2021/04/22 - AULA 04.1

Laboratório de Sistemas Computacionais Complexos

https://uclab.xyz/sistemas-complexos-2021-aula04-1

Renato Cordeiro Ferreira
renatocf@ime.usp.br

João Francisco Daniel
joaofran@ime.usp.br

Alfredo Goldman
gold@ime.usp.br

Thatiane de Oliveira Rosa
thatiane@ime.usp.br

Em caso de dúvidas

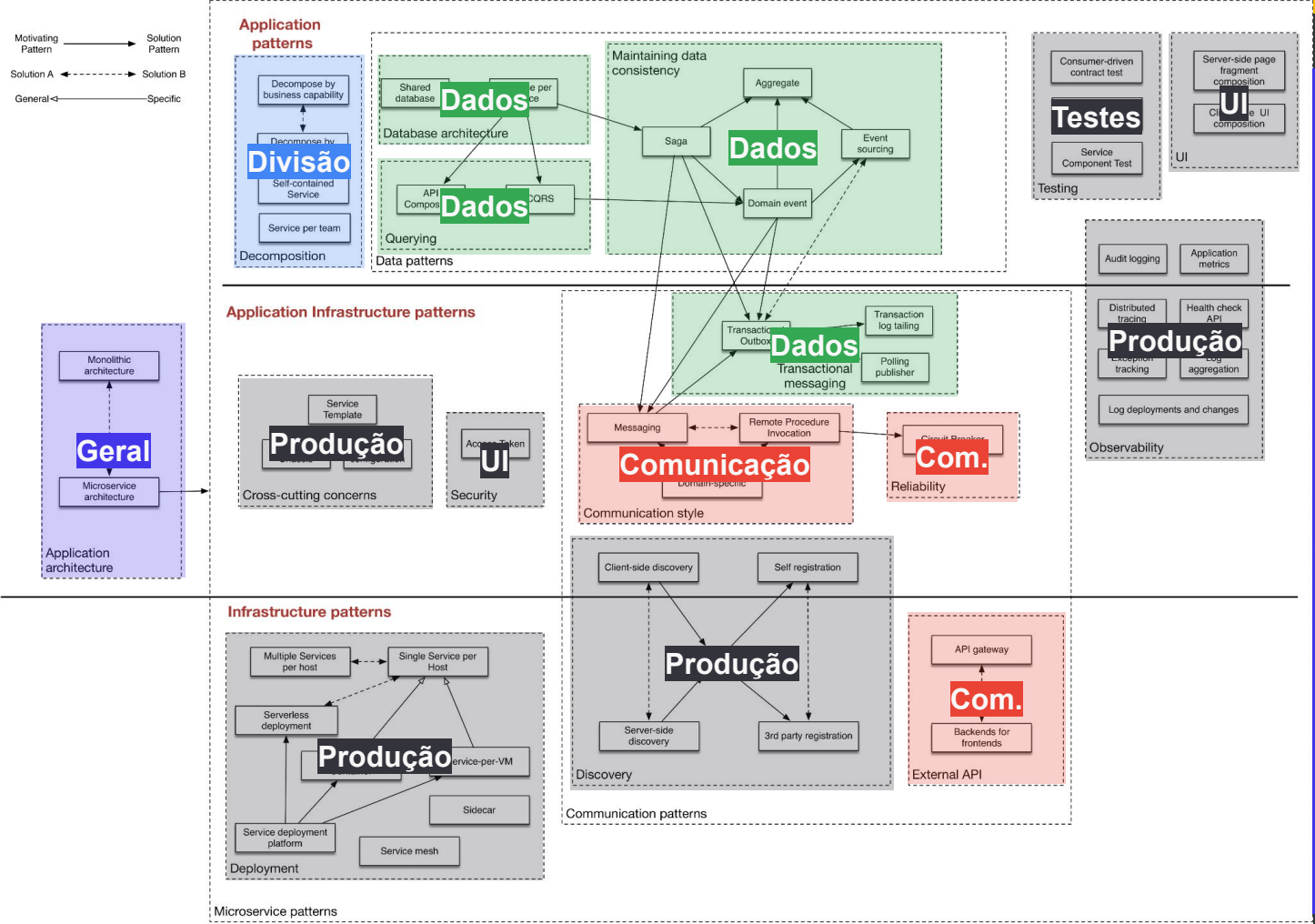
Acessem slido.com com #complexos

ou

Agenda

Tema da aula:
Introdução aos Padrões de Microsserviços

1. Desafios do uso de Microsserviços
2. Arquitetura Dados: Shared Database vs. Database per Service
3. Consulta Dados: API Composition vs. Command-Query Responsability Seggregation
4. Transação Dados: Orchestrated Sagas vs. Choreographed Sagas
5. Consistência Dados: Transactional Outbox vs. Event Sourcing
6. Sistemas Reativos
7. Microsserviços Reativos

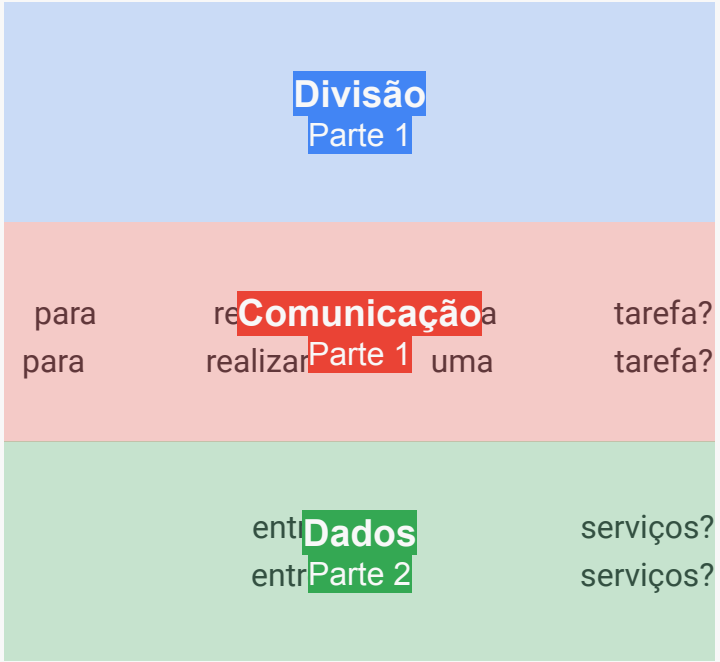


Microservices Pattern Language

Chris Richardson

Desafios do uso de Microsserviços

1. **Divisão**
   - Como definir a responsabilidade de cada serviço?
   - Como definir a quantidade de serviços?
   - Como definir o tamanho dos serviços?
2. **Comunicação**
