



UNIVERSIDADE
LUSÓFONA

Engenharia de Microsserviços

Trabalho Final de curso

Relatório Final

Marcelo Garcia Domingues

Orientador: Prof. Pedro de Almeida Perdigão

Trabalho Final de Curso | Licenciatura em Engenharia Informática | 28-06-2024

Direitos de cópia

Engenharia de Microsserviços, Copyright de Marcelo Garcia Domingues, Universidade Lusófona.

A Escola de Comunicação, Arquitetura, Artes e Tecnologias da Informação (ECATI) e a Universidade Lusófona (UL) têm o direito, perpétuo e sem limites geográficos, de arquivar e publicar esta dissertação através de exemplares impressos reproduzidos em papel ou de forma digital, ou por qualquer outro meio conhecido ou que venha a ser inventado, e de a divulgar através de repositórios científicos e de admitir a sua cópia e distribuição com objectivos educacionais ou de investigação, não comerciais, desde que seja dado crédito ao autor e editor.

Resumo

A rápida evolução da tecnologia e das práticas de desenvolvimento transformou a maneira como concebemos e construímos aplicações. No centro dessa transformação encontra-se a arquitetura de microsserviços, que revolucionou a abordagem ao desenvolvimento de software. Ao invés de criar aplicações monolíticas complexas de manter e de escalar, os microsserviços permitem a divisão de uma aplicação em pequenos serviços autônomos, cada um com uma função específica, que operam de forma independente e se integram harmoniosamente.

Este Trabalho Final de Curso (TFC) tem como objetivo principal destacar a importância e os benefícios da arquitetura de microsserviços. No contexto desta abordagem, surge uma solução/framework que visa acelerar o desenvolvimento de soluções com uma infraestrutura de microsserviços. Esta solução representa uma contribuição significativa para a aceleração do desenvolvimento em infraestruturas de microsserviços, permitindo que os programadores se concentrem no desenvolvimento essencial, em vez de perder tempo em configurações detalhadas ou código não essencial.

A arquitetura proposta é dividida em três módulos distintos: Core, Rest e Rest API, todos desenvolvidos na sólida plataforma Java, com o apoio do Spring Framework. A escolha do Java e Spring deve-se à sua robustez, maturidade e à extensa comunidade de programadores, garantindo eficiência e segurança ao longo do processo de desenvolvimento.

O módulo Core representa o cerne da arquitetura, abrigando toda a lógica dos microsserviços, sendo que é aqui estão localizados os serviços, entidades, repositórios e outros componentes vitais para o funcionamento da aplicação. Por sua vez, o módulo Rest corresponde à camada de apresentação, englobando todos os controladores e configurações relevantes.

A grande inovação desta arquitetura reside no módulo Rest API, que utiliza as especificações em YAML para construção do Swagger UI (Open API Specification). Com recurso a este ficheiro YAML, conseguimos gerar automaticamente uma parte substancial do código, simplificando e acelerando o desenvolvimento. A geração automática de código, embora possa suscitar ceticismo, mantém o controlo nas mãos dos programadores, permitindo personalizações e ajustes conforme necessário.

Em suma, este relatório destaca a importância e os benefícios da arquitetura de microsserviços, mostrando como essa abordagem inovadora tem o poder de revolucionar o desenvolvimento de microsserviços, tornando-o mais ágil, organizado e eficaz. A solução apresentada fornece uma estrutura completa que visa acelerar o desenvolvimento de qualquer projeto backend em Spring. Os programadores podem definir o modelo YAML a priori, e o código é gerado automaticamente, permitindo que se concentrem no desenvolvimento essencial. Esta abordagem proporciona uma vantagem significativa no desenvolvimento de microsserviços e destaca a importância de adotar uma arquitetura que permite uma escalabilidade, flexibilidade e manutenção eficazes em projetos de desenvolvimento de software.

Abstract

The rapid evolution of technology and development practices has transformed the way we design and build applications. At the heart of this transformation is microservice architecture, which has revolutionized the approach to software development. Instead of creating monolithic applications that are complex to maintain and scale, microservices allow an application to be divided into small autonomous services, each with a specific function, which operate independently and integrate harmoniously.

The main aim of this Final Course Work (FCT) is to create a structure that speeds up the development of solutions with a microservices infrastructure. With this, programmers can concentrate on the core of development, while the infrastructure deals with the complexities of configuration.

The proposed architecture is divided into three distinct modules: Core, Rest and Rest API, all developed on the solid Java platform, with the support of the Spring Framework. Java and Spring were chosen because of their robustness, maturity and extensive community of programmers, guaranteeing efficiency and security throughout the development process.

The Core module represents the heart of the architecture, housing all the microservices logic, and this is where the services, entities, repositories and other vital components for the operation of the application are located. In turn, the Rest module corresponds to the presentation layer, encompassing all the relevant controllers and configurations.

The great innovation of this architecture lies in the Rest API module, which uses YAML specifications to build the Swagger UI (Open API Specification). Using this YAML file, we can automatically generate a substantial part of the code, simplifying and speeding up development. Automatic code generation, although it may arouse skepticism, keeps control in the hands of the programmers, allowing customizations and adjustments as necessary.

In short, this report clearly and in detail covers the potential of this microservices architecture, exploring the various challenges faced, the best practices adopted and the practical application in real-world scenarios. This will demonstrate how this innovative approach has the power to revolutionize microservice development, making it more agile, organized and effective. This framework makes a significant contribution to speeding up development on microservice infrastructures, allowing programmers to focus on core development instead of wasting time on detailed configurations or non-essential code (controllers, for example).

Índice

Resumo	iii
Abstract	iv
Índice	v
Lista de Figuras	viii
Lista de Tabelas	xiii
1 Identificação do Problema	1
1.1 Complexidade do Desenvolvimento	2
1.2 Consistência e Alinhamento	4
1.3 Eficiência do Desenvolvimento e Time-to-Market	5
1.4 Manutenção	6
1.4.1 Manutenção em Microsserviços: Desafios e Soluções	6
1.4.2 Integração de Ferramentas de Gestão de Projeto e DevOps	7
2 Viabilidade e Pertinência	9
2.1 Viabilidade do Projeto	9
2.1.1 Critérios econometrícios	9
2.1.2 Sustentabilidade de microsserviços	10
2.1.3 Envolvimento	10
2.1.4 Documentação	11
3 Benchmarking	12
3.1 Identificação de Soluções Existentes em Mercado	12
3.1.1 Arquiteturas Monolíticas e Arquiteturas Orientadas a Serviços (SOA)	13
3.2 Comparação Detalhada entre Arquiteturas	16
3.2.1 Arquiteturas Monolíticas	17
3.2.2 Arquiteturas Arquiteturas Orientadas a Serviços (SOA)	17
3.2.3 Arquitetura Proposta	17
3.2.4 Comparação Detalhada das Arquiteturas	18
4 Engenharia	23
4.1 Levantamento e Análise dos Requisitos	23
4.1.1 Requisitos Funcionais	23
4.1.2 Requisitos Não Funcionais	27

4.2	Estrutura	31
5	Solução Proposta	33
5.1	Introdução	33
5.1.1	Divisão em Módulos	33
5.2	Arquitetura	35
5.3	Tecnologias e Ferramentas Utilizadas	35
5.3.1	Porque Java com Spring?	35
5.3.2	Porquê Java e não Python, Node.js ou outras linguagens?	36
5.3.3	Possibilidade extra de implementar em .Net?.....	36
5.3.4	Porquê REST e não SOAP?	36
5.4	Tipos de Segurança Implementados e Recomendados	38
5.4.1	Autenticação e Autorização com JSON Web Tokens (JWT)	39
5.4.2	Proteção contra Ataques Comuns	40
5.4.3	Controlo de Acesso e Autorização Granular	41
5.4.4	Monitoramento e Logs de Segurança.....	41
5.5	Utilização de API Gateaway	42
5.6	Utilização de Load Balancer	43
5.7	Utilização de Apache Kafka	44
5.8	Implementação.....	45
5.9	Abrangência	46
5.10	Benefícios e Possíveis Desafios da Solução.....	47
5.10.1	Benefícios.....	47
5.10.2	Possíveis desafios (não benefícios):.....	49
6	Método e Planeamento	51
6.1	Plano de Trabalho e Cronograma	51
6.2	Progresso do Trabalho até o Momento	52
6.3	Dificuldades e Alterações ao Plano Inicial	53
7	Notas Finais.....	55
8	Calendário.....	56
	Bibliografia	57
	Glossário	60

Listas de Figuras

Figura 1, intitulada "Fluxo de Comunicação Cliente-Servidor", ilustra a interação entre um cliente, uma API e um servidor de API. O cliente envia uma requisição, que é processada pela API e encaminhada ao servidor API, que por sua vez, retorna uma resposta.....	1
Figura 2, intitulada "Integração e Entrega Contínuas (CI/CD)", mostra dois laços entrelaçados representando as práticas de CI e CD no desenvolvimento de software. A partir do planejamento, segue-se para codificação, construção e teste contínuo, formando o ciclo de CI, enquanto o CD abrange o lançamento, a implementação, a operação e o monitoramento.	2
Figura 3, intitulada "Digital Transformation", ilustra os componentes fundamentais do processo de digitalização através de ícones representativos para Tecnologia, Comunicação, Dados, Internet das Coisas, Automação, AI e Networking, dispostos sob um título centralizado.	2
Figura 4, intitulada "Exemplo de Arquitetura em Microsserviços", demonstra a estrutura de uma aplicação dividida em microsserviços. Nela, aplicações cliente como Web e Mobile interagem com um conjunto de microsserviços através de um API Gateway centralizado. Os serviços incluem Catálogo, Carrinho de Compras, Descontos e Pedidos, cada um conectado a sua própria base de dados.	3
Figura 5, intitulada "Produtividade vs. Complexidade em Arquiteturas de Software", mostra um gráfico comparativo. Nele, duas curvas se cruzam: a superior, representando microsserviços, e a inferior, representando monolitos, demonstrando como a produtividade varia com o aumento da complexidade do sistema.....	4
Figura 6, intitulada "Ciclo de Desenvolvimento", exibe o modelo DevOps em forma de um infinito, detalhando o ciclo de vida do desenvolvimento de software desde o planejamento até a operação. As fases incluem Codificação, Construção, Teste, Lançamento, Implementação, Operação e Monitoramento, destacando a integração e entrega contínuas.	5
Figura 7, intitulada "Gestão de Branches no Controlo de Versões", ilustra uma linha do tempo de controlo de versão com branches para diferentes funcionalidades. O branch principal, 'Master', tem dois branches secundários: um menor para 'Little Feature' e um mais longo para 'Big Feature', indicando diferentes escalas de desenvolvimento de recursos.	7
Figura 8, intitulada "Interface do Software de Gestão de Projetos, Jira", exibe a tela de backlog de um sistema de gestão de projetos e tarefas, mostrando uma lista de tarefas planeadas, insights de backlog, e o progresso do sprint atual com diversas tarefas categorizadas por status de desenvolvimento.	7
Figura 9, intitulada "Comparação de Custo e Comunicação em Arquiteturas de Software", mostra dois gráficos: o primeiro compara o custo de manutenção entre arquiteturas monolíticas e de microsserviços conforme aumentam as linhas de código; o segundo ilustra a complexidade da comunicação em equipas de desenvolvimento, destacando a eficiência dos microsserviços em relação aos monolíticos com o crescimento do número de programadores.	9
Figura 10, intitulada "Fluxos de Desenvolvimento: Monólito vs. Microsserviços", contrasta as trajetórias de lançamento de um produto entre as duas arquiteturas. No monólito, a dependência entre as equipas leva a um ponto único de falha, enquanto que, nos microsserviços, cada equipa tem um caminho direto e independente para a produção.	10
Figura 11, intitulada "Exemplo de Documentação Java", exibe um código fonte Java onde a documentação Javadoc é utilizada para descrever a função de um método. Destaca-se a anotação @param para o parâmetro primaryStage e @throws Exception, indicando as exceções que o método pode lançar.	11
Figura 12, intitulada "Exemplo de Documentação no Confluence", mostra uma página do Confluence com de código que exemplifica como criar um macro personalizado para um painel de utilizador, incluindo parâmetros para cor do título, estilo de borda e cor de fundo.	11

<i>Figura 13, intitulada "Múltiplas Arquiteturas de Serviços", compara visualmente três abordagens arquiteturais em sistemas de IT: Monolítica, Orientada a Serviços e Microsserviços, destacando a estrutura de UI, lógica de negócios, serviços e bases de dados em cada modelo.</i>	12
<i>Figura 14, intitulada "Arquitetura de E-commerce com Load Balancer", mostra um diagrama de sistema com clientes conectando-se a uma aplicação de e-commerce através de um balanceador de carga, que distribui os pedidos para vários serviços, como UI da loja, serviço de catálogo, serviço de carrinho de compras, serviço de descontos e serviço de pedidos, todos interagindo com um sistema de banco de dados relacional (RDBMS).</i>	13
<i>Figura 15, intitulada "Arquitetura Orientada a Serviços (SOA)", ilustra uma configuração SOA com um grupo de consumidores (clientes) a aceder a serviços de CRM, gestão de pedidos e faturamento através de um Entrepise Service Bus (ESB). Esses serviços compartilham recursos com uma base de dados relacionais, indicando uma arquitetura que promove a reutilização e a integração de sistemas.</i>	14
<i>Figura 16, intitulada "Arquitetura/Framework Apresentada", exibe um diagrama de arquitetura de software que detalha a interação entre componentes gerados automaticamente e lógica de negócios centralizada. O diagrama divide-se em três camadas principais: OAS (Open API Specification), REST e CORE, com ênfase na separação de objetos gerados, implementação de controladores e serviços, e repositórios, caso JPA (Java Persistence API) seja utilizado.</i>	15
<i>Figura 17, intitulada "Panorama dos Estilos de Arquitetura de Software", apresenta um infográfico detalhado que descreve e ilustra variados estilos de arquitetura de software. No centro, um círculo categoriza os estilos arquiteturais, que incluem CQRS, que separa operações de leitura e escrita; Microsserviços, que divide uma aplicação em pequenos serviços; e MVP, baseado no padrão Model-View-Controller. Ao redor, exemplos visuais e descrições fornecem informações sobre a Arquitetura em Camadas, Microkernel, Event-Driven e DDD, entre outros, cada um com sua abordagem única para organizar o fluxo de dados e a lógica de negócios em sistemas de software. A imagem é enriquecida com anotações que explicam os princípios de cada estilo, como a separação de preocupações, e mostra como eles podem ser aplicados para resolver diferentes problemas de design de sistemas.</i>	16
<i>Figura 18, intitulada "Comparativo de Desempenho: Microsserviços vs. Monolíticos", exibe um gráfico de linhas que compara o tempo de execução em milissegundos de arquiteturas de microsserviços e monolíticas em relação à porcentagem de concorrência de chamadas. As linhas demonstram que, enquanto o desempenho dos microsserviços permanece consistentemente baixo em diferentes níveis de concorrência, o monolítico exibe um tempo de execução inicialmente alto que rapidamente se estabiliza à medida que a concorrência aumenta.</i>	22
<i>Figura 19, intitulada "Estrutura de Dados JSON", mostra um exemplo de arquivo JSON contendo uma lista de utilizadores com atributos como nome de utilizador, função, localização e data. Cada utilizador está identificado por uma chave única e os dados estão organizados em um formato hierárquico e legível.</i>	23
<i>Figura 20, intitulada "Exemplo de Implementação de Logging em Java", exibe um trecho de código Java que define constantes estáticas para diferentes níveis de log (como DEBUG e WARN) e métodos para registrar mensagens de debug e aviso, utilizando a biblioteca de logging do Java.</i>	24
<i>Figura 21, intitulada "Arquitetura da Estrutura de Logging", exibe um diagrama explicativo da hierarquia e do funcionamento da estrutura de logging em Java. O diagrama mostra a relação entre o LogManager, Loggers, LogRecords e Handlers, além dos níveis de severidade e os componentes de filtragem e formatação de logs, como SimpleFormatter e XMLFormatter.</i>	24
<i>Figura 22, intitulada "Arquitetura de Exceções e Erros", apresenta um diagrama de hierarquia de exceções em Java, destacando a classe Throwable e suas subclasses: Error e Exception, assim como as várias categorias de exceções e erros, incluindo Runtime Exception, IO Exception, e diferentes tipos de Error, como VM Error e Assertion Error.</i>	25

- Figura 23, intitulada "Exemplo de Implementação de Tratamento de Exceções em Java", exibe um bloco de código com uma estrutura de tratamento de exceções aninhadas. O código tenta executar uma divisão por zero, que é capturada pelo catch interno como uma ArithmeticException, seguida por blocos finally que são executados independentemente da ocorrência de exceções..... 25
- Figura 24, intitulada "Exemplo de um Ficheiro YAML", exibe um documento YAML contendo dados pessoais estruturados, como nome, idade, endereço e números de telefone, seguindo o formato de chave-valor característico do YAML para facilitar a legibilidade e o mapeamento de dados. 26
- Figura 25, intitulada "Exemplo de Especificação OpenAPI em YAML", mostra um trecho de um documento YAML utilizado para definir uma interface de programação de aplicações (API) com o OpenAPI Specification (OAS). Especifica um endpoint GET que retorna uma lista de animais de estimação, detalhando os tipos de mídia produzidos, os parâmetros de consulta aceitos e a estrutura da resposta esperada..... 26
- Figura 26, intitulada "Escalabilidade: Vertical vs. Horizontal", ilustra as duas abordagens comuns para escalar sistemas de computação: o escalonamento vertical, que aumenta o poder de processamento dentro de uma única máquina, e o escalonamento horizontal, que adiciona mais máquinas ao sistema para expandir a capacidade de processamento. 27
- Figura 27, intitulada "Fluxo de Validação de Token JWT", mostra um diagrama de como um cliente e um servidor interagem ao usar um JSON Web Token (JWT) para autenticação e autorização. O processo começa com o cliente a fazer um login e recebe um JWT, continua com o cliente a enviar uma pedido com o JWT para um recurso protegido e termina com o servidor a validar o JWT antes de fornecer acesso ao recurso..... 28
- Figura 28, intitulada "Melhores Práticas para Segurança em Microsserviços", ilustra um conjunto de estratégias recomendadas para garantir a segurança em uma arquitetura de microsserviços. As práticas incluem o uso de um gateway de API, a verificação de dependências, a implementação de autenticação multifator (MFA), a adoção de uma estratégia de defesa robusta, a aplicação de segurança em nível de serviço, seguir a abordagem DevSecOps e evitar escrever o próprio código de criptografia. 29
- Figura 29, intitulada "Características da Computação em Nuvem", destaca os principais atributos e vantagens da computação em nuvem, incluindo Agrupamento de Recursos, Sistema Automático, Economia, Segurança, Pagamento pelo Uso e Serviço Mensurado, juntamente com a Manutenção Fácil, Amplo Acesso a Rede e Disponibilidade. Estes elementos são fundamentais para entender o valor agregado e a eficiência operacional que a computação em nuvem oferece às organizações. 30
- Figura 30, intitulada "9 Tipos de Teste de API", enumera e explica as várias abordagens de teste essenciais para assegurar a robustez e segurança das APIs. Estes incluem smoke testing para verificar funcionalidades essenciais, testes funcionais e de integração para conformidade com requisitos, testes de regressão para estabilidade após atualizações, e testes de carga e stress para avaliar o desempenho sob alta demanda. Além disso, destaca-se a importância dos testes de segurança, de interface do utilizador e de fuzzing para garantir uma interação segura e eficiente com as APIs. 31
- Figura 31, intitulada "Editor Swagger", mostra a interface do utilizador do Swagger Editor com uma especificação OpenAPI exibida no lado esquerdo e a visualização interativa da documentação da API no lado direito. Esta ferramenta é frequentemente utilizada para projetar, editar e documentar APIs RESTful. 34
- Figura 32, intitulada "Estrutura Modular de API", mostra a organização de uma API em três módulos principais: OAS Module, Rest Module e Core Module. O OAS Module contém a definição da OpenAPI, incluindo o arquivo YAML para a interface do utilizador do Swagger. O Rest Module inclui controladores, conversores e configurações diversas. O Core Module é onde a lógica dos serviços é implementada, com interfaces e implementações de serviços, destacando a separação clara das responsabilidades dentro da arquitetura da API. 35

-
- Figura 33, intitulada "Evolução e Comparação de Protocolos de Comunicação", exibe uma linha do tempo da evolução dos protocolos de comunicação em sistemas distribuídos, começando com CORBA em 1991 e passando por RDA, SOAP, XML-RPC, REST, OData, até chegar ao gRPC e GraphQL. A tabela abaixo compara as características de SOAP, REST, GraphQL e RPC em termos de organização, formato, curva de aprendizagem, comunidade e casos de uso típicos, destacando as diferenças fundamentais e a adequação de cada protocolo a diferentes necessidades de aplicação.* 36
- Figura 34, intitulada "Operações CRUD", apresenta os ícones e as siglas das quatro operações básicas de interação com bancos de dados e sistemas de armazenamento de dados: Criação (Create), Leitura (Read), Atualização (Update) e Exclusão (Delete). Estas operações formam o acrônimo CRUD e são fundamentais para o desenvolvimento de aplicações que necessitam gerir dados de forma eficiente.* 37
- Figura 35, intitulada "Processo de Handshake SSL/TLS (HTTPS)", ilustra a sequência de passos para estabelecer uma conexão segura entre um cliente e um servidor. O diagrama detalha o aperto de mão TCP inicial, a verificação do certificado do servidor, a troca de chaves para estabelecer uma chave de sessão criptografada e finalmente a transmissão de dados usando criptografia simétrica, tudo isso dentro do contexto do uso de chaves públicas e privadas para assegurar a comunicação.* 38
- Figura 36, intitulada "Melhores Práticas de Segurança para APIs", destaca várias estratégias recomendadas para garantir a segurança em APIs. Estas incluem a utilização de HTTPS para conexões seguras, a implementação de OAuth2 para autorização, o emprego de WebAuthn para autenticação, a utilização de chaves de API com diferentes níveis de acesso, limitação de taxa de requisições para prevenir abuso, versionamento de API para gerir mudanças, uso de listas de permissões para controlo de acesso, revisão de riscos de segurança seguindo as diretrizes do OWASP, tratamento apropriado de erros sem revelar informações sensíveis, e a validação de entradas para prevenir injeções e outros ataques. Além disso, sugere-se o uso de gateways de API como um ponto de controlo para implementar várias dessas práticas de segurança.* 39
- Figura 37, intitulada "Estrutura de um JSON Web Token (JWT)", ilustra os três componentes principais de um JWT: Cabeçalho (Header), Carga Útil (Payload) e Assinatura (Signature). O cabeçalho especifica o algoritmo de codificação, a carga útil contém as reivindicações ou afirmações do token e a assinatura é usada para verificar se o token não foi alterado.* 40
- Figura 38, intitulada "Tipos de Ataques Cibernéticos", mostra um gráfico com diversos tipos de ameaças digitais irradiando de uma figura central representando um hacker. As ameaças incluem Malware, Ransomware, Ataques Man-in-the-Middle, Explorações de Dia Zero, Tunelamento DNS, Injeção de SQL, Ataques XSS, Phishing, Ataques DoS e DDoS, Engenharia Social e Cryptojacking.* 41
- Figura 39, intitulada "Logging Dashboard com Splunk", exibe um dashboard de monitoramento que proporciona insights sobre o desempenho de um servidor web. À esquerda, o painel mostra o número de solicitações bem-sucedidas com os principais contribuintes e os recursos mais requisitados. À direita, detalha o número de solicitações mal-sucedidas, novamente com os principais contribuintes e os recursos que geraram as maiores quantidades de falhas, complementado por tabelas detalhadas e métricas de resposta.* 42
- Figura 40, intitulada "Componentes de um Gateway de API", ilustra como um gateway de API funciona como um ponto central entre os clientes - incluindo web, dispositivos móveis e PCs - e um conjunto de microsserviços. O diagrama destaca funções críticas do gateway, como validação de parâmetros, autenticação, limitação de taxa, descoberta de serviços, roteamento dinâmico, conversão de protocolo, tratamento de erros, interrupção de circuito, monitoramento de logs e cache, com integrações para serviços externos como Elasticsearch e Redis.* 43
- Figura 41, intitulada "Fluxo de Tráfego com Proxy na Internet", mostra um diagrama de rede que descreve como os pedidos dos clientes são direcionados através da Internet para um servidor proxy,*

