



Arquitetura de referência frontend - INEP

Índice

| | |
|--|----|
| 1. Introdução | 2 |
| 1.1. Definições, Acrônimos e Abreviações | 2 |
| 1.2. Documentos de Referência | 2 |
| 2. Objetivo do Documento | 3 |
| 3. Macro Arquitetura | 4 |
| 4. Microserviços | 5 |
| 4.1. Visão Geral das Camadas Arquiteturais | 6 |
| 4.2. Requisitos para Implementação das Camadas | 8 |
| 5. Containerização e Orquestração | 9 |
| 5.1. Docker | 9 |
| 5.2. Kubernetes | 10 |
| 5.3. OpenShift | 10 |
| 6. Empacotamento e Configuração de aplicativos | 12 |
| 6.1. Princípios | 12 |
| 6.2. Benefícios | 12 |
| 7. Configuração externalizada e centralizada | 13 |
| 8. Registro e descoberta de serviços | 14 |
| 9. Balanceamento | 15 |
| 10. Resiliência | 16 |
| 11. Integração entre serviços | 17 |
| 11.1. Chamadas entre serviços | 17 |
| 11.2. Troca de mensagens | 17 |
| 12. API Gateway | 18 |
| 13. Monitoramento e Rastreamento | 19 |
| 13.1. Monitoramento | 19 |
| 13.2. Rastreo de requisições | 20 |
| 14. Cache Distribuído | 21 |
| 15. Single Page Apps e Progressive Web App | 22 |
| 15.1. Single Page Apps (SPA) | 23 |
| 15.2. Progressive Web Apps (PWA) | 23 |
| 16. Implantação | 24 |
| 16.1. Controle de versão e gerenciamento de mudanças | 24 |
| 16.2. Entrega contínua de software | 24 |
| 16.3. Inspeção contínua de código | 25 |
| 16.4. Implantação contínua de software | 25 |

Histórico de Revisões

| Data | Versão | Descrição | Autor | Revisor |
|-------------|---------------|----------------------|-----------------------|-----------------------------|
| 18/08/2023 | 1.0.0 | Criação do documento | Vitor Santos Ferreira | Enus Carneiro do Nascimento |

1. Introdução

Este documento visa descrever a arquitetura de referência utilizada nos projetos.

1.1. Definições, Acrônimos e Abreviações

- **JSON**: JavaScript Object Notation.
- **XML**: Linguagem de Marcação Extensível.
- **Heartbeat**: mecanismo que visa saber se um serviço está disponível.
- **APM**: Gerenciamento de desempenho de aplicativos (Application Performance Management).

1.2. Documentos de Referência

- @angular: <https://angular.io/docs>
- Istio: <https://istio.io>
- 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/>

2. Objetivo do Documento

O objetivo deste documento é apresentar a arquitetura de referência utilizada nos projetos.