   - Como coordenar múltiplos serviços para realizar uma tarefa?
   - Como acessar múltiplos serviços para realizar uma tarefa?
   - Como resistir à perda de um serviço?
3. Dados
   - Como dividir dados entre serviços?
   - Como juntar dados entre serviços?
   - Como replicar dados entre serviços?



Arquitetura de Dados

agente	requisito
autonomia	complexidade

Problema

Qual a melhor arquitetura de banco de dados para aplicações compostas de serviços?

Forças

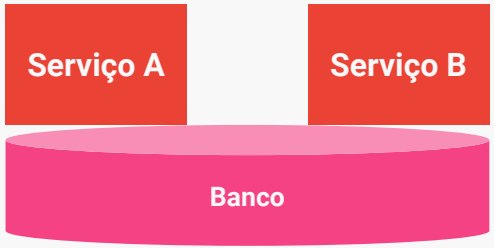
- Serviços devem ser fracamente acoplados para serem desenvolvidos, entregues e escalados independentemente
- Algumas transações precisam impor invariantes sobre dados que pertencem a múltiplos serviços.
- Algumas transações precisam atualizar dados que pertencem a múltiplos serviços.
- Algumas transações precisam consultar dados que pertencem a múltiplos serviços.
- Algumas consultas devem cruzar dados que pertencem a múltiplos serviços.
- Bancos de dados devem ser replicados e fragmentados para escalar.
- Diferentes serviços requerem diferentes modelos de armazenamento (relacional, documento, grafos, etc.)

