

## *O que vamos construir?*

Aplicação baseada em microsserviços. Os serviços irão conversar entre si de forma assíncrona utilizando rabbitMQ

## *Para quem é recomendado o curso?*

Desenvolvedor que já possui o conhecimento básico de Java e Spring porém deseja se aprofundar em um nível mais técnico.

Desenvolvedor Junior/Pleno que trabalham com tecnologias mais legadas. Principalmente de órgãos públicos, aplicações monolíticas e etc...



## *Qual diferencial do curso?*

Existem muitos cursos que ensinam backend Java com Spring por aí. Mas neste curso quero abordar duas coisas que considero como diferencias.

- 1) Explicação técnica do que está acontecendo, não apenas sair digitando código e sim explicar o porquê.
- 2) Aplicação WEB.




# *Redes sociais*

[Linkedin](#)

[Youtube](#)



# *O que você irá aprender?*

- 1 - Conceitos básicos do Docker. Subir imagem da aplicação web localmente
  - 2 - Spring Framework + Principais anotações. O que é um @Bean?
  - 3 - Spring Data JPA - Ensinando melhor como funciona. O que é ORM?
  - 4 - Agendador de tarefas com @Scheduled
  - 5 - Spring Web + anotações. Como as nossas requisições chegam até nossa controller
  - 6 - MapStruct para converter nossos dtos em entidades e vice-versa
  - 7- Implementação do RabbitMQ no código
  - 8- Resiliência da aplicação. Dead Letter Queue
  - 9- Amazon SNS para enviar mensagem SMS (ms-notificação)
  - 10- Padrões de projeto
- 

# O que é o Docker?

**Cenário:** Considere que você tenha uma aplicação com dependências e configurações específicas para funcionar. Como Java 8 + MySql 5.7 por exemplo

**Problema:** As pessoas que desejam utilizar sua aplicação terão que realizar as mesmas configurações para executar localmente.

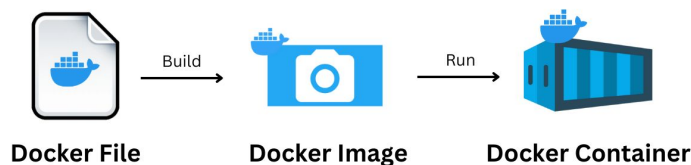
**Solução:** Com o docker, podemos criar uma imagem da nossa aplicação e subi-lá num container. Com isso, temos nosso aplicativo rodando localmente sem a necessidade de configurá-lo manualmente.



# O que é o Docker?

Através de um Dockerfile, podemos definir um conjunto de configurações e instruções para criar a imagem da nossa aplicação.

Essa imagem funciona como um espelho do nosso aplicativo, obtendo não apenas o código, mas também todas as configurações e dependências necessárias



# Subir imagem da nossa Aplicação WEB

**`docker run -d -p 80:80 --name proposta-web-container matheuspieropan/proposta-web`**

**`docker run -d`** significa que você quer inicializar um container no modo detach. Em resumo, seu container é executado em segundo plano e seu terminal fica liberado para uso.

**`-p 80:80`** significa que você quer que sua máquina local ao bater na porta 80 redirecione para a porta 80 do container. É como um mapeamento.

**`--name proposta-web-container`** é o nome do container que nós iremos subir. Se não passarmos esta flag o docker cria com nome aleatório

**`matheuspieropan/proposta-web`** é a imagem que nós queremos subir





# Configurando nosso banco de dados

*passar as seguintes configurações para o application.properties*

```
spring.datasource.url=jdbc:postgresql://localhost:5432/seubanco
```

```
spring.datasource.password=suasenha
```

```
spring.datasource.username=seuusuario
```



# Subindo nosso banco de dados através do Docker (OPCIONAL)

*`docker run --name postgres-container -d -e POSTGRES_PASSWORD=123 -e POSTGRES_DB=propostadb -p 5433:5432 postgres`*

*`POSTGRES_PASSWORD` = senha do banco de dados*

*`POSTGRES` = nome do banco de dados*

*Não passamos o usuário pois estamos utilizando o padrão que é postgres. Mas caso você queira mudar*

*`POSTGRES_USER` = nome do usuário*



# O que é o JPA?

Java Persistence API é a API do java que dita como os frameworks devem ser implementados para prover persistência num banco de dados relacional.

