APRESENTAÇÃO
DE APOIO

# MICROSERVIÇOS

Luis Fernando Planella Gonzalez – Aula 03

PUCRS online uol edtech

# Professores

**VINICIUS SOARES**
Professor Convidado

Entusiasta da Computação Distribuída, Vinicius Soares é Head de Tecnologia em uma das principais empresas do sul do país, ajudando clientes e empresas a alcançarem seus resultados de forma rápida e assertiva. Apaixonado por Java, Arquitetura de Sistemas e Computação em Nuvem, Vinicius possui sólida experiência liderando equipes de Arquitetura usando SOA e Microserviços com tecnologias Open-sources. Compartilha suas experiências através de conteúdo online e eventos nacionais e internacionais como Devoxx, TDC e Campus Party. Como empreendedor, já ajudou mais de 1000 pessoas a se qualificarem para o mercado de TI e atuarem de forma representativa na área.

**LUIS FERNANDO PLANELLA GONZALEZ**
Professor PUCRS

Doutor Ciências da Computação (PUCRS, 2018). Desenvolvedor e arquiteto Java com experiência profissional desde 1999, certificado pela Sun como programador e desenvolvedor de componentes web na plataforma Java. Entusiasta de software livre.

# Ementa da disciplina

Estudo sobre a arquitetura de microserviços. Estudo sobre os conceitos de particionamento de serviços, replicação e distribuição, comunicação assíncrona via filas e Soluções serveless..

PUCRS online

APRESENTAÇÃO
DE APOIO

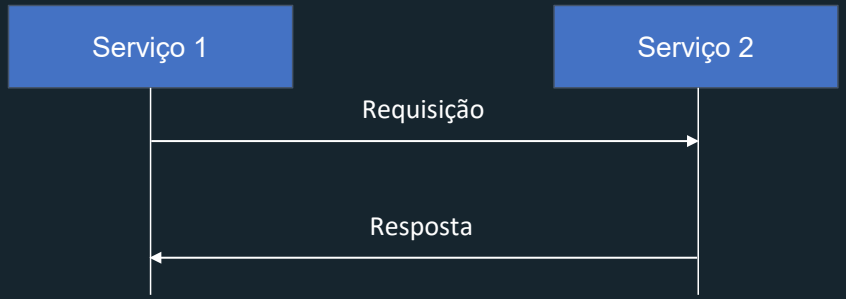
# MicroServiços

**Por Luis Fernando Planella Gonzalez**

# Comunicação assíncrona via filas

# Revisão - Modelo de requisição / resposta

- Também chamada de cliente / servidor
- Pode ser síncrono ou assíncrono
- O cliente envia uma requisição ao servidor
- O servidor retorna uma resposta ao cliente
  - Ou ocorre um erro!

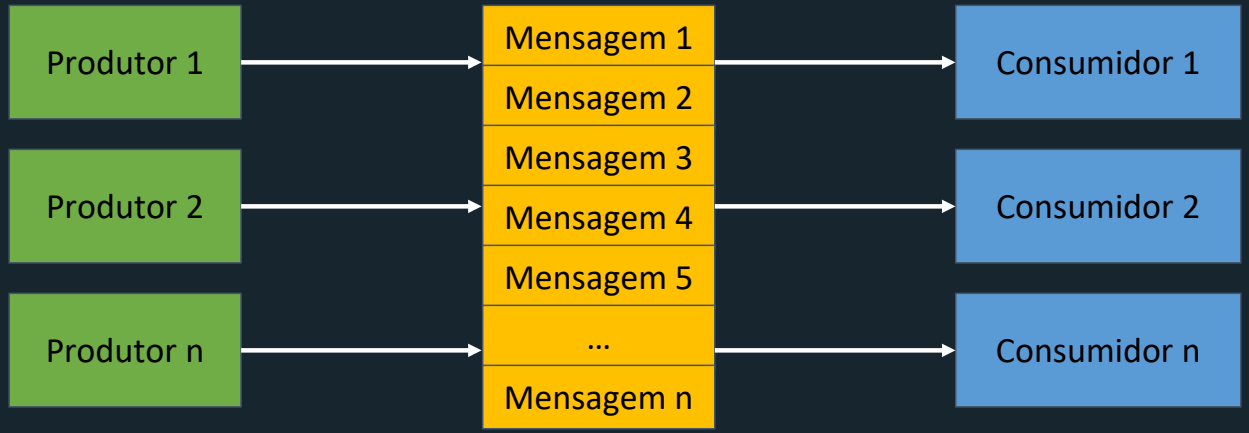


# Revisão - Modelo de requisição / resposta

- Há um alto acoplamento entre o cliente e o servidor
- Ambos serviços precisam estar responsivos no momento
- Uma falha no servidor gera uma falha no cliente