3. Macro Arquitetura

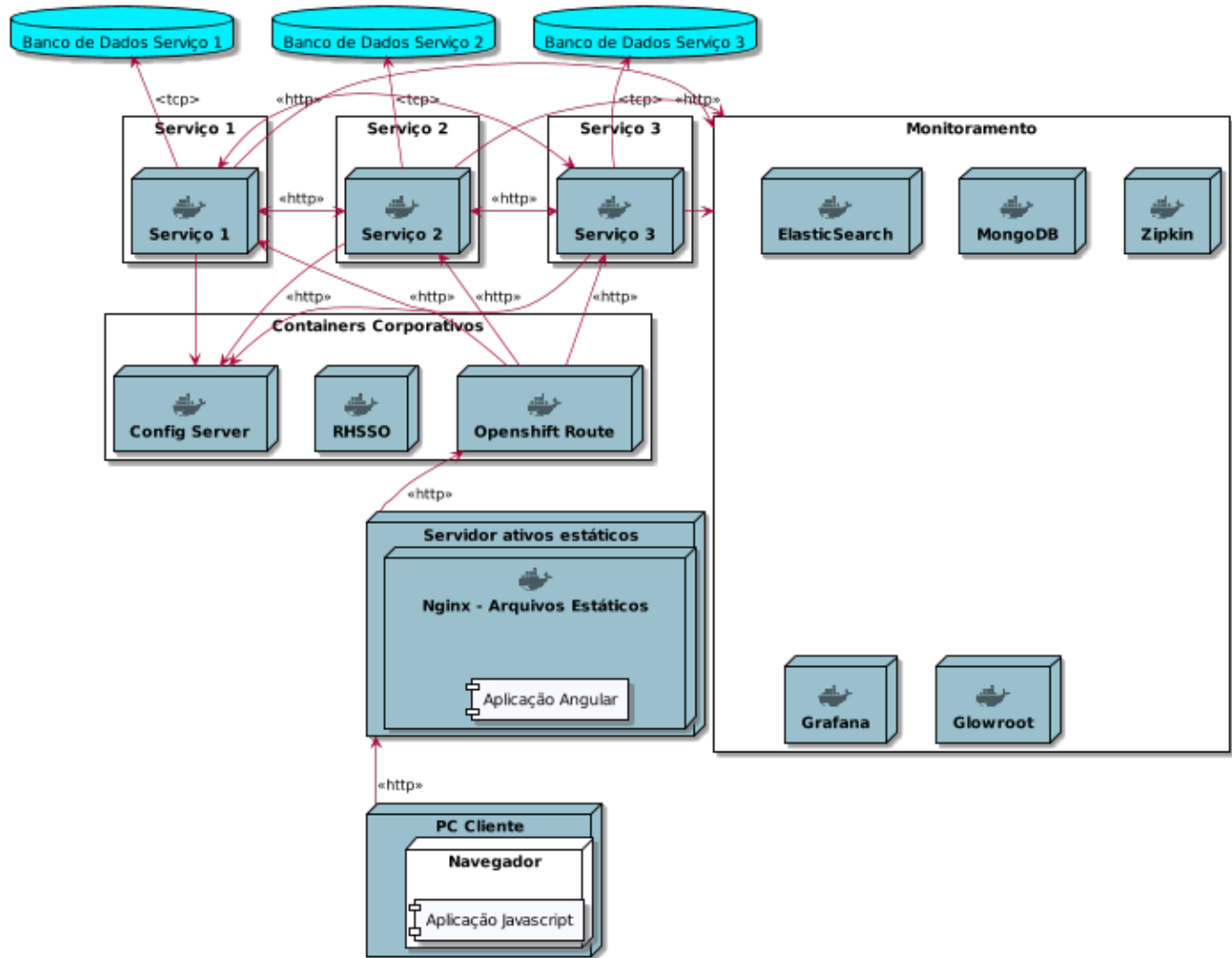


Figura 1. Visão Geral da Macro-Arquitetura

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.

