UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Mauro Humberto de Oliveira Júnior

Microprofile e a evolução de suas implementações

Uberlândia, Brasil

2022

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Mauro Humberto de Oliveira Júnior

Microprofile e a evolução de suas implementações

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, Minas Gerais, como requisito exigido parcial à obtenção do grau de Bacharel em Sistemas de Informação.

Orientador: Luiz Claudio Theodoro

Universidade Federal de Uberlândia – UFU

Faculdade de Ciência da Computação

Bacharelado em Sistemas de Informação

Uberlândia, Brasil

2022

Mauro Humberto de Oliveira Júnior

Microprofile e a evolução de suas implementações

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, Minas Gerais, como requisito exigido parcial à obtenção do grau de Bacharel em Sistemas de Informação.

Trabalho aprovado. Uberlândia, Brasil, 24 de novembro de 2012:

Luiz Claudio Theodoro
Orientador

Professor

Professor

Uberlândia, Brasil
2022

Agradecimentos

Agradeço primeiramente ao professor Luiz Claudio Theodoro pela orientação na criação deste trabalho. Agradeço ao Felipe Mizara que contribuiu de forma espontânea e ilustre neste trabalho, me orientando e participando da criação do caso de uso com microprofile demonstrado neste trabalho, e a todos que de alguma forma fizeram parte da minha formação acadêmica.

Resumo

Este trabalho tem como objetivo estudar a tecnologia Microfile Java, com o intuito de análisar sua evolução e sua utilização no desenvolvimento de aplicações. Com o avanço da tecnologia, arquiteturas monotilicas estão cada vez mais em desuso, e a arquitetura de microserviços vem dominando o mercado. O Microprofile Java chega com o intuito de melhorar a performance de microserviços, trazendo consigo além de implementações técnicas, uma série de conceitos para que possamos criar aplicações independentes, performaticas e nativas do mundo da computação em nuvem.

Palavras-chave: Microprofile, Java, Microserviços, JavEE.

Lista de ilustrações

Lista de abreviaturas e siglas

JVM	Java Virtual Machine
HTTP	HyperText Transfer Protocol
TI	Tecnologia da Informação
CDI	Injeção de Dependência e Contextos Java
JSF	JavaServer Faces
EJB	Enterprise JavaBeans
Java EE	Java Enterprise Edition
IoC	Inversion of Control
XML	Extensible Markup Language
API	Application Programming Interface
EL	Expression Language
JSF	JavaServer Faces
SPI	Service Provider Interface
CI	Continuous Integration
CD	Continuous Deployment
JDBC	Java Database Connectivity
EJB	Enterprise Java Beans
JTA	Java Transaction API
JSP	Java Server Pages
JPA	Java Persistence API
JCA	Java Connector Architecture
JSF	Java Server Faces
JMS	Java Message Service

MSA	Microservice Architecture
OAS	Open API
OICD	OpenIS Connect
JWT	JSON Web Tokens
REST	Representational State Transfer
IDE	Ambiente de desenvolvimento integrado
JRE	Java Runtime Environment
CVE	Common Vulnerabilities and Exposures

Sumário

1 Introdução

Uma das grandes disrupções que tivemos na evolução da área de TI (Tecnologia da Informação) foi a história dos contextos e da injeção de dependências que podemos referenciar como CDI. Neste caminho, tecnologias impactantes como JSF (JavaServer Faces) e EJB (Enterprise JavaBeans) foram fundamentais (BEERNINK; TIJMS, 2019). A primeira versão do Java EE (J2EE na época) introduziu o conceito de inversão de controle (IoC), o que significa que o container assumiria o controle de seu código de negócios e forneceria serviços técnicos (como transações ou gerenciamento de segurança). Assumir o controle significava gerenciar o ciclo de vida dos componentes, trazendo injeção de dependência e configuração para seus componentes. Esses serviços foram incorporados ao container e os programadores tiveram que esperar até versões posteriores do Java EE para ter acesso a eles. A configuração de componentes foi possível nas primeiras versões com descritores de implantação XML, mas tivemos que esperar que o Java EE 5 e o Java EE 6 tivessem uma API fácil e robusta para fazer o gerenciamento do ciclo de vida e a injeção de dependência (GONCALVES, 2013).

O CDI introduziu uma abordagem para melhor interação entre seus componentes (Java EE), com um avançado modelo de ciclo de vida. Podemos citar, com base numa publicação da Devmedia, entre os principais recursos adicionados pelo CDI ao Java EE, teremos os seguintes aspectos:

- Integração unificada de Expression Language (EL): Permite que qualquer Bean do CDI possa ser exposto em um componente JSF;
- Eventos: CDI permite um mecanismo de troca de notificações (Eventos) Type safe;
- Injeção de Dependência: CDI possibilita injeção de Dependência Type safe de qualquer componente Java EE;
- Métodos Produtores: Traz uma abordagem para criar injeções polimórficas durante a execução;
- Decorators: CDI utiliza o padrão de projeto Decorator para desacoplar questões técnicas da lógica de negócio;
- Interceptors: Definido na especificação Java Interceptors, recurso que cria um meio para interceptar métodos e fazer algum tipo de processamento no mesmo.
- Conversation Scope: CDI adiciona um novo escopo além dos já definidos Request, Application e Session;

- Service Provider Interface (SPI): Permite a integração de frameworks third-party no ambiente Java EE.

Na sequência de evolução, foi proposta a arquitetura de microsserviços, uma abordagem popular para criar aplicativos nativos na nuvem, de modo que que cada componente é um serviço individual que cumpre um propósito específico. Ele permite que equipes reduzidas e autônomas, possam desenvolver, implantar e dimensionar seus respectivos serviços de forma independente. Uma vantagem substancial é que o aplicativo pode ser dimensionado em um nível mais granular porque cada serviço é construído e gerenciado de forma independente. Mesmo em tráfego intenso, serviços podem ser dimensionados individualmente para usar os recursos com eficiência, em vez de ampliar o sistema inteiro. Outro benefício é que as falhas em um serviço podem ser isoladas do resto do sistema; se um serviço falhar, os serviços independentes não serão afetados enquanto serviços dependentes podem empregar estratégias de tolerância a falhas para evitar que a falha cascata para outros serviços (CAI et al., 2018).

Assim, surgiu o MicroProfile, um conjunto modular de tecnologias projetadas para que um desenvolvedor possa escrever microsserviços nativamente no Java na nuvem. Cloud-native é uma abordagem de todo o setor para desenvolver e implantar rapidamente aplicativos para a nuvem em escala. Os aplicativos nativos da nuvem são projetados em torno de microsserviços alinhados à equipe e desenvolvidos usando práticas ágeis e integração/entrega contínua (CI/CD) para simplificar a implantação. Com uma gama de fornecedores que fornecem plataformas em nuvem, código aberto e padrões abertos são essenciais facilitadores para evitar o aprisionamento do fornecedor. O MicroProfile permite desenvolver e implantar aplicativos Java nativos da nuvem como serviços leves e pouco acoplados, cada um representando uma função de negócios exclusiva. Esta abordagem é modular e torna a aplicação fácil de entender, fácil de desenvolver, fácil de testar e fácil de manter (WETHERBEE et al., 2018).