Arquitetura de Dados

agente	requisito
autonomia	complexidade

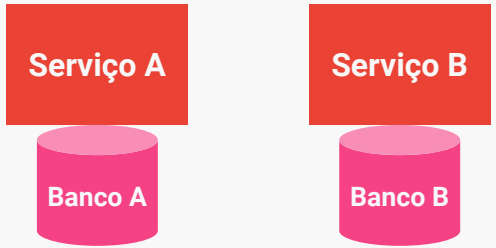
Shared Database

Mantenha dados dos serviços **acessíveis por um banco de dados compartilhado**. Transações ACID podem **acessar livremente dados de todos serviços**.



Single Database per Service

Mantenha dados dos serviços **acessíveis apenas por chamadas às suas APIs**. Transações ACID podem **acessar apenas dados de um serviço específico**.



Arquitetura de Dados

agente	requisito
autonomia	complexidade

Shared Database

Mantenha dados dos serviços **acessíveis por um banco de dados compartilhado**. Transações ACID podem **acessar livremente dados de todos serviços**.

- Serviços usam um único tipo de banco de dados.
- Serviços possuem alto acoplamento pois mudanças no schema de dados afetam múltiplos serviços.
- Consistência mais simples usando transações ACID.
- Cruzamento de dados mais simples usando joins.
- Manutenção mais simples com único banco de dados.

Single Database per Service

Mantenha dados dos serviços **acessíveis apenas por chamadas às suas APIs**. Transações ACID podem **acessar apenas dados de um serviço específico**.

- Serviços podem usar vários tipos de banco de dados.
- Serviços possuem baixo acoplamento pois mudanças no schema de dados afetam um único serviço.
- Consistência mais complexa sem transações ACID.
- Cruzamento de dados mais complexo sem joins.
- Manutenção mais complexa com vários bancos de dados.

Consulta, Transação, Consistência

Problemas

- Como implementar consultas que recuperam dados de múltiplos serviços?
- Como implementar transações envolvendo múltiplos serviços?
- Como atualizar um banco de dados e publicar mensagens/eventos de modo confiável e atômico?
- Como publicar um evento quando o estado de um banco de dados muda?
- Como publicar um evento quando o estado de um serviço muda?

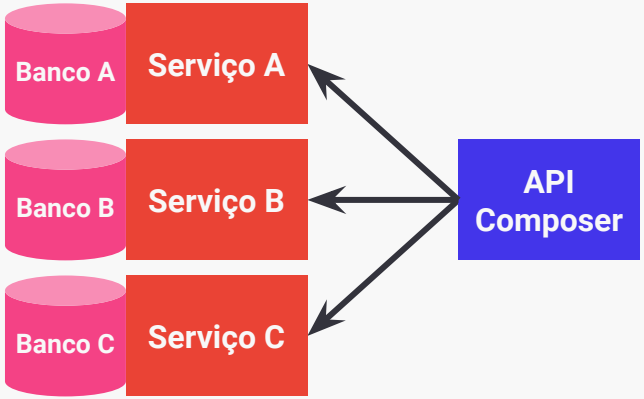
Forças

- Uso de Two-Phase Commit (2PC) para transações distribuídas não é uma opção.

Consulta de Dados

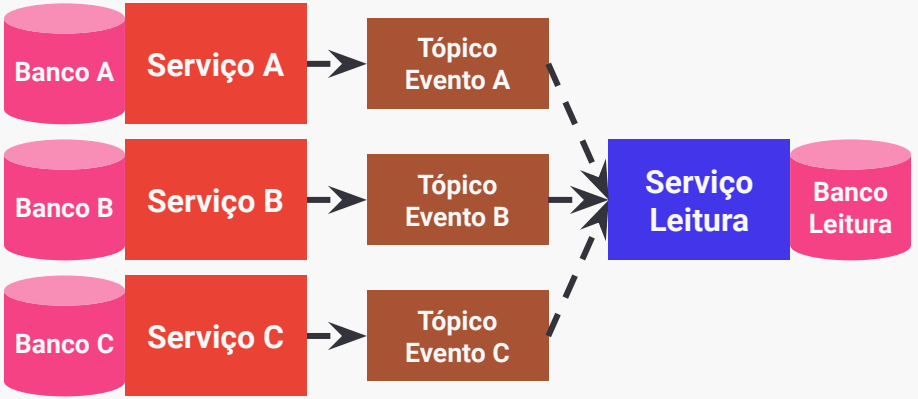
API Composition

Implemente uma consulta via um **API Composer**, que consulta os serviços que possuem os dados e faz uma cruzamento em memória dos resultados.