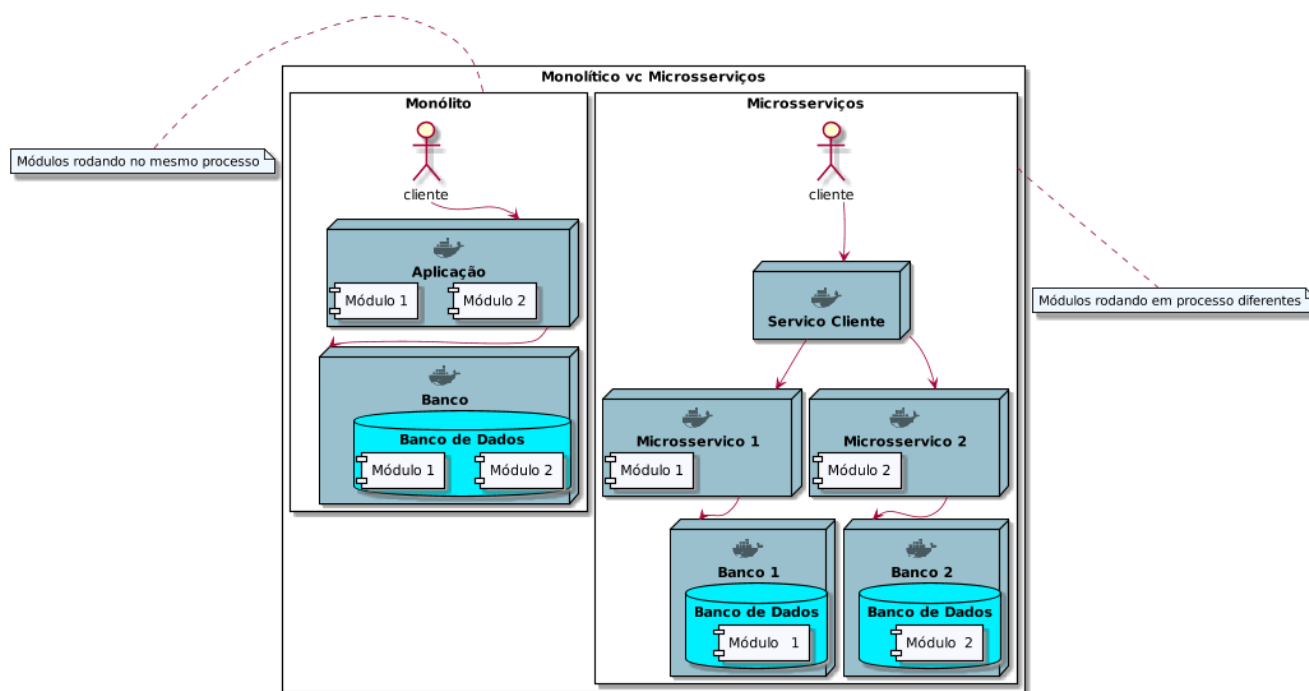


Figura 2. Microsserviços

Uma abordagem de sistemas distribuídos (Microsserviços) 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 softwares 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 URI dos serviços e a definição dos serviços RESTful será documentada utilizando a especificação OpenAPI.

Será utilizada a pattern de um banco de dados por microsserviço. Para atender aos requisitos de performance o uso do Elasticsearch juntamente com banco de dados relacional deverá ser utilizado, dessa forma aumentando a performance da aplicação no acesso as buscas do sistema que necessitam desse requisito não funcional. Em endpoints que serão necessários a agregação de informações de múltiplas entidades deverá ser criado um novo índice com todas as informações agregadas no Elasticsearch para minimizar o custo de acesso aos dados.

4.1. Visão Geral das Camadas Arquiteturais

Os microsserviços serão organizados em camadas. Em arquitetura de sistemas, damos o nome de camada a uma estrutura lógica de componentes de um software que interagem entre si para cumprir um objetivo específico, responsabilizando-se pela execução de uma dada tarefa ou um conjunto de tarefas semelhantes. Em alguns casos, esse agrupamento pode não ser apenas lógico, mas sim assumir um caráter físico. Isso ocorre quando uma camada possui a habilidade de ser executada separadamente do restante da aplicação.

A arquitetura baseada em múltiplas camadas oferece um modelo flexível e reutilizável para o desenvolvimento de software. Ao se dividir uma aplicação em camadas, evitamos o forte acoplamento entre os componentes, de forma que a maioria das alterações necessárias possam ser feitas de maneira mais isolada e pontual, reduzindo o esforço necessário à manutenção do sistema.

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.

O Swagger será utilizado para documentar as APIs REST.

