Tribunal Regional do Trabalho da 4ª Região

Guia de Arquitetura de Software

1ª Edição — 20/07/2018

Apresentação

Este Documento visa apresentar uma série de recomendações a serem aplicadas na construção, ou adaptação, de artefatos de software construídos no Tribunal Regional do Trabalho da 4a Região.

Essas recomendações servem como balizadores práticos para a atividade de desenvolvimento de software, de modo a alinhar a instituição em relação às melhores práticas do mercado, levando em consideração tanto o estado da arte quanto a realidade tecnológica e cultural vivida no presente. Nesse sentido, buscamos equilibrar a necessidade de criar uma arquitetura que tenha foco no domínio da aplicação, minimizando dependências de detalhes tecnológicos, e a necessidade, em contrapartida, de minimizar a diversidade tecnológica [1].

A criação de uma arquitetura que tenha foco no domínio da aplicação, ou seja, no código que entrega mais valor aos usuários, parte da descrição dos processos de negócio, de forma a produzir uma representação de suas regras subjacentes, facilitando sua compreensão, manutenção e reutilização, daí sua importância central.

Para a produção desse software, entretanto, temos hoje à disposição um número cada vez maior de alternativas em termos de tecnologias. Essa diversidade tem um custo, porque o treinamento, aprimoramento e constante atualização do pessoal envolvido no desenvolvimento de software aumenta com o aumento do número de tecnologias utilizadas, especialmente no cenário atual, marcado pela transformação tecnológica cada vez mais rápida. Dessa forma, é necessário buscar constantemente limitar o conjunto de tecnologias capaz de satisfazer as necessidades de desenvolvimento de software da instituição.

Com base nessas duas premissas, o documento apresenta diversos princípios aplicáveis de forma geral e ainda recomendações de níveis de utilização, que tem por objetivo permitir aos desenvolvedores avaliarem a sua aplicação em cada caso. Por último, o documento recomenda, onde cabível, a adoção de certas tecnologias, como forma de homogeneizar o desenvolvimento de software, com vistas à redução de seu custo.

Objetivos

- Definir princípios gerais para guiar a escolha da arquitetura de aplicativos corporativos web ou desktop de propósito específico (com alto desempenho, escalabilidade, robustez, gestão central, hospedadas em servidores, e grande número de usuários em rede).
- Definir recomendações para o projeto/implementação de funcionalidades frequentes nos sistemas de informação desenvolvidos e/ou mantidos pelo TRT4
- Propor níveis de aplicação dos princípios e recomendações
- Propor o uso de tecnologias específicas, quando for o caso

Contexto

O objetivo da escolha de uma arquitetura é conferir à aplicação certas características que resultem em software de qualidade. Em vista disso apontamos características desejáveis e aspectos a serem evitados na construção de um software. Além disso, apresentamos uma comparação entre arquiteturas populares em relação a algumas dessas características.

O universo ao qual o documento se aplica inclui aplicativos de gestão de processo de negócio, gestão de conteúdo, recursos humanos, contabilidade, gestão de cobranças e pagamentos, gestão de patrimônio, integração de aplicações corporativas e automação de formulários. Não fazem parte desse universo aplicações que, apesar de utilizadas no tribunal, não são desenvolvidas por ele, tais como relacionamento com cliente, BI, gestão de TI, backup, gestão de projetos, gestão de ciclo de vida de produto, manufatura, e saúde e segurança ocupacional, bem como software de infraestrutura corporativa (ex.: bases de dados), de propósito geral (ex.: editores de texto), de desenvolvimento (ex.: IDEs), de plataforma (ex.: ferramentas de administrador de sistema) ou embarcado (ex.: firmware)

Características desejáveis

- Desempenho, escalabilidade e robustez
- Modularidade (de código, objeto, mensages ou serviço)

- Manutenibilidade, reusabilidade e baixo acoplamento, obtidos por modularidade
- Extensibilidade e adaptabilidade a mudanças de requisitos, para desenvolvimento ágil e iterativo
- Separação de responsabilidades em lugares bem definidos
- Desenvolvimento paralelizável
- Baixa complexidade
- Testabilidade

A tabela 1 apresenta uma comparação entre arquiteturas populares em relação a algumas das características desejáveis.