Efetivamente, o modelo de programação CDI pode auxiliar no desacoplamento de aplicações preexistentes. As empresas aprenderam como transformar sua estratégia de deployment de forma transparente usando CDI extensions, e assim, o desacoplamento permitiu o deploy das aplicações em ambientes de Cloud. Isto levou à mudança de paradigma da decomposição de um monolito Java EE para a arquitetura de Microserviços. Este trabalho tem o objetivo de apresentar o MicroProfile, detalhando suas implementações, destacando cases que possam mostrar claramente algumas questões de ordem prática para que sirva de roteiro ou orientação para quem quiser se aventurar nesta tecnologia.

2 Historico de versões do Java EE

Java é uma linguagem de programação e plataforma computacional lançada pela primeira vez pela Sun Microsystems na década de 90 e utilizada em cerca de 97% dos desktops corporativos pelo mundo segundo a Oracle, companhia que detém os direitos do software nos dias atuais, apesar de, em 2017, a Oracle ter passado a administração da plataforma para a Eclipse Foundation, com o intuito de criar uma solução de código aberto Obtenha... (2022). Muitas aplicações e sites que utilizamos nos dias de hoje são dependentes desta linguagem, se fazendo necessário que o Java esteja instalado na máquina para que se possa executar o código desejado. Isso se deve ao fato da tecnologia Java nos dar a possibilidade de escrever o código apenas uma vez e rodá-lo em diferentes dispositivos e ambientes, pois o software não é compilado em "código nativo" para ser executado diretamente pelo computador, mas sim em um código intermediário chamado "bytecode", que então é interpretado e executado pela máquina virtual Java JVM. De laptops a datacenters, consoles de games a supercomputadores científicos, telefones celulares à Internet, o Java pode ser encontrado em várias aplicações e serviços.
A tecnologia Java é composta de um grande número de tecnologias que se dividem entre o ambiente de desenvolvimento e de execução de software. Para que a tecnologia atenda a essa grande gama de aplicações e ambientes, o Java é dívido em quatro grandes plataformas, são elas:

- Java SE - Java Standard Edition;
- Java EE - Java Standard Edition;
- Java ME - Java Micro Edition;
- Java FX - Software Multimedia Platform;

Para falarmos sobre Microprofile Java, precisaremos entender um pouco mais sobre o Java EE, ou Java Platform Enterprise Edition, também chamado de Jakarta EE nos dias atuais. O Java EE é uma plataforma ou ambiente para desenvolvimento de aplicações de grande porte e aplicações web que possui bibliotecas e funcionalidades que implementam softwares baseados na linguagem Java. Sua principal utilização é para o desenvolvimento de software em ambientes corporativos, com aplicações como web e rede, além de outros recursos seguros, escaláveis e multicamadas.
Diferentemente da Java Standard Edition, conhecida como Java SE, a Java EE trás consigo uma adição de bibliotecas que, por meio das suas funcionalidades, tornam possível implantar software Java multicamadas, distribuído e tolerante a erros.

A arquitetura Java EE define o padrão para desenvolvimento de aplicações corporativas multi-camadas. Além disso, a arquitetura define um conjunto de especificações que são implementadas por diferentes empresas através dos chamados Servidores da Aplicações Java EE. O Java EE define um modelo de programação para criar aplicações onde diversas tarefas comuns, como por exemplo, persistência de dados, validações, transacionais, tratamento de requisições HTTP, e etc. são especificadas.
Dentre as funcionalidades que o Java EE nos apresenta, temos: Java... (2022)

- Java Database Connectivity (JDBC): Usado para acesso a bancos de dados.
- Enterprise Java Beans (EJBs): Usados para desenvolver componentes de software. Graças a esses recursos, o programador consegue focar sua atenção nas necessidades de cada cliente, porque detalhes como escalabilidade, segurança, disponibilidade e infraestrutura ficam sob responsabilidade do servidor de aplicações.
- Servelets: Esse recurso é usado para desenvolver aplicações web com conteúdo dinâmico. Ele é composto por uma API que tem como objetivo fornecer ao programador os recursos do servidor web de uma forma mais simples e fácil de operar.
- Java Transaction API (JTA): Trata-se de uma API que busca padronizar o tratamento de transações em uma aplicação Java.
- Java Server Pages (JSP): Pode ser conceituado como uma especialização do servlet que possibilita ao programador desenvolver com mais facilidade conteúdos dinâmicos.
- Java Persistence API (JPA): Trata-se de uma API que tem como objetivo padronizar o acesso a banco de dados por meio de um mapeamento Objeto Relacional dos Enterprise Java Beans.
- Java Connector Architecture (JCA): Consiste em uma API que uniformiza a ligação a aplicações legadas.
- Java Server Faces (JSF): Trata-se de uma especificação Java que ajuda o programador na construção de interfaces de usuário a partir de componentes para aplicação em ambiente web.
- Java Message Service (JMS): Por meio desta API para middleware orientada a mensagens, torna-se possível a comunicação entre aplicações de forma assíncrona.

No ano de seu lançamento, a plataforma era conhecida como Java 2 Platform, Enterprise Edition ou J2EE, nome este que foi alterado para Java EE em 2006. Atualmente, a versão mais recente do Java EE é a Java EE 8. Vejamos um histórico de versões do Java EE:

- J2EE 1.2 (12 de dezembro de 1999)
- J2EE 1.3 (24 de setembro de 2001)
- J2EE 1.4 (11 de novembro de 2003)
- Java EE 5 (11 de maio de 2006)
- Java EE 6 (10 de dezembro de 2009)
- Java EE 7 (28 de maio de 2013,[3] - 5 de abril de 2013 de acordo com o documento de especificação)
- Java EE 8 (31 de Agosto de 2017)