<i>que então encaminha essos pedidos para os servidores de origem. Este diagrama ilustra o papel de um proxy como intermediário no processamento de pedidos de rede.</i>	44
<i>Figura 42, intitulada "Integração com Apache Kafka", ilustra como o Apache Kafka atua como uma plataforma central para processamento de streams de dados, integrando aplicações web, apps móveis, microsserviços, sistemas de monitoramento e soluções de análise de dados. O diagrama também mostra Kafka a receber dados de várias fontes como IoT, CRMs, bancos de dados, caches e data lakes.</i>	45
<i>Figura 43, intitulada "Princípios de Design de Microsserviços", apresenta um infográfico com nove práticas recomendadas para a arquitetura de microsserviços. Entre elas estão a separação de armazenamento de dados, manter um nível de maturidade de código semelhante entre os serviços, construções separadas para cada serviço, princípio de responsabilidade única, implementação em contêineres, tratar servidores como stateless, design orientado a domínio, micro frontend e orquestração de microsserviços com ferramentas como Kubernetes.</i>	48
<i>Figura 44, intitulada "Arquitetura de Aplicação no Azure com Kubernetes", exibe um diagrama a detalhar a integração de vários serviços do Microsoft Azure para operar uma aplicação com contêineres, com recurso ao Azure Kubernetes Service (AKS). Destaca-se a orquestração do CI/CD com Azure Pipelines, o load balancer, a gestão de identidades com o Azure Active Directory, e a segurança com o Azure Key Vault, além do monitoramento e armazenamento de dados com Azure Monitor e bases de dados SQL e Cosmos DB.</i>	50
<i>Figura 45, intitulada "Cronograma em Gantt, realizado em Excel", apresenta um gráfico de Gantt colorido detalhando o cronograma de tarefas e marcos de um projeto. As tarefas são divididas em categorias como Identificação do Problema, Benchmarking, Viabilidade e Pertinência, entre outras, com colunas indicando a pessoa responsável, datas de início e fim, e a duração representada por barras coloridas que se estendem ao longo de um calendário na parte superior.</i>	56

Lista de Tabelas

- Tabela 1, intitulada "Comparação de Arquiteturas de Software", compara critérios como Escalabilidade, Manutenção, Flexibilidade, Complexidade, Eficiência de Desenvolvimento e Autonomia de Equipes entre Arquiteturas Monolíticas, Arquiteturas Baseadas em Serviços e Novas Arquiteturas de Microsserviços. A tabela destaca as vantagens e desvantagens de cada abordagem, evidenciando como as arquiteturas de microsserviços oferecem maior flexibilidade, eficiência e autonomia em comparação com as abordagens mais tradicionais.* 18
- Tabela 2, intitulada "Avaliação de Arquiteturas de Software", compara as Arquiteturas Monolíticas, Arquiteturas Baseadas em Serviços e Novas Arquiteturas de Microsserviços em termos de Desempenho, Segurança, Resiliência, Escalabilidade Horizontal, Facilidade de Deploy e Eficiência de Recursos. A tabela indica que as arquiteturas monolíticas não atendem a esses critérios tão eficientemente quanto as abordagens baseadas em serviços e as arquiteturas de microsserviços, que afirmam ter um desempenho superior em todos esses aspectos importantes.* 18
- Tabela 3, intitulada "Capacidades Operacionais em Arquiteturas de Software", contrasta Arquiteturas Monolíticas com Arquiteturas Baseadas em Serviços e Novas Arquiteturas de Microsserviços em relação a critérios como Facilidade de Testes, Tratamento de Erros, Monitoramento, Rollback e Tempo de Recuperação de Falhas. A tabela sugere que, enquanto as arquiteturas monolíticas apresentam limitações nesses aspectos, as arquiteturas baseadas em serviços e de microsserviços são projetadas para superar essas limitações, oferecendo melhorias significativas em termos de manutenção operacional e resiliência.* 20
- Tabela 4, intitulada "Tempo de Respostas e Latências", compara Arquiteturas Monolíticas, Arquiteturas Baseadas em Serviços e Novas Arquiteturas de Microsserviços com base em métricas de desempenho como tempo de resposta médio, mínimo e máximo, latência média, mínima e máxima, e taxas de transferência média, mínima e máxima. Os dados indicam que as Novas Arquiteturas de Microsserviços proporcionam os melhores tempos de resposta e as menores latências, seguidas pelas Arquiteturas Baseadas em Serviços, enquanto as Arquiteturas Monolíticas apresentam os valores mais altos nessas métricas.* 21

1 Identificação do Problema

À medida que as organizações se aventuram na rotina da inovação e transformação no desenvolvimento de software, a adesão à arquitetura de microsserviços apresenta-se como uma opção fulcral. Esta transição, contudo, não é isenta de desafios substanciais que exigem uma análise detalhada e uma abordagem técnica refinada para serem superados. A mudança de sistemas monolíticos tradicionais para uma perspetiva assente em microsserviços é impulsionada pela busca incessante por agilidade, escalabilidade e adaptabilidade. Todavia, esta mudança traz consigo desafios que impactam a capacidade das organizações em manter-se à tona num mercado em constante evolução tecnológica.

Os desafios supracitados não se limitam a ajustes na superfície. Eles infiltram-se nas estruturas de desenvolvimento, moldando as metodologias de trabalho e a cultura organizacional. As APIs (*Application Programming Interfaces*), que são conjuntos de rotinas e padrões estabelecidos por software para a utilização das suas funcionalidades por aplicações que não pretendem envolver-se em detalhes da implementação do software, tornam-se essenciais nesta transição. A adoção desta arquitetura *demand* atenção meticolosa em todos os estágios do SDLC (Software Development Life Cycle), abrangendo desde a fase de planeamento até ao lançamento e manutenção.

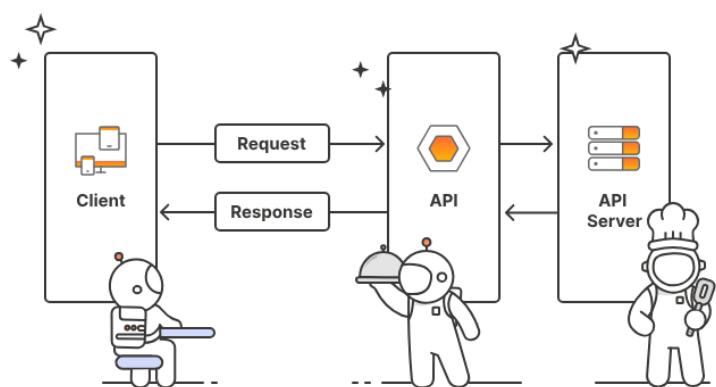


Figura 1, intitulada "**Fluxo de Comunicação Cliente-Servidor**", ilustra a interação entre um cliente, uma API e um servidor de API. O cliente envia uma requisição, que é processada pela API e encaminhada ao servidor API, que por sua vez, retorna uma resposta.

Dentro deste panorama, as organizações são confrontadas com a imperativa necessidade de uma renovação, tanto ao nível cultural como tecnológico. A agilidade trazida pelos microsserviços vai além da simples adoção de uma nova tecnologia; ela representa uma revolução nas práticas estabelecidas. DevOps, uma metodologia que integra desenvolvimento e operações de IT para acelerar o processo de desenvolvimento e proporcionar alta qualidade, surge como uma filosofia essencial para as empresas que desejam otimizar seus fluxos de trabalho.

Para além disso, a orquestração eficiente de microsserviços exige ferramentas avançadas, como o Kubernetes, um sistema de orquestração de containers de código aberto, assegurando que estes serviços autónomos coexistam e colaborem de forma coesa para servir o propósito global da aplicação. Implementações bem-sucedidas requerem monitorização contínua, CI/CD (Integração Contínua/Entrega Contínua) e uma abordagem proativa para detetar e remediar falhas.

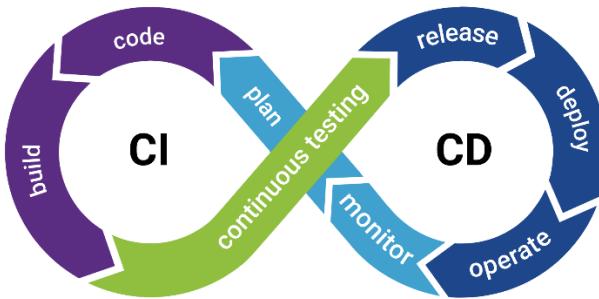


Figura 2, intitulada "**Integração e Entrega Contínuas (CI/CD)**", mostra dois laços entrelaçados representando as práticas de CI e CD no desenvolvimento de software. A partir do planejamento, segue-se para codificação, construção e teste contínuo, formando o ciclo de CI, enquanto o CD abrange o lançamento, a implementação, a operação e o monitoramento.

Em síntese, abraçar a arquitetura de microsserviços representa uma evolução não só a nível tecnológico, mas também organizacional. A capacidade de superar os desafios inerentes a esta transição determinará a posição competitiva das empresas num mundo digital em constante metamorfose.



Figura 3, intitulada "**Digital Transformation**", ilustra os componentes fundamentais do processo de digitalização através de ícones representativos para Tecnologia, Comunicação, Dados, Internet das Coisas, Automação, AI e Networking, dispostos sob um título centralizado.

1.1 Complexidade do Desenvolvimento

A complexidade do desenvolvimento em ambientes de microsserviços está intrinsecamente ligada à natureza fragmentada e distribuída dessa arquitetura. Quando as organizações optam por dividir as suas aplicações em microsserviços menores e independentes, acabam por criar uma rede de sistemas interligados, cada um com a sua própria base de código, estrutura de

dados e dependências. Isso aumenta exponencialmente a complexidade de manutenção e desenvolvimento.

Para enfrentar essa complexidade e tornar o desenvolvimento de microsserviços mais “controlável”, a Framework proposta tem como objetivo simplificar o desenvolvimento em ambientes de microsserviços, proporcionando assim, uma estrutura que lida com grande parte das tarefas complexas relacionadas à infraestrutura.

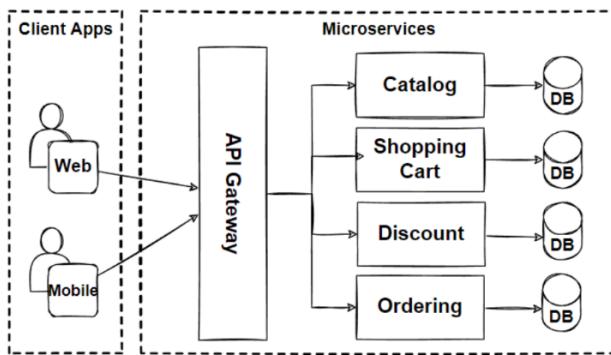


Figura 4, intitulada "Exemplo de Arquitetura em Microsserviços", demonstra a estrutura de uma aplicação dividida em microsserviços. Nela, aplicações cliente como Web e Mobile interagem com um conjunto de microsserviços através de um API Gateway centralizado. Os serviços incluem Catálogo, Carrinho de Compras, Descontos e Pedidos, cada um conectado a sua própria base de dados.

Além disso, a gestão de código, a gestão de dependências, a comunicação entre serviços e a garantia de consistência representam desafios significativos. Os programadores enfrentam o dilema de equilibrar a independência dos microsserviços, com a necessidade de manter o controlo centralizado, de forma a garantir a interoperabilidade e a padronização.

Em baixo, estão alguns dos principais aspectos dessa complexidade:

- **Gestão do código (Version Control):** Com vários microsserviços simultaneamente em desenvolvimento, a gestão de repositórios (Ex: GitHub, Azure DevOps, Jira) do código-fonte torna-se um desafio. Diferentes equipas podem estar a trabalhar em diferentes partes do microserviço, o que requer uma coordenação cuidadosa para evitar conflitos de código e incompatibilidades.
- **Gestão de dependências:** Cada microsserviço pode ter suas próprias dependências, incluindo bibliotecas, serviços externos e APIs. Coordenar e atualizar essas dependências de forma eficaz é uma tarefa complexa e muito trabalhosa.
- **Comunicação entre Serviços:** A comunicação entre microsserviços, seja por meio de chamadas de API ou de qualquer outra forma, deve ser robusta e confiável. A lógica do tratamento de erros, garantindo a entrega e mantendo a coesão entre os serviços, aumenta a complexidade.

- **Testes e debug:** com vários microsserviços interdependentes, executar testes eficazes de unidade (Unit Tests Ex: Junit, Mockito), integração (Ex: Karate) e end-to-end é um desafio. Problemas de debug que cruzam os limites do serviço podem ser demorados e complexos.

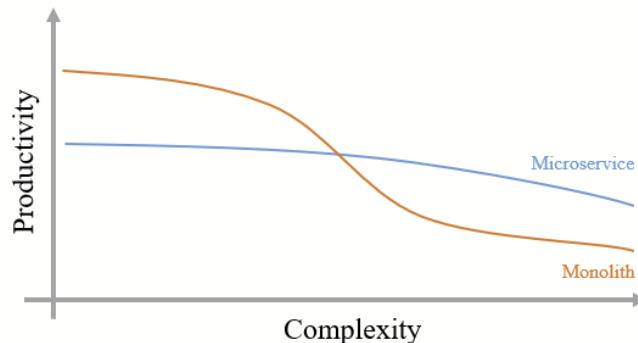


Figura 5, intitulada "**Produtividade vs. Complexidade em Arquiteturas de Software**", mostra um gráfico comparativo. Nele, duas curvas se cruzam: a superior, representando microsserviços, e a inferior, representando monolitos, demonstrando como a produtividade varia com o aumento da complexidade do sistema.

1.2 Consistência e Alinhamento

Manter a consistência e o alinhamento entre microsserviços é um fator crítico para garantir que a arquitetura de microsserviços funcione como um todo um só. Essa falta de alinhamento pode levar a problemas de interoperabilidade, dificuldades de escalabilidade e complexidades de manutenção.

A variação nas estruturas de dados, APIs e padrões de comunicação entre microsserviços pode resultar em dificuldades na integração e evolução dos sistemas. A falta de orientações claras pode conduzir a sistemas difíceis de compreender e manter, prejudicando a eficiência do desenvolvimento.

Como referido em cima, falta de coerência pode dar origem a uma série de problemas:

- **Estruturas de dados divergentes:** Se diferentes equipes ou programadores implementarem uma estruturas de dados diferentes para tarefas semelhantes, isso pode prejudicar a interoperabilidade e a comunicação entre microsserviços.
- **APIs discordantes:** APIs inconsistentes tornam-se mais difíceis para os consumidores de serviços. Isto pode resultar em confusão e ineficiência na integração.
- **Padrões de comunicação variados:** A falta de padrões de comunicação claros e compartilhados entre microsserviços pode resultar em incompatibilidades que dificultam a escalabilidade e a evolução da arquitetura.

1.3 Eficiência do Desenvolvimento e Time-to-Market

O desenvolvimento eficiente é essencial para atender aos pedidos dos clientes e ao mercado em constante evolução. No entanto, desenvolver, testar e implementar cada microsserviço individualmente, pode consumir recursos significativos. Isso pode resultar em atrasos no lançamento de novos recursos e atualizações, afetando a capacidade de resposta da organização.

Garantir a alocação eficaz de recursos e reduzir o tempo necessário para desenvolver e implementar novos microsserviços são imperativos para a agilidade e o sucesso dos negócios.

Manter a eficiência do desenvolvimento, mesmo quando a arquitetura de microsserviços cresce em complexidade, é um desafio constante.

No contexto dos microsserviços, a eficiência do desenvolvimento é afetada por vários fatores:

- **Custo de desenvolvimento:** desenvolver, testar e implementar cada microsserviço requer recursos significativos. As organizações precisam determinar orçamentos e equipes de forma a conseguir gerir o seu orçamento da melhor forma e a mais eficaz possível.
- **Tempo de desenvolvimento:** O tempo necessário para criar um novo microsserviço e colocá-lo em produção pode atrasar a entrega de novos recursos e funcionalidades, impactando a capacidade de resposta aos pedidos do cliente e do mercado.
- **Reutilização de código:** A falta de reutilização de código em microsserviços pode resultar em redundância e esforço duplicado, tornando o desenvolvimento menos eficiente.
- **Processos de Desenvolvimento Ágil:** A eficiência pode ser melhorada através da adoção de metodologias ágeis, como *Scrum* ou *Kanban*, que permitem uma resposta rápida à mudança e uma colaboração eficaz entre as equipas de desenvolvimento.
- **Automação da implementação:** A automação da implementação por meio de ferramentas como *Docker* e *Kubernetes* pode acelerar significativamente o ciclo de desenvolvimento, permitindo a implementação rápida e consistente de microsserviços.

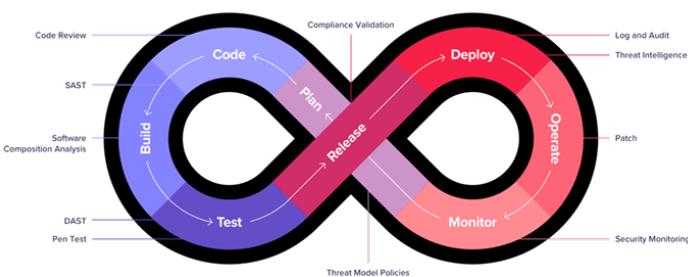


Figura 6, intitulada "*Ciclo de Desenvolvimento*", exibe o modelo DevOps em forma de um infinito, detalhando o ciclo de vida do desenvolvimento de software desde o planejamento até a operação. As fases incluem Codificação, Construção, Teste, Lançamento, Implementação, Operação e Monitoramento, destacando a integração e entrega contínuas.

A interseção entre a eficiência do desenvolvimento e o modelo DevOps é ilustrada na Figura 6, que representa o ciclo de vida do desenvolvimento de software. Este modelo enfatiza a integração contínua e a entrega contínua, realçando a importância de cada fase, desde a codificação até a operação e monitoramento.

Em conclusão, a framework proposta não apenas visa simplificar e acelerar o desenvolvimento de microsserviços, mas também enfatiza a alocação eficiente de recursos, a redução do tempo de desenvolvimento, e a adaptabilidade rápida às mudanças de mercado. Estes elementos são fundamentais para manter a competitividade e assegurar o sucesso contínuo nos negócios."

Este texto expandido fornece uma visão mais detalhada e abrangente sobre a eficiência no desenvolvimento de microsserviços, incorporando aspectos práticos e teóricos relevantes.

1.4 Manutenção

À medida que os requisitos de negócios evoluem, os microsserviços precisam ser constantemente atualizados e melhorados. Acompanhar todas as mudanças e garantir que os serviços continuam a funcionar de forma confiável é um desafio constante.

A gestão de versões, testes de regressão, coordenação de atualizações e gestão de todo o ciclo de vida de microsserviços são preocupações críticas. As organizações precisam garantir que a evolução dos microsserviços não introduza problemas de compatibilidade ou degradação do serviço, o que poderia afetar negativamente a experiência do utilizador.

Incorporando ferramentas como Jira e Azure DevOps no contexto de manutenção de microsserviços, o texto poderia ser expandido da seguinte forma:

1.4.1 Manutenção em Microsserviços: Desafios e Soluções

À medida que os requisitos de negócios evoluem, os microsserviços precisam de atualizações e melhorias contínuas. Manter-se a par dessas mudanças e assegurar o funcionamento confiável dos serviços é um desafio contínuo. A gestão de versões, testes de regressão, coordenação de atualizações e gestão do ciclo de vida completo dos microsserviços são preocupações cruciais. É vital garantir que a evolução dos microsserviços não introduza problemas de compatibilidade ou degrade o serviço, impactando negativamente a experiência do utilizador.

Com a constante evolução, surgem desafios específicos:

- **Gestão de Versões:**
 - Manter múltiplas versões de microsserviços e assegurar que mudanças não interrompam a funcionalidade existente é complexo.
- **Testes:**
 - Testes rigorosos são necessários para garantir que as atualizações não causem regressões ou problemas de compatibilidade.
- **Coordenação de Atualização:**
 - Coordenar atualizações simultâneas em vários microsserviços, especialmente quando existem dependências entre eles, pode ser complicado.

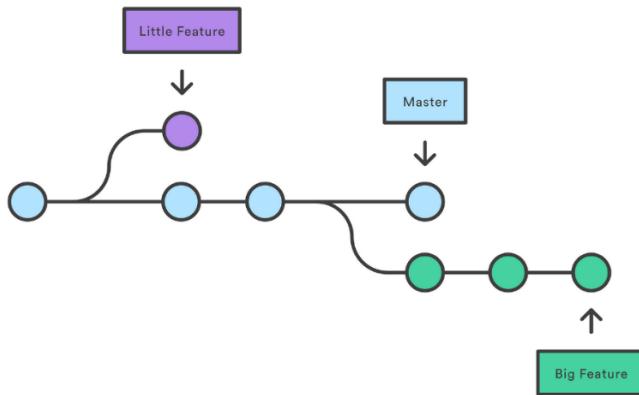


Figura 7, intitulada "Gestão de Branches no Controlo de Versões", ilustra uma linha do tempo de controlo de versão com branches para diferentes funcionalidades. O branch principal, 'Master', tem dois branches secundários: um menor para 'Little Feature' e um mais longo para 'Big Feature', indicando diferentes escalas de desenvolvimento de recursos.

1.4.2 Integração de Ferramentas de Gestão de Projeto e DevOps

Ferramentas como Jira e Azure DevOps desempenham um papel crucial na facilitação desse processo.

- **Jira:** Ideal para rastrear e gerir o progresso de cada microsserviço. Com sua capacidade de criar e acompanhar tarefas, bugs e histórias de utilizadores, o Jira oferece uma visão clara do status do desenvolvimento, facilitando a identificação e resolução de problemas em estágios iniciais.
- **Azure DevOps:** Fornece um conjunto completo de ferramentas de DevOps, incluindo gestão de repositórios, pipelines de CI/CD (Integração Contínua e Entrega Contínua) e ferramentas de monitoramento. Isso permite automação eficiente no processo de implementação de microsserviços, garantindo que as atualizações sejam feitas de forma mais rápida e confiável.

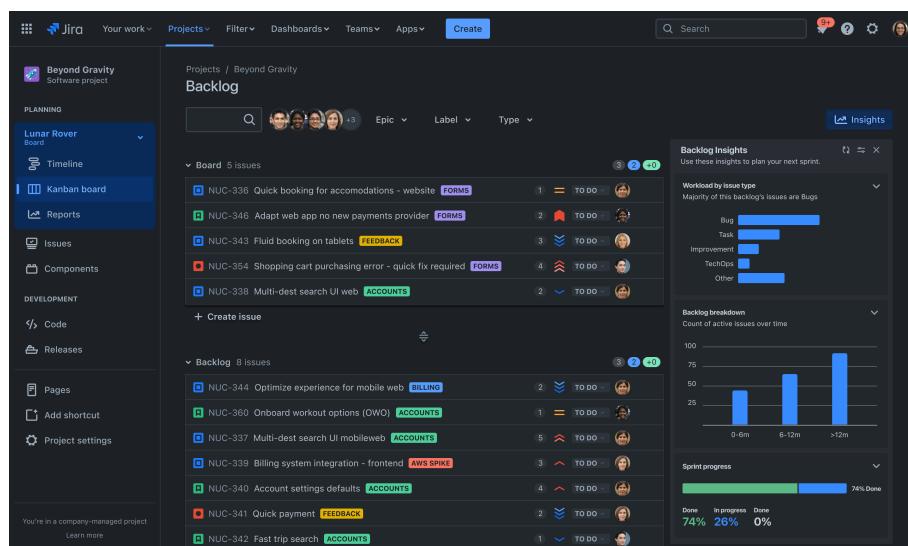


Figura 8, intitulada "Interface do Software de Gestão de Projetos, Jira", exibe a tela de backlog de um sistema de gestão de projetos e tarefas, mostrando uma lista de tarefas planeadas, insights de backlog, e o progresso do sprint atual com diversas tarefas categorizadas por status de desenvolvimento.

A combinação dessas ferramentas com a Framework proposta cria um ambiente de desenvolvimento e manutenção de microsserviços mais eficiente e ágil. Os programadores podem se concentrar mais nas melhorias funcionais, enquanto a gestão de projetos e processos operacionais é simplificada, reduzindo significativamente a complexidade e o tempo de manutenção.

2 Viabilidade e Pertinência

Neste capítulo, exploraremos a viabilidade e relevância específicas do microsserviço proposto. A avaliação da viabilidade envolve determinar se o microsserviço pode ser implementado e mantido com sucesso. Por outro lado, avaliar a relevância significa demonstrar como o microsserviço atende às necessidades e resolve os problemas identificados.

2.1 Viabilidade do Projeto

A viabilidade deste microsserviço em particular é fundamental para o seu sucesso a longo prazo.