Tabela 1. Comparação de Arquiteturas

Arquitetura[1]	Monolítica	SOA[2]	MSA[3]	MVC[4]	HMVC[5]	MVA[6]	MVP[7]
Complexidade	Variável	Alta	Alta	Média	Média	Média	Média
Modularidade	Baixa	Alta	Alta	Média	Alta	Alta	Alta
Extensibilidade	Baixa	Variável	Alta	Média	Alta	Alta	Alta
Adaptabilidade	Baixa	Variável	Alta	Média	Alta	Alta	Alta
Testabilidade	Baixa	Média	Média	Média	Média	Média	Alta
Desempenho	Variável	Variável	Baixa	Variável	Alta	Variável	Variável
Escalabilidade	Baixa	Variável	Alta	Variável	Variável	Variável	Variável
Robustez	Baixa	Variável	Alta	Variável	Variável	Variável	Variável
Separação de funcionalidades	Baixa	Alta	Alta	Média	Alta	Média	Média

Impactos negativos

Crise de software: dificuldade de escrever programas úteis, corretos e eficientes gastando pouco tempo/dinheiro. Resulta da falta de gestão do crescimento rápido da complexidade do programa.

[1] A descrição das arquiteturas citadas encontra-se nos anexos deste documento
[2] *Service-Oriented Architecture*
[3] *Microservice Architecture*
[4] *Model-View-Controller pattern*
[5] *Hierarchical MVC pattern*
[6] *Model-View-Adapter pattern*
[7] *Model-View-Presenter pattern*

Dívida técnica: custo de retrabalho por escolher inicialmente uma solução fácil ao invés de uma abordagem melhor que custaria mais tempo. Não é ruim em prova de conceito, mas tende a se acumular. Pode ser causada por: indefinição inicial, pressões corporativas, mudanças de especificação no último instante, falta de processo ou de compreensão, falta de conhecimento, componentes altamente acoplados, falta de teste, falta de documentação, falta de colaboração, desenvolvimento paralelo, refatoração tardia, falta de alinhamento aos padrões, ausência de autonomia sobre código externo, ou falta de liderança técnica.

Erosão de arquitetura: divergência entre a arquitetura planejada e a implementada, constituindo uma dívida técnica. Resulta de não cumprir o plano ou de violar os princípios da arquitetura. Pode ser detectada usando modelos de reflexão (comparação entre o modelo de alto nível do arquiteto com a implementação) e linguagens de domínio específico (para verificação de restrições arquiteturais). Pode ser corrigida aplicando conformação de arquitetura orientada a processo, gestão de evolução de arquiteturas, cumprimento de projeto de arquitetura, vinculação de arquitetura à implementação, auto-adaptação, e restauração: recuperação, descoberta e conciliação.

Paralisia de Análise: Análise exagerada de uma situação, de forma que uma decisão ou ação nunca chega a ser tomada. É causada pelo planejamento inicial total (BDUF) baseado em conhecimento incompleto do usuário em relação ao problema apresentado. Pode ser evitada pelo emprego de desenvolvimento iterativo e modular.

Proposta

Dadas as considerações feitas até aqui, este documento apresenta uma proposta de arquitetura para as aplicações desenvolvidas ou mantidas pelo TRT4. A proposta aqui apresentada é dividida em duas partes. De um lado, define-se a forma pela qual devem ser organizadas/dispostas as funcionalidades da aplicação a partir da definição dos processo de negócio.

A implementação dessas funcionalidades centrais do negócio depende de um estrutura tecnológica que tem elementos comuns (que tendem a se repetir na implementação de várias aplicações/funcionalidades). Em vista disso, listamos alguns aspectos recorrentes e sugestões sobre como devem ser tratados. Para cada aspecto apresentado, são indicados o objetivo a ser alcançado, os diferentes níveis de aderência às sugestões e, quando for o caso, indicações de tecnologias a serem utilizadas.

Arquitetura

Descrição

Este documento propõe o uso de uma arquitetura baseada em serviços para o desenvolvimento de aplicações. Nesse contexto, um serviço é uma funcionalidade (ou conjunto de funcionalidades), oferecida na rede para ser acessada por diferentes clientes, utilizando um protocolo de comunicação estabelecido. Cada serviço deve prover uma interface bem definida, além de ter políticas próprias de controle de acesso.

Assim, sob o ponto de vista dessa arquitetura, as aplicações constituem-se de conjuntos de serviços disponibilizados e acessados através da rede, organizados de forma coerente para atingir os objetivos da aplicação. Tais serviços podem ser criados especificamente para a aplicação ou serem utilizados de catálogo de serviços já implementados.