3 Microserviços

Este capítulo tem por objetivo trazer uma pequena amostragem sobre a arquitetura de software conhecida como microserviços. Por fim, abortar um pouco de sua evolução e como sua utilização no campo da tecnologia vem sendo cada vez mais adotada.
Microserviços pode ser considerado um framework de arquitetura de software inspirado na computação orientada a serviços que tem bastante adesão no desenvolvimento de novas tecnologias, e basicamente consiste em pequenos serviços independentes que se comunicam usando APIs bem definidas, trazendo consigo algumas caracteristicas básicas aos serviços criados, como autônomia, especialização de serviço, velocidade de comunicação, escalabilidade, rastreabilidade e outros.
Com a adoção de uma arquitetura de microserviços, as aplicações são desmembradas em componentes mínimos e independentes. Diferentemente da abordagem tradicional monolítica em que toda a aplicação é criada como um único bloco, os microsserviços são componentes separados que trabalham juntos para realizar as mesmas tarefas. Cada um dos componentes ou processos é um microsserviço. Essa abordagem de desenvolvimento de software valoriza a granularidade, a leveza e a capacidade de compartilhar processos semelhantes entre várias aplicações. Trata-se de um componente indispensável para a otimização do desenvolvimento de aplicações para um modelo nativo em nuvem, por exemplo. What... (2022)
A adoção de uma arquitetura monolítica no desenvolvimento de aplicações em geral, pode criar complicações quando falamos em escalabilidade, desenvolvimento, reutilização de código, segurança, dentre outros. A arquitetura de microsserviços surgiu como uma alternativa a estes problemas, e hoje é vista como um dos principais métodos de desenvolvimento de software. A arquitetura de microserviços (MSA) traz consigo uma gama de vantagens que fazem com que o desenvolvimento de aplicações monolíticas não seja mais vantajoso na maioria dos casos. Um relatório da consultoria O'Reilly aponta que 92% das empresas que utilizavam o formato microservices consideram a estratégia um sucesso. Lima (2022)

3.1 Quais as vantagens da arquitetura de microserviços

Como vimos até então, a arquitetura de microserviços é baseada em uma abordagem modular. Na prática isto significa desenvolver um sistema em pequenas partes, os chamados microsserviços, que geralmente são colocados dentro de containers. Cada container é um ambiente isolado e independente, contendo códigos, binários, bibliotecas e o que mais for necessário para que a aplicação seja executada. Isto permite, por exemplo,

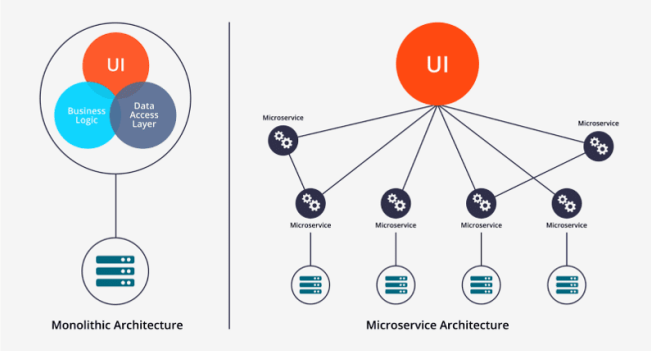

Figura 1 – Comparativo ilustrado entre uma arquitetura monolítica e uma arquitetura de microserviços.

que o desenvolvedor trabalhe de maneira mais fácil com diferentes tipos de linguagens de programação dentro de um mesmo sistema.

As Vantagens de trabalhar nesse modelo, são várias, mas aqui iremos alencar algumas delas: MICROSSERVIÇOS... (2022)

- Aceleração dos ciclos de desenvolvimento - Se cada componente é independente, é possível direcionar vários desenvolvedores para trabalhar simultaneamente em uma mesma aplicação. Dessa forma, os ciclos de desenvolvimento das aplicações são consideravelmente acelerados.

- Independência para escalabilidade - A autonomia dos microserviços se estende à sua escalabilidade, citada como uma das principais vantagens do formato por 92% das empresas que responderam a uma pesquisa da consultoria O'Reilly sobre o tema. Isso significa que, se a demanda por determinado serviço ou componente aumenta, é possível fazer implantações e alterações em servidores distintos para adequar a infraestrutura ao que se faça necessário. Desse modo, cada microsserviço é capaz de ser escalado conforme a maneira mais adequada — o que pode tornar a manutenção da infraestrutura ainda mais barata, por conta de sua escalabilidade horizontal.

- Facilidade para experimentação - Uma vez que os microservices possibilitam entregas constantes, fica mais fácil testar novas ideias e até mesmo descartar essas novas ideias em caso de falha sobre algum componente. Afinal, cada microsserviço funciona em seu próprio microuniverso independente.

- Abertura para a escolha de ferramentas e tecnologias - Os desenvolvedores têm total liberdade para escolher o melhor recurso ou ferramenta para solucionar um problema, o que garante à arquitetura de microsserviços uma abordagem plural, em nada generalista. Ou seja, o desenvolvedor tem maior abertura para a escolha de ferramentas e tecnologias que se façam mais indicadas para cada caso.

- Reutilização de aplicativos - Um software fragmentado em módulos reduzidos e bem definidos possibilita à equipe utilizar a mesma aplicação para finalidades distintas. Em outras palavras, um recurso criado para atender a uma função específica pode ser reutilizado como componente de outro recurso sem que haja a necessidade de escrever um novo código.

- Maior resistência à falhas - O microsserviço independente potencializa a resistência da aplicação a eventuais falhas. Diferentemente do formato monolítico, em que o erro de um componente pode gerar a falha de todo o sistema, os microsserviços são independentes e não demandam a interrupção de todo o aplicativo em caso de problema.

- Referência para domínios - Os microsserviços garantem uma referência clara e bem definida para domínios. Desse modo, se um componente fica responsável por seu usuário, esse componente também se tornará a única referência para alteração ou atualização de dados desse usuário. Assim, se um outro serviço necessita de uma informação de um usuário, esse dado deve ser alcançado a partir do microsserviço que abriga esse dado, não dependendo do sistema como um todo, mas sim apenas daquele módulo.

4 Microprofile - conceito e funcionamento

Durante anos, a plataforma do Java EE foi usada para o desenvolvimento de aplicações e microsserviços, e este fator continua em uso nos dias atuais. Porém, o uso do Java EE nativo no desenvolvimento destas aplicações apresenta algumas situações adversas, tais como uma complexidade relevante na lógica de programação, alto uso de recursos como mémoria e processamento, tempo de inicialização e etc. Estes são alguns fatores que não são bem vistos quando temos a intenção de criar aplicações ágeis baseadas em microserviços. Jansen (2022) Buscando melhorias que atendam a estes requisitos, surgiu o Microprofile.

Antes da criação do MicroProfile, a plataforma Java EE tornou-se estável e madura, resultando em lançamentos menos frequentes. Apesar da desaceleração da inovação Java EE, os serviços da Web continuaram a evoluir, levando à criação de novas tecnologias, como serviços da Web RESTful, JSON, HTTP/2 e arquitetura de microsserviços.

