valor para uma routing key utilizada no protótipo funcional:

rk.gerenciamentoocorrencia.boletimocorrencia[.novo]*

Percebe-se que o recurso é uma routing key (rk), cujo microserviço publisher é o gerenciamentoocorrencia que irá publicar mensagens contendo como payload um boletimocorrencia sempre que um novo for criado.

9.4.2 Configuração da mensageria no projeto Spring Boot

Para o correto funcionamento da mensageria no projeto Spring Boot são necessários algumas configurações comuns.

9.4.2.1 Configuração da dependência

O Spring Boot conta com uma dependência para integração com o protocolo AMQP, que facilita e resolve grande parte das nuances da implementação. Esta pode ser incorporada ao projeto pela adição de sua identificação ao arquivo **pom.xml** conforme código a seguir.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-amqp</artifactId>
</dependency>
```

Código 9.1 – Configuração da Dependência

9.4.2.2 Conexão com message broker

É possível configurar a integração com o message broker utilizando as propriedades específicas no arquivo **application.properties**. No exemplo abaixo foram adicionadas as propriedades referentes ao host, porta, usuário e senha de conexão ao message broker.

```
spring.rabbitmq.host=rabbitmq
spring.rabbitmq.port=5672
spring.rabbitmq.username=sespmt
spring.rabbitmq.password=sespmt
```

Código 9.2 – Conexão com message broker

9.4.2.3 Conversor de conteúdo das mensagens

Por padrão, a troca de mensagens é realizada considerando o conteúdo das mensagens (payload) como texto simples, ou seja, contendo **text/plain** como valor do cabeçalho **content-type** das mensagens.

Nesta implementação optou-se pela utilização do payload destas mensagens em JSON. Essa estratégia possibilita a conversão automática do payload em um objeto que represente sua estrutura quando recebido pela aplicação.

Para permitir o tratamento do payload em JSON, no entanto, faz-se necessário configurar a injeção de dependência do **MessageConverter** como sendo uma instância de **Jackson2JsonMessageConverter**. Isso pode ser feito com a definição do **@Bean** conforme código a seguir.

```
@Bean
public MessageConverter messageConverter() {
    return new Jackson2JsonMessageConverter();
}
```

Código 9.3 – Conversor de conteúdo das mensagens

9.4.2.4 Template para envio de mensagens

O template é o recurso responsável pelo tratamento das mensagens de envio. Este recurso conhece as informações de conexão com o message broker e é capaz de utilizar o conversor para envio e recebimento de mensagens.

Esta configuração pode ser realizada pela criação do **@Bean** para injeção do template. Neste exemplo, o conversor é vinculado ao template no momento da criação da instância deste.

```
@Bean
public RabbitTemplate template(CachingConnectionFactory connectionFactory) {
    RabbitTemplate template = new RabbitTemplate(connectionFactory);
    template.setMessageConverter(messageConverter());

    return template;
}
```

Código 9.4 – Template para envio de mensagens

9.4.2.5 Configuração dos exchanges

Além das configurações padrões, a configuração do exchange exige mais 3 etapas:

1. Configuração da injeção do *exchange*;
2. Configuração das filas relacionadas ao *exchange*;
3. *Binding* (vínculo) das filas com o *exchange* configurado

A seguir, pode-se visualizar como seria a configuração de um *Direct Exchange*, da fila vinculada a ele e do bind entre esses dois recursos:

```
@Bean
public DirectExchange registroGeralDirectExchange() {
    return new DirectExchange("de.registrogeral.pessoa");
}

@Bean
public Queue registroGeralPessoaQueue() {
    return new Queue("q.registrogeral.pessoa");
}

@Bean
public Binding binding(DirectExchange registroGeralDirectExchange, Queue
    registroGeralQueue) {
    return BindingBuilder.bind(registroGeralQueue)
            .to(registroGeralDirectExchange)
            .with("rk.registrogeral.pessoa.buscarporemail");
}
```

Código 9.5 – Configuração dos exchanges

Configurações de outros tipos seguem a mesma lógica, com a diferença de que existe a possibilidade de se vincular mais de uma fila para o mesmo *exchange*, o que exigiria também a configuração de múltiplos *binds*. Este cenário é exemplificado na configuração de um *Topic Exchange e HeaderExchange* conforme código abaixo:

```
@Bean
public TopicExchange gerenciamentoOcorrenciaTopicExchange() {
    return new TopicExchange("te.gerenciamentoocorrencia.boletimocorrencia");
}

@Bean
public Queue procuradoQueue() {
    return new Queue("q.gerenciamentoprocurado.boletimocorrencia");
}

@Bean
public Queue patrulhaQueue() {
    return new Queue("q.gerenciamentopatrulha.boletimocorrencia");
}

@Bean
public Binding procuradoBinding(Queue procuradoQueue, TopicExchange
    gerenciamentoOcorrenciaTopicExchange) {
    return BindingBuilder.bind(procuradoQueue)
            .to(gerenciamentoOcorrenciaTopicExchange)
            .with("#.#.boletimocorrencia.novo");
}

@Bean
public Binding patrulhaBinding(Queue patrulhaQueue, TopicExchange
    gerenciamentoOcorrenciaTopicExchange) {
    return BindingBuilder.bind(patrulhaQueue)
            .to(gerenciamentoOcorrenciaTopicExchange)
            .with("#.#.boletimocorrencia.novo");
}

@Bean
public Binding procuradoStatusBoletimBinding(Queue procuradoBoCanceladoQueue,
    HeadersExchange gerenciamentoOcorrenciaHeadersExchange){

    Map<String, Object> propriedades = new HashMap<>();

    propriedades.put("x-match", "all");
    propriedades.put("tipo","boletimocorrencia");
    propriedades.put("situacao","cancelado");

    return BindingBuilder
            .bind(procuradoBoCanceladoQueue)
            .to(gerenciamentoOcorrenciaHeadersExchange)
            .whereAll(propriedades).match();
    }

@Bean
public Binding exampleBServiceBinding(Queue queueB, HeaderExchange
    gerenciamentoOcorrenciaHeaderExchange) {
    Map<String,Object> mapForQueue = new HashMap<>();
    mapForQueue.put("status", "finalizado");
    mapForQueue.put("formato", "pdf");

    return BindingBuilder.bind(queueB)
            .to(gerenciamentoOcorrenciaHeaderExchange)
            .whereAny(mapForQueue);
}
```

Código 9.6 – Topic Exchange e HeaderExchange

A Implementação de *FanoutExchange* segue a mesma estratégia de *TopicExchange* e pode ser implementada utilizando a entidade *FanoutExchange*.

9.4.2.6 Envio de mensagens

O envio de mensagens deve ser feito utilizando o template, por meio de injeção de dependência na classe onde ele será utilizado, geralmente em uma classe de serviço.

O código a seguir representa o envio de uma mensagem realizada no serviço Gerenciamento de Ocorrência. Outros atributos e métodos não relacionados com a implementação de mensageria foram omitidos para brevidade e foco na solução.

```
@Service
public class BoletimOcorrenciaService {

    private static final Logger LOGGER = Logger.getLogger(BoletimOcorrenciaService.
        class.getName());
    private static final String MENSAGEM_ENVIADA = "Mensagem enviada [%s]: %s";
    private static final String ROUTING_KEY_PESSOA = "rk.registrogeral.pessoa.
        buscarporemail";

    private RabbitTemplate template;

    private DirectExchange registroGeralDirectExchange;

    // Outros, metodos e injecoes de dependencia

    @Autowired
    public BoletimOcorrenciaService(RabbitTemplate template, DirectExchange
        registroGeralDirectExchange) {
        this.template = template;
        this.registroGeralDirectExchange = registroGeralDirectExchange;
    }

    private Long obterIdPessoa(String emailVitima) {
        PessoaDTO pessoaDTO = new PessoaDTO(emailVitima);

        pessoaDTO = template.convertSendAndReceiveAsType(registroGeralDirectExchange.
            getName(),
                ROUTING_KEY_PESSOA,
                pessoaDTO,
                new ParameterizedTypeReference<>() {});

        return pessoaDTO == null ? null : pessoaDTO.getId();
    }

    private void enviaMensagemBoletimCancelado(BoletimOcorrenciaDTO
        boletimOcorrenciaDTO) {

        MessageProperties propriedades = MessagePropertiesBuilder.newInstance()
                .setHeader("tipo", "boletimocorrencia")
                .setHeader("situacao", "cancelado")
                .setContentType(MessageProperties.CONTENT_TYPE_JSON)
                .build();

        MessageConverter messageConverter = new SimpleMessageConverter();
        Message mensagem = messageConverter.toMessage(boletimOcorrenciaDTO.toString(),
                propriedades);

```

```
43          template.convertSendAndReceiveAsType(gerenciamentoOcorrenciaHeadersExchange.
                getName(),
44                  "",
45                  mensagem,
46                  new ParameterizedTypeReference<>() {
47                  });
48          String statusBO = boletimOcorrenciaDTO.getSituacao();
49      }
50  }
```

Código 9.7 – Envio de mensagens

No exemplo acima, percebe-se a utilização do template, na linha **24** para a produção de uma mensagem. O método **convertSendAndReceive** foi utilizado para construir a chamada, enviar a mensagem, e aguardar um retorno. Neste caso, se a mensagem não for recebida dentro das tentativas esperadas, por qualquer motivo, a implementação prosseguirá e o valor **null** será atribuído à variável **pessoaDTO**. Diante desta característica, é essencial considerar e implementar os tratamentos dos possíveis valores de respostas após receber o retorno da mensagem.

9.4.2.7 Recebimento de mensagens

O recebimento de mensagens pode ser implementado com a utilização de um Listener. No exemplo abaixo foi utilizada a anotação **@RabbitListener** com o valor **queues**, que representa quais filas estarão vinculadas a este listener.

```
1 @RabbitListener(queues = "q.registrogeral.pessoa")
2 public PessoaDTO buscarPorEmail(PessoaDTO pessoaDTO) {
3   // Implementacao da logica necessaria para busca por email
4 }
```

Código 9.8 – Recebimento de mensagens

Com isso, uma vez que uma mensagem for recebida na fila **q.registrogeral.pessoa**, este listener irá "escutá-la" e recebê-la, já com os valores atribuídos ao parâmetro **pessoaDTO**.

9.4.3 Entidades AMQP do Protótipo Funcional

O modelo apresentado abaixo, presente na versão 0.7.0, foi atualizado para atender às necessidades das transações distribuídas. As alterações podem ser consultadas na sessão 4.4.

Abaixo apresentamos todas as entidades AMQP (*Exchanges, Queues, Bindings*) presentes no protótipo funcional, conforme figura 79.

A tabela 28 apresenta a finalidade de cada fila dentro do protótipo funcional:

9.4.4 Atualização das entidades AMQP do Protótipo Funcional

Para atender às necessidades das transações distribuídas foi necessário atualizar o fluxo dos eventos publicados e consumidos pelos serviços. Esta atualização teve por objetivo a melhoria da solução, tornando-a mais robusta e capaz de processar a compensação de tarefas quando necessário.

A seguir pode-se observar o fluxo de eventos atualizado para a rotina de cadastro de novo boletim de ocorrência.

Figura 79 – Entidades AMQP do Protótipo Funcional

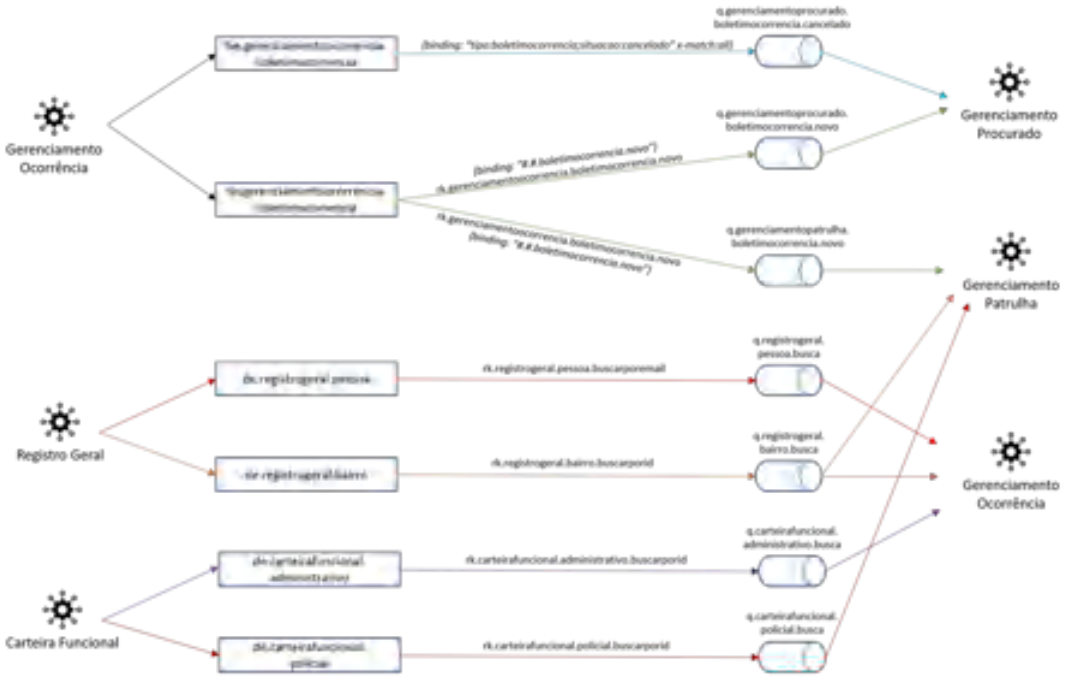
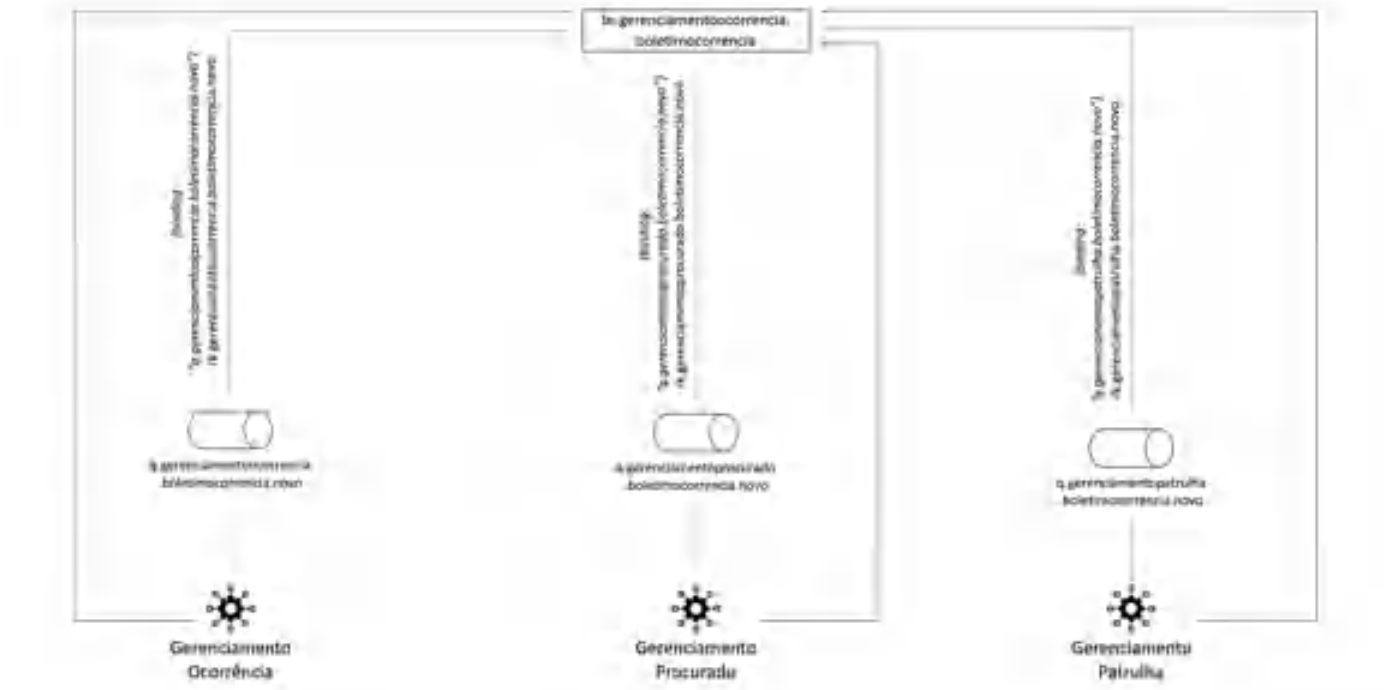


Tabela 28 – Finalidade das Filas

Fila	Propósito
q.gerenciamentoprocurado.boletimocorrencia.cancelado	Recebe boletins de ocorrência quando estes tiverem sua situação alterada para 'CANCELADO'. Mensagem consumida pelo microserviço Gerenciamento Procurado que remove os suspeitos pertencentes a esse boletim da lista de procurados caso estes não tenham outros BOs associados.
q.gerenciamentoprocurado.boletimocorrencia.novo	Recebe os novos boletins de ocorrência (recém criados). Mensagem consumida pelo microserviço Gerenciamento Procurado para a inclusão dos suspeitos na lista de procurados.
q.gerenciamentopatrulha.boletimocorrencia.novo	Recebe os novos boletins de ocorrência (recém criados). Mensagem consumida pelo microserviço Gerenciamento Patrulha para notificação das viaturas responsáveis pela região onde ocorreu o fato.
q.registrogeral.pessoa.busca	Fila que pode ser utilizada para consumo por diversos microsserviços que desejam realizar buscas de pessoas pelo e-mail. Utilizada quando o login social é realizado para buscar apenas boletins de ocorrência desta vítima.
q.registrogeral.bairro.busca	Fila que pode ser utilizada para consumo por diversos microsserviços que desejam realizar buscas de bairro pelo id. Consumida pelos pelo microsserviços Gerenciamento Ocorrência, para a exibição do nome do bairro onde ocorreu o fato relatado no boletim, e Gerenciamento Patrulha, para exibição do nome do bairro ao qual cada viatura está vinculada.
q.carteirafuncional.administrativo.busca	Fila que pode ser utilizada para consumo por diversos microsserviços que desejam realizar buscas de servidores administrativos pelo id. Consumida pelo microserviço Gerenciamento Ocorrência para exibição do nome do servidor responsável pelo cadastro do Boletim de Ocorrência.
q.carteirafuncional.policial.busca	Fila que pode ser utilizada para consumo por diversos microsserviços que desejam realizar buscas de policiais pelo id. Consumida pelo microserviço Gerenciamento Patrulha para a exibição dos nomes dos policiais em cada viatura.

Figura 80 – Entidades AMQP do Protótipo Funcional a partir da versão 0.8.0



As filas foram mantidas, mas as routing keys foram alteradas para possibilitar uma progressão linear do fluxo de cadastro de um novo boletim de ocorrência, ao contrário do modelo anterior, que fazia uso de um wildcard nas routing keys.

Esta comparação pode ser analisada em detalhes na tabela a seguir.

Tabela 29 – Fluxo anterior (v0.7.0) vs Fluxo atual

Fluxo anterior	**Fluxo atual**
Utilização de *wildcard* nas routing keys.	*Routing keys* específicas, sem a utilização de *wildcards*.
Serviços de Gerenciamento de Procurados e Gerenciamento de Patrulha consomem a mensagem publicada pelo serviço Gerenciamento Ocorrência **simultaneamente**, sendo esta a última etapa do fluxo.	Serviços consomem as mensagens em **ordem determinada**, sendo: 1. Gerenciamento de Ocorrência 2. Gerenciamento Procurado 3. Gerenciamento Patrulha 4. Gerenciamento de Ocorrência (novamente) O fluxo encerra com o serviço de **Gerenciamento de Ocorrência**, tanto em caso de sucesso quanto em caso de compensação de tarefas.
Somente **uma mensagem** é publicada, pelo serviço Gerenciamento de Ocorrência.	São necessárias **três mensagens** para a conclusão de fluxo.

CAPÍTULO 10

Log

Log permite o rastreio e a auditoria acerca dos eventos que estão ocorrendo no sistema e das ações dos usuários.

10.1 Introdução

Log está relacionado ao **registro de eventos** relevantes em um sistema computacional. Esse registro pode ser utilizado para restabelecer o estado original de um sistema ou para que um administrador conheça e possa analisar o comportamento histórico da aplicação. Um arquivo de log também pode ser utilizado para **auditoria** e **diagnóstico** de problemas em sistemas computacionais.

Ao organizar os dados de um log precisamos observar as seguintes características:

10.1.1 A Informação é Suficiente?

Seus logs devem fornecer informações suficientes para responder **quando**, **quem** e **onde** para qualquer evento. Por exemplo, digamos que você deseja ter logs de acesso. Se houve um evento de acesso para o seu sistema, a linha de log desse evento de acesso deve responder quando o acesso ocorreu. O log também deve incluir quem acessou o sistema. Frequentemente, esse detalhe vem na forma de um endereço IP ou outro identificador, e o "quem" deve incluir não apenas a ação humana, mas também as ações de outros sistemas. Sempre registre todos os acessos. Por fim, a logline deve incluir onde o sistema foi acessado, ou seja, qual serviço foi acessado ou qual parte da plataforma foi acessada.

10.1.2 A Informação está Suficientemente Detalhada?

Seus logs devem fornecer detalhes suficientes para responder o que exatamente aconteceu durante qualquer evento único. Simplesmente responder a quem, quando e onde não diz realmente o que aconteceu, e uma resposta básica como "este serviço reinicializou"não fornece detalhes suficientes para diferenciar se foi um código da aplicação que acionou a reinicialização ou se foi por uma falha do sistema.

10.1.3 Histórico dos Eventos

Finalmente, seus logs devem fornecer histórico suficiente para responder **por que** algo aconteceu. Os logs em geral ajudam a entender o **contexto da ação realizada**. Registrar logs em um terminal sem armazená-los é inútil. Uma única linha do log não fornece contexto suficiente para entender o ”por quê”de qualquer ação ou evento.

10.1.4 Níveis de Log

Os níveis de log referem-se à gravidade do impacto de uma mensagem em um sistema, ajudando as operações a entender como reagir. De forma geral temos pelo menos seis diferentes níveis para escolher, dependendo de qual padrão você está seguindo. Vamos dar uma olhada no padrão log4j:

TRACE - uma ação aconteceu - qualquer ação. Esse nível diminui significativamente o desempenho porque qualquer pequeno acontecimento é registrado.

DEBUG - Aconteceu uma ação que transferiu dados dentro de um componente (um microsserviço, uma API, um banco de dados, etc.). As mensagens de depuração são provavelmente muito familiares para a maioria dos desenvolvedores; nós as usamos para entender quando funções ou métodos foram chamados e como os dados podem ter se transformado em um único plano.

INFO - Aconteceu uma ação que transferiu dados de um componente para outro. Geralmente, este nível é o mais usado como uma mensagem geral.

AVISO (WARN) - Há algo errado, mas o problema não está impedindo o componente de continuar funcionando.

ERRO(R) - Algo deu errado com um componente específico do seu sistema, causando uma falha em parte das funções do componente, mas o restante do componente ainda está funcionando. Nesse caso, uma chamada de função pode ter falhado, mas o componente ainda está processando dados.

FATAL - O componente inteiro travou e não há esperança de se recuperar sem intervenção manual.

Os níveis de log indicam a importância de um determinado evento. **Ao definir o nível de log de seu interesse, você limita todos os níveis de log menos importantes a serem ignorados**. Portanto, se você definir o nível de log raiz como AVISO (WARN), só obterá eventos de log com os níveis AVISO, ERRO e FATAL:

Como recomendação geral, utilize níveis de log de forma consistente. Tenha um plano definido para todas as equipes. Você precisa chegar a um acordo de no mínimo sobre quais níveis usar para cada situação, pois os padrões de linguagem nem sempre são rígidos. Um **padrão corporativo/institucional** é ainda melhor para permitir o uso efetivo de sistemas de log pela equipe de operações, tendo em vista que podem lidar com vários ambientes de várias equipes diferentes de desenvolvimento.

10.1.5 Registre Apenas o Necessário

Por outro lado, não use apenas INFO para tudo. Os níveis permitem que você ajuste seus logs para mostrar apenas as informações de que você precisa e quando precisa delas. Você está em ambiente de produção e precisa saber apenas quando um sistema está falhando? Relate apenas ERROR ou

FATAL e certifique-se de que seu código sinalize cada erro com o nível de log apropriado. Está em um ambiente dev e deseja ver os dados fluindo por todo o sistema? Reporte para DEBUG ou mesmo TRACE e atualize seu código de acordo. Se você não usar níveis, seus sistemas serão menos eficientes, pois provavelmente você estará registrando muito e não terá as informações de que precisa na urgência para corrigir problemas quando eles surgirem.

Você não deve registrar tudo o que acontece em seu sistema o tempo todo. Registrar tudo o tempo todo leva à sobrecarga de informações, que sobrecarrega o sistema e os arquivos de log, e dificulta a extração de informação útil.

10.1.6 Geração de Log

Embora você obtenha logs gerais de suas plataformas, sistemas operacionais e servidores de aplicação, não obterá logs da sua aplicação, a menos que os adicione explicitamente em seu código fonte, e não obterá logs para necessidades adicionais, como auditoria, sem adicioná-los explicitamente. **Você precisa gerar logs para todas as suas diferentes necessidades em toda a profundidade de sua arquitetura**. Existem muitos tipos de log: logs de sistema, logs de aplicativo, logs de auditoria, etc. Se um nível de sua arquitetura não estiver registrando no mesmo sistema de gerenciamento de log que você está usando em todos os outros lugares, corrija isso imediatamente.

Não presuma que os logs gerados automaticamente são bons o suficiente. Você pode estar perdendo dados ou gerando e armazenando dados inúteis.

Cuidado com o texto da mensagem, erros de ortografia e procure utilizar uma linguagem ubíqua. Ao analisar mensagens e logs de pesquisa, uma única palavra com erro ortográfico ou de sentido duvidoso pode prejudicar toda informação, a rastreabilidade e entendimento do contexto. Reserve um tempo para revisar suas mensagens de log no código fonte antes de confirmá-las em uma branch compartilhada.

Não adicione “anexos” aos seus logs, qualquer tipo de artefato gerado que não seja uma mensagem de log ou um objeto de log estruturado não pertence ao seu fluxo de log. Por exemplo, um pdf, uma planilha. Esse tipo de artefato pode ocupar rapidamente grandes espaços em memória no ambiente de produção.

10.2 Tipos de Log

Existem duas metodologias gerais para log em arquiteturas modernas:

- log baseado em texto, e
- log estruturado.

10.2.1 O que são?

Os logs **baseados em texto** são uma série de mensagens baseadas em texto, uma para cada evento registrado. Geralmente, essas mensagens baseadas em texto são cadeias de caracteres e devem ser legíveis por pessoas. A mensagem de log é frequentemente construída com uma ou mais variáveis na string para permitir que dados específicos apareçam para o usuário examinar. Essa prática

pode fazer os logs parecerem bastante uniformes. No entanto, não são adequados para serem interpretados por algoritmos.

Os logs **estruturados**, por outro lado, são uma série de objetos enriquecidos com dados, um objeto para cada evento registrado. Você pode pensar neles como conjuntos de metadados sobre um evento específico, sempre incluindo data/hora e normalmente também incluindo informações sobre o sistema, plataforma, aplicativo e várias funções. Exemplos de formatos de log estruturados são JSON e XML; JSON é de longe o padrão de fato, simplesmente por ser mais econômico e muito difundido. Ao contrário dos logs baseados em texto, os logs estruturados são facilmente interpretados por algoritmos.

Apesar de logs estruturados serem destinados a máquinas, ainda vale a pena incluir um único campo legível por humanos com strings tokenizadas usando linguagem simples e direta. Essas mensagens podem ser exibidas no sistema de gerenciamento de log de sua escolha, garantindo que você ainda obtenha uma visão geral rápida **de uma linha** sem a necessidade de inspecionar o objeto.

10.2.2 Como eles são utilizados?

Em geral, os logs estruturados são preferidos aos logs baseados em texto quando você precisa executar funções automatizadas como alerta, monitoramento e análise ou grandes trabalhos de processamento de dados, como pesquisa. Logs estruturados são mais fáceis para uma máquina analisar e agir, e essa facilidade de análise leva a tempos de pesquisa mais rápidos e resultados mais limpos. Você também obterá maior consistência em diversas plataformas. Lembre-se: com logs estruturados as máquinas são o usuário principal, não um ser humano. Você não está usando os logs diretamente, apenas os resultados analisados por máquina.

10.3 Log em Microsserviços

Quanto ao log, há várias diferenças entre um monolito e uma arquitetura de microsserviços.

A primeira diferença é que em microsserviços temos mais componentes, o que acaba gerando mais logs já que cada microserviço gera seu próprio conjunto de logs.

A segunda diferença é que os logs de cada microserviço de forma isolada não são suficientes para entender o fluxo dos acontecimentos, já que microsserviços estão constantemente se comunicando com outros para atingir um objetivo. **É preciso entender como os microsserviços estão interagindo**.

Outro desafio é o escopo e tipos de logs gerados por cada microserviço que **tende a não ser uniforme**, já que a natureza independente e distribuída dessa arquitetura não é apenas de solução tecnológica, mas também de escopo e equipes.

Além de tudo isso microsserviços podem estar em ambientes distribuídos muito distintos e com diferentes restrições com o armazenamento do log nesses diferentes locais.

Felizmente encontramos boas práticas que ajudam a tratar de forma efetiva esses desafios de log em uma arquitetura de microsserviços:

10.3.1 Boas Práticas

Agregue e analise os dados de log em um **ponto centralizador** para obter uma visão holística da aplicação.

Utilize **identificadores personalizados** e únicos para contextualizar e mapear as interações entre os microsserviços.

Escreva **filtros personalizados** para o log central já que a estrutura pode variar de um microserviço para outro.

Persista os registros de log. Microsserviços costumam ser executados dentro de infraestruturas como as de contêiner - que carece de armazenamento persistente. Uma prática recomendada básica e essencial nesse caso é garantir que os dados de log sejam gravados em algum lugar onde serão armazenados de forma persistente e permanecerão disponíveis se o contêiner for encerrado.

Você pode fazer isso modificando o código-fonte ou as configurações do seu contêiner para garantir que os logs sejam gravados em um volume de armazenamento externo. Uma abordagem mais fácil, no entanto, é executar um agente de log que coletará dados do microsserviço em contêiner em tempo real e os agregará em um local de armazenamento confiável.

É interessante **manter todo o código de log no código fonte da aplicação em produção**. Dessa forma é possível criar uma estratégia para habilitar um maior ou menor detalhamento dos acontecimentos, sem a necessidade de reiniciar a aplicação.

Se um **ERROR** ocorrer, procure **coletar e registrar a maior quantidade de informação possível**. Esta ação pode demorar mas não é crítico tendo em vista que o processamento normal falhou, melhor do que esperar que a mesma condição ocorra com um depurador conectado.

Permita que a aplicação possa **alterar o nível de log de forma dinâmica**, sem a necessidade de reinicialização.

Lembre-se que logs normalmente são utilizados para monitoramento, identificação e resolução de problemas. **Se coloque no lugar daquele que precisa resolver o problema** e pense em quais dados e formatos gostaria de ter acesso quando um erro ocorrer fora do horário de expediente.

10.3.2 Dados Pertinentes para o Log

Quanto às informações pertinentes a serem registradas em log podemos elencar:

- Identificação do microserviço;
- Identificação do usuário;
- Endereço IP;
- Request ID;
- Correlation ID;
- Instante do tempo em que a mensagem foi recebiba (UTC);
- Nome do método/operação;
- Pilha de execução.

10.3.2.1 Request e Correlation IDs

O protocolo HTTP[1] permite o uso dos cabeçalhos **Request-ID** e **X-Correlation-ID**. A identificação da requisição permite identificar todos os eventos ocorridos por aquela requisição, já o identificador de correlação é muito importante em sistemas distribuídos já que permite rastrear o fluxo dos acontecimentos por tarefa (que pode envolver diversas requisições). Sem o uso desse tipo de informação poderíamos filtrar os logs por usuário e acompanhar o timestamp para identificar a ordem de execução, mas isso não seria suficiente para identificar quando começa e termina o log de uma determinada atividade disparada por esse usuário.

10.4 Níveis de Log do Protótipo Funcional

A depender do framework de log selecionado pode haver diversos níveis de log disponíveis para uso, mas quanto menos níveis de log utilizarmos, mais fácil será seguir algumas convenções para usá-los de forma consistente, reduzindo a chance de dubiedade quanto ao nível em que determinado evento deve ser registrado. Portanto, foram definidos o uso dos níveis abaixo para o protótipo funcional:

ERROR – WARN – INFO – DEBUG – TRACE

10.4.1 ERROR

O nível ERROR só deve ser usado quando o aplicativo realmente está com problemas, afetando os usuários e impedindo a realização das atividades.

Alguém deve ser alertado para corrigir imediatamente o problema, mesmo que seja no meio da noite. Deve haver algum tipo de alerta em vigor para eventos de log ERROR no ambiente de produção.

Normalmente, utiliza-se este nível quando um requisito de negócio muito importante não pode ser concluído devido a problemas técnicos ou um bug.

Atenção: cuidado para não usar este nível de forma abundante, tendo em vista que adicionaria muito ruído aos registros e reduziria a importância de um único evento ERROR.

10.4.2 WARN

O nível WARN deve ser utilizado quando algo não recomendado ou inesperado acontece, ou seja, há algo errado, mas o problema não está impedindo o componente de continuar funcionando.

Como os eventos ERROR, os eventos WARN devem ser atendidos por uma pessoa de desenvolvimento ou operação, portanto, deve haver algum tipo de alerta para o ambiente de produção.

Um exemplo concreto para uma mensagem WARN é quando um sistema falha ao se conectar a um recurso externo, mas tentará novamente automaticamente. Isso pode resultar em um log ERROR quando o mecanismo de nova tentativa também falhar. Outro exemplo é quando um serviço está demorando muito tempo para responder.

[1] https://en.wikipedia.org/wiki/List_of_HTTP_header_fields

O nível WARN é o nível que deve estar ativo nos sistemas de produção por padrão, para que apenas as mensagens WARN e ERROR sejam relatadas, economizando capacidade de armazenamento e desempenho.

Se o armazenamento e o desempenho não são um problema e o servidor de log fornece bons recursos de pesquisa, pode-se até relatar eventos INFO e DEBUG e apenas filtrá-los quando for de interesse.

10.4.3 INFO

O nível INFO deve ser usado para documentar as alterações de estado no microserviço ou de alguma entidade gerenciada pelo microserviço.

Essas informações podem ser úteis durante o desenvolvimento e às vezes até mesmo em produção para rastrear o que realmente está acontecendo no sistema.

Podemos citar como exemplos:

- o aplicativo foi iniciado com o parâmetro de configuração X com valor Y;
- um novo boletim de ocorrência foi criado;
- o boletim de ocorrência XYZ mudou a situação de PENDENTE para REGISTRADO;
- um trabalho em lote agendado regularmente terminou e processou X itens.

10.4.4 DEBUG

É mais difícil definir quais informações registrar neste nível. O objetivo é registrar qualquer informação que nos ajude a identificar o que deu errado a nível de depuração.

Podemos citar como exemplos:

- mensagens de erro de uma requisição HTTP malformada, resultando em um status HTTP 4xx;
- valor de variáveis da lógica de negócio.

Este nível pode ser usado de forma mais abundante do que os níveis acima, mas o código não deve ser repleto de instruções, pois reduz a legibilidade e polui o log.

10.4.5 TRACE

Comparado ao DEBUG, é muito fácil definir o que fazer no TRACE. Como o nome sugere, queremos registrar todas as informações que nos ajudam a rastrear o processamento de uma solicitação recebida pelo microserviço.

Isso inclui:

- início ou fim de um método, possivelmente incluindo o tempo de processamento;
- URL dos endpoints que foram chamados;
- início e fim do processamento de uma solicitação recebida ou trabalho agendado.

10.4.6 Padrão de Mensagem Texto

Como já mencionado, mesmo ao utilizar logs estruturados é importante adicionar um log de texto para interpretação humana:

```
“–TIME˙STAMP” –APPLICATION – SERVICE” –LOG˙LEVEL” –LOG˙MESSAGE””
```

A mensagem de log deve ser resumida e direta para tentar não ocupar mais de uma linha e facilitar a leitura.

Por exemplo:

```
2020-01-01T10:43:25Z /gerenciamento-ocorrencia INFO  boletim de ocorrencia 123 criado
2020-02-02T02:12:31Z /gerenciamento-roubo      ERROR servico de autenticacao nao
    responde
2020-03-03T08:37:12Z /gerenciamento-patrulha   INFO  viatura 123 foi notificada
2020-04-04T07:01:34Z /gerenciamento-ocorrencia DEBUG requisicao malformada GET /
    servico/...
2020-05-05T11:11:11Z /gerenciamento-procurado  INFO  suspeito X inserido na lista de
    procurados
```

Código 10.1 – Padrão de Mensagens

10.4.7 Nível de Log por Ambiente

Tabela 30 – Nível de Log por Ambiente

Ambiente	Nível do Log	
Desenvolvimento	TRACE/DEBUG	Nível utilizado apenas no ambiente dev e portanto descartáveis. O objetivo é auxiliar o desenvolvedor na depuração do serviço.
Homologação	INFO	O ambiente de homologação tem por objetivo auxiliar os usuários do sistema que estão testando/validando os serviços que serão disponibilizados. Tendo em vista o escopo reduzido de usuários, tempo de uso, de dados e até mesmo de desempenho, o nível INFO se mostra suficiente para validação da aplicação.
Produção	INFO/WARN	Se deseja fazer um registro de auditoria, por exemplo, faz-se necessário ativar o nível INFO em produção. Caso o log seja utilizado apenas para identificação de problemas e/ou há escassez quanto aos recursos computacionais, recomenda-se apenas a utilização do nível WARN.

10.5 Soluções Tecnológicas

A seguir apresentamos algumas soluções tecnológicas pesquisadas que podem ser utilizadas e atender diferentes objetivos de log.

É importante mencionar que foi realizado um teste com o **Spring Boot Admin** como possível estratégia descentralizada para acompanhamento dos eventos de log, mas se mostrou inviável tendo em vista que a ferramenta não possibilita uma visão holística dos acontecimentos, apenas uma visão isolada por microserviço, não permitindo o acompanhamento do fluxo de comunicação.

Optou-se pelo uso do **Log4j2**, dentre possíveis alternativas como Log4j e Logback que também são amplamente utilizados. A escolha ocorreu por conta do Log4J2 ser mais recente, considerado mais performático e mais avançado das três opções. Possui suporte a SLF4J, reload automático das configurações de log, suporte a opções de filtros avançados, avaliação de logs por meio de expressões lambda, possibilita loggers assíncronos para sistemas de baixa latência e fornece um modo garbage-free para evitar latências causadas pelas operações do garbage collector.

10.5.1 Utilizando Broker de Mensageria

Uma alternativa para consolidação e centralização de mensagens de log é o uso do próprio broker de mensageria (RabbitMQ), criando uma fila própria para recebimento das mensagens de log.

Esse tipo de solução pode ser interessante para realizar ações personalizadas, implementando consumidores que, por exemplo, enviem um SMS ou e-mail quando do recebimento de logs do tipo ERROR.

É uma solução que também permite de forma simples centralizar as mensagens de log em sistemas distribuídos, mas não fornece nenhum recurso adicional para armazenamento, filtro e análise, ficando a cargo de implementação própria e/ou uso de softwares complementares. Deve-se também levar em consideração a sobrecarga que esse tipo de solução pode acarretar ao broker, exigindo um redimensionamento dos recursos computacionais para a mensageria.

Um exemplo prático desse tipo de solução pode ser encontrada no artigo *Creating a Logging Application in RabbitMQ With Spring Boot and Logback*[2].

A implementação teste foi realizada no microserviço Gerenciamento Ocorrência. O teste foi realizado com sucesso e as modificações no projeto envolveram:

10.5.2 Configuração no Log4j2

Foi criado o arquivo **log4j2.xml** em **src/main/resources** a adição do appender abaixo:

```
<?xml version="1.0" encoding="UTF-8"?>
...
    <Appenders>
        ...
        <RabbitMQ name="rabbitmq-appender"
                  addresses="rabbitmq:5672"
                  user="sespmt"
                  password="sespmt"
                  virtualHost="/"
                  exchange="de.log"
                  applicationId="gerenciamento-ocorrencia"
                  routingKeyPattern="rk.log"
                  contentType="text/plain"
                  contentEncoding="UTF-8"
                  generateId="false"
```

[2] https://dzone.com/articles/logging-application-log-in-rabbitmq-using-springbo

```
                    deliveryMode="PERSISTENT"
                    charset="UTF-8"
                    senderPoolSize="5"
                    maxSenderRetries="10">
            </RabbitMQ>
            ...
        </Appenders>
        ...
        <Logger name="br.gov.mt.sesp" level="trace" additivity="false">
            ...
            <AppenderRef ref="rabbitmq-appender" />
            ....
        </Logger>
...
```

Código 10.2 – Configuração no Log4j2

E na classe **MessagingConfiguration.java** foram adicionadas as entidades AMQP para log:

```
...
@Bean
DirectExchange logDirectExchange() {
   return new DirectExchange("de.log");
}
...
@Bean
public Queue logQueue(){
   return new Queue("q.log");
}
...
@Bean
public Binding logRabbitBinding(Queue logQueue,
                                DirectExchange logDirectExchange){
    return BindingBuilder.bind(logQueue)
            .to(logDirectExchange)
            .with("rk.log");
}
```

Código 10.3 – Entidades AMQP

A partir deste momento todas as mensagens de log terão sua saída na forma de mensagens entregues ao exchange **de.log** que as encaminhará para a fila **q.log**.

10.5.2.1 Logstash

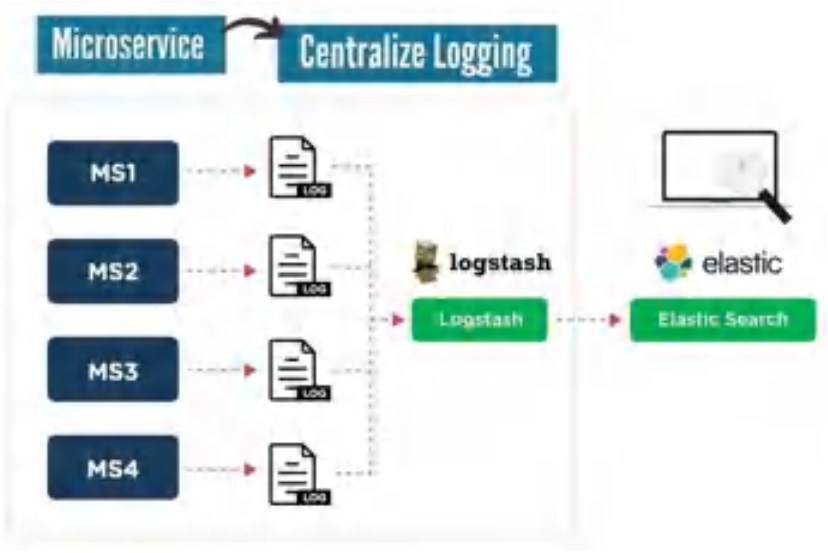
O Logstash é um dos produtos principais do Elastic Stack, é usado para agregar e processar dados e enviá-los ao Elasticsearch. O Logstash é um *pipeline* de processamento de dados do lado do servidor que permite fazer a **ingestão de dados** de várias fontes simultaneamente e enriquecer e transformar esses dados antes de serem indexados no Elasticsearch.

Essa capacidade de "ingerir" dados de diversas fontes permite consolidar e centralizar os diversos logs distribuídos dos microsserviços.

10.5.2.2 Elasticsearch

Já o Elasticsearch é responsável pela **indexação de dados**. No Elasticsearch os usuários podem executar consultas complexas com base em seus dados e usar agregações para recuperar resumos complexos dos dados.

Figura 81 – Elasticsearch + Logstash



É uma ferramenta rápida, construída sobre o Lucene e que permite fazer escalabilidade horizontal para centenas de servidores, podendo manipular um volume de dados na ordem dos petabytes.

Possui integração natural com Logstash sendo um stack bastante popular para solução de logs em sistemas distribuídos como os de microsserviços.

10.5.3 Configuração

10.5.3.1 Dependências

No arquivo pom.xml do microserviço realizamos a exclusão do log padrão do spring boot e adicionamos dependências para o **Log4J2** e para o **Logstash**, este último responsável pela centralização dos dados de log.

