Subsecretaria de Tecnologia da Informação e Comunicação

DOCUMENTO DE ARQUITETURA DE SOFTWARE – DAS

Sistema: SGC CONTROLE DE ACESSO

Versão 1.2.0

Ministério da Educação

PÁTRIA AMADA BRASIL GOVERNO FEDERAL

Sumário

DOCUMENTO DE ARQUITETURA DE SOFTWARE – DAS

Controle de Versões			
Versão	**Data**	**Autor**	**Descrição**
1.0.0	01/06/2021	Wedson Quintanilha da Silva	Criação do documento.
1.1.0	14/06/2021	Wedson Quintanilha da Silva	Conversão para o template do layout do MEC.
1.2.0	18/10/2021	Wisley Alves couto	Atualização do documento

1. INTRODUÇÃO

Este documento visa descrever a arquitetura lógica e física do projeto SGC, Sistema de Gestão Corporativa, descrevendo o processo de controle de acesso e gerenciamento de segurança que os sistemas que compõem a solução do SGC devem observar.

1.1. DEFINIÇÕES, ACRÔNIMOS E ABREVIAÇÕES

Abreviação	Descrição
OID	OpenID Connect, protocolo de autenticação.
REST	Transferência de Estado Representacional é um estilo de arquitetura que define uma série de restrições pra criação de serviços web com protocolo HTTP.
JWT	JSON Web Token é um padrão (RFC 7519) para criação de tokens de acessos.
OAuth	É um padrão aberto de delegação de acessos.
SSO	Single Sign-On, conceito de autenticação única e centralizada para várias aplicações.

Tabela **1**-Tabela de abreviações

1.2. REFERÊNCIAS

Fonte
Spring Cloud: http://cloud.spring.io/
Spring Cloud Netflix: https://cloud.spring.io/spring-cloud-netflix/
Prometheus: https://prometheus.io/docs/concepts/metric_types/
Elastic Stack: https://www.elastic.co/elk-stack
Shard Elasticsearch: https://www.elastic.co/guide/en/elasticsearch/reference/current/_basic_concepts.html
Node Elasticsearch: https://www.elastic.co/guide/en/elasticsearch/reference/current/modules- node.html
Angular: https://angular.io/
OKD: https://www.okd.io/

Tabela 2-Tabela de referências

2. OBJETIVO DO DOCUMENTO

O objetivo deste documento é apresentar ao cliente a descrição da arquitetura do projeto.

3. MACRO ARQUITETURA

3.1. Visão Geral das Camadas Arquiteturais

Na figura abaixo, apresentamos uma visão simplificada das camadas que compõem o sistema e a forma como elas se relacionam. Nas seções a seguir, detalhamos conceitualmente cada camada e enumeramos as ferramentas e plataformas que serão utilizadas para sua construção.

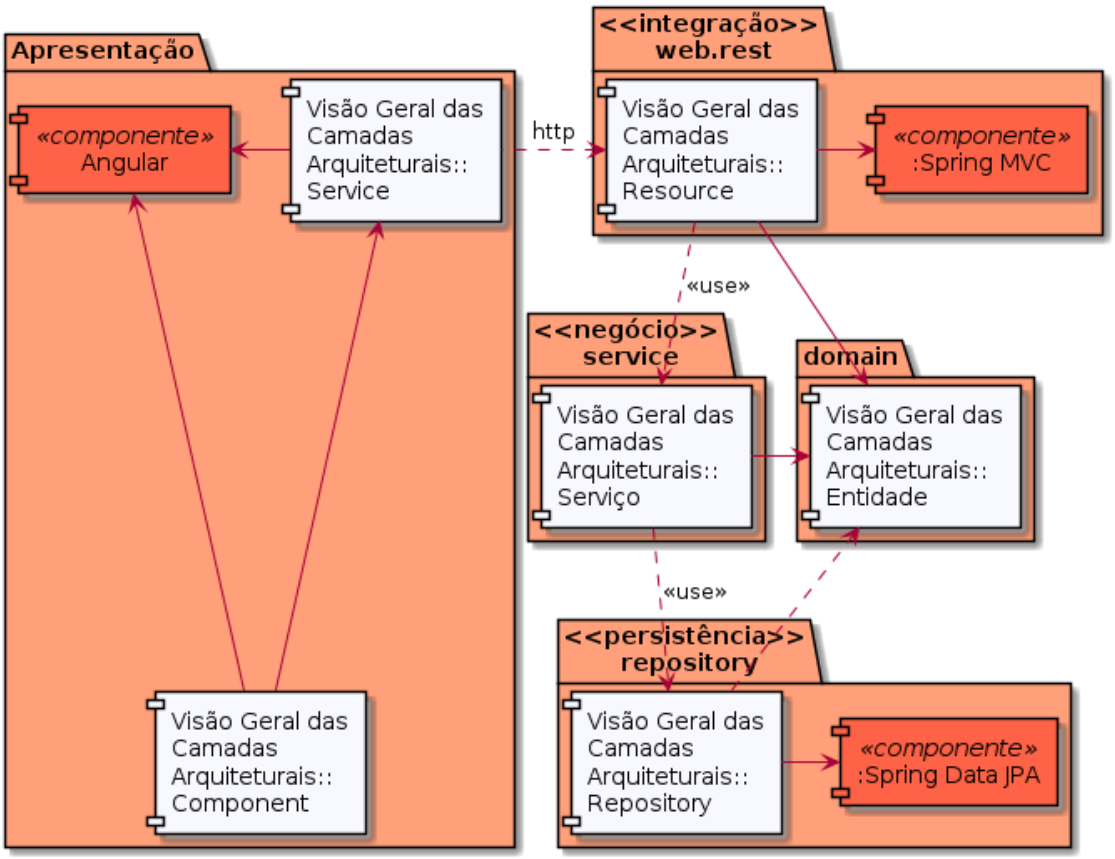

Figura 1. Visão Geral das Camadas Arquiteturais

3.1.1. Camada de Apresentação

A camada de apresentação é responsável por fazer a lógica de construção das páginas para serem exibidas pelos usuários, tratar os eventos do Browser, como cliques, e gerenciar o fluxo de execução do sistema.

3.1.2. Camada de Negócio

A camada de negócios é responsável pela implementação lógica da aplicação. Ela expõe os serviços para a camada de apresentação por meio de uma interface bem definida e obtém as informações necessárias para mostrar ao usuário por meio da Camada de Persistência.

3.1.3. Camada de Persistência

A camada de persistência é responsável pela lógica de acesso ao banco de dados e pelo mapeamento dos dados em entidades representativas. O objetivo em mapear o banco de dados em entidades representativas ao sistema é diminuir a diferença semântica entre o modelo abstrato do banco e o problema do mundo real.

Nota 1: Para armazenar os perfis de um usuário após efetuar o login, será utilizado o SGBD REDIS que receberá os dados do perfil em formato JSON (composto pela identificação do sistema que ele tem acesso e, para cada sistema, seus respectivos papéis naquele sistema específico) e os manterá armazenado para que as aplicações autorizadas para o usuário possam consultar estes perfis sempre que necessário.

Nota 2: Como esta é uma arquitetura baseada em microsserviços, para garantir a continuidade da operação caso algum microsserviço falhe ou fique fora do ar por algum motivo qualquer, o indicado é que cada microsserviço tenha o seu banco de dados exclusivo. No entanto, caso isso não seja possível, a alternativa é manter um schema por microsserviço em separado e, caso isso também não seja possível, o projeto do banco de dados deve prever que as tabelas envolvidas em um domínio de negócio atendido por um microsserviço específico não sejam relacionadas com tabelas de outro microsserviço, apenas entre tabelas do mesmo domínio negocial atendido por um único microsserviço.

3.1.4. Camada de Serviços

A camada de serviços encapsula diversos serviços que são providos às outras camadas da aplicação.

Os serviços são responsáveis por realizar as regras de negócio e **estes são organizados por domínio negocial** e serão acessados por meio de API's REST.

Os acessos aos serviços serão efetuados mediante a presença do token JWT (jwt-token) de autenticação válido, presente no cookie ou no header da requisição.

As API's presentes nesta camada serão organizados conforme padrões e princípios estabelecidos na especificação RFC7231, onde **resumidamente** temos:

- Uso apropriado dos verbos GET, PUT, POST, DELETE e PATCH conforme ações desejadas:
  - GET: Usado para recuperar um recurso na Api;
  - PUT: Usado para substituir um recurso inteiro enviado para Api;
  - POST: Utilizado para inserir um novo recurso enviado para a Api;
  - PATCH: utilizado para modificar partes de um recurso existente na Api;
  - DELETE: utilizado para apagar um recurso na Api.
- Uso correto do Status de respostas (código de status) conforme as operações realizadas, observando e diferenciando erros de negócio de erros de sistema;
- Definição dos URI's (endpoints) de forma adequada, observando os relacionamentos entre os conceitos de negócio adequadamente;
- Versionamento correto adotando os prefixos para o caso de haver a necessidade de alterações nos comportamentos ou na estrutura de dados enviada ou recebida a partir de um serviço.

Exemplo:

Versão única:

- /api/perfil/5

Este mesmo recurso na versão 2 deve ser formatado com a seguinte URI:

- /api/v2/perfil/5

3.1.5. Requisitos para Implementação das Camadas

Os itens apresentados a seguir referem-se à lista de tecnologias e requisitos necessários para suportar as camadas de apresentação, negócio e persistência:

- Java 11
- Maven 3
- Docker
- Docker-compose
- NodeJS
- Undertow

- TypeScript
- Kubernetes
- Istio
- Git
- Helm
- Chart

Bibliotecas a serem utilizadas pela aplicação:

- Spring Cloud Hoxton
- Spring Security
- Spring Data Redis
- Angular 11 ou ReactJs
- Dsgov
- Liquibase
- Nexus

3.2. Visão Geral do processo

O diagrama de sequência abaixo apresenta o fluxo de informações do sistema e suas interfaces através das camadas arquiteturais.

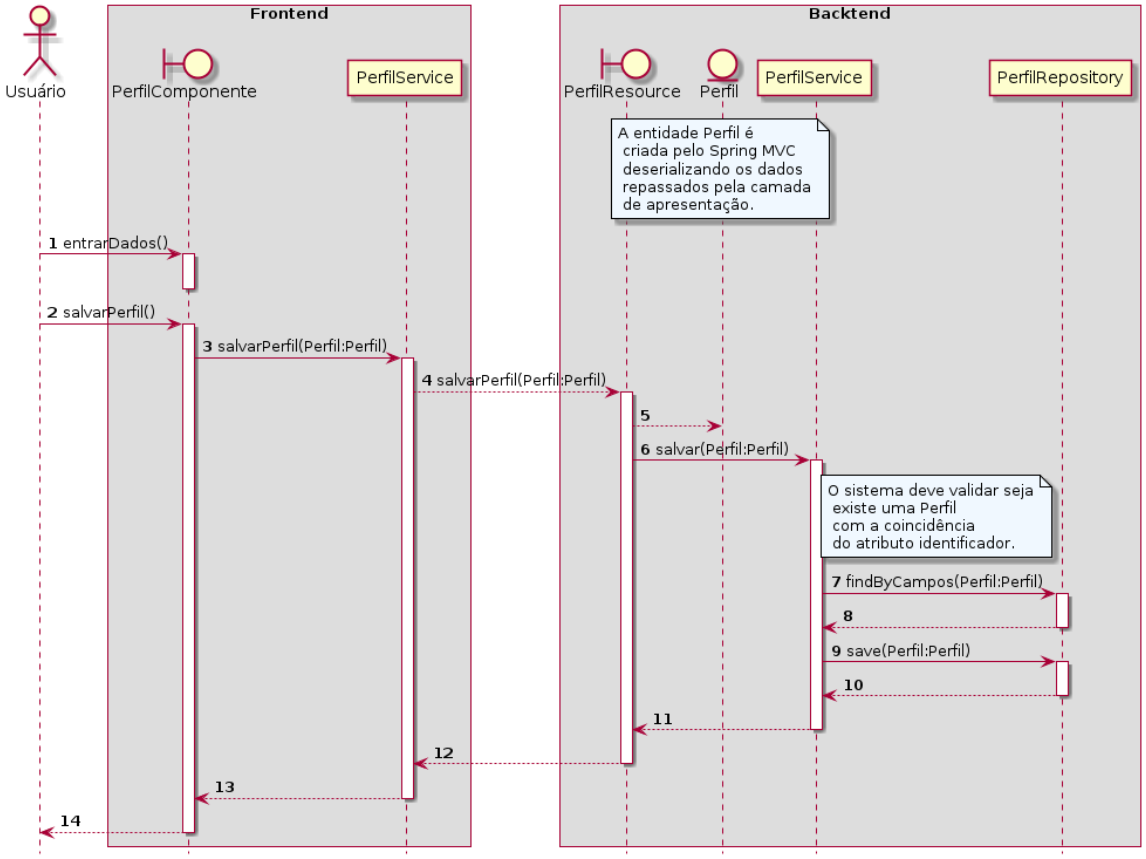

Figura 2. Visões da Arquitetura

3.3. Visão dos Módulos do Sistema

Neste item apresentamos uma visão geral dos módulos do projeto, ilustrando-os graficamente através de um diagrama. Em seguida, detalhamos as responsabilidades de cada módulo, em concordância com a especificação funcional.

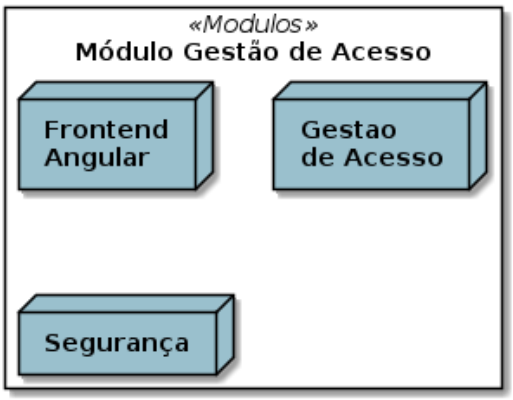

Figura 3. Visão Geral dos módulos

Módulo	Descrição
Front-End	Este módulo contém a aplicação client web, escrita em Angular ou React ficando a cargo da provedora da solução qual framework utilizar. Ele é responsável por fornecer as interfaces web para que os usuários interajam com os serviços executados pela aplicação.
Gestão de Acesso	Módulo responsável por realizar as regras de negócio para definir os módulos, usuários e seus respectivos papéis dentro de cada módulo da solução SGC.
Segurança	Este módulo é responsável por realizar as ações de autenticação para as aplicações. Ele tem todo o fluxo de comunicação com o SSO, geração dos perfis, criação do jwt-token, cookies de segurança dentre outras ações e, assim, abstrair todo este processo das demais aplicações.

3.4. Visão de Integração

A visão de integração apresenta as integrações pertinentes ao sistema. Essas integrações podem ser internas ou externas e é de suma importância que os projetistas e desenvolvedores conheçam em detalhes essas integrações, bem como os contratos e protocolos de comunicação entre os sistemas/componentes envolvidos na integração.

O Sistema deve se integrar aos seguintes sistemas a fim de atender os requisitos funcionais e não-funcionais da aplicação:

- Registro de sistemas - Istio Pilot;
- Acesso.gov (SSO do Governo Federal);
- SPO - Programação Financeira;
- SPO - Emendas Parlamentares;
- SPO - Planejamento Orçamentário;
- SPO - AJ / Ações Judiciais;
- PPA - Monitoramento e Avaliação;

- SPO - TED.