# Modelo produtor / consumidor (pub / sub)

- **Produtor** ou **publicador**: gera as mensagens
- **Consumidor** ou **subscritor**: notificado quando há mensagens
- Há um **desacoplamento** entre o produtor e o consumidor



# Fila de mensagens

- Estrutura em que cada mensagem produzida por um produtor é entregue a um único consumidor
- Adequado para a distribuição de carga
- Quando não há nenhum consumidor registrado ou disponível, a mensagem geralmente é armazenada
  - Assim que um consumidor estiver disponível, a mensagem é entregue

# Tópico de mensagens

- Estrutura em que cada mensagem produzida por um produtor é entregue a todos os consumidores registrados
- *Multicast*
- Geralmente não há persistência das mensagens
- Somente consumidores registrados no momento em que a mensagem é gerada a recebem

*Message broker*

# *Message broker*

- Sistema especializado em recepção e envio de mensagens
- Desconhece detalhes sobre os produtores e consumidores
- Capazes de persistir mensagens
- Capazes de entregar novamente uma mensagem em caso de falha do consumidor
- Existem diversos serviços de mensageria bem conhecidos
  - Exemplos: Kafka, ActiveMQ, RabbitMQ

# *Message broker*

- Um *message broker* confiável é essencial em uma arquitetura de microsserviços
- Importante evitar um ponto único de falha!
  - Ou seja, um componente que, caso falhe, impacta ou indisponibiliza o sistema todo!
  - Replicação / redundância
- Também é importante lidar com problemas de escalabilidade
  - Monitoramento constante, já que tende a ser um componente do sistema bastante demandado

# Exemplos de código

- Os exemplos de código a seguir utilizam o protocolo AMQP (*Advanced Message Queuing Protocol*)
  - Implementado pelo ActiveMQ, RabbitMQ e outros
- Escritos em TypeScript, rodando sobre nodejs
- Disponível em: https://github.com/luisfpg/produtor-consumidor

# Utilitário - messageBroker.ts

```
import client from 'amqplib';
import dotenv from 'dotenv';
dotenv.config();

/** Nome da fila de mensagens */
export const FILA = 'mensagens';

/** Nome do tópico de mensagens */
export const TOPICO = 'eventos';

/** Conecta, cria um canal e o retorna */
export async function conectar() {
  const url = 'amqp://'
    + process.env.AMQP_USERNAME + ':' + process.env.AMQP_PASSWORD
    + '@' + process.env.AMQP_HOST + ':' + process.env.AMQP_PORT;
  const connection = await client.connect(url);
  return await connection.createChannel();
}
```

# Fila - produtor.ts

```
import * as readline from 'readline/promises';
import { conectar, FILA } from '../messageBroker';
const read = readline.createInterface(
  { input: process.stdin, output: process.stdout });

async function main() {
  const canal = await conectar();
  await canal.assertQueue(FILA);

  // Lê o console e envia mensagens para a fila
  while (true) {
    const mensagem = await read.question('Qual a mensagem? ');
    if (mensagem.toLowerCase() == 'sair') { process.exit(0); }
    canal.sendToQueue(FILA, Buffer.from(mensagem));
    console.log(`Mensagem enviada para a fila '${FILA}'\n`);
  }
}

main();
```

# Fila - consumidor.ts

```
import { conectar, FILA } from '../messageBroker';

async function main() {
  const canal = await conectar();
  await canal.assertQueue(FILA);

  console.log(`Escutando mensagens na fila '${FILA}'...`);

  await canal.consume(FILA, msg => {
    console.log(`Mensagem recebida: ${msg.content}\n`);
    canal.ack(msg); // Indica ao canal que a mensagem foi processada
  });
}

main();
```