```
...
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
  <exclusions>
    <exclusion>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-logging</artifactId>
    </exclusion>
  </exclusions>
</dependency>
...
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-log4j2</artifactId>
</dependency>

<dependency>
  <groupId>biz.paluch.logging</groupId>
  <artifactId>logstash-gelf</artifactId>
  <version>1.13.0</version>
</dependency>
...
```

Código 10.4 – Configuração do Log4J2 no pom.xml

10.5.3.2 Log4j2

No arquivo de configuração **log4j2.xml**, realizamos a configuração da **localização do logstash** (host e porta) e o appender indicando a **saída padrão**, sendo que temos dois appenders configurados:

- **log texto**: para o console em um formato de mensagem contendo o nível do log, o timestamp, a thread, a classe, o método e a mensagem;
- **log estruturado**: contendo os seguintes dados: microserviço, nome qualificado da classe, ID de correlação, IP, nível do log, método, host, nome simples da classe, timestamp e usuário.

Atenção: Quanto maior o número de dados que irão compor a mensagem de log maior é a carga de processamento necessária, que pode causar impacto relevante a depender da escala de uso. Alguns desses dados são mais custosos computacionalmente que outros, como é o caso do nome do método, tendo em vista a necessidade de um processo de reflexão sobre o objeto para a extração desse dado.

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration monitorInterval="60">
    <Properties>
        <Property name="logstash.host">udp:graylog</Property>
        <Property name="logstash.port">12201</Property>
    </Properties>
    <Appenders>
        <Console name="Console-Appender" target="SYSTEM_OUT">
            <PatternLayout>
                <pattern>
                    [%-5level] %d{yyyy-MM-dd HH:mm:ss.SSS} [%t] %c{1} %M - %msg%n
                </pattern>
            </PatternLayout>
        </Console>
        <Gelf name="Gelf"
              host="${logstash.host}"
              port="${logstash.port}"
              version="1.1"
              extractStackTrace="true"
              filterStackTrace="false"
              originHost="%host{fqdn}">
            <Field name="application" pattern="registro-geral"/>
            <Field name="className" pattern="%C"/>
            <Field name="correlationId" pattern="%X{LogFilter.correlationId}"/>
            <Field name="ip" pattern="%X{LogFilter.ip}"/>
            <Field name="level" pattern="%level"/>
            <Field name="method" pattern="%M"/>
            <Field name="server" pattern="%host"/>
            <Field name="simpleClassName" pattern="%C{1}"/>
            <Field name="timestamp" pattern="%d{dd MMM yyyy HH:mm:ss,SSS}"/>
            <Field name="username" pattern="%X{LogFilter.username}"/>
        </Gelf>
    </Appenders>
    <Loggers>
        <Logger name="org.springframework.web" level="info" additivity="false">
```

```
            <AppenderRef ref="SpringBoot-Appender"/>
            <AppenderRef ref="Console-Appender"/>
        </Logger>
        <Logger name="br.gov.mt.sesp" level="info" additivity="false">
            <AppenderRef ref="App-Appender"/>
            <AppenderRef ref="Console-Appender"/>
            <AppenderRef ref="Gelf"/>
        </Logger>
        <Root level="info">
            <AppenderRef ref="Console-Appender"/>
            <AppenderRef ref="Gelf"/>
        </Root>
    </Loggers>
</Configuration>
```

Código 10.5 – Arquivo Log4J2.xml

10.5.3.3 LogFilter.java e LogConfiguration.java

Tendo em vista a quantidade de dados a serem registrados em log que podem atravessar diversas camadas do sistema é que optamos por compartilhar esses dados por meio da técnica MDC (*Mapped Diagnostic Context*) suportada pelo Log4J2.

O MDC gerencia informações contextuais por *thread*, portanto, uma vez que você tenha definido a chave em uma *thread*, todas as solicitações de log realizadas a partir dessa *thread* usarão esse contexto e isso facilita o compartilhamento de dados.

Implementamos portanto um filtro para criação desse contexto e compartilhamento dos dados de **IP**, **ID** de **correlação** e **usuário**:

```
@Component
public class LogFilter extends OncePerRequestFilter {

    ...

    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse
        response, FilterChain filterChain) throws ServletException, IOException {
        try {
            final String valorCorrelationId = obterCorrelationId(request);

            MDC.put(correlationId, valorCorrelationId);
            MDC.put(ip, obterIp(request));
            MDC.put(nomeUsuario, obterNomeUsuario(request));

            if (!StringUtils.isEmpty(responseHeader)) {
                response.addHeader(responseHeader, valorCorrelationId);
            }

            filterChain.doFilter(request, response);
        } finally {
            MDC.remove(correlationId);
            MDC.remove(ip);
        }
    }

    private String obterCorrelationId(final HttpServletRequest request) {
        final String correlationId;

```

```
29          if (!StringUtils.isEmpty(requestHeader) &&
30                  !StringUtils.isEmpty(request.getHeader(requestHeader))) {
31              correlationId = request.getHeader(requestHeader);
32          } else {
33              correlationId = UUID.randomUUID().toString().toUpperCase().replace("-", ""
                    );
34          }
35
36          return correlationId;
37      }
38
39      private String obterIp(final HttpServletRequest request) {
40          final String ip;
41
42          if (request.getHeader("X-Forwarded-For") != null) {
43              ip = request.getHeader("X-Forwarded-For").split(",")[0];
44          } else {
45              ip = request.getRemoteAddr();
46          }
47
48          return ip;
49      }
50
51      private String obterNomeUsuario(final HttpServletRequest request) {
52          String authorizationToken = request.getHeader(AUTHORIZATION_HEADER);
53
54          if (StringUtils.isBlank(authorizationToken)) {
55              return null;
56          }
57
58          authorizationToken = authorizationToken.startsWith("Bearer ")
59                  ? authorizationToken.substring(7)
60                  : authorizationToken;
61
62          String encodedPayload = authorizationToken.split("\\.")[1];
63          JSONObject payload = new JSONObject(new String(BASE64.decode(encodedPayload)))
                ;
64
65          return payload.getString(TOKEN_USERNAME);
66      }
67  }
```

Código 10.6 – Filtro LogFilter

E ativamos o filtro por meio da classe **LogConfiguration.java**:

```
1   ...
2   @Configuration
3   public class LogConfiguration {
4
5       public static final String CABECALHO_CORRELATION_ID = "X-Correlation-ID";
6       public static final String PROPRIEDADE_CORRELATION_ID = "LogFilter.correlationId";
7       public static final String PROPRIEDADE_IP = "LogFilter.ip";
8       public static final String PROPRIEDADE_NOME_USUARIO = "LogFilter.username";
9
10      private String requestHeader = null;
11      private String responseHeader = CABECALHO_CORRELATION_ID;
12      private String correlationId = PROPRIEDADE_CORRELATION_ID;
13      private String ip = PROPRIEDADE_IP;
14      private String nomeUsuario = PROPRIEDADE_NOME_USUARIO;
15
16      @Bean
```

```
17      public FilterRegistrationBean servletRegistrationBean() {
18          final FilterRegistrationBean registrationBean = new FilterRegistrationBean();
19
20          final LogFilter logFilter = new LogFilter(requestHeader, responseHeader,
                correlationId, ip, nomeUsuario);
21          registrationBean.setFilter(logFilter);
22          registrationBean.setOrder(2);
23
24          return registrationBean;
25      }
26  }
```

Código 10.7 – Ativação do Filtro LogFilter

Na classe de aplicação do microserviço adicionamos um **BeanPostProcessor** para que instâncias de **RabbitTemplate** enviem de forma automática o ID de correlação (**X-Correlation-Id**) no cabeçalho das mensagens:

```
1  public class GerenciamentoOcorrenciaApplication {
2      ...
3      @Bean
4      public static BeanPostProcessor bpp() {
5          return new BeanPostProcessor() {
6
7              @Override
8              public Object postProcessAfterInitialization(Object bean, String beanName)
                   throws BeansException {
9                  if (bean instanceof RabbitTemplate) {
10                     ((RabbitTemplate) bean).setBeforePublishPostProcessors(m -> {
11                         m.getMessageProperties().getHeaders().put("X-Correlation-Id",
                               MDC.get(LogConfiguration.PROPRIEDADE_CORRELATION_ID));
12                         return m;
13                     });
14                 }
15                 return bean;
16             }
17         };
18     }
19
20 }
```

Código 10.8 – BeanPostProcessor

Dessa forma consumidores podem extrair esse dado por meio da anotação **@Header**:

```
1  @Service
2  public class ProcuradoService {
3      ...
4      @RabbitListener(queues = "q.gerenciamentoprocurado.boletimocorrencia.novo")
5      public void atualizarProcurados(BoletimOcorrenciaDTO boletimOcorrenciaDTO, @Header
           ("X-Correlation-Id") String correlationId) {
6          MDC.put(LogConfiguration.PROPRIEDADE_CORRELATION_ID, correlationId);
7          ...
8      }
9  }
```

Código 10.9 – Extração de Dados com @Header

CAPÍTULO 11

Monitoramento

Neste capítulo apresentamos meios práticos para o acompanhamento dos eventos que ocorrem no ambiente de microsserviços de forma assistida e automatizada.

11.1 Introdução

Definimos monitoramento como um processo de captura do comportamento de um sistema por meio de verificações de integridade e métricas ao longo do tempo. Isso ajuda a detectar anomalias: quando o sistema está indisponível, apresenta uma carga incomum, esgotamento de certos recursos ou de outra forma não se comporta dentro de seus parâmetros normais (esperados). O monitoramento envolve a coleta e o armazenamento de métricas de longo prazo, o que é importante, mais do que a detecção de anomalias, mas também a análise da causa desde sua raiz, detecção de tendências e planejamento de capacidade.

Estamos acostumados a monitorar um sistema por vez, quando precisamos olhar um sistema diferente mudamos para outro monitor ou ferramenta. No entanto, microsserviços estão mais para diversos instrumentos em uma simfonia, observar um único instrumento não nos permite capturar toda composição musical, precisamos observar o conjunto em operação.

Isso não quer dizer que não é importante poder monitorar um aspecto ou serviço da sua aplicação, mas que não é suficiente.

Neste projeto o foco está no monitoramento dos serviços, do broker de mensageria e o histórico do fluxo de eventos ocorridos (log). Nao está no escopo o monitoramento de recursos de um bare metal, máquinas virtuais ou contêineres, esse monitoramento fica a cargo das ferramentas utilizadas pela equipe de operação.

Vislumbramos 3 elementos para monitoramento do ambiente do protótipo funcional:

- Microsserviço: observar cada microserviço e seu estado de execução;
- Broker de Mensageria: observar as entidades AMQP e a própria Erlang VM;

- Log: observar o comportamento histórico da simfonia como um todo e utilizar esses dados para identificação de problemas ou possíveis eventos que poderm estar compromentendo ou vir a comprometer o uso do sistema.

11.2 Microsserviço

O monitoramento de microsserviços é uma tarefa necessária a fim de garantir que os sistemas se comportem da maneira prevista. Mais do que apenas verificar seu status no momento, isto é, se está ativo ou não, métricas como uso de memória, armazenamento, taxas de Entrada/Saída, Logs e requisições contribuem na identificação de problemas de performance que podem impactar a aplicação e causar sua falha. O cenário de microsserviços combinados com o deployment conteinerizado resulta em sistemas com vários dados percorrendo diversas camadas e tais dados, com o monitoramento correto, podem ser úteis para investigação e resolução possíveis problemas.

11.2.1 Actuators

O Spring Boot Actuator é um sub-projeto do Spring Boot que inclui funcionalidades que ajudam a monitorar e gerenciar aplicações Spring. Com ele é possível obter métricas, dados de saúde da aplicação e auditoria de maneira simples, seja através de requisições HTTP ou com auxílio do JMX.

Os três principais recursos do Spring Boot Actuator são:

1. **Endpoints**: através deles é possível acessar dados e interagir com a aplicação. Diversos endpoints já são incluídos por padrão, sendo possível habilitá-los ou desabilitá-los individualmente, e são precedidos por /actuator. O Spring Boot Actuator também permite criar novos endpoints de acordo com a necessidade de cada cenário.

2. **Métricas**: Graças à integração do Spring Boot ao micrometer, uma ferramenta para fornecimento de métricas de aplicações, é possível ter acesso a dados dimensionais dos serviços.

3. **Auditoria**: O Spring Boot Actuator possui um framework flexível para auditoria de eventos caso o Spring Security esteja sendo utilizado, publicando eventos de sucesso de autenticação, falha ou acesso negado, por exemplo. Traz as informações no endpoint /actuator/auditevents, sendo necessário para isso um bean de **AuditEventRepository**.

Para habilitar os actuators em um microserviço é necessário primeiramente adicionar a dependência ao Maven, conforme segue:

```
...
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
...
```

Código 11.1 – Dependências no pom.xml

Dessa forma, já é possível ter acesso aos endpoints dos actuators. Por padrão, apenas /info e /health são ativos, assim, para ativar os restantes é necessário adicionar ao arquivo application.properties a configuração:

```
management.endpoints.web.exposure.include = *
```

Código 11.2 – Arquivo de Propriedades

É recomendado que também se adicione ao arquivo de propriedades informações de porta e endereço do host do actuator:

```
management.server.port: 10002
management.server.address: gerenciamento-ocorrencia  #nome do conteiner docker
```

Código 11.3 – Arquivo de Propriedades

Agora, para ter acesso aos dados, basta acessar o endpoint **/actuator** da aplicação. Abaixo é exibido o retorno de uma requisição GET ao endpoint do actuator no serviço Autenticação.


```
{
  "_links": {
    "self": {
      "href": "http://localhost:8000/v0-7/autenticacao/actuator",
      "templated": false
    },
    "beans": {
      "href": "http://localhost:8000/v0-7/autenticacao/actuator/beans",
      "templated": false
    },
    "caches-cache": {
      "href": "http://localhost:8000/v0-7/autenticacao/actuator/caches/{cache}",
      "templated": true
    },
    "caches": {
      "href": "http://localhost:8000/v0-7/autenticacao/actuator/caches",
      "templated": false
    },
    "health-path": {
      "href": "http://localhost:8000/v0-7/autenticacao/actuator/health/{*path}",
      "templated": true
    },
    "health": {
      "href": "http://localhost:8000/v0-7/autenticacao/actuator/health",
      "templated": false
    },
    "info": {
      "href": "http://localhost:8000/v0-7/autenticacao/actuator/info",
      "templated": false
    },
    "conditions": {
      "href": "http://localhost:8000/v0-7/autenticacao/actuator/conditions",
      "templated": false
    },
    "configprops": {
      "href": "http://localhost:8000/v0-7/autenticacao/actuator/configprops",
      "templated": false
    },
    "env": {
      "href": "http://localhost:8000/v0-7/autenticacao/actuator/env",
      "templated": false
    },
```

```
    "env-toMatch": {
      "href": "http://localhost:8000/v0-7/autenticacao/actuator/env/{toMatch}",
      "templated": true
    },
    "loggers": {
      "href": "http://localhost:8000/v0-7/autenticacao/actuator/loggers",
      "templated": false
    },
    "loggers-name": {
      "href": "http://localhost:8000/v0-7/autenticacao/actuator/loggers/{name}",
      "templated": true
    },
    "heapdump": {
      "href": "http://localhost:8000/v0-7/autenticacao/actuator/heapdump",
      "templated": false
    },
    "threaddump": {
      "href": "http://localhost:8000/v0-7/autenticacao/actuator/threaddump",
      "templated": false
    },
    "metrics": {
      "href": "http://localhost:8000/v0-7/autenticacao/actuator/metrics",
      "templated": false
    },
    "metrics-requiredMetricName": {
      "href": "http://localhost:8000/v0-7/autenticacao/actuator/metrics/{
          requiredMetricName}",
      "templated": true
    },
    "scheduledtasks": {
      "href": "http://localhost:8000/v0-7/autenticacao/actuator/scheduledtasks",
      "templated": false
    },
    "mappings": {
      "href": "http://localhost:8000/v0-7/autenticacao/actuator/mappings",
      "templated": false
    }
  }
}
```


Código 11.4 – Endpoint **/actuator**

Cada item retornado apresenta informações específicas da aplicação, segue tabela descrevendo cada recurso acessível pelo endpoint:

Para mais informações acerca do Spring Boot Actuator, recomenda-se acessar:

- Spring Boot Actuator: Production-ready Features[1]
- Getting Started — Building a RESTful Web Service with Spring Boot Actuator[2]
- Spring Boot Actuator Web API Documentation[3]

[1] https://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-features.html
[2] https://spring.io/guides/gs/actuator-service/
[3] https://docs.spring.io/spring-boot/docs/current/actuator-api/htmlsingle/

Tabela 31 – Recursos Acessíveis pelos endpoints

ID	Informação
auditevents	Dados de auditoria
beans	Beans do Spring no aplicação
caches	Caches disponíveis.
health	Informações gerais das condições de aplicação.
info	Dados arbitrários, como nome, descrição, versão do Java.
conditions	Verifica se as condições definidas nas classes de configuração foram ou não atingidas.
configprops	Exibe uma lista de @ConfigurationProperties
env	Expõe propriedades de ConfigurableEnvironment
loggers	Exibe as configurações de Log de aplicação
heapdump	Exibe as informações dos objetos em memória na JVM
threaddump	Exibe os estados de todas as threads em execução
metrics	Exibe as métricas de aplicação
scheduledtasks	Retornar as tarefas agendadas.
mappings	Retorna uma lista de todos os endpoints @RequestMapping

11.2.2 Ferramentas de Monitoramento

11.2.2.1 Prometheus e Grafana

Uma solução de monitoramento de serviços bastante adotada no mercado é resultado da combinação de duas ferramentas:

- **Prometheus:** Ferramenta open-source de monitoramento de aplicações. Inicialmente desenvolvido pela Soundcloud, apresenta diversos recursos para acompanhamento de métricas e indicadores do lado do servidor. Permite o armazenamento e recuperação de métricas temporais, bem como consultas através da linguagem PromQL. Possui também suporte a customização de alertas e suporte nativo ao Grafana. Pode ser combinada aos actuators para obter os dados da aplicação.

- **Grafana:** Aplicação para visualização de dados massivos que possui suporte a diversas fontes de dados, como PostgreSQL, InfluxDB, Prometheus, Elasticsearch, entre outros. Permite a criação de dashboards personalizadas e complexas e também possui suporte a alertas.

Primeiramente, deve-se habilitar a coleta de métricas para que estas sejam fornecidas ao Prometheus pelo Actuator, é necessário adicionar a dependência que segue ao arquivo **pom.xml** da aplicação Java:

```
<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-registry-prometheus</artifactId>
```

```
</dependency>
```

Código 11.5 – Configuração pom.xml

Dessa forma, ao ser realizada uma requisição ao endpoint **/actuator/prometheus** as métricas são fornecidas no formato requisitado pelo Prometheus, como visto em parte da saída ilustrada:

```
...
# TYPE jvm_memory_max_bytes gauge
jvm_memory_max_bytes{area="nonheap",id="CodeHeap 'profiled nmethods'",} 1.22912768E8
jvm_memory_max_bytes{area="heap",id="G1 Survivor Space",} -1.0
jvm_memory_max_bytes{area="heap",id="G1 Old Gen",} 1.560281088E9
jvm_memory_max_bytes{area="nonheap",id="Metaspace",} -1.0
jvm_memory_max_bytes{area="nonheap",id="CodeHeap 'non-nmethods'",} 5828608.0
jvm_memory_max_bytes{area="heap",id="G1 Eden Space",} -1.0
jvm_memory_max_bytes{area="nonheap",id="Compressed Class Space",} 1.073741824E9
jvm_memory_max_bytes{area="nonheap",id="CodeHeap 'non-profiled nmethods'",} 1.22916864
    E8
# HELP tomcat_sessions_active_max_sessions
# TYPE tomcat_sessions_active_max_sessions gauge
tomcat_sessions_active_max_sessions 482.0
# HELP process_cpu_usage The "recent cpu usage" for the Java Virtual Machine process
# TYPE process_cpu_usage gauge
process_cpu_usage 0.03807106598984772
...
```

Código 11.6 – Métricas

Agora é necessário informar ao Prometheus a fonte de métricas que ele pode utilizar para coleta. Para isso bastar editar o arquivo **prometheus.yml** localizado em **/etc/prometheus** e adicionar o conteúdo na seção **scrape_configs**:

```
  - job_name: 'autenticacao'
    metrics_path: '/v0-6/autenticacao/actuator/prometheus'
    scrape_interval: 5s
    static_configs:
      - targets: ['autenticacao:8080']
    basic_auth:
      username: 'username'
      password: '******'
```

Código 11.7 – Arquivo prometheus.yml

Os parâmetros:

- job_name deve conter o nome desejado para o registro;
- metrics_patch indica o caminho do endpoint dos dados no formato do Prometheus;
- scrape_interval define o intervalo de busca por novos dados pelo servidor;
- targets define o hostname e porta da aplicação a ser monitorada;
- última seção é destinada à autenticação para acessar o endpoint /prometheus, caso necessária.

Com isso, ao ser acessada a interface inicial do Prometheus, localizada em http://localhost:9090, e posteriormente o menu Status -¿ Targets, é possível visualizar uma lista de aplicações monitoradas.

Figura 82 – Interface inicial do Prometheus

O Prometheus é uma ferramenta que também permite a configuração de alertas para eventos específicos e é possível visualizar tais eventos no menu **Alerts**. Para os microsserviços foram definidos os dois alertas na tabela a seguir.

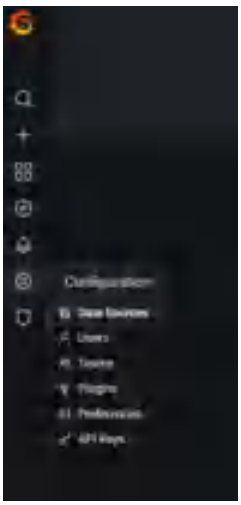
Tabela 32 – Configuração de Alertas no Prometheus

Alerta	Expressão
Informa que a instância está com a memória quase se esgotando (Acima de 80%)	(sum by(instance) (jvm_memory_used_bytes{area="heap"}) / sum by(instance) (jvm_memory_max_bytes{area="heap"})) * 100 > 80
Verifica se a instância está inativa	up == 0

Finalmente, a fim de de disponibilizar as informações coletadas e armazenadas pelo Prometheus no Grafana, basta integrar as duas ferramentas.

Para isso, deve-se acessar o Grafana, localizado em **http://localhost:3000**, e adicionar uma nova fonte de dados. Então, navega-se no menu **Data Sources**, localizado na barra lateral do Grafana, conforme figura 83.

Figura 83 – Configuração do Grafana



Posteriormente, deve-se clicar em **Add data source**, na janela exibida abaixo e selecionar Prometheus conforme figuras 84 e 85.

Finalmente, é necessário realizar a configuração, informando a url do servidor Prometheus, conforme figura 86.

Com a fonte de dados configurada, o próximo passo é criar uma nova dashboard para exibição das métricas, acessando o menu abaixo e clicando em Manage, conforme figura 87.

