**SECAO 1- INTRODUCAO**

**- ARQUITETURA MONOLITICA**

É uma aplicação de software em camadas que consiste em ter uma interface de usuário e a lógica de processamento dos dados em um **único programa**, geralmente feita a partir de uma única plataforma. Exemplo: Java, C#, PHP, Node.js, entre outros.

Frameworks comuns para aplicações monolíticas:

* Spring MVC
* .NET Framework
* Java Server Faces (JSF)
* Express.js (com template engines)
* Laravel
* Rails
* Django

Back-end e Front-end geralmente são o **mesmo projeto** e muitas vezes utilizam padrões como MVC (Model, View, Controller).

Geralmente o back-end é responsável por renderizar as páginas HTML (no caso de aplicação web) ou de realizar a interface gráfica (no caso de aplicações desktop).

O software por completo fica em apenas um local.

Builds e alterações no estado devem ser bem preparadas antes, pois ao parar a aplicação, todas as funcionalidades ficam indisponibilizadas até que a aplicação volte a operar.

Desenvolvimento geralmente mais rápido por manter back-end, banco de dados e front-end na mesma aplicação.

Geralmente aplicações monolíticas **não são modularizadas** (não é uma regra).

**- Arquitetura de microsserviços**

É uma abordagem para o desenvolvimento de software em que uma aplicação é **desmembrada** 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.

* Geralmente são desenvolvidos em formatos de API REST.
* Quando necessário, comunicam-se com outros microsserviços de maneira síncrona e assíncrona.
* São módulos **independentes**.
* Em uma única solução, pode-se ter diversas tecnologias e linguagens de programação distintas, desde que consigam se comunicar.
* Tem escalabilidade.
* Uma boa prática é que cada microsserviço possua sua própria base de dados para acesso.
* Quando há necessidade de deploy, a aplicação não é totalmente impedida de operar, apenas a funcionalidade ao qual o microsserviço responsável estará temporariamente indisponível.

Fácil de interagir com diversos sistemas e plataformas.

- Comparativa entre monolítico e microsserviços

|  |  |
| --- | --- |
| **Monolítico** | **Microsserviços** |
| Desenvolvimento mais rápido para implementar novas funcionalidades. | Conforme cresce o número de serviços, fica mais complexo para implementar novas funcionalidades. |
| Dificuldade com escalabilidade e necessidade de manutenções. | Escalável e possibilita manutenções diárias sem grandes impactos. |
| Toda a aplicação é desenvolvida em apenas um bloco contendo front-end, lógica de back-end e banco de dados. | O front-end geralmente é uma aplicação web própria que consome os dados dos microsserviços. Back-ends separados. |
| Grande acoplamento entre as dependências da aplicação. | Baixo acoplamento, apenas realizando comunicações síncronas e assíncronas. |
| Geralmente, os testes são da aplicação toda, e conforme cresce, fica cada vez mais difícil manter. | Facilidade em teste e monitoramento. Testes rodam muito mais rápido e com maior desempenho. |

**SECAO 2 – COMUNICAÇÃO SINCRONA E ASSINCRONA E RABBITMQ**

**- Comunicação síncrona**

Chamadas a APIs são síncronas, porém, o processo executado pela API pode ser assíncrono. Exemplo:

<https://sistema-exemplo.com.br/api/processar-vendas>

Você pode programar essa API para esperar para dar a resposta assim que todas as vendas forem processadas, ou você pode devolver uma resposta como “Processo iniciado.”, e deixar o processamento executando em background.

O processamento da API é assíncrono, pode terminar daqui a 10, 15, 20 minutos etc.

Mas a requisição HTTP não é, você faz a requisição e tem algum tipo de resposta, seja ela 200, 400, 500, etc.

**- Métodos HTTP**

Não há como falar de comunicação síncrona sem falar de métodos HTTP, pois eles serão nossa principal forma de comunicar nossas APIs de maneira síncrona!

Os métodos HTTP descrevem a ação a ser executada para determinado recurso. Eles definem se será um recurso para recuperar uma informação, salvar, atualizar, remover, alterar apenas alguns parâmetros, entre outras operações.

GET – recuperar um recurso

HEAD – recuperar um recurso porém sem o corpo da resposta

POST – salvar um recurso

PUT – atualizar um recurso completamente

DELETE – remover um recurso

OPTIONS – descrever a comunicação com um recurso de destino

PATCH – atualizar parcialmente um recurso

Métodos HTTP também podem ser:

**Seguro (Safe)** – não altera o estado do servidor, operações apenas de leitura: GET, HEAD, OPTIONS

**Idempotentes** – métodos que não surtem efeitos diferentes caso as requisições sejam feitas várias vezes de maneira idêntica: GET, HEAD, PUT, DELETE

Obs.: Todo método seguro é idempotente, mas nem todo método idempotente é seguro, exemplo, o PUT e o DELETE.