Com seu ciclo de lançamento mais lento, o Java EE não conseguiu acompanhar as mudanças do setor. No entanto, cientes das habilidades e investimentos que fornecedores e empresas colocam no Java EE, vários fornecedores tais como IBM e RedHat, apoiados pela comunidade Java Open Source ativa, decidiram se unir e criar o MicroProfile, com o intuito de padronizar, otimizar e criar definições no desenvolvimento de microserviços Java, criando-se uma tecnologia de código aberto que vem sendo amplamente adotada nos dias atuais. Em sua definição, o MicroProfile é um projeto que fornece uma coleção

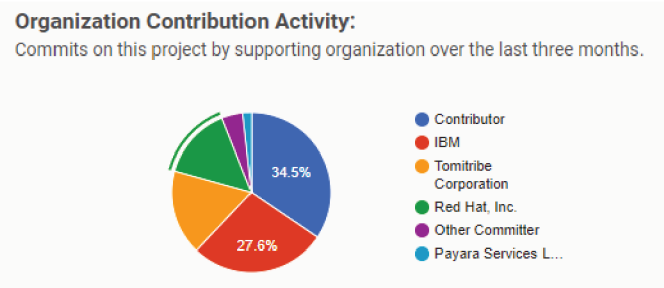

Figura 2 – Contribuição organizacional no código do Microprofile Who's... (2022)

de especificações projetadas para ajudar os desenvolvedores a criar microsserviços, tendo como objetivo definir coleções de API's para suportar o desenvolvimento e otimizar as aplicações e suas arquituras. Essas APIs foram originalmente adotadas do padrão Java EE (versões MicroProfile 3.x no Java EE 8) e foram estendidas por aquelas que são úteis para o padrão de microsserviços. Desde a versão do MicroProfile 4.0, o Java EE foi substituído pelo Jakarta EE (versão mais atual do Java EE).

A primeira versão do Microprofile apresentada em setembro 2016 trazia consigo algumas

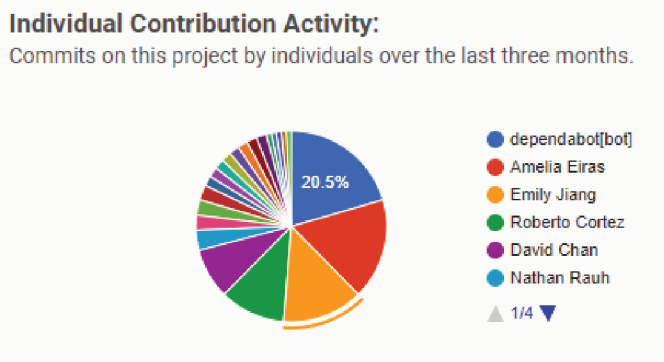

Figura 3 – Contribuição individual no código do Microprofile Who's... (2022)

funcionalidades (API's) básicas para o funcionamento de microserviços, são elas:

- CDI 1.2 - Responsável pela parte de injeção de dependências
- Json-P 1.0 - Responsável pelo processamento de pacotes Json
- Jax-RS 2.0 - Responsável pela disponibiliazação de serviços

Nos dias atuais, a tecnologia Microprofile se encontra na versão 5.0, trazendo treze especificações já consolidadas pelo mercado.

4.1 Implementações do Microprofile

4.1.1 Rastreamento distribuído

O tracing distribuído ou rastreamento distribuído, é uma técnica que permite rastrear o fluxo das requisições entre os serviços, e dentro de um ambiente de microsserviços isso é extremamente útil e necessário para que você consiga desenhar todo o caminho da requisição e com isso fazer diversos tipos de análises, resolução de problemas, observabilidade da aplicação e muito mais. Ferreira (2022) Essa implementação pode ser obtida pelo MicroProfile OpenTracing 3.0.

Open Tracing permite que os serviços participem facilmente em um ambiente de rastreamento distribuído e define procedimentos e uma API para acesso ao objeto OpenTracing-compliant Tracer dentro de um microserviço. Os logs gerados por este rastreamento podem ser consumidos por outras aplicações que trabalhem com rastreamento distribuído como Zipkin ou Jaeger (CHANG et al., 2019).

4.1.2 Interface agnóstica para APIs

A exposição de APIs é sem dúvida uma das coisas mais importantes no desenvolvimento de qualquer aplicação moderna. A especificação Open API (OAS) define uma

interface padrão agnóstica de linguagem de programação para desenvolvimento de APIs, podendo ser definida como um conjunto de interfaces e modelos para construção de contratos a partir de um serviço RESTful. Não implementa algo técnico, apenas definições e melhores práticas de uso.

A adoção da OpenAPI alivia a carga dos criadores de API que tentam educar seus usuários com eficiência. Porém, é mais eficaz quando permite que os desenvolvedores não apenas aprendam melhor, mas aprendam menos. É tão difícil ter a atenção dos desenvolvedores que, quando a temos, queremos educá-los sobre tópicos importantes, em vez de detalhes logísticos automatizáveis. Leventhal (2020) O MicroProfile OpenAPI 3.0 contempla essa interface.

4.1.3 Utilização de serviços RESTful

O MicroProfile Rest Client fornece uma abordagem segura para utilização dos serviços RESTful utilizando o protocolo HTTP, garantindo consistência e reusabilidade.

O MicroProfile Rest Client gera automaticamente uma instância do cliente com base no que é definido e anotado na interface do modelo. Assim, você não precisa se preocupar com todo o código padrão, como configurar uma classe de cliente, conectar-se ao servidor remoto ou invocar o URI correto com os parâmetros corretos. Consuming... (2022)

4.1.4 Configuração de ambiente

Na maioria das aplicações corporativas atuias, se faz necessário que as configurações sejam armazenadas e aplicadas de acordo com um ambiente específico (produção, homologação, desenvolvimento e etc). A especificação Microprofile Config permite essa flexibilidade e a proteção dos dados de configuração além de nos permitir injetar propriedades de configuração estáticas e dinâmicas para microsserviço. MicroProfile... (2022)

4.1.5 Tolerância a falhas

Um dos desafios trazidos pela natureza distribuída dos microsserviços é que a comunicação com sistemas externos é inerentemente não confiável. Isso aumenta a demanda por resiliência de aplicativos. SMALLRYE... () Toda aplicação produtiva deve estabelecar um nível de resiliência, e uma das maneiras de garantir essa funcionalidade é com a utilização de uma estratégia de tolerância a falhas. A especificação Fault Tolerance basicamente consiste em levantar alguma estratégia para orientar o fluxo de execução do serviço em caso de queda por exemplo políticas de retry, circuit e breakers.

4.1.6 Observabilidade e análise de métricas

A especificação Metrics permite a coleta de dados sobre os serviços implementados, permitindo avaliar diversas coisas, como por exemplo volume de tráfego das APIs, saúde e quantidade de erros. A funcionalidade em questão provê uma maneira unificada para os servidores de MicroProfile exportarem e também analisar dados de monitoramento.

Você pode monitorar métricas para determinar o desempenho e a integridade de um serviço. Também pode usá-los para identificar problemas, coletar dados para planejamento de capacidade ou decidir quando dimensionar um serviço para ser executado com mais ou menos recursos.Providing... ()

4.1.7 Segurança de comunicação

Usualmente, na construção de serviços RESTful, é comum encontrarmos padrões OpenIS Connect (OICD), OAuth2 ou JSON Web Token(JWT) sendo utilizados como espeficação de segurança. Um JSON Web Token (JWT) é um token autocontido projetado para transmitir informações com segurança como um objeto JSON. As informações neste objeto JSON são assinadas digitalmente e podem ser confiáveis e verificadas pelo destinatário. Para microsserviços, um mecanismo de autenticação baseado em token oferece uma maneira leve de controles de segurança e tokens de segurança para propagar identidades de usuários em diferentes serviços. O JSON Web Token está se tornando o formato de token mais comum porque segue padrões bem definidos e conhecidos. Securing... ()

O Microprofile JWT Propagation descreve como os tokens JWT assinados e emitidos pelo OICD e outros fornecedores confiáveis podem ser verificados e suas reivindicações usadas para controle de acesso do serviço.