CQRS (Command and Query Responsibility Segregation)

Implemente a consulta via um **Serviço de Leitura**, que recebe eventos dos serviços que possuem os dados e altera um banco para leitura de dados.



Consulta de Dados

agente	requisito
autonomia	complexidade

API Composition

Implemente uma consulta via um **API Composer**, que consulta os serviços que possuem os dados e faz uma cruzamento em memória dos resultados.

- Maior dependência do uso da memória para operações.
- Implementação de comandos e consultas mais simples.
- Menor complexidade de manutenção por requerer a criação de um serviço sem banco de dados (stateless).
- Menor potencial de duplicação de código.
- Consistência forte pela comunicação síncrona para consulta dos serviços proprietários dos dados.

CQRS (Command and Query Responsibility Segregation)

Implemente a consulta via um **Serviço de Leitura**, que recebe eventos dos serviços que possuem os dados e altera um banco para leitura de dados.

- Menor dependência do uso da memória para operações
- Modelo de comando e consulta mais simples
- Maior complexidade de manutenção por requerer a criação de um serviço com banco de dados (stateful)
- Maior potencial de duplicação de código.
- Consistência eventual pela comunicação assíncrona para atualização a partir dos serviços proprietários dos dados.

Transação de Dados

agente	requisito
autonomia	complexidade

Saga

Implemente cada transação de negócio que passa por múltiplos serviços como uma saga. A saga é uma sequência de múltiplas transações locais. Cada transação local atualiza o banco de dados e publica um comando ou evento que dispara a próxima transação local da saga. Se uma transação local falha porque violou a regra de negócio, então a saga executa uma série de transações de compensação que desfazem as mudanças feitas pelas transações locais anteriores.

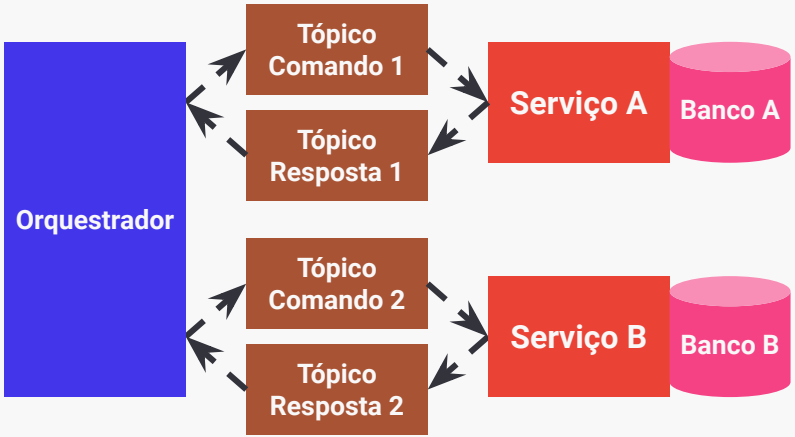
- Aumenta acoplamento durante execução pois atrela funcionamento de cliente e servidor da requisição.
- Requer utilização de protocolos de rede que geralmente não demandam infraestrutura adicional.
- Possui disponibilidade reduzida pois protocolo de rede exige que cliente e servidor estejam disponíveis durante a interação requisição-resposta.

Transação de Dados

agente	requisito
autonomia	complexidade

Saga Orquestrada

Um **orquestrador** dispara comandos para serviços realizarem transações locais e compõe as respostas.