Figura 84 – Configuração do Grafana

Figura 85 – Configuração do Grafana

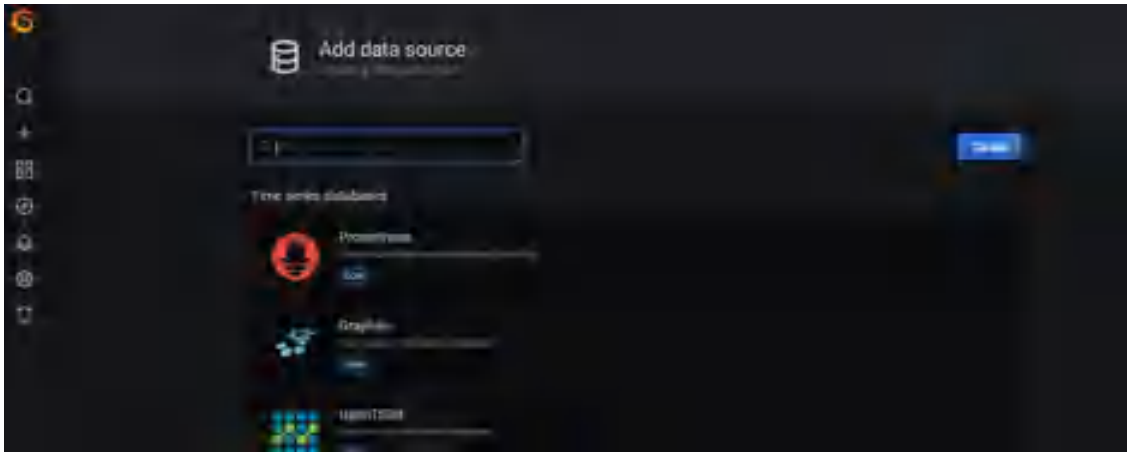
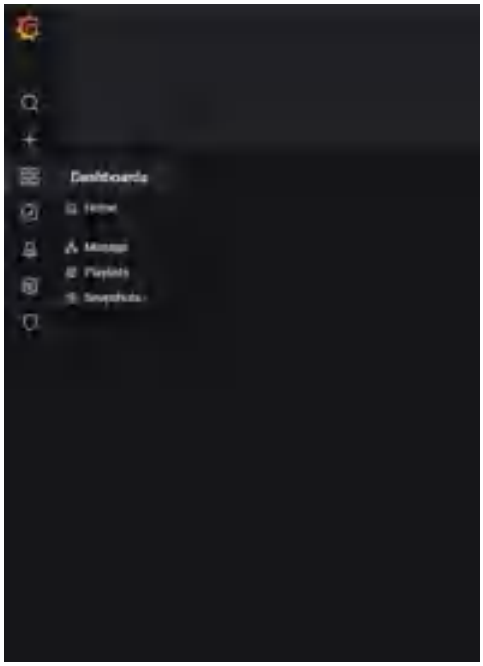


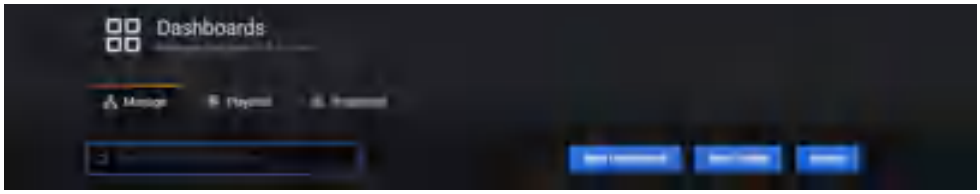
Figura 86 – Configuração do Grafana

Figura 87 – Configuração do Grafana



O Grafana permite a criação manual, bem como a importação de dashboards, que pode ser a partir de um código JSON ou do catálogo disponível em Grafana Dashboards [4], que possui modelos prontos oficiais ou feitos pela comunidade. Para a visualização das métricas dos microsserviços deste projeto, foi utilizado a dashboard JVM (Micrometer), de ID: 4701, disponível em **JVM (Micrometer)** dashboard for Grafana[5]. Para realizar a importação, deve-se clicar em Import no menu exibido na figura 88.

Figura 88 – Configuração do Grafana



Depois basta adicionar o id da dashboard no campo e clicar em load, conforme figura 89.

Na tela seguinte, conforme figura 90, os dados são checados e a fonte de dados cadastrada anteriormente (Prometheus) é selecionada. Também é possível alterar o nome conforme necessário.

Dessa forma, já é possível acessar a dashboard e verificar as informações coletadas para monitoramento dos serviços. São disponibilizados gráficos da utilização da memória, threads, processamento, eventos de log entre outros. O Grafana personalizar a dashboard importada adicionando ou removendo algum gráfico de acordo com o cenário. Um exemplo pode ser observado na figura 91.

Para alternar entre os serviços a serem verificados, deve-se clicar no menu instance, conforme figura 92, localizado na barra superior da dashboard, e selecionar o serviço que deseja verificar.

Para mais opções de configuração e personalização do Prometheus e Grafana, recomenda-se:

[4] https://grafana.com/grafana/dashboards
[5] https://grafana.com/grafana/dashboards/4701

Figura 89 – Configuração do Grafana

Figura 90 – Configuração do Grafana

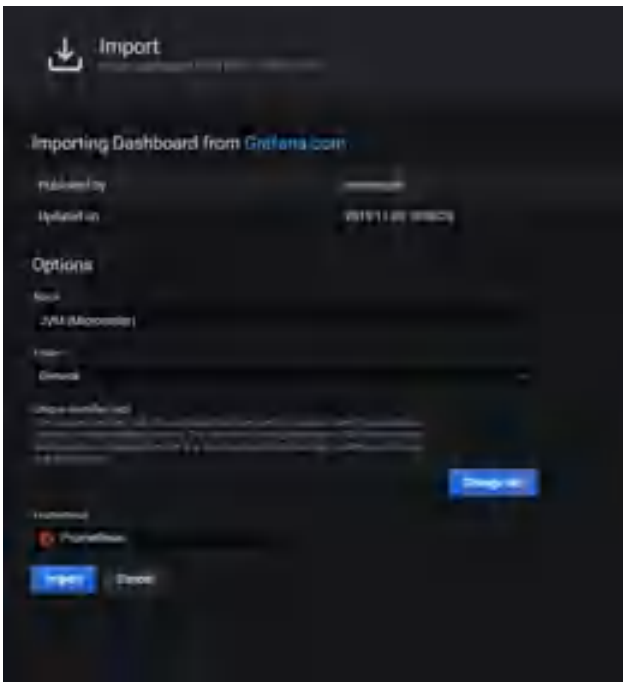


Figura 91 – Dashboard do Grafana

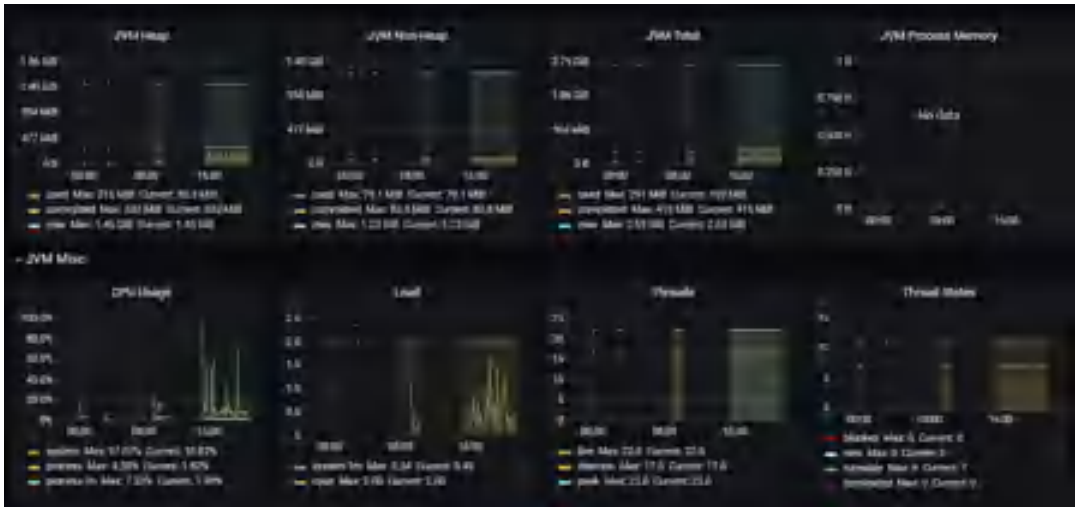
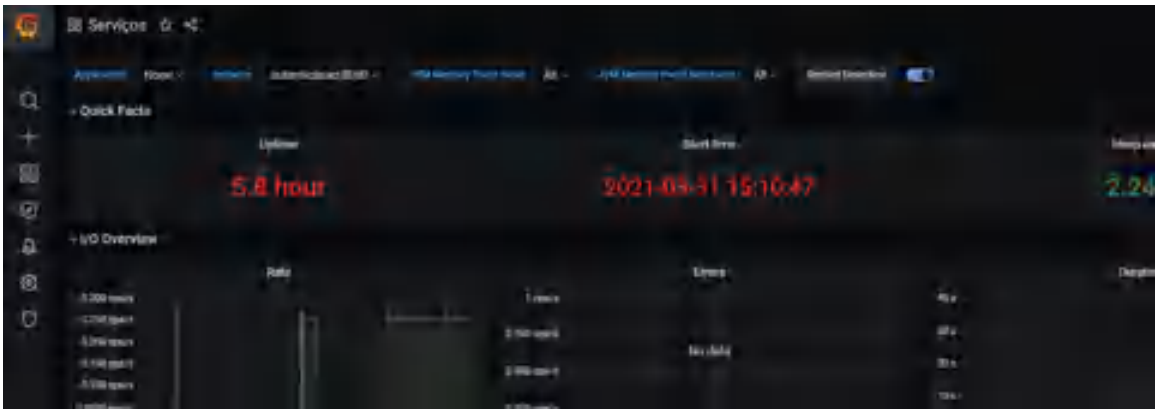


Figura 92 – Alternando entre Serviços



- Documentação do Prometheus[6]
- Documentação do Grafana[7]

11.2.2.2 Spring Boot Admin

Outra ferramenta disponível para monitoramento dos microsserviços é o **Spring Boot Admin**, um projeto desenvolvido pela empresa *codecentric* que apresenta os dados disponibilizados nos endpoints **/actuator/*** em uma interface web feita com Vue.js. Consiste em uma alternativa interessante, de simples configuração e que permite o gerenciamento centralizado de várias instâncias de serviços distintos, como exibido na figura 93, referente ao protótipo funcional.

Para utilizar Spring Boot Admin, é necessário criar uma aplicação Spring https://gitlab.com/sesp-mt/spring-boot-admin.git - Connect to preview e configurá-la como servidor para então configurar os microsserviços que se deseja monitorar como aplicações clientes desse servidor. Assim que forem executados, eles irão se registrar automaticamente na lista de serviços monitorados.

Primeiramente, na aplicação com o papel de servidor do Spring Boot Admin, deve-se adicionar ao arquivo **pom.xml** a dependência:

[6] https://prometheus.io/docs/introduction/overview/
[7] https://grafana.com/docs/grafana/latest/

Figura 93 – Spring Boot Admin


```
<dependency>
    <groupId>de.codecentric</groupId>
    <artifactId>spring-boot-admin-starter-server</artifactId>
</dependency>
```

Código 11.8 – Dependência no pom.xml

Bem como:

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>de.codecentric</groupId>
            <artifactId>spring-boot-admin-dependencies</artifactId>
            <version>${spring-boot-admin.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

Código 11.9 – Dependência no pom.xml

Recomenda-se também adicionar a dependência para ativar a página de login no servidor:

```
<dependency>
     <groupId>de.codecentric</groupId>
   <artifactId>spring-boot-admin-server-ui</artifactId>
</dependency>
```

Código 11.10 – Dependência no pom.xml

Após isso, deve-se realizar as configurações de segurança (**SecurityConfigurations.java**) no projeto **spring-boot-admin** a fim de possibilitar o acesso às informações fornecidas pelo Spring Boot Admin de maneira autenticada.

```
...
 http.authorizeRequests()
    .antMatchers(adminContextPath + "/assets/**").permitAll()
    .antMatchers(adminContextPath + "/login").permitAll()
```

```
    .anyRequest().authenticated()
    .and().formLogin()
    .loginPage(adminContextPath + "/login").successHandler(successHandler)
    .and().logout().logoutUrl(adminContextPath + "/logout")
    .and().httpBasic()
    .and().csrf().csrfTokenRepository(CookieCsrfTokenRepository.withHttpOnlyFalse())
    .ignoringAntMatchers(adminContextPath + "/instances", adminContextPath + "/actuator
        /**");
...
```

Código 11.11 – Configuração de Segurança

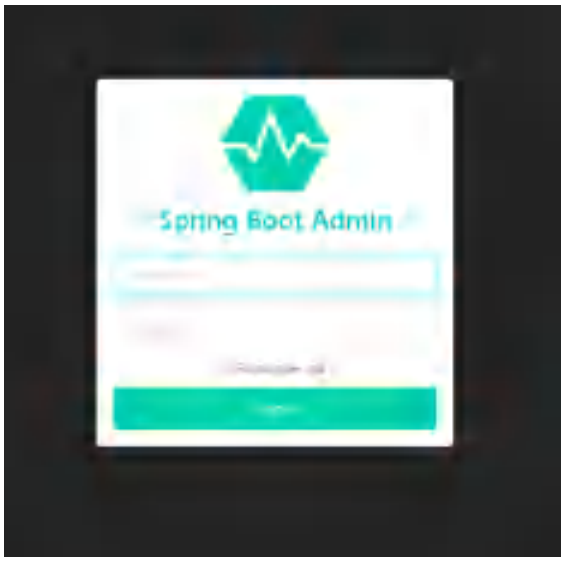
E no arquivo **application.properties**:

```
spring.security.user.name=sespmt
spring.security.user.password=sespmt
```

Código 11.12 – Configuração de propriedades

Desse modo, já é possível acessar a página de login do Spring Boot Admin (http://localhost:8007) e realizar a autenticação, embora ainda não haja nenhuma aplicação registrada para o monitoramento, conforme ilustrado na figura 94.

Figura 94 – Login do Spring Boot Admin



Nos serviços que se registrarão como clientes (microsserviços) também é necessário adicionar uma dependência ao arquivo pom.xml:

```
<dependency>
    <groupId>de.codecentric</groupId>
    <artifactId>spring-boot-admin-starter-client</artifactId>
    <version>2.4.0</version>
</dependency>
```

Código 11.13 – Configuração pom.xml dos Microsserviços

Bem como:

```
...
<dependencyManagement>
    <dependencies>
        <dependency>
```

```
            <groupId>de.codecentric</groupId>
            <artifactId>spring-boot-admin-starter-client</artifactId>
            <version>${spring-boot-admin.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
...
```

Código 11.14 – Configuração pom.xml dos Microsserviços

Agora é necessário realizar adições ao arquivo de configuração **application.properties** para que a aplicação cliente registre-se no servidor. Tomando como exemplo o microsserviço Registro Geral, teríamos:

```
spring.application.name=Registro Geral
spring.boot.admin.client.url=http://spring-boot-admin:8080/
spring.boot.admin.client.instance.name=${spring.application.name}
spring.boot.admin.client.username=sespmt
spring.boot.admin.client.password=sespmt
```

Código 11.15 – Arquivo de propriedades dos Microsserviços

Caso haja configuração de segurança para o endpoints /actuator/* também deve-se passar essa informação no arquivo:

```
spring.boot.admin.client.instance.metadata.user.name=username
spring.boot.admin.client.instance.metadata.user.password=password
```

Código 11.16 – Arquivo de propriedades dos Microsserviços

Para mais informações acerca do Spring Boot Admin, recomenda-se o acesso de: Spring Boot Admin Reference Guide[8]

O Spring Boot Admin pode também pode utilizar valores passados a parâmetros dos actuators, como as referentes ao endpoint /actuator/info

```
info.app.description=@project.description@
info.app.version=@project.version@
info.app.encoding=@project.build.sourceEncoding@
info.app.java.version=@java.version@
```

Código 11.17 – Parâmetros dos actuators

Que, no exemplo acima, são obtidas do arquivo pom.xml

```
...
<name>registro-geral</name>
<description>Servico responsavel pela manutencao do registro geral da SESP-MT</
    description>

<properties>
    <java.version>11</java.version>
    <spring-boot-admin.version>2.3.1</spring-boot-admin.version>
</properties>
...
```

Código 11.18 – Arquivo pom.xml

[8] https://codecentric.github.io/spring-boot-admin/2.3.1/

Após realizada essa configuração, os microsserviços conseguem se registrar como aplicações clientes do Spring Boot Admin e podem ser monitoradas na interface do servidor.

Figura 95 – Interface do Servidor

Ao selecionar um dos microsserviços monitorados, pode-se verificar os detalhes fornecidos pelos actuators, conforme figura 96.

Figura 96 – Detalhes de um microsserviço monitorado

É possível especificar métricas para monitoramento ao acessar o menu Métricas, na barra lateral, figura 97 e 98.

O Spring Boot Admin também permite acessar os outros endpoints de **/actuator**, exibir informações de **threaddump** e **heapdump**, bem como informações de Logs. Além disso, é possível alterar os níveis de log em tempo de execução ao navegar-se para o menu **Loggers**, conforme figura 99.

É possível personalizar o actuator, especializando endpoints, veja exemplo em https://www.baeldung.com/sprin boot-actuators.

Figura 97 – Detalhes de um microsserviço monitorado

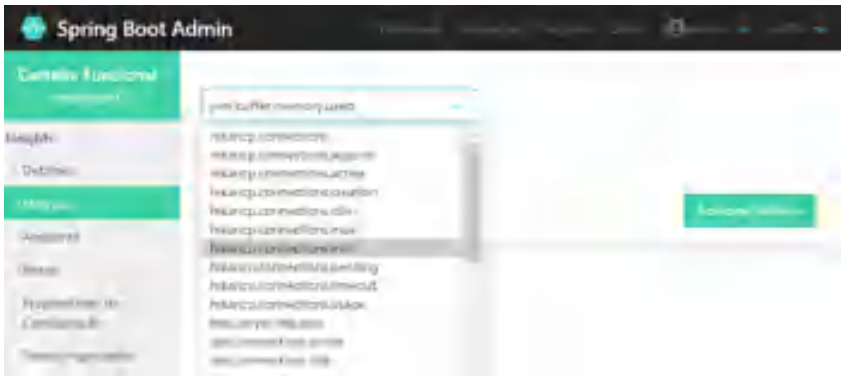
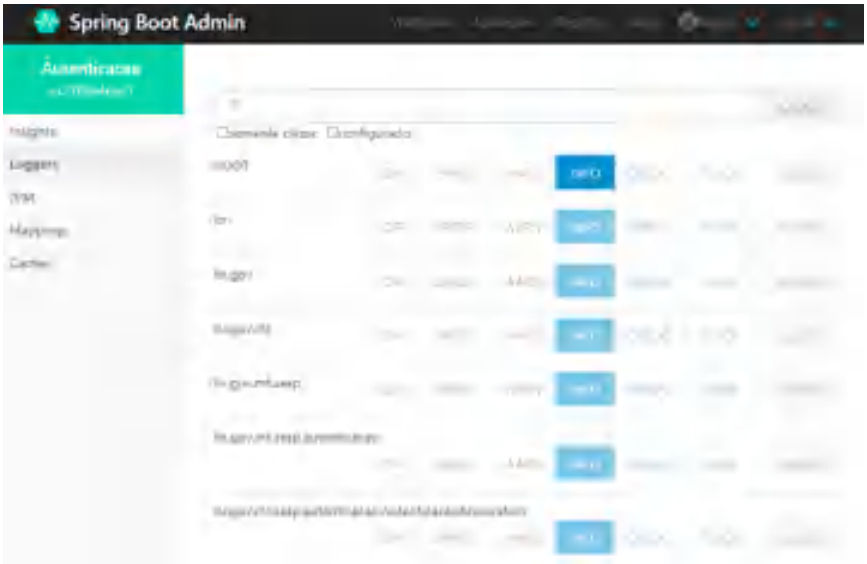


Figura 98 – Detalhes de um microsserviço monitorado

Figura 99 – Alteração de Níveis de de Log



11.3 Mensageria

Uma verificação de integridade é o aspecto mais básico do monitoramento. Envolve um comando ou conjunto de comandos que coletam algumas métricas essenciais do sistema monitorado ao longo do tempo e as testam. Por exemplo, se a VM Erlang do RabbitMQ está em execução é uma dessas verificações. A métrica, neste caso é "um processo do sistema operacional está em execução?". Os parâmetros operacionais normais são "o processo deve estar em execução". Finalmente, há uma etapa de avaliação.

Embora as verificações de integridade sejam uma ferramenta útil, elas fornecem apenas alguns *insights* sobre o estado do sistema porque são, por design, focadas em uma ou várias métricas, geralmente verificam um único nó e só podem verificar o estado desse nó em um determinado momento no tempo. Para uma avaliação mais abrangente, colete mais métricas ao longo do tempo. Isso detecta mais tipos de anomalias, pois algumas só podem ser identificadas em períodos mais longos. Isso geralmente é feito por ferramentas conhecidas como ferramentas de monitoramento.

11.3.1 Métricas de Sistema e do RabbitMQ

Algumas métricas são específicas do RabbitMQ: elas são coletadas e relatadas pelos nós do RabbitMQ. Os exemplos incluem o número de conexões abertas, número total de mensagens enfileiradas ou taxas de tráfego de comunicação entre nós. Outras métricas são coletadas e relatadas pelo kernel do sistema operacional. Essas métricas são frequentemente chamadas de métricas de sistema ou métricas de infraestrutura. As métricas de sistema não são específicas para o RabbitMQ. Os exemplos incluem taxa de utilização de CPU, quantidade de memória usada por processos, taxa de perda de pacotes de rede etc. Ambos os tipos são importantes para rastrear. As métricas individuais nem sempre são úteis, mas quando analisadas em conjunto, podem fornecer uma visão mais completa do estado do sistema. Em seguida, os operadores podem formar uma hipótese sobre o que está acontecendo e precisa ser tratada.

O RabbitMQ fornece o utilitário **rabbitmq-diagnostics** para observar **métricas do broker de mensageria e métricas de sistema**.

Para informações detalhadas acerca dos diversos parâmetros do utilitário rabbitmq-diagnostics acesse

https://www.rabbitmq.com/rabbitmq-diagnostics.8.html.

Para informações acerca de outros comandos utilitários acesse https://www.rabbitmq.com/cli.html.

Ao acessar o contêiner rabbitmq e executar o comando abaixo:

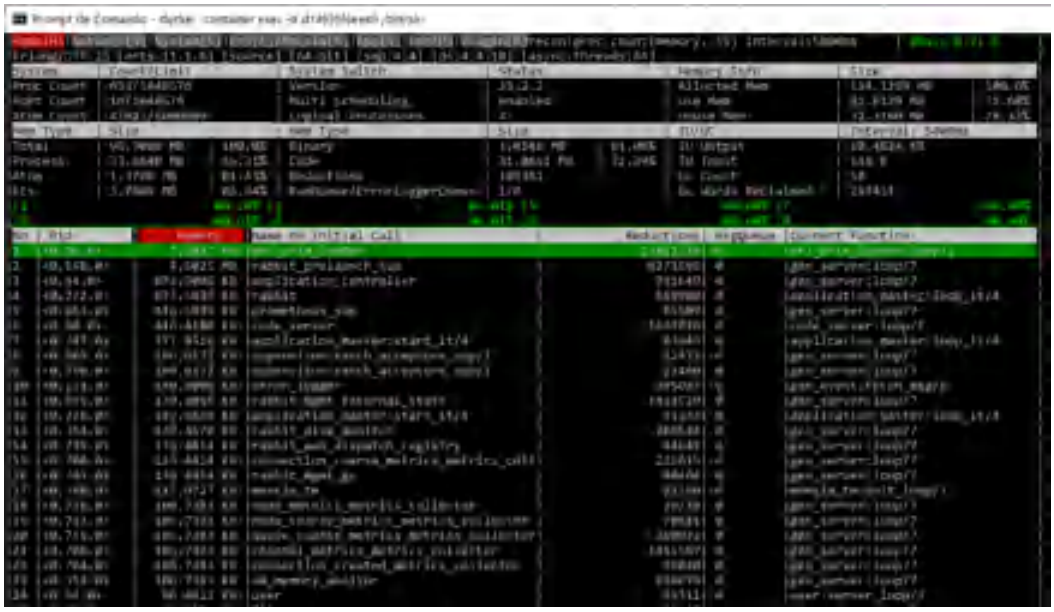
./opt/rabbitmq/sbin/rabbitmq-diagnostics observer –interval 15

Temos como resultado uma série de métricas apresentadas e atualizadas a cada 15 segundos, conforme figura 100:

Perceba que é uma ferramenta similar a **top**, **htop**, **vmstat** e que fornece acesso a métricas de sistema, do RabbitMQ e detalhes acerca de cada processo em execução:

- Informação sobre a versão em execução;

Figura 100 – Apresentação de Métricas



- Estatísticas da CPU e do agendador de tarefas;
- Alocação de memória e estatísticas de uso;
- Processos ordenados por consumo de CPU e memória;
- Informações detalhadas de cada processo, como estatísticas de TCP socket.