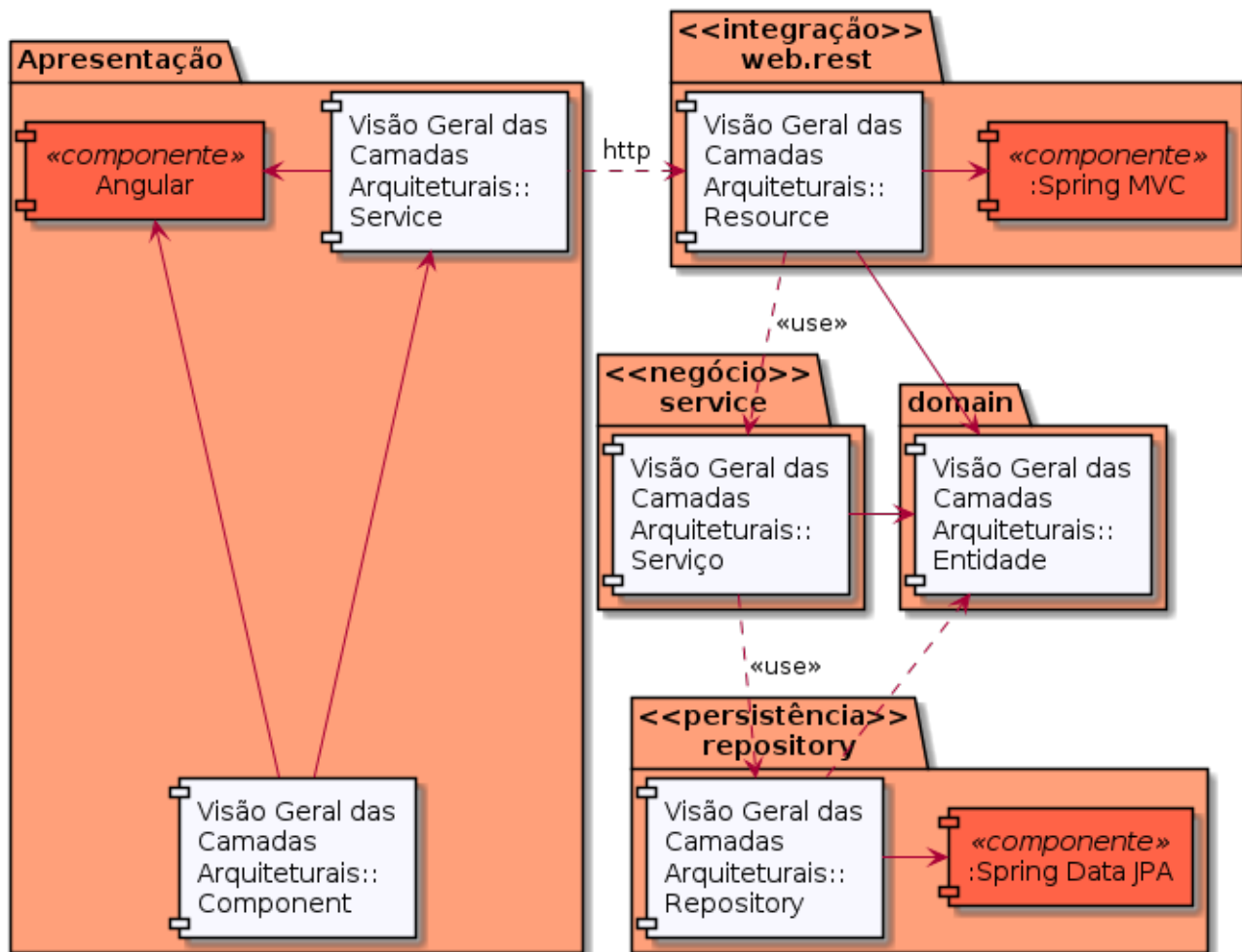


Figura 3. Visão Geral das Camadas Arquiteturais

4.1.1. 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 navegador, como cliques, e gerenciar o fluxo de execução do sistema.

4.1.2. Camada de Negócio

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

4.1.3. 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 real.

Ainda na persistência dos dados será utilizada exclusão lógica e não física dos registros. Haverá também auditoria das operações efetuadas pelo usuário.

4.1.4. Camada de Serviços

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

4.1.5. Camada de Integração

O objetivo da camada de integração é prover um meio de comunicação entre o sistema que vai ser desenvolvido e os demais sistemas que o circundam. Os objetos dessa camada fornecem à camada de negócio uma simplificação para o acesso aos sistemas externos, livrando-a da responsabilidade de tratar detalhes inerentes aos protocolos de comunicação envolvidos, formatações e conversões de tipos de dados e demais particularidades sintáticas e semânticas inerentes aos sistemas externos.