Os serviços devem apresentar:

1. **Granularidade adequada**: cada serviço deve ter tamanho e escopo adequados, representando uma funcionalidade do negócio relevante para o usuário;

2. **Abstração**: serviços devem oferecidos como caixas-pretas, com a lógica interna escondida de quem consome o serviço;

3. **Autonomia**: serviços devem ser independentes e auto-contidos, controlando toda a funcionalidade que encapsulam; Os serviços devem ser projetados para

serem desacoplados, apesar de coesos em relação aos demais serviços da aplicação;

4. **Ausência de estado (*statelessness*)**: serviços não devem guardar estados em memória entre diferentes chamadas, apenas retornando o valor requerido ou lançando uma exceção. Qualquer noção de estado da aplicação deve ficar gravada no armazenamento secundário, na forma como o problema em questão está representado (ex. um serviço que atue sobre um processo eletrônico não deve levar em consideração chamadas anteriores a este ou outro serviço, mas apenas o estado em que se encontra o processo);

5. **Composição**: serviços podem ser compostos por outros serviços;

6. **Localização Transparente**: serviços devem poder ser chamados a partir de qualquer lugar na rede em que estão;

7. **Respeito a Contratos (definido na API)**: alterações posteriores nas funcionalidades de um serviço não devem impactar os consumidores.

8. **Normalização**: serviços devem ser decompostos ou unificados para minimizar redundância (exceto em casos em que isso não seja possível ou interessante, como otimização de performance ou restrições de acesso específicas);

Com isso, o fluxo básico para desenvolvimento de uma aplicação seguindo a arquitetura proposta é o seguinte:

1. Identificar as funções de negócio (ou seja, aquelas que o usuário perceba como tendo valor) que a aplicação deve abranger;
2. Identificar funcionalidades com potencial de serem utilizadas em diversas aplicações;
3. Identificar as menores unidades que fornecem as funções identificadas nas etapas anteriores e criar um serviço para cada uma delas (ou reutilizar serviços já existentes que atendam a essas funções);

4. Criar uma interface (de usuário) para possibilitar o uso coordenado dos serviços que compõem a aplicação. A interface em si pode ser servida por um serviço próprio.

No caso de manutenção de aplicações existentes, esse fluxo se aplica na inclusão de novas funcionalidades (com o passo 4 passando a ser a adequação da interface existente para utilizar o serviço que implementa a nova funcionalidade). É interessante que o fluxo seja aplicado de forma iterativa, conforme discutido anteriormente.

A arquitetura, conforme proposta, apresenta benefícios que podem ser correlacionados às características desejáveis listadas no início do documento. Tais benefícios são listados a seguir.

Benefícios da Arquitetura

- Possibilidade de substituir partes do sistema, independentemente das demais
- Propicia a integração e entrega contínua: mudanças em um serviço requer a substituição apenas daquele serviço (possivelmente representando uma pequena parte da aplicação)
- Facilita a escalabilidade (exemplo: facilidade de replicação dos serviços sob maior demanda)
- Autonomia de desenvolvimento, manutenção e implantação dos serviços por equipes diferentes - e possivelmente menores (por encapsular funcionalidades mais coesas, pode-se entender o todo mais facilmente, facilitando manutenção, diminuindo a curva de aprendizado para novos membros ou equipes)
- Diferentes serviços podem usar diferentes tecnologias (ex. linguagens de programação), permitindo usar ferramenta adequada para cada problema, melhorando integração com sistemas legados (bastando encapsulá-lo e expor uma interface), e diminuindo o custo de atualização tecnológica (pode-se selecionar os serviços a serem atualizados - “nada prende as equipes a uma tecnologia específica”)

- Facilidade de integração entre sistemas internos ou de terceiros (por uma interface padronizada em lugar de acoplamento entre BDs)
- Facilidade de atualização/implantação das aplicações (se é preciso dar manutenção de apenas uma parte da funcionalidade, apenas esse serviço deve ser “desligado/substituído” e não a aplicação completa, com menor impacto sobre os usuários; deploy de sistemas menores - os serviços - é geralmente mais simples)
- Organização do código em torno de necessidades de negócio (*um serviço para cada necessidade*)
- Isolamento de falhas (se um serviço falha, os demais - e aplicação majoritariamente - pode continuar funcionando, sendo possível reiniciar/reparar partes isoladas da aplicação)