Nota: Inicialmente, os módulos mapeados estão descritos acima, no entanto, a estes módulos podem ser acrescentados novos módulos que serão apenas cadastrados (identificação, url de acesso, ícone, etc.) e disponibilizados para os perfis dos usuários, sem que haja a necessidade de alterações de programas e códigos.

3.5. Visão de Autenticação e Segurança

3.5.1. Acesso.gov

O Acesso.gov é a solução de SSO do governo federal para acesso aos serviços digitais públicos e oferece integração por meio do OpenID.

Esta solução prevê o uso do Acesso.gov como provedor de autenticação SSO para as aplicações.

Abaixo, o exemplo de um fluxo padrão via OpenID:

- O usuário precisa acessar sua conta dentro do sistema;
- O sistema solicita ao usuário o OpenID;
- O Usuário entra com o OpenID;
- O sistema redireciona o usuário para o provider do OpenID (nesse caso o Brasil Cidadão);
- O usuário autentica no provedor do OpenID;
- O provedor do OpenID redireciona o usuário de volta ao sistema;
- O sistema verifica e confirma a autorização do usuário.

OpenID utiliza o chamado id_token, que é um token de segurança que permite ao cliente verificar a identidade do usuário e obter as informações básicas do seu perfil.

3.5.2. Visão de Autenticação e Segurança

O diagrama de Sequência de Sistemas abaixo representa a comunicação entre os sistemas envolvidos para prover a segurança no cenário de acesso a uma aplicação (aqui representada por App1), **sem o token de autenticação**:

Nota: *As urls citadas nos diagramas a seguir são apenas exemplos. O MEC deve usar as URLS definidas no seu registro de domínio e autorizadas no Ministério da Economia (ME) para utilização do Acesso.Gov. Cada sistema que compor o SGC deve ter o seu cadastro no ME contendo o seu identificador (clientid) e sua chave (client-secred) para utilização do Acesso.Gov, sendo assim, cada aplicação terá a instância do módulo segurança, alterando os valores destas chaves nos arquivos de configuração (configMap).*

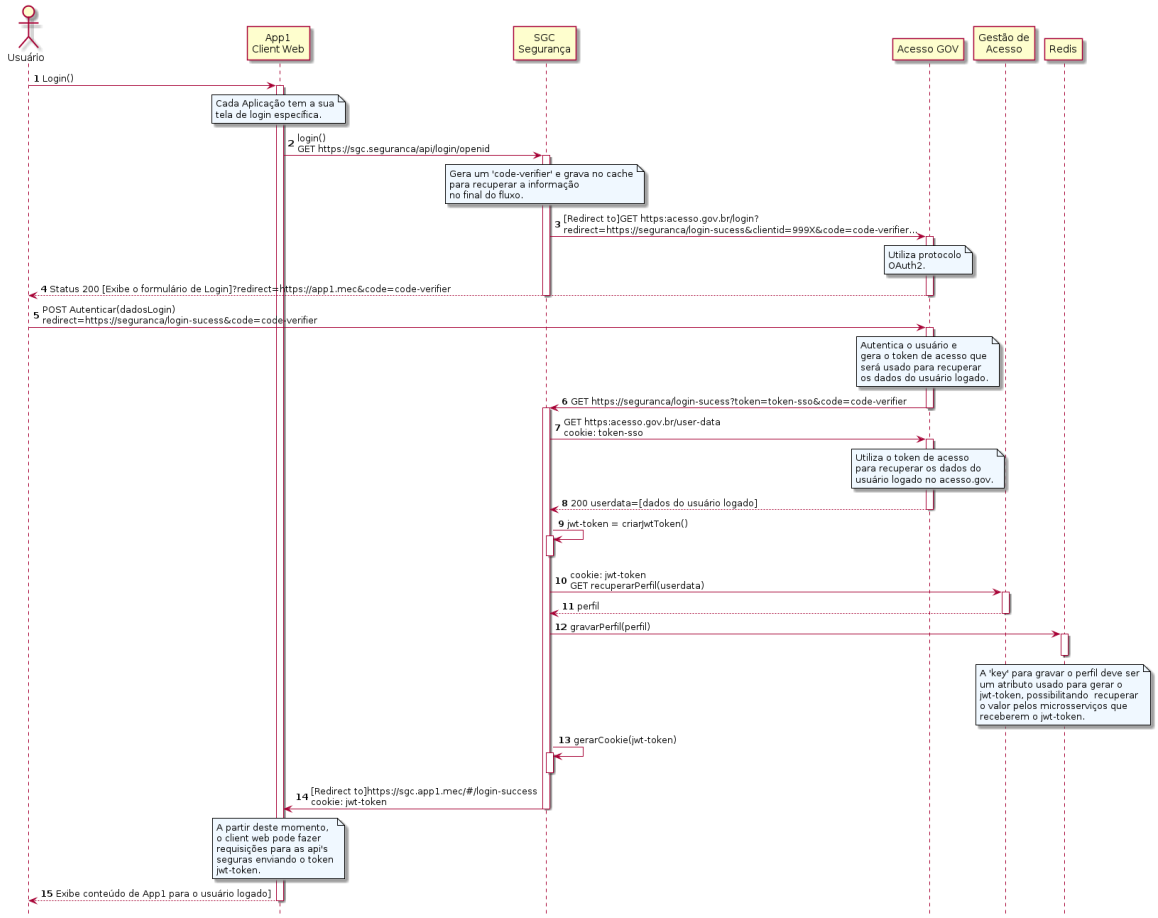

Figura 4. Acesso não autenticado

O diagrama de Sequência de Sistemas abaixo representa a comunicação entre os sistemas envolvidos no cenário de acesso de um **usuário já autenticado**:

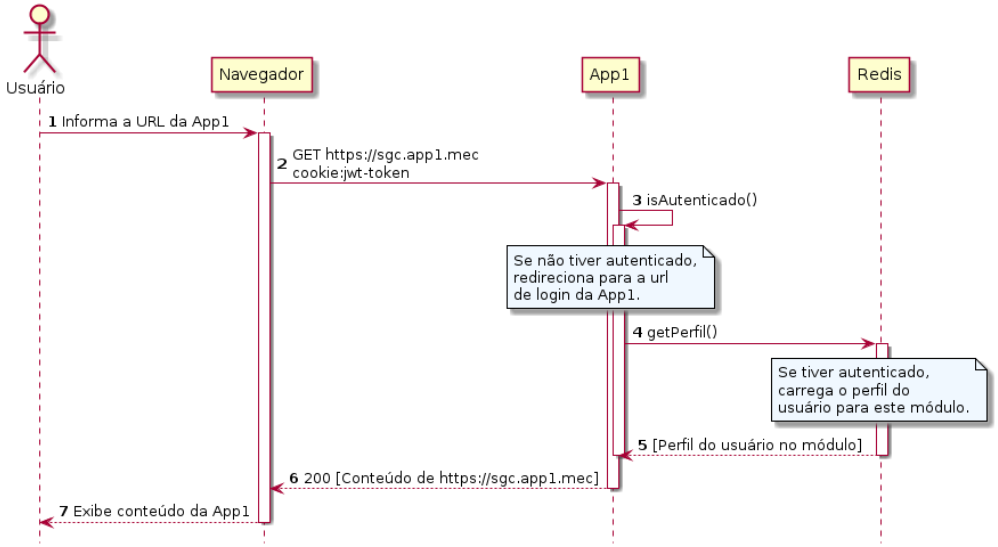

Figura 5. Acesso de um usuário autenticado

O diagrama de Sequência de Sistemas abaixo representa a comunicação entre os sistemas envolvidos no cenário de **acesso a uma API** de algum microsserviço qualquer:

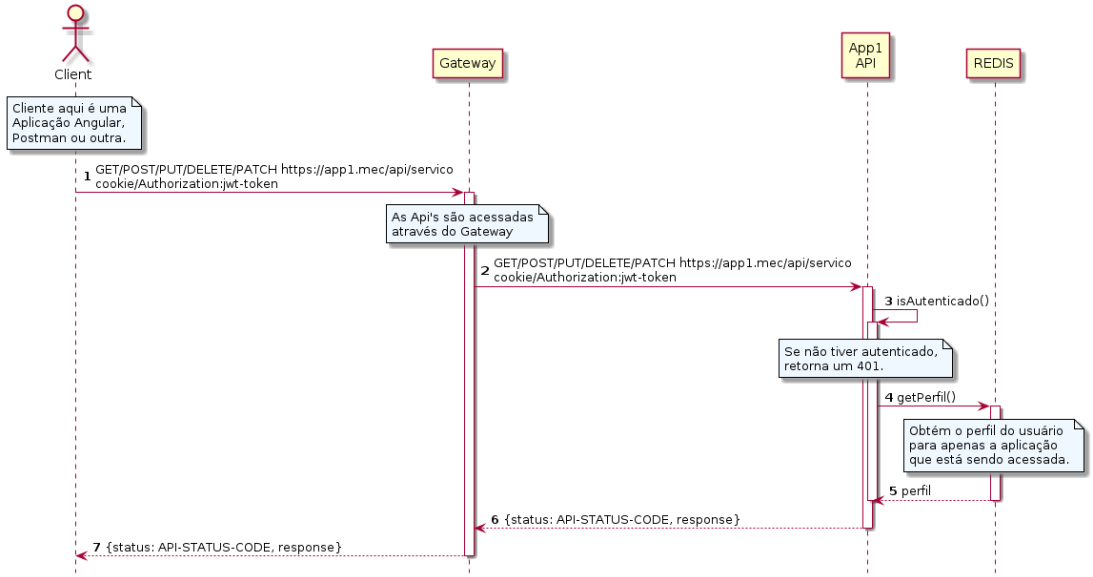

Figura 6. Acesso a uma API

O diagrama de Sequência de Sistemas abaixo demonstra o **acesso de um usuário autenticado à url principal (home) do módulo de Controle de Acesso**, onde será apresentado ao usuário um **painel contendo links para acesso aos sistemas** que ele tem permissão de acessar:

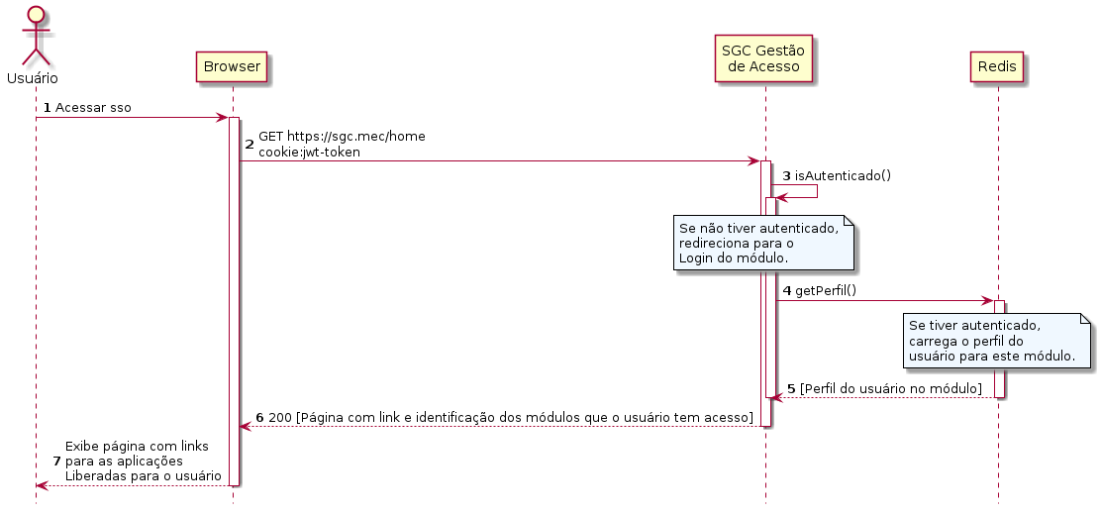

Figura 7. Acesso à Home do Controle de Acesso

Nota: Observe que cada aplicação que se integrará com este modelo de segurança deve obter o perfil do usuário autenticado no REDIS, informando quem é o usuário e qual é a aplicação visto que apenas os perfis da aplicação corrente serão retornados.

3.6. Visão de Classes

Nesta seção, estão descritos os diagramas/fluxogramas de classe, sequência e atividades para os requisitos críticos do sistema:

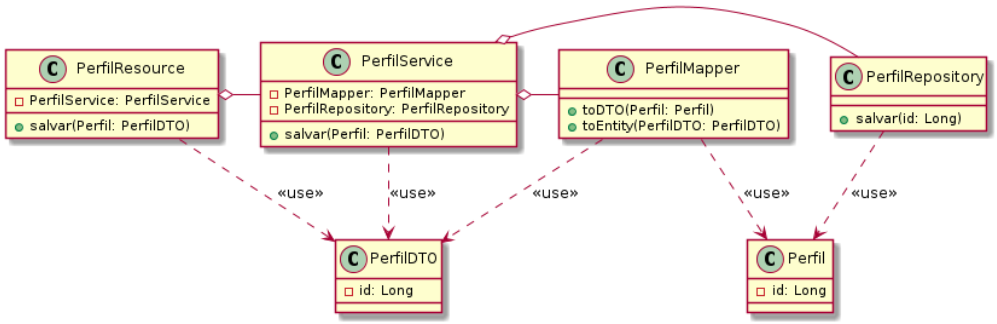

Figura 8. Visão de Classes

3.7. Ambiente de Implantação

3.7.1. Ambiente de Implantação

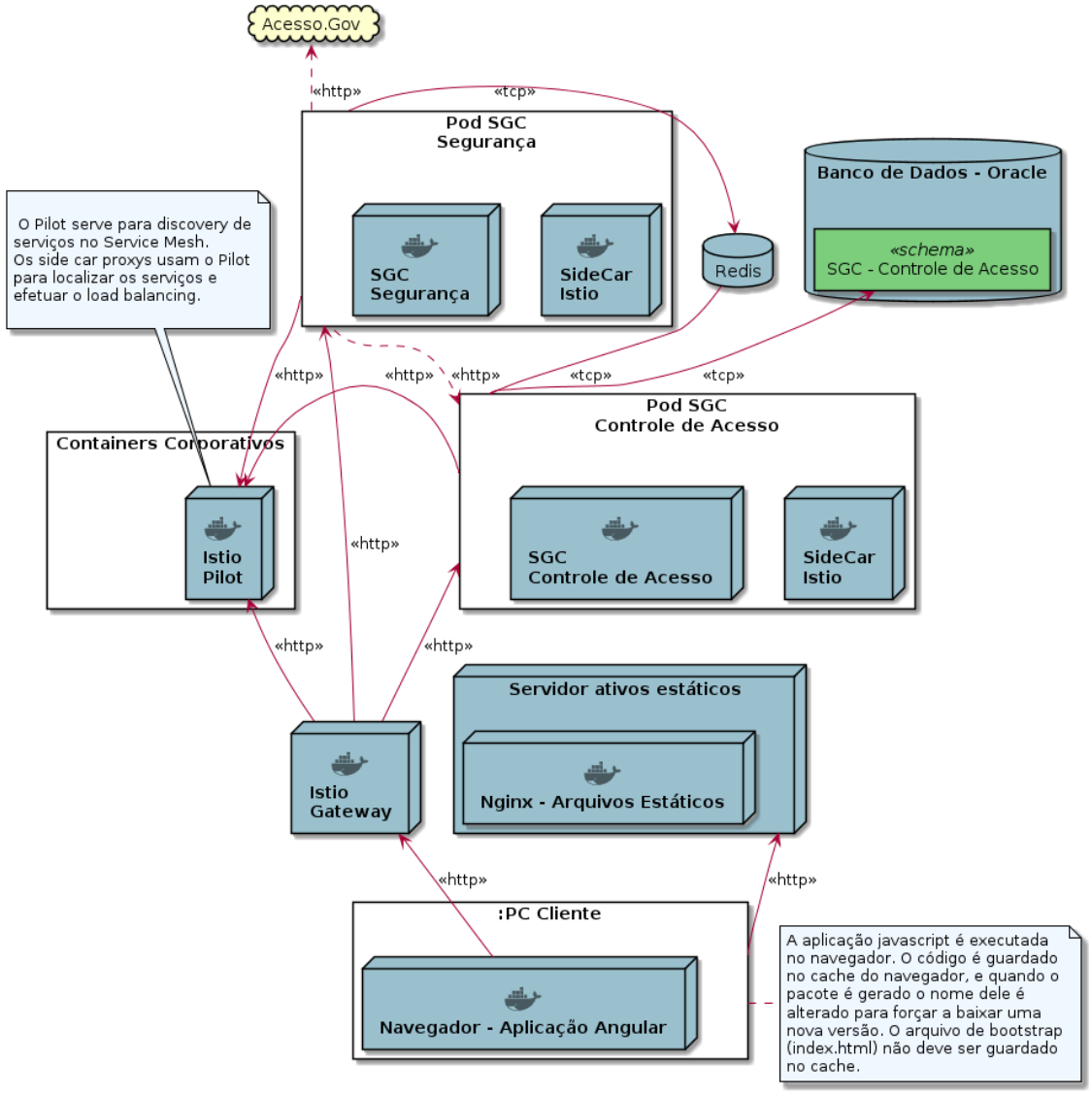

Figura 9. Diagrama de Implantação

3.7.2. Componentes principais

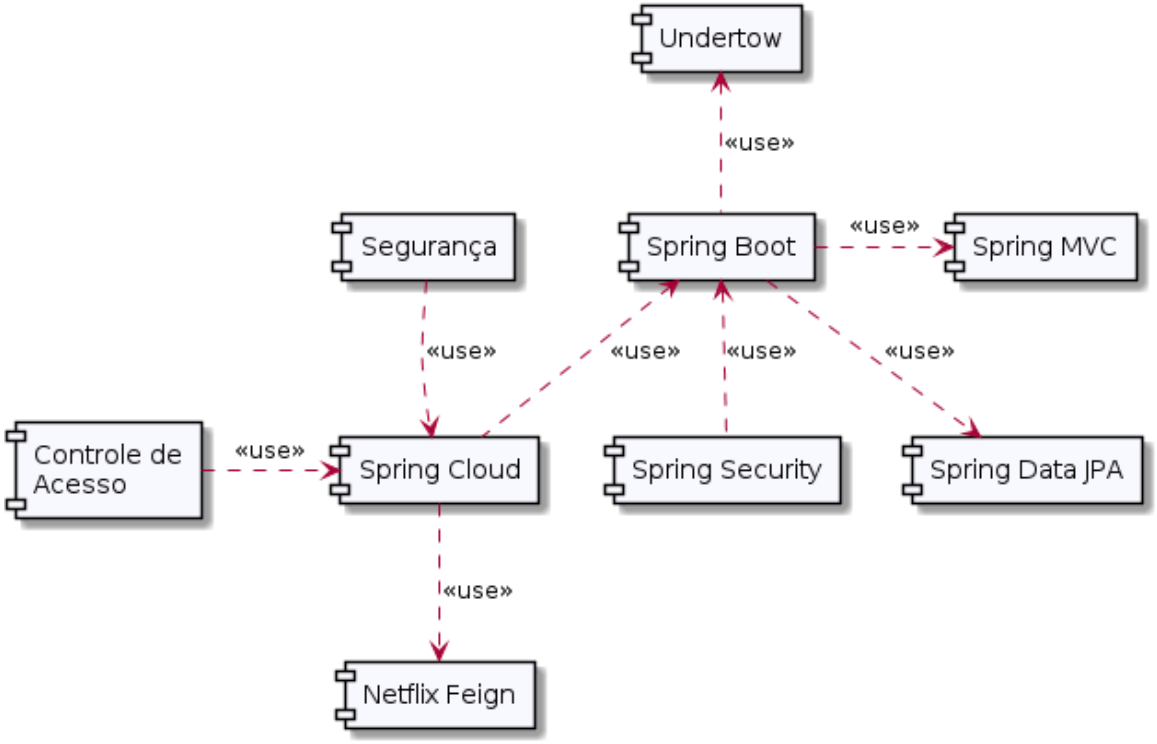

Figura 10. Visão Geral dos Componentes

4. MICROSSERVIÇOS

O termo "Arquitetura de Microsserviços" surgiu nos últimos anos para descrever uma maneira particular de projetar aplicações de software, ou seja, um conjunto de programas que pode ser implantado como serviços independentes. Embora não exista uma definição precisa desse estilo de arquitetura, existem certas características comuns em torno da organização e da capacidade do negócio, como: necessidade de implantação automatizada, inteligência nos serviços, descentralização das tecnologias, linguagens e dados.

O conceito de arquitetura de microsserviços vem gradualmente encontrando o seu espaço no desenvolvimento de software, como um sucessor da arquitetura baseada em serviços (SOA - Service Oriented Architecture), os microsserviços podem ser categorizados como sistemas distribuídos e usam muitos conceitos e práticas do SOA. Eles se diferem, entretanto, no escopo da responsabilidade dada para cada serviço individualmente. No SOA, um serviço pode ser responsável por tratar diversas funcionalidades e domínios, enquanto que uma regra geral para um micro serviço é que ele seja responsável por gerenciar um único domínio e as funcionalidades que manipulam esse domínio.

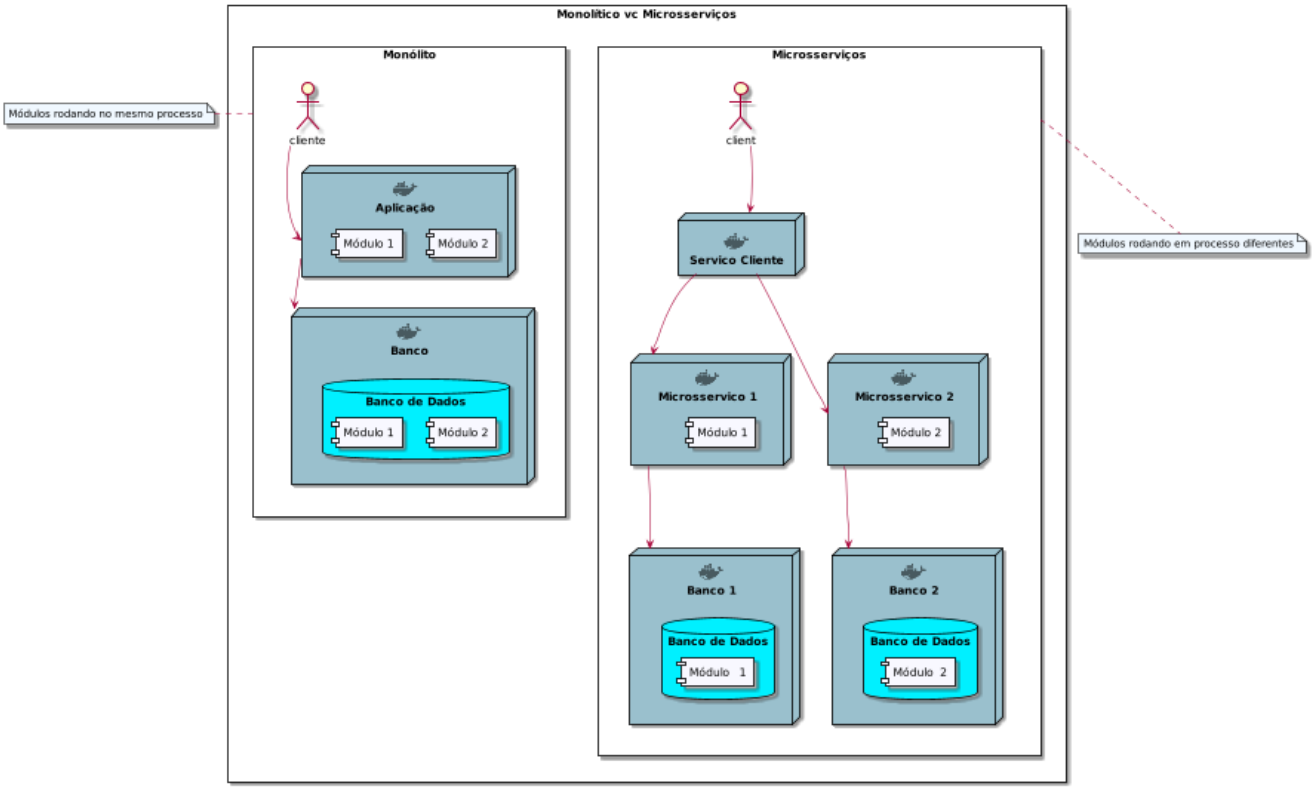

Figura 11. Microsserviços

Uma abordagem de sistemas distribuídos (Microsservicos) consiste em decompor a infraestrutura monolítica de um serviço em subsistemas escaláveis, organizados através de um corte vertical

envolvendo cada uma das camadas interconectadas do sistema por uma camada de transporte comum, visando decompor os componentes de um sistema monolítico em unidades individuais de distribuição, capazes de evoluir com as suas próprias exigências de escalabilidade independente de outros subsistemas. Isso significa que o impacto de um recurso no sistema como um todo pode ser gerenciado de forma mais eficiente e a conexão entre componentes pode compartilhar um contrato menos rígido, pois a dependência não é mais gerenciada através de apenas um ambiente de execução.

Em suma, o estilo arquitetural de um microsserviço consiste em uma abordagem para desenvolver uma única aplicação como um conjunto de pequenos serviços, podendo até usar bibliotecas, mas sua principal forma de criar componentes é dividir-se em serviços.

Um dos principais motivos para usar serviços como componentes (em vez de bibliotecas) é que eles podem ser implementados de maneira independente, cada um executando seu próprio processo e comunicando-se com mecanismos leves (API de recurso HTTP), com base em necessidades de negócios e implantados de forma independente, utilizando software de implantação automatizados.

Algumas mudanças podem alterar interfaces de serviços compartilhados, resultando em vários impactos, mas o objetivo de uma boa arquitetura de microsserviço é minimizar esses impactos, limitando os serviços coesos, para isso serão utilizados o controle de versionamento na url dos serviços e a definição dos serviços RESTful será documentada utilizando a especificação OpenAPI.

Banco de Dados

A solução pode prever acessos às seguintes bases de dados com as respectivas versões, ficando a cargo a prestadora a escolha de uns dos SGBD abaixo, no entanto recomendamos que seja oracle ou postegresql nas versões abaixo;

- Oracle versão (12.2.0.1);
- PostgresSql versão (11.00);

O controle de versão do banco de dados será feito utilizando a ferramenta Liquibase. O Liquibase permite a implantação de novas versões do banco de dados de forma segura, rápida e agnóstica ao banco de dados. As atualizações das versões do banco de dados serão executada na inicialização de cada microsserviço, conforme exemplo abaixo.

Name	Last commit	Last update
..		
changelog	BASIS-229783 ultimas modificacoes no liquibase	5 days ago
Dockerfile	BASIS-197934 - atualização charts	5 months ago
liquibase.properties	Atualização liquibase - Atualizar versão - BASIS-179814	6 months ago
master.xml	BASIS-229783 correcoes liquibase	4 days ago
postgresql.jar	BASIS-128816 - Criação das Entidades para a adesão de escolas	1 year ago

Name	Last commit	Last update
..		
liquibase	Merge branch 'integracao-release3' of nto feature/dev-BASIS-15...	1 month ago
tls	Setup inicial do MS de usuário - BASIS-143561	1 year ago
application-dev.yml	Merge branch 'integracao-release3' of nto feature/dev-BASIS-15...	2 months ago
application-login-automatico.yml	Setup inicial do MS de usuário - BASIS-143561	1 year ago
application-prod.yml	Resolve BASIS-197169 "Paramentrizacao urls wsdl"	5 months ago
application-tls.yml	Setup inicial do MS de usuário - BASIS-143561	1 year ago
application.yml	BASIS-200236-Ajustes sistec comum	3 months ago
bootstrap.yml	Setup inicial do MS de usuário - BASIS-143561	1 year ago

5. CONTEINERIZAÇÃO E ORQUESTRAÇÃO

Muitas vezes há muitos obstáculos que se interpõem no caminho de se mover facilmente seu aplicativo através do ciclo de desenvolvimento para a produção. Além do trabalho real de desenvolver seu aplicativo para responder de forma apropriada em cada ambiente, você também pode se deparar com problemas com rastreamento de dependências, escalabilidade de sua aplicação, e atualização de componentes individuais sem afetar a aplicação inteira.

A conteinerização e os projetos orientados a arquitetura de microservicos vieram para contribuir na solução de muitos desses problemas. As aplicações podem ser quebradas em componentes funcionais, gerenciáveis, empacotados individualmente com todas as suas dependências e implantados facilmente em qualquer infraestrutura, gerando uma simplificação nas atualizações de componentes e na escalabilidade.

5.1. Docker

É uma plataforma de código aberto que possibilita o empacotamento de uma aplicação ou ambiente dentro de um container, com isso o ambiente inteiro torna-se portável para qualquer outro host que contenha o Docker instalado. Como se reduz drasticamente o tempo de deploy de algumas infraestruturas e até mesmo aplicações, pois não há necessidade de ajustes de ambiente para o correto funcionamento do serviço, o ambiente é sempre o mesmo, configure-o uma vez e replique-o quantas vezes quiser.

A plataforma de contêineres é uma solução completa que permite que as organizações implantem sistemas complexos sem a necessidade do uso de máquinas virtuais, e seu custo inerente da necessidade do uso sistemas operacionais, em infraestruturas complicadas. É mais do que uma peça de tecnologia e orquestração, proporciona benefícios sustentáveis em toda a organização, fornecendo todas as peças que uma operação corporativa exige, incluindo segurança, governança, automação, suporte e certificação durante todo o ciclo de vida da aplicação. O Docker permite que os líderes de TI escolham como construir e gerenciar de maneira econômica todo o seu portfólio de aplicativos em seu próprio ritmo sem medo de bloqueio de infraestrutura e arquitetura.