4.1.8 Verificação de integridade do serviço

A funcionalidade de Health promove a verificação de integridade da aplicação, e são de extrema importância para garantir a resiliência do serviço, analisando a saúde do ambiente e verificando se dentro do cluster existe alguma unidade de serviço ainda ativa e em funcionamento, ou um nó prejudicado e assim poder substituir por outro que esteja com a saúde melhor.

O MicroProfile Health permite que os serviços relatem sua integridade e publica o status geral da integridade para um endpoint definido. Um serviço informa UP se estiver disponível e informa DOWN se não estiver disponível. Um serviço verifica sua própria integridade realizando as autoverificações necessárias e, em seguida, relata seu status geral implementando a API fornecida pelo MicroProfile Health. Uma autoverificação pode ser uma verificação de qualquer coisa que o serviço precise, como uma dependência, uma

conexão bem-sucedida com um terminal, uma propriedade do sistema, uma conexão com o banco de dados ou a disponibilidade dos recursos necessários.Adding... ()

4.1.9 Controle de dependências e ciclo de vida da aplicação

Contexts and Dependency Injection (CDI) define um rico conjunto de serviços complementares que melhoram a estrutura do aplicativo. Os serviços mais fundamentais fornecidos pelo CDI são contextos que vinculam o ciclo de vida de componentes com estado a contextos bem definidos e injeção de dependência, que é a capacidade de injetar componentes em um aplicativo de maneira segura. Com o CDI, o contêiner faz todo o trabalho assustador de instanciar dependências e controlar exatamente quando e como esses componentes são instanciados e destruídos. Injecting... ()

O CDI é sem dúvidas uma das especificações mais importantes. Basicamente nos permite gerenciar o ciclo de vida dos componentes com estado através de contextos de ciclo de vida do domínio e também permite injetar componentes nos objetos cliente de maneira segura.

4.1.10 Formato de comunicação

Quando falamos em API's RESTful, logo pensamos em JSON. JSON é uma maneira bem comum de se transmitir dados entre RESTful APIs, e a especificação JSON-P é a especificação responsável pelo processamento desse conteúdo. Além do JSON-P, temos também o JSON-B. JSON-B (Java API for JSON Binding) é a especificação responsável pela conversão de objetos Java para JSON e vice-versa, de extrema importncia para a transmissão de dados e complementa as funcionalidades citadas anteriormente JAX-RS e JSON-P. JSON-P... ()

4.1.11 Especificação para criação de serviços

A fuincionalidade JAX-RS (Java API for RESTful Web Services) é a especificação para serviços REST e é de suma importância para qualquer aplicação web e até mesmo microsserviços, pois ela provê uma série de APIs para a construção desses serviços e recursos REST. Phill ()

4.2 Caso de uso

Como descrito em capítulos anteriores, a construção de um sistema utilizando a tecnologia de microprofiles exige a preparação de um ambiente para execução. O ambiente de execução consiste no runtime, na IDE de escolha, na JVM e versão de microprofiles. De acordo com os estudos para realização deste projeto, escolhemos trabalhar com a versão

3.2, com o servidor de aplicação Quarkus, Java 11 e VSCode como IDE.
Desenvolvimento moderno necessita de desburocratização da codificação e agilidade no feedback constante ao programador, todas características essenciais em uma cultura devops e entrega contínua. (FARLEY, 2010) Desde a popularização do desenvolvimento com contêineres, entre 2014 e 2016, os sistemas veem se tornando mais críticos e ferramentas foram criadas para se adaptar a demanda constante de mudanças. Java EE foi perdendo popularidade, dando espaço para outras linguagens como Python, nodejs, etc. (TIOBE, 2021)
Cada vez mais sistemas foram utilizando os conceitos de servidores de aplicação, com pilha única de serviços rodando dentro de um ponto único de falha. Um mudança pequena era necessário parar, mesmo que por alguns milésimos de segundos, todos os outros serviços na mesma pilha. Foram criados vários artifícios, técnicas, arquiteturas para contornar o problema, mesmo período que java estava no topo de sua popularidade dos últimos 10 anos. (TIOBE, 2021)
O padrão de Micro-serviços se tornou necessário e uma linguagem tão popular como java precisava de uma especificação que atendesse. O ambiente agora precisa ser independente e os serviços também. Quarkus é um runtime de micro-serviços para java. O seu objetivo é adaptar com ambiente java EE para a realidade necessária para desenvolvimento massivo de micro-serviços em um cenário de entrega contínua. Acelerando a velocidade de um deploy, modernizando a máquina virtual para trabalhar em containers kubernetes e implementa as especificação de microprolies estabelecidas. (CLINGAN J., 2022)
Aqui demonstrarei como iniciar 2 aplicações quarkus comunicando entre si, no modo de desenvolvimento do Quarkus, onde o código pode ser alterado e as alterações são refletidas instantemente na execução. Primeiramente vamos realizar a instalação do Quarkus. O guia para inicio de utilização Starter... (2022b) direciona a instalação para os seguintes passos:

4.2.1 Download e Instalação do Quarkus

Faremos a instalação do Quarkus através do comando abaixo:

```
curl -Ls https://sh.jbang.dev | bash
-s - app install --fresh --force quarkus@quarkusio
```

Para máquinas windows, utilize o comando abaixo no powershell:

```
iex "& { $(iwr https://ps.jbang.dev) }
app install --fresh --force quarkus@quarkusio"
```

Com isto foi baixado e instalado o Quarkus na máquina que você executou o comando. Para o próximo o passo vamos testar a instalação, criando uma aplicação de

testes e executando-a. Execute o comando em uma sessão do terminal nova e diferente da sessão anterior:

```
quarkus create && cd code-with-quarkus.
```

A execução deve ter um comportamento similar ao das imagems abaixo:

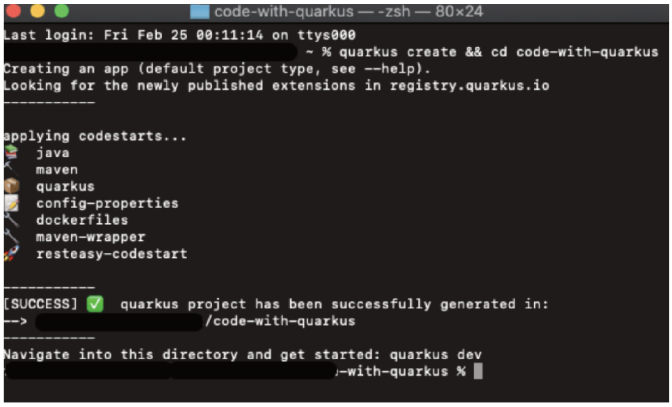

Figura 4 – Instalando Quarkus

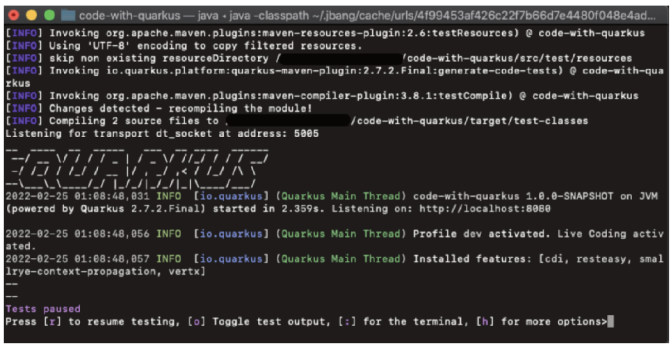

Figura 5 – Executando Quarkus

Com os passos acima, criamos uma aplicação teste no Quarkus. Os arquivos de código-fontes foram criados dentro da pasta de usuário que você rodou o comando de iniciação. Agora vá até o seu browser e digite localhost:8080. Se a imagem abaixo aparecer estamos prontos para o próximo passo, criar uma aplicação.

4.2.2 Aplicação teste de comunicação de micro-serviços

O código foi gerado pela aplicação Microprofiler Starter Starter... (2022a), e iremos passar aqui para melhor entendimento. O pacote principal criado foi br.ufu.tcc.mptcc, aqui criaremos a aplicação restful com o código abaixo:

```
package br.ufu.tcc.mptcc;
import javax.ws.rs.ApplicationPath;
import javax.ws.rs.core.Application;
```

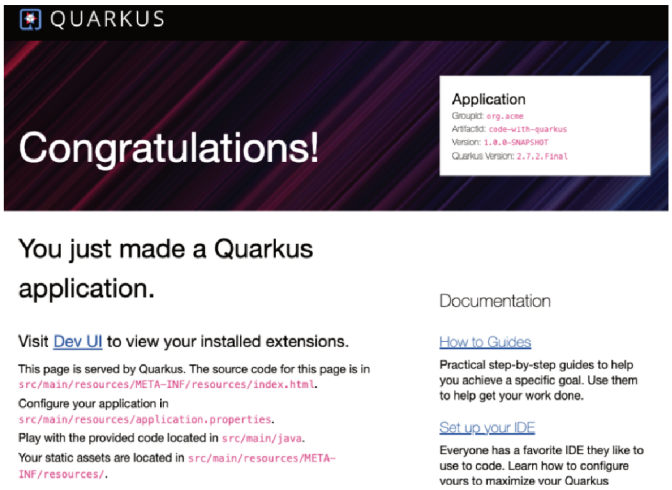

Figura 6 – Tela inicial Quarkus

```
4
5  @ApplicationPath("/data")
6  public class MptccRestApplication extends Application {
7  }
```

Listing 4.1 – Código fonte em Java

E agora vamos criar o primeiro serviço REST simples e local, ou seja, no mesmo container ou pod:

```
1  package br.ufu.tcc.mptcc;
2  import javax.inject.Singleton;
3  import javax.ws.rs.GET;
4  import javax.ws.rs.Path;
5
6  @Path("/hello")
7  @Singleton
8  public class HelloController {
9
10     @GET
11     public String sayHello() {
12         return "Hello World";
13     }
14 }
```

Listing 4.2 – Código fonte em Java

Perceba aqui que criamos um serviço rest padrão da api RESTful Java, ou seja, ainda não estamos utilizando microprofile. No próximo passo, já vamos utilizar o microprofile e tirar proveito da capacidade e herança do JAVA EE porém otimizada para os padrões de

micro-serviços, como a ingestão de dependências através de interfaces. Utilizando o RegisterRestClient. Vamos criar também um client de um micro-serviço que estará rodando em outro ambiente, vamos criar o pacote br.ufu.tcc.mptcc.client. Dentro do pacote criaremos uma interface e uma classe. A interface abaixo está usando a referência microprofile para registrar um cliente de micro-serviços onde passará um parâmetro em string.

```
1  package br.ufu.tcc.mptcc.client;
2  import org.eclipse.microprofile.rest.client.inject.
       RegisterRestClient;
3
4  import javax.enterprise.context.ApplicationScoped;
5  import javax.ws.rs.GET;
6  import javax.ws.rs.Path;
7  import javax.ws.rs.PathParam;
8
9  @RegisterRestClient
10 @ApplicationScoped
11 public interface Service {
12
13     @GET
14     @Path("/{parameter}")
15     String doSomething(@PathParam("parameter") String parameter);
16 }
```

Listing 4.3 – Código fonte em Java

Agora vamos criar a classe que será o cliente do serviço-b, através de uma ingestão do serviço dentro da classe cliente, também parte da especificação microprofiles.

```
1  package br.ufu.tcc.mptcc.client;
2  import org.eclipse.microprofile.rest.client.inject.RestClient;
3
4  import javax.enterprise.context.ApplicationScoped;
5  import javax.inject.Inject;
6  import javax.ws.rs.GET;
7  import javax.ws.rs.Path;
8  import javax.ws.rs.PathParam;
9
10 @Path("/client")
11 @ApplicationScoped
12 public class ClientController {
13
14     @Inject
15     @RestClient
```

```
16      // https://quarkus.io/guides/cdi-reference#private-members
17      // - private Service service;
18      Service service;
19
20      @GET
21      @Path("/test/{parameter}")
22      public String onClientSide(@PathParam("parameter") String
           parameter) {
23          return service.doSomething(parameter);
24      }
25  }
```

Listing 4.4 – Código fonte em Java

Perceba aqui, na classe de controle, a utilização da anotação RestClient para a ingestão do cliente registrado anteriormente. Importante configurar o arquivo properties para poder levantar a aplicação dentro do Quarkus, o arquivo abaixo foi criado (Application.properties).

```
1  injected.value=Injected value
2  value=lookup value
3  quarkus.package.output-name=mptcc
4  br.ufu.tcc.mptcc.client.Service/mp-rest/url=http://localhost
      :8180/data/client/service
5  quarkus.ssl.native=true
6  quarkus.smallrye-jwt.enabled=false
7  quarkus.jaeger.service-name=Demo-Service-A
8  quarkus.jaeger.sampler-type=const
9  quarkus.jaeger.sampler-param=1
10 quarkus.jaeger.endpoint=http://localhost:14268/api/traces
11 quarkus.native.additional-build-args=-H:Log=registerR
```