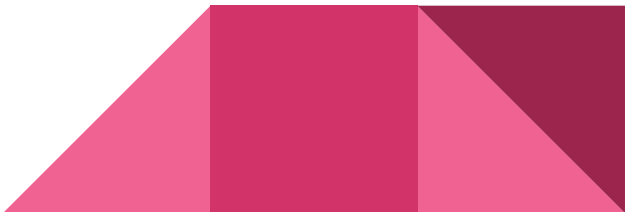
A Java Persistence API (JPA) foi introduzida pela primeira vez como parte da especificação Java EE 5.

JPA é uma especificação. Isso é, um conjunto de interfaces, anotações e padrões que definem um contrato para persistência do banco de dados.

## **ALGUMAS ANOTAÇÕES**

*@Entity* - Ao anotarmos nossa classe *Usuario* com esta anotação, estamos indicando que ela será mapeada para uma tabela no banco de dados

*@Table* - Em conjunto com anotação acima, usamos esta anotação para especificar o nome que nossa classe *Usuario* receberá no banco de dados



# O que NÃO é o JPA?

*O JPA não é o hibernate.*

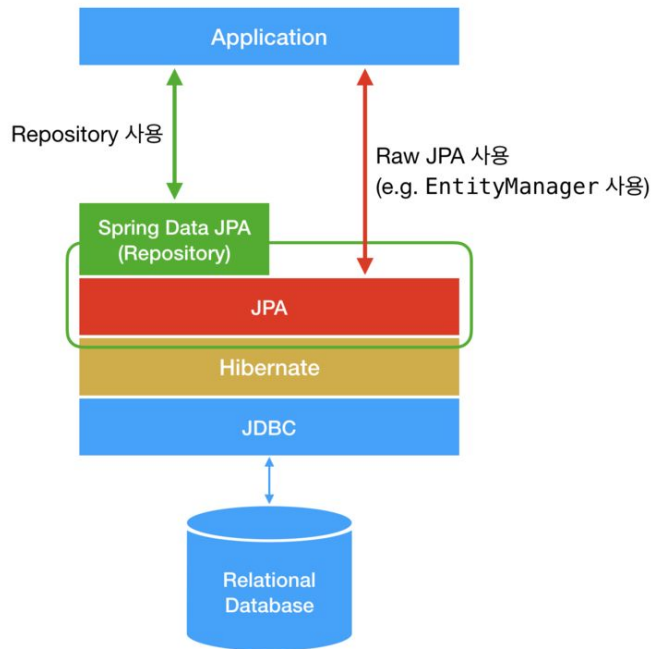
Muitas pessoas fazem confusão. Porém precisamos entender que JPA como foi dito anteriormente, é uma especificação e ela por si própria não funciona sozinha.

Java Persistence API precisa de uma implementação para funcionar, que neste caso é o **HIBERNATE**

Quando adicionamos a lib `spring-boot-starter-data-jpa` estamos utilizando o JPA com a implementação do hibernate + os recursos que o Spring Data nos traz como configuração da `@EntityManager` `@EntityManagerFactory`, repositórios, consultas derivadas etc..



# Desenhando para melhor compreensão



# Diferentes tipos de @GeneratedValue

`GenerationType.IDENTITY` -> Neste tipo, a geração do valor da chave primária é delegada ao banco de dados.