É importante saber que durante a operação, os nós do RabbitMQ consumirão uma quantidade variável de memória e espaço em disco com base na carga de trabalho. Quando o uso aumenta, tanto a memória quanto o espaço livre em disco podem atingir níveis potencialmente perigosos. No caso de memória, o nó pode ser eliminado pelo mecanismo de segurança do próprio sistema operacional.

Para reduzir a probabilidade desse cenário o RabbitMQ possui limites de segurança que podem bloquear as conexões que publicam mensagens. **O limite de memória padrão é definido como 40% da RAM instalada**. Observe que isso não impede que o servidor RabbitMQ de utilizar mais de 40%, é apenas o ponto em que os publishers são limitados. O coletor de lixo de Erlang pode, no pior dos casos, aumentar para o dobro da quantidade de memória a ser usada (por padrão, 80% de RAM). É altamente recomendável que o swap do SO ou a paginação de arquivos estejam habilitados.

O limite de memória pode ser modificado no arquivo de configuração do RabbitMQ:

```
vm_memory_high_watermark.relative = 0.4
```

Para informações mais detalhadas recomendamos fortemente a leitura dos seguintes tópicos:

- https://www.rabbitmq.com/monitoring.html
- https://rabbitmq.com/alarms.html
- https://rabbitmq.com/memory.html
- https://rabbitmq.com/disk-alarms.html

11.3.2 Frequência de Monitoramento

Muitos sistemas de monitoramento pesquisam seus serviços monitorados periodicamente. A frequência com que isso é feito varia de ferramenta para ferramenta, mas geralmente pode ser configurada pelo operador.

Pesquisas muito frequentes podem ter consequências negativas no sistema sob monitoramento. Por exemplo, verificações excessivas do balanceador de carga que precisam abrir uma conexão TCP de teste podem levar a uma alta rotatividade de conexão. Verificações excessivas de canais e filas no RabbitMQ aumentarão seu consumo de CPU.

O intervalo de coleta de métricas recomendado é de 15 segundos. Para coletar em um intervalo mais próximo do tempo real, use 5 segundos - mas não menos.

Para **sistemas em produção**, um intervalo de coleta de **30 ou mesmo 60 segundos** é recomendado. A API do Prometheus foi projetada para ser analisada a cada 15 segundos, incluindo sistemas em produção.

11.3.3 Interface de Gerenciamento e Sistemas de Monitoramento Externos

11.3.3.1 Interface de Gerenciamento

A localização padrão é http://¡host¿:15672, sendo que a ferramenta permite não apenas o monitoramento mas também o gerenciamento de canais, conexões, exchanges e filas.

Overview

A tela de início é de Overview que fornece uma visão geral. Nessa primeira aba temos:

- Últimas mensagens enviadas;
- Conexões abertas;
- Nodes com métricas de sistema;
- Estatísticas das métricas do broker de mensageria;
- Portas que estão sendo utilizadas.

Como podemos observar na figura 101, temos **6 conexões** (de cada microserviço), **14 exchanges**, **8 filas**, **11 consumidores** que no momento não estão consumindo. Os **12 channels** representam os canais de comunicação estabelecidos com recursos, como filas, por exemplo.

Connections

Na aba Connections podemos visualizar as conexões abertas, conforme figura 102.

Temos **6 conexões, uma para cada microserviço**. Perceba que algumas conexões possuem 4 channels, outras 2 channels, que representam **canais estabelecidos com fila**s por essa conexão.

Ao clicar em uma dessas conexões será possível visualizar métricas próprias, como o tráfego de dados e quais os channels estabelecidos. Ao clicar em um channel visualizará qual a fila relacionada a esse channel.

Figura 101 – Aba Overview

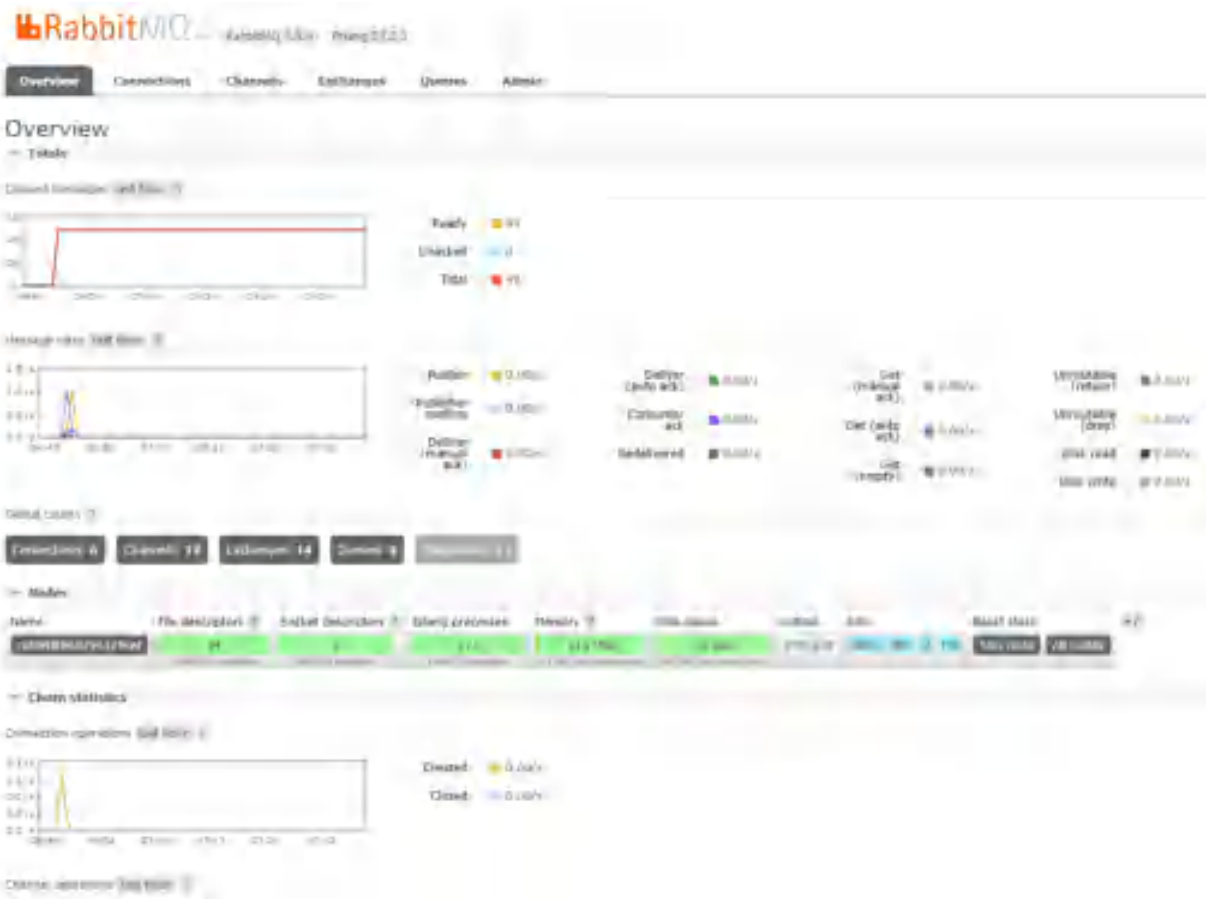


Figura 102 – Aba Connections

Channels

Para trabalhar com filas você precisa criar um canal de comunicação (**channel**), nessa aba você pode gerenciar os canais abertos. Da mesma forma que em connections, você pode clicar em um dos canais para ter mais detalhes, que são os mesmos explanados na seção anterior. A figura 103 apresenta essa interface.

Figura 103 – Aba Channels

Perceba portanto que temos **6 microsserviços** se comunicando com **8 filas** por meio de **12 canais** de comunicação.

Exchanges

Quanto aos exchanges podemos visualizar aqueles existentes, sendo que por padrão o RabbitMQ já fornece 7 exchanges que começam com nome amq. Os do protótipo funcional são os **7 últimos** que aparecem na lista, ilustradsa nas figuras 104 e 105.

Ao clicar em uma exchange é possível:

- Visualizar o tráfego de mensagens;
- Visualizar a quais filas está relacionado (binding) e a chave de roteamento;
- Criar novos bindings;
- Publicar mensagens;
- Excluir o exchange.

Queues

Nessa aba podemos visualizar todas as filas (8 para o protótipo funcional) e gerenciá-las. Um exemplo dessa interface é apresentada na figura 106.

Figura 104 – Aba Exchanges

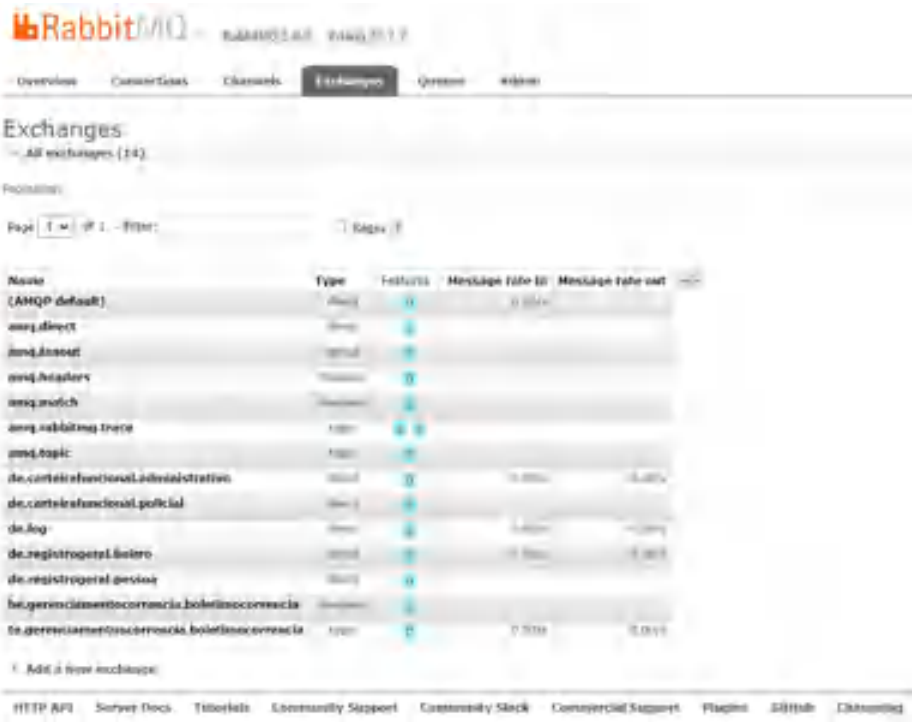


Figura 105 – Exchange de.registrogeral.bairro

Figura 106 – Aba Queues

Ao clicar em uma fila temos visualização de métricas relacionadas ao tráfego de mensagens, bem como podemos visualizar os canais abertos (channels) e as relações dessa fila com a exchange e qual a chave de roteamento, ilustrado na figura 107.

Figura 107 – Detalhe q.registrogeral.bairro.busca

No caso da imagem acima podemos visualizar a existência de 2 canais de comunicação abertos

com essa fila que possivelmente estão relacionadas a dois microsserviços.

Considerações Finais

O RabbitMQ vem com uma UI de gerenciamento e API HTTP que expõe uma série de métricas RabbitMQ para nós, conexões, filas, taxas de mensagens e assim por diante. Esta é uma opção conveniente para desenvolvimento e em ambientes onde o monitoramento externo é difícil ou impossível de introduzir.

No entanto, a UI de gerenciamento do RabbitMQ tem uma série de limitações:

- O sistema de monitoramento está interligado com o sistema que está sendo monitorado;
- Uma certa quantidade de overhead;
- Ele apenas armazena dados recentes (pense em horas, não em dias ou meses);
- Possui uma interface de usuário básica;
- Seu design enfatiza a facilidade de uso sobre a melhor disponibilidade possível.

Armazenamento de métricas de longo prazo e serviços de visualização, como Prometheus e Grafana, são opções mais adequadas para sistemas em produção. Eles oferecem:

- Desacoplamento do sistema de monitoramento do sistema que está sendo monitorado;
- Baixa sobrecarga;
- Armazenamento métrico de longo prazo;
- Acesso a outras métricas relacionadas, como as de tempo de execução Erlang;
- Interface de usuário mais poderosa e personalizável;
- Facilidade de compartilhamento de dados métricos: tanto o estado da métrica quanto os painéis;
- Permissões de acesso de métrica não são específicas para RabbitMQ.

RabbitMQ fornece suporte padrão para Prometheus a partir da versão 3.8. É o recomendado para ambientes de produção.

11.3.3.2 AlertManager

Antes de configurarmos o Prometheus é interessante habilitar e configurar o AlertManager, uma ferramenta que permite receber os alertas do Prometheus e realizar notificações em diferentes meios de comunicação. A configuração abaixo é para o envio dos alertas por e-mail:

```
#alertmanager.yml
global:
  smtp_smarthost: 'mailhog:1025'
  smtp_from: 'admin@alertmanager.localhost'
  #smtp_auth_username: 'alertmanager'
  #smtp_auth_password: 'password'
  smtp_require_tls: false

templates:
- '/etc/alertmanager/template/*.tmpl'

route:
  group_by: ['alertname', 'cluster', 'service']
  group_wait: 30s
  group_interval: 1m
  repeat_interval: 30m

receivers:
- name: 'default-root-route'
  email_configs:
  - to: 'responsavel@alertmanager.localhost'
```

Código 11.19 – Configuração do AlertManager

Sobre a configuração **route** é importante conhecer as propriedades:

- **group_by:** permite agrupar alertas que possuam labels iguais. Por exemplo, múltiplos alertas do **cluster=A** e **alertname=LatencyHigh** podem ser agrupadas para um único envio.
- **group_wait:** determina quanto tempo aguardar para enviar a notificação de alertas pertencente a um grupo quando da primeira ocorrência. Permite assim coletar diversos alertas disparados de um mesmo grupo. Normalmente é um valor de 0s a poucos minutos, o valor default é de 30s.
- **group_interval:** determina quanto tempo aguardar antes de enviar nova notificação sobre novos alertas que foram adicionados ao grupo de alertas que já teve uma notificação enviada. Normalmente é um valor de aproximado de 5m ou mais, o valor default é de 5m.
- **repeat_interval:** determina quanto tempo aguardar antes de enviar nova notificação sobre alerta que já teve uma notificação enviada. É como um lembrete sobre um alerta que ainda não foi resolvido. Normalmente é um valor de aproximado de 3h ou mais, o valor default é de 4h.

Para maiores informações sobre a configuração dos alertas acesse https://prometheus.io/docs/alerting/latest/configuration/.

11.3.3.3 Prometheus

Por padrão a versão do RabbitMQ no protótipo funcional já está com o plugin do prometheus habilitado, mas pode ser necessário habilitar o plugin do prometheus por meio do comando abaixo:

./opt/rabbitmq/sbin/rabbitmq-plugins enable rabbitmq_prometheus

Para saber se o plugin está ativo basta acessar a URL http://localhost:15692/metrics e verificar se está respondendo.

Para configuração do Prometheus devemos alterar o arquivo **prometheus.yml** localizado em **/etc/prometheus**:

```
#prometheus.yml
global:
  scrape_interval:     15s
  evaluation_interval: 15s

rule_files:
  - alerts.yml

scrape_configs:
  - job_name: 'prometheus-docker'
    static_configs:
      - targets: ['localhost:9090']

  - job_name: 'rabbitmq'
    scrape_interval: 5s
    static_configs:
      - targets: ['rabbitmq:15692']

alerting:
  alertmanagers:
    - static_configs:
      - targets: ['alertmanager:9093']
```

Código 11.20 – Configuração do Prometheus

Nesse arquivo de configuração a atualização está programada de forma global a cada 15 segundos e foi adicionado mais um elemento a ser monitorado, de job_name: rabbitmq cujo alvo é o endereço rabbitmq:15692. Para o rabbitmq a coleta das métricas ocorre a cada 5 segundos. Perceba no final do arquivo de configuração que também foi adicionada uma referência ao AlertManager, para que os alertas sejam encaminhados e as notificações sejam realizadas.

Caso o arquivo de configuração sofra modificações é possível recarregar as configurações por meio do comando:

curl -X POST http://localhost:9090/-/reload

O Prometheus, que responde na porta 9090, possui uma interface amigável para visualização das configurações, conforme figura 108.

Na tabela 33 apresentamos os alertas configurados no Prometheus (**alerts.yml**):

Uma relação de alertas para diversas ferramentas pode ser obtida em https://awesome-prometheus-alerts.grep.to/rules.

A seguir apresentamos a primeira parte do arquivo **alerts.yml** demonstrando a criação de um alerta:

```
#prometheus.yml
global:
  scrape_interval:     15s
  evaluation_interval: 15s

rule_files:
  - alerts.yml
```

Figura 108 – Interface do Prometheus

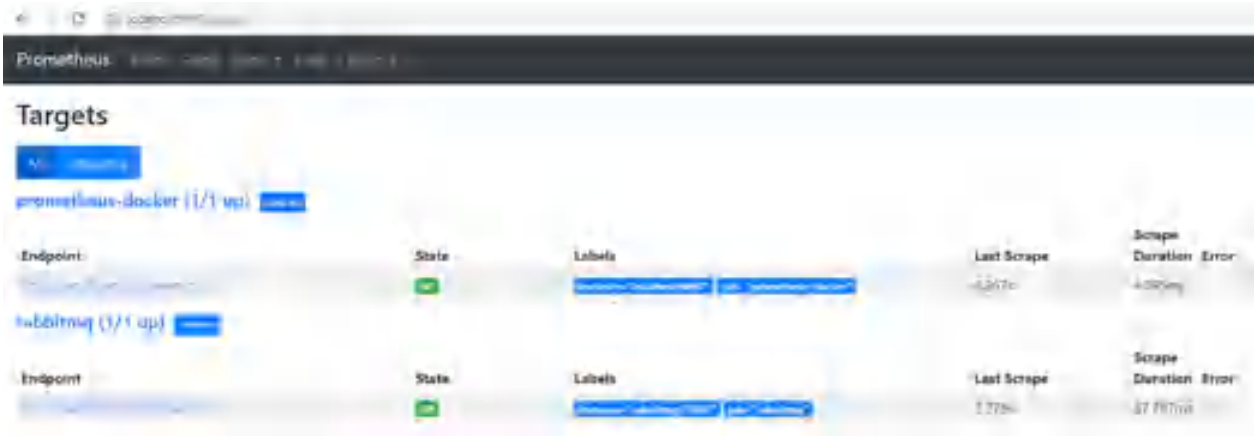


Tabela 33 – Recursos Acessíveis pelos endpoints

Alerta	Expressão
Verifica se há pelo menos uma instância do RabbitMQ ativa	`up{job="rabbitmq"} < 1`
Verifica se há pelo menos 3 nós (cluster) para o RabbitMQ	`sum(rabbitmq_build_info) < 3`
Verifica se há instâncias de versões diferentes do RabbitMQ em execução	`count(count(rabbitmq_build_info) by (rabbitmq_version)) > 1`
Verifica se o RabbitMQ está utilizando mais de 90% da memória RAM disponibilizada	`rabbitmq_process_resident_memory_bytes / rabbitmq_resident_memory_limit_bytes * 100 > 90`
Verifica se o RabbitMQ está utilizando mais de 90% dos file descriptors disponíveis	`rabbitmq_process_open_fds / rabbitmq_process_max_fds * 100 > 90`
Verifica se a quantidade de mensagens sem confirmação está muito alta (> 1000)	`sum(rabbitmq_queue_messages_unacked) by (queue) > 1000`
Verifica se a quantidade de conexões abertas está muito alta (> 1000)	`rabbitmq_connections > 1000`
Verifica se há filas sem um consumidor associado	`rabbitmq_queue_consumers < 1`
Verifica se há mensagens que não estão sendo roteadas	`increase(rabbitmq_channel_messages_unroutable_returned_total[1m]) > 0 or increase(rabbitmq_channel_messages_unroutable_dropped_total[1m]) > 0`

```
8
9  scrape_configs:
10   - job_name: 'prometheus-docker'
11     static_configs:
12       - targets: ['localhost:9090']
13
14   - job_name: 'rabbitmq'
15     scrape_interval: 5s
16     static_configs:
17       - targets: ['rabbitmq:15692']
18
19 alerting:
20   alertmanagers:
21     - static_configs:
22       - targets: ['alertmanager:9093']
```

Código 11.21 – Arquivo alerts.yml

Definimos todos os alertas dentro de um grupo intitulado **RabbitMQAlerts** e o primeiro alerta é o de verificação se há uma instância do RabbitMQ em execução (**InstanceDown**).

O parâmetro **for: 0m** indica que o Prometheus não deve aguardar um período de tempo para verificar novamente se a instância continua inativa, ou seja, assim que é detectada a inatividade do RabbitMQ o alerta deve ser imediatamente disparado.

Em **labels** podemos definir alguns parâmetros como a severidade (**severity**), podendo ter valores como **critical** ou **warning**, por exemplo. Esses parâmetros podem auxiliar na realização de filtros/seleção de alertas.

Já em **annotations** podemos descrever melhor o que o alerta significa, inserindo uma mensagem resumida (**summary**) ou mais detalhada (**description**).

Os alertas são apresentados conforme figura 109 na ferramenta:

Figura 109 – Alertas do Prometheus

Os alertas na cor verde (Inactive) indicam que os alertas não estão ativos, visto que a condição de disparada não é verdadeira.