De modo geral, os containers são mais interessantes devido a alguns fatores, como:

- Simplicidade: um container pode ser criado, iniciado, desligado, transportado e apagado de forma extremamente fácil.
- Leves: As imagens Docker são geralmente muito pequenas, o que facilita a entrega reduzindo o tempo para implantar novos recipientes de aplicativos.
- Rápida Implantação: Recursos podem ser rapidamente disponibilizados para desenvolvedores, testadores ou em produção.
- Portabilidade: É fácil transportar uma aplicação de um ambiente para outro ou migrar de um centro de processamento de dados local para um ambiente em nuvem pública. Isso gera independência de ambiente, de tecnologia utilizada ou do provedor que suporta as aplicações, e uma enorme liberdade dada aos usuários que podem fazer a escolha solução mais adequada a sua realidade.
- Manutenção simplificada: reduz o esforço e o risco de problemas com dependências de aplicativos.
- Elasticidade: Containers podem ser elasticamente criados e destruídos, permitindo tratar de forma eficiente flutuações de demanda. Se aplicando especialmente as aplicações baseadas em microsserviços.
- Controle de versão e reutilização de componentes: pode-se rastrear versões sucessivas de um container, inspecionar diferenças ou reverter para versões anteriores. Os recipientes reutilizam componentes das camadas anteriores, o que os torna visivelmente leves.

- Compartilhamento: Pode-se usar um repositório remoto para compartilhar seu contêiner com outras pessoas. Empresas fornecem registros públicos para esse propósito, e também é possível configurar seu próprio repositório particular.

5.2. Kubernetes

Com a escalada no uso de containers, surgiram alguns obstáculos, no sentido de como domar a complexidade no gerenciamento de aplicações compostas por centenas de containers juntamente com o desafio de empregá-los em larga escala garantindo a elasticidade e resiliência das aplicações. Uma das soluções veio com o Kubernetes, um sistema de código aberto que foi desenvolvido pelo Google para gerenciamento de aplicativos em containers através de múltiplos hosts de cluster utilizando Docker.

Seu principal objetivo é auxiliar na adoção da tecnologia de containers pelo mercado, bem como facilitar a implantação de aplicativos baseados em microsserviços. O Kubernetes fornece uma plataforma que provê a automatização, distribuição de carga, monitoramento e orquestração entre containers, eliminando diversas ineficiências relacionadas à gestão de containers graças a sua organização em pods(menores unidades dentro de um cluster) que acrescentam uma camada de abstração aos containers agrupados.

Em resumo, o Kubernetes oferece aos usuários uma plataforma simples de gestão de infraestrutura e orquestração de aplicações com containers. Enquanto o Docker se concentra no empacotamento de uma aplicação e suas dependências num container e em sua implantação num servidor, Kubernetes vai além, sua função é orquestrar a implantação de aplicações compostas por múltiplos containers, gerenciá-las e monitorá-las, garantindo resiliência e escalabilidade em clusters de servidores distribuídos.

O uso da plataforma Kubernetes para automatizar a maior parte dos processos manuais necessários para implantar e escalar microsserviços será utilizado como parte fundamental na infraestrutura da arquitetura, dessa forma viabilizando e minimizando a complexidade na implantação de sistema distribuídos complexos.

5.3. Rancher

O Rancher oferece suporte a qualquer distribuição Kubernetes certificada. Para cargas de trabalho locais, oferecemos o RKE. Para a nuvem pública, oferecemos suporte a todas as principais distribuições, incluindo EKS, AKS e GKE. Para cargas de trabalho edge, branch e desktop, oferecemos K3s, uma distribuição leve e certificada de Kubernetes.

No MEC Utilizamos as versões:

MEC Rancher	v2.4.5
User Interface	v2.4.28
Helm	v2.16.8-rancher1
Machine	v0.15.0-rancher43

EMPACOTAMENTO E CONFIGURAÇÃO DE APLICATIVOS

As aplicações serão desenvolvidas utilizando Java e Spring Boot. Muitas pessoas perdem tempo e as vezes não conseguem configurar uma aplicação do zero. Geralmente são necessárias várias configurações para só então começar a codificar. Imagine pular toda essa parte fastidiosa de configurações e criar um projeto onde você já tenha tudo o que precisa.

O Spring Boot é um projeto da Spring que veio para facilitar o processo de configuração e publicação das aplicações, sua intenção é ter o seu projeto rodando o mais rápido possível e sem complicações. O Spring Boot é construído em cima do Spring Framework, com isso ele obtém os benefícios de sua maturidade, escondendo a sua complexidade com um middleware que auxilia no desenvolvimento de microsserviços. Seu objetivo não é trazer novas soluções para problemas que já foram resolvidos, mas sim reaproveitar estas tecnologias e aumentar a produtividade do desenvolvedor.

Os pacotes dentro dos microsserviços serã criados de acordo com o seguinte padrão de nomeclatura: br.gov.mec.<app>.<microsservico>

5.4. Princípios

- Prover uma experiência de início de projeto (getting started experience) extremamente rápida e direta.
- Apresentar uma visão opinativa sobre o modo como devemos configurar nossos projetos Spring, mas ao mesmo tempo sendo flexível o suficiente para que a configuração possa ser facilmente substituída de acordo com os requisitos do projeto.
- Fornece uma série de requisitos não funcionais já pré-configurados para o desenvolvedor como, por exemplo: métricas, segurança, acesso a base de dados, servidor de aplicações/servlet embarcado etc.
- Não prover nenhuma geração de código e minimizar a zero a necessidade de arquivos XML.

5.5. Benefícios

5.5.1. Configurações

As configurações do Spring Boot permitem aos microsserviços ir muito longe, em alguns casos sem a necessidade de sobrescrever absolutamente nada. Quando um serviço tiver que ir a produção, certas propriedade, como a porta que o contêiner embarcado usará, podem ser trocadas no ambiente, o framework fornece várias maneiras de sobrescrever as configurações padrões do projeto.

5.5.2. Empacotamento

Uma vez que o micro serviço está pronto para ser inWstalado, o ferramental do Spring Boot ajuda a gerar um artefato leve e executável, pois fornece plugins para Gradle e Maven, que permitem criar um arquivo JAR executável para distribuição.

Esse modelo de empacotamento (standalone JAR) casa-se muito bem com as unidades de execução desejáveis em uma arquitetura de microsserviços e conteinerização, pois permite que aplicações feitas em Spring Boot sejam empacotadas, gerando a facilidade de serem disponibilizados em ambientes de execução embarcado, com Undertow que é o motor de servlet do Wildfly capaz de iniciar em segundos.

Em suma, o Spring Boot reconhece desde o início os benefícios de decompor serviços monolíticos em microsserviços distribuídos, ele foi projetado para tornar o desenvolvimento e a construção dos microsserviços mais simples, permitindo que as aplicações se liguem a um poderoso subconjunto de funcionalidades que de outra forma precisariam ser configuradas de forma explícita ou serem feitas programaticamente.

Seu trabalho é feito de maneira muito elegante, livrando-nos das preocupações desnecessárias com dependências e configurações dos projetos que são na realidade as mesmas em 99% dos casos.

6. CONFIGURAÇÃO EXTERNALIZADA E CENTRALIZADA

Uma aplicação frequentemente usa serviços de infraestrutura como banco de dados, registro de serviços, mensageria etc. Além disso, é possível que a aplicação precise se integrar com vários outros serviços de terceiros. Daí surge o problema de como possibilitar que um serviço seja executado em múltiplos ambientes (desenvolvimento, teste, homologação, produção) sem modificação e/ou recompilação. A solução é usar mecanismos que permitam a aplicação, no momento de sua inicialização, ler configurações a partir de um serviço externo.

O Spring Cloud Config fornece suporte para configuração externalizada em ambiente distribuídos. Os dados de configurações podem ser lidos a partir de várias fontes como sistema de arquivos, git, configmaps do Kubernetes, vault etc. Também é possível adicionar implementações alternativas e configurá-las através do Spring.

Com uso de microsserviços, podemos criar um servidor de configuração central onde todos os parâmetros configuráveis dos sistemas distribuídos são controlados por versão. O benefício de um servidor de configuração central é que se alterarmos uma propriedade para um microsserviço, essa alteração de reflete automaticamente sem precisar de redeploy do aplicativo ou reinicialização do contêiner.

As configurações serão armazenadas em configmaps e disponibilizadas como arquivos para o Spring Cloud Config server usando o mecânismo do Kubernetes de mapeamento de configmaps como arquivos.

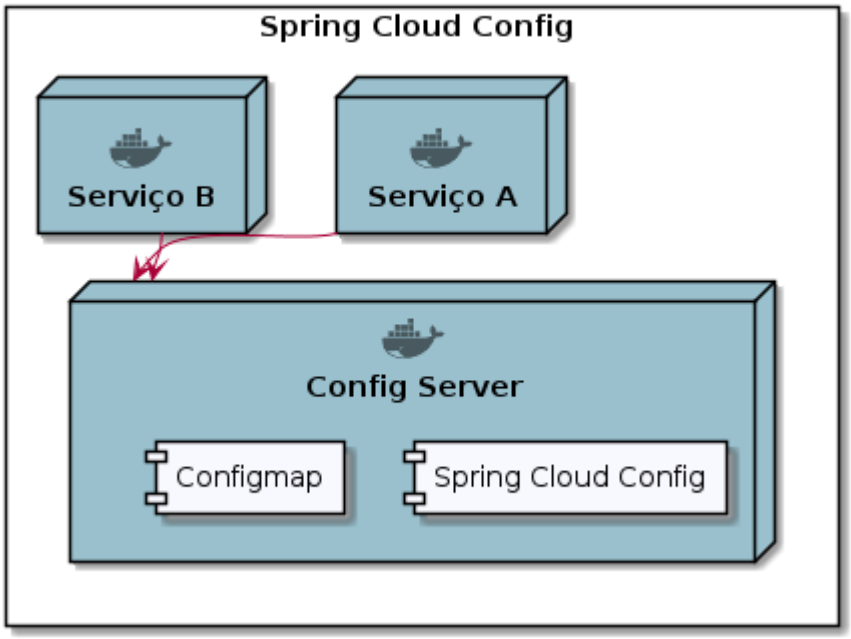

Figura 12. Configuração externalizada e centralizada - Spring Cloud Config

7. REGISTRO E DESCOBERTA DE SERVIÇOS

Nos sistemas desenvolvidos temos muitos serviços dinamicamente distribuídos na rede, onde as instâncias dos serviços mudam a todo instante devido a escala automática, falhas, atualizações e não temos controle de endereços IPs e nem sobre o nome das instâncias. Com isso surge o problema de como descobrir a localização das instâncias de um determinado serviço e se essas instâncias estão prontas para uso.

A solução ideal para essa situação é que o serviço se comunique e se registre a um servidor (único) para que o mesmo disponibilize os serviços disponíveis para uso, seguindo o padrão "The Server- Side Pattern Discovery".

O Spring Cloud fornece suporte ao "registro e descoberta de serviços" via “Spring Cloud Netflix Eureka“, permite que os serviços se encontrem e se comuniquem uns aos outros sem o nome do host e porta vinculados ao serviço. O ponto fixo em tal arquitetura consiste em um registro único de serviço com o qual cada serviço precisa se conectar.

O Eureka recebe mensagens de "heartbeat" de cada instância que pertencente a um serviço, caso o heartbeat falhe ao longo de um período configurável, a instância é marcada como não disponível e se não volta a enviar sinal de vida é removida do registro.

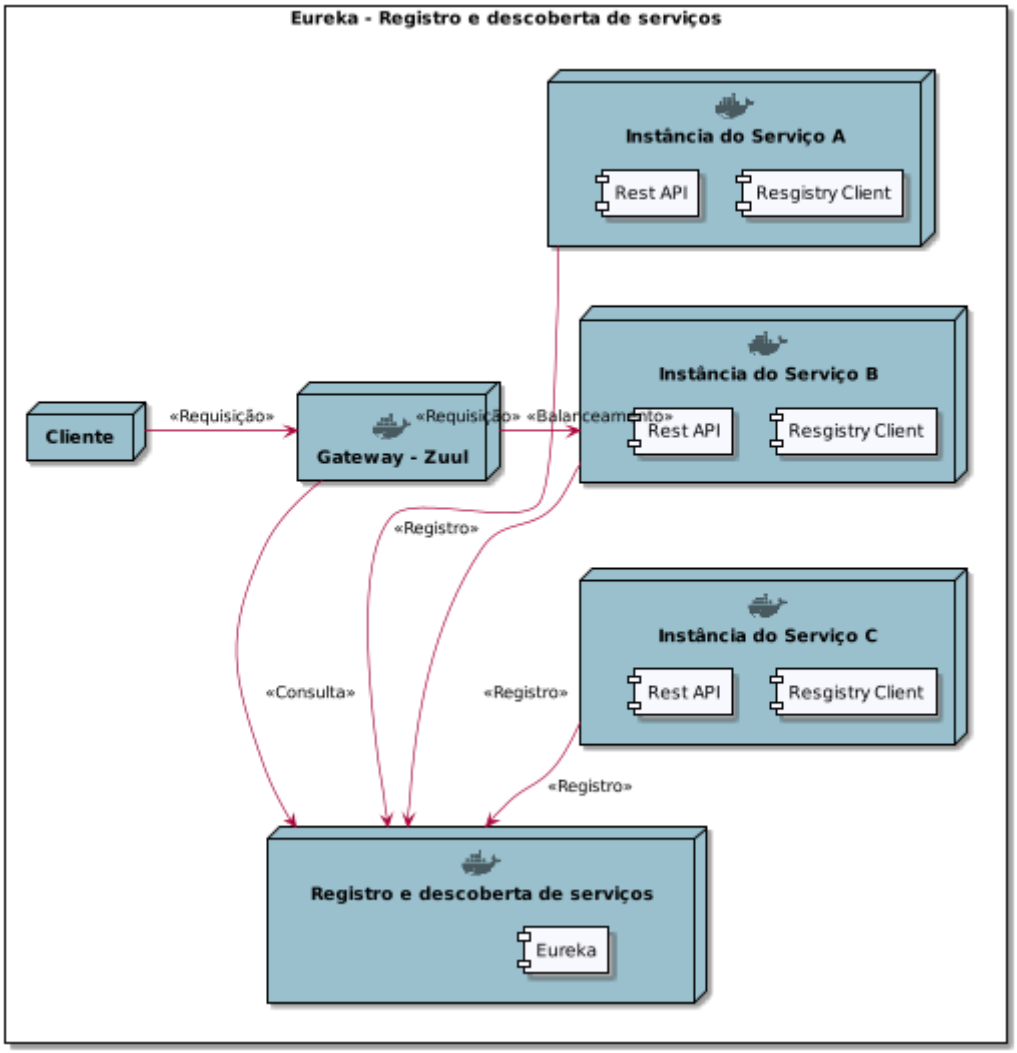

Figura 13. Registro e descoberta de serviços – Eureka

8. BALANCEAMENTO

O balanceamento client-side começou a se tornar popular devido a maior adoção de SOA e microsserviços. Diferente do balanceamento server-side, o balanceamento client-side não espera que um outro serviço distribua a carga, o próprio cliente é responsável por decidir para onde enviar o tráfego.

O Spring Cloud Netflix Ribbon é um balanceador de carga client-side, ele fornece controle sobre o comportamento de clientes HTTP e TCP, é possível escolher algoritmos de balanceamento de carga de forma declarativa como roundRobinRule, availabilityFilteringRule, weightedResponseTimeRule ou customizar suas próprias regras. Quando usado em conjunto com o Spring Cloud Netflix Eureka a lista de serviços disponíveis é fornecida automaticamente através do serviço de descoberta, que também é responsável por determinar se o serviço está ativo. Caso não se use o Eureka, a lista de serviços pode ser pré-definida em um arquivo de configuração da aplicação.