`GenerationType.SEQUENCE` -> Este tipo usa um objeto de sequência definido no banco de dados para gerar valores únicos

`GenerationType.AUTO` -> Provedor JPA (hibernate) escolhe qual usar

`GenerationType.NONE` -> Aplicação se torna responsável em fornecer a chave primária



# @OneToOne

Precisamos informar ao JPA como que será esse relacionamento entre Proposta e Usuario.

Nossa classe de referência será a Proposta. Ela representará uma tabela no banco de dados e terá a coluna id\_usuario para referenciar o usuário que aquela proposta pertence.

*Abaixo, as duas propriedades que serão importantes em nossa aplicação*

```
spring.jpa.hibernate.ddl-auto =update
```

```
spring.jpa.properties.hibernate.dialect =org.hibernate.dialect.PostgreSQLDialect
```



# *Criando nosso primeiro endpoint*

Vamos desenvolver uma API REST

Precisamos fornecer um endpoint que será responsável em receber os dados que virão do front-end e realizar a persistência no banco de dados.

## ***Afinal, o que é REST?***

Rest é uma arquitetura de software. É uma forma de projetar sistemas distribuídos. O rest segue padrões, regras e conceitos pré estabelecidos.

Tudo em rest é um recurso e cada recurso é identificado por uma URL

*{host}/api/usuarios : GET*





# Mais sobre *REST*

É comum utilizarmos REST com o protocolo HTTP e seus verbos.

**GET** <- Obtém algo

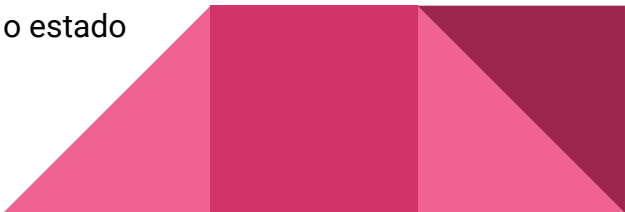
**POST** -> Persiste algo

**DELETE** -> Indica algo que será deletado

**PUT** -> Atualiza um registro

Os recursos geralmente são retornados em JSON.

E uma das principais características é que são Stateless. Cada requisição que nós fazemos é tratada como nova, o nosso servidor (nossa aplicação) não mantém o estado do cliente que solicitou.



# *O que isso significa?*

Isso significa que nossa aplicação não tem conhecimento prévio de quem somos. Diferente de uma aplicação tradicional, guardamos na sessão os dados que nos identifica. Como nosso nome, id, cpf e coisas do tipo.

Numa aplicação stateless precisamos fornecer esses dados em cada requisição.

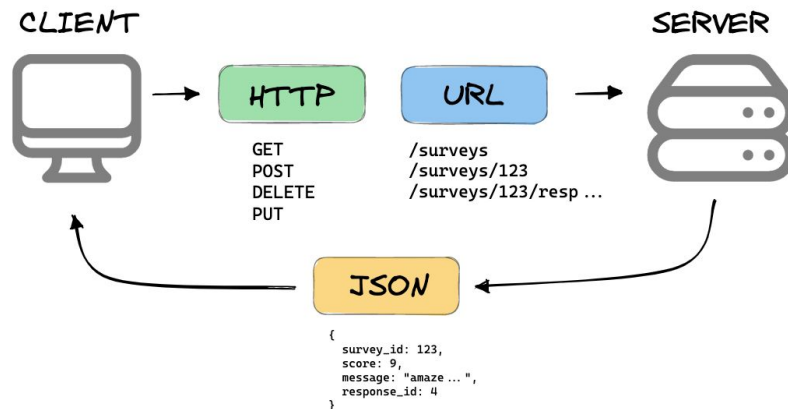
Ao cadastrarmos em um sistema e desejarmos posteriormente modificar nossos dados, precisamos informar o nosso ID/CPF ou alguma outra informação que nos identifica para nossa requisição.

Geralmente isso é feito através de tokens criptografados que recebemos no cabeçalho da requisição e através desses tokens obtemos o usuário que está solicitando o recurso e assim atualizamos os seus dados.



