
ChoppCloud

# O que vamos ver?


1) Como criar e integrar 5 microservices;
2) Usar Spring-Boot;
3) Usar RabbitMQ
4) Usar MySQL;
5) Simular uma integração por API;
6) Testar o Circiut Break, tolerância a falhas, com Hystrix;
7) Usar o Consul.io;
8) Testar o Loadbalancer;





# O que não vamos ver?

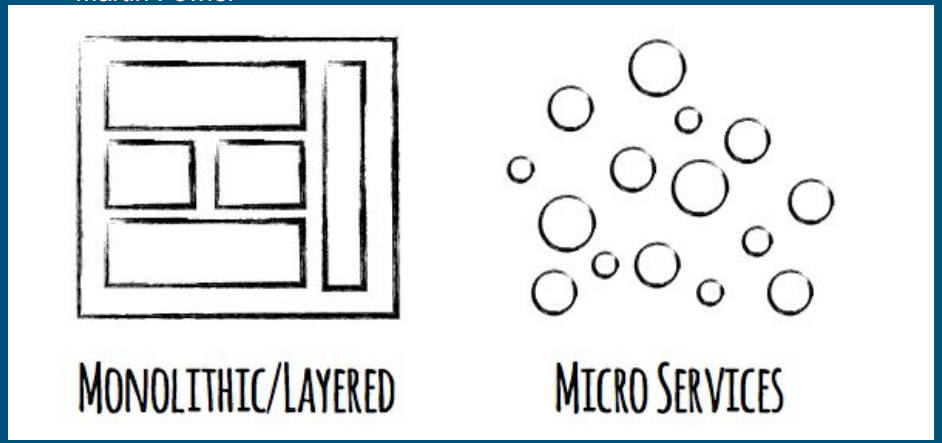
1) Testes. Testes unitários, integração, etc...
2) Arquitetura de código;

# O que é microservice?

A microservice architecture builds software as suites of collaborating services.

Uma arquitetura Microservice constrói software como suites de serviços de colaboração.

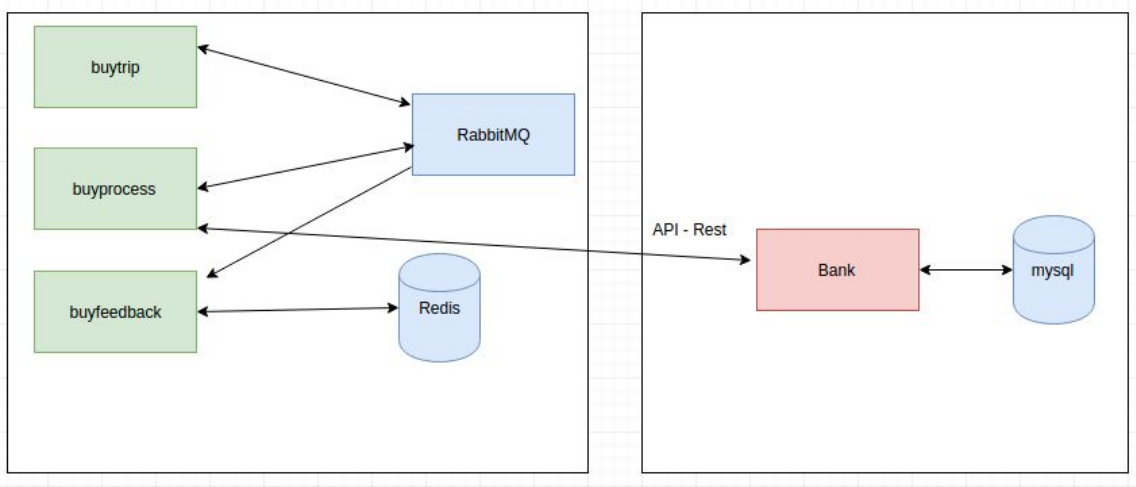
-- Martin Fowler




# Só existe a plataforma Spring?



# O que vamos desenvolver?




## Resumo do sistema

Sistema de compra de viagem.