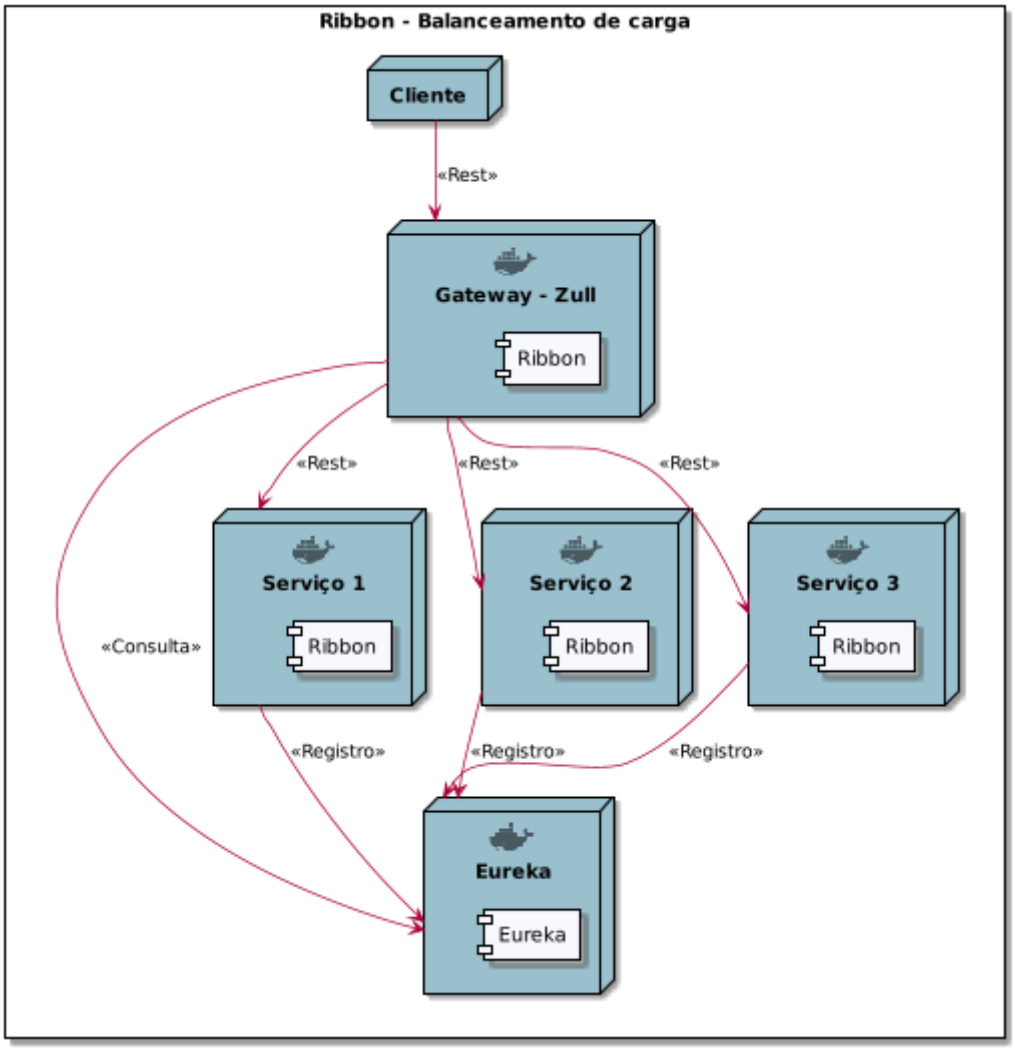

Figura 14. Balanceamento de carga - Ribbon

9. RESILIÊNCIA

No cenário de sistemas desenvolvidos com microsserviços é comum que existem várias chamadas remotas para diferentes serviços distribuídos em uma rede. A falha ou demora na resposta de algum desses serviços pode desencadear erros em cascata e fazer o sistema falhar como um todo.

O Hystrix é uma biblioteca fornecida pelo Spring Cloud Netflix que garante a tolerância a falhas, fornece otimizações para grandes volumes de requisições e implementa o padrão corporativo Circuit Breaker.

9.1. Circuit Breaker Pattern

O padrão Circuit Breaker protege o sistema evitando chamadas aos pontos de integração problemáticos. Os timeouts são usados para que após um determinado período de tempo a aplicação possa voltar a utilizar um serviço que estava causando erros.

A implementação do Hystrix visa descrever uma estratégia contra a falhas em cascata em diferentes níveis em uma aplicação, tendo seu uso bastante simplificado através do Spring Cloud.

9.2. Hystrix Health Check

O painel de saúde do Hystrix funciona junto com Spring Boot Actuator, fornecendo informações de várias partes da aplicação, como detalhes de beans, versões, configurações, logs e etc.

Quando é habilitado o Hystrix nos microsservicos, o Spring Actuator adiciona automaticamente o Hystrix Health nos endpoints das aplicações, com isso o actuator pode interagir invocando diferentes endpoints HTTP da aplicação ajudando no fornecimento de informações para o painel de saúde.

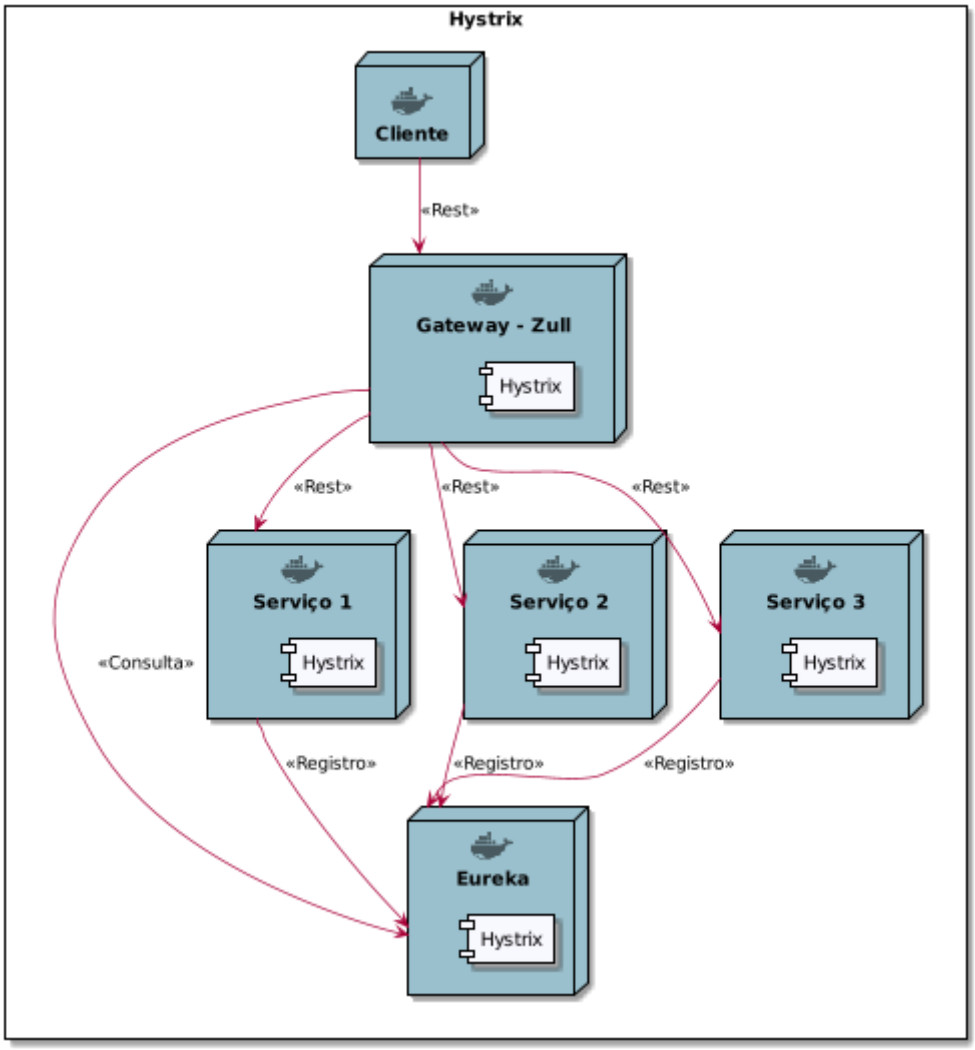

Figura 15. Resiliência - Hystrix

10. INTEGRAÇÃO ENTRE SERVIÇOS

10.1. Chamadas entre serviços

Em uma arquitetura de microsserviço, temos que nos integrar a diversos serviços para que uma funcionalidade ou sistemas fiquem completos, é quase inevitável escrever códigos repetidos de "web services clientes" para consumo de serviços.

Com o intuito de facilitar a integração de serviços, o Spring Cloud fornece o projeto Feign, uma maneira declarativa de criar web services clientes, gerando o mínimo de configuração, escrita e sobrecarga de código.

O Feign conecta seu código a APIs http utilizando decodificadores personalizáveis e tratamento de erros, funciona processando anotações em um modelo de template request. Embora o Feign esteja limitado ao suporte a APIs baseadas em texto, ele simplifica drasticamente o trabalho de desenvolvimento.

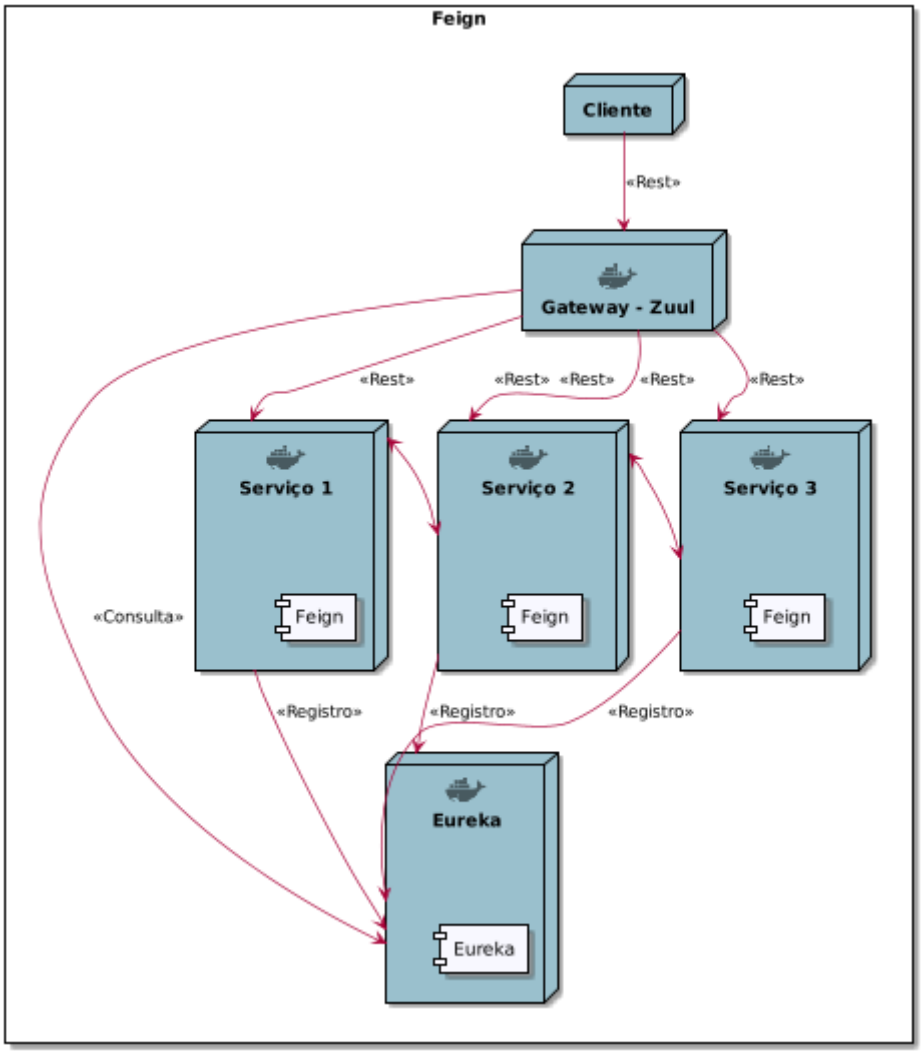

Figura 16. Integração entre serviços - Feign

10.2. Troca de mensagens

Message broker é um padrão de arquitetura para validação, transformação e roteamento de mensagens. Isso consiste em fazer a mediação de comunicação entre aplicações minimizando o conhecimento comum que os aplicativos devem ter uns dos outros, a fim de poder trocar mensagens, implementando efetivamente o desacoplamento.

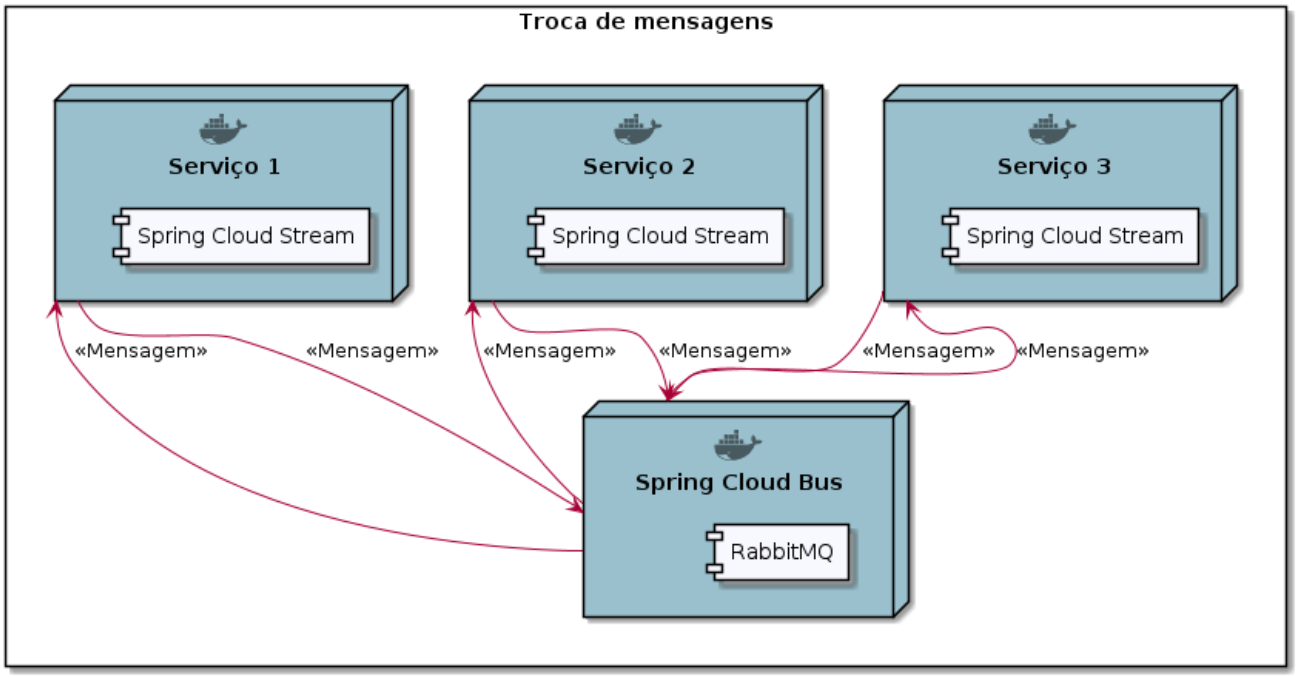

Figura 17. Troca de mensagens

10.2.1. Spring Cloud Bus

Responsável por fazer a interligação de nós dos sistemas distribuídos com o "message broker", podendo ser utilizado para transmitir alterações de estados como por exemplo: alterações de configurações ou instruções de gerenciamento.