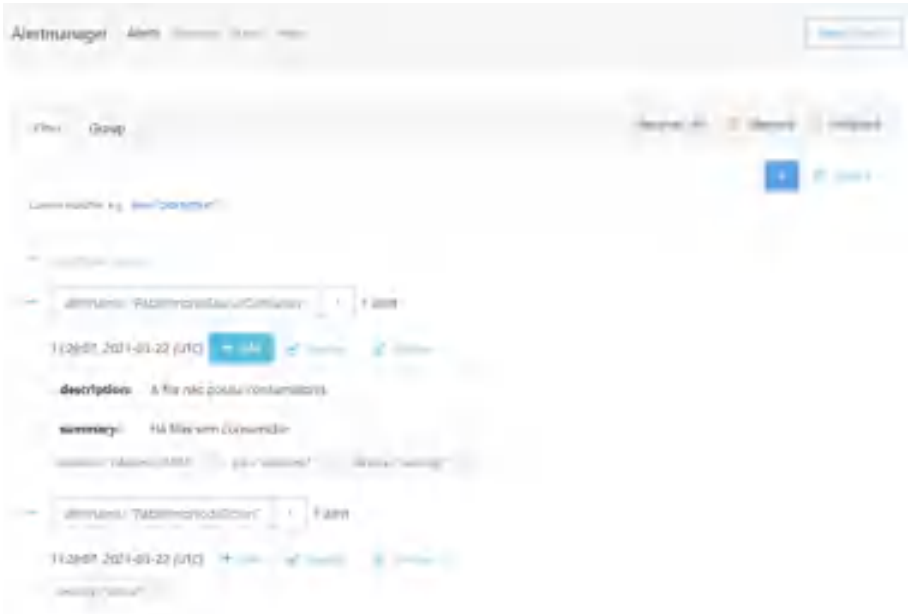
Os alertas em cor amarela (Pending) indicam que as condições para disparada do alerta ocorreram, mas está aguardando o tempo configurado em for, que representa a espera para ver se a condição volta a normalidade.

Caso a condição de disparada do alerta permaneça após o tempo configurado em for, então o alerta é disparado e representado na cor vermelha (Firing), conforme figura 110:

Figura 110 – Alertas do Prometheus

Os alertas que foram disparados são enviados para o AlertManager, conforme pode ser averiguado na figura 111:

Figura 111 – Interface AlertManager



E como descrito anteriormente o AlertManager está configurado para o envio de e-mails de notificação para os alertas recebidos, conforme figura 112:

É possível observar e-mails de alerta provenientes tanto do AlertManager quanto do Grafana.

Figura 112 – Configuração de e-mail no AlertManager

11.3.3.4 Grafana

A ferramenta Grafana permite personalizar Dashboards apresentando as métricas em diversos tipos de gráficos e também permite a criação de alertas que podem notificar em diversos meios de comunicação. No protótipo funcional a notificação está configurada e pode ocorrer tanto pelo AlertManager como pelo Grafana.

Na figura 113 apresentamos o dashboard personalizado para visualização das métricas do RabbitMQ:

Figura 113 – Dashboard personalizado

Os painéis do RabbitMQ no Grafana usam cores diferentes para capturar os seguintes estados métricos:

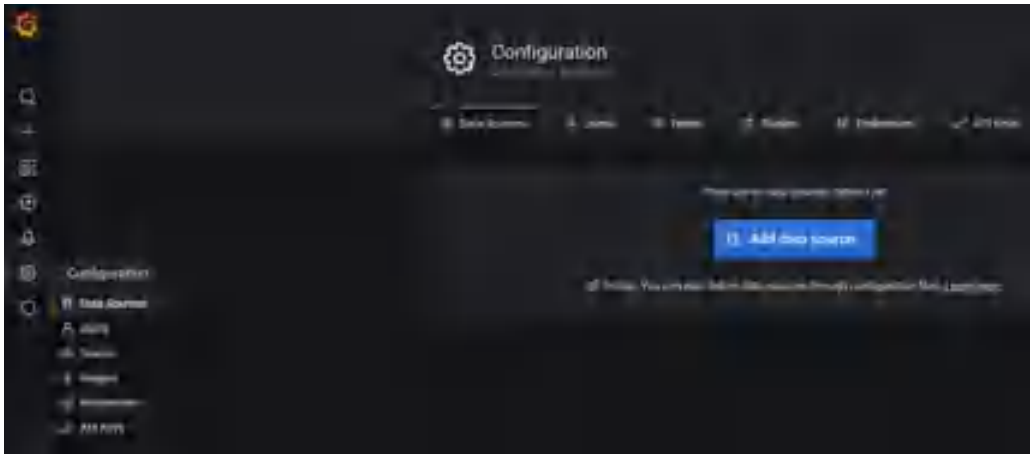
- Verde: significa que o valor da métrica está dentro de uma faixa saudável;
- Azul: significa subutilização ou alguma forma de degradação;
- Vermelho: significa que o valor da métrica está abaixo ou acima da faixa considerada saudável.

Importando Dashboard para o RabbitMQ

Para podermos construir ou importar um dashboard precisamos primeiro configurar a fonte de dados para as métricas, em nosso caso a localização do Prometheus.

Primeiro devemos acessar o menu **Configuration** – **Data Sources** e adicionar um datasource (**Add data source**), conforme figura 114:

Figura 114 – Importar Dashboard



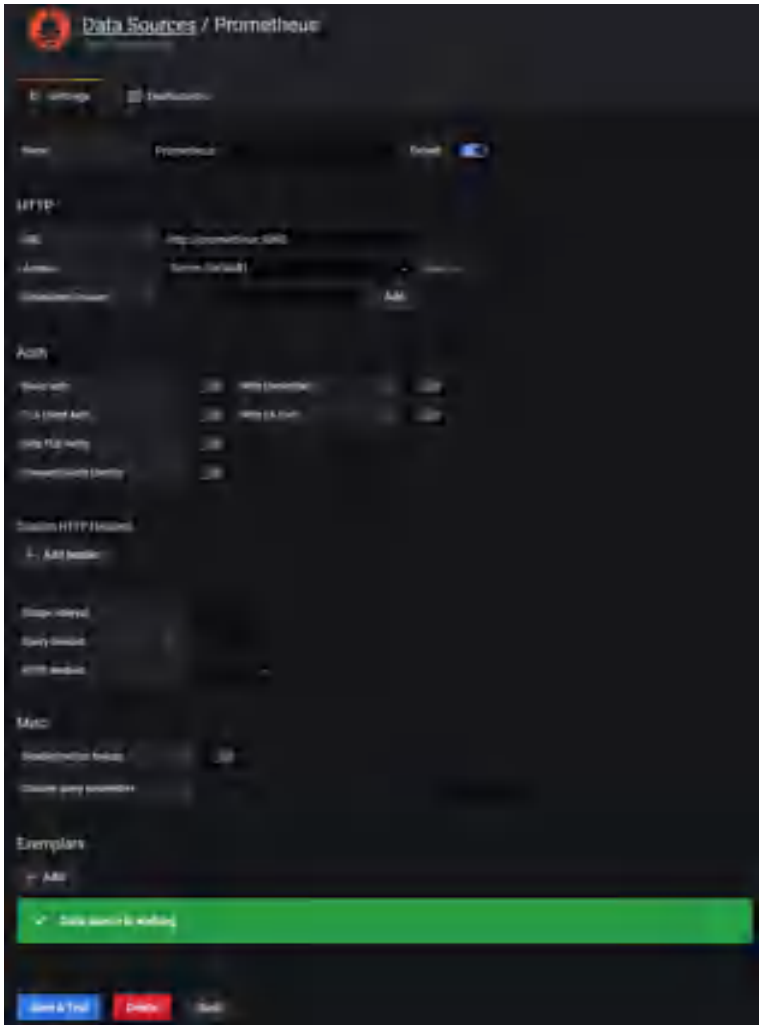
A ferramenta permite obter dados de uma série diversificada de fontes de dados, o Prometheus é a primeira da lista, conforme figura 115:

Figura 115 – Lista Dashboard

Agora podemos configurar o Data Source do Prometheus conforme figura 116:

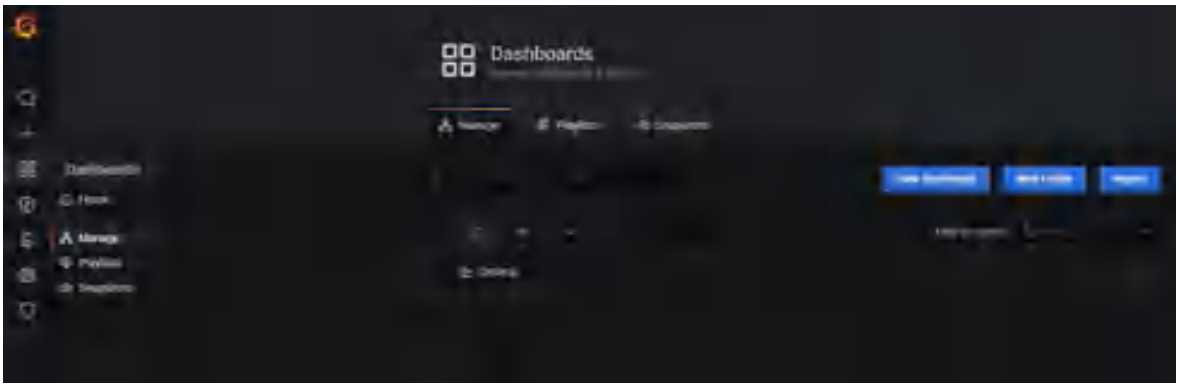
Existem diversos dashboards personalizados disponíveis na Internet e de fácil importação pela ferramenta. A busca pode ser realizada em https://grafana.com/grafana/dashboards. O que foi importado para o protótipo funcional é o dashboard de **ID 10991** (RabbitMQ - Overview), localizado em https://grafana.com/grafana/dashboards/10991.

Figura 116 – Data Source do Prometheus



Para importação precisamos acessar o menu **Dashboards** – **Manage**. É possível criar seu próprio dashboard, organizar diversos em pastas e importar. Vamos utilizar a opção de Importação, conforme ilustrado na figura 117:

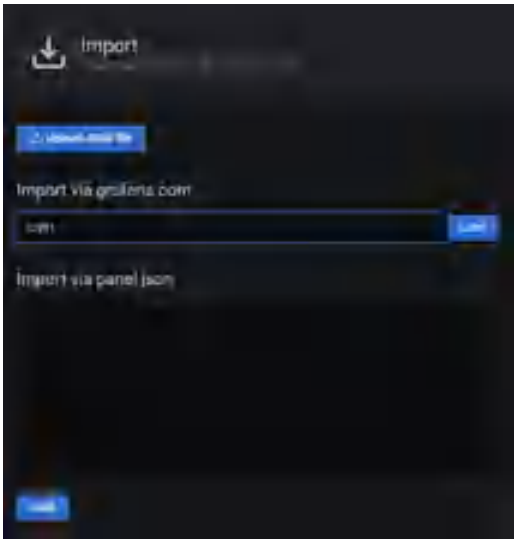
Figura 117 – Dashboards Manage



Agora basta informar o ID do dashboard ou carregar o JSON para importação, conforme figura 118:

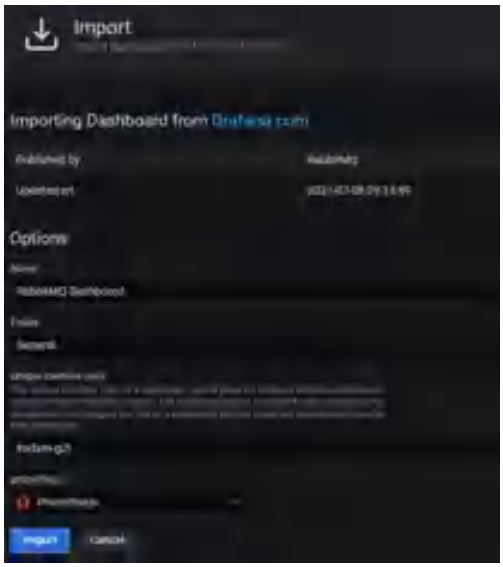
Após fornecido o ID do dashboard a ser importado são apresentadas abaixo as configurações

Figura 118 – Carregar o JSON



solicitando o apontamento para a fonte de dados desse dashboard, por isso a seleção do Prometheus, conforme figura 119:

Figura 119 – Importação do Dashborad do Grafana

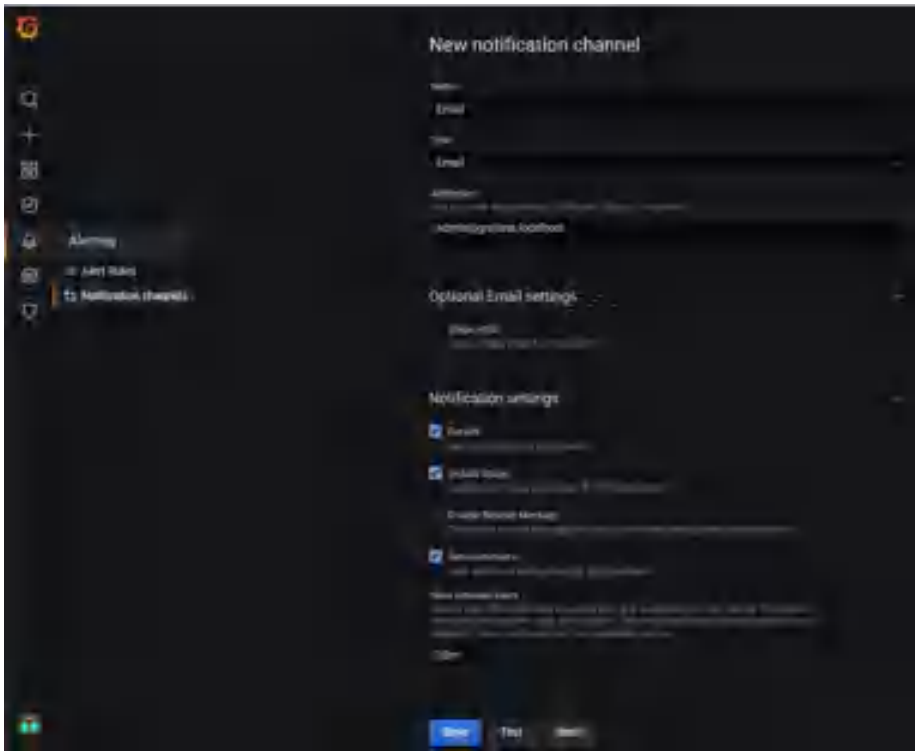


Criando Alertas

Para criação de alertas precisamos primeiro configurar qual o canal de comunicação que será utilizado para o envio das notificações. Primeiro devemos acessar o menu **Alerting** – **Notification** channels e teremos a imagem abaixo com os dados de configuração de um canal para envio de e-mails, conforme figura 120:

Configuramos para o e-mail ser o canal de configuração padrão, caso não seja definido um para o alerta. A mensagem enviada por e-mail irá conter também a imagem capturada quando o alerta for disparado e definimos que a cada 30 minutos um novo e-mail deve ser enviado caso o alerta continue ativo.

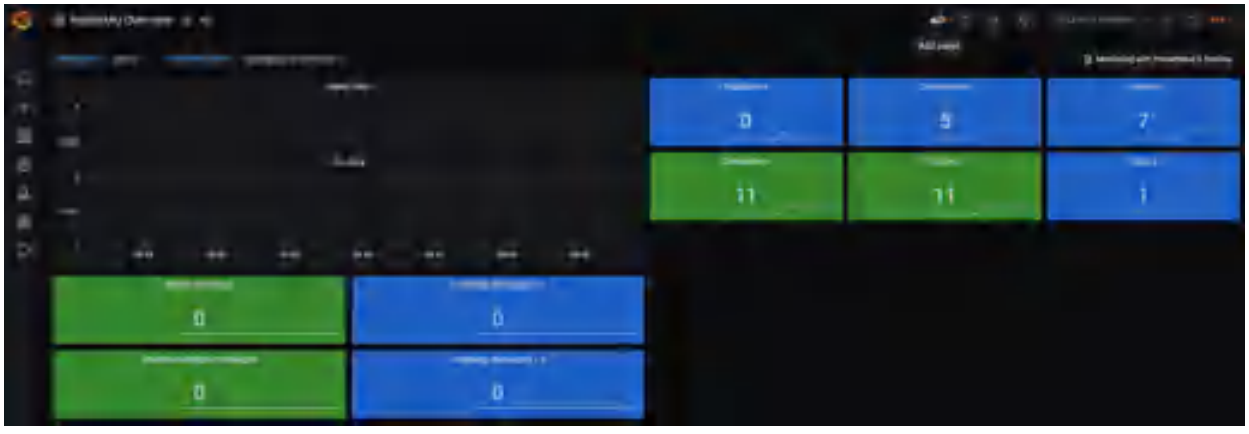
Figura 120 – Configurando Alertas



Para configuração do canal de comunicação por meio de arquivo, como é o caso do protótipo funcional, acesse https://grafana.com/docs/grafana/latest/administration/provisioning/.

Para poder criar alertas a ferramenta Grafana exige que um painel seja criado para cada expressão que se deseja observar, mesmo que não haja o interesse de observar graficamente o valor da expressão. No canto superior direito temos um ícone para "*Add Panel*". Veja que na 121 temos um novo painel sem dados.

Figura 121 – Painel de Dados



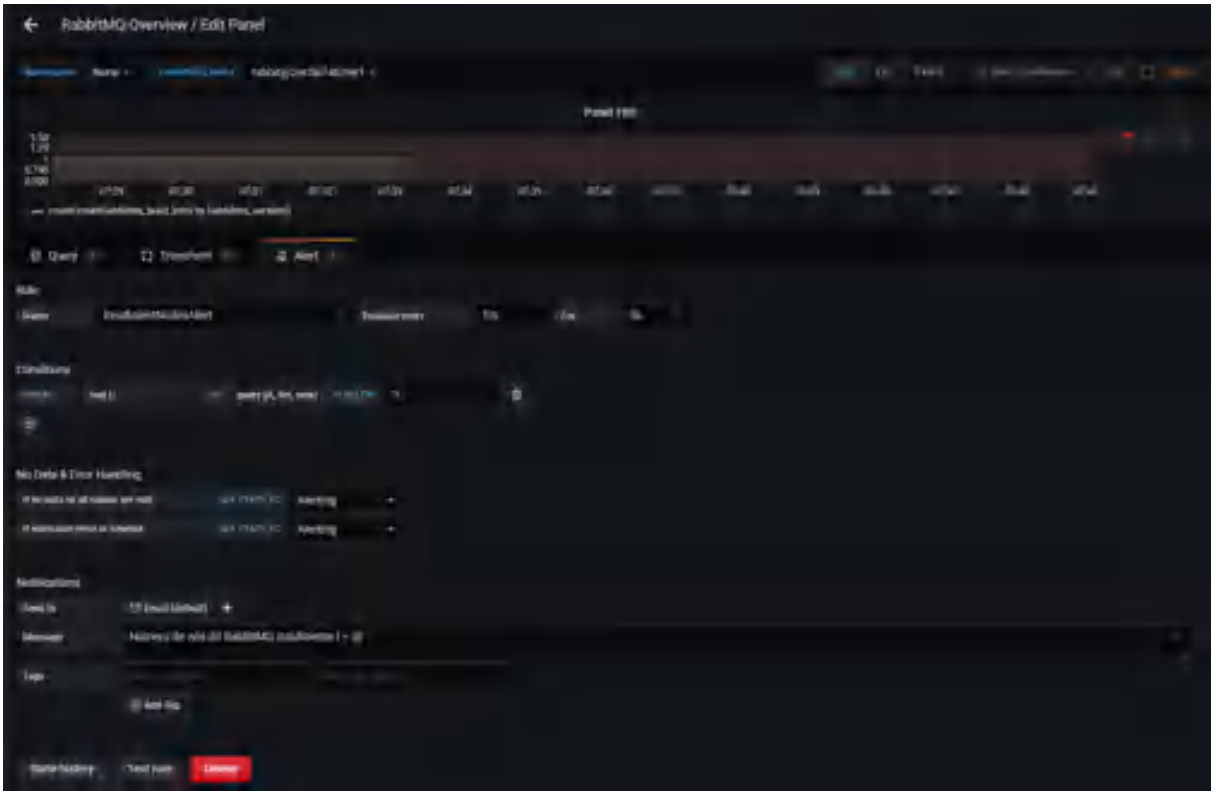
Ao editar o painel devemos na aba **Query** selecionar a fonte de dados das métricas (**Prometheus**) e definimos como exemplo a expressão **sum(rabbitmq_build_info)** para verificar a qtde de instâncias RabbitMQ disponíveis. Veja a configuração na figura 122:

Com a expressão definida e executada podemos definir um alerta na aba Alert. Foi definido o nome do alerta como InsuficientNodesAlert que é avaliado a cada 1 minuto e que não possui tempo

Figura 122 – Instâncias RabbitMQ Disponíveis

de espera (for=0s). A condição de disparada do alerta é quando o último (**last**) valor da query for abaixo (**IS BELOW**) de 3. Veja a configuração na figura 123 e perceba que após configurado o alerta o gráfico de linha ficou na cor vermelha, ou seja, um alerta será disparado:

Figura 123 – Definição do Alert



Perceba também que foi configurado para que caso a expressão não retorne valores ou se ocorrer erro de execução ou timeout, o alerta será disparado.

11.4 Log

Duas ferramentas são muito populares no mercado para o monitoramento de log, são elas Graylog e Kibana, nas próximas seções realizamos uma breve análise dessas duas.

11.4.1 Graylog (ELG)

ELG é um acrônimo para Elasticsearch, Logstash e Graylog, três ferramentas open-source que são utlizadas para armazenamento de grande volume de dados, processamento e coleta de dados de log e visualização de dados, respectivamente. Juntas compõem um stack comumente utilzado para gerenciamento de logs que respeita o seguinte fluxo: os dados de log são entregues a um servidor e processados com o Logstash e inseridos no Elasticsearch para, posteriormente, serem visualizados com o Graylog.

O Elasticsearch foi o primeiro a ser lançado, em 2010, como um mecanismo de busca distribuído. Sua combinação com as outras duas ferramentas fornece uma solução completa para monitoramento bastante utilizada no mercado, que pode ser complementada com a utilização de plugins.

O Graylog é uma ferramenta de gerenciamento de logs criada em 2009 como software livre e que teve sua primeira versão comercial lançada em 2016. Permite a captura, armazenamento e análise em tempo real de um grande volume de dados através da utilização de arquitetura e armazenamento escalável utilizando outras duas ferramentas, Elasticserch, para armazenamento dos dados de log, e MongoDB, onde os dados de configuração são salvos.

Possui um sistema de notificações e ações que permite tomar decisões baseadas em eventos preestabelecidos. Apresenta também uma sintaxe de busca que possibilita realizar consultas nos dados armazenados.

O Graylog também fornece uma interface web para gerenciamento dos logs e permite a criação de gráficos básicos para a análise visual. Além disso, possui uma Api Rest que facilita a integração com outras ferramentas de monitoramento e automação enriquecendo a coleta e análise dos dados.

Sua versão gratuita possui mecanismos de autenticação e autorização incluídos bem como integração com LDAP.

11.4.2 Kibana (ELK)

ELK também é um acrônimo para Elasticsearch, Logstash e Kibana, ferramentas open-source que seguem a mesma stack apresentada anteriormente, sendo que a visualização dos dados é por meio da ferramenta Kibana e não Graylog.

O ELK é uma stack de código aberto mas possui suporte pago pela Elastic. Além disso, há a possibilidade da utilização paga como SaaS caso o usuário não deseje realizar a implantação manual do ambiente.