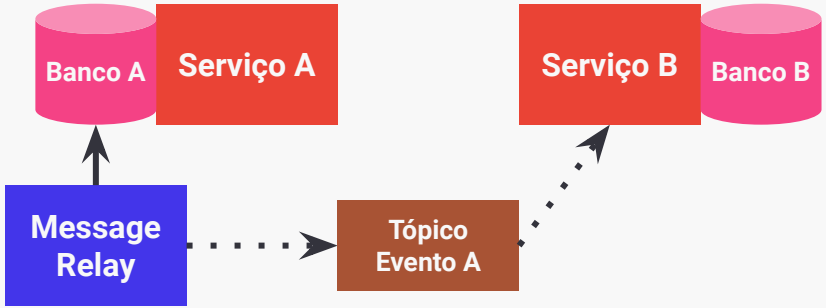
Saga Coreografada

Serviços realizam transações locais em reação a **eventos de domínio** disparados por outros serviços.

Consistência de Dados

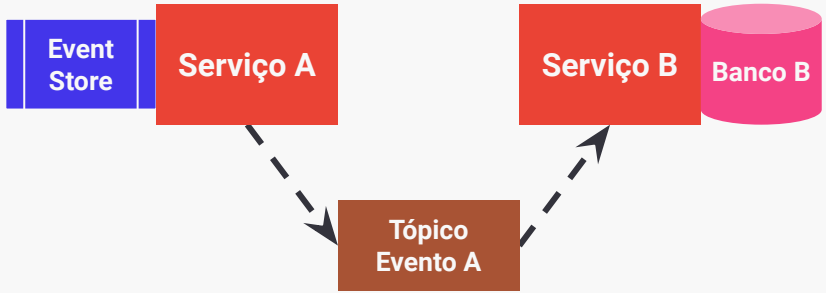
Transactional Outbox

Utilize um **Message Relay** para acompanhar atualizações feitas no banco de dados do serviço e publicar eventos de domínio para outros serviços.



Event Sourcing

Utilize uma **Event Store** para publicar eventos de domínio sobre as atualizações ocorridas ne disponibilizá-lo para outros serviços.



Consistência de Dados

Transactional Outbox

Utilize um **Message Relay** para acompanhar atualizações feitas no banco de dados do serviço e publicar eventos de domínio para outros serviços.

- Maior dependência do uso da memória para operações.
- Implementação de comandos e consultas mais simples.
- Menor complexidade de manutenção por requerer a criação de um serviço sem banco de dados (stateless).
- Menor potencial de duplicação de código.
- Consistência forte pela comunicação síncrona para consulta dos serviços proprietários dos dados.

Event Sourcing

Utilize uma **Event Store** para publicar eventos de domínio sobre as atualizações ocorridas ne disponibilizá-lo para outros serviços.

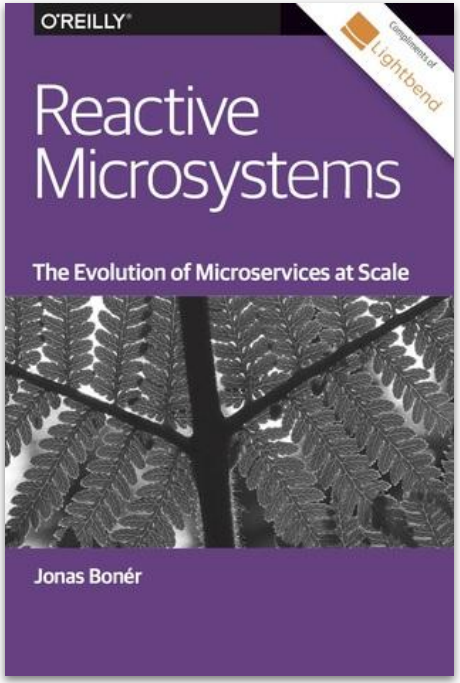
- Menor dependência do uso da memória para operações
- Modelo de comando e consulta mais simples
- Maior complexidade de manutenção por requerer a criação de um serviço com banco de dados (stateful)
- Maior potencial de duplicação de código.
- Consistência eventual pela comunicação assíncrona para atualização a partir dos serviços proprietários dos dados.