4.2. Requisitos para Implementação das Camadas

Uma arquitetura de microsserviços possibilita a uso de diversas linguagens de programação que podem ser utilizadas dependendo dos requisitos funcionais e não funcionais da aplicação.

4.2.1. Typescript

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

- NPM
- Docker

Bibliotecas a serem utilizadas pela aplicação:

- @angular 9
- @angular/material 9
- keycloak-angular 9
- keycloak-js 3
- lodash 4
- moment 2
- ng-recaptcha 5
- ng2-charts 2
- ngx-currency 2

5. Containerização 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 containerização e os projetos orientados a arquitetura de microsserviços 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 portátil para qualquer outro host que contenha o Docker instalado. Com ele 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 partes dos processos manuais necessário 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. OpenShift

O OKD é uma distribuição do Kubernetes otimizada para desenvolvimento contínuo de aplicações de implantação multi-tenant. O OpenShift agrega funcionalidades importantes para a implantação e manutenção dos containers como por exemplo:

- Builds: O OpenShift utilizada o Kubernetes para criar containers do docker e publicá-los em uma registry Docker. Ele possui três estratégias: Docker build, Source-to-Image e Custom build;

- Image Streams: Uma image stream pode ser usada para automaticamente realizar uma ação no Kubernetes quando uma nova imagem é modificada no registro do docker.

6. 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.

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.

6.1. Princípios

- Prover uma experiência de início de projeto (getting started experience) extremamente rápida e direta.
- Fornecer 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.

6.2. Benefícios

6.2.1. Configurações

Será utilizado variável de ambiente para a configuração dos microsserviços que serão incluídos no Kubernetes como `secrets`.

6.2.2. Empacotamento

As aplicações irão utilizar o Docker para poder criar um container executável e configurável em qualquer ambiente. E as dependências serão gerenciadas pelo NPM.

7. 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.

8. 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".

A descoberta dos serviços é feita por intermédio de um proxy Envoy que é implantado no mesmo pod de cada microserviço. O Envoy é um proxy de alta performance desenvolvido em C++ que media os tráfegos de entrada e saída de todos os serviços na rede. Usando o abstract model, o Pilot configura os proxies Envoy para executar balanceamento de carga para solicitações de serviço, substituindo qualquer recurso de balanceamento de carga específico da plataforma subjacente. Na falta de regras de roteamento mais específicas, o Envoy distribuirá o tráfego entre as instâncias no pool de balanceamento de carga do serviço de chamada, de acordo com o Pilot abstract model e a configuração do balanceador de carga.