Permite a criação de pipelines customizados de log com o Logstash, filtragem detalhada de dados e modelagem da maneira como os dados devem ser salvos para poderem ser visualizados com o Kibana, que fornece opções de gráficos avançados.

No ELK os mecanismos de autenticação e autorização são fornecidos na versão paga, bem como soluções avançadas para monitoramento dos dados.

11.4.3 Tabela Comparativa

Tabela 34 – Comparativo Graylog vs Kibana

	Graylog	Kibana
O que é	Ferramenta de gerenciamento de logs que funciona em conjunto com o Elasticsearch e o MongoDB.	Utilizado originalmente para análise de Big Data e começou a ser usada também para gerenciamento de logs.
Sistema de alerta	Disponível gratuitamente	Disponível apenas na versão paga
Visualização	Básica com a utilização da interface web, havendo a possibilidade da utilização de outras ferramentas como o Grafana para a elaboração de gráficos mais avançados.	Possibilidade de criação de gráficos complexos com o Kibana.
Plugins	Poucos disponíveis	Grande variedade
Consumo de Recursos	Menor	Maior
Usabilidade	Simples de utilizar com uma interface intuitiva.	Possui uma curva de aprendizado mais elevada, demandando maior refinado e pesquisa para uma operação completa.

Tendo em vista os pontos abaixo é que optamos pelo Graylog para monitoramento de log:

- Ferramenta especializada em log;
- Menor curva de aprendizado;
- Sistema de alerta é grautito.

11.4.4 Filtrando Log

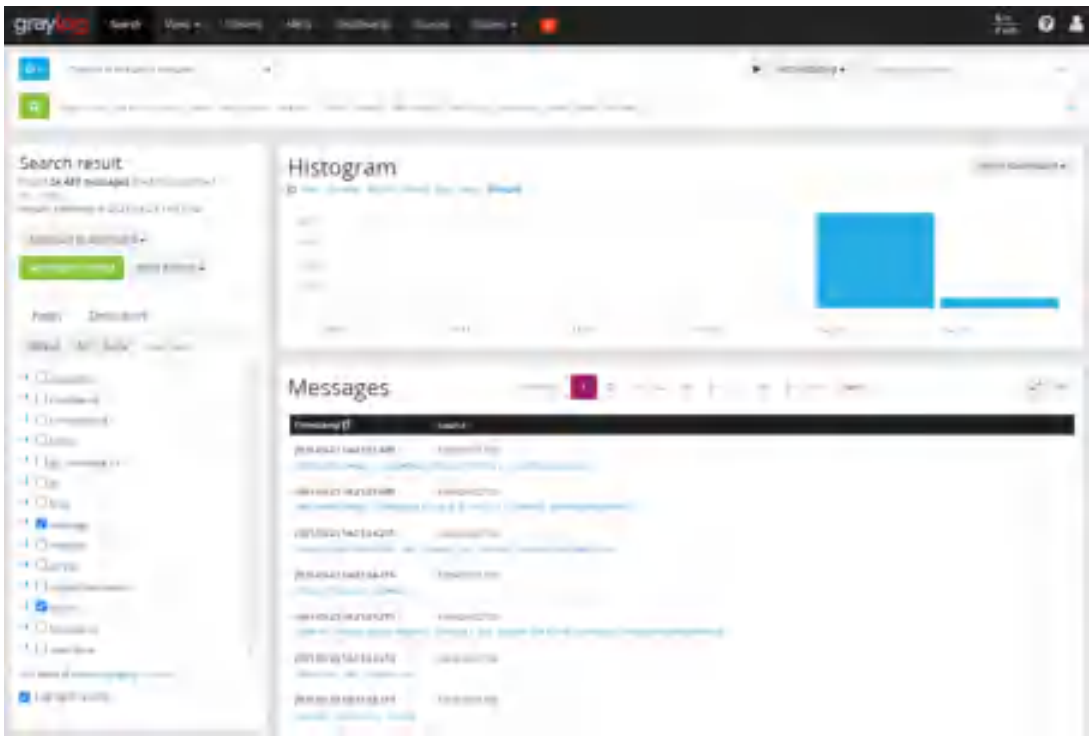
Quando acessamos o Graylog a primeira opção de menu é a Search, que permite realizar busca sobre os dados de log, conforme figura 124.

Por padrão estamos enviando todos os níveis de log e por isso a quantidade de mais de 20 mil logs registrados quando iniciamos a aplicação gerenciamento-roubo. Perceba que por padrão a busca apresenta os resultados dos últimos 5 minutos por padrão, mas que pode ser modificado na lista suspensa.

No lado esquerdo da interface podemos visualizar diversos campos que podem ser utilizados para filtro. Normalmente os campos abaixo são muito utilizados na pesquisa:

- **application**: filtra os logs de determinada aplicação ou microserviço em nosso caso;
- **correlationId**: filtra os logs de uma determinada ação do usuário;
- **level**: filtra os logs por nível de severidade (info, warning, error,...)
- **username**: filtra os logs de um usuário.

Figura 124 – Filtrando Log



Com relação aos níveis de severidade (level) o Graylog identifica de acordo com o modelo syslog:

- **0 Emergency**:erro fatal, sistema está inoperante
- **1 Alert**: uma ação precisa ser realizada imediatamente
- **2 Critical**: condições críticas
- **3 Error**: erro
- **4 Warning**: atenção
- **5 Notice**: normal
- **6 Informational**: informativo
- **7 Debug**: depuração

Como estamos utilizando Log4J2 o mapeamento correspondente ocorre de acordo com:

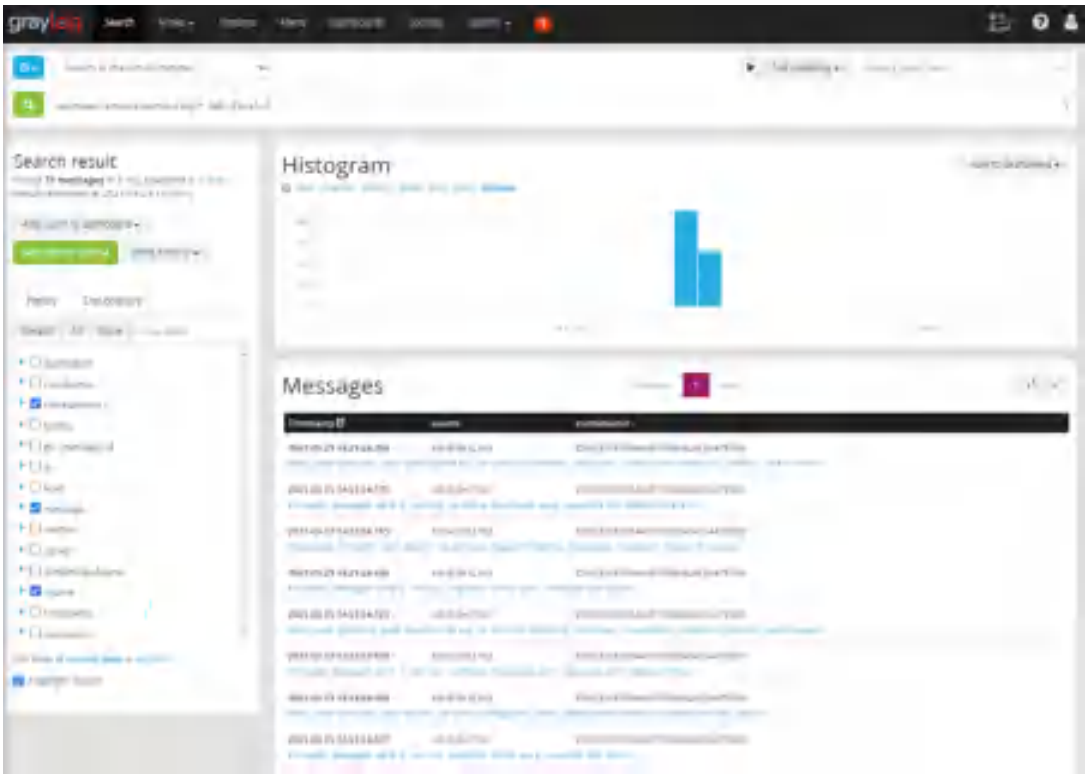
- **2 FATAL**
- **3 ERROR**
- **4 WARN**
- **6 INFO**
- **7 Outros (DEBUG, TRACE, ...)**

Por exemplo, podemos filtrar os logs de nível **INFO** do usuário **antedimentociosp** de acordo com a expressão abaixo:

username:atendimentociosp* AND level:6

Vamos marcar para exibir no resultado o **correlationId** tbm e veja na figura 125 que o resultado apresenta logs das ações realizadas por esse usuário em nível INFO.

Figura 125 – Filtrando com correlationId



Mas temos como resultado o log de diversas ações diferentes realizadas por esse usuário, caso deseje visualizar os logs relacionados a uma ação específica do usuário precisamos identificar o correlationId. Ao adicionar na pesquisa uma busca por parte da mensagem que consta “novo boletim“ vamos identificar o correlationId referente a ação do usuário responsável pelo cadastro de um novo boletim, conforme figura 126.

username:atendimentociosp* AND level:6 AND message:”novo boletim”

Veja que obtivemos um único resultado e que o correlationId é 4198031B3D83479394057AEF422E528E. Agora podemos filtrar todos os logs decorrente da ação de cadastro de boletim de ocorrência por esse usuário, conforme figura 127.

username:atendimentociosp* AND level:6 AND correlationId:4198031B3D83479394057AEF422E528E

Temos como resultado 6 mensagens de log de nível INFO para a acão de cadastro de boletim de ocorrência e conseguimos perceber o fluxo dos acontecimentos, desde a consulta ao resgistro

Figura 126 – Filtrando com parte do nome da ação

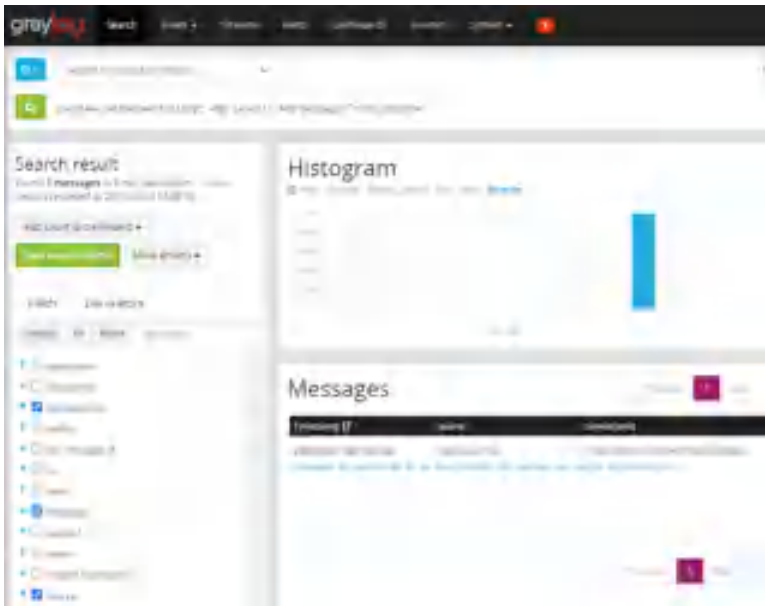
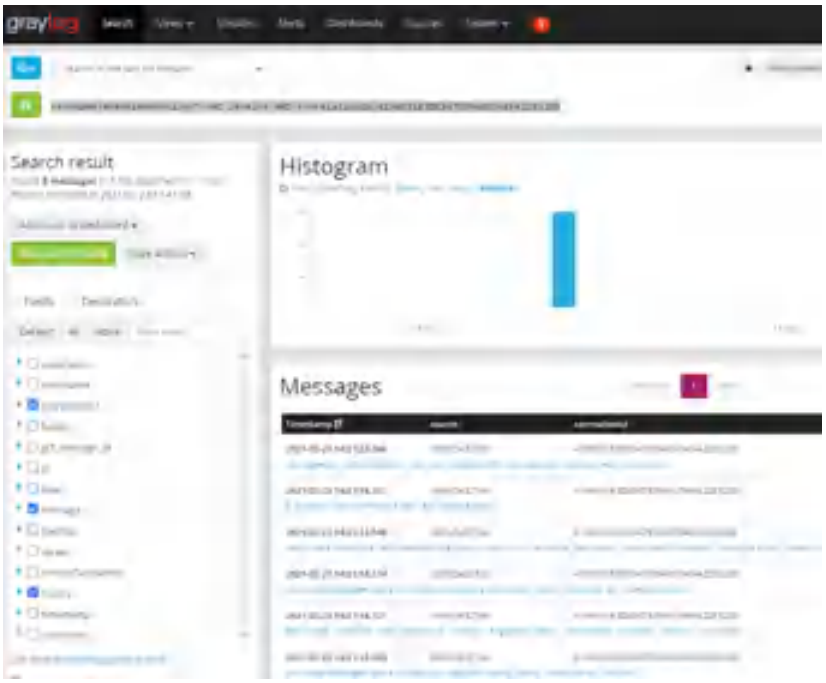


Figura 127 – Ocorrências de Log por usuário



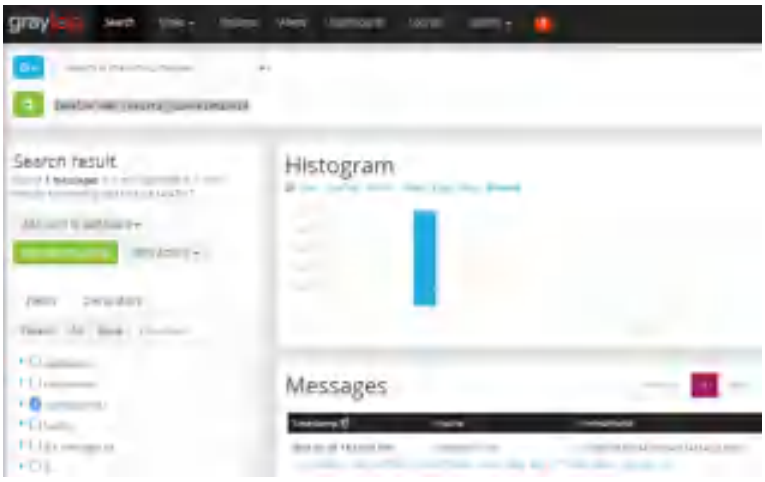
geral para obtenção dos dados relacionados ao bairro, da busca dos dados do servidor na carteira funcional, da criação do boletim de ocorrência de número 305, ...

Caso tenhamos interesse em filtrar, por exemplo, mensagens de WARNING derivadas de ações do usuário, podemos executar a expressão abaixo:

level:4 AND _exists_:correlationId

Veja o resultado na figura 128.

Figura 128 – Ocorrências de Log por usuário e nível WARNING

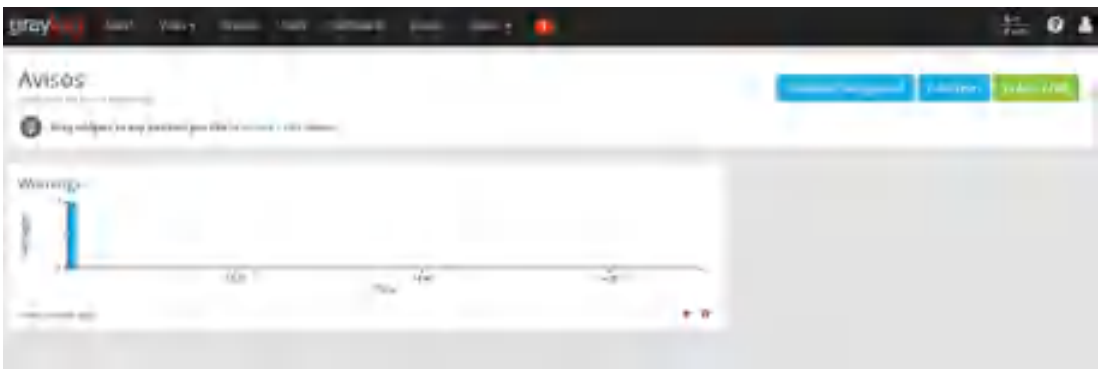


Obtivemos um único resultado que é um aviso sobre o tempo de criação do boletim de ocorrência. Perceba que tem o mesmo correlationId da pesquisa anterior, mas que não apareceu nos resultados porque filtramos apenas os de nível INFO.

Perceba que é possível adicionar o Histograma a um Dashboard para monitoramento (botão *Add to dashboard* no próprio gráfico).

Podemos então compor um dashboard com os gráficos de monitoramento das expressões de interesse. Neste caso ao adicionar foi criado um dashboard "Avisos" para estatísticas de Erro e Warnings. Veja figura 129.

Figura 129 – Gráficos de monitoramento



Caso não deseje visualizar mensagens de log de bibliotecas, frameworks utilizados, ou recursos de terceiros, pode-se filtrar pelo **className**, i.e. **className:br.gov.mt.sesp***.

11.4.5 Alertas e Notificações

O menu "*Alerts*" permite a criação de alertas e também de estratégias de notificação, como o envio de um e-mail por exemplo. Tendo em vista que temos em nosso ambiente o AlertManager, é mais interessante centralizar as regras de notificação por ele, assim temos um modelo único para configuração das regras, caso contrário teríamos regras de notificação do RabbitMQ, outras regras para o Grafana e outras para o Graylog.

Para que os alertas do Graylog sejam encaminhados para AlertManager, faz-se necessário a instalação do plugin Graylog-Plugin-AlertManager-Callback[9].

11.4.5.1 Criando um Alerta

Para criar um alerta acesse o menu **Alerts** e depois clique no botão **Event Definition**s presente no canto superior direito da página e adicione/crei um novo evento.

Vamos criar um evento simples em que alertas são emitidos quando do recebimento de mensagens de log de nível Warning (**level: 4**), veja a ilustração na figura 130.

Figura 130 – Configuração de Alerts

Conforme imagem acima no primeiro passo definimos:

- **Title**: WarningEvents
- **Description**: Eventos de Warning
- **Priority**: Normal

Caso fosse um alerta de erro poderíamos utilizar prioridade *High*, por exemplo.

O próximo passo envolve descrever a expressão utilizada para filtro, portanto, conforme figura 131.

- **Condition Type**: Filter & Aggregation
- **Search Query**: level:4
- **Search within the last**: 1 minute
- **Execute search every**: 5 seconds

[9] https://github.com/GDATASoftwareAG/Graylog-Plugin-AlertManager-Callback

Figura 131 – Parâmetros de Configuração de Alerts

- **Enable**: true
- **Create Events for Definition if...**: Filter has results

Definimos que a busca deve ocorrer a cada 5 segundos apenas para verificar o alerta ocorrendo o mais rápido possível, mas esse é um valor muito baixo e deve ser evitado.

Em seguida vamos avançar até o passo **Notifications** e adicionar uma notificação (**Add Notification**). A notificação é enviada sempre que o alerta for disparado. Vamos configurar conforme figura 132:

Podemos verificar que o plugin instalado foi carregado porque apareceu a opção **Legacy AlertManager Callback** em **Choose Legacy Notification**. Perceba que a notificação é enviada para o AlertManager na porta 9093. Veja na ilustração da figura 133.

Adicionada a notificação podemos avançar até o final e finalizar (**Done**). Para teste basta realizar o cadastro de um boletim de ocorrência e veremos que os alertas serão disparados, podendo verificar o resultado tanto na página Alerts do Graylog, ou na página do AlertManager ou no gerenciador de e-mails do MailHog. Os resultados podem ser observados nas figuras 134, 135 e 136.

Para maiores informações sobre criação de alertas recomendamos a leitura de https://docs.graylog.org/en/3.1/pages/streams/alerts.html.

Figura 132 – Configuração de Notifications

Figura 133 – Choose Legacy Notification

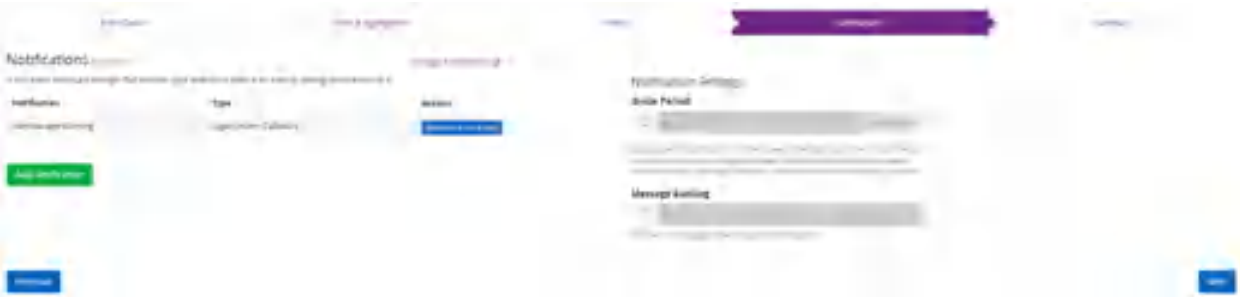


Figura 134 – Resultado do alerta no Graylog

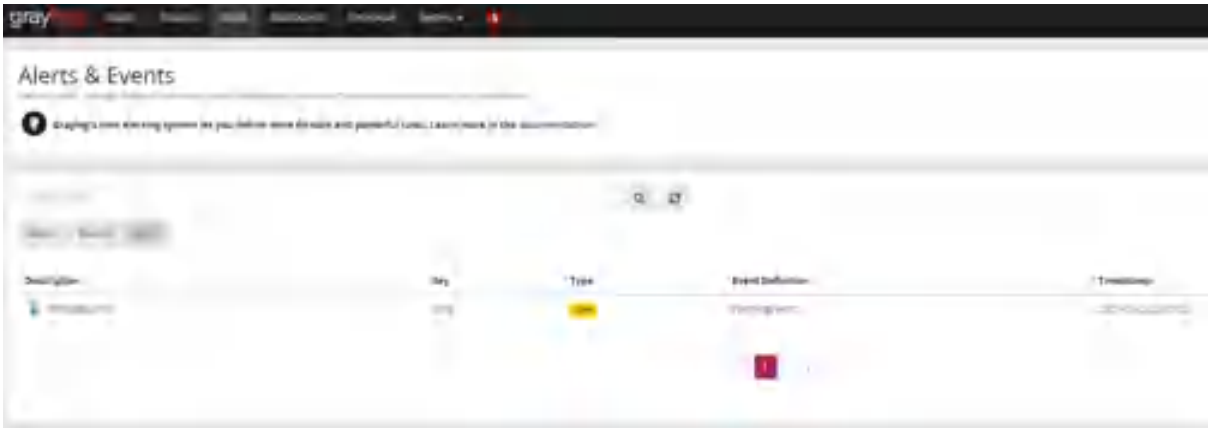


Figura 135 – Resultado do alerta do Graylog recebido no AlertManager (“Warning Log Alert”)

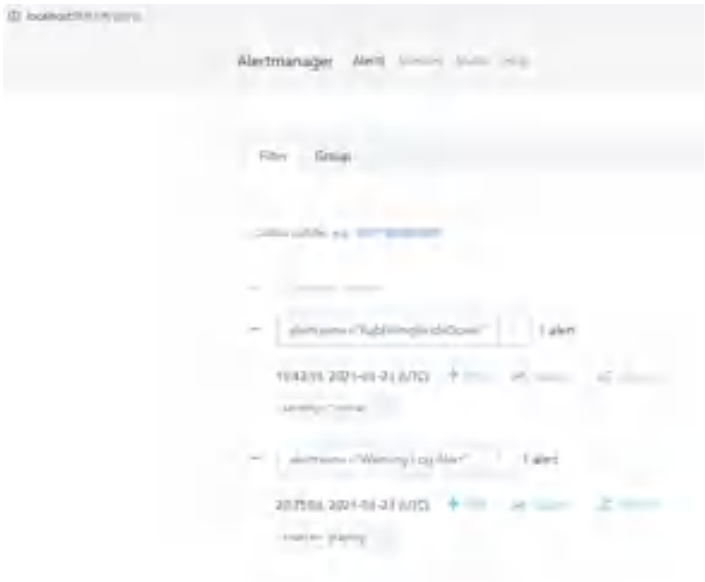


Figura 136 – Resultado do alerta enviado por e-mail

CAPÍTULO 12

Documentação Funcional

Encerramos com um capítulo que apresenta como expor de forma mais amigável a documentação dos serviços disponibilizados pelos microsserviços.