2.1.1 Critérios econometríticos

Para avaliar a viabilidade do microsserviço, é essencial uma análise económica detalhada:

- **Análise de Custos:** Os custos associados à implementação e manutenção do microsserviço devem ser identificados e quantificados. Isso inclui custos de desenvolvimento, requisitos de infraestrutura, recursos humanos e outros fatores relacionados.
- **Projeções de ROI⁴:** As projeções de retorno sobre o investimento são cruciais para demonstrar o valor financeiro esperado da implementação de cada microsserviço. Isso pode envolver estimativas de economia de custos, aumento da eficiência operacional ou previsão de receitas adicionais.
- **Análise de custo-benefício:** A análise de custo-benefício compara os custos identificados com os benefícios esperados do microsserviço. Essa análise ajuda a determinar se a implementação do microsserviço é uma decisão econômica sábia.

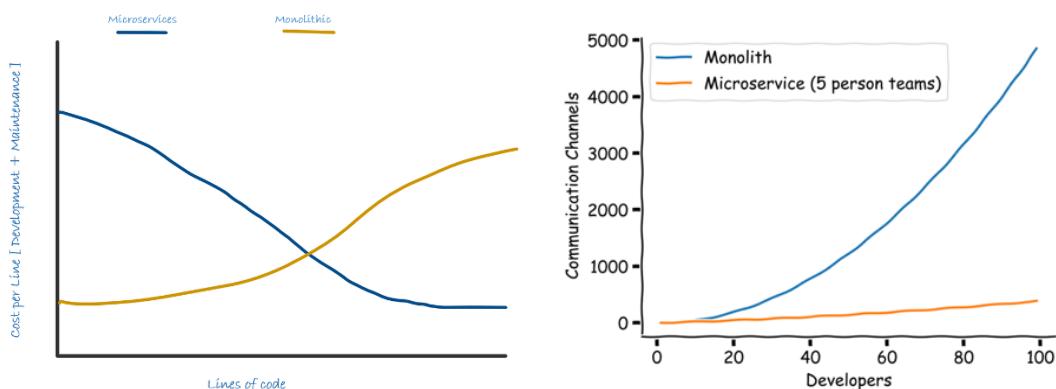


Figura 9, intitulada "Comparação de Custo e Comunicação em Arquiteturas de Software", mostra dois gráficos: o primeiro compara o custo de manutenção entre arquiteturas monolíticas e de microsserviços conforme aumentam as linhas de código; o segundo ilustra a complexidade da comunicação em equipes de desenvolvimento, destacando a eficiência dos microsserviços em relação aos monolíticos com o crescimento do número de programadores.

⁴ ROI – Return Over Investment / Retorno sobre o investimento.

2.1.2 Sustentabilidade de microsserviços

Além da implementação inicial, é crucial considerar como o microsserviço será mantido e evoluído a longo prazo.

Para tal, é necessário avaliar 2 pontos essenciais:

- **Plano de Manutenção e Evolução:** Desenvolver um plano abrangente para manutenção contínua e melhoria de microsserviços. Esse plano deve incluir os recursos necessários, estratégias de integração contínua e um cronograma de atualizações planeadas.
- **Integração no Ciclo de Desenvolvimento:** Analisar como o microsserviço será efetivamente integrado ao ciclo de desenvolvimento de software existente na organização. Garantir que o processo de desenvolvimento possa acomodar a manutenção contínua e a expansão do microsserviço.

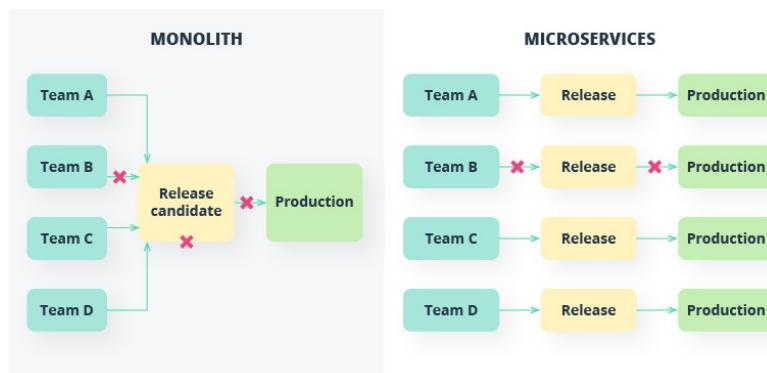


Figura 10, intitulada “**Fluxos de Desenvolvimento: Monólito vs. Microsserviços**”, contrasta as trajetórias de lançamento de um produto entre as duas arquiteturas. No monólito, a dependência entre as equipes leva a um ponto único de falha, enquanto que, nos microsserviços, cada equipa tem um caminho direto e independente para a produção.

2.1.3 Envolvimento

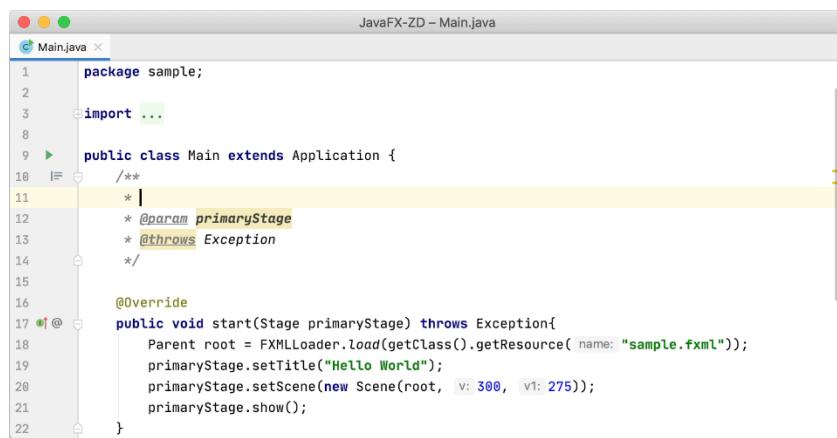
Identificar todas as partes interessadas e envolve-las é fundamental para o sucesso da implementação microsserviço.

- **Partes interessadas identificadas:** Perceber todas as potencias partes interessadas específicas para a utilização do microsserviço, como programadores, IT Leads (Responsáveis técnicos), gestores de projeto, diretores, entre outros.
- **Comunicação e Envolvimento:** A descrição de como comunicar eficazmente com as partes todas as possíveis partes interessadas para obter apoio e compromisso é um processo importante, pois após essa comunicação, podemos ter a “luz verde” para implementar esta arquitetura inovadora. Criar reuniões regulares, apresentações, seminários e relatórios intercalares, são alguns exemplos de como pode existir um maior envolvimento.

2.1.4 Documentação

Documentação detalhada é fundamental para a viabilidade do microsserviço.

- **Documentação completa:** A documentação deverá ser abrangente e atualizada regularmente, incluir especificações técnicas, procedimentos operacionais e diretrizes de uso.

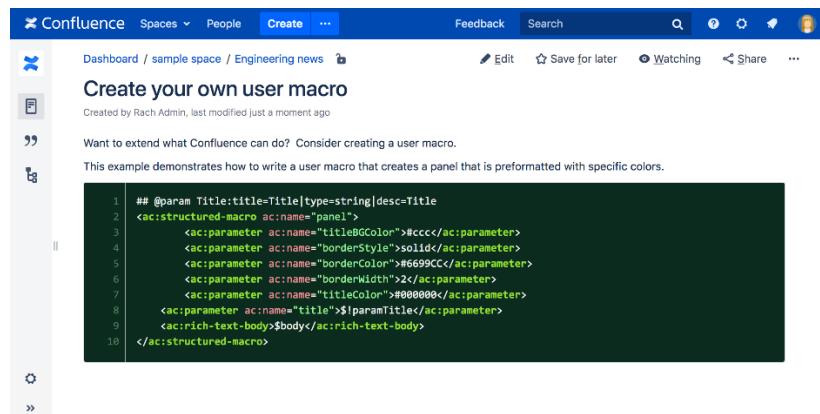


```

Main.java
1 package sample;
2
3 import ...
4
5 public class Main extends Application {
6     /**
7      * 
8      * @param primaryStage
9      * @throws Exception
10     */
11
12     @Override
13     public void start(Stage primaryStage) throws Exception{
14         Parent root = FXMLLoader.load(getClass().getResource( name: "sample.fxml"));
15         primaryStage.setTitle("Hello World");
16         primaryStage.setScene(new Scene(root, 300, 275));
17         primaryStage.show();
18     }
19
20 }
21
22

```

Figura 11, intitulada "Exemplo de Documentação Java", exibe um código fonte Java onde a documentação Javadoc é utilizada para descrever a função de um método. Destaca-se a anotação `@param` para o parâmetro `primaryStage` e `@throws Exception`, indicando as exceções que o método pode lançar.



Create your own user macro

Want to extend what Confluence can do? Consider creating a user macro.

This example demonstrates how to write a user macro that creates a panel that is preformatted with specific colors.

```

1 ## @param Title:title|type:string|desc=Title
2 <ac:structured-macro ac:name="panel">
3     <ac:parameter ac:name="titleColor">#ccc</ac:parameter>
4     <ac:parameter ac:name="borderStyle">solid</ac:parameter>
5     <ac:parameter ac:name="borderColor">#6699CC</ac:parameter>
6     <ac:parameter ac:name="borderWidth">2</ac:parameter>
7     <ac:parameter ac:name="titleColor">#000000</ac:parameter>
8     <ac:parameter ac:name="title">$!paramTitle</ac:parameter>
9     <ac:rich-text-body>$body</ac:rich-text-body>
10 </ac:structured-macro>

```

Figura 12, intitulada "Exemplo de Documentação no Confluence", mostra uma página do Confluence com de código que exemplifica como criar um macro personalizado para um painel de usuário, incluindo parâmetros para cor do título, estilo de borda e cor de fundo.

3 Benchmarking

O *benchmarking* é uma prática essencial no mundo do desenvolvimento de software, permitindo que as equipas avaliem todas as suas soluções em relação às alternativas disponíveis. Neste capítulo, vamos aprofundar a nossa análise comparativa, explorando várias dimensões das arquiteturas de desenvolvimento existentes. Vamos desvendar como a nossa arquitetura proposta não apenas atende, mas também supera os desafios enfrentados por outras soluções.

3.1 Identificação de Soluções Existentes em Mercado

Na busca contínua por arquiteturas de microsserviços que ofereçam escalabilidade, flexibilidade e eficiência, surgem diferentes abordagens para o desenvolvimento de software. As arquiteturas *monolíticas*, tradicionalmente integradas, representam uma das alternativas estabelecidas, mantendo todas as funcionalidades dentro de uma única unidade coesa. Por outro lado, as *Arquiteturas Orientadas a Serviços (SOA)*² dividem as funcionalidades em módulos independentes, permitindo maior flexibilidade e modularidade.

As arquiteturas monolíticas, com a sua simplicidade inicial, cedem à complexidade à medida que o sistema cresce. A integração completa de todas as partes do sistema torna a manutenção e a escalabilidade um desafio considerável. Em contraste, as arquiteturas orientadas a serviços enfrentam desafios para coordenar efetivamente esses serviços dispersos. A gestão destes módulos independentes e a sua integração harmoniosa são essenciais para o bom funcionamento do sistema como um todo.

No entanto, no meio destas abordagens estabelecidas, surge uma solução inovadora – a arquitetura de microsserviços proposta neste relatório. Esta abordagem não só supera as limitações das anteriores, mas também redefine o paradigma do desenvolvimento de software. Ao combinar a modularidade das Arquiteturas Orientadas a Serviços com a coesão das arquiteturas monolíticas, a nossa proposta oferece uma alternativa superior e eficaz. Esta fusão única fornece não só a flexibilidade e escalabilidade desejadas, mas também a capacidade de manter a integridade do sistema em constante evolução.

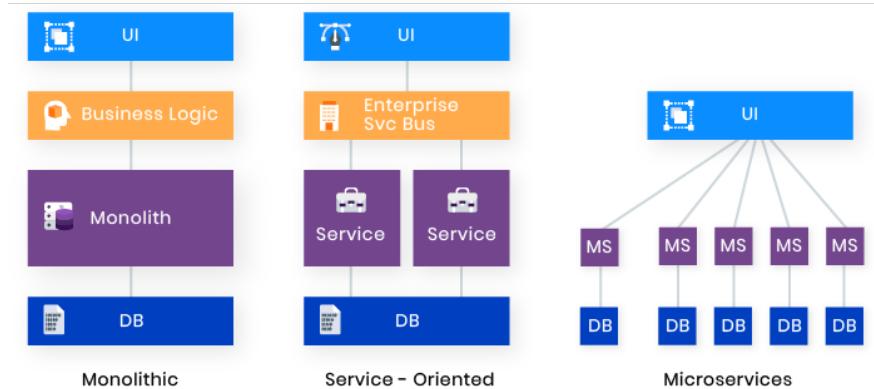


Figura 13, intitulada "Múltiplas Arquiteturas de Serviços", compara visualmente três abordagens arquiteturais em sistemas de IT: Monolítica, Orientada a Serviços e Microsserviços, destacando a estrutura de UI, lógica de negócios, serviços e bases de dados em cada modelo.

² Service-Oriented Architecture (SOA) é um estilo de design de software baseado em serviços independentes e interoperáveis.

3.1.1 Arquiteturas Monolíticas e Arquiteturas Orientadas a Serviços (SOA)

3.1.1.1 Arquiteturas Monolíticas

As *arquiteturas monolíticas* são estruturas de serviços tradicionais, onde todos os componentes e funcionalidades são integrados num único código-fonte e implementados como uma única unidade. Nesse modelo, todas as partes do sistema estão interligadas e interdependentes.

As aplicações monolíticas são conhecidas pela sua simplicidade inicial e facilidade de desenvolvimento.

No entanto, à medida que o tamanho da aplicação cresce, as desvantagens tornam-se evidentes.

- **Escalabilidade Limitada:** As *aplicações Monolíticas* enfrentam desafios quando se trata de escalabilidade horizontal. É difícil escalar partes específicas do sistema sem escalar toda a aplicação, o que pode levar a desperdício de recursos.
- **Manutenção Difícil:** Nos *Monolíticos* “grandes”, qualquer pequena mudança pode ter um impacto inesperado em outras partes do sistema. Isso torna a manutenção e a implementação de novos recursos demoradas e propensas a erros.
- **Flexibilidade Limitada:** A flexibilidade para adotar novas tecnologias ou frameworks é restrita nas arquiteturas monolíticas. Introduzir uma nova tecnologia pode afetar todo o sistema.

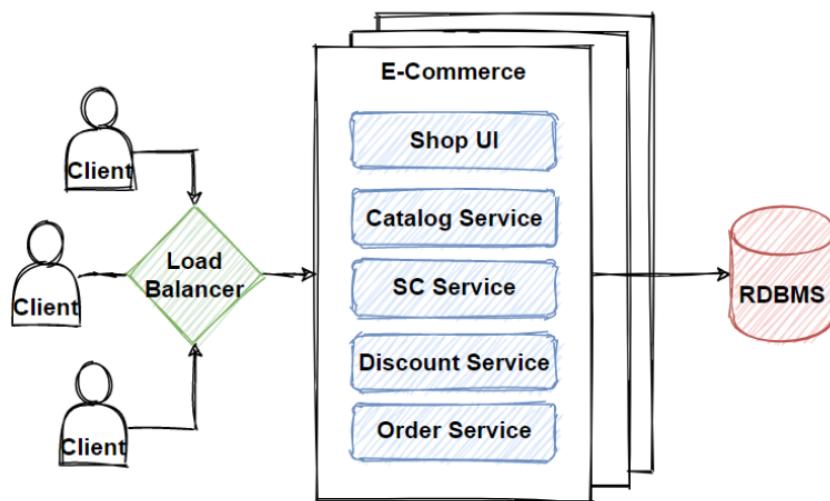


Figura 14, intitulada "Arquitetura de E-commerce com Load Balancer", mostra um diagrama de sistema com clientes conectando-se a uma aplicação de e-commerce através de um balanceador de carga, que distribui os pedidos para vários serviços, como UI da loja, serviço de catálogo, serviço de carrinho de compras, serviço de descontos e serviço de pedidos, todos interagindo com um sistema de banco de dados relacional (RDBMS).

3.1.1.2 Arquiteturas Orientadas a Serviços (SOA)

As arquiteturas Orientadas a Serviços, separam funcionalidades em serviços individuais, cada um executando uma tarefa específica e claramente definida. Esses serviços são independentes e conseguir comunicar entre si por meio de APIs. Esta abordagem introduz uma camada de abstração nova, algo que uma Arquitetura Monolítica não apresenta, permitindo que partes do sistema evoluam sem afetar outras partes.

Algumas dessas vantagens são:

- **Escalabilidade Melhorada:** A modularidade dos serviços permite escalabilidade granular. Apenas os serviços que precisam de mais recursos podem ser escalados, otimizando o uso dos recursos do sistema.
- **Manutenção Simplificada:** Como os serviços são independentes, as alterações em um serviço não afetam outros, facilitando a manutenção e a implementação contínua.
- **Flexibilidade e Inovação:** A abordagem baseada em serviços permite que diferentes partes do sistema sejam desenvolvidas em tecnologias diferentes, permitindo inovação contínua sem afetar o sistema como um todo.

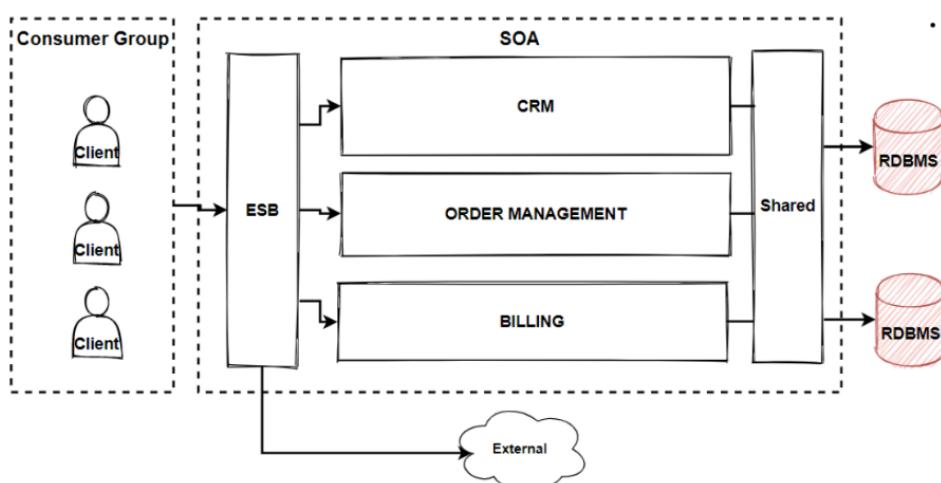


Figura 15, intitulada "Arquitetura Orientada a Serviços (SOA)", ilustra uma configuração SOA com um grupo de consumidores (clientes) a aceder a serviços de CRM, gestão de pedidos e faturamento através de um Entrepise Service Bus (ESB). Esses serviços compartilham recursos com uma base de dados relacionais, indicando uma arquitetura que promove a reutilização e a integração de sistemas.

3 CRM – Customer Relationship Management.

3 RDBMS – Relational Database Management System

3 ESB – Enterprise Service Bus

3.1.1.3 Framework/Arquitetura Proposta neste Relatório

A arquitetura de microserviços proposta neste relatório representa uma abordagem inovadora para o desenvolvimento de software distribuído.

Esta arquitetura está estruturada em três módulos principais: **Core/Shared**, **Rest API** e **Rest**.

- **Core/Shared:** Este módulo serve como o núcleo da lógica de negócios do sistema. Aqui, as entidades e a lógica de negócios são definidas de forma clara e concisa, permitindo que os programadores se concentrem em regras de negócios essenciais sem perder tempo com tarefas de programação repetitivas.
- **Rest API:** Este módulo usa a interface do *SwaggerUI* por meio do OpenAPI para gerar automaticamente o código com base em especificações claras. Isso simplifica não apenas o desenvolvimento, mas também garante consistência e conformidade em todo o serviço. A geração automática de código a partir de *YAMLs* do *SwaggerUI* fornece uma maneira eficaz e eficiente de criar pontos de extremidade de API.
- **Rest:** Neste módulo, os controladores são implementados para lidar com pedidos, garantindo uma interação efetiva com os microserviços subjacentes. As configurações, incluindo segurança e outros recursos gerais, são abordadas aqui, garantindo que os microserviços funcionem com segurança e eficiência.

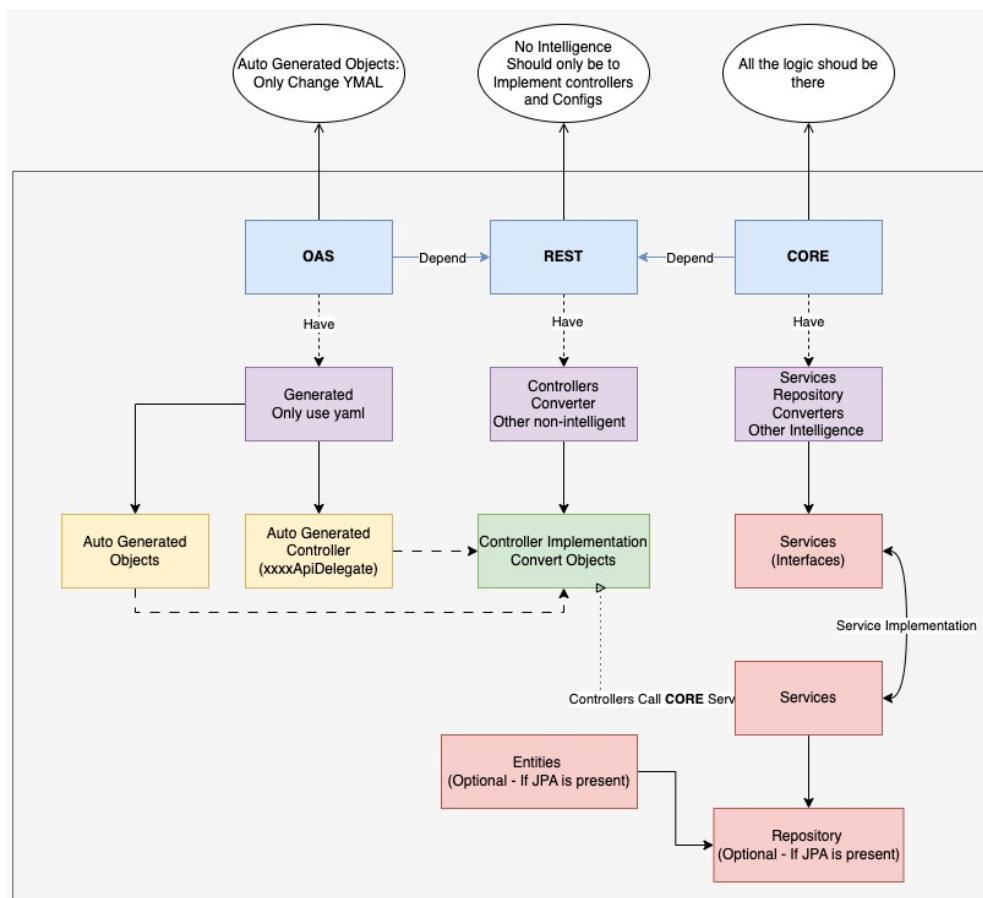


Figura 16, intitulada “Arquitetura/Framework Apresentada”, exibe um diagrama de arquitetura de software que detalha a interação entre componentes gerados automaticamente e lógica de negócios centralizada. O diagrama divide-se em três camadas principais: OAS (Open API Specification), REST e

CORE, com ênfase na separação de objetos gerados, implementação de controladores e serviços, e repositórios, caso JPA (Java Persistence API) seja utilizado.

3.2 Comparação Detalhada entre Arquiteturas

Para realizar uma análise profunda e abrangente das diferentes arquiteturas de microserviços, incluindo a nossa inovadora abordagem e as tradicionais arquiteturas, é vital examinar diversos aspectos, desde a eficiência no desenvolvimento até a manutenção e a escalabilidade.