Sistemas Reativos

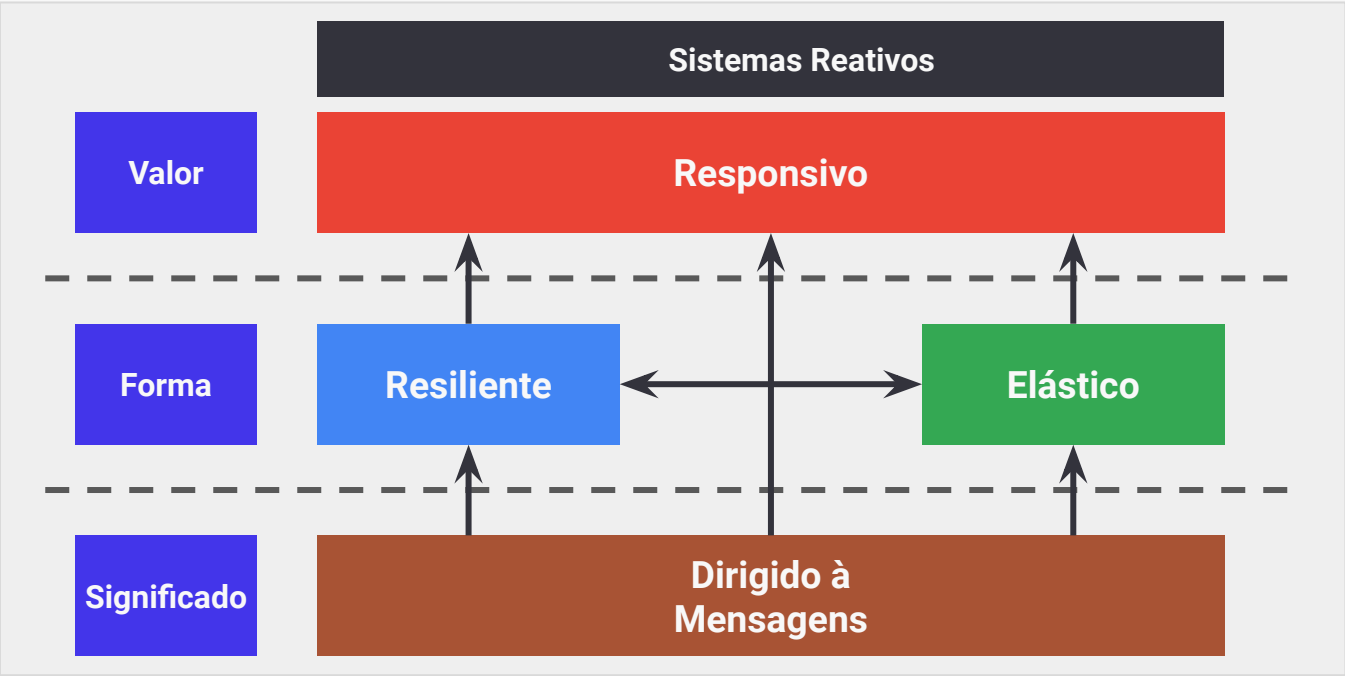
*"Sistemas Reativos — como definido pelo **Manifesto Reativo** — é um conjunto de princípios de design arquitetural para construção de sistemas distribuídos modernos que estão bem preparados para corresponder às demandas de responsividade sob falhas (resiliência) e sob carga (elasticidade) que aplicações precisam hoje em dia.*

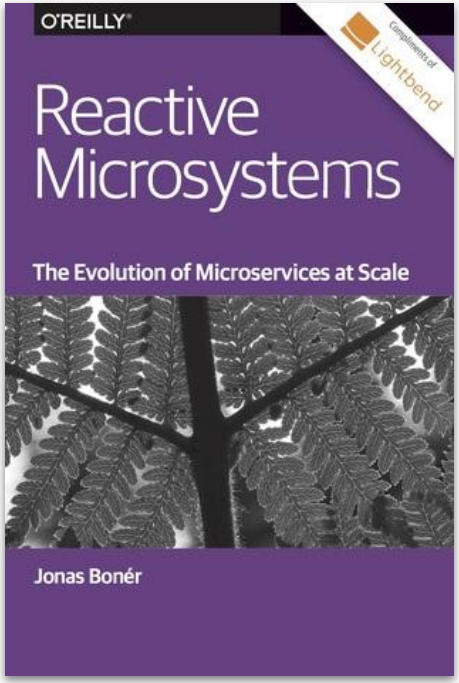
A fundação de um Sistema Reativo é a passagem assíncrona de mensagens, que ajuda a construir sistemas fracamente acoplados com componentes autônomos e colaborativos. Ter uma fronteira assíncrona entre componentes é necessário para desacoplar a eles e a seu fluxo de comunicação no tempo (o que permite concorrência) e espaço (o que permite distribuição e mobilidade)."