Listing 4.5 – Código fonte em Java

Para facilitar os testes foi criado um arquivo index.html dentro da pasta resources com os links para os 2 serviços que criamos:

```
1  <a href="data/hello" target="_blank" >Hello JAX-RS endpoint</a> <
      br/>
2
3  <h3>Rest Client</h3>
4  <a href="data/client/test/valordeteste" target="_blank" >Chama o
       REST endpoint usando um cliente baseado em interface</a> <br/>
```

Listing 4.6 – Código fonte em Java

Importante notar que a partir daqui uma instalação do maven é necessário e a adição dos binários dentro do path do terminal utilizado para rodar. Em um terminal vá na raíz do projeto e digite o comando:

```
1   quarkus dev.
```

Listing 4.7 – Código fonte em Java

Isto levantará a aplicação, que pode ser testada através do endereço localhost:8080. Agora vamos trabalhar na aplicação service-b. Primeiro vamos criar a aplicação, semelhante ao código do service-a, conforme abaixo:

```
1  package br.ufu.tcc.mptcc;
2
3  import org.eclipse.microprofile.auth.LoginConfig;
4
5  import javax.annotation.security.DeclareRoles;
6
7  import javax.ws.rs.ApplicationPath;
8  import javax.ws.rs.core.Application;
9  @ApplicationPath("/data")
10
11 @LoginConfig(authMethod = "MP-JWT", realmName = "jwt-jaspi")
12 @DeclareRoles({"protected"})
13
14 public class MptccRestApplication extends Application {
15 }
```

Listing 4.8 – Código fonte em Java

Neste código é importante notar que a comunicação entre 2 micro-serviços está utilizando a autenticação MP-JWT da especificação microprofiles, e aqui também estamos declarando que o grupo de autenticação “protected"terá acesso ao serviço. Trabalhos futuros encarregarão de explorar o Security framework do microprofiles. No pacote client, criaremos a classe que implementará o serviço que foi chamado no service-a.

```
1  package br.ufu.tcc.mptcc.client;
2
3  import javax.ws.rs.GET;
4  import javax.ws.rs.Path;
5  import javax.ws.rs.PathParam;
6
7  @Path("/client/service")
8  public class ServiceController {
9
```

```
10      @GET
11      @Path("/{parameter}")
12      public String doSomething(@PathParam("parameter") String
           parameter) {
13          return String.format(" Coisa de Parametro passado '%s'
              olha s  que legal", parameter);
14      }
15  }
```

Listing 4.9 – Código fonte em Java

Agora vamos também estabelecer o arquivo Application.properties, com um detalhe, vamos especificar a porta 8180 para subir e não conflitar com o service-a.

```
1  quarkus.ssl.native=true
2  quarkus.package.output-name=mptcc
3  quarkus.http.port=8180
4  mp.jwt.verify.publickey.location=META-INF/resources/publicKey.pem
5  mp.jwt.verify.issuer=https://server.example.com
6  quarkus.smallrye-jwt.enabled=true
7  quarkus.jaeger.service-name=Demo-Service-B
8  quarkus.jaeger.sampler-type=const
9  quarkus.jaeger.sampler-param=1
10 quarkus.jaeger.endpoint=http://localhost:14268/api/traces
```

Listing 4.10 – Código fonte em Java

Vamos rodar com o mesmo processo que o anterior, somente entrando na pasta do código, realizando um "maven install"e depois o comando "quarkus dev". Com isto temos agora 2 pods de micro-serviços rodando e comunicando entre si.

A utilização do micropofile neste exemplo foi para demonstrar a capacidade, ou até mesmo a simplicidade, da criação de um serviço rest, com interfaces e a sua implementação através de ingestão em containers separados. Isto foi devido a utilização do pacote LoginConfig, RestClient e RegisterRestClient. Além da própria especificação microprofile da OpenAPI (como javax.ws.rs).

O pacote de autenticação utilizado aqui (LoginConfig), pode ser utilizado para gerar diferentes grupos e níveis de autenticação e controle de acesso aos serviços que criamos. No exemplo descrito aqui neste trabalho vimos que o papel "protected"foi definido como nível de acesso principal. Aqui segue como a documentação do microprofile define os pacotes que utilizamos: Microprofile... (2022)

Login config: Anotação de segurança descrevendo o método de autenticação e o realm de segurança utilizado nesta aplicação.

Rest Cliente: Anotação para qualificar um ponto de injeção e indicar que será uma instância do tipo Tupe-Safe Rest Cliente. Causando o ponto de injeção ser satisfeito pelo cliente Rest runtime do microprofile. Rest... (2022)

Register Rest cliente: Aqui é uma anotação para registrar o cliente no runtime. Este método é para quando a visão de arquitetura é centralizada em serviços. Portanto a invocação dos métodos na interface é como acontece se estivessem rodando na mesma VM, porém é um serviço remoto. Register... (2022)

5 Conclusão

Java ainda é, em números, a linguagem de programação mais popular no ramo de tecnologia. Entretanto, em 2016 protagonizou uma expressiva queda de 32% nos resultados do Google Scholar, ou 14.000 resultados a menos que o primeiro ano da pesquisa (2012). Dentre as prováveis razões desta queda podemos destacar o alto número de vulnerabilidades encontradas no JRE (Java Runtime Environment): 819 desde 2010, de acordo com o banco de dados do CVE - Common Vulnerabilities and Exposures (CVE, 2017).ANALISANDO... (2022)

Além de fatores de segurança, outros fatores como a complexidade computacional e a ascensão de aplicações Cloud Native (aplicações que já nascem preparadas para trabalharem em um ambiente de Cloud), fizeram com que a comunidade de tecnologia buscasse novos métodos e linguagens que proporcionassem um desenvolvimento facilitado, seguro, rápido e com foco em computação de nuvem, utilizando de ferramentas com o Kubernetes e Docker para suportar tais aplicações.

Visando esse cenário em ascensão, o Microprofile ganhou bastante popularidade. O maior facilitador da incorporação do Microprofile no mercado com certeza foi o fato de utilizar da linguagem Java para criar serviços de cloud native. Como o Java já possui um nome de peso no merdado, além de uma grande familiaridade dos desenvolvedores com a linguagem, foi fácil fazer com o microprofile fosse aceito no meio de tecnologia.

Assim como apontado neste trabalho, é muito fácil criar uma aplicação utilizando microprofile, e acima de tudo, podendo explorar esse mundo utilizando-se da linguagem Java. Com as implementações do microprofile, como o Quarkus, Payara, WildFly entre outros, criar microserviços tornou-se algo fácil, rápido e seguro, entregando várias melhorias que antes eram difíceis de serem exploradas utilizando Java nativo. Criar esteiras de Continuous Deployment e Continuous Delivery com o microprofile pode ser muito mais vantajoso, bem como fazer a gestão destas aplicações.

Para evolução do que foi discutido aqui, tendo em vista uma tendência atual do mercado em migrar para aplicações cloud native e uma arquitetura de microserviços, podemos sugerir um estudo aprofundado nas aplicações do microprofile, construção de grandes arquitetura de microserviços bem como um aprofundamento no estudo da gestão de microserviços, tema este que por sua vez é de suma importância na área de TI.