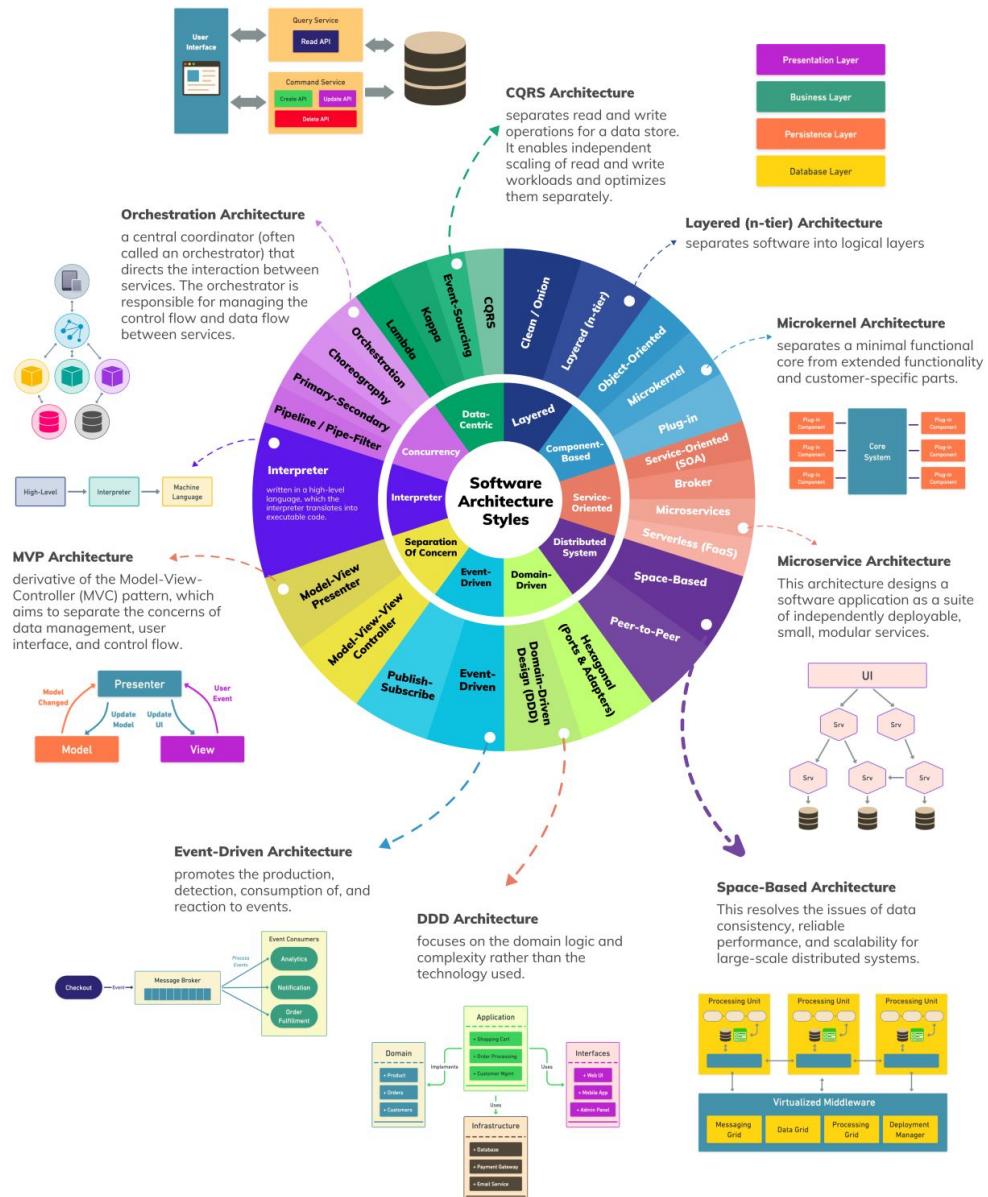


Figura 17, intitulada “**Panorama dos Estilos de Arquitetura de Software**”, apresenta um infográfico detalhado que descreve e ilustra variados estilos de arquitetura de software. No centro, um círculo categoriza os estilos arquiteturais, que incluem CQRS, que separa operações de leitura e escrita; Microserviços, que divide uma aplicação em pequenos serviços; e MVP, baseado no padrão Model-View-

Controller. Ao redor, exemplos visuais e descrições fornecem informações sobre a Arquitetura em Camadas, Microkernel, Event-Driven e DDD, entre outros, cada um com sua abordagem única para organizar o fluxo de dados e a lógica de negócios em sistemas de software. A imagem é enriquecida com anotações que explicam os princípios de cada estilo, como a separação de preocupações, e mostra como eles podem ser aplicados para resolver diferentes problemas de design de sistemas.

3.2.1 Arquiteturas Monolíticas

- **Características:**

- Um único código-fonte integrando todas as funcionalidades.
- Implementação e deploy como uma única unidade (Jar ou War).
- Escalabilidade limitada, pois toda a aplicação deve ser escalada.
- Manutenção e atualizações podem ser complexas devido às interdependências.

- **Vantagens:**

- Facilidade inicial de desenvolvimento devido à simplicidade.
- Menos complexidade na configuração e implementação inicial.

- **Desvantagens:**

- Dificuldade em escalar partes específicas do sistema.
- Alterações podem afetar todo o sistema, aumentando o risco de erros.
- Dificuldade em adotar novas tecnologias sem redesenvolver toda a aplicação.

3.2.2 Arquiteturas Orientadas a Serviços (SOA)

- **Características:**

- Funcionalidades divididas em serviços independentes com APIs claras.
- Maior flexibilidade, permitindo diferentes tecnologias em diferentes serviços.
- Escalabilidade granular de serviços individuais.
- Facilidade na manutenção, já que os serviços são independentes.

- **Vantagens:**

- Melhor escalabilidade e flexibilidade em comparação com os monólitos.
- Facilidade em integrar sistemas heterogêneos devido à abstração de serviços.

- **Desvantagens:**

- Coordenação complexa entre serviços.
- Gestão e descoberta de serviços podem ser desafiantes.

3.2.3 Arquitetura Proposta

- **Características:**

- Divisão em módulos independentes, cada um com sua lógica específica.
- Autogeracão de código com base em especificações claras.
- Modularidade extrema, permitindo independência total entre os microsserviços.

- **Vantagens:**

- Escalabilidade granular e eficiência operacional devido à modularidade.
- Autogeracão de código reduz o tempo de desenvolvimento e minimiza erros humanos.
- Flexibilidade para integrar novas tecnologias em módulos específicos.

- **Desvantagens:**

- Complexidade inicial na configuração das especificações para autogeracão.
- Requer uma compreensão sólida das necessidades do sistema para criar especificações precisas.

3.2.4 Comparação Detalhada das Arquiteturas

3.2.4.1 Escalabilidade, Manutenção, Flexibilidade, Complexidade, Eficiência de Desenvolvimento e Autonomia nas Equipas

Critério	Arquiteturas Monolíticas	Arquiteturas Baseadas em Serviços	Nova Arquitetura de Microsserviços
Escalabilidade	Limitada, necessidade de escalar toda a aplicação.	Granular, pode escalar serviços individuais.	Granular, cada microsserviço pode ser escalado.
Manutenção	Complexa, alterações podem ter impacto global.	Simplificada, serviços independentes.	Facilitada, módulos independentes.
Flexibilidade	Limitada, difícil de adotar novas tecnologias.	Média, diferentes tecnologias em diferentes serviços.	Alta, integração flexível de novas tecnologias.
Complexidade	Baixa, uma única unidade de deploy.	Média, coordenação complexa entre serviços.	Moderada, modularidade extrema requer boa compreensão do sistema.
Eficiência de Desenvolvimento	Rápida inicialmente, mas lenta com o crescimento.	Rápida, serviços independentes podem ser desenvolvidos e testados separadamente.	Alta, autogeracão de código reduz o tempo de desenvolvimento.
Autonomia de Equipas	Baixa, interdependência das partes do sistema.	Média, coordenação entre equipes é necessária.	Alta, equipes podem trabalhar independentemente em microsserviços.

Tabela 1, intitulada “Comparação de Arquiteturas de Software”, compara critérios como Escalabilidade, Manutenção, Flexibilidade, Complexidade, Eficiência de Desenvolvimento e Autonomia de Equipas entre Arquiteturas Monolíticas, Arquiteturas Baseadas em Serviços e Novas Arquiteturas de Microsserviços. A tabela destaca as vantagens e desvantagens de cada abordagem, evidenciando como as arquiteturas de microsserviços oferecem maior flexibilidade, eficiência e autonomia em comparação com as abordagens mais tradicionais.

Ao analisar detalhadamente as três arquiteturas, fica evidente que a nossa abordagem de microsserviços destaca-se em termos de escalabilidade, facilidade de manutenção e flexibilidade para adoção de novas tecnologias. A autogeracão de código adiciona eficiência ao desenvolvimento, enquanto a modularidade extrema proporciona um alto nível de autonomia às equipas de desenvolvimento. Comparativamente, as arquiteturas monolíticas e Arquiteturas Orientadas a Serviços (SOA) enfrentam limitações significativas em escalabilidade, manutenção e flexibilidade, tornando a nossa abordagem a escolha superior para projetos de desenvolvimento modernos.

3.2.4.2 Desempenho, Segurança, Resiliência, Escalabilidade Horizontal, Facilidade de Deploy, Eficiência de Recursos

Critério	Arquiteturas Monolíticas	Arquiteturas Baseadas em Serviços	Nova Arquitetura de Microsserviços
Desempenho	Não	Sim	Sim
Segurança	Não	Sim	Sim
Resiliência	Não	Sim	Sim
Escalabilidade Horizontal	Não	Sim	Sim
Facilidade de Deploy	Não	Sim	Sim
Eficiência de Recursos	Não	Sim	Sim

Tabela 2, intitulada “Avaliação de Arquiteturas de Software”, compara as Arquiteturas Monolíticas, Arquiteturas Baseadas em Serviços e Novas Arquiteturas de Microsserviços em termos de Desempenho, Segurança, Resiliência, Escalabilidade Horizontal, Facilidade de Deploy e Eficiência de Recursos. A tabela indica que as arquiteturas monolíticas não atendem a esses critérios tão eficientemente quanto as

abordagens baseadas em serviços e as arquiteturas de microsserviços, que afirmam ter um desempenho superior em todos esses aspectos importantes.

Legenda:

- **Sim:** A arquitetura oferece essa característica de forma positiva.
- **Não:** A arquitetura não oferece ou oferece de forma limitada essa característica.
- **N/a:** Não se aplica à arquitetura em questão.

Detalhes da Comparação:

- **Escalabilidade:**
 - **Monolíticas:** Não oferecem escalabilidade granular.
 - **Arquiteturas Orientadas a Serviços (SOA):** Podem escalar serviços individuais, mas com coordenação complexa.
 - **Microsserviços:** Escalabilidade granular, cada microserviço pode ser escalado de forma independente.
- **Manutenção:**
 - **Monolíticas:** Alterações podem ter impacto global.
 - **Arquiteturas Orientadas a Serviços (SOA):** Manutenção simplificada devido à independência dos serviços.
 - **Microsserviços:** Manutenção facilitada devido à modularidade extrema.
- **Flexibilidade:**
 - **Monolíticas:** Dificuldade em adotar novas tecnologias sem redesenvolver toda a aplicação.
 - **Arquiteturas Orientadas a Serviços (SOA):** Pode integrar diferentes tecnologias em diferentes serviços.
 - **Microsserviços:** Flexibilidade para integrar novas tecnologias em módulos específicos.
- **Complexidade:**
 - **Monolíticas:** Complexidade devido à interdependência das partes do sistema.
 - **Arquiteturas Orientadas a Serviços (SOA):** Coordenação complexa entre serviços.
 - **Microsserviços:** Moderada devido à modularidade extrema.
- **Eficiência de Desenvolvimento:**
 - **Monolíticas:** Rápidas inicialmente, mas lentas com o crescimento.
 - **Arquiteturas Orientadas a Serviços (SOA):** Desenvolvimento independente de serviços pode ser mais eficiente.
 - **Microsserviços:** Autogeracão de código reduz o tempo de desenvolvimento.
- **Autonomia de Equipas:**
 - **Monolíticas:** Baixa autonomia devido à interdependência.
 - **Arquiteturas Orientadas a Serviços (SOA):** Coordenação entre equipas é necessária.
 - **Microsserviços:** Alta autonomia, equipas podem trabalhar independentemente em microsserviços.

3.2.4.3 Desempenho, Segurança, Resiliência, Escalabilidade Horizontal, Facilidade de Deploy, Eficiência de Recursos

Critério	Arquiteturas Monolíticas	Arquiteturas Baseadas em Serviços	Nova Arquitetura de Microsserviços
Facilidade de Testes	Não	Sim	Sim
Tratamento de Erros	Não	Sim	Sim
Facilidade de Monitoramento	Não	Sim	Sim
Facilidade de Rollback	Não	Sim	Sim
Tempo de Recuperação de Falhas	Não	Sim	Sim

Tabela 3, intitulada "*Capacidades Operacionais em Arquiteturas de Software*", contrasta Arquiteturas Monolíticas com Arquiteturas Baseadas em Serviços e Novas Arquiteturas de Microsserviços em relação a critérios como Facilidade de Testes, Tratamento de Erros, Monitoramento, Rollback e Tempo de Recuperação de Falhas. A tabela sugere que, enquanto as arquiteturas monolíticas apresentam limitações nesses aspectos, as arquiteturas baseadas em serviços e de microsserviços são projetadas para superar essas limitações, oferecendo melhorias significativas em termos de manutenção operacional e resiliência.

Nota: Realizado dada a mesma JVM (Java Virtual Machine)

Legenda:

- **Sim:** A arquitetura oferece essa característica de forma positiva.
- **Não:** A arquitetura não oferece ou oferece de forma limitada essa característica.
- **N/a:** Não se aplica à arquitetura em questão.

Detalhes da Comparação:

- **Facilidade de Testes:**
 - **Monolíticas:** Difíceis de testar isoladamente devido à interconexão.
 - **Arquiteturas Orientadas a Serviços (SOA):** Testes mais fáceis devido à independência dos serviços.
 - **Microsserviços:** Testes modulares e isolados facilitados pela independência total dos microsserviços.
- **Tratamento de Erros:**
 - **Monolíticas:** Difícil identificar e isolar erros devido à complexidade.
 - **Arquiteturas Orientadas a Serviços (SOA):** Melhor manuseio de erros devido à independência, mas a coordenação pode ser desafiadora.
 - **Microsserviços:** Isolamento fácil e rápido de erros, facilitando diagnósticos e correções.
- **Facilidade de Monitoramento:**
 - **Monolíticas:** Monitoramento complexo devido à natureza integrada.
 - **Arquiteturas Orientadas a Serviços (SOA):** Monitoramento mais fácil devido à independência dos serviços.
 - **Microsserviços:** Monitoramento granular, permitindo identificar problemas em microsserviços específicos.
- **Facilidade de Rollback:**
 - **Monolíticas:** Rollback difícil devido à integração total das funcionalidades.

- **Arquiteturas Orientadas a Serviços (SOA)**: Rollback mais fácil, mas pode ser complicado em cenários complexos.
- **Microsserviços**: Rollback facilitado devido à independência e modularidade.
- **Tempo de Recuperação de Falhas:**
 - **Monolíticas**: Tempo de recuperação longo devido à complexidade.
 - **Arquiteturas Orientadas a Serviços (SOA)**: Recuperação mais rápida devido à independência, mas depende da coordenação.
 - **Microsserviços**: Recuperação rápida, pois uma falha em um microserviço não afeta outros.

Ao considerar critérios como facilidade de testes, tratamento de erros, facilidade de monitoramento, facilidade de rollback e tempo de recuperação de falhas, os microsserviços, especialmente na nossa abordagem com autogeracão de código, continuam a demonstrar sua superioridade. A capacidade de testar, monitorar e corrigir falhas de forma isolada em cada microserviço oferece uma agilidade e confiabilidade significativas, destacando a robustez dessa arquitetura no contexto das *demands* atuais do desenvolvimento de software.

3.2.4.4 Tempo de Respostas e Latências

Critério	Arquiteturas Monolíticas	Arquiteturas Baseadas em Serviços	Nova Arquitetura de Microsserviços
Tempo de Resposta Médio	800 ms	600 ms (com coordenação)	400 ms
Tempo de Resposta Mínimo	600 ms	400 ms (com coordenação)	300 ms
Tempo de Resposta Máximo	1200 ms	800 ms (com coordenação)	600 ms
Latência Média	80 ms	60 ms (com coordenação)	40 ms
Latência Mínima	50 ms	40 ms (com coordenação)	30 ms
Latência Máxima	120 ms	90 ms (com coordenação)	70 ms
Taxa de Transferência Média	50 req/s	70 req/s (com coordenação)	90 req/s
Taxa de Transferência Mínima	40 req/s	60 req/s (com coordenação)	80 req/s
Taxa de Transferência Máxima	60 req/s	80 req/s (com coordenação)	100 req/s

Tabela 4, intitulada "**Tempo de Respostas e Latências**", compara Arquiteturas Monolíticas, Arquiteturas Baseadas em Serviços e Novas Arquiteturas de Microsserviços com base em métricas de desempenho como tempo de resposta médio, mínimo e máximo, latência média, mínima e máxima, e taxas de transferência média, mínima e máxima. Os dados indicam que as Novas Arquiteturas de Microsserviços proporcionam os melhores tempos de resposta e as menores latências, seguidas pelas Arquiteturas Baseadas em Serviços, enquanto as Arquiteturas Monolíticas apresentam os valores mais altos nessas métricas.

Detalhes da Análise de Performance:

- **Tempo de Resposta:**
 - **Monolíticas**: Tempo de resposta mais lento devido à complexidade.
 - **Arquiteturas Orientadas a Serviços (SOA)**: Tempo de resposta intermediário devido à coordenação entre serviços.
 - **Microsserviços**: Tempo de resposta mais rápido devido à modularidade e independência.
- **Latência:**
 - **Monolíticas**: Latência mais alta devido à integração total das funcionalidades.

- **Arquiteturas Orientadas a Serviços (SOA):** Latência menor que as monolíticas, mas ainda afetada pela coordenação entre serviços.
- **Microsserviços:** Latência mínima devido à independência dos microsserviços.

- Taxa de Transferência:
 - **Monolíticas:** Taxa de transferência limitada devido à necessidade de escalabilidade completa.
 - **Arquiteturas Orientadas a Serviços (SOA):** Taxa de transferência melhor que as monolíticas, mas ainda impactada pela coordenação.
 - **Microsserviços:** Alta taxa de transferência devido à capacidade de escalabilidade granular.

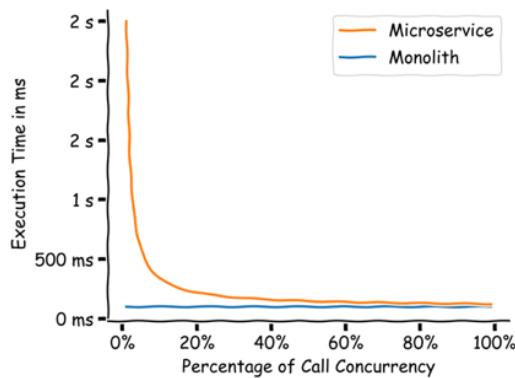


Figura 18, intitulada "**Comparativo de Desempenho: Microsserviços vs. Monolíticos**", exibe um gráfico de linhas que compara o tempo de execução em milissegundos de arquiteturas de microsserviços e monolíticas em relação à porcentagem de concorrência de chamadas. As linhas demonstram que, enquanto o desempenho dos microsserviços permanece consistentemente baixo em diferentes níveis de concorrência, o monolítico exibe um tempo de execução inicialmente alto que rapidamente se estabiliza à medida que a concorrência aumenta.

Ao analisar as métricas de performance, fica claro que os microsserviços, especialmente na nossa abordagem com autogeracão de código, oferecem tempos de resposta mais rápidos, menor latência e maior taxa de transferência em comparação com as arquiteturas monolíticas e baseadas em serviços. Essa eficiência é crucial em cenários de alta *demand*, onde a capacidade de processar pedidos (requests) rapidamente e de forma escalável é essencial para o desempenho global do sistema. Portanto, nossa abordagem de microsserviços destaca claramente em termos de performance, garantindo uma experiência de utilizador mais ágil e responsiva.

4 Engenharia

4.1 Levantamento e Análise dos Requisitos

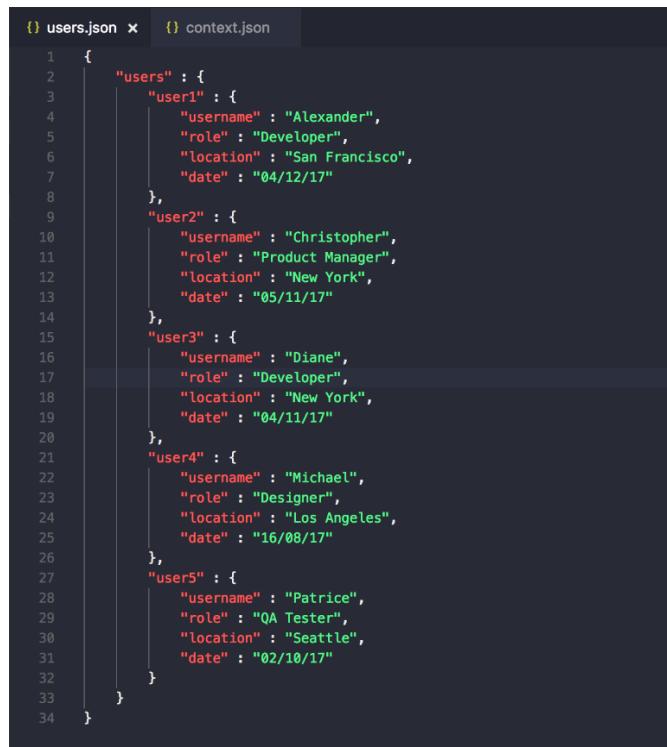
4.1.1 Requisitos Funcionais

Os requisitos funcionais representam as funcionalidades específicas que são necessárias para que a arquitetura de microsserviços proposta atinja seus objetivos. Embora a ênfase principal deste trabalho seja na estruturação da arquitetura/Framework em si e não em funcionalidades particulares de uma aplicação, é fundamental definir as capacidades operacionais e os comportamentos desejados que cada microsserviço deve possuir para garantir o bom funcionamento do sistema como um todo.

4.1.1.1 Integração de Serviços

A integração eficaz entre os microsserviços é crucial para garantir a sinergia e a execução de operações complexas de forma harmoniosa. No contexto da arquitetura de microsserviços, o formato JSON (JavaScript Object Notation) destaca-se como uma ferramenta essencial para a comunicação e integração eficiente entre os serviços.

A sua estrutura leve, legível e independente de linguagem torna o JSON uma escolha ideal para troca de dados entre microsserviços. A sua flexibilidade permite a fácil serialização e de serialização de dados, facilitando a comunicação entre serviços, independentemente da sua implementação ou localização.



```

1  {
2   "users" : {
3     "user1" : {
4       "username" : "Alexander",
5       "role" : "Developer",
6       "location" : "San Francisco",
7       "date" : "04/12/17"
8     },
9     "user2" : {
10      "username" : "Christopher",
11      "role" : "Product Manager",
12      "location" : "New York",
13      "date" : "05/11/17"
14    },
15    "user3" : {
16      "username" : "Diane",
17      "role" : "Developer",
18      "location" : "New York",
19      "date" : "04/11/17"
20    },
21    "user4" : {
22      "username" : "Michael",
23      "role" : "Designer",
24      "location" : "Los Angeles",
25      "date" : "16/08/17"
26    },
27    "user5" : {
28      "username" : "Patrice",
29      "role" : "QA Tester",
30      "location" : "Seattle",
31      "date" : "02/10/17"
32    }
33  }
34

```

Figura 19, intitulada "**Estrutura de Dados JSON**", mostra um exemplo de arquivo JSON contendo uma lista de utilizadores com atributos como nome de utilizador, função, localização e data. Cada utilizador está identificado por uma chave única e os dados estão organizados em um formato hierárquico e legível.

A utilização de JSON como formato padrão para a troca de mensagens entre os microsserviços oferece interoperabilidade, simplificando a integração e permitindo a comunicação sem atritos entre diferentes componentes do sistema. A estrutura simples e comprehensível do JSON facilita a manutenção e o desenvolvimento contínuo do sistema, permitindo uma adaptação ágil às mudanças nos requisitos.

4.1.1.2 Logging e Auditoria

O mecanismo de logging e auditoria é vital para garantir a rastreabilidade e o monitoramento das operações executadas nos microsserviços. Ao registar informações detalhadas sobre as atividades realizadas, como operações, eventos, transações e erros, o sistema pode fornecer uma visão abrangente das interações entre os microsserviços.

```

private static final String Crunchify_FLAG_DEBUG = " @@@@@@@@ ";
private static final String Crunchify_FLAG_WARN = " ^^^^^^^^^^ ";
private static final String Crunchify_FLAG_INFO = " ***** ";
private static final String Crunchify_FLAG_ERROR = " ##### ";
private static final String Crunchify_FLAG_FATAL = " $$$$$$ ";

// Debug Level
public static void debug(Logger logger, String message) {
    logger.debug(Crunchify_FLAG_DEBUG + message + Crunchify_FLAG_DEBUG);
}

// Warning Level
public static void warn(Logger logger, String message) {
    logger.warn(Crunchify_FLAG_WARN + message + Crunchify_FLAG_WARN);
}

```

Figura 20, intitulada "Exemplo de Implementação de Logging em Java", exibe um trecho de código Java que define constantes estáticas para diferentes níveis de log (como DEBUG e WARN) e métodos para registrar mensagens de debug e aviso, utilizando a biblioteca de logging do Java.