-- Jonas Bonér, **Reactive Microsystems: The Evolution of Microservices at Scale** (Cap 5)

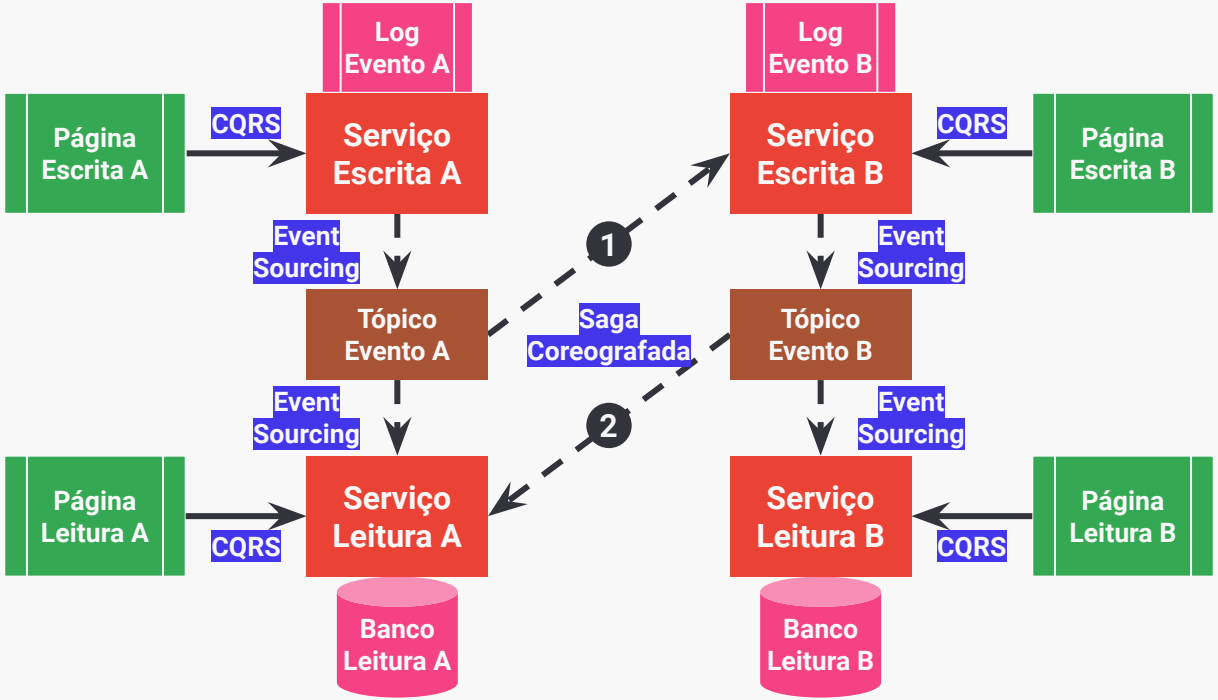


Sistemas Reativos





Microsserviços Reativos



Licença

Estes slides são concedidos sob uma Licença Creative Commons. Sob as seguintes condições: **Atribuição, Uso Não-Comercial e Compartilhamento pela mesma Licença**

Mais detalhes sobre essa licença em: creativecommons.org/licenses/by-nc-sa/3.0/

Créditos

Imagens usadas nesta apresentação são provenientes de: freepik.com

Frequência

Senha do Estudante: **2lkkwk**

2021/04/22 - AULA 04.1

Laboratório de Sistemas Computacionais Complexos

https://uclab.xyz/sistemas-complexos-2021-aula04-1

Renato Cordeiro Ferreira
renatocf@ime.usp.br

João Francisco Daniel
joaofran@ime.usp.br

Alfredo Goldman
gold@ime.usp.br

Thatiane de Oliveira Rosa
thatiane@ime.usp.br