Desvantagens

- Aumento da complexidade inerente a sistemas distribuídos, como tolerância a falhas, latência de rede, diversidade de formatos de mensagem, escalabilidade
- Aumento da dificuldade na realização de testes e integração dos sistemas complexos, devido ao desenvolvimento distribuído, principalmente quando o número de serviços aumenta.
- Com muitos serviços surge a necessidade de gerenciar: diferentes formas de acesso, interface, o entendimento sobre as entidades dos domínios
- Risco de duplicação de esforço
- Casos de uso podem envolver diversos serviços, implicando em dificuldades de manter atomicidade das transações e integridade dos dados, além de comunicação e integração entre as equipes

Princípios Norteadores

A proposta de arquitetura apresentada neste documento foi concebida levando em consideração os princípios *SOLID* [3], resumidos a seguir:

- ***Single Responsibility Principle***: cada módulo deve ser responsável por apenas uma parte da funcionalidade da aplicação, que deve estar completamente contida no módulo e todos os serviços providos pelo módulo devem dizer respeito a essa responsabilidade.
- ***Open/Closed Principle***: um artefato de software deve ser aberto para extensão, mas fechado para modificação (ou seja, propriedades/funcionalidades adicionais não devem ser incluídas pela modificação do artefato de software original, mas pela criação de artefatos de software que o especializem).
- ***Liskov Substitution Principle***: as propriedades comportamentais de um tipo de objeto devem ser preservadas pelos subtipos, de modo que seja possível substituir uma instância do tipo por uma instância do subtipo sem que o programa se torne incorreto.
- ***Interface Segregation Principle***: o “cliente” de um módulo não deve depender de funções desse módulo que não utilize. Assim, o módulo deve oferecer interfaces em maior granularidade, específicas para cada tipo de cliente.
- ***Dependency Encapsulation Principle***: um artefato de software não deve depender diretamente de artefatos de software externos à aplicação, mas sim de abstrações desses artefatos (ou seja, dependências externas devem ser encapsuladas).

Aplicação a Aspectos Recorrentes

Aqui descrevemos aspectos técnicos e funcionais que ocorrem frequentemente no desenvolvimento de aplicações. Esses aspectos são apresentados como recomendações para balizar o desenvolvimento dos serviços.

Dada a complexidade de definir uma arquitetura genérica para todas as aplicações do tribunal, essas recomendações são propostas como aspectos independentes, ortogonais em relação à organização das funcionalidades em serviços (ou seja, aspectos presentes em cada serviço).

Para cada aspecto considerado, estão definidos três níveis de aderência — mínimo, desejável e ideal — ao que julgamos adequado na implementação de um serviço. Esses níveis, na ordem proposta, indicam um caminho gradativo para alcançar o objetivo descrito em cada aspecto, levando em consideração a realidade de cada grupo de desenvolvimento e cada aplicação específica.

Comunicação

Descrição: Para utilizar as funcionalidades implementadas por um serviço, o usuário deve acessá-lo de forma simples e confiável.

Objetivo: Possibilitar a comunicação simples e confiável entre os usuários e os serviços.

Níveis:

- Mínimo: Comunicação usando REST API com transporte via HTTP e codificado usando JSON
- Desejável: Seguir o padrão definido no catálogo de serviços
- Ideal: Seguir o padrão definido no catálogo e disponibilizar um serviço que descreva a API (swagger)

Persistência de dados

Descrição: Serviços devem ser desacoplados para que possam ser implantados e escalados de forma independente. Isto requer que a camada de persistência de dados não interfira nesse processo, trazendo acoplamento a diferentes serviços. Serviços podem ter diferentes requisitos de armazenamento de dados, portanto cada serviço deve usar a melhor ferramenta para suas necessidades.

Objetivo: Minimizar a dependência da aplicação em relação à implementação específica de armazenamento de dados (evitando a dependência de fornecedor). Obter um nível maior de independência do exato modelo de dados no banco. Regrar o acesso de cada serviço ao seus próprios dados, mantendo o acesso a estes dados privados apenas pela sua API.

Níveis:

- Mínimo: Banco de dados “acessível” apenas por uma interface independente das regras de negócio da aplicação. Não utilizar idioma nativo do banco de dados diretamente nas regras de negócio. Cada serviço deve ter seu próprio conjunto de tabelas, não compartilhando-as com outros serviços.
- Desejável: Utilização de algum tipo de mapeamento de dados (ex. ORM) para aumento da independência em relação ao fornecedor. Cada serviço deve ter seu próprio esquema de dados.
- Ideal: Independência da aplicação em relação ao fornecedor de banco de dados, tornando possível alterar o banco de dados sem ter de alterar elementos na aplicação (apenas configurações).