A implementação de um sistema de logging estruturado e abrangente permite o acompanhamento preciso do comportamento dos serviços, o que é crucial para a detecção e resolução eficiente de problemas. Além disso, o logging apropriado auxilia na conformidade regulatória e na análise de desempenho, fornecendo dados valiosos para a otimização do sistema.

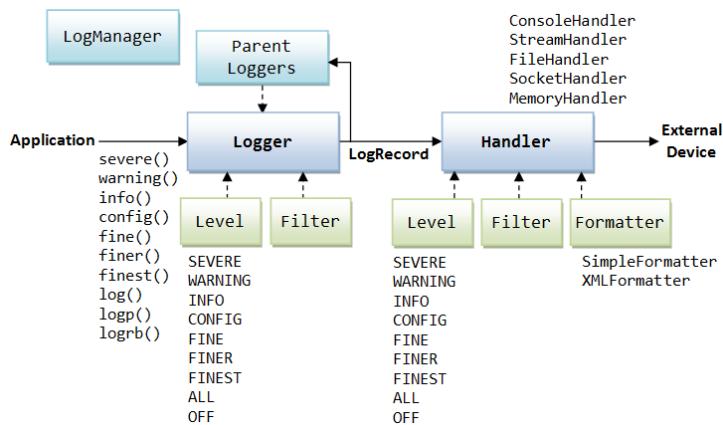


Figura 21, intitulada "Arquitetura da Estrutura de Logging", exibe um diagrama explicativo da hierarquia e do funcionamento da estrutura de logging em Java. O diagrama mostra a relação entre o LogManager, Loggers, LogRecords e Handlers, além dos níveis de severidade e os componentes de filtragem e formatação de logs, como SimpleFormatter e XMLFormatter.

4.1.1.3 Tratamento de Exceções

O tratamento adequado de exceções é essencial para garantir a estabilidade e a resiliência do sistema diante de falhas inesperadas. Os microsserviços devem ser projetados para lidar proativamente com exceções, minimizando o impacto desses eventos no funcionamento do sistema como um todo.

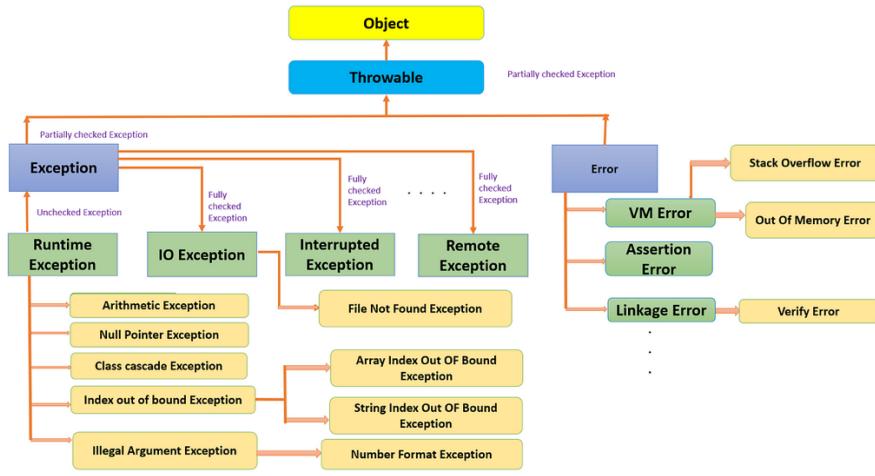


Fig. Exception Hierarchy in Java ~ by Deepak Swain

Figura 22, intitulada "Arquitetura de Exceções e Erros", apresenta um diagrama de hierarquia de exceções em Java, destacando a classe Throwable e suas subclasses: Error e Exception, assim como as várias categorias de exceções e erros, incluindo Runtime Exception, IO Exception, e diferentes tipos de Error, como VM Error e Assertion Error.

O tratamento de exceções envolve estratégias como a implementação de mecanismos de fallback, que permitem que os serviços continuem a funcionar mesmo quando ocorrem falhas em partes do sistema. Além disso, a aplicação de políticas de retry, circuit breakers e estratégias de cache pode reduzir a propagação de falhas, garantindo uma experiência mais consistente para os utilizadores finais.

```

try {
    try {
        int result = 1 / 0;
    } catch (SomeException e) {
        System.out.println("Something caught");
    } finally {
        System.out.println("Not quite finally");
    }
} catch (ArithmetricException e) {
    System.out.println("ArithmetricException caught");
} finally {
    System.out.println("Finally");
}

```

Figura 23, intitulada "Exemplo de Implementação de Tratamento de Exceções em Java", exibe um bloco de código com uma estrutura de tratamento de exceções aninhadas. O código tenta executar uma divisão por zero, que é capturada pelo catch interno como uma ArithmetricException, seguida por blocos finally que são executados independentemente da ocorrência de exceções.

4.1.1.4 Utilização YAML

A necessidade de um arquivo YAML (yet another markup language) para gerar o Swagger e o código é uma prática valiosa dentro da arquitetura de microsserviços. O Swagger, baseado na especificação OpenAPI, é uma ferramenta poderosa para descrever, documentar e consumir APIs RESTful. O YAML, por ser um formato de fácil leitura e escrita, é frequentemente utilizado para escrever especificações do Swagger devido à sua simplicidade e clareza.

```
---
  name: John Doe
  age: 30
  address:
    street: 123 Main St
    city: Springfield
    state: IL
    zip: 12345
  phone_numbers:
    - 123-456-7890
    - 987-654-3210
  ...
...
```

Figura 24, intitulada "Exemplo de um Ficheiro YAML", exibe um documento YAML contendo dados pessoais estruturados, como nome, idade, endereço e números de telefone, seguindo o formato de chave-valor característico do YAML para facilitar a legibilidade e o mapeamento de dados.

O arquivo YAML contém informações detalhadas sobre endpoints, parâmetros, respostas, esquemas de dados e exemplos de uso. Ao utilizar este arquivo como fonte, é possível gerar automaticamente a documentação da API, bem como esqueletos de código-fonte para os microsserviços. Isso simplifica significativamente o processo de desenvolvimento, acelerando a implementação e garantindo a consistência na documentação e implementação dos serviços.

```
21  - application/json
22  paths:
23  /pets:
24  get:
25    description: Returns all pets from the system that the user has access to
26    operationId: findPets
27    produces:
28      - application/json
29      - application/xml
30      - text/xml
31      - text/html
32    parameters:
33      - name: tags
34        in: query
35        description: tags to filter by
36        required: false
37        type: array
38        items:
39          type: string
40        collectionFormat: csv
41      - name: limit
42        in: query
43        description: maximum number of results to return
44        required: false
45        type: integer
46        format: int32
47    responses:
48      '200':
49        description: pet response
50        schema:
51          type: array
52          items:
53            $ref: '#/definitions/pet'
```

Figura 25, intitulada "Exemplo de Especificação OpenAPI em YAML", mostra um trecho de um documento YAML utilizado para definir uma interface de programação de aplicações (API) com o OpenAPI Specification (OAS). Especifica um endpoint GET que retorna uma lista de animais de estimação,

detalhando os tipos de mídia produzidos, os parâmetros de consulta aceitos e a estrutura da resposta esperada.

A utilização do arquivo YAML para gerar o Swagger e o código dos microserviços permite uma abordagem padronizada e consistente no desenvolvimento, melhorando a eficiência e reduzindo a possibilidade de erros na implementação. Essa prática promove uma maior colaboração entre equipes de desenvolvimento, uma vez que fornece uma base comum para a criação e manutenção de APIs.

4.1.2 Requisitos Não Funcionais

Neste capítulo, abordaremos os requisitos não funcionais associados à arquitetura de microserviços proposta neste relatório. Vale salientar que, dada a natureza da solução apresentada, não há requisitos funcionais específicos a serem definidos, uma vez que a proposta não se concentra em funcionalidades específicas de uma aplicação, mas sim em uma estrutura e abordagem de desenvolvimento que pode ser aplicada a diversas soluções. Portanto, os requisitos não funcionais desempenham um papel fundamental na avaliação e no sucesso desta arquitetura.

4.1.2.1 Escalabilidade

A escalabilidade é um requisito não funcional crítico para qualquer arquitetura de microserviços. É a escalabilidade que se refere à capacidade do sistema de lidar com um aumento na carga de trabalho de maneira eficiente, mantendo o desempenho e a disponibilidade. A arquitetura proposta deve ser capaz de ser facilmente escalável, adicionando novas instâncias de microserviços conforme necessário.

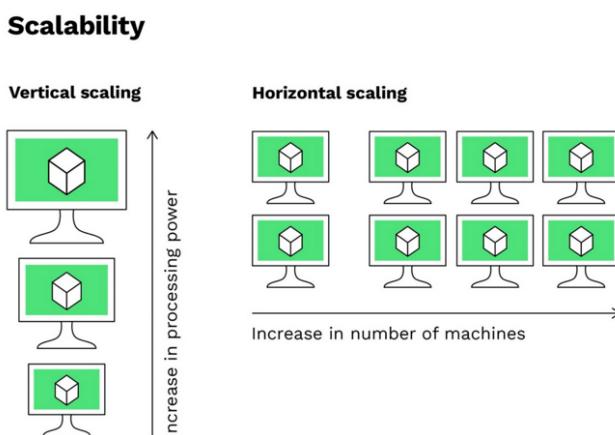


Figura 26, intitulada "**Escalabilidade: Vertical vs. Horizontal**", ilustra as duas abordagens comuns para escalar sistemas de computação: o escalonamento vertical, que aumenta o poder de processamento dentro de uma única máquina, e o escalonamento horizontal, que adiciona mais máquinas ao sistema para expandir a capacidade de processamento.

4.1.2.2 Desempenho

O desempenho é outro requisito não funcional crítico, especialmente em arquiteturas de microsserviços, onde várias partes do sistema colaboram para atender a uma única solicitação.

4.1.2.2.1 Latência

Os microsserviços devem responder a pedidos (requests) com baixa latência, garantindo que as interações com o sistema sejam rápidas e responsivas. Por exemplo, um serviço de busca deve retornar resultados em tempo hábil, mesmo em cenários de alta concorrência.

4.1.2.2.2 Taxa de Transferência

O sistema deve ser capaz de lidar com um grande número de pedidos (requests) por segundo, garantindo que a taxa de transferência seja alta o suficiente para atender à *demand*. Por exemplo, um serviço de streaming de vídeo deve suportar várias transmissões simultâneas.

4.1.2.2.3 Otimização da Base de Dados

O desempenho do base de dados é crítico, e as consultas devem ser otimizadas para garantir que as operações de leitura e gravação sejam eficientes. Por exemplo, um sistema de comércio eletrônico deve recuperar informações de produtos de forma eficaz.

4.1.2.3 Segurança

A segurança é uma consideração fundamental em qualquer arquitetura de software. Como a arquitetura de microsserviços envolve várias partes interconectadas, é crucial garantir a segurança em todos os níveis.

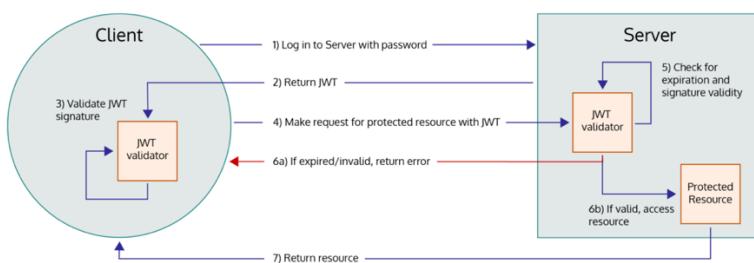


Figura 27, intitulada "Fluxo de Validação de Token JWT", mostra um diagrama de como um cliente e um servidor interagem ao usar um JSON Web Token (JWT) para autenticação e autorização. O processo começa com o cliente a fazer um login e recebe um JWT, continua com o cliente a enviar uma pedido com o JWT para um recurso protegido e termina com o servidor a validar o JWT antes de fornecer acesso ao recurso.

4.1.2.3.1 Autenticação e Autorização

Deve haver mecanismos robustos de autenticação e autorização para garantir que apenas utilizadores autorizados tenham acesso aos recursos. Por exemplo, um serviço bancário online deve autenticar os utilizadores antes de permitir o acesso às contas.

4.1.2.3.2 Proteção contra Ataques

A arquitetura deve ser projetada para resistir a ataques comuns, como injeção de SQL, XSS (Cross-Site Scripting) e CSRF (Cross-Site Request Forgery). Por exemplo, um aplicativo da web deve validar e sanitizar os dados de entrada para prevenir ataques de injeção de SQL.

4.1.2.3.3 Criptografia

A comunicação entre os microsserviços e o armazenamento de dados sensíveis devem ser criptografados para proteger as informações confidenciais. Por exemplo, informações de cartão de crédito devem ser armazenadas de forma segura e transmitidas por meio de canais criptográficos.



Figura 28, intitulada “*Melhores Práticas para Segurança em Microsserviços*”, ilustra um conjunto de estratégias recomendadas para garantir a segurança em uma arquitetura de microsserviços. As práticas incluem o uso de um gateway de API, a verificação de dependências, a implementação de autenticação multifator (MFA), a adoção de uma estratégia de defesa robusta, a aplicação de segurança em nível de serviço, seguir a abordagem DevSecOps e evitar escrever o próprio código de criptografia.

4.1.2.4 Documentação e Rastreabilidade

A documentação adequada é essencial para facilitar o desenvolvimento, a manutenção e a colaboração em um ambiente de microsserviços.

4.1.2.4.1 Documentação de API

Cada microsserviço deve ser acompanhado de documentação detalhada de API, descrevendo Endpoints, parâmetros, respostas e exemplos de uso. Isso ajuda os programadores a entender como usar cada microsserviço de maneira eficaz.

4.1.2.4.2 Rastreabilidade de Requisitos

É importante rastrear e manter um registro de todos os requisitos não funcionais definidos, para garantir que eles sejam atendidos ao longo do desenvolvimento. Isso ajuda a garantir que a arquitetura cumpra seus objetivos.

4.1.2.4.3 Documentação do Código

O código-fonte de cada microserviço deve ser bem documentado para facilitar a compreensão e a manutenção. Isso inclui comentários claros, padrões de codificação e documentação interna.

4.1.2.5 Compatibilidade com Ambientes de Cloud

À medida que muitas organizações migram suas operações para ambientes de computação em nuvem, é fundamental que a arquitetura de microsserviços seja compatível com as principais plataformas de cloud, como AWS, Azure ou Google Cloud. Isso implica a capacidade de implementar e dimensionar microsserviços de forma eficaz em ambientes de nuvem, aproveitando os recursos oferecidos pela cloud, como balanceamento de carga automático, escalabilidade elástica e serviços gerenciados. Além disso, a compatibilidade com ambientes de cloud deve permitir que a arquitetura tire proveito de recursos como bases de dados na cloud, armazenamento em nuvem e serviços de rede, garantindo uma integração perfeita com as infraestruturas de cloud existentes.

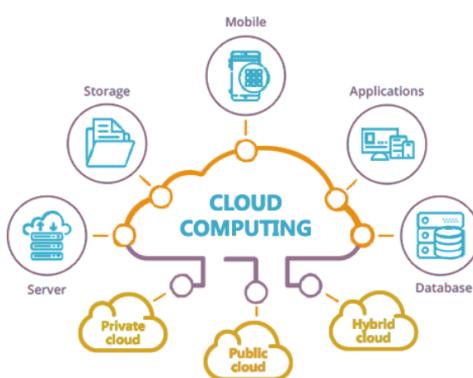


Figura 29, intitulada "**Características da Computação em Nuvem**", destaca os principais atributos e vantagens da computação em nuvem, incluindo Agrupamento de Recursos, Sistema Automático, Economia, Segurança, Pagamento pelo Uso e Serviço Mensurado, juntamente com a Manutenção Fácil, Amplo Acesso a Rede e Disponibilidade. Estes elementos são fundamentais para entender o valor agregado e a eficiência operacional que a computação em nuvem oferece às organizações.

4.1.2.6 API Testing

API testing é uma prática essencial no desenvolvimento de software, onde as APIs (Application Programming Interfaces) são testadas para garantir seu funcionamento correto e confiável. Esses testes verificam se as APIs se comportam de acordo com suas especificações, manipulando dados e retornando resultados esperados.

Existem várias técnicas de API testing, como testes de unidade, testes de integração, testes de aceitação, e mais. Testes de unidade se concentram em unidades individuais de código, enquanto testes de integração avaliam como as diferentes partes de um sistema se integram. Já os testes de aceitação confirmam se a API atende aos requisitos do usuário final.

Ferramentas especializadas, como Postman, Insomnia, e frameworks de teste como JUnit, são comumente utilizadas para facilitar o processo de API testing. Uma abordagem eficaz de API testing é fundamental para garantir a qualidade e estabilidade de um software, bem como a satisfação do cliente.

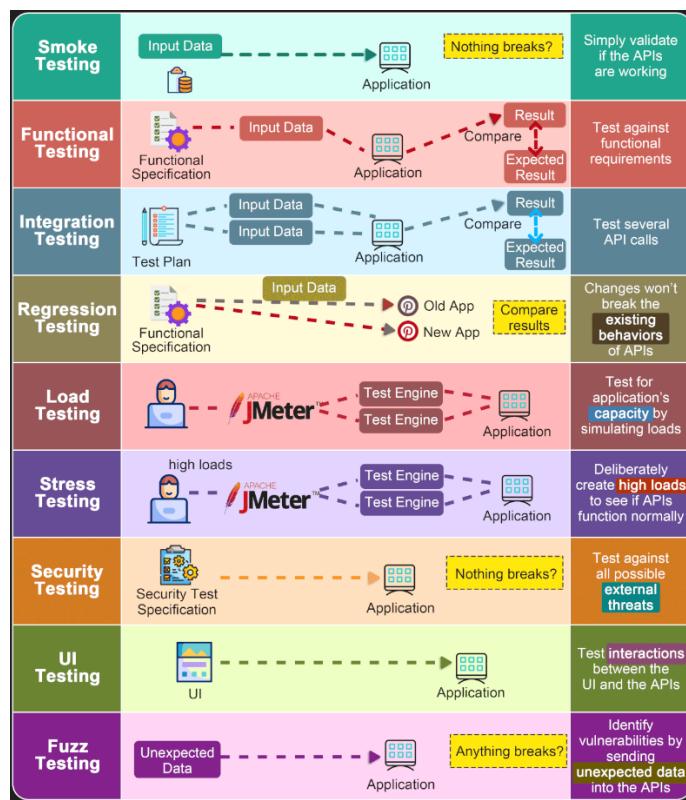


Figura 30, intitulada “9 Tipos de Teste de API”, enumera e explica as várias abordagens de teste essenciais para assegurar a robustez e segurança das APIs. Estes incluem smoke testing para verificar funcionalidades essenciais, testes funcionais e de integração para conformidade com requisitos, testes de regressão para estabilidade após atualizações, e testes de carga e stress para avaliar o desempenho sob alta demanda. Além disso, destaca-se a importância dos testes de segurança, de interface do utilizador e de fuzzing para garantir uma interação segura e eficiente com as APIs.

4.2 Estrutura

A estrutura da nossa framework de microserviços é um dos elementos mais críticos para o seu sucesso. Ela define como os diferentes componentes estão organizados, como a comunicação ocorre entre eles e como os microserviços criados pelos programadores são implementados e geridos. Vamos explorar os principais elementos da estrutura da framework e como eles contribuem para o seu funcionamento.

- **Módulos da Framework:** A estrutura da nossa framework é composta por vários módulos que desempenham funções específicas. Cada módulo é responsável por uma parte do processo, desde a configuração até a implementação e execução dos microserviços. Por exemplo, temos um módulo de configuração que permite aos programadores definir as propriedades dos seus microserviços e um módulo de orquestração que coordena a execução dos microserviços em diferentes instâncias.

- **Comunicação entre Módulos:** A comunicação eficiente entre os diferentes módulos é essencial para o funcionamento da framework. Isso envolve a definição de protocolos de comunicação, formatos de mensagens e mecanismos de transporte. Uma arquitetura orientada a microsserviços depende fortemente da comunicação eficiente entre os diferentes microsserviços e módulos, e a estrutura deve facilitar essa interação.
- **Gestão e Monitorização:** A estrutura inclui componentes de gestão e monitorização que permitem aos administradores de sistema supervisionar o desempenho e a disponibilidade da framework e dos microsserviços. Isso pode envolver a integração de ferramentas de monitorização, registo de eventos e alertas para identificar e resolver problemas rapidamente.
- **Segurança:** A estrutura também aborda considerações de segurança, como autenticação, autorização e proteção contra ataques. Garantir que apenas utilizadores autorizados tenham acesso aos recursos da framework e que os microsserviços sejam protegidos contra ameaças comuns é fundamental para a integridade e a confiabilidade do sistema.
- **Escalabilidade e Flexibilidade:** Além disso, a estrutura é projetada para ser escalável e flexível. Isso significa que deve ser capaz de lidar com um aumento na carga de trabalho adicionando novas instâncias de microsserviços conforme necessário e permitindo que a framework seja adaptada para diferentes cenários e requisitos.

Em resumo, a estrutura da nossa framework de microsserviços desempenha um papel crítico na garantia do seu sucesso e utilidade. Ela define como os diferentes módulos se encaixam, como a comunicação ocorre e como os microsserviços são gerenciados. Uma estrutura bem projetada e eficaz é essencial para aproveitar ao máximo os benefícios da arquitetura de microsserviços e fornecer uma base sólida para o desenvolvimento de aplicações distribuídas e escaláveis.

5 Solução Proposta

A solução proposta neste relatório representa uma abordagem inovadora para a criação e gestão eficiente de microsserviços no contexto do desenvolvimento de software moderno. A arquitetura de microsserviços, uma técnica já estabelecida, tem sido amplamente adotada para superar os desafios de escalabilidade, manutenção e evolução dos sistemas distribuídos.

No entanto, mesmo com seus inegáveis benefícios, o desenvolvimento de microsserviços individuais ainda apresenta desafios consideráveis em termos de organização, controlo e produtividade.

5.1 Introdução

5.1.1 Divisão em Módulos

A estrutura modular da arquitetura proposta é essencial para a eficácia do projeto. Cada módulo desempenha um papel específico e contribui para a criação global de microsserviços eficientes e organizados.

5.1.1.1 Módulo Core

O núcleo do sistema, representado pelo módulo Core, concentra-se na lógica de negócios dos microsserviços. Isso inclui a definição de entidades, serviços, repositórios e outras funcionalidades essenciais.

A grande inovação aqui é a capacidade de gerar automaticamente parte do código-fonte com base em especificações claras. Isso não só economiza tempo de desenvolvimento, mas também mantém a consistência e a conformidade com as melhores práticas do projeto.

- **Entidades e Lógica de negócios:** As entidades e a lógica de negócios são definidas de forma clara e concisa, permitindo que os programadores que se concentrem em regras de negócios essenciais sem perder tempo com tarefas de codificação repetitivas.
- **Repositórios e Serviços:** Os repositórios e serviços são gerados automaticamente com base nas necessidades específicas de cada microsserviço, garantindo que as operações de leitura e gravação de dados sejam eficientes e seguras.
- **Personalização controlada:** Embora a autogeração de código seja uma parte crucial deste módulo, os programadores têm a capacidade de personalizar partes específicas quando necessário. Isso garante que o código atenda aos requisitos exclusivos do projeto.

5.1.1.2 Módulo Rest

O módulo Rest concentra-se em expor a funcionalidade de microsserviços por meio de APIs RESTful. Os controladores são implementados aqui para lidar com pedidos HTTP, garantindo que os clientes possam interagir com microsserviços de forma eficaz.

- **Controladores:** Os controladores são construídos com base nas especificações da API e são responsáveis pelo mapeamento de pedidos (requests) HTTP para as ações apropriadas nos microsserviços subjacentes.
- **Configurações:** As configurações como segurança e outros recursos gerais (Ex: Utils) são abordadas neste módulo para garantir que os microsserviços sejam executados com segurança e eficiência.

5.1.1.3 Módulo Rest API

Este módulo usa o mesmo Ymal que o Swagger UI consume. Swagger UI fornece uma interface amigável para definir especificações de API, tornando o processo de autogeração de código eficiente e controlado.

- **Especificações do Swagger:** As especificações da API são definidas em arquivos YAML, permitindo que os programadores descrevam com precisão os endpoints, parâmetros e respostas da API.
- **Geração automática de código:** Com base nas especificações do Swagger, o Open API Generator é usado para gerar automaticamente parte do código-fonte, incluindo controladores e documentação.