1) Sistema deve disponibilizar uma entrada para ser executado uma compra de viagem;

2) O sistema deve aceitar o cartão Bank e integrar sua API para aprovação da compra;

3) O sistema deve possibilitar uma consulta para saber o status da compra;

# Bank

O sistema BANK vai representar um recurso externo ao nosso sistema onde será acessado via API.

# Padrão 1: Design for Data Separation

Livro “*Modern Java EE Design Patterns*”
“Considere um aplicativo tradicional monolítico que armazena dados em um único banco de dados relacional. Cada parte do aplicativo acessa os mesmos objetos de domínio e geralmente não há problemas em torno de transações ou separação de dados. ***A separação de dados é diferente com microservices***. Se dois ou mais serviços operam no mesmo armazenamento de dados, você terá problemas de consistência. Existem maneiras possíveis de contornar isso (por exemplo, transações), mas geralmente é considerado um ***antipattern***.”

Cada microservice deve usar seu próprio banco de dados.

# Mão na massa

1) Criar uma pasta, choppcloud, e criar o pom abaixo. (bank-1-pom.xml)

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/
maven-4.0.0.xsd">
        <modelVersion>4.0.0</modelVersion>

        <groupId>com.br.choppcloud</groupId>
        <artifactId>choppcloud-parent</artifactId>
        <version>0.0.0.-SNAPSHOT</version>
        <packaging>pom</packaging>

        <name>choppcloud-parent</name>

        <modules>
                <module>choppcloud-bank</module>
        </modules>
</project>
```

# Mão na massa

2) criar uma nova pasta, choppcloud-bank

3) criar o pom.xml - (bank-2-pom.xml)

4) criar as pastas
src/main/java
src/main/resources

5) importar no eclipse

6) criar pacote ***com.br.choppcloud.bank***

7) criar a classe ApplicationBank.java (bank-3-ApplicationBank.java)

# Mão na massa

8) Run As / Java Application

```java
package com.br.choppcloud.bank;

import org.springframework.boot.SpringApplication;

@SpringBootApplication
public class ApplicationBank {

    public static void main(String[] args) {
        SpringApplication.run(ApplicationBank.class, args);
    }