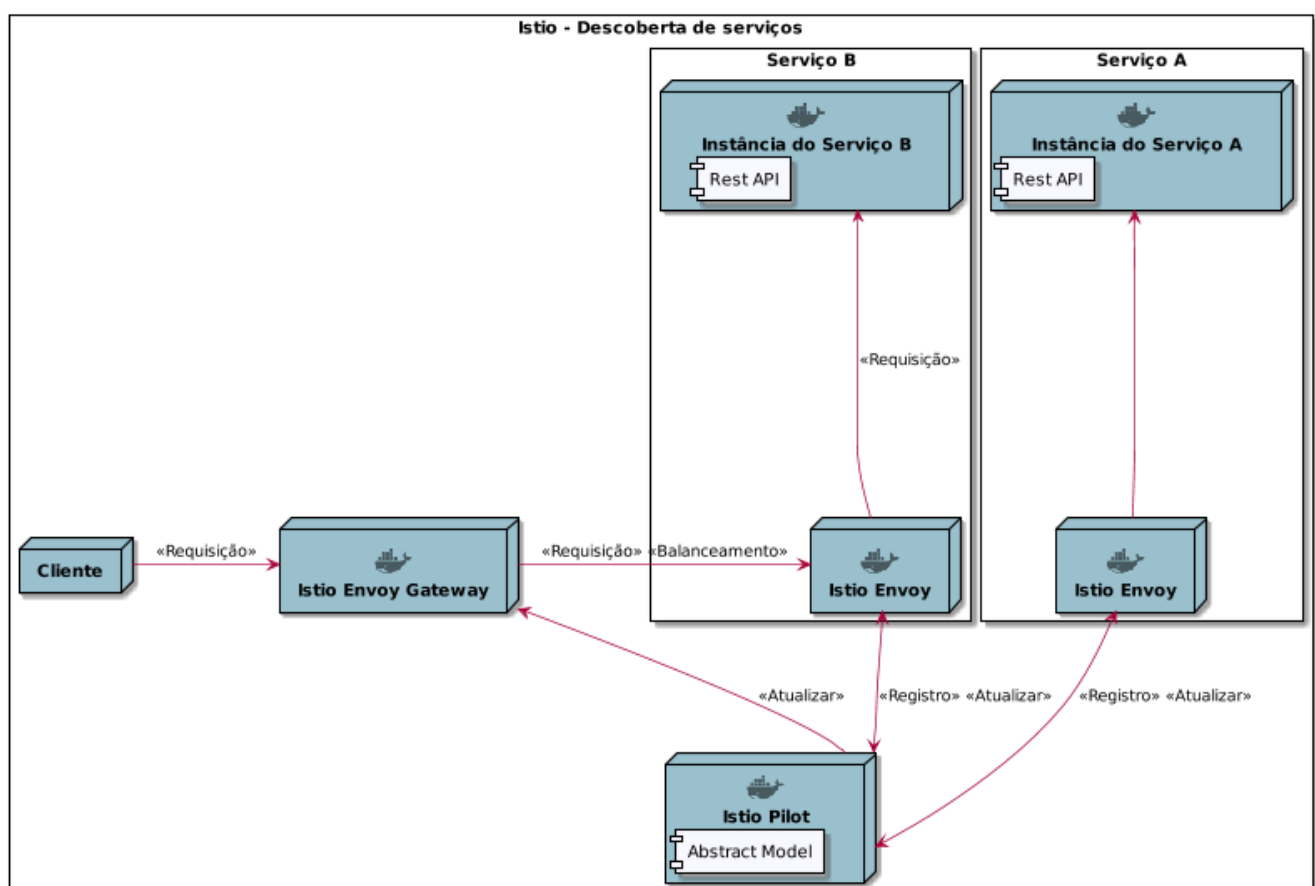


Figura 4. Registro e descoberta de serviços - Istio Pilot

9. Balanceamento

Será utilizado balanceamento server-site usando os **routes** e **services** do Openshift. Uma rota do OpenShift Container Platform expõe um serviço em um nome de host, para que clientes externos possam acessá-lo pelo nome.

A resolução DNS para um nome de host é tratada separadamente do roteamento. Seu administrador pode ter configurado uma entrada curinga DNS que será resolvida para o nó do OpenShift Container Platform que está executando o roteador do OpenShift Container Platform. Se você estiver usando um nome de host diferente, talvez seja necessário modificar seus registros DNS de forma independente para resolver o nó que está executando o roteador.

10. Resiliência

No cenário de sistemas desenvolvidos com microsserviços é comum que existam 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 Istio fornece capacidade diversas funcionalidades que podem ser configuradas utilizando suas regras de tráfego por meio de recursos customizados do Kubernetes (CRD):

- **Tempos limite e tentativas:** Um tempo limite é a quantidade de tempo que o Istio aguarda por uma resposta a uma solicitação. Uma nova tentativa é uma tentativa de concluir uma operação várias vezes se ela falhar. Você pode definir padrões e especificar substituições no nível da solicitação para tempos limite e tentativas ou para um ou outro.
- **Circuit breakers:** Os Circuit breakers impedem que o aplicativo pare enquanto aguarda que um serviço responda. Você pode configurar um Circuit breakers com base em várias condições, como limites de conexão e solicitação.
- **Injeção de falha:** A injeção de falhas é um método de teste que introduz erros em um sistema para garantir que ele possa resistir e se recuperar de condições de erro. Você pode injetar falhas na camada do aplicativo, em vez da camada de rede, para obter resultados mais relevantes.
- **Tolerância ao erro:** Você pode usar os recursos de recuperação de falhas do Istio para complementar as bibliotecas de tolerância a falhas no nível do aplicativo em situações em que seus comportamentos não estão em conflito.

