Introdução

Nesse módulo estudamos sobre sistemas reativos e como podemos tirar vantagens programação assíncrona em alto nível de programação.

Sistemas reativos

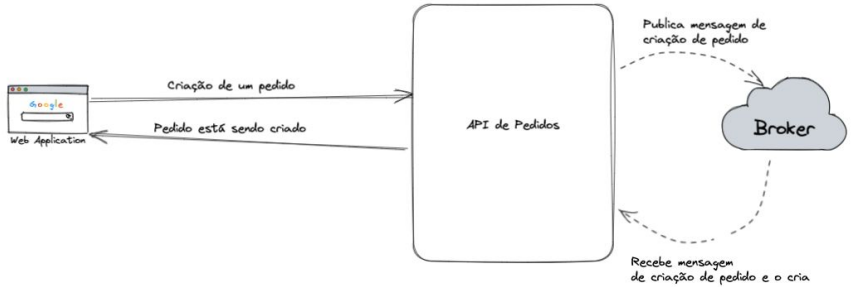
Ao longo dos anos os requisitos de sistemas de grande escala mudaram, exigindo com que as aplicações fossem mais resilientes, escaláveis e performáticas. Nesse sentido, em 2014 foi publicado o **Manifesto Reativo**, documento que define 4 características de Sistemas Reativos:

1. **Responsivo**
2. **Resiliente**
3. **Elástico**
4. **Orientado à Mensagens**

Para ver em detalhes essas 4 características, acesse o site do Manifesto: https://www.reactivemanifesto.org/pt-BR

Mensageria

Mensageria é um conceito que se aplica à comunicação assíncrona entre serviços. Normalmente é usada quando uma aplicação cliente não precisa de uma resposta imediata à sua requisição.



Na imagem acima é descrito um fluxo onde uma aplicação web faz uma requisição de criação de pedido. Ao invés de aguardar o resultado da criação do pedido (o que pode levar alguns minutos devido validações de segurança), a API responde que o pedido está sendo criado e que em breve estará disponível para consulta. Enquanto isso, a API produz uma mensagem em uma fila de criação de pedidos e aguarda seu processamento.

Spring JMS

Spring JMS é um framework **Spring** que torna mais simples a utilização da **JMS API** (**Java Message Service API**), tornando bem fácil sua utilização em nossos projetos **Spring Boot**:

- Documentação da JMS API: https://docs.oracle.com/javaee/6/tutorial/doc/bncdr.html
- Guia do Spring JMS: https://spring.io/guides/gs/messaging-jms/

ActiveMQ

O **ActiveMQ** é um *broker* de mensageria *open source* e multi-protocolo. Ele suporta os padrões de mercado mais comuns, o que permite integração com as linguagens mais populares como Java, JavaScript, C, C++, Python, .NET e outras. Site oficial do ActiveMQ: https://activemq.apache.org/

RabbitMQ

RabbitMQ é também um servidor de mensagem extremamente popular e robusto. Foi desenvolvido na linguagem Erlang e projetado para suportar o protocolo AMQP (Advanced Message Queuing Protocol). Também é possível integrá-lo aos projetos Spring usando o framework Spring AMQP:

- Site oficial do RabbitMQ: https://www.rabbitmq.com/
- Protocolo AMQP: https://pt.wikipedia.org/wiki/Advanced_Message_Queuing_Protocol
- Documentação do Spring AMQP: https://spring.io/projects/spring-amqp

Apache Kafka

O Apache Kafka é uma plataforma de mensageria distribuída desenvolvida pelos engenheiros do LinkedIn e foi projetada para ser robusta, performática e confiável.

Se diferencia por também persistir todas as mensagens que foram processadas, permitindo que, se necessário, sejam reprocessadas. É a mais completa (e complexa) plataforma de mensagem, ideal para sistemas distribuídos de larga escala:

Site oficial do Apache Kafka: https://kafka.apache.org/

Microsserviços

Microsserviços é uma abordagem de arquitetura e organização de software que consiste em pequenos serviços que agem em conjunto para formar um sistema completo.

A arquitetura de microsserviços contrasta diretamente com a arquitetura monolítica. Para exemplificar, vamos considerar um sistema de e-commerce.