# Exemplo de saída

```
❯ npm run produtor-fila

> exemplo-rabbitmq@1.0.0 produtor-fila
> ts-node src/fila/produtor.ts

Qual a mensagem? a
Mensagem enviada para a fila 'mensagens'

Qual a mensagem? b
Mensagem enviada para a fila 'mensagens'

Qual a mensagem? c
Mensagem enviada para a fila 'mensagens'

Qual a mensagem? d
Mensagem enviada para a fila 'mensagens'
```

```
❯ npm run consumidor-fila

> exemplo-rabbitmq@1.0.0 consumidor-fila
> ts-node src/fila/consumidor.ts

Escutando mensagens na fila 'mensagens'...
Mensagem recebida: a

Mensagem recebida: c
```

```
❯ npm run consumidor-fila

> exemplo-rabbitmq@1.0.0 consumidor-fila
> ts-node src/fila/consumidor.ts

Escutando mensagens na fila 'mensagens'...
Mensagem recebida: b

Mensagem recebida: d
```

Cada mensagem enviada é entregue a um único consumidor

# Tópico - produtor.ts

```
import * as readline from 'readline/promises';
import { conectar, TOPICO } from '../messageBroker';
const read = readline.createInterface(
  { input: process.stdin, output: process.stdout });

async function main() {
  const canal = await conectar();
  await canal.assertExchange(TOPICO, 'topic', { durable: false });

  while (true) {
    // Lê e envia uma mensagem para a fila
    const mensagem = await read.question('Qual a mensagem? ');
    if (mensagem.toLowerCase() == 'sair') { process.exit(0); }
    canal.publish(TOPICO, '', Buffer.from(mensagem));
    console.log(`Mensagem enviada para o tópico '${TOPICO}'\n`);
  }
}

main();
```

# Tópico - consumidor.ts

```
import { conectar, TOPICO } from '../messageBroker';

async function main() {
  const canal = await conectar();
  await canal.assertExchange(TOPICO, 'topic', { durable: false });
  const fila = await canal.assertQueue('', { exclusive: true });
  await canal.bindQueue(fila.queue, TOPICO, '');

  console.log(`Escutando eventos no tópico '${TOPICO}'...`);

  await canal.consume(fila.queue, msg => {
    console.log('Mensagem recebida: ' + msg.content); console.log();
  }, { noAck: true });
}

main();
```

# Tópico - Exemplo de saída

```
❯ npm run produtor-topico

> exemplo-rabbitmq@1.0.0 produtor-topico
> ts-node src/topico/produtor.ts

Qual a mensagem? a
Mensagem enviada para o tópico 'eventos'

Qual a mensagem? b
Mensagem enviada para o tópico 'eventos'

Qual a mensagem? c
Mensagem enviada para o tópico 'eventos'

Qual a mensagem? d
Mensagem enviada para o tópico 'eventos'
```

```
❯ npm run consumidor-topico

> exemplo-rabbitmq@1.0.0 consumidor-topico
> ts-node src/topico/consumidor.ts

Escutando eventos no tópico 'eventos'...
Mensagem recebida: a

Mensagem recebida: b

Mensagem recebida: c

Mensagem recebida: d
```

```
❯ npm run consumidor-topico

> exemplo-rabbitmq@1.0.0 consumidor-topico
> ts-node src/topico/consumidor.ts

Escutando eventos no tópico 'eventos'...
Mensagem recebida: a

Mensagem recebida: b

Mensagem recebida: c

Mensagem recebida: d
```

Cada mensagem enviada é entregue a todos os consumidores

Não há persistência dos eventos

# *Tipos de garantia de entrega de mensagens*

Diferentes políticas oferecidas pelo *message broker*

# Entrega “**no máximo uma vez**”

***at-most-once delivery***

- Há uma única tentativa de entrega da mensagem
- Ela é perdida em caso de erro
- Nenhum estado é mantido, portanto é a implementação mais simples e rápida
- Ideal para IoT, por exemplo, com sensores constantemente enviando medições
- Mas não pode ser usada quando perdas eventuais de mensagens não são toleradas

# Entrega “**ao menos uma vez**”
## at-most-once delivery

- É realizada a entrega da mensagem. Em caso de erro ou limite de tempo, ela será entregue novamente
- Há necessidade de manter estado no componente de entrega
- Poderá duplicar o processamento ou resultado
  - Por isso é essencial que o tratamento de mensagens seja **idempotente**, isto é, não deixe o estado do sistema inconsistente se executado mais de uma vez