# Desenhando o modelo REST

WHAT IS A REST API?



mannhowie.com

## *Padrão DTO - O que significa?*

É uma boa prática no desenvolvimento de software utilizarmos o padrão DTO que significa Data Transfer Object

Com este padrão, mapeamentos exatamente aquilo que estamos recebendo/devolvendo na requisição.

Nossa entidade Proposta tem campos como aprovada, integrada, observação etc...

Campos esses que não recebemos no request da requisição, então por isso não colocamos no **@RequestBody** a entidade Proposta. Queremos evitar transacionar dados desnecessários/não utilizados



# Como o Spring Framework funciona?

O spring usa anotações para indicar que uma classe é gerenciada por ele.


Essas anotações fazem parte do pacote `org.springframework.stereotype`

Quando anotamos nossa classe com algum destes estereótipos, jogamos para o Spring a responsabilidade gerenciar a classe. Inclusive ele que fará a instância (new)

Para recuperarmos a instância que o próprio Spring fez, podemos utilizar a anotação ***@Autowired ou pelo construtor***



# *Entenda o código de resposta HTTP*

- **1xx (Informativo)** – o servidor recebeu a solicitação e a está processando.
  - **2xx (Confirmação)** – o servidor recebeu a solicitação e enviou de volta a resposta esperada.
  - **3xx (Redirecionamento)** – indica que algo mais precisa ser feito ou precisou ser feito para completar a solicitação.
  - **4xx (Erro do cliente)** – indica que a solicitação não pode ser concluída ou contém a sintaxe incorreta.
  - **5xx (Erro no servidor)** – o servidor falhou ao concluir a solicitação.
- 

# *Entendendo mais o CASCADE*


`CascadeType.PERSIST` - Esse tipo tem o objetivo de atualizar as filhas toda vez que a entidade pai estiver sendo persistida no banco de dados (criação)

Utilizamos ele pois ainda não tínhamos salvado o usuário no banco de dados.

Agora considere que já temos uma proposta salva mas por algum motivo queremos atualiza-lá. Ao utilizarmos o `Type ALL` ele vai replicar a cascata para o usuário também. Realizando operações que não queríamos e consumindo recurso computacional sem necessidade



# Entendendo o erro CORS

1. ***SIGNIFICADO: Compartilhamento de recursos com origens diferentes***
  2. ***É uma medida de segurança implementada pelos navegadores***
  3. ***Visa proteger os usuários contra ataques de scripts maliciosos que tentam roubar informações de outros sites***
  4. ***Ocorre quando tentamos fazer uma solicitação para um domínio diferente daquela página***
  5. ***O erro é contornado no backend quando informamos nos cabeçalhos o que nossa aplicação permitirá ou não***
- 