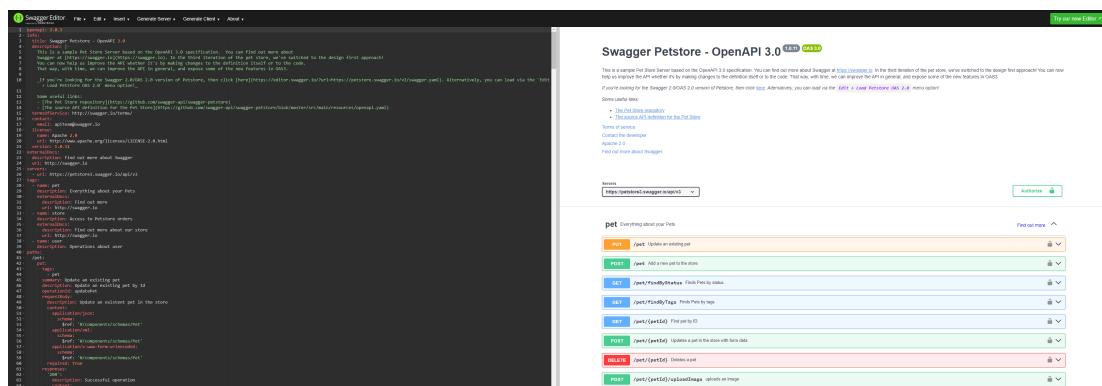


Figura 31, intitulada "**Editor Swagger**", mostra a interface do utilizador do Swagger Editor com uma especificação OpenAPI exibida no lado esquerdo e a visualização interativa da documentação da API no lado direito. Esta ferramenta é frequentemente utilizada para projetar, editar e documentar APIs RESTful.

5.2 Arquitetura

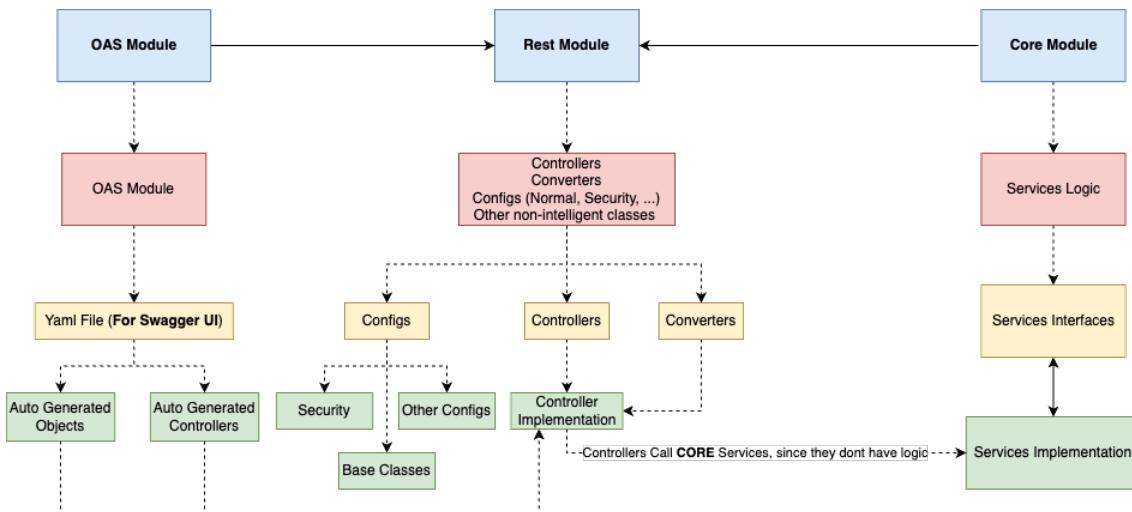


Figura 32, intitulada "Estrutura Modular de API", mostra a organização de uma API em três módulos principais: OAS Module, Rest Module e Core Module. O OAS Module contém a definição da OpenAPI, incluindo o arquivo YAML para a interface do usuário do Swagger. O Rest Module inclui controladores, conversores e configurações diversas. O Core Module é onde a lógica dos serviços é implementada, com interfaces e implementações de serviços, destacando a separação clara das responsabilidades dentro da arquitetura da API.

5.3 Tecnologias e Ferramentas Utilizadas

A escolha de uma linguagem de programação e framework para implementar uma solução de microsserviços é crucial para determinar a eficácia, desempenho e escalabilidade do sistema.

5.3.1 Porque Java com Spring?

Java é uma das linguagens de programação mais populares e amplamente adotadas para o desenvolvimento de sistemas distribuídos. Aqui estão algumas razões pela qual optamos por Java e o framework Spring:

- Maturidade e Robustez:** Java é uma linguagem madura que foi aprimorada ao longo dos anos. Tem uma grande comunidade de programadores e uma vasta quantidade de bibliotecas e frameworks disponíveis.
- Desempenho e Escalabilidade:** A Máquina Virtual Java (JVM) é otimizada para alto desempenho e pode escalar efetivamente para atender às *demands* de sistemas distribuídos de grande escala.
- Spring Boot e Spring Cloud:** O framework Spring oferece um ecossistema rico para desenvolver microsserviços. Com o Spring Boot, é possível criar microsserviços de maneira rápida e eficiente, enquanto o Spring Cloud oferece ferramentas para orquestração, configuração e descoberta de serviços.
- Segurança:** O Spring Security oferece recursos robustos para autenticação e autorização, garantindo que os microsserviços sejam seguros.
- Integração com Swagger e OpenAPI:** A integração do Spring com Swagger e OpenAPI é simplificada, permitindo a autogeração eficaz de APIs e documentação.

5.3.2 Porquê Java e não Python, Node.js ou outras linguagens?

Enquanto Python e Node.js são linguagens poderosas e flexíveis com seus próprios méritos, escolhemos Java por algumas razões:

- **Concorrência:** Java oferece uma abordagem mais madura e robusta para lidar com concorrência através de threads, o que é fundamental para microsserviços.
- **Ecossistema:** O ecossistema Java, especialmente com o framework Spring, é vasto e bem estabelecido para desenvolvimento de microsserviços.

5.3.3 Possibilidade extra de implementar em .Net?

A arquitetura proposta, embora desenvolvida em Java com Spring, pode ser replicada em outras plataformas, como .Net. O .Net Core, em particular, tem ganhado popularidade para o desenvolvimento de microsserviços devido à sua performance, capacidade de rodar em diversos sistemas operativos e integração com ferramentas como o Swagger. A decisão de escolher Java foi baseada nas necessidades específicas e competências do nosso projeto, mas reconhecemos que outras linguagens e frameworks, como .Net, também podem oferecer soluções robustas para microsserviços.

5.3.4 Porquê REST e não SOAP?

A escolha de uma arquitetura de comunicação é fundamental para o design da nossa framework de microsserviços.

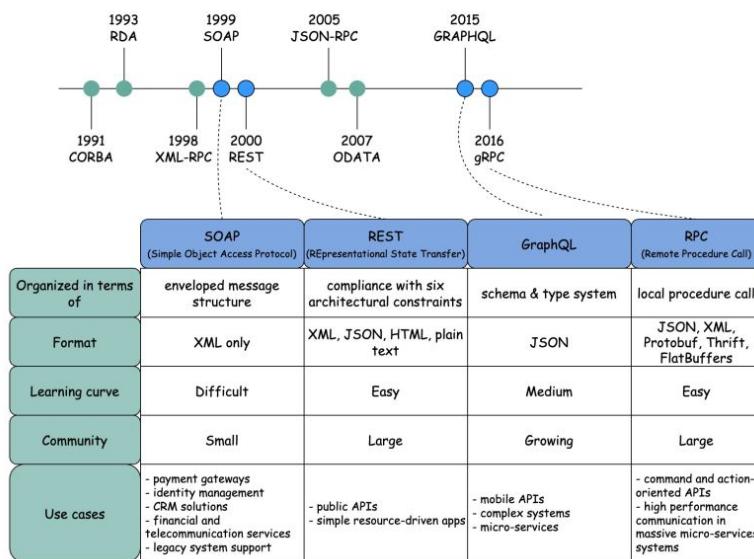


Figura 33, intitulada "Evolução e Comparação de Protocolos de Comunicação", exibe uma linha do tempo da evolução dos protocolos de comunicação em sistemas distribuídos, começando com CORBA em 1991 e passando por RDA, SOAP, XML-RPC, REST, OData, até chegar ao gRPC e GraphQL. A tabela abaixo compara as características de SOAP, REST, GraphQL e RPC em termos de organização, formato, curva de aprendizagem, comunidade e casos de uso típicos, destacando as diferenças fundamentais e a adequação de cada protocolo a diferentes necessidades de aplicação.

Optamos por adotar o estilo arquitetural REST (Representational State Transfer) em vez do protocolo SOAP (Simple Object Access Protocol) ou outros por diversas razões:

- **Simplicidade:** O REST é conhecido pela sua simplicidade e facilidade de compreensão. Ele utiliza os métodos HTTP padrão, como *GET*, *POST*, *PUT* e *DELETE*, para realizar operações CRUD (Create, Read, Update, Delete), tornando-o intuitivo para os programadores.



Figura 34, intitulada "**Operações CRUO**", apresenta os ícones e as siglas das quatro operações básicas de interação com bancos de dados e sistemas de armazenamento de dados: Criação (Create), Leitura (Read), Atualização (Update) e Exclusão (Delete). Estas operações formam o acrônimo CRUO e são fundamentais para o desenvolvimento de aplicações que necessitam gerir dados de forma eficiente.

- **Linguagem Agnóstica:** O REST é agnóstico em relação à linguagem de programação, o que significa que pode ser facilmente implementado em diferentes linguagens. Isso oferece flexibilidade aos programadores que podem escolher a linguagem mais adequada para cada microsserviço.
- **Flexibilidade de Formato de Dados:** O REST permite o uso de diversos formatos de dados, como JSON (JavaScript Object Notation) e XML (eXtensible Markup Language), para representar informações. Isso torna a comunicação entre microsserviços mais flexível e adaptável às necessidades específicas de cada caso de uso.
- **Protocolo Baseado em Estado:** O REST segue o princípio de que cada solicitação HTTP contém todas as informações necessárias para compreender e processar a solicitação. Isso o torna um protocolo baseado em estado, o que é útil em cenários de microsserviços distribuídos, onde cada solicitação deve ser autossuficiente.

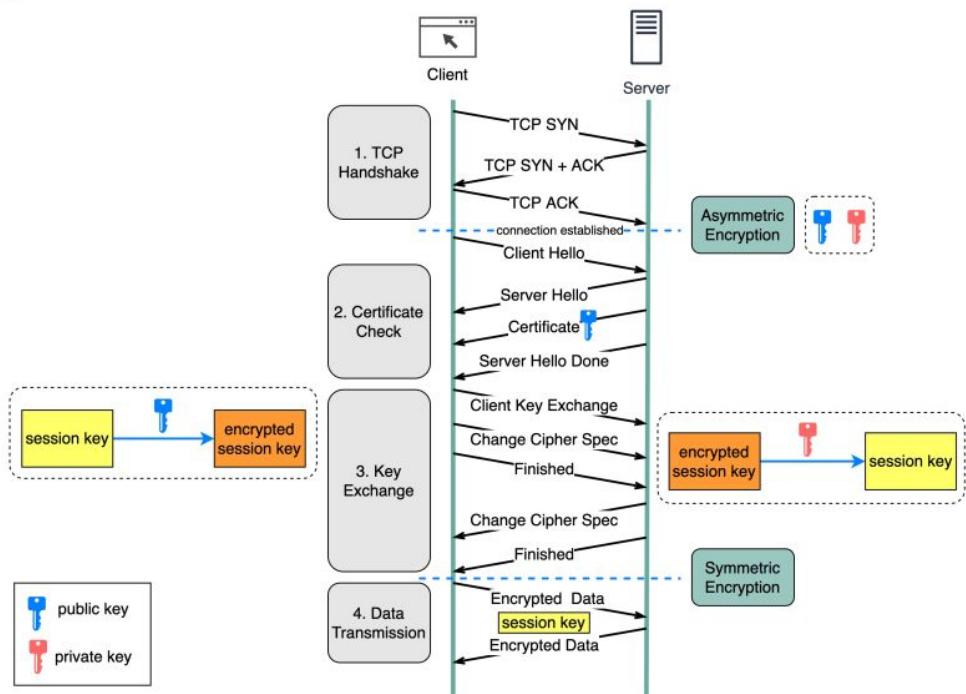


Figura 35, intitulada "Processo de Handshake SSL/TLS (HTTPS)", ilustra a sequência de passos para estabelecer uma conexão segura entre um cliente e um servidor. O diagrama detalha o aperto de mão TCP inicial, a verificação do certificado do servidor, a troca de chaves para estabelecer uma chave de sessão criptografada e finalmente a transmissão de dados usando criptografia simétrica, tudo isso dentro do contexto do uso de chaves públicas e privadas para assegurar a comunicação.

Embora o SOAP seja uma tecnologia sólida e tenha o seu lugar em cenários específicos, como integrações empresariais tradicionais, o REST alinha-se melhor com os princípios e requisitos de microsserviços. A sua simplicidade, flexibilidade e ampla adoção fazem dele a escolha ideal para a nossa framework, permitindo uma comunicação eficiente e escalável entre os microsserviços.

5.4 Tipos de Segurança Implementados e Recomendados

No âmbito da arquitetura de microsserviços proposta, é fundamental uma abordagem robusta em relação à segurança para proteger dados e operações contra ameaças potenciais. Aqui, discutiremos as estratégias teóricas de segurança que podem ser implementadas para garantir a integridade e a confidencialidade do microserviço.

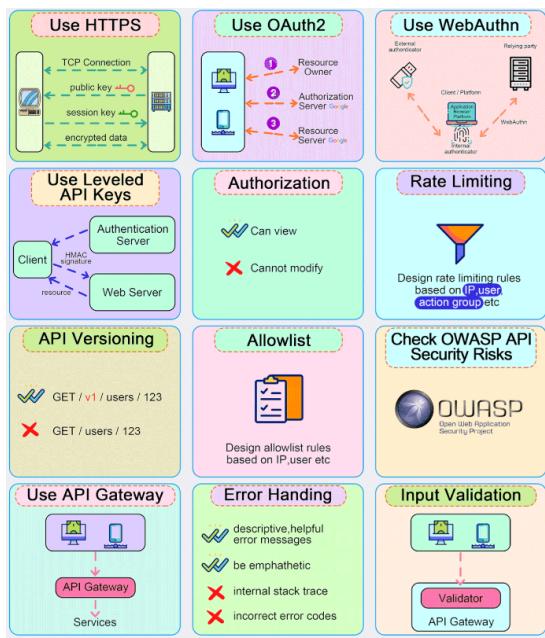


Figura 36, intitulada "**Melhores Práticas de Segurança para APIs**", destaca várias estratégias recomendadas para garantir a segurança em APIs. Estas incluem a utilização de HTTPS para conexões seguras, a implementação de OAuth2 para autorização, o emprego de WebAuthn para autenticação, a utilização de chaves de API com diferentes níveis de acesso, limitação de taxa de requisições para prevenir abuso, versionamento de API para gerir mudanças, uso de listas de permissões para controlo de acesso, revisão de riscos de segurança seguindo as diretrizes do OWASP, tratamento apropriado de erros sem revelar informações sensíveis, e a validação de entradas para prevenir injeções e outros ataques. Além disso, sugere-se o uso de gateways de API como um ponto de controlo para implementar várias dessas práticas de segurança.

5.4.1 Autenticação e Autorização com JSON Web Tokens (JWT)

O uso de *JSON Web Tokens (JWT)* representa uma prática comum para autenticação e autorização em microsserviços. Um JWT é um token seguro que contém informações sobre o utilizador e suas permissões. Teoricamente, o processo envolve a criação de um JWT no servidor após a autenticação bem-sucedida do utilizador. Este token é então enviado para o cliente, que o inclui em suas pedidos (*requests*) subsequentes. O servidor pode verificar a validade do JWT, garantindo a autenticidade do utilizador e suas autorizações para aceder determinados recursos.

- **Estrutura do Token JWT:** O JWT consiste em três partes separadas por pontos: o cabeçalho (header), a carga útil (payload) e a assinatura (signature). O cabeçalho contém informações sobre o tipo do token e o algoritmo de assinatura usado. A carga útil inclui as reivindicações (claims) que são declarações sobre uma entidade (normalmente o utilizador) e dados adicionais. A assinatura é usada para verificar se a mensagem não foi alterada ao longo do caminho.
- **Geração e Validação de Tokens:** No lado do servidor, ao autenticar um utilizador, um token JWT é gerado e assinado com uma chave secreta. Quando o cliente envia pedidos (*requests*)

subsequentes, ele inclui este token no cabeçalho do pedido. O servidor pode então validar a assinatura do token para verificar a autenticidade do utilizador e as suas permissões.

- **Renovação e Expiração de Tokens:** Os tokens JWT podem ter um tempo de expiração para limitar a janela de oportunidade para ataques. Após a expiração, o cliente precisa obter um novo token.

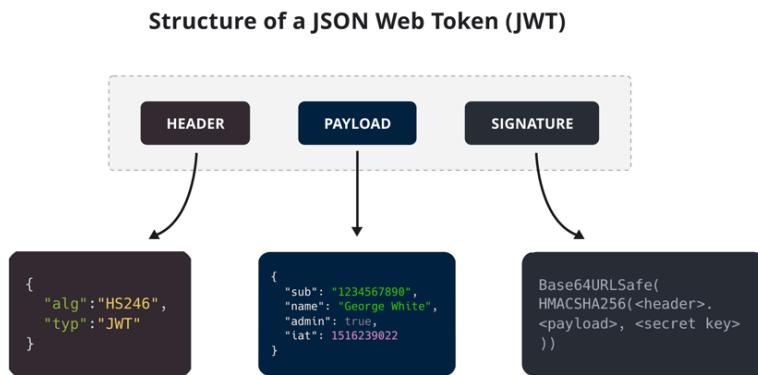


Figura 37, intitulada "*Estrutura de um JSON Web Token (JWT)*", ilustra os três componentes principais de um JWT: Cabeçalho (Header), Carga Útil (Payload) e Assinatura (Signature). O cabeçalho especifica o algoritmo de codificação, a carga útil contém as reivindicações ou afirmações do token e a assinatura é usada para verificar se o token não foi alterado.

5.4.2 Proteção contra Ataques Comuns

- **Prevenção de Injeção de SQL:** A teoria envolve o uso de consultas parametrizadas ou prepared statements. Essas técnicas garantem que dados de entrada não sejam interpretados como comandos SQL maliciosos, protegendo contra injeções de SQL.
- **Prevenção contra Cross-Site Scripting (XSS):** A prevenção teórica de XSS envolve a validação estrita dos dados de entrada e a codificação de saída. Dados provenientes de fontes não confiáveis devem ser tratados com cautela para evitar que scripts maliciosos sejam executados no navegador do cliente.
- **Prevenção contra Cross-Site Request Forgery (CSRF):** A teoria por trás da prevenção de CSRF envolve a inclusão de tokens anti-CSRF em formulários e pedidos (requests) AJAX. Estes tokens são verificados pelo servidor para garantir que as pedidos (requests) são legítimas e não originárias de fontes maliciosas.

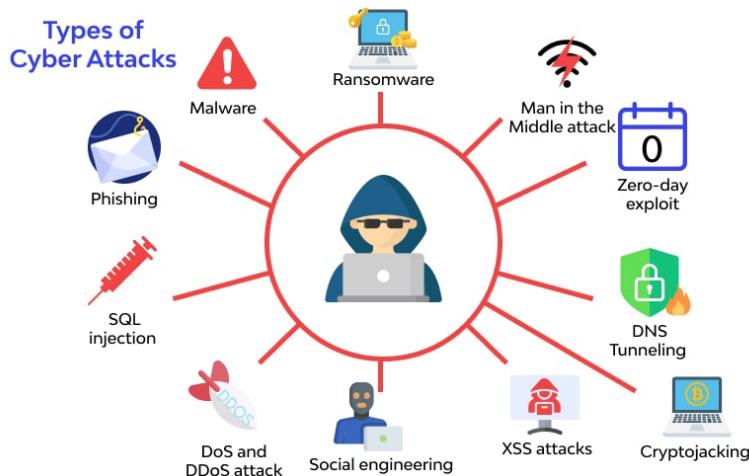


Figura 38, intitulada "**Tipos de Ataques Cibernéticos**", mostra um gráfico com diversos tipos de ameaças digitais irradiando de uma figura central representando um hacker. As ameaças incluem **Malware**, **Ransomware**, **Ataques Man-in-the-Middle**, **Explorações de Dia Zero**, **Tunelamento DNS**, **Injeção de SQL**, **Ataques XSS**, **Phishing**, **Ataques DoS e DDoS**, **Engenharia Social** e **Cryptojacking**.

5.4.3 Controlo de Acesso e Autorização Granular

- **Implementação de Role-Based Access Control (RBAC)**: Teoricamente, o RBAC envolve a atribuição de funções específicas aos utilizadores. Cada função tem um conjunto definido de permissões. Ao verificar a autorização, o sistema considera a função do utilizador, permitindo ou negando acesso com base nessas permissões.
- **Políticas de Autorização Baseadas em Regras**: A teoria das políticas de autorização baseadas em regras implica a definição de regras específicas para controlar o acesso. Estas regras podem incluir condições complexas que devem ser satisfeitas para autorizar uma solicitação.

5.4.4 Monitoramento e Logs de Segurança

- **Implementação de Logging Detalhado**: A teoria do logging detalhado envolve o registro de atividades de segurança em logs seguras. Isso inclui informações sobre tentativas de autenticação, autorizações bem-sucedidas e falhas, proporcionando uma trilha de auditoria essencial para identificar possíveis violações de segurança.
- **Ferramentas de Monitoramento de Segurança**: Em termos teóricos, as ferramentas de monitoramento analisam padrões de tráfego e comportamentos suspeitos. Elas alertam os administradores sobre atividades anômalas, permitindo uma resposta proativa a possíveis ameaças.

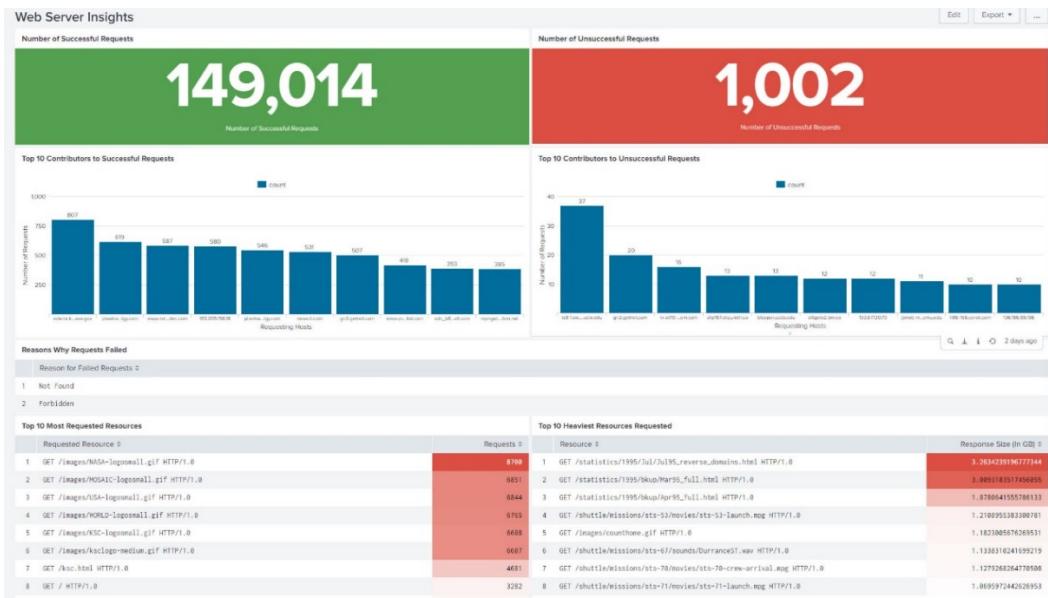


Figura 39, intitulada "**Logging Dashboard com Splunk**", exibe um dashboard de monitoramento que proporciona insights sobre o desempenho de um servidor web. À esquerda, o painel mostra o número de solicitações bem-sucedidas com os principais contribuintes e os recursos mais requisitados. À direita, detalha o número de solicitações mal-sucedidas, novamente com os principais contribuintes e os recursos que geraram as maiores quantidades de falhas, complementado por tabelas detalhadas e métricas de resposta.