11. Integração entre serviços

11.1. 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.

11.2. 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.

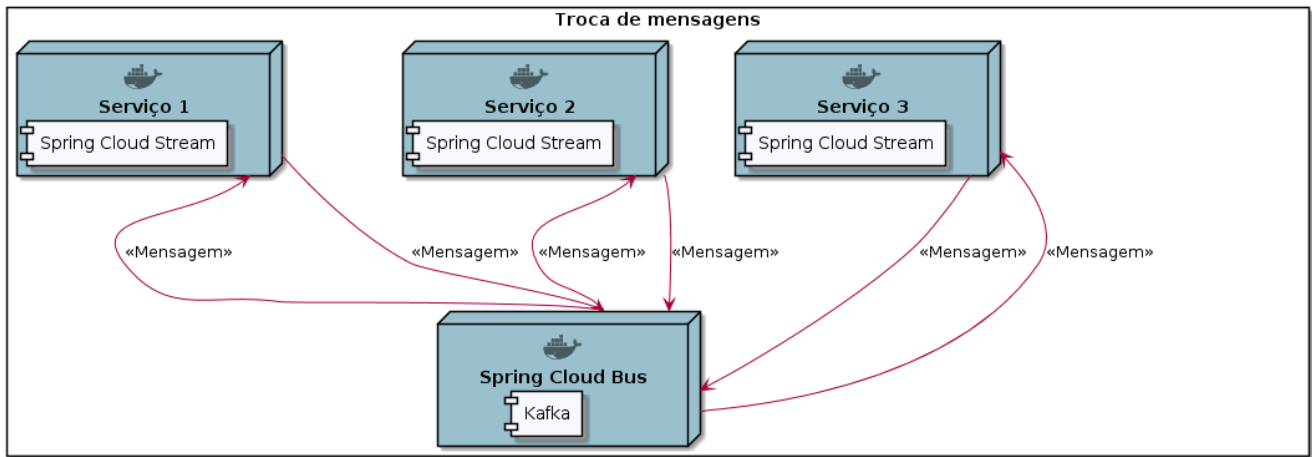


Figura 5. Troca de mensagens

11.2.1. Kafka

Apache Kafka é uma plataforma de streaming de eventos distribuídos de código aberto usada por milhares de empresas para pipelines de dados de alto desempenho, análise de streaming, integração de dados e aplicativos de missão crítica.

12. 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 de 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".

Você usa um gateway para gerenciar o tráfego de entrada e saída do service mesh. Você pode gerenciar vários tipos de tráfego com um gateway.

Os gateways são implementados no openshift pela configuração do `routes` e `services` para cada frontend.

A segurança da aplicação será realizada utilizando o RHSSO com protocolo OpenID no Serviço de Autenticação. O Serviço de autenticação irá realizar a comunicação com o RHSSO 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.

13. 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.

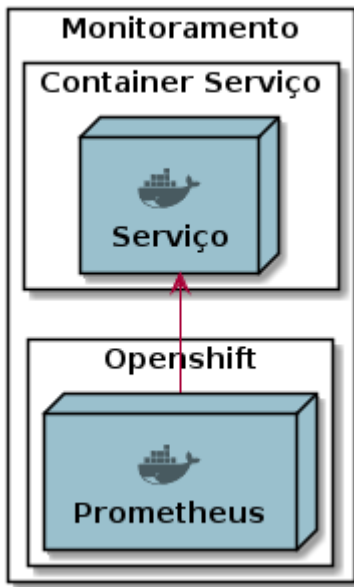


Figura 6. Monitoramento e Rastreamento

13.1. Monitoramento

13.1.1. 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 mínimo 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.