# Entrega “**exatamente uma vez**”
## exactly-once delivery

- Há a garantia de que cada mensagem seja entregue uma única vez, mesmo que hajam falhas ou limite de tempo
- É o mecanismo mais complexo, pois exige estado em ambos os componentes de entrega e recepção
- O componente de envio deve manter estado para retransmitir mensagens falhadas…
- … e o de recepção deve manter estado para ignorar mensagens que já tenham sido previamente enviadas

# *Exemplos de uso de mensagens*

# Notificar eventos de domínio

- Nesta abordagem, o sistema gera mensagens que representam eventos que ocorrem em um microsserviço
- Para que outros microsserviços possam reagir de acordo
- Exemplos:
  - usuario:registrado:<id>
  - pedido:realizado:<id>
  - pagamento:aprovado:<id>
  - …

# Atualizar caches de dados de outros serviços

- Semelhante a eventos de domínio, mas com os dados
- Importante cuidar do versionamento / lock otimista
- Exemplo:

```
{
 "evento": "usuario:atualizado",
 "versao": 3,
 "nome": "Fulano de Tal",
 "email": "fulano@email.com",
 "login": "fulano"
}
```

# Processamento paralelo

- Neste caso, os consumidores são “trabalhadores”
- Cada evento contém uma seleção do que será processado
  - Identificador(es), intervalo de datas, etc
- Exemplo: quando há grandes volumes de dados a serem processados, pode-se dividir o processamento

*Serverless*

# Aplicações *serverless*

- Aplicações *serverless* **necessitam** de um servidor para rodar!
- Mas elas não sabem **qual** servidor vai rodá-las
- O conceito de serverless é geralmente relacionado ao **FaaS**
  - *Function as a service*
- Mas o conceito pode ser considerado mais amplo
  - Serviços gerenciados (bases de dados, buscas, mensageria, etc) também podem ser considerados *serverless*

# Aplicações *serverless*

- Aplicações *serverless* retiram do operador do sistema a responsabilidade de gerenciar a infraestrutura do sistema
  - Atualizações de segurança do sistema operacional
  - Atualizações do software de base (bibliotecas)
  - Administração de capacidade ou escala
- Assim, o desenvolvedor / operador pode focar-se **apenas** na aplicação

*FaaS - Functions as a Service*

# *FaaS - Functions as a service*

- Neste modelo, o desenvolvedor empacota funções
- Geralmente é utilizado um container (como o Docker)
- A medida que há demanda, o ambiente aloca recursos para executar a função
- Quando a demanda cessa, o ambiente libera recursos
- Adequado para funções de processamento, não para sistemas de persistência de dados

# *FaaS - Functions as a service*

- Todos os principais provedores cloud ofertam *FaaS*
- Mas… Cuidado com o *vendor lock-in*!
- Ocorre se as funções forem escritas utilizando a API de um provedor específico
- Neste caso será muito difícil uma eventual migração para outro provedor se necessário

# *FaaS* - Vantagens

- Otimização de custos, pois somente será cobrado quando houver demanda
- Escala flexível: a infraestrutura vai alocar mais recursos com o aumento de demanda, e desalocar recursos desnecessários
  - Inclusive até chegar ao ponto de nenhum recurso alocado, ou seja, custo zero

# *FaaS* - Desvantagens

- Aumento na complexidade da infraestrutura
- Difícil prever o custo final, pois depende da demanda
- Maior dificuldade na depuração
  - O código de cada função é isolado em um container
- Quando é necessário aumentar a escala, pode ocorrer um atraso devido ao tempo necessário para inicializar a função
  - O tempo de “aquecimento” da função pode impactar na experiência do usuário. O Java é um exemplo notável.