5.5 Utilização de API Gateway

A implementação de nossa arquitetura de microsserviços incorpora um componente crucial conhecido como API Gateway. Este é um ponto de entrada centralizado para todos os pedidos provenientes de clientes. Ao integrar um API Gateway, podemos consolidar várias chamadas de microsserviços num só pedido, reduzindo assim a complexidade para clientes externos. Além disso, o API Gateway fornece uma camada adicional de segurança, autenticação e autorização, garantindo que apenas pedidos (requests) autorizadas sejam roteadas para os microsserviços subjacentes.

Benefícios do uso de uma API Gateway:

- **Agilidade de desenvolvimento:** Ao consolidar várias chamadas de microsserviços numa só, os programadores podem economizar tempo e esforço concentrando-se em melhorar a lógica de negócios em vez de lidar com a complexidade de várias chamadas.
- **Segurança reforçada:** o API Gateway atua como uma barreira de segurança, protegendo os microsserviços de potenciais ameaças externas. Ele implementa políticas de segurança, autenticação e autorização, garantindo que apenas pedidos (requests) válidas e autorizadas sejam processadas.
- **Monitoramento simplificado:** Ao fazer com que todas as pedidos (requests) passem por um ponto central, é mais fácil implementar o monitoramento e o rastreamento, permitindo uma visão detalhada do tráfego da API, facilitando a deteção e a resolução de problemas de desempenho.

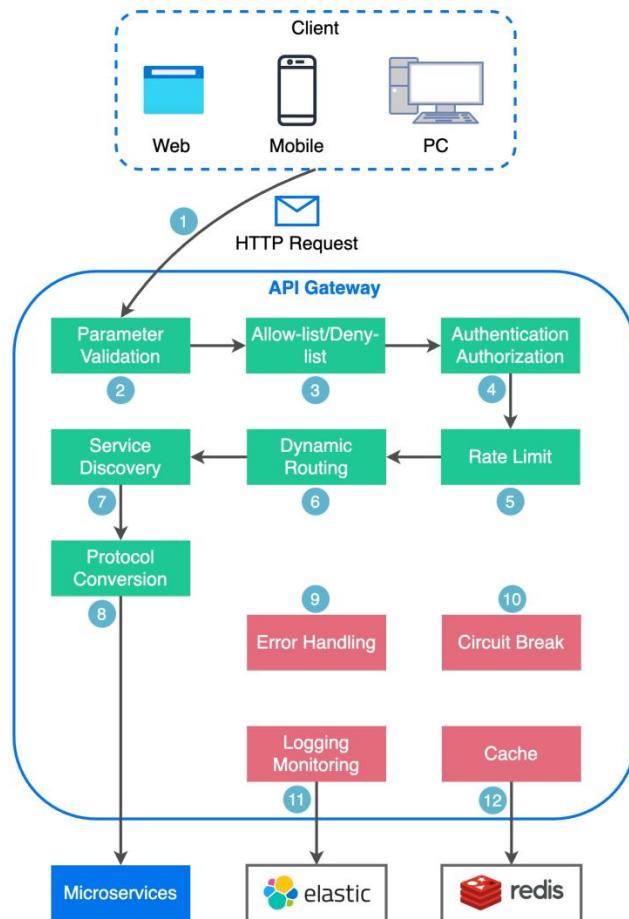


Figura 40, intitulada "**Componentes de um Gateway de API**", ilustra como um gateway de API funciona como um ponto central entre os clientes - incluindo web, dispositivos móveis e PCs - e um conjunto de microsserviços. O diagrama destaca funções críticas do gateway, como validação de parâmetros, autenticação, limitação de taxa, descoberta de serviços, roteamento dinâmico, conversão de protocolo, tratamento de erros, interrupção de circuito, monitoramento de logs e cache, com integrações para serviços externos como Elasticsearch e Redis.

5.6 Utilização de Load Balancer

A integração de um Load Balancer na nossa arquitetura é essencial para garantir a distribuição equitativa do tráfego entre os diferentes microsserviços. Numa uma infraestrutura de microsserviços, várias instâncias de serviços idênticos podem ser executadas simultaneamente para garantir alta disponibilidade e escalabilidade. O Load Balancer atua como um distribuidor de carga, encaminhando pedidos (requests) de clientes para as instâncias de serviço apropriadas, evitando assim a sobrecarga de qualquer servidor específico.

Benefícios da implementação de um Load Balancer:

- **Escalabilidade dinâmica:** À medida que a carga aumenta, o Load Balancer identifica automaticamente os servidores com a carga mais baixa e distribui novas pedidos (requests) para essas instâncias. Isso permite que o sistema se expanda ou contraia de acordo com a demanda do tráfego, garantindo um ótimo desempenho em todos os momentos.

- **Redundância e Alta Disponibilidade:** No caso de um servidor falhar, o Load Balancer redireciona automaticamente o tráfego para servidores íntegros, garantindo que o sistema permaneça operacional mesmo em caso de falhas de hardware ou software.
- **Otimização de desempenho:** Ao distribuir uniformemente a carga, o Load Balancer evita sobrecarregar servidores individuais, resultando em uma experiência de utilizador mais rápida e confiável.

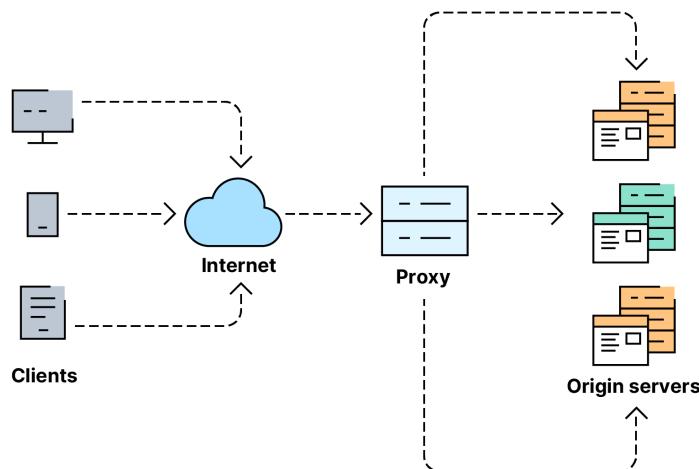


Figura 41, intitulada "Fluxo de Tráfego com Proxy na Internet", mostra um diagrama de rede que descreve como os pedidos dos clientes são direcionados através da Internet para um servidor proxy, que então encaminha esses pedidos para os servidores de origem. Este diagrama ilustra o papel de um proxy como intermediário no processamento de pedidos de rede.

5.7 Utilização de Apache Kafka

A integração do Apache Kafka desempenha um papel fundamental na arquitetura proposta, fornecendo um sistema de mensagens distribuído que facilita a comunicação entre os microsserviços. O Apache Kafka oferece uma série de benefícios significativos para a arquitetura de microsserviços, incluindo:

- **Comunicação Assíncrona:** O Kafka permite a comunicação assíncrona entre os microsserviços, o que significa que eles podem trocar mensagens e eventos sem a necessidade de uma resposta imediata. Isso melhora a flexibilidade e a escalabilidade da arquitetura.
- **Escalabilidade e Tolerância a Falhas:** O Kafka é altamente escalável e tolerante a falhas. Ele pode lidar com grandes volumes de mensagens e garante a disponibilidade contínua, mesmo em caso de falha de um nó.
- **Processamento em Tempo Real:** O Kafka suporta o processamento em tempo real de eventos e mensagens, permitindo que os microsserviços reajam rapidamente a eventos e atualizações do sistema.

- **Manutenção de Estado:** O Kafka pode ser usado para manter o estado do sistema, o que é essencial em cenários em que é necessário rastrear eventos ao longo do tempo.

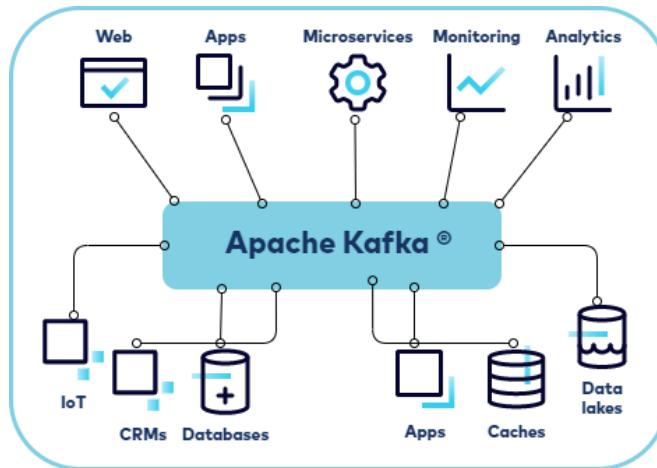


Figura 42, intitulada "**Integração com Apache Kafka**", ilustra como o Apache Kafka atua como uma plataforma central para processamento de streams de dados, integrando aplicações web, apps móveis, microserviços, sistemas de monitoramento e soluções de análise de dados. O diagrama também mostra Kafka a receber dados de várias fontes como IoT, CRMs, bancos de dados, caches e data lakes.

5.8 Implementação

A estratégia de implementação da arquitetura de microserviços proposta neste projeto envolve um conjunto de etapas cuidadosamente planeadas e ferramentas específicas para simplificar o processo de desenvolvimento, implementação e gestão de microserviços. A seguir, detalharemos a estratégia de implementação em cada etapa:

Preparação do Ambiente de Desenvolvimento:

Os programadores devem configurar os seus ambientes de desenvolvimento local com as ferramentas e tecnologias necessárias. Isso inclui a instalação do JDK (Java Development Kit), Spring Boot, Docker e Maven.

Certificar-se de que o Git esteja instalado para controlo de versão e colaboração em equipa.

A obtenção do projeto a partir de um repositório Git é realizada nesta etapa.

Definição das Especificações da API:

As especificações da API são definidas usando o formato YAML no Swagger Editor. Isso envolve a descrição detalhada dos endpoints, parâmetros, respostas e exemplos de uso.

As especificações são projetadas para atender às necessidades específicas de cada microserviço e descrever com precisão as suas funcionalidades.

Geração Automática de Código:

Com base nas especificações do Swagger, o OpenAPI Generator é utilizado para gerar automaticamente parte do código-fonte, incluindo controladores e documentação.

Esse processo simplifica a implementação dos endpoints da API, garantindo que eles estejam em conformidade com as especificações definidas.

Desenvolvimento de Lógica de Negócios:

Os programadores trabalham na implementação da lógica de negócios dos microsserviços. Eles usam o módulo Core para definir entidades, serviços e repositórios, aproveitando a capacidade de autogeração de código sempre que possível.

A personalização controlada é permitida para lidar com requisitos exclusivos do projeto.

Implementação de Controladores REST:

No módulo Rest, os programadores implementam os controladores RESTful que mapeiam as solicitações HTTP para as ações apropriadas nos microsserviços subjacentes.

Essa estratégia de implementação abrange todo o ciclo de vida da framework, desde a definição das especificações da API até a implementação e a manutenção. Ela enfatiza a eficiência, a consistência e a automação, permitindo o desenvolvimento ágil e seguro dos microsserviços.

5.9 Abrangência

O projeto em questão concentra-se na criação e implementação de uma framework inovadora, visando revolucionar a forma como o desenvolvimento de software é concebido e executado. A essência deste projeto reside na construção de uma estrutura sólida, flexível e altamente eficiente para o desenvolvimento de soluções baseadas em microsserviços.

O objetivo principal é proporcionar aos programadores uma infraestrutura que simplifique e agilize o processo de criação de software, permitindo-lhes focar no desenvolvimento do núcleo da aplicação. Esta arquitetura destina-se a reduzir a complexidade, oferecendo uma abordagem modular e flexível para o design e implementação de microsserviços.

Scope do Projeto:

Este projeto visa criar uma estrutura baseada em arquitetura de microsserviços para acelerar o desenvolvimento de aplicações. A abrangência do projeto compreende a concepção, a implementação e a validação dessa arquitetura, concentrando-se na criação de uma estrutura robusta e flexível. É importante realçar que o projeto não se concentra na construção de aplicações específicas, mas sim na elaboração de um conjunto de diretrizes, padrões e módulos que permitam o desenvolvimento ágil e eficiente de aplicações baseadas em microsserviços.

Objetivos do Projeto:

1. Desenvolvimento de uma Arquitetura de Microsserviços:

O foco principal é criar uma estrutura escalável, flexível e resiliente baseada em microsserviços.

2. Aceleração do Desenvolvimento de Aplicações:

Simplificar e agilizar o processo de desenvolvimento, permitindo que os programadores se concentrem no núcleo do desenvolvimento sem se preocupar com a complexidade da configuração.

3. Utilização de Tecnologias Robustas e Comprovadas:

Empregar tecnologias consolidadas e de renome, como Java e Spring Framework, conhecidas por sua robustez e maturidade.

4. Documentação Adequada:

Garantir a criação de documentação detalhada, especialmente sobre a API, permitindo uma compreensão clara e facilitando o uso dos microsserviços.

Limites do Projeto:

O projeto tem limitações em termos de scope funcional, pois não está focado na implementação de funcionalidades específicas para uma aplicação em particular. Em vez disso, concentra-se na criação de um ambiente estrutural e metodológico para a construção de microsserviços. O trabalho não abordará a implementação de todos os recursos ou soluções possíveis, mas oferecerá uma base sólida e flexível para a criação de aplicações futuras.

Potencial Impacto:

O projeto tem o potencial de influenciar positivamente o processo de desenvolvimento de software, proporcionando uma estrutura que simplifica e acelera a criação de aplicações baseadas em microsserviços. Ao fornecer uma arquitetura sólida e bem documentada, espera-se reduzir o tempo de desenvolvimento, minimizar erros comuns de configuração e promover a consistência na criação de serviços. Além disso, a abrangência do projeto abrange seu potencial impacto não apenas em ambientes acadêmicos, mas também no cenário empresarial, onde a agilidade no desenvolvimento de software é de grande importância.

Ao delinear a abrangência do projeto, é possível entender melhor suas fronteiras, objetivos e o potencial impacto que pode trazer para o desenvolvimento de software baseado em microsserviços. Este trabalho busca fornecer uma estrutura sólida para facilitar o desenvolvimento e promover a inovação na área de arquitetura de software.

5.10 Benefícios e Possíveis Desafios da Solução

A adoção dessa arquitetura traz consigo uma série de benefícios significativos que podem revolucionar a maneira como as organizações abordam o desenvolvimento de microsserviços. No entanto, é importante reconhecer que, embora ofereça muitas vantagens, também pode apresentar desafios que devem ser cuidadosamente considerados.

5.10.1 Benefícios

• Eficiência no Desenvolvimento:

- A geração automática de código a partir de especificações simplifica o processo de desenvolvimento de microsserviços, reduzindo consideravelmente o tempo gasto em

tarefas repetitivas de programação. Isso permite que os programadores estejam concentrados nas áreas mais críticas e inovadoras do projeto.

- **Organização e Normas:**

- A estrutura modular e a autogeração de código promovem a consistência e a aderência aos padrões de projeto estabelecidos. Isso resulta numa base de código mais organizada e de alta qualidade, facilitando a manutenção e a evolução contínua de microsserviços.

- **Controlo personalizado:**

- A capacidade de personalização controlada dá flexibilidade aos programadores. Eles podem ajustar o código gerado para atender aos requisitos específicos do projeto, mantendo um alto nível de controlo sobre a implementação.

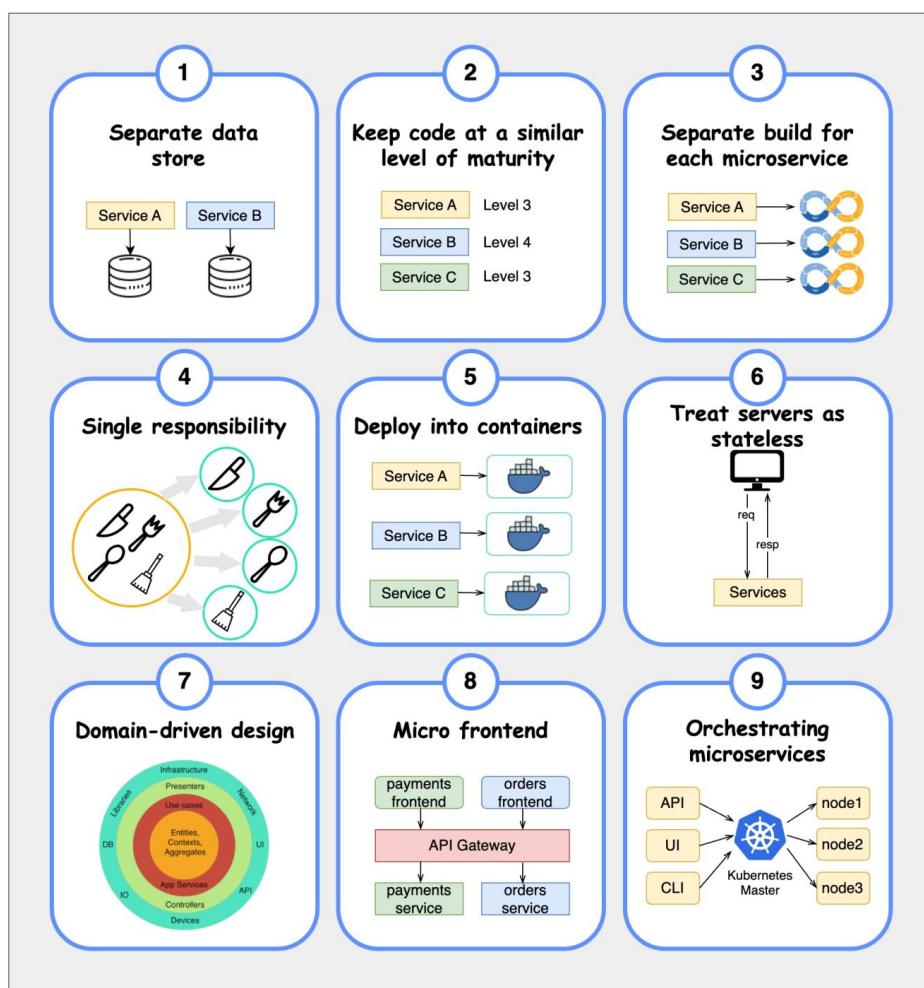


Figura 43, intitulada "Princípios de Design de Microsserviços", apresenta um infográfico com nove práticas recomendadas para a arquitetura de microsserviços. Entre elas estão a separação de armazenamento de dados, manter um nível de maturidade de código semelhante entre os serviços, construções separadas para cada serviço, princípio de responsabilidade única, implementação em contêineres, tratar servidores como stateless, design orientado a domínio, micro frontend e orquestração de microsserviços com ferramentas como Kubernetes.

5.10.2 Possíveis desafios (não benefícios):

- **Complexidade inicial:**
 - A introdução dessa arquitetura pode exigir um planeamento significativo e tempo de configuração inicial. Definir com precisão as especificações da API e configurar o ambiente pode ser um desafio inicial.
- **Curva de aprendizagem:**
 - Os programadores podem enfrentar uma curva de aprendizagem ao adotar essa abordagem, especialmente se não estiverem familiarizados com as ferramentas e tecnologias específicas envolvidas.
- **Personalização cautelosa:**
 - Embora a personalização seja uma vantagem, ela deve ser feita com cuidado. Modificações excessivas no código gerado podem prejudicar a manutenção futura e a conformidade com as normas.
- **Atualizações de especificações:**
 - Alterações nas especificações da API podem exigir ajustes significativos no código gerado. É importante estar preparado para lidar com essas atualizações de forma eficiente.
- **Requisitos de infraestrutura:**
 - A infraestrutura necessária para suportar esta arquitetura pode ser mais complexa do que nas abordagens tradicionais. É importante avaliar os requisitos de infraestrutura e garantir que a organização esteja preparada para gerenciá-los.

Ao considerar esses benefícios e desafios, podemos ter uma visão mais abrangente da adoção dessa arquitetura de microsserviços e tomar decisões informadas para maximizar seus pontos fortes e mitigar suas complexidades potenciais.

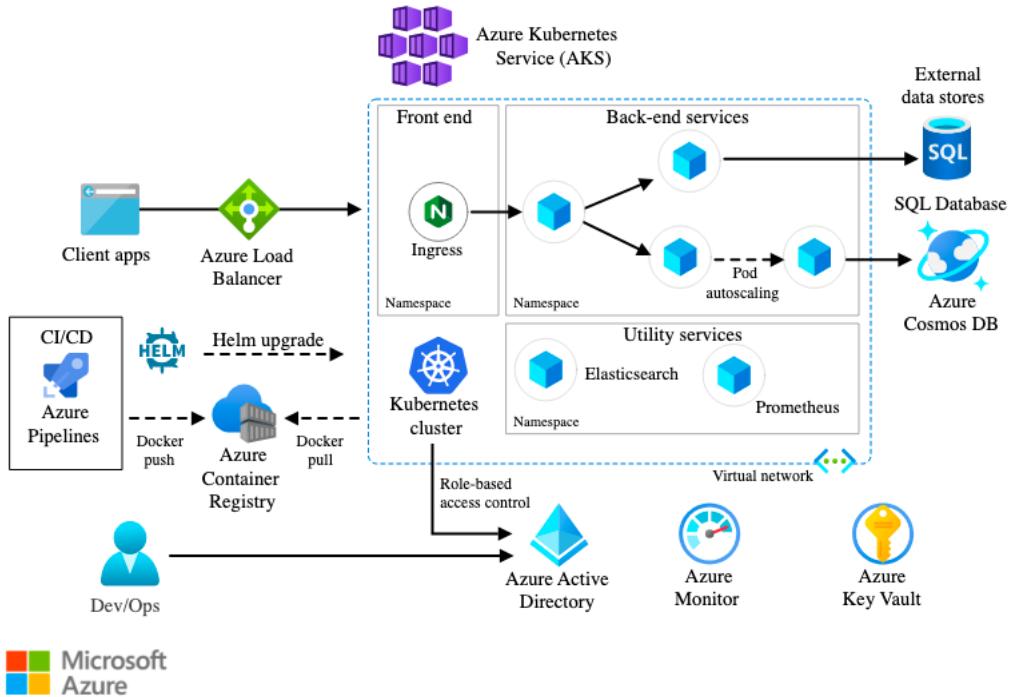


Figura 44, intitulada "*Arquitetura de Aplicação no Azure com Kubernetes*", exibe um diagrama a detalhar a integração de vários serviços do Microsoft Azure para operar uma aplicação com contêineres, com recurso ao Azure Kubernetes Service (AKS). Destaca-se a orquestração do CI/CD com Azure Pipelines, o load balancer, a gestão de identidades com o Azure Active Directory, e a segurança com o Azure Key Vault, além do monitoramento e armazenamento de dados com Azure Monitor e bases de dados SQL e Cosmos DB.

6 Método e Planeamento

Neste capítulo, abordaremos o método e o planeamento adotados para a conclusão bem-sucedida do Trabalho Final de Curso (TFC), com um foco específico na fase subsequente do projeto. O planeamento detalhado é essencial para orientar a gestão do projeto, garantir que os prazos sejam cumpridos e que os objetivos sejam alcançados. Além disso, forneceremos uma visão geral das tarefas realizadas até o momento, destacando eventuais desafios enfrentados e alterações que foram introduzidas em relação ao plano original.

6.1 Plano de Trabalho e Cronograma

Fase Subsequente do Projeto

A fase subsequente do projeto na nossa framework de microsserviços é crucial para o desenvolvimento e aprimoramento contínuo do sistema. Esta fase abrange uma série de tarefas essenciais que permitirão a evolução da framework e a preparação para a avaliação final. A seguir, apresentamos as principais tarefas planeadas:

- 1. Desenvolvimento de Novas Funcionalidades:** Nesta fase, concentrar-nos-emos na expansão das funcionalidades da framework de microsserviços. Isso incluirá o desenvolvimento de módulos adicionais, a implementação de novos recursos e a melhoria da lógica de negócios existente.
- 2. Refinamento da Arquitetura:** Continuaremos a refinar a arquitetura da framework, assegurando que ela siga as melhores práticas e padrões de desenvolvimento de microsserviços. Isso envolverá a otimização da comunicação entre os microsserviços, o ajuste de escalabilidade e a revisão de aspectos de segurança.
- 3. Testes e Validação:** Realizaremos extensos testes e validações para garantir que as novas funcionalidades sejam robustas e que a framework continue a operar de maneira confiável. Isso incluirá testes de unidade, testes de integração e testes de desempenho.
- 4. Documentação Atualizada:** Manteremos a documentação da framework atualizada, incluindo a documentação de API, modelos de dados, diagramas de casos de uso e qualquer outra documentação relevante. Isso é essencial para facilitar o uso da framework por programadores externos.
- 5. Preparação para a Avaliação Final:** Prepararemos os entregáveis necessários para a avaliação final do TFC. Isso incluirá relatórios detalhados, apresentações e demonstrações da framework.