# Monolítico

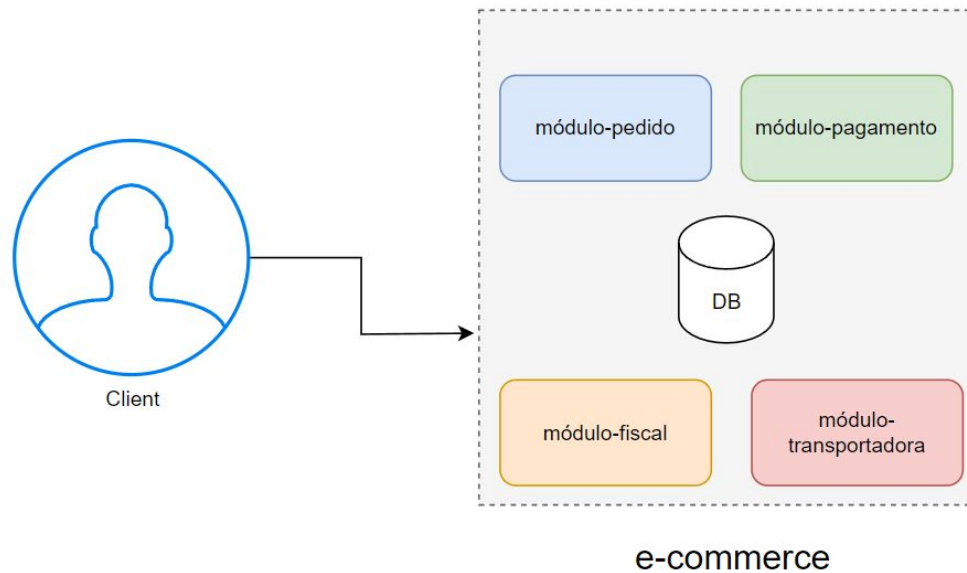
## VANTAGENS

1. *Menor tráfego de rede*
2. *Maior facilidade de depuração*
3. *Deploy único*
4. *Repositório único. Pode ser algo mais fácil principalmente para quem está iniciando no desenvolvimento*

## DESVANTAGENS

1. *Posso ter um commit que acaba prejudicando minha aplicação fazendo com que ela não permaneça no ar.*
  2. *Vários commits de equipes/squads diferentes podem influenciar no funcionamento da aplicação*
  3. *Maior dificuldade em adotar novas tecnologias*
- 

# Monolítico




# Microserviços

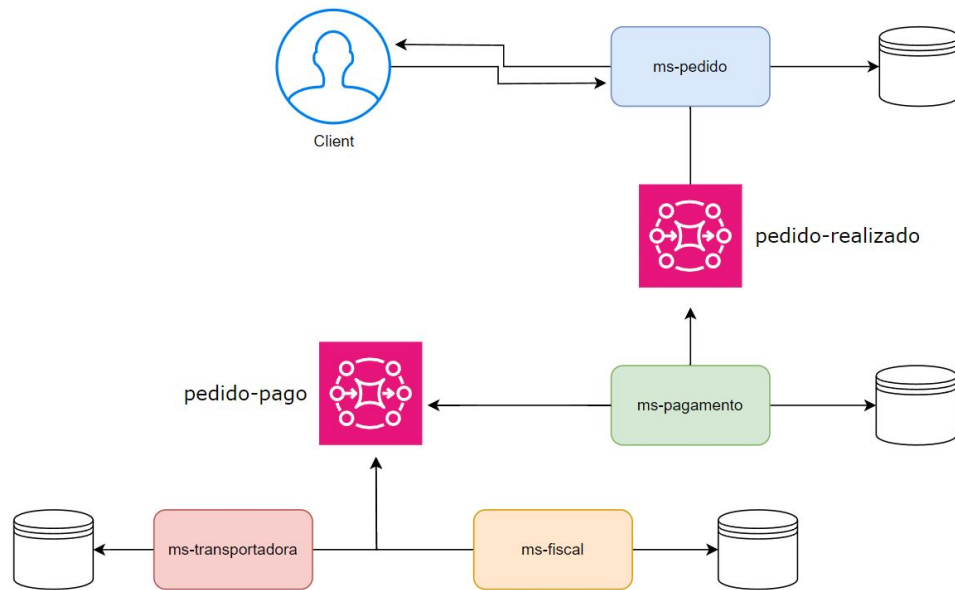
## VANTAGENS

1. *Melhor para se trabalhar quando se tem muitos profissionais atuando na aplicação*
2. *Se um apresentar algum problema/instabilidade os outros continuam funcionando normalmente*
3. *Maior facilidade em adotar escalabilidade horizontal*
4. *Mais fácil adotar novas tecnologias/seguir padrões de projeto - boas práticas*

## DESVANTAGENS

1. *Maior tráfego de rede*
  2. *Deploy se torna mais algo mais complexo uma vez que todos microserviços precisam estar 'casando' entre as versões*
  3. *Maior dificuldade em depuração*
- 