# *Knative*

- Uma solução padronizada emerge: Knative
- Roda sobre o Kubernetes (k8s), que é um padrão de fato
  - Suportado pela grande maioria dos provedores de cloud

# *Knative*

- Expõe cada função em um containers (Docker)
- O *Knative* atribui uma URL para invocar cada função
- A função deve levantar um servidor HTTP
- O *Knative* roteia uma requisição para a função
- Como a única responsabilidade da função é escutar em uma porta e retornar uma resposta, ela pode ser escrita em qualquer linguagem de programação

# *Kubernetes*

- Funciona com containers (Docker)
- Resolução DNS e balanceamento de carga para os serviços
- Escalonamento de serviços, permitindo inclusive a auto-escala (aumento e diminuição na quantidade de réplicas de um serviço de acordo com a demanda)

# *Kubernetes*

- O Kubernetes é um sistema complexo
  - Mas que busca resolver um problema complexo
- Pode ser instalado desde a máquina do desenvolvedor…
  - KinD (Kubernetes in Docker) ou Minikube
- … até clusters de milhares de nodos!
  - Na versão 1.25, até 5.000 nodos!

# Exemplo de código

- O exemplo de código a seguir é escrito em TypeScript, rodando sobre nodejs
- É utilizada a biblioteca Hapi: https://hapi.dev/
- Disponível em: https://github.com/luisfpg/exemplo-knative

# Dockerfile

```
FROM node:16-alpine
RUN mkdir /app
WORKDIR /app
ADD node_modules ./node_modules
ADD src ./src
ADD package.json .
ENTRYPOINT [ "npx", "ts-node", "src/server.ts" ]
```

# server.ts

```
import { server as HapiServer } from '@hapi/hapi';
import { handler } from './handler';
const init = async () => {
  const server = HapiServer({
    port: 3000
  });
  server.route({
    method: '*',
    path: '/',
    handler: handler
  });
  await server.start();
  console.log('Servidor iniciado em %s', server.info.uri);
};
process.on('unhandledRejection', (err) => {
 console.log(err);
});
init();
```

# handler.ts

```ts
import { Lifecycle } from '@hapi/hapi';

export const handler: Lifecycle.Method = (req) => {
 const nome = req.query.nome || 'Mundo';
 return {
   mensagem: `Olá ${nome}!`
 };
};
```

# ola-mundo.yaml

```yaml
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
 name: ola-mundo
 namespace: exemplo
spec:
 template:
   spec:
     containers:
       - image: kind.local/ola-mundo
         imagePullPolicy: IfNotPresent
         ports:
           - containerPort: 3000
```

# Exemplo de saída

```
❯ curl -s "http://ola-mundo.default.127.0.0.1.sslip.io/" | jq
{
  "mensagem": "Olá Mundo!"
}
❯ curl -s "http://ola-mundo.default.127.0.0.1.sslip.io/?nome=Exemplo" | jq
{
  "mensagem": "Olá Exemplo!"
}
```

Nestes exemplos o comando *curl* exibe o resultado de uma requisição HTTP, *-s* para a opção silenciosa (nenhum texto impresso exceto a resposta) e o *jq* para formatar um JSON.

# Aplicações adequadas para *FaaS*

- Nem todo o tipo de aplicação é adequado para o modelo
- Ideal para funções disparadas por eventos
  - Sensores
  - Eventos do domínio de negócio que demandam processamento adicional
  - Processamento multimídia
  - Extração, transformação e carga (ETL) de dados
- Mas não para aplicações que ficam o tempo todo rodando
  - Ou que fazem processamentos excessivamente longos

# Restrições para aplicações *FaaS*

- Como o container que roda a função é efêmero, isto é, será criado e destruído muitas vezes, ele deve:
  - Iniciar rápido (milisegundos)
  - Ocupar pouca memória
  - Responder rápido
- As funções não devem ter estado! Qualquer estado deve ser lido a cada execução da base de dados
- Alguns provedores limitam o tempo máximo que uma função pode rodar (10 - 15 minutos)

# Palavras finais

# Microsserviços

- A arquitetura de microsserviços é uma evolução natural da arquitetura monolítica
- Incorpora aspectos como comunicação de rede, escalabilidade, distribuição / replicação e muito mais!
- Ou seja: é um mundo complexo!
- São muitos conceitos, aspectos, modelos e tecnologias em constante evolução…

# Microsserviços

- Mas todos esses elementos foram desenvolvidos a partir da necessidade de sistemas mais **escaláveis** e **confiáveis**
- Nestes casos, o monolito não é a melhor opção
  - Ou mesmo uma opção viável!

- Fica a sugestão: a melhor forma de consolidar todos os conceitos apresentados é com a prática!

PUCRS online UOL edtech_


PUCRS online
uol edtech