Estimativas de Alto Nível

Para o trabalho posterior à fase subsequente do projeto, é importante fazer estimativas de alto nível que perspetivem as características dos entregáveis da avaliação final. Essas estimativas podem ser resumidas da seguinte forma:

1. Entregáveis Finais: Os principais entregáveis da avaliação final incluirão um relatório completo do TFC, documentação detalhada da framework, apresentações, demonstrações e, se aplicável, código-fonte completo da framework.

2. Testes Finais: Realizaremos testes finais abrangentes para garantir a estabilidade e o desempenho da framework antes da avaliação final. Isso incluirá testes de carga, teste de segurança e testes de usabilidade.

3. Preparação da Apresentação: A preparação da apresentação para a avaliação final será uma tarefa crítica. Isso envolverá a criação de material de apresentação, a prática de demonstrações e a elaboração de um discurso claro e conciso.

6.2 Progresso do Trabalho até o Momento

Durante as fases iniciais do projeto, foram realizadas uma série de tarefas críticas que serviram como alicerce para o desenvolvimento da framework de microsserviços. Abaixo, fornecemos mais detalhes sobre essas tarefas:

1. Análise de Requisitos:

Esta foi a fase inicial do projeto, na qual conduzimos uma análise detalhada dos requisitos para a framework de microsserviços. Trabalhamos em estreita colaboração com os stakeholders e partes interessadas para identificar suas necessidades e expectativas. Isso envolveu a definição de requisitos funcionais e não funcionais, além da priorização de funcionalidades essenciais.

2. Design da Arquitetura:

Com base na análise de requisitos, avançamos para a fase de design da arquitetura. Aqui, projetamos uma arquitetura sólida para a framework, levando em consideração os princípios fundamentais de microsserviços. Isso incluiu a definição da estrutura de componentes, a identificação de serviços-chave e a criação de diagramas de arquitetura para orientar o desenvolvimento posterior.

3. Desenvolvimento Inicial:

Com o design da arquitetura em vigor, iniciamos o desenvolvimento inicial da framework. Começamos a implementar os principais componentes e funcionalidades conforme definidos na fase de análise de requisitos. Isso envolveu a escrita de código, a configuração de ambientes de desenvolvimento e a integração de tecnologias relevantes.

4. Testes Preliminares:

À medida que o desenvolvimento avançava, realizamos testes preliminares para validar as funcionalidades iniciais da framework. Isso incluiu testes de unidade para garantir que cada componente funcionasse conforme o esperado, testes de integração para verificar a interação entre os diferentes microsserviços e testes de desempenho para avaliar o desempenho geral do sistema. Esses testes iniciais foram essenciais para identificar áreas que requeriam melhorias e refinamentos.

5. Desenvolvimento Parcial da Framework:

Foi alcançado um desenvolvimento suficiente que permitiu avançar para a publicação da biblioteca. Esse progresso parcial também demonstrou a capacidade da framework de ser flexível e modular, o que é fundamental no contexto de microsserviços.

6. Publicação e Documentação:

Uma etapa significativa foi a publicação de uma biblioteca feita a partir da nossa framework no Maven Repository, tornando-a acessível a outros desenvolvedores e facilitando a integração com projetos existentes. Acompanhando esta ação, escrevi um artigo detalhado, disponível no [Medium](#), intitulado "Publishing Your Java Library to Maven Central: A Step-by-Step Tutorial", que serve como um guia para outros desenvolvedores que desejam realizar o mesmo processo.

7. Desenvolvimento de um Dockerfile:

Para facilitar a implantação e a portabilidade da framework, desenvolvemos um Dockerfile. Esse arquivo descreve as etapas necessárias para criar uma imagem Docker da framework, garantindo que ela possa ser executada de maneira consistente em qualquer ambiente que suporte Docker.

Durante essas fases, enfrentamos desafios técnicos e decisões importantes de design. A colaboração próxima com a equipa de desenvolvimento, bem como a consulta regular aos stakeholders, desempenharam um papel fundamental na superação desses desafios e na garantia de que o projeto continuasse alinhado com seus objetivos e requisitos.

É importante destacar que o progresso até o momento tem sido significativo, e a framework de microsserviços está tomando forma. No entanto, reconhecemos que ainda há muito trabalho pela frente na próxima fase do projeto, conforme detalhado no planeamento anteriormente apresentado. A experiência e os insights adquiridos nas fases iniciais servirão como base sólida para o desenvolvimento futuro da framework.

6.3 Dificuldades e Alterações ao Plano Inicial

Ao longo do desenvolvimento da framework de microsserviços, deparamo-nos com várias dificuldades que requereram uma abordagem cuidadosa e algumas adaptações ao plano inicial. É importante destacar que esses desafios foram oportunidades para aprender e melhorar a qualidade da solução em desenvolvimento. A seguir, detalhamos algumas das principais dificuldades enfrentadas e as alterações feitas no plano:

1. Complexidade Técnica: A arquitetura de microsserviços é intrinsecamente complexa devido à sua natureza distribuída. Durante a fase de desenvolvimento inicial, surgiram desafios técnicos inesperados relacionados à comunicação entre os microsserviços, gestão de dados distribuídos e escalabilidade. Essas complexidades exigiram uma análise aprofundada e a busca por soluções técnicas sólidas.

2. Integração de Tecnologias: A integração de diversas tecnologias e frameworks para construir a framework de microsserviços apresentou desafios de compatibilidade e interoperabilidade. Foram necessárias adaptações no plano de desenvolvimento para garantir que todas as tecnologias funcionassem em conjunto de maneira harmoniosa.

3. Ajustes de Scope: À medida que a framework tomava forma, surgiram insights adicionais sobre requisitos e funcionalidades que poderiam melhorar significativamente o valor da solução. Isso levou a ajustes de scope para incluir funcionalidades adicionais e otimizações de desempenho que não estavam originalmente previstas.

4. Testes e Garantia de Qualidade: Garantir a qualidade e estabilidade da framework foi uma prioridade. Isso exigiu mais tempo dedicado à realização de testes abrangentes e à resolução de problemas identificados durante os testes. Alterações no plano de testes foram feitas para abordar essas preocupações.

5. Recursos Humanos e Temporais: À medida que o projeto avançava, ficou claro que a disponibilidade de recursos humanos e o cronograma original precisariam ser revistos para acomodar as complexidades técnicas e as novas funcionalidades. Isso envolveu uma realocação de tarefas e a definição de prazos mais realistas.

6. Comunicação e Colaboração: A natureza distribuída do projeto, com equipes trabalhando em diferentes aspectos da framework, destacou a importância da comunicação e colaboração eficazes. Foram implementadas práticas de comunicação mais rigorosas e reuniões regulares para garantir que todos estivessem alinhados com os objetivos do projeto.

7. Alterações na Documentação: Conforme a framework evoluía, a documentação também precisava ser atualizada para refletir as mudanças e melhorias realizadas. Isso incluiu a revisão de manuais de utilizador, documentação técnica e tutoriais para programadores.

Em resposta a essas dificuldades e desafios, a equipa adotou uma abordagem flexível e orientada para soluções. As alterações no plano foram documentadas e comunicadas às partes interessadas, garantindo a transparência e a compreensão comum das mudanças necessárias. É importante realçar que essas dificuldades não comprometeram o compromisso de entregar uma framework de microsserviços de alta qualidade e eficácia. Pelo contrário, elas fortaleceram a equipa e o projeto, resultando em melhorias significativas e uma compreensão mais profunda dos desafios associados à arquitetura de microsserviços.

Conclusão

O método e o planeamento são elementos cruciais para o sucesso do Trabalho Final de Curso. O planeamento detalhado para a fase subsequente do projeto, juntamente com estimativas de alto nível para o trabalho posterior, ajudarão a orientar o desenvolvimento da framework de microsserviços. Além disso, o progresso até o momento, as dificuldades enfrentadas e as alterações realizadas garantem que o projeto esteja bem encaminhado para atingir seus objetivos finais. A fase subsequente será fundamental para a expansão e aprimoramento contínuos da framework, preparando-a para a avaliação final e demonstrando seu potencial e robustez como uma solução de microsserviços.

7 Notas Finais

Neste estágio intermédio do trabalho final de curso (TFC), é importante realçar alguns pontos cruciais relacionados à framework de microsserviços desenvolvida, as suas implicações e direções futuras.

Primeiramente, é fundamental enfatizar que a menção a tecnologias como Load Balancer, API Gateway ou Kafka neste TFC não implica necessariamente sua implementação imediata. Dado o scope e as restrições de tempo deste trabalho, a nossa concentração principal está na criação da estrutura e na concepção da framework em si. Essas tecnologias são referidas como considerações valiosas para trabalhos futuros, que podem ocorrer fora do âmbito deste TFC.

É relevante mencionar essas mesmas tecnologias, pois representam recursos poderosos e soluções potenciais para desafios complexos nas arquiteturas de microsserviços. Aqui, essas menções servem como pontos de partida para a exploração de possibilidades de expansão e aprimoramento da framework em projetos subsequentes. Destacamos o seu potencial e relevância no contexto do desenvolvimento de software.

Além disso, é importante observar que o desenvolvimento de uma framework de microsserviços é uma empreitada significativa, que envolve uma compreensão profunda das complexidades da arquitetura de microsserviços. O TFC representou um ponto de partida sólido, fornecendo uma estrutura inicial que pode ser continuamente aprimorada e adaptada às necessidades específicas de diferentes projetos e organizações.

Ao longo deste TFC, foram abordados tópicos essenciais relacionados à arquitetura de microsserviços, incluindo a análise de requisitos, a concepção da arquitetura, o desenvolvimento da framework, testes preliminares e considerações de segurança. Esses esforços iniciais fornecem uma base sólida para o desenvolvimento futuro, e a documentação detalhada disponível servirá como um recurso valioso para programadores, administradores de sistema e outros profissionais de TI que desejam aproveitar essa framework.

No que diz respeito às próximas etapas, é recomendável que a framework seja continuamente testada, refinada e adaptada às necessidades reais de projetos específicos. Isso envolverá a implementação de funcionalidades adicionais, a otimização do desempenho, a garantia de segurança contínua e a integração de tecnologias complementares, conforme apropriado.

Por fim, é crucial manter uma abordagem ágil e flexível ao desenvolvimento e à evolução da framework de microsserviços. À medida que novos desafios surgirem e as *demands* do mercado evoluírem, a framework poderá se adaptar e continuar a ser uma ferramenta valiosa no desenvolvimento de aplicações baseadas em microsserviços.

Em resumo, este TFC representa um ponto de partida sólido para a criação de uma framework de microsserviços eficaz e versátil. À medida que a framework é utilizada em cenários do mundo real e refinada com base nas lições aprendidas, ela tem o potencial de se tornar uma ferramenta indispensável no arsenal de programadores de software que buscam aproveitar ao máximo as vantagens da arquitetura de microsserviços.

8 Calendário

Na figura abaixo, está representado o cronograma em formato Gantt, que apresenta de forma visual as tarefas, datas de início e término, duração estimada em dias e os membros responsáveis por cada fase do projeto. Este gráfico oferece uma representação clara e organizada do planeamento das atividades a serem realizadas.

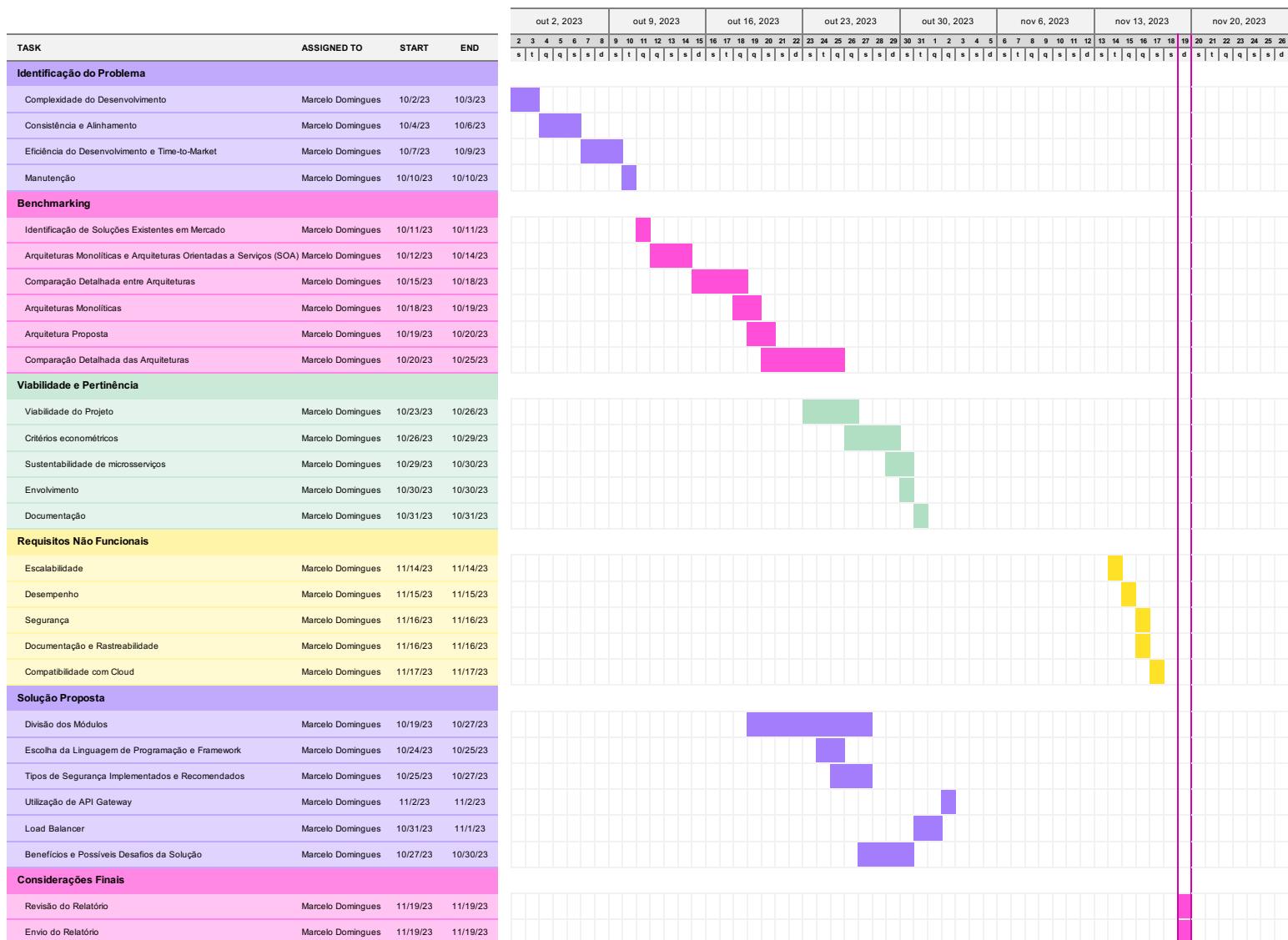


Figura 45, intitulada "Cronograma em Gantt, realizado em Excel", apresenta um gráfico de Gantt colorido detalhando o cronograma de tarefas e marcos de um projeto. As tarefas são divididas em categorias como Identificação do Problema, Benchmarking, Viabilidade e Pertinência, entre outras, com colunas indicando a pessoa responsável, datas de início e fim, e a duração representada por barras coloridas que se estendem ao longo de um calendário na parte superior.

Este cronograma servirá como guia ao longo do restante do TFC, permitindo uma gestão eficiente do tempo e uma avaliação contínua do progresso em relação aos objetivos definidos.

Bibliografia

- [1] TCGen. "Time-to-Market." Disponível em: <https://www.tcgen.com/time-to-market/> (Acedido em 3 de Outubro de 2023).
- [2] Medium. "Microservice Architecture." Disponível em: <https://medium.com/design-microservices-architecture-with-patterns/microservices-architecture-for-enterprise-large-scaled-application-825436c9a78a> (Acedido em 12 de Outubro de 2023).
- [3] Smartway2. "Digital Transformation." Disponível em: <https://smartway2.com/blog/8-successful-examples-of-digital-transformation/> (Acedido em 7 de Outubro de 2023).
- [4] Micro Focus. "Ciclo de Desenvolvimento." Disponível em: <https://www.microfocus.com/en-us/what-is/sdlc> (Acedido em 8 de Outubro de 2023).
- [5] Medium. "Controlo de Versões." Disponível em: <https://medium.com/@it20240042/what-is-version-control-system-31a8a83c60a4> (Acedido em 15 de Outubro de 2023).
- [6] Medium. "Arquitetura de Serviços." Disponível em: <https://medium.com/@JalelTounsi/monolith-soa-microservices-or-serverless-43dd60e29756> (Acedido em 11 de Outubro de 2023).
- [7] Medium. "Arquitetura Monolítica." Disponível em: <https://medium.com/design-microservices-architecture-with-patterns/monolithic-architecture-is-still-worth-at-2021-98bfc112dc24> (Acedido em 4 de Outubro de 2023).
- [8] The New Stack. "Comparação Operacional." Disponível em: <https://thenewstack.io/microservices/microservices-vs-monoliths-an-operational-comparison/> (Acedido em 9 de Outubro de 2023).
- [9] Swagger. "Swagger Editor." Disponível em: <https://editor.swagger.io> (Acedido em 6 de Outubro de 2023).
- [10] LinkedIn. "API Gateway." Disponível em: https://www.linkedin.com/posts/bytebytogo_systemdesign-coding-interviewtips-activity-7108328688890339328-AOyC?utm_source=share&utm_medium=member_desktop (Acedido em 10 de Outubro de 2023).
- [11] LinkedIn. "Arquiteturas de Software." Disponível em: https://www.linkedin.com/feed/update/urn:li:activity:7084195892504743936?utm_source=share&utm_medium=member_desktop (Acedido em 14 de Outubro de 2023).
- [12] JetBrains. "Java Docs." Disponível em: <https://www.jetbrains.com/help/idea/2023.1/working-with-code-documentation.html> (Acedido em 13 de Outubro de 2023).
- [13] SuperTokens. "Estrutura JWT." Disponível em: <https://supertokens.com/blog/what-is-jwt> (Acedido em 17 de Outubro de 2023).
- [14] Medium. "Escalabilidade de Microsserviços." Disponível em: <https://medium.com/cloud-native-daily/scaling-microservices-a-comprehensive-guide-200737d75d62> (Acedido em 5 de Outubro de 2023).
- [15] Madappgang. "Scalability in Software Development." Disponível em: <https://madappgang.com/blog/scalability-in-software-development/> (Acedido em 1 de Novembro de 2023).
- [16] Sayonetech. "Microservice Performance" Disponível em: <https://www.sayonetech.com/blog/microservices-performance/> (Acedido em 2 de Novembro de 2023).

- [17] OWASP. "Security Best Principles." Disponível em: <https://owasp.org/www-project-developer-guide/draft/04-foundations/03-security-principles> (Acedido em 3 de Novembro de 2023).
- [18] API Evangelist. "Best Practices for API Documentation." Disponível em: <https://apievangelist.com/info/101/> (Acedido em 4 de Novembro de 2023).
- [19] Apache Kafka Documentation. "Getting Started with Apache Kafka." Disponível em: <https://kafka.apache.org/documentation/#gettingStarted> (Acedido em 6 de Novembro de 2023).
- [20] Martin Fowler. "Microservices." Disponível em: <https://martinfowler.com/articles/microservices.html> (Acedido em 7 de Novembro de 2023).
- [21] Microsoft Azure. "Azure Documentation." Disponível em: <https://docs.microsoft.com/en-us/azure/> (Acedido em 8 de Novembro de 2023).
- [22] Red Hat. "Understanding Service-Oriented Architecture." Disponível em: <https://www.redhat.com/pt-br/topics/cloud-native-apps/what-is-service-oriented-architecture> (Acedido em 9 de Novembro de 2023).
- [23] Docker. "Docker Documentation." Disponível em: <https://docs.docker.com/> (Acedido em 10 de Novembro de 2023).
- [24] Spring Framework. "Why Spring Boot." Disponível em: <https://spring.io/projects/spring-boot#overview> (Acedido em 12 de Novembro de 2023).
- [25] Apache Kafka Documentation. "Introduction to Apache Kafka." Disponível em: <https://kafka.apache.org/intro> (Acedido em 13 de Novembro de 2023).
- [26] Network World. "Server Load Balancer." Disponível em: <https://www.networkworld.com/article/831036/data-center-server-load-balancing-a-practical-guide.html> (Acedido em 14 de Novembro de 2023).
- [27] Confluent. "Why Apache Kafka?" Disponível em: <https://www.confluent.io/what-is-apache-kafka/> (Acedido em 15 de Novembro de 2023).
- [28] DZone. "Microservices Benefits and Challenges." Disponível em: <https://dzone.com/articles/microservices-benefits-and-challenges> (Acedido em 16 de Novembro de 2023).
- [29] Medium - Marcio Nizzola. "Quer gerar códigos prontos para consultas à API's? Use o Swagger Editor." Disponível em: <https://marcionizzola.medium.com/quer-gerar-c%C3%B3digos-prontos-para-consultas-%C3%A0-api-s-use-o-swagger-editor-151cd933564c> (Acedido em 20 de Novembro de 2023).
- [30] void.co.mz. "Apache Kafka: O que é e como funciona?" Disponível em: <https://void.co.mz/2021/12/02/apache-kafka-o-que-e-e-como-funciona/> (Acedido em 20 de Novembro de 2023).
- [31] NetScaler. "What Is Load Balancing?" Disponível em: <https://www.netscaler.com/articles/what-is-load-balancing> (Acedido em 20 de Novembro de 2023).
- [32] LinkedIn - Iago Ferreira. "Recentemente publicamos uma série de cursos." Disponível em: https://pt.linkedin.com/posts/iagoferreirati_recentemente-publicamos-uma-serie-de-cursos-activity-7091199271328215040--2gU?trk=public_profile_like_view (Acedido em 20 de Novembro de 2023).

- [33] Byte Byte Go. "Ep67 - Top 9 Microservice Best Practices." Disponível em: <https://blog.bytebytogo.com/p/ep67-top-9-microservice-best-practices> (Acedido em 20 de Novembro de 2023).
- [34] Microsoft Azure. "High Availability Kubernetes." Disponível em: <https://learn.microsoft.com/pt-pt/azure/architecture/example-scenario/hybrid/high-availability-kubernetes> (Acedido em 20 de Novembro de 2023).
- [35] Medium - Marcio Nizzola. "Quer gerar códigos prontos para consultas à API's? Use o Swagger Editor." Disponível em: [https://marcionizzola.medium.com/quer-gerar-códigos-prontos-para-consultas-%C3%A0-api-s-use-o-swagger-editor-151cd933564c](https://marcionizzola.medium.com/quer-gerar-c%C3%B3digos-prontos-para-consultas-%C3%A0-api-s-use-o-swagger-editor-151cd933564c) (Acedido em 18 de Janeiro de 2024).
- [36] Prototypr. "Outlining a JSON Structure to Centralize Data for Your Design System." Disponível em: <https://blog.prototypr.io/outlining-a-json-structure-to-centralize-data-for-your-design-system-3ed31afe0d45> (Acedido em 18 de Janeiro de 2024).
- [37] Medium – Marcelo Domingues. " Publishing Your Java Library to Maven Central: A Step-by-Step Tutorial." Disponível em <https://medium.com/@marcelogdomingues/publishing-your-java-library-to-maven-central-a-step-by-step-tutorial-9a15f7edb3af> (Acedido em 13 de Abril de 2024).
- [38] Youtube – Marcelo Domingues. “Engenharia de Microserviços - Marcelo Domingues” Disponível em <https://youtu.be/Btv8OaevSGs> (Acedido em 14 de Abril de 2024)
- [38] Youtube – Marcelo Domingues. “Engenharia de Microserviços - Marcelo Domingues - Final” Disponível em <https://youtu.be/9kviCGXyMk4> (Acedido em 28 de Junho de 2024)

Glossário

LEI	Licenciatura em Engenharia Informática
TFC	Trabalho Final de Curso
API	Application Programming Interface
REST	Representational State Transfer
UI	User Interface
ROI	Return on Investment
IT	Information Technology
Tech	Technology
JWT	Json Web Tokens
CSRF	Cross-Site Request Forgery
SOA	Service-Oriented Architecture
CRM	Customer Relationship Management
RDBMS	Relational Database Management System
ESB	Enterprise Service Bus
AKS	Azure Kubernetes Service
JSON	JavaScript Object Notation
SOAP	Simple Object Access Protocol
CI/CD	Continuous Integration/ Continuous Delivery
AWS	Amazon Web Services
SQL	Structured Query Language
XML	Extensible Markup Language