Uma ideia chave é que o barramento de mensagem seja como um mediador distribuído para aplicativos Spring Boot, mas também pode ser usado como um canal de comunicações.

O projeto usa atualmente como implementação um intermediário que utiliza o protocolo de mensagem AMQP (Advanced Message Queuing Protocol), que permite que aplicativos clientes se comuniquem em conformidade com softwares de MOM (Message-oriented middleware), mais especificamente utilizamos como implementação o software RabbitMQ.

10.2.2. Spring Cloud Stream

É um framework que ajuda na criação de microsserviços orientados a eventos ou acionados por mensagens. Ele utiliza o Spring Integration para fornecer conectividade a "message brokers", provendo interfaces pré-definidas prontas para uso conforme os conceitos (publish-subscribe, consumer groups, partitions), simplificando a escrita de aplicações e microsserviços acionados por mensagens. Exemplos:

- As mensagens designadas para destinos são entregues pelo padrão de mensagens Publish- Subscribe.
- Os editores categorizam as mensagens em tópicos, cada um identificado por um nome.
- Os assinantes podem manifestar interesse em um ou mais tópicos.
- Filtro de mensagens, entrega de mensagens por tópicos de interesse dos assinantes.
- Conversão de mensagens para tipos de conteúdo específicos.

10.2.3. RabbitMQ

É o "message broker" utilizado, ou seja, é um intermediário de mensagens oferecem aos aplicativos uma plataforma comum para enviar e receber mensagens, provendo segurança, confiabilidade, roteamento flexível, clusterização, federação entre clusters, filas de mensagens seguras e uma interface de monitoramento para controle e rastreamento das informações.

11. API GATEWAY

Um dos maiores objetivos quando se constrói uma aplicação baseada em microsserviços é poder criar serviços que possam ser escalados e disponibilizados independentemente. A quantidade de serviços pode ser muito grande e gerenciar os detalhes de conexão como URLs e portas pode se tornar algo muito trabalhoso e suscetível a erros. Para isolar os sistemas externos a rede de microsserviço de sua complexidade inerente à sistemas distribuídos complexos o uso de API Gateway se torna imprescindível. Para organizar a criação se gateways e diminuir a quantidade de pontos de falha das aplicações que tem a necessidade de interagir com os microsserviços será feito o uso da variação do padrão de projeto "Backends for frontends".

O Zuul é um API Gateway e funciona como um único ponto de entrada para todos os clientes. Ele recebe todas as requisições e as delega para os microsserviços internos.

Além disso podem ser implementadas regras de roteamento ou qualquer implementação de filtro. Outros aspectos comuns de um sistema web, como autenticação, segurança e monitoramento não precisam ser replicados para cada serviço, essas configurações podem ser centralizadas no API Gateway.

Um ganho adicional nessa estratégia é que o dado trafegado entre o consumidor e o serviço de backend se torna mais apropriado. Por exemplo, o Gateway pode ter alguma lógica em sua camada de serviço para reconhecer quando um grande volume de requisição está sendo feitas para obter por exemplo: os detalhes de um produto específico, e ao invés de chamar o microsserviço que retorna os detalhes do produto em cada requisição, ele pode decidir servir o dado a partir de um cache por algum período de tempo pré-definido. Esse efeito pode melhorar dramaticamente o desempenho e reduzir a carga de rede.

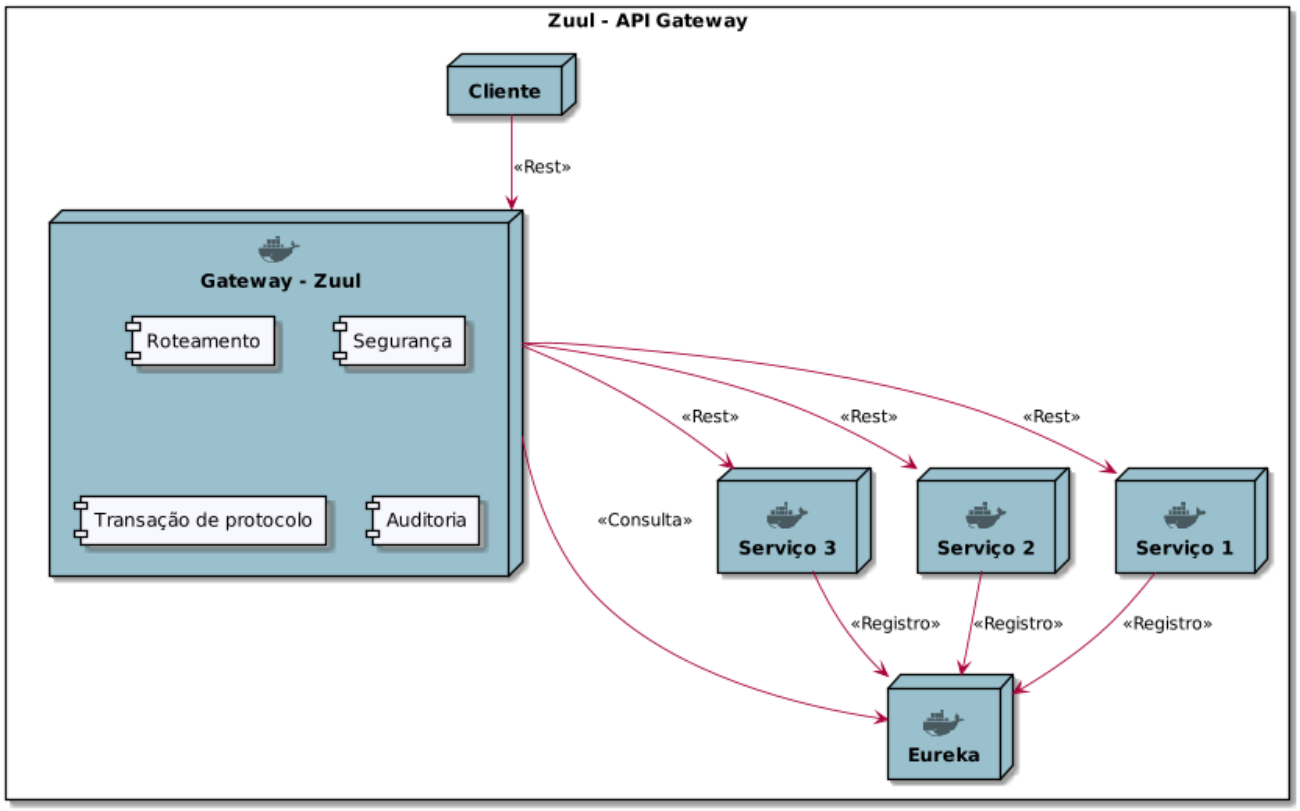

Figura 18. API Gateway - Zuul

A segurança da aplicação será realizada utilizando o protocolo OpenID no gateway. O gateway irá realizar a comunicação com o SSO e gerar um token de autenticação JWT que será repassado para os serviços enquanto o usuário estiver logado no SSO, os dados de autenticação e autorização serão armazenados em um banco de dados Redis para melhorar a performance das requisições.

12. MONITORAMENTO E RASTREAMENTO

Em sistemas desenvolvidos como monólitos é difícil identificar exatamente onde estão os gargalos ou comportamentos inesperados das aplicações, pois as funcionalidades estão agrupadas em único arquivo de deploy. Com o desenvolvimento de sistemas distribuídos (microsserviços) gera-se uma maior necessidade no uso de ferramentas de monitoramento, rastreamento de logs, métricas e gerenciamento de aplicativos.

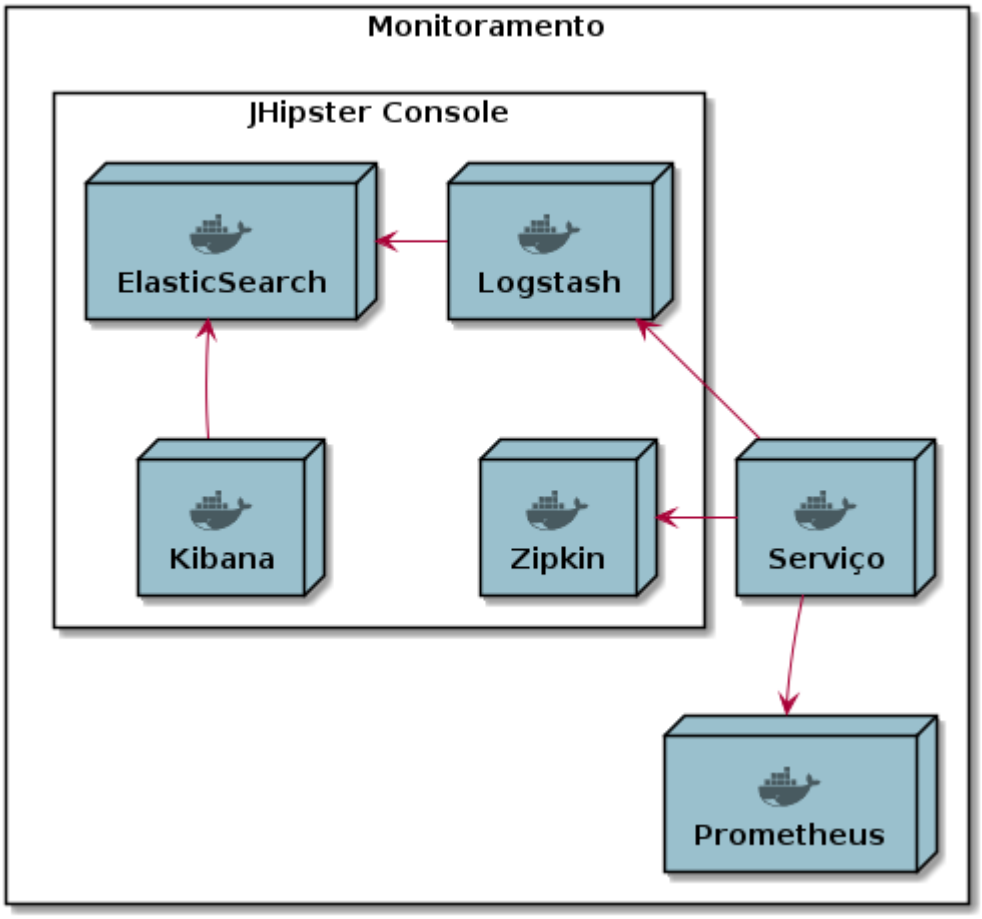

Figura 19. Monitoramento e Rastreamento

12.1. Monitoramento

Visando resolver problemas de monitoramento a empresa Elastic criou uma solução composta por 3 projetos opensource: Elasticsearch, Logstash, Kibana (ELK Stack), provendo de forma simples suporte a pesquisa, análise, captura, visualização e indexação de dados.

A ideia do "ELK Stack" é simples, mas muito interessante, em resumo o Logstash recebe os logs de distintas fontes, realiza transformações, normalizações, agrupamentos e envia os dados para a

pilha do Elasticsearch, que faz indexação das informações. O Kibana, por sua vez, permite que os usuários visualizem os dados em gráficos e tabelas.

12.1.1. Losgstash

É um pipeline de processamento de dados do lado do servidor, podendo reunir, enriquecer e unificar os dados dinamicamente, independentemente do formato ou de esquema. O processamento em tempo real fica mais poderoso quando associado a plataforma Elasticsearch, podendo inserir dados de várias fontes simultaneamente.

12.1.2. Elasticsearch

É uma plataforma altamente escalável para a busca, indexação e análise de dados. É acessível a partir da interface de serviço RESTful e utiliza o esquema less JSON (JavaScript Object Notation) para armazenar os dados. Ele é baseado na linguagem de programação Java, fazendo com que o Elasticsearch seja executado em diferentes plataformas e permitindo que seus usuários explorem grandes volumes de dados com extrema velocidade em tempo real.

12.1.3. Kibana

É uma plataforma analítica e visual desenhada para trabalhar com o Elasticsearch, é utilizada para visualização e interação com os dados armazenados nos índices do Elasticsearch. Disponibiliza uma interface amigável para apresentação dos dados via gráficos, tabelas, mapas em tempo real.

12.1.4. JHispter Console

É uma ferramenta de monitoramento baseada na ELK Stack (Elasticsearch, Logstash e Kibana), ele fornece painéis de controle e ferramentas de análise possibilitando uma visão geral de desempenho da infraestrutura em tempo real. No caso de análise das evoluções das métricas no decorrer do tempo, os dados podem ser direcionados via JHispter para uma outra ferramenta, no caso utilizamos o Prometheus.

Com o JHipster podemos fazer alterações visuais no Kibana em minutos, em vez de horas, caso fosse necessário executar toda configuração de infraestrutura do ELK Stack.

O JHipster Console suporta o monitoramento de uma arquitetura de microsserviços, fornecendo os seguintes recursos:

- Rastreamento distribuído com Zipkin;
- Enriquecimento do log com nome do serviço, id da instância, id’s de correlação do Zipkin;
- Servidor Zipkin e UI para visualizar traços e expansões;
- Vinculação entre a interface do usuário do Zipkin e Kibana para que se navegue nos logs;

- Encaminhamento de métricas para sistemas alternativos, possibilitando a integração com o Prometheus.

12.1.5. Prometheus

É um kit de ferramentas open source para monitoramento e alerta, tem um bom funcionamento ao gravar séries temporais numéricas. Ele se ajusta tanto ao monitoramento centralizado em máquina (machine-centric), quanto ao monitoramento de arquiteturas dinâmicas orientadas a serviços.

Em um mundo de microsserviços, fornece grande suporte as coletas e consultas de dados multidimensionais. Foi projetado para gerar confiabilidade, no intuito de ser o sistema em que se confie durante uma interrupção, no sentido de permitir e facilitar rápidos diagnósticos de problemas.

Cada servidor Prometheus é independente, ou seja, não depende do armazenamento de rede ou de outros serviços remotos, a ideia é fornecer confiabilidade quando outras partes da infraestrutura estiverem quebradas, necessitando o minimo possível da infraestrutura para utilizá-lo.

Principais características:

- Modelo de dados multidimensional organizados na forma de chave/valor e agrupados de forma temporal para cada métrica;
- Linguagem de consulta flexível;
- Ausência de dependência de armazenamentos distribuídos; Nós únicos e autônomos em cada servidor;
- Coleta de dados históricos por meio de modelo "PULL" sobre o HTTP;
- Suporte a gráficos e painéis de controle.

12.1.6. Rastreio de requisições

Tracing é uma forma de rastrear as requisições dos sistemas possibilitando a equipe de TI identificar problemas em cada serviço. Além de rastrear, é possível contextualizar cada requisição, essa contextualização permite por exemplo: comparar métricas atuais com dados históricos de cada tipo de requisição, podendo se identificar o tempo de resposta e comparar a média histórica.

12.1.7. Sleuth

Na arquitetura proposta, o Spring Cloud Sleuth é a ferramenta usada para fazer o tracing, adicionando informações que permitem identificar unicamente cada requisição. À medida que a requisição passa de serviço a serviço novas informações são adicionadas. É possível calibrar o Sleuth para rastrear apenas parte das requisições e dessa forma não sobrecarregar os

mecanismos de log com informações inúteis. Pode-se rastrear, por exemplo, apenas as requisições que estão respondendo com erro 500 ou indicar que apenas 10% das requisições devem ser rastreadas.