Recomendação: Utilizar Hibernate como data mapper para Java EE.

Testabilidade

Descrição: A divisão de um sistema em serviços possui um relevante impacto nos testes, trazendo desafios que não são encontrados em outros padrões arquiteturais.

Objetivo: Testar eficientemente e eficazmente funcionalidades de um sistema distribuído como o de microserviços.

Níveis:

- Mínimo: Testes unitários para as funcionalidades de cada serviço.
- Desejável: Testes da interface exposta por um serviço, criando *stubs* para colaboradores externos para que apenas o serviço em si seja escopo do teste.

- Ideal: Testes de ponta a ponta (integração), levando em consideração interação entre serviços.

Recomendação: Utilização de frameworks de teste unitário (JUnit para java), frameworks de stub/mocking e de automatização de testes.

Regras de negócio

Descrição: Lógica que determina o funcionamento do software conforme esperado pelo usuário. É o conhecimento do negócio realizado na forma de algoritmos.

Objetivo: Permitir que as regras de negócio, sendo a parte mais importante do software, sejam, tanto quanto possível, isoladas do resto do código, de modo a facilitar sua adaptação e reaproveitamento.

Níveis

- Mínimo: Separar regras de negócio e elementos de domínio em módulos isolados do resto do software
- Desejável: organização das regras de negócio e as interfaces de que elas dependem em um módulo isolado
- Ideal: Nos pontos onde as regras de negócio dependem do resto do software inverter a relação usando o princípio da inversão de dependência

Recomendação: Não se aplica recomendação de tecnologia por tratar-se de um conceito que antecede a escolha e aplicação de uma tecnologia particular.

Apresentação

Descrição: A apresentação diz respeito à parte do software responsável pela interação com o usuário, é onde todos os recursos do software são disponibilizados de forma coesa e organizada para a utilização por parte desse último.

Objetivo: Serviço de apresentação independente, podendo ser alterado a qualquer momento sem necessidade de alteração nos outros serviços.

- Mínimo: Separação de Back-End e Front-End
- Desejável: Todas as regras e validações de segurança devem estar presentes no Back-End. Regras no Front-End são opcionais e apenas com o objetivo de facilitar a utilização (ex. organizar o fluxo de uso, a ordem em que os serviços são chamados).
- Ideal: O serviço de apresentação deve receber do Back-End uma descrição das regras a serem ser usadas na apresentação (para evitar a duplicação de regras de negócio).

Recomendação de tecnologias: Back-End Java EE, Front-End Angular 6+

Autenticação e Autorização

Descrição: Ao integrar diversos serviços, o controle de acesso é necessário para evitar utilização incorreta dos serviços.

Objetivo: Realizar um controle de acesso com tempo de expiração e evitar o armazenamento dessas sessões no servidor que hospeda o serviço. A implementação do controle de acesso deve ser realizada no **pré-processamento** da requisição, usando papéis para configurar o seu acesso.

- Mínimo: Cada aplicação controla o acesso usando por token.
- Desejável: Autenticação externa com controle de sessão no serviço. Ao identificar um acesso não autorizado, o serviço redireciona o usuário ao serviço de acesso, que atribuíra um token ao usuário. O serviço usará esse token para autenticar o usuário.
- Recomendado: Autorização e controle de acesso geridos pelo serviço de autenticação, que gera um token assinado com uma chave assimétrica que é conferida pelo serviço no momento de acesso. Cada serviço deve utilizar papéis para o controle de acesso, que serão conhecidos pelo serviço de autenticação.

Recomendação: JWT

Definições

Single Responsibility Principle (SRP): Conceito que propõe relacionar cada artefato de software com um problema a ser resolvido.

Interface Segregation Principle (ISP): Conceito de código com limites bem definidos.

Arquitetura de software: Definição da estrutura de alto nível de um software que disciplina as relações entre suas partes, cuja alteração posterior é custosa.

Padrão arquitetural: solução geral e reutilizável para um problema comum em arquitetura de software em certo contexto (como desempenho, disponibilidade ou risco operacional).

Integração contínua: mesclagem frequente de linhas de trabalho independente numa única linha principal, para evitar o acúmulo de conflitos de alteração, e normalmente com controle de qualidade automatizado.