12.1 Introdução

A elaboração da documentação de um serviço é uma tarefa essencial que contribui para a melhor compreensão do que é desenvolvido, principalmente quando se trata de uma equipe composta por vários membros. Além disso, facilita em futuras evoluções minimizando riscos de retrabalho uma vez que o que existe está bem descrito.

Existem diversas soluções no mercado para facilitar a criação de documentação funcional. Para este projeto, optou-se por utilizar o Swagger, uma ferramenta que utiliza a especificação OpenAPI e fornece uma interface amigável e rica para descrever o serviço desenvolvido e seus endpoints.

Configurar manualmente o **Swagger** pode ser trabalhoso e, por isso, existem ferramentas que podem agilizar este processo. Em aplicações Spring Boot recomenda-se a biblioteca **SpringFox**.

12.2 Configuração do SpringFox

Para realizar a integração do Swagger com o microsserviço, primeiramente deve-se adicionar ao arquivo pom.xml a dependência para utilização do SpringFox.

```
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-boot-starter</artifactId>
    <version>3.0.0</version>
</dependency>
```

Código 12.1 – Dependência no pom.xml

Em seguida, deve-se adicionar a anotação **@EnableSwagger2** na classe principal da aplicação.

```
...
@EnableSpringDataWebSupport
```

```
@EnableSwagger2
@SpringBootApplication(exclude = {
    SecurityAutoConfiguration.class,
    UserDetailsServiceAutoConfiguration.class
})
public class GerenciamentoOcorrenciaApplication {
...
```

Código 12.2 – Anotação da Classe Principal

Com isso o Swagger estará ativado.

12.3 Documentação dos serviços

Para definir os *endpoints* que serão expostos na documentação funcional é necessário configurar um objeto **Docket**.

12.3.1 Configuração do Docket

Para isso, foi adicionado o conteúdo que segue na classe **SwaggerConfiguration**.

```
1  @Configuration
2  public class SwaggerConfiguration {
3
4      @Bean
5      public Docket servicoOcorrencia(){
6          return new Docket(DocumentationType.SWAGGER_2)
7                  .select()
8                  .apis(RequestHandlerSelectors.basePackage("br.gov.mt.sesp.
9                  gerenciamentoocorrencia"))
10                 .paths(PathSelectors.ant("/**"))
11                 .build()
12                 .apiInfo(apiInfo());
13
14     }
15
16     private ApiInfo apiInfo() {
17         return new ApiInfoBuilder().title("Gerenciamento Ocorrencia")
18                 .description("Documentacao Funcional do microsservico
19                 de Gerenciamento de Ocorrrencia").termsOfServiceUrl("")
20                 .contact(new Contact("SESP-MT", "", "contato@email.com"))
21                 .version("0.9")
22                 .build();
23     }
24 }
```

Código 12.3 – Configuração do Docket

A partir do código acima é possível perceber:

- **Linha 8**: No método **apis()** é informado o pacote da aplicação a partir do qual o Swagger buscará as classes que exponham endpoints.
- **Linha 10**: No método **paths()** é definido que essa busca deve ocorrer em todos os caminhos no interior do pacote apontado.

- **Linha 16**: No método **apiInfo()** são passadas as informações úteis acerca da aplicação.

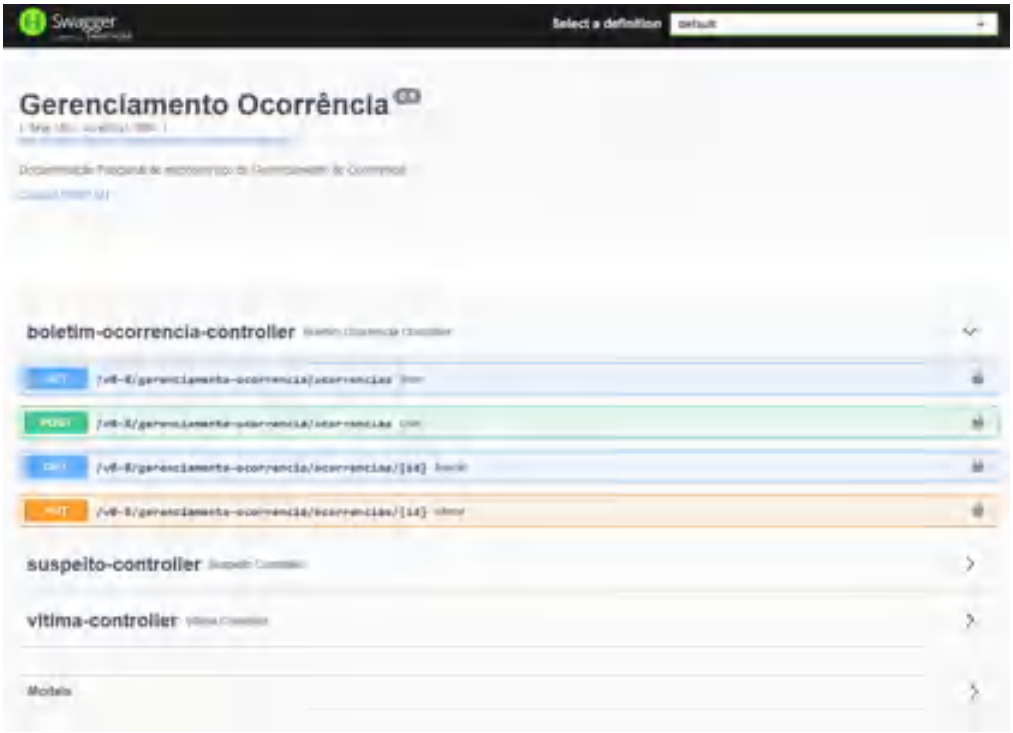
Com isso, o Swagger já estará disponível pelo contexto **/swagger-ui**.

12.3.2 Acessando a documentação

Por padrão, o Swagger estará disponível pelo contexto /swagger-ui. Para o microsserviço Gerenciamento Ocorrência, o endereço é: http://localhost:8002/v1-0/gerenciamento-ocorrencia/swagger-ui/

Na figura 137 é exibida a interface do Swagger no microsserviço Gerenciamento Ocorrência.

Figura 137 – Interface do Swagger no microsserviço Gerenciamento Ocorrência



Como pode ser observado, o Swagger gera automaticamente a descrição de todos os endpoints definidos nas classes dos microsserviços contendo o método HTTP, o endereço, o nome do método e em qual classe ele está contido.

Ao clicar no endpoint desejado serão exibidos os detalhes, incluindo os parâmetros que ele recebe e quais suas possíveis respostas, como pode ser visto nas figuras 138 e 139.

12.3.3 Personalização das informações

É possível personalizar as informações apresentadas, como por exemplo a mensagem de resposta. Para isso é necessário utilizar as anotações **@ApiResponses** e **@ApiResponse** sobre o método correspondente, como pode ser visto no exemplo a seguir.

```
1 @PostMapping(consumes = "application/json", produces = "application/json")
2 @ApiResponses(value = {
3     @ApiResponse(code = 201, message = "Cria um novo boletim de ocorrencia"),
4     @ApiResponse(code = 401, message = "Voce nao tem autorizacao para acessar este
          recurso"),
```

Figura 138 – Exemplo de documentação do endpoint para criação de boletim de ocorrência

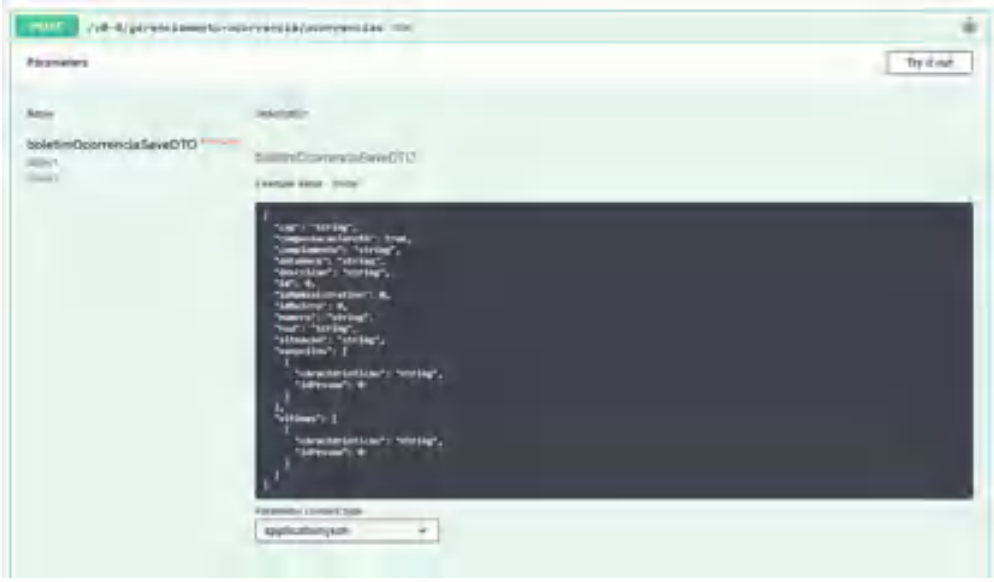


Figura 139 – Exemplo da documentação das respostas possíveis para um endpoint

```
5       @ApiResponse(code = 403, message = "Voce nao tem permissao para acessar este
            recurso"),
6       @ApiResponse(code = 500, message = "Foi gerada uma excecao"),
7   })
8   public ResponseEntity<BoletimOcorrenciaDTO> criar(
```

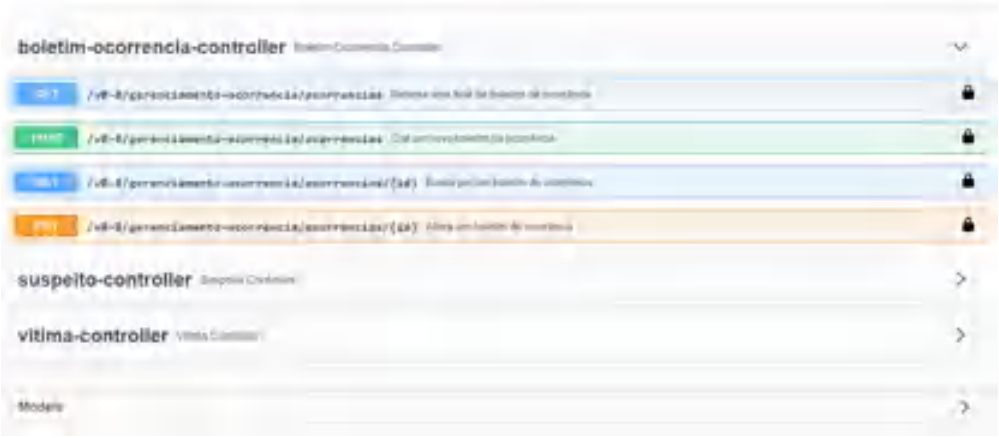
Código 12.4 – Personalização das informações

É possível também informar qual tipo de conteúdo o endpoint consome e produz adicionando os atributos **consumes** e **produces**, respectivamente, como observado no exemplo anterior.

Finalmente, ainda é possivel descrever a operação que o endpoint realiza adicionando a anotação **@ApiOperations**

Assim, o Swagger exibe mensagens mais descritivas e relevantes para a documentação, como visto nas figuras 140 e 141.

Figura 140 – Exemplo de informações personalizadas para os endpoints



12.3.4 Executando endpoints a partir da documentação funcional

Outro recurso do Swagger é a possibilidade de testar os endpoints pela própria interface da documentação. Esta funcionalidade pode ser acessada pelo botão **Try it out**, localizado logo abaixo do título de cada endpoint. No painel exibido será necessário preencher os parâmetros correspondente e clicar em Execute.

A figura 142 exibe o retorno obtido de uma consulta por um boletim de ocorrência.

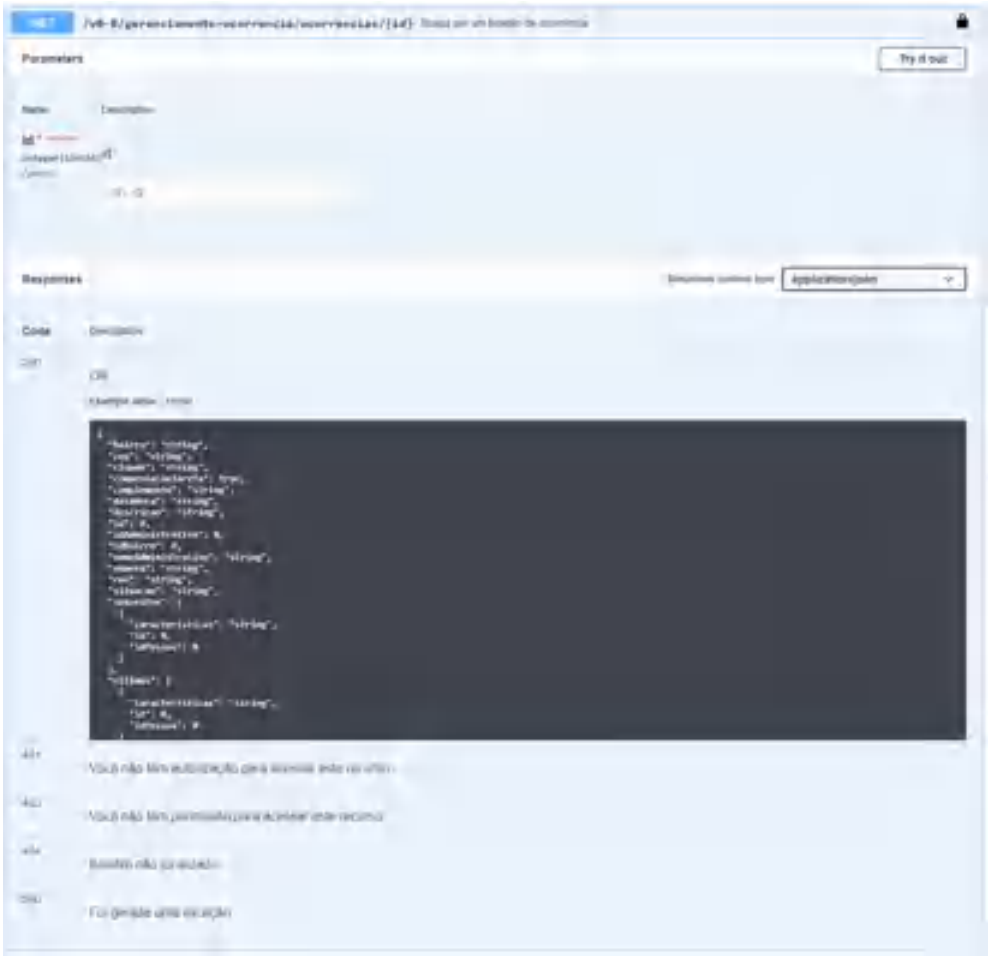
Como pode ser observado, ao ser passado o **id** do boletim de ocorrência que se deseja buscar, é exibido o código da resposta da requisição e o resultado *Json*.

12.3.5 Configurando autenticação para endpoints protegidos

Para endpoints protegidos por autenticação é necessário realizar uma configuração adicional. São utilizados os métodos **securitySchemes**, onde é definido a forma de autenticação, e **securityContexts**, onde pode-se especificar endpoints protegidos.

A seguir é exibido o conteúdo da classe **SwaggerConfiguration** com as configurações de autenticação.

Figura 141 – Exemplo de informações personalizadas para as respostas do endpoint



```
@Configuration
public class SwaggerConfiguration {

    @Bean
    public Docket servicoOcorrencia(){
        return new Docket(DocumentationType.SWAGGER_2)
                .select()
                .apis(RequestHandlerSelectors.basePackage("br.gov.mt.sesp.
                gerenciamentoocorrencia"))
                .paths(PathSelectors.ant("/**"))
                .build()
                .apiInfo(apiInfo())
                .securitySchemes(Arrays.asList(new ApiKey("Token Access",
                HttpHeaders.AUTHORIZATION, In.HEADER.name())))
                .securityContexts(Arrays.asList(securityContext()));

    }

    private ApiInfo apiInfo() {
        return new ApiInfoBuilder().title("Gerenciamento Ocorrencia")
                .description("Documentacao Funcional do microsservico de
                Gerenciamento de Ocorrrencia").termsOfServiceUrl("")
                .contact(new Contact("SESP-MT", "", "sesp@email.com"))
                .version("0.9")
                .build();
    }
```

Figura 142 – Interface para execução de um endpoint a partir da página da documentação funcional

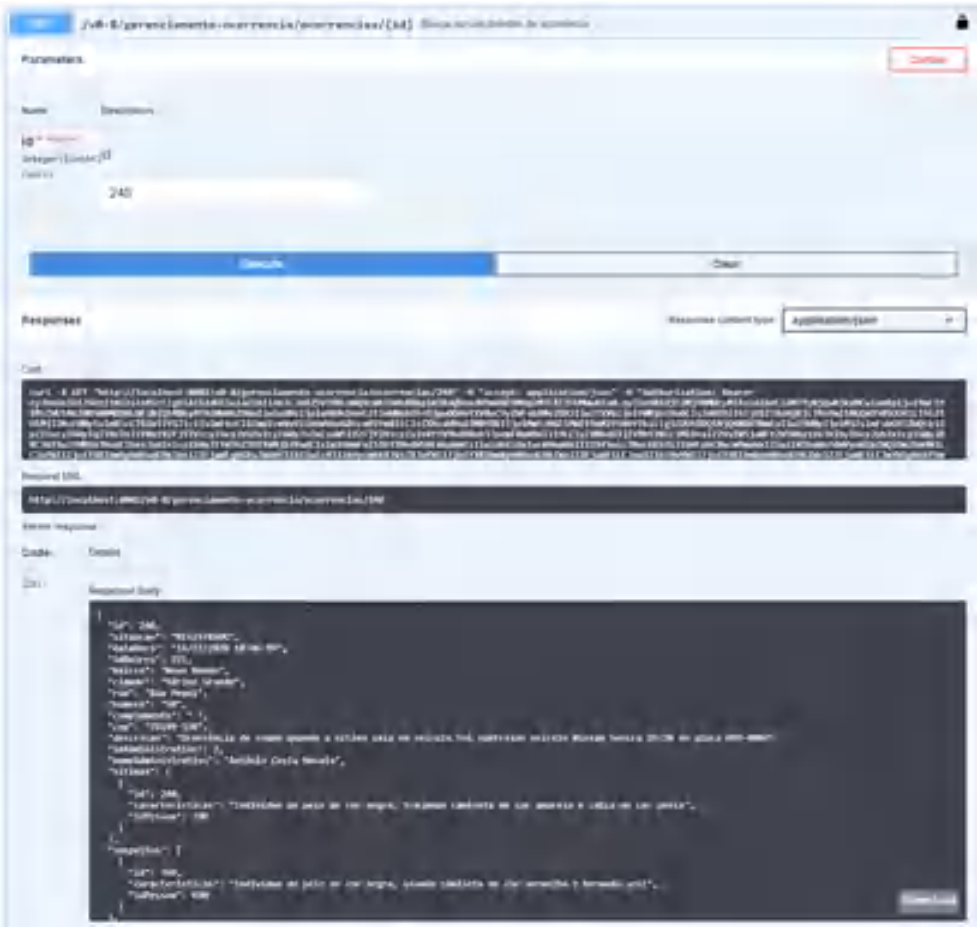


Tabela 35 – Recursos Acessíveis pelos endpoints

ID	Informação
auditevents	Dados de auditoria
beans	Beans do Spring no aplicação
caches	Caches disponíveis.
health	Informações gerais das condições de aplicação.
info	Dados arbitrários, como nome, descrição, versão do Java.
conditions	Verifica se as condições definidas nas classes de configuração foram ou não atingidas.
configprops	Exibe uma lista de @ConfigurationProperties
env	Expõe propriedades de ConfigurableEnvironment
loggers	Exibe as configurações de Log de aplicação
heapdump	Exibe as informações dos objetos em memória na JVM
threaddump	Exibe os estados de todas as threads em execução
metrics	Exibe as métricas de aplicação
scheduledtasks	Retornar as tarefas agendadas.
mappings	Retorna uma lista de todos os endpoints @RequestMapping

```
27
28      private SecurityContext securityContext() {
29          return SecurityContext.builder()
30                  .securityReferences(defaultAuth())
31                  .build();
32      }
33
34      List<SecurityReference> defaultAuth() {
35          AuthorizationScope authorizationScope
36                  = new AuthorizationScope("global", "accessEverything");
37          AuthorizationScope[] authorizationScopes = new AuthorizationScope[1];
38          authorizationScopes[0] = authorizationScope;
39          return Arrays.asList(
40                  new SecurityReference("Token Access", authorizationScopes));
41      }
42  }
```

Código 12.5 – Classe SwaggerConfiguration com as configurações de autenticação

A partir do código anterior pode-se perceber:

- Linha 13: É especificado o modo ApiKey, que espera um token JWT para ser inserido no cabeçalho da requisição realizada pelo Swagger
- Linha 35: A autenticação é configurada em escopo global.

Com isso o botão **Authorize** será exibido na página da documentação, permitindo a autenticação, conforme apresentado na figura 143.

Figura 143 – Interface da documentação funcional após configurações de segurança

Clicando no botão Authorize, será exibida uma janela de diálogo, ilustrada na figura 144, para inserção do Token JWT.

Para mais informações acerca da integração do Swagger com aplicações Spring Boot, recomenda-se:

- Springfox Reference Documentation[1]

[1] https://springfox.github.io/springfox/docs/current/#configuring-springfox

Figura 144 – Interface da janela de diálogo exibida ao clicar sobre o botão “Authorize”

- Setting Up Swagger 2 with a Spring REST API — Baeldung[2]

[2] https://www.baeldung.com/swagger-2-documentation-for-spring-rest-api