# Microserviços



# Explicando o RabbitMQ

1. *Ele opera de forma assíncrona*
2. *Age como um intermediário que recebe nossas mensagens*
3. *Notifica aos consumidores que uma nova mensagem chegou*
4. *Nenhuma mensagem é enviada diretamente para a fila e sim exchange*
5. *A exchange é responsável em encaminhar para nossa fila*
6. *Diferente do kafka. Meus consumidores concorrem entre si ao consumir de uma única fila*



# Subir imagem do RabbitMQ

**`docker run -d -p 5672:5672 -p 15672:15672 --name my-rabbit rabbitmq:3-management`**

**`docker run -d`** significa que você quer inicializar um container no modo detach. Em resumo, seu container é executado em segundo plano e seu terminal fica liberado para uso.

**`-p 5672:5672 -p 15672:15672`** significa que você quer que sua máquina local ao bater na porta 5672 redirecione para a porta 5672 do container. É como um mapeamento, o mesmo para 15672

**`--name my-rabbit`** é o nome do container que nós iremos subir. Se não passarmos esta flag o docker cria com nome aleatório

**`rabbitmq:3-management`** é a imagem que nós queremos subir



## *Atualizar o application.properties*

```
spring.rabbitmq.host=localhost
```


```
spring.rabbitmq.port=5672
```

```
spring.rabbitmq.username=guest
```

```
spring.rabbitmq.password=guest
```



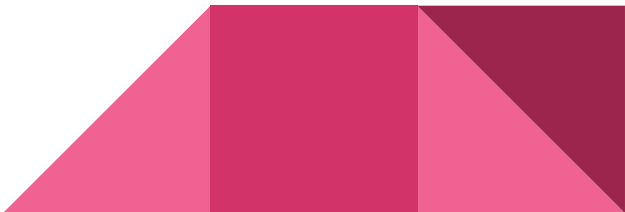
# Algoritmo da análise de crédito

1. *Vamos criar um código no qual cada condição concede X pontos.*
  2. *Trabalharemos com as classes randômicas do java, isto é, valores que serão gerados de forma aleatória.*
  3. *Se o valor mínimo de pontos for alcançado, proposta aceita, se não ela será negada.*
  4. *Algumas condições já limam a análise completa. Nome negativado é uma delas*
  5. *Vamos implementar o @Order para que o item anterior seja atendido.*
  6. *Padrão de projeto Strategy*
  7. *Consultas de serviços externos são simulados*
  8. *Use sua criatividade.*
  9. *Com o processamento feito, vamos publicar na fila de proposta-concluída.*
- 



# Dead Letter Queue

Uma fila de mensagens não entregues (DLQ) é um tipo especial de fila de mensagens que armazena temporariamente mensagens que um sistema de software não pode processar devido a erros

- 1. Erro ao consumir uma mensagem. Pois dentro da regra de negócio (código) aconteceu alguma exceção em tempo de execução, que são exceções não tratadas (RuntimeException). Considere que ao consumir uma mensagem, você recebe um determinado objeto e em certo ponto você acaba tomando um NullPointerException não esperado.*
  - 2. Uma fila foi configurado com um número máximo de mensagens e este número foi excedido.*
  - 3. Tamanho da mensagem excedido*
  - 4. Tempo máximo para mensagem ser consumida*
- 

# WebSocket

Um WebSocket é um protocolo de comunicação bidirecional que fornece uma conexão persistente e em tempo real entre um cliente e um servidor via a web

- 1. Tanto o cliente quanto o servidor podem enviar mensagens um para o outro a qualquer momento*
- 2. Uma vez estabelecida, a conexão WebSocket permanece aberta, permitindo a comunicação bidirecional eficiente entre o cliente e servidor.*
- 3. Isso é diferente do modelo de solicitação e resposta do HTTP, onde uma nova conexão é estabelecida para cada solicitação.*