Retrocompatibilidade: interoperabilidade com comportamentos de um sistema mais antigo

Encapsulamento: Isolamento do acesso aos mecanismos internos de um componente de software.

Cenário: narrativa das interações previstas entre atores humanos e um sistema, normalmente feita por especialistas em usabilidade

Funcionalidade: capacidade de um sistema que atende total ou parcialmente de um requisito do usuário.

Interface: descrição do comportamento da conexão entre dois sistemas ou componentes que lhes permite trocar informações

API: interface básica composta por subrotinas e protocolos que permite o desenvolvimento de aplicações

Endpoint: ponto de entrada (URL) para serviços web relacionados a uma API.

Módulo: separação de um aspecto/funcionalidade de um software com o objetivo de reduzir sua dependência/acoplamento de outras partes do mesmo software

Sistema: software constituído de vários componentes e/ou módulos, incluindo especificações, testes e documentação, que atende à uma ou mais necessidade(s) do negócio

Serviço: conjunto de funcionalidades reutilizáveis por diferentes clientes, com interface bem definida e políticas de controle de acesso

Sistema Satélite: Sistema que complementa a funcionalidade de outro sistema principal.

Catálogo de serviço: Serviço que lista e descreve os serviços existentes em uma organização relacionando cada funcionalidade com seu endpoint.

Escalabilidade: capacidade de um sistema em lidar com maiores quantidade de trabalho

Monitoramento: observação do estado de execução (disponibilidade, eficiência, comportamento) de um ou mais sistemas

Segurança: conjunto de funcionalidades reutilizáveis por diferentes clientes, com interface bem definida e políticas de controle de acesso. Práticas que visam assegurar confidencialidade, integridade e disponibilidade de dados de usuários

Auditoria: verificação e validação do comportamento e do uso de um sistema

Falha: Comportamento divergente em relação a especificação.

Tolerância a falhas: capacidade de continuar em operação total ou parcial após uma falha

Referências

[1] **Architectural Principles**. Disponível em:

<http://www.opengroup.org/public/arch/p4/princ/princ.htm>

[2] Martin Fowler**, on Characteristics of Microservices**. Disponível em:

<https://www.infoq.com/news/2014/11/gotober-fowler-microservices>

[3] MARTIN, R. C. **Clean architecture: a craftsman's guide to software structure and design/ Robert C. Martin**. Traducao . [s.l.] Prentice Hall, 2018.

[4] **Microservice Trade-Offs**. Disponível em:

<https://martinfowler.com/articles/microservice-trade-offs.html>

[5] **Microservices**. Disponível em:

<https://martinfowler.com/articles/microservices.html>

[6] **Microservices - Not a free lunch! - High Scalability -**. Disponível em:

<http://highscalability.com/blog/2014/4/8/microservices-not-a-free-lunch.html>

[7] **Microservices - Not a free lunch! - High Scalability -**. Disponível em:

<http://highscalability.com/blog/2014/4/8/microservices-not-a-free-lunch.html>

[8] **Microservices Pattern: Microservice Architecture pattern**. Disponível em:

<http://microservices.io/patterns/microservices.html>

[9] NADAREISHVILI, I. et al. **Microservice Architecture: Aligning Principles, Practices, And Culture**. Traducao . [s.l.] O'Reilly, 2016.

[10] NEWMAN, S. **Building Microservices**. Traducao . [s.l: s.n.].

[11] **Owning Your Dependencies**. Disponível em:

<https://8thlight.com/blog/patrick-gombert/2012/09/24/owning-your-dependencies.html>

[12] **Software architecture**. Disponível em:

<https://en.wikipedia.org/wiki/Software_architecture#Software_architecture_topics>

[13] **Testing Strategies in a Microservice Architecture**. Disponível em:

<https://martinfowler.com/articles/microservice-testing/>

[14] **What is Microservices Architecture?** Disponível em:

<https://smartbear.com/learn/api-design/what-are-microservices/>

[15] WOLFF, E. **Microservices: flexible software architecture**. Traducao . [s.l.] Addison-Wesley, 2017.

[16] WOLFF, E. **Microservices: flexible software architecture**. Traducao . [s.l.] Addison-Wesley, Pearson, 2017.

[17] **bliki: MicroservicePrerequisites**. Disponível em:

<https://martinfowler.com/bliki/MicroservicePrerequisites.html>

Anexos

Ver este documento.