}
```

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v1.4.3.RELEASE)

2017-01-19 14:45:35.568  INFO 12258 --- [           main] com.br.choppcloud.bank.ApplicationBank   : Starting ApplicationBank on marcelo-desktop with PID
2017-01-19 14:45:35.568  INFO 12258 --- [           main] com.br.choppcloud.bank.ApplicationBank   : No active profile set, falling back to default profil
2017-01-19 14:45:35.568  INFO 12258 --- [           main] ationConfigEmbeddedWebApplicationContext : Refreshing org.springframework.boot.context.embedded.
2017-01-19 14:45:35.568  INFO 12258 --- [           main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat initialized with port(s): 8080 (http)
2017-01-19 14:45:35.568  INFO 12258 --- [           main] o.apache.catalina.core.StandardService   : Starting service Tomcat
2017-01-19 14:45:35.568  INFO 12258 --- [           main] org.apache.catalina.core.StandardEngine  : Starting Servlet Engine: Apache Tomcat/8.5.5
2017-01-19 14:45:35.681  INFO 12258 --- [ost-startStop-1] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicationContext
2017-01-19 14:45:35.681  INFO 12258 --- [ost-startStop-1] o.s.web.context.ContextLoader            : Root WebApplicationContext: initialization completed
2017-01-19 14:45:35.773  INFO 12258 --- [ost-startStop-1] o.s.b.w.servlet.ServletRegistrationBean  : Mapping servlet: 'dispatcherServlet' to [/]
2017-01-19 14:45:35.776  INFO 12258 --- [ost-startStop-1] o.s.b.w.servlet.FilterRegistrationBean   : Mapping filter: 'characterEncodingFilter' to: [/*]
2017-01-19 14:45:35.776  INFO 12258 --- [ost-startStop-1] o.s.b.w.servlet.FilterRegistrationBean   : Mapping filter: 'hiddenHttpMethodFilter' to: [/*]
2017-01-19 14:45:35.776  INFO 12258 --- [ost-startStop-1] o.s.b.w.servlet.FilterRegistrationBean   : Mapping filter: 'httpPutFormContentFilter' to: [/*]
2017-01-19 14:45:35.776  INFO 12258 --- [ost-startStop-1] o.s.b.w.servlet.FilterRegistrationBean   : Mapping filter: 'requestContextFilter' to: [/*]
2017-01-19 14:45:35.990  INFO 12258 --- [           main] s.w.s.m.m.a.RequestMappingHandlerAdapter : Looking for @ControllerAdvice: org.springframework.bo
2017-01-19 14:45:36.046  INFO 12258 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/error],produces=[text/html]}" onto public
2017-01-19 14:45:36.047  INFO 12258 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/error]}" onto public org.springframework.h
2017-01-19 14:45:36.075  INFO 12258 --- [           main] o.s.w.s.handler.SimpleUrlHandlerMapping  : Mapped URL path [/webjars/**] onto handler of type [c
2017-01-19 14:45:36.075  INFO 12258 --- [           main] o.s.w.s.handler.SimpleUrlHandlerMapping  : Mapped URL path [/**] onto handler of type [class org
2017-01-19 14:45:36.108  INFO 12258 --- [           main] o.s.w.s.handler.SimpleUrlHandlerMapping  : Mapped URL path [/**/favicon.ico] onto handler of typ
2017-01-19 14:45:36.275  INFO 12258 --- [           main] o.s.j.e.a.AnnotationMBeanExporter        : Registering beans for JMX exposure on startup
2017-01-19 14:45:36.313  INFO 12258 --- [           main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080 (http)
2017-01-19 14:45:36.318  INFO 12258 --- [           main] com.br.choppcloud.bank.ApplicationBank   : Started ApplicationBank in 2.23 seconds (JVM running
```

# Vamos agora criar o endpoint de pagamento

1) criar o pacote ***pagamento***;
2) criar a classe ***PagamentoJson.java***, que vai ser os dados que o endpoint vai receber. (bank-4-PagamentoJson.java)
3) criar a classe ***RetornoJson.java***, que vai ser o dados que o endpoint vai retornar. (bank-5-RetornoJson.java)
4) criar a classe ***PagamentoController.java***. Este vai ser nosso endpoint. (bank-6-PagamentoController.java)

```java
package com.br.choppcloud.bank.pagamento;

import javax.validation.Valid;
import javax.validation.constraints.NotNull;

import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class PagamentoController {

    @RequestMapping(path = "/pagamento", method = RequestMethod.POST)
    public ResponseEntity<RetornoJson> pagamento(
            @Valid @NotNull @RequestBody PagamentoJson pagamentoJson) {

        RetornoJson retorno = new RetornoJson();
        return new ResponseEntity<RetornoJson>(retorno, HttpStatus.OK);
    }
}
```

```
Looking for @ControllerAdvice: org.
Mapped "{[/pagamento],methods=[POST
Mapped "{[/error],produces=[text/ht
Mapped "{[/error]}" onto public org
Mapped URL path [/webjars/**] onto
Mapped URL path [/**] onto handler
```

Url http://localhost:8080/pagamento

Method POST

Query Parameters

Headers

Content-Type: applic...

Body

```
{
   "nroCartao": 13456,
   "codigoSegurancaCartao":35,
   "valorCompra":2545.55
}
```

# Vamos agora criar nossa regra de negócio

1) add arquivo application.properties; (bank-8-application.properties) alterar o key ***spring.datasource.url*** para um banco válido.
2) criar pacote com.br.choppcloud.bank.cartao;
3) criar classe Cartao; (bank-9-Cartao.java)
4) criar interface CartaoRepository; (bank-10-CartaoRepository.java)

```
public interface CartaoRepository extends Repository<Cartao, Long>{

    @Query("select count(obj.id) from Cartao obj where obj.codigoSegurancaCartao = ?1 and obj.nroCartao = ?2")
    