Referências

ADDING health reports to microservices. Disponível em: <https://openliberty.io/guides/microprofile-health.html>. Acesso em: 09 mar. 2022. Citado na página 21.

ANALISANDO O USO DE LINGUAGENS DE PROGRAMAÇÃO EM ÂMBITO ACADÊMICO ENTRE PÁGINAS EM PORTUGUÊS E INGLÊS DE 2012 A 2016: ASCENSÃO DO PYTHON E A QUEDA DO JAVA. 2022. Disponível em: <http://www.jornacitec.fatecbt.edu.br/index.php/VIIJTC/VIIJTC/paper/viewFile/1694/2148>. Acesso em: 09 mar. 2022. Citado na página 30.

BEERNINK, J.; TIJMS, A. History of cdi. In: *Pro CDI 2 in Java EE 8*. [S.l.]: Springer, 2019. p. 1–36. Citado na página 9.

CAI, E. et al. Building microservices in a cloud-native world using eclipse microprofile and open liberty. In: *Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering*. [S.l.: s.n.], 2018. p. 350–353. Citado na página 10.

CHANG, Y.-K. et al. Hands-on workshop on fast, efficient & seriously open cloud-native java. In: *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*. [S.l.: s.n.], 2019. p. 373–375. Citado na página 18.

CLINGAN J., F. K. Kubernetes native microservices with quarkus and microprofile. In: . [S.l.]: 2022, 2022. Citado na página 22.

CONSUMING RESTful services with template interfaces. 2022. Disponível em: <https://openliberty.io/guides/microprofile-rest-client.html>. Acesso em: 09 mar. 2022. Citado na página 19.

FARLEY, D. History of cdi. In: *Continuous delivery: reliable software releases through build, test, and deployment automation.* [S.l.]: 2010, 2010. Citado na página 22.

FERREIRA, M. *Eclipse Microprofile, um caminho para construção de microsserviços com Java*. 2022. 27 mar. 2020. Disponível em: <https://codificante.com.br/eclipse-microprofile-um-caminho-para-construcao-de-microsservicos-com-java>. Acesso em: 15 fev. 2022. Citado na página 18.

GONCALVES, A. Context and dependency injection. In: *Beginning Java EE 7*. [S.l.]: Springer, 2013. p. 23–66. Citado na página 9.

INJECTING dependencies into microservices. Disponível em: <https://openliberty.io/guides/cdi-intro.html>. Acesso em: 09 mar. 2022. Citado na página 21.

JANSEN, G. *What is MicroProfile?* 2022. 06 July, 2021. Disponível em: <https://developer.ibm.com/series/what-is-microprofile/>. Acesso em: 15 fev. 2022. Citado na página 17.

JAVA EE: entenda a plataforma e principais funcionalidades. 2022. Disponível em: <https://www.zup.com.br/blog/java-ee-principais-funcionalidades>. Acesso em: 15 fev. 2022. Citado na página 12.

JSON-P and JSON-B. Disponível em: <https://www.openliberty.io/docs/21.0.0.3/json-p-b.html>. Acesso em: 09 mar. 2022. Citado na página 21.

LEVENTHAL, A. *Usando a OpenAPI para criar APIs inteligentes que ajudam os desenvolvedores*. 2020. Disponível em: <https://www.infoq.com/br/articles/openapi/>. Acesso em: 09 mar. 2022. Citado na página 19.

LIMA, V. *Microservices: O que são, e como funcionam?* 2022. 17 September, 2018. Disponível em: <https://blog.schoolofnet.com/o-que-sao-microservices/>. Acesso em: 15 fev. 2022. Citado na página 14.

MICROPROFILE Config API. 2022. Disponível em: <https://www.ibm.com/docs/en/was-liberty/base?topic=api-microprofile-config>. Acesso em: 09 mar. 2022. Citado na página 19.

MICROPROFILE Documentation. 2022. Disponível em: <https://download.eclipse.org/microprofile/microprofile-jwt-auth-1.1/apidocs/>. Acesso em: 09 mar. 2022. Citado na página 28.

MICROSSERVIÇOS: QUAIS SÃO AS VANTAGENS DE OPTAR PELA ARQUITETURA? 2022. 22 September, 2019. Disponível em: <https://www.squadra.com.br/blog/microsservicos/>. Acesso em: 15 fev. 2022. Citado na página 15.

OBTENHA Informações sobre a Tecnologia Java. 2022. Disponível em: <https://www.java.com/pt-BR/about/#:~:text=Com%20mais%20de%209%20milh%C3%B5es,use%20aplica%C3%A7%C3%B5es%20e%20servi%C3%A7os%20estimulantes.> Acesso em: 15 fev. 2022. Citado na página 11.

PHILL. *JAX-RS Tutorial*. Disponível em: <https://rieckpil.de/whatis-jakarta-restful-web-services-jax-rs/>. Acesso em: 09 mar. 2022. Citado na página 21.

PROVIDING metrics from a microservice. Disponível em: <https://openliberty.io/guides/microprofile-metrics.html>. Acesso em: 09 mar. 2022. Citado na página 20.

REGISTER Microprofile. 2022. Disponível em: <https://download.eclipse.org/microprofile/microprofile-rest-client-1.3/apidocs/>. Acesso em: 09 mar. 2022. Citado na página 29.

REST Microprofile. 2022. Disponível em: <https://download.eclipse.org/microprofile/microprofile-rest-client-1.3/apidocs>. Acesso em: 09 mar. 2022. Citado na página 29.

SECURING microservices with JSON Web Tokens. Disponível em: <https://openliberty.io/guides/microprofile-jwt.html>. Acesso em: 09 mar. 2022. Citado na página 20.

SMALLRYE FAULT TOLERANCE. Disponível em: <https://quarkus.io/guides/smallrye-fault-tolerance>. Acesso em: 09 mar. 2022. Citado na página 19.

STARTER Microprofile. 2022. Disponível em: <https://start.microprofile.io/>. Acesso em: 05 mar. 2022. Citado na página 23.

STARTER Quarkus. 2022. Disponível em: <https://quarkus.io/get-started/>. Acesso em: 09 mar. 2022. Citado na página 22.

TIOBE. T. i. o. b. e. In: . [S.l.]: 2021, 2021. Citado na página 22.

WETHERBEE, J. et al. *Beginning EJB in Java EE 8: Building Applications with Enterprise JavaBeans*. [S.l.]: Springer, 2018. Citado na página 10.

WHAT are microservices? 2022. 13 April, 2018. Disponível em: <https://www.redhat.com/en/topics/microservices/what-are-microservices>. Acesso em: 15 fev. 2022. Citado na página 14.

WHO'S Involved. 2022. Disponível em: <https://projects.eclipse.org/projects/technology.microprofile/who>. Acesso em: 15 fev. 2022. Citado 2 vezes nas páginas 17 e 18.