13.2. Rastreo 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.

13.2.1. Zipkin

Na arquitetura proposta, o Zipkin é a ferramenta usada para fazer o tracing, ela lê os dados adicionados nas requisições tramitadas pelo Istio Envoy permitindo identificar unicamente cada requisição à medida que a requisição passa de serviço a serviço novas informações são adicionadas. Ela é usada para entender os dados coletados, fornecendo meios de analisar e pesquisar dados. Através dessa análise é possível identificar áreas onde as melhorias devem ser aplicadas.

14. Cache Distribuído

O Redis é um **in-memory data structure store** de código aberto (licenciado pelo BSD) usado como banco de dados, cache, agente de mensagens e mecanismo de streaming. O Redis fornece estruturas de dados como strings, hashes, listas, conjuntos, conjuntos classificados com consultas de intervalo, bitmaps, hyperloglogs, índices geoespaciais e fluxos. O Redis possui replicação integrada, script Lua, remoção de LRU, transações e diferentes níveis de persistência em disco e fornece alta disponibilidade via Redis Sentinel e particionamento automático com Redis Cluster.

15. 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 recente 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 detalhado dois dos principais conceitos utilizados pelo Angular, o conceito de SPA (single page apps) e PWA (progressive web apps).

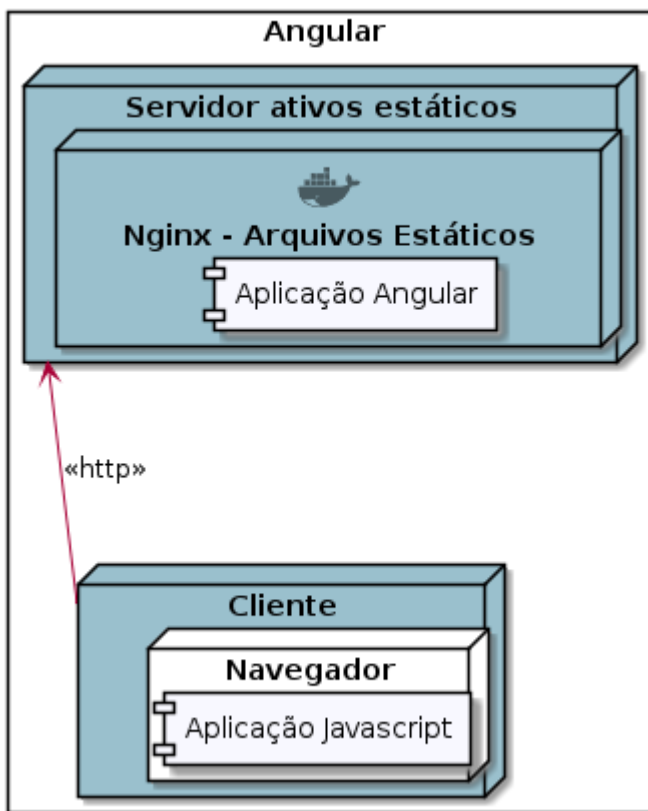


Figura 7. Aplicação Angular

15.1. 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-se 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.

15.2. 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 do 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 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 o acesso offline. Permitindo um controle de como a aplicação web é instalada, carregada e atualizada.

16. 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.

16.1. 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` conterà 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.

16.2. 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 branches 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.

16.3. Inspeção contínua de código

O SonarQube é uma ferramenta que possibilita o processo de inspeção continua 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 continua de código e de implantação contínua de software. Seus indicadores deverão ser seguidos de acordos com as definições contratuais.

O código fonte deverá seguir as conversões de nome e formatação de código do Java e do angular.

16.4. Implantação contínua de software

Argo CD é uma ferramenta declarativa de entrega contínua GitOps para Kubernetes.

Definições, configurações e ambientes de aplicativos devem ser declarativos e controlados por versão. A implantação de aplicativos e o gerenciamento do ciclo de vida devem ser automatizados, auditáveis e fáceis de entender.