**Cacheable** – uma resposta é cacheada quando pode ser armazenada e recuperada posteriormente, evitando uma nova requisição e consumindo recursos do servidor.

**- Códigos de status HTTP**

Os status que mais utilizamos atualmente:

200 – OK (sucesso)

201 – CREATED (criado)

400 – BAD REQUEST (requisição inválida, ex: CPF inválido)

401 – UNAUTHORIZED (sem autenticação, não estando autenticado)

403 – FORBIDDEN (proibido, possui permissão, porém não pode visualizar o recurso, sem autorização)

404 – NOT FOUND (recurso não encontrado no servidor)

405 – METHOD NOT ALLOWED (método não permitido, ex: enviar método POST em uma requisição GET)

415 – UNSUPPORTED MEDIA TYPE (Mídia não suportada, geralmente um dado informado em um formato inválido)

429 – TOO MANY REQUESTS (muitas requisições feitas ao mesmo recurso, ex: site do Enem)

500 – INTERNAL SERVER ERROR (erro interno no servidor, algo deu errado no back-end)

502 – BAD GATEWAY (servidor intermediário que recebeu uma resposta inválida de outro serviço)

503 – SERVICE UNAVALIABLE (servidor não está pronto para lidar com a requisição, sobrecarregado ou em manutenção)

504 – GATEWAY TIMEOUT (servidor não recebe a resposta de um gateway)

- comunicação assíncrona, RabbitMQ e protocolo

É uma comunicação em que há um agente emissor e um receptor, com a diferença que o receptor não irá receber a mensagem quando o emissor emiti-la, sua recepção será atemporal, ou seja, não se sabe quando irá receber.

A melhor maneira de exemplificar é com uma fila de mensagens, ao qual um emissor apenas envia uma informação a uma fila, e algum outro agente receptor responsável por apenas escutar as mensagens recebidas dessa fila irá processar a mensagem sequencialmente conforme forem sendo recebidas.

Hoje o protocolo que melhor implementa as filas de mensagens é o protocolo AMQP – Advanced Message Queue Protocol, ou protocolo avançado de enfileiramento de mensagens.

O RabbitMQ tem alguns conceitos interessantes, como os tipos de Exchange, ou seja, regras de roteamento das mensagens. Nós temos:

* Direct Exchange
* Fanout Exchange
* Topic Exchange (esta que iremos utilizar)
* Headers Exchange

**- Tipos de Exchange e características de filas**

**a) Direct Exchange**

Uma fila é vinculada a uma Direct Exchange baseada em sua routing key.

A routing key é um atributo da mensagem utilizada pela Exchange escolhida para decidir para qual rota a mensagem será enviada.

* Uma fila é vinculada à Direct Exchange pela routing key (K).
* Quando uma nova mensagem com uma routing key (R) é enviada a uma Direct Exchange, a Exchange irá rotear apenas se R = K.

**b) Fanout Exchange**

A Fanout Exchange roteia mensagens para todas as filas vinculadas a ela e a routing key é **ignorada**. Se N filas são vinculadas a uma Fanout Exchange, quando uma nova mensagem é publicada, é feita uma cópia dessa mensagem, que é entregue a todas as N filas vinculadas à Fanout Exchange.

**c) Topic Exchange**

A Topic Exchange roteia mensagens para uma ou mais filas baseadas em uma correspondência entre a routing key e o padrão utilizado para vincular uma fila à Exchange.

É muito utilizada para implementar vários padrões de publisheres e subscribers.

Topic Exchanges são muito utilizados para roteamento multicast de mensagens.

Diagrama

Descrição gerada automaticamente

**d) Headers Exchange**

A Header Exchange é projetada para roteamento em vários atributos que são mais facilmente expressos como cabeçalhos (headers) da mensagem ao invés do uso de uma routing key.

A Header Exchange **ignora** a routing key.

Em vez disso, os atributos usados para roteamento são obtidos do atributo headers. Uma mensagem é considerada correspondente **se o valor do header for igual ao valor especificado no vínculo**.

Obs: E os **exchanges** possuem alguns atributos:

* Nome
* Durabilidade (se irão continuar existindo quando o broker reiniciar)
* Auto-delete (Exchange é deletado quando a sua última fila é desvinculada dele)
* Argumentos (são opcionais, para plugins e features específicas)

Obs: As **filas**, assim como os exchanges, possuem atributos, como:

* Nome
* Durabilidade (se irão sobreviver a um restart do broker)
* Exclusivas (usadas apenas para uma conexão, e removidas quando a conexão é fechada)
* Auto-delete (é removida quando seu último consumer se desvincula dela)

Argumentos (mesma lógica das exchanges)