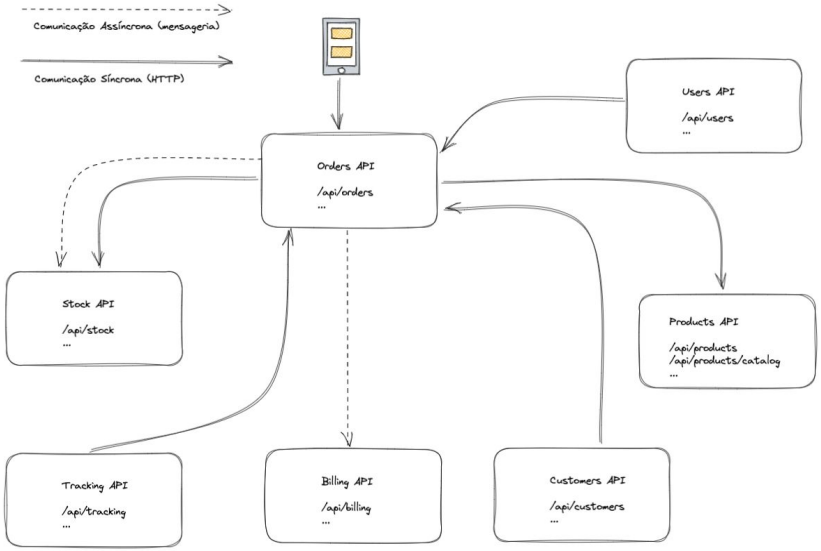
Nesse sistema, temos os seguintes requisitos:

- O sistema capaz prover um catálogo de produtos;
- Prover um gerenciamento de estoque;
- Realizar pedidos de venda;
- Realizar rastreio das compras;
- Cobranças no método de pagamento informado;
- Realizar gerenciamento de clientes;
- Gerenciar usuários administradores da plataforma.

Na arquitetura monolítica teríamos apenas uma única aplicação que implementaria todos esses requisitos e proveria acesso a ele por meio das rotas criadas:

Em contrapartida, a arquitetura de microsserviços orienta dividirmos o sistema em pequenos serviços (APIs) que se comunicam por meio de um protocolo de rede, seja HTTP ou Mensageria.

No cenário do e-commerce, teremos algumas APIs que implementam os requisitos do sistema separadamente:



Na versão de microsserviços, dividimos o sistema em **7 APIs** que se comunicam por meio de mensagens e requisições HTTP que são representadas por meio de setas tracejadas (comunicação assíncrona por meio de mensagens) e de linhas (comunicação síncrona por meio de requisição HTTP).
Vamos exemplificar o fluxo de criação de pedido de venda:

- O cliente faz a requisição para a API de pedidos (**Orders API**);
- A API de pedidos precisa validar se os itens do pedido estão em estoque e no catálogo de produtos. Por isso a API de pedidos faz as requisições para a API de estoque (**Stock API**) e produtos (**Products API**);
- Se os itens existirem em estoque e no catálogo, a API de pedidos publica uma mensagem na fila de cobrança que será consumida pela API de cobranças (**Billing API**);
- Uma vez que o pedido é cobrado pela API de cobrança, o serviço pedidos (**Orders API**) faz a atualização do estoque publicando uma mensagem na fila de atualização de estoque que será consumida pela API de estoque (**Stock API**)

Note que todo o fluxo de venda acontece de uma vez só para os usuários clientes da aplicação e eles não sabem se estão usando uma aplicação monolítica ou de microsserviços.

Características dos microsserviços

Uma arquitetura de microsserviços possui vantagens e desvantagens. As vantagens são:

- APIs mais simples e pequenas;
- Possibilidade de escalonar recursos computacionais específicos para cada serviço;
- Liberdade tecnológica, permitindo que cada serviço utilize uma linguagem e/ou framework diferente específico para sua necessidade, desde que se comunique por um protocolo padrão entre todos os serviços;
- Resiliência, impedindo que um sistema inteiro fique indisponível quando um serviço fica fora do ar;

Existem outras vantagens que são aplicáveis aos microsserviços. Mas agora vamos às desvantagens.

- É uma arquitetura mais complexa de se manter, visto que é necessário compreender os fluxo de comunicação e encontrar formas de documentá-los;

- Possui um custo financeiro muito maior, pois cada serviço terá seu próprio servidor e banco de dados. Além disso, para tornar cada serviço resiliente é necessário que eles tenham réplicas, o que pode aumentar exponencialmente o custo;
- Precisa de mais desenvolvedores para manter e evoluir os serviços, o que aumenta o custo com times de desenvolvimento.

Cuidados com microsserviços

Uma arquitetura de microsserviços não deve ser a primeira abordagem para criação de sistemas, visto que é uma arquitetura muito mais complexa de se manter e que exige vários times de desenvolvedores para cuidarem de uma ou mais aplicações do sistema. Sempre busque criar sistemas primeiro em uma arquitetura monolítica e com o tempo, conforme for surgindo a necessidade, quebrar as partes em serviços menores.

Para conhecer mais detalhes da arquitetura de microsserviços, recomendamos esse artigo (em inglês) do blog do **Martin Fowler**: https://martinfowler.com/articles/microservices.html

Também recomendamos esse site da Amazon AWS:

https://aws.amazon.com/pt/microservices/#:~:text=Microsservi%C3%A7os%20s%C3%A3o%20uma%20abordagem%20arquitet%C3%B4nica,pertencem%20a%20pequenas%20equipes%20autossuficientes.