12.1.8. Zipkin

Ferramenta usada para entender os dados coletados pelo Sleuth, ele fornece meios de analisar e pesquisar dados. Através dessa análise é possível identificar áreas onde as melhorias devem ser aplicadas.

12.2. Auditoria de aplicações

A auditoria de aplicações tem por objetivo padronizar e controlar a forma como as aplicações devem realizar o processo de auditoria de informações, acessos etc.

Para isso, serão utilizados os recursos apresentados nas seções Troca de Mensagens, Monitoramento e Rastreio de requisições.

A estrutura de auditoria resume-se em a aplicação postar mensagens de eventos de auditoria para o serviço de mensageria (RabbitMQ). Ele irá encaminhar essa mensagem recebida, para o módulo da aplicação que irá processar a mensagem e posteriormente (de forma assíncrona) armazená-la no Elastichsearch.

A figura abaixo ilustra o fluxo de comunicação entre os elementos da aplicação durante o processo de auditoria.

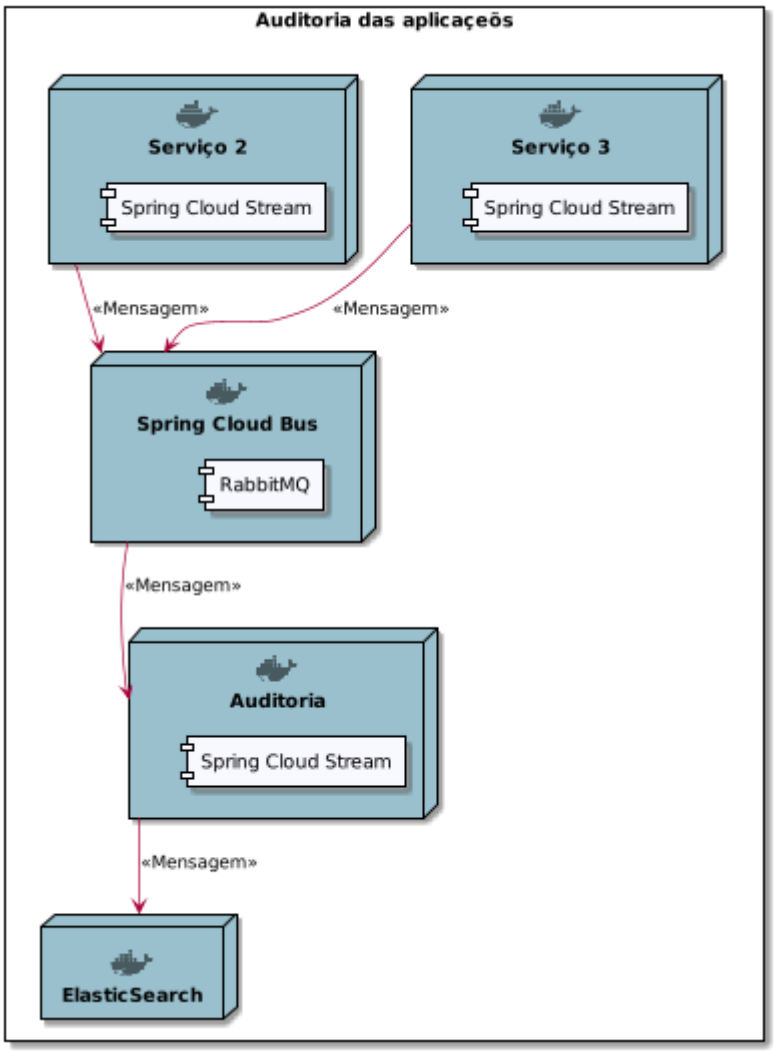

Figura 20. Auditoria de aplicações

Dentro da mensagem de evento de auditoria, existem informações extras tais como, IP de quem executou a ação, login do usuário, hora da ação e CorrelationID (permite agrupar trilhas de auditoria de de uma mesma requisição entre vários serviços). Essas informações são inseridas pelo microsserviço de auditória.

12.3. Desempenho de aplicativos

GlowRoot é uma ferramenta de APM (Application Performance Management), seu funcionamento consiste no gerenciamento, monitoramento de desempenho e disponibilidade dos aplicativos de software por períodos configuráveis de tempo para cada ação de usuário na aplicação, bem como a visualização dos dados de forma gráfica e configurável. Sua principal

finalidade é diagnosticar problemas complexos de desempenho de software visando manter um nível de qualidade do serviço.

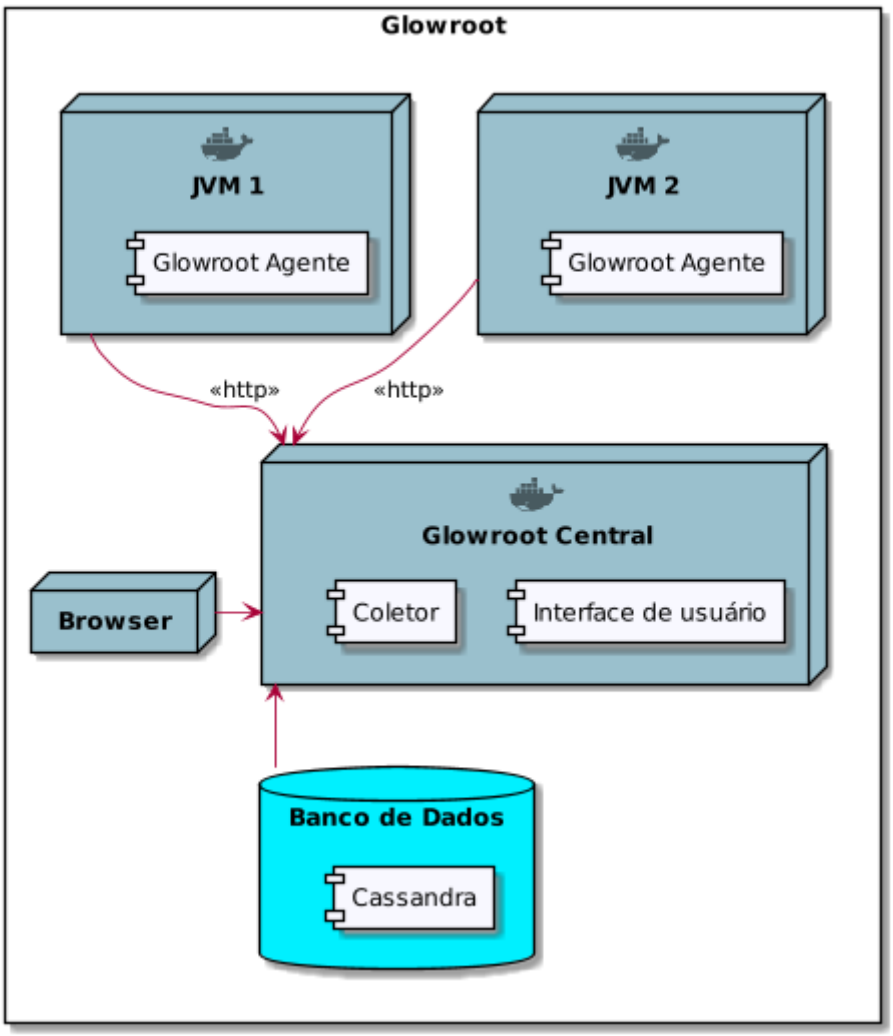

Figura 21. Desempenho de aplicativos

Principais características:

- Captura de rastreamento para solicitações e erros lentos;
- Perfil de usuários com filtros;
- Gráficos de tempo de resposta;
- Gráficos de percentuais;
- Captura e agregação de SQL;
- Captura e agregação de chamadas de serviço;

- Captura de atributos MBean com gráficos;
- Alerta configurável;
- Histórico de todos os dados com retenção configurável;
- Suporte total para solicitações assíncronas;
- Interface gráfica responsiva para suporte mobile;
- Coletor central opcional.

13. **CACHE DISTRIBUÍDO**

Hazelcast é uma plataforma distribuída de dados em memória para Java, sua arquitetura contribui para alta escalabilidade e distribuição de dados em ambientes clusterizados, ajudando na descoberta automática de nós e sincronizações inteligentes.

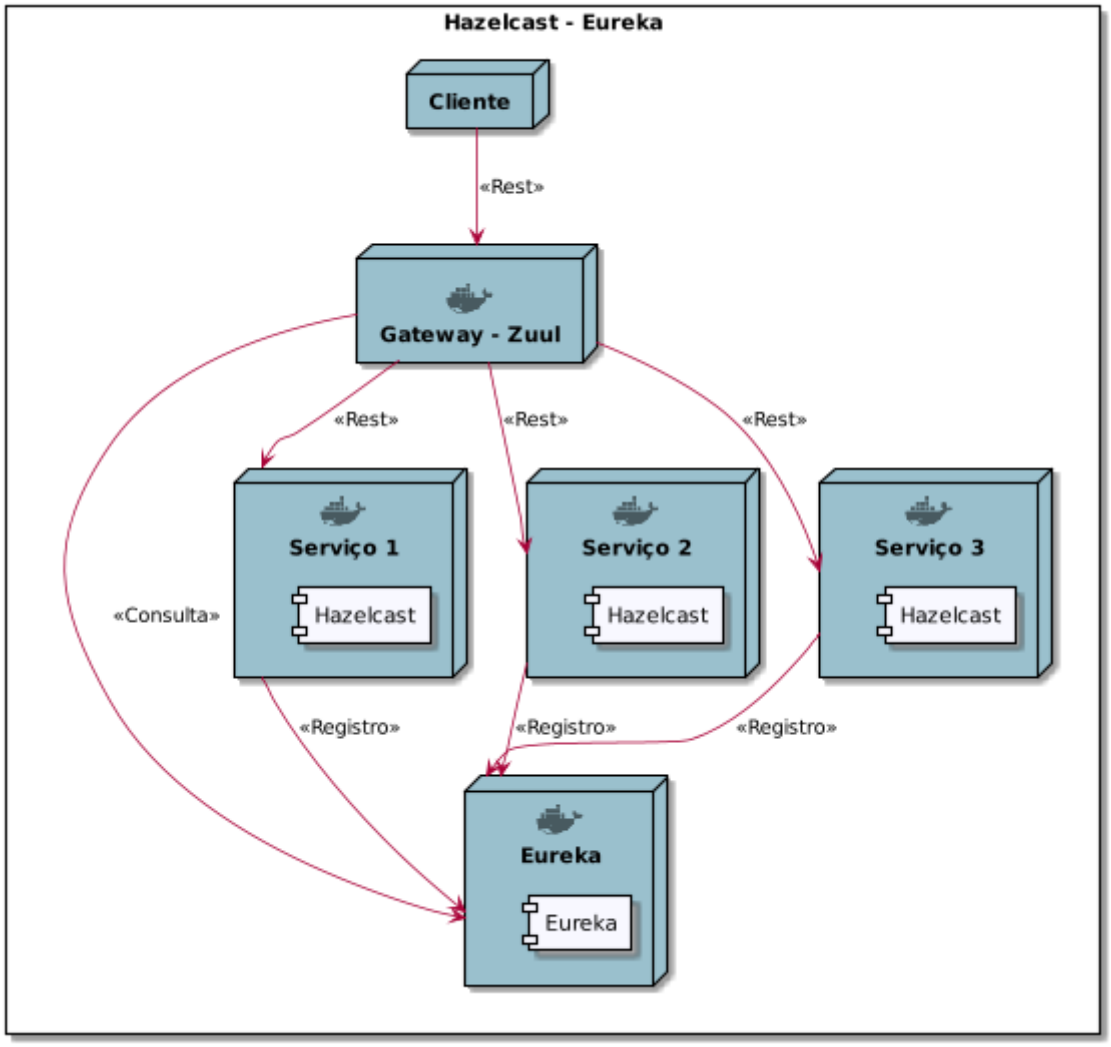

Figura 22. Hazelcast integrado com a registry do Eureka

13.1. Hazelcast integrado com a registry do Eureka

13.1.1. Serviço de descoberta

Como executar mais de um processo e fazê-los se encontrarem? Teoricamente, existem duas abordagens, entrar em contato com todos os processos ou com um subconjunto deles.

Em uma solução em que a seleção de máquinas é possível, a forma mais simples é citá-las, a instância "A" tenta localizar a instância "B" na máquina número "x" que tem o endereço de

máquina especificado em um arquivo de configuração, uma solução simples, mas o problema vem em ambientes virtualizados.

Quando novas máquinas são criadas ou re-alocadas é difícil conhecer os nomes dos hosts ou endereços IPs com antecedência, portanto não é possível atualizar os arquivos de configurações e consequentemente não podemos dizer a uma instância a localização de outra instância se não temos essa informação.

A solução está no "registro e descoberta de serviços" via Spring Cloud Netflix Eureka juntamente com o Hazelcast. Com o Eureka , podemos encontrar os serviços sem os nomes dos hosts e portas vinculados, o único "ponto fixo" consiste em um registro de serviços únicos com o qual cada serviço precisa se conectar, consequentemente o "Eureka" é um provedor de serviços ao Hazelcast, que por sua vez obtém os hosts que precisam ser guardados, podendo ele "Hazelcast" ser um cluster de servers com um nível controlado de segurança de dados.

13.1.2. Segurança dos dados

Os dados têm um valor variável para cada empresa, entretanto, é de comum entendimento que as informações mais valiosas não podem ser perdidas.

Então, como colocar cópias de dados em máquinas espelhadas que não falharão juntas, lembrando que a melhor opção depende de fatores externos?

A segurança de dados no Hazelcast é configurável, na configuração padrão, os dados do mapa têm duas cópias, há um mestre e um escravo de backup, podendo então, perder uma cópia dos dados que ainda restará outra cópia. Caso se configure o Hazelcast para uma configuração mais alta, os dados do mapa têm três cópias, um mestre e dois escravos de backup, podendo então, perder duas cópias que ainda restará uma cópia. Com a utilização dessas configurações, significa que os dados estão seguros no cluster caso não se perca todas as cópias dos dados, mas nem sempre é suficiente ter somente mais de uma cópia dos dados no cluster do Hazelcast, as cópias devem ser mantidas separadas para protege-las de problemas como falhas de máquinas. Apesar do Hazelcast conhecer os endereços dos clusters utilizando os registros do Eureka, não é possível sabe quais deles estão no mesmo gabinete, compartilham uma fonte de alimentação ou qualquer outra informação, portanto no sentido de promover uma maior segurança dos dados deve-se utilizar a funcionalidade "Partition Groups" do Hazelcast, que consiste em criar uma lista de nomes por grupos, associando os hosts a cada grupo com o objetivo de identificar a cópia principal dos dados em diferentes grupos, ou seja, a cópia principal dos backups dos registros não ficam em mesmo grupo de máquinas, melhorando a segurança dos dados no caso de falhas de máquinas.

Vale ressaltar que os movimentos físicos de uma infraestrutura devem ser monitorados, a segurança dos dados vem do conhecimento dos aspectos físicos das configurações das

máquinas, mas se no futuro alguma máquina for removida de gabinete os grupos do Hazelcast devem ser ajustados.

13.2. Cache de segundo nível JPA (Hibernate)

O Hibernate é uma biblioteca de mapeamento objeto-relacional (ORM), permite que você acesse dados relacionais de forma fácil e rápida, fornecendo mecanismos de mapeamento de tabelas de banco de dados relacionais, linhas, colunas, chaves estrangeiras para classes Java anotadas, gera consultas SQL com uma sintaxe fácil de usar e consulta dados com base no modelo de domínio orientado a objetos Java.

Com o intuito de prover acessos mais rápidos e menores tempos de latências as consultas nas bases de dados, o Hazelcast é utilizado como cache de segundo nível conectado ao Hibernate, mantendo os dados dos objetos pré-provisionados em memória. O cache é associado ao objeto Session Factory, isso significa que os objetos não estão restritos a uma única sessão, o cache é compartilhado entre todas as sessões deixando os dados disponíveis aos diversos aplicativos e não apenas a um único usuário. O principal objetivo é melhorar consideravelmente o desempenho dos aplicativos já que os dados consultados podem ser mantidos em memória. No primeiro acesso o Hibernate irá acessar o seu cache de primeiro nível, e caso a entidade não seja encontrada, ele irá para o cache de segundo nível provido pelo Hazelcast.

13.3. Spring Cache

O Hazelcast é utilizado para armazenar cache nos aplicativos baseados no Spring Framework, seu objetivo é distribuir os dados compartilhados em tempo real. Como o Spring Framework fornece uma camada de abstração para provedores de armazenamento em cache, utilizamos o hazelcast como implementação dessa camada "Spring Cache Abstraction", gerando a facilidade de poder anotar os métodos como @Cacheable e registrá-los no gerenciador de cache do Spring.

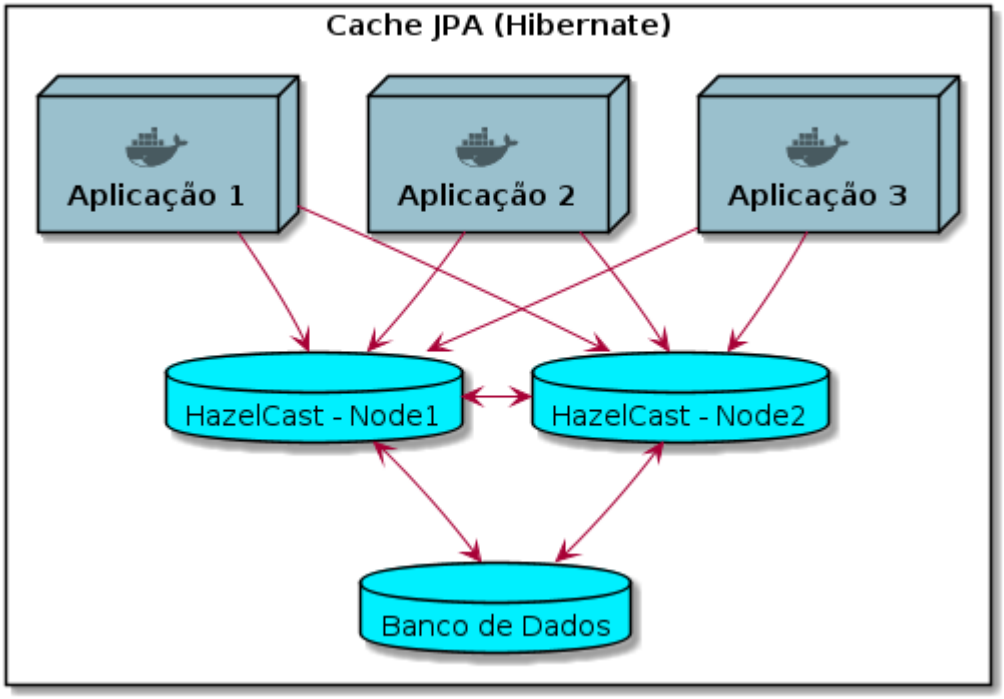

Figura 23. Cache de segundo nível JPA (Hibernate)

14. SINGLE PAGE APPS E PROGRESSIVE WEB APP

O Angular é uma plataforma que facilita a criação de aplicativos web, combina modelos declarativos, injeção de dependência, ferramentas de ponta com a integração de melhores práticas recomendadas para resolver desafios de desenvolvimento.

Angular permite que os desenvolvedores criem aplicativos que vivem em dispositivos móveis e web. Em setembro de 2016, a Google anunciou o novo Angular em sua versão final,uma versão completamente nova da antiga plataforma AngularJS, ou seja, foi lançado um novo Angular, compatível com as novas evoluções da web e do javascript, trazendo novos conceitos, melhorando o desempenho e as formas de programação, simplificando as APIs e as formas de debug.

O time do Angular planeja soltar atualizações das versões principais a cada seis meses, sempre melhorando e adicionando novos recursos na plataforma, entretanto as versões mais recentes devem ser atualizadas para que recebam as evoluções, o lado bom é que a equipe do angular promete manter a compatibilidade entre os lançamentos das versões principais e consecutivas.

O Angular utiliza Typescript como linguagem principal de desenvolvimento. O Typescript é um superconjunto do JavaScript com suporte a checagem de tipos, ele permite que aplicações grandes sejam construídas usando JavaScript usando práticas e ferramentas de alta produtividade como por exemplo checagem estática de código.

Nos tópicos abaixo serão detalhados dois dos principais conceitos utilizados pelo Angular, o conceito de SPA (single page apps) e PWA (progressive web apps).

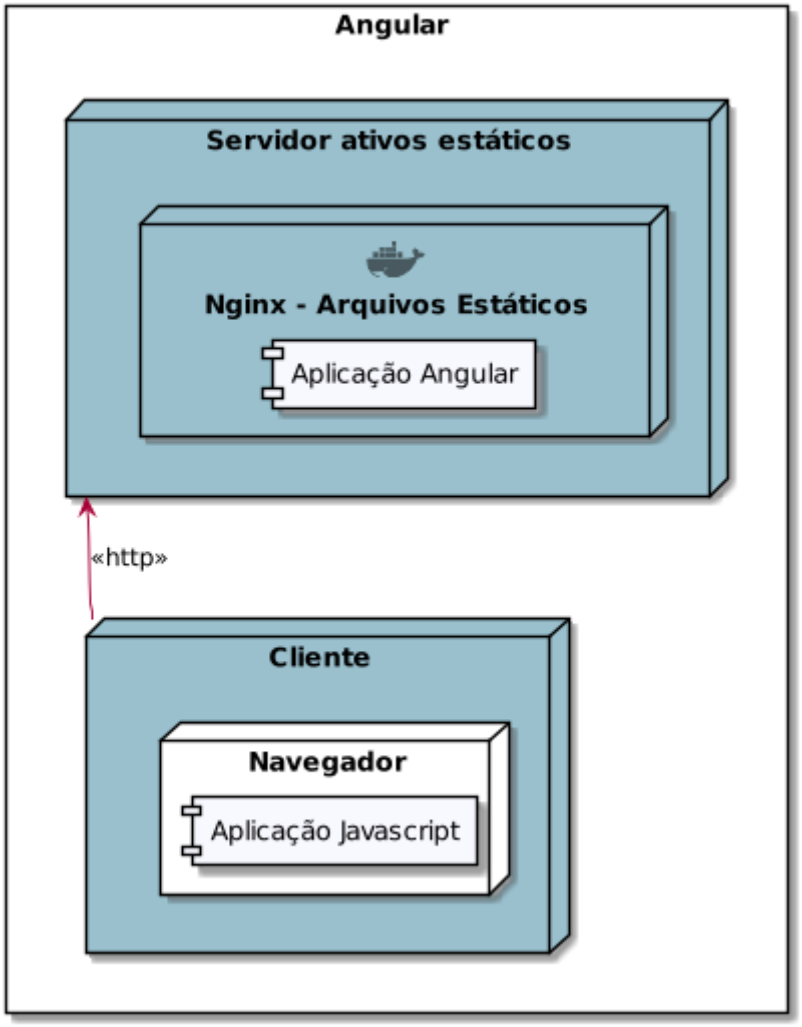

Figura 24. Aplicação Angular

14.1. Single Page Apps (SPA)

O conceito de SPA utilizado pelo Angular tem por característica se ajustar a uma única página da Web, onde os recursos são dinamicamente carregados e adicionados à página conforme necessidade, sem atualizar a página inteira.

As interações podem ser manipuladas sem atingir o servidor, podendo melhorar o desempenho da aplicação de várias maneiras.

Quando se navega em uma página, ela não é totalmente recarregada, apenas os novos dados são enviados pela rede enquanto o usuário navega pelo aplicativo.

A segurança entre a SPA e o gateway da aplicação será realizada com o uso de tokens JWT, que são tokens de acesso com duração limitada assinados digitalmente.

Benefícios:

- Velocidade: a maioria dos recursos (HTML + CSS + Scripts) são carregada uma vez ao longo da vida útil do aplicativo, apenas os dados são transmitidos de um lado para outro.
- Simplificar desenvolvimento: não há necessidade de escrever código para renderizar páginas
- No servidor, pode ser rodar o aplicativo sem a necessidade de utilizar um servidor.
- Facilidade em Depuração: consegue-se depurar o código utilizando navegadores web, podendo monitorar operações de rede, investigar elementos de página e dados associados.
- Reutilização de código: pode-se reutilizar o mesmo código de back-end para aplicativos da Web e aplicativos móveis nativos.
- Armazenamento de dados offline: pode-se armazenar em cache qualquer dado local de forma eficaz.

14.2. Progressive web Apps (PWA)

O termo Progressive Web App refere-se a um grupo de tecnologias, como service workers e notificações push que trazem confiabilidade, desempenho e uma melhora na experiência dos usuários com aplicativos web de forma nativa.

Service worker é um script que roda no navegador executando em segundo plano, separado da página da Web, possibilitando a execução de recursos sem interação do usuário.

Experiências off-line avançadas, sincronizações periódicas em segundo plano, notificações push, gerenciamento de armazenamento em cache, funcionalidades que normalmente exigem a implementação de um aplicativo nativo estão chegando à Web e no estão no Angular.

Os "service workers" oferecem a base técnica necessária para todos esses recursos.

O uso de progressive web apps será feito de acordo com a necessidade da aplicação de proporcionar uma o acesso offline. Permitindo um controle de como a aplicação web é instalada, carregada e atualizada.

14.3. Padrão de layout

Os sistemas irão utilizar os padrões de layouts definidos pelo MEC, que podem ser encontrados no Padrão de Sistemas.

15. IMPLANTAÇÃO

O desenvolvimento de microsserviços e a crescente demanda de mudanças negociais em sistemas de software modernos exigem que sistemas de software consigam se adaptar e sejam modificados com grande frequência. A adoção de microsserviços para desenvolver sistemas distribuídos pouco acoplados e coesos do ponto de vista negocial possibilita a criação e a evolução do software de forma isolada. Para possibilitar a entrega de software de forma mais frequentes em ambientes reais outras técnicas devem ser usadas e aderidas no ciclo de vida do desenvolvimento e entrega do software.

15.1. Controle de versão e gerenciamento de mudanças

O git é um sistema de controle de versão distribuído desenvolvido para controlar o código fonte de grandes projetos de forma rápida e eficiente. O modelo de branching do git permite que versões do código fonte coexistam de forma independentes e sejam integradas em questões de segundo.

O código fonte de cada aplicação e seus respectivos microsserviços serão separados em um repositório do git. Cada microsserviço será criado em uma nova pasta e eles serão decompostos por capacidade negocial. Por exemplo uma aplicação com capacidade negocial servico1 e capacidade negocial servico2 teria a seguinte estrutura de pasta:

```
aplicacao/
   charts/
   servico1/
src/
main/
docker/ servico2/
src/
main/
docker/ Jenkinsfile
```

Nota-se o arquivo Jenkinsfile que terá as configurações da pipeline de entrega contínua da aplicação. A pasta charts conterá o pacote da aplicação para o Kubernetes e a pasta src/main/docker em cada serviço que irá conter as definições das imagens do docker de cada microsserviço.

15.2. Entrega contínua de software

A integração contínua de código é uma prática de desenvolvimento de requer que o código seja integrado no repositório diariamente, possibilitando um fluxo de mudanças contínuos e constante

em aplicações. Quando o código é integrado frequentemente os erros são detectados rapidamente:

- Diminuindo a necessidade de integrações longas e demoradas;
- Aumenta a visibilidades das mudanças diminuindo o retrabalho;
- Rápida descoberta de erros;
- Diminui o tempo gasto debugando o código;
- Diminui os erros de integração permitindo que o software seja entregue mais rapidamente.

Para possibilitar a entrega contínua de software será utilizado o Jenkins que é um servidor de automação, construção e entrega de projetos de software. Ele possibilita a criação de pipelines de entrega e integração contínua para cada ambiente de software utilizando plugin de integração com o Kubernetes, Docker e Git. Para os projetos serão utilizados pipelines de entrega e integração contínua nas branchs do git respectivas com o uso de shared libraries. Será adotado o uso das configurações das pipelines de entrega contínua versionados nos repositórios de cada projeto.

15.3. Inspeção contínua de código

O SonarQube é uma ferramenta que possibilita o processo de inspeção contínua da qualidade do código, automatizando e identificando possíveis falhas de software e problemas de segurança de forma automática, e possibilitando a identificação e catalogação de falsos positivos.

O sonar realiza análise estática de código para detectar defeitos, vulnerabilidades, linhas de código duplicadas, uso de padrões de codificação, complexidade de código e se integra com ferramentas para gerar relatórios de testes e cobertura de código.

O sonar será utilizado na pipeline de entrega contínua de código e de implantação contínua de software. Seus indicadores deverão ser seguidos de acordos com as definições contratuais.

15.4. Implantação contínua de software

Helm é uma ferramenta para gerenciar e criar pacotes de software para serem implantados em ambientes Kubernetes, ela possibilita o controle de entrega e a atualização de pacotes de software em ambientes Kubernetes. O uso do helm chart para descrever a criação de pacotes será utilizado para realizar as implantações contínuas das aplicações nas pipelines do Jenkins. Os charts serão versionados nos repositórios do código fonte. Além disso será criado um repositório de configuração para cada projeto onde ficarão as configurações específicas para a implantação de cada chart em seu respectivo ambiente.

Branch Master: Essa branch é de total responsabilidade da equipe de INFRAESTRUTURA do MEC, visto que a integração-continua escuta essa branch e qualquer atualização nesta, a pipeline de

Estrutura dos ambientes no Gitlab, deploy é disparada e assim o ambiente de produção é atualizado de forma automática quando o código-fonte é "*mergeado"* para branch máster.

Desta forma os deploys em produção partirão obrigatoriamente do ambiente de homologação e serão disparados para equipe de infra via chamado (BMC). Caso tenha script de banco de dados, antes da execução do script um chamado deverá ser encaminhado para equipe de administração de dados para validação. O chamado deverá ser criado pelo responsável designado do sistema, e com o apoio das informações repassadas pelo time de arquitetura.

Branch Homologação: Essa branch é de total responsabilidade da equipe de ARQUITETURA do MEC, visto que a integração-continua escuta essa branch e qualquer atualização nesta, a pipeline de deploy é disparada e assim o ambiente de homologação é atualizado de forma automática, não havendo intervenção humana.

Branch Desenvolvimento: Essa branch é de total responsabilidade da equipe da FABRICA DE SOFTWARE do MEC, visto que a integração-continua escuta essa branch e qualquer atualização nesta, a pipeline de deploy é disparada e assim o ambiente de desenvolvimento é atualizado de forma automática, não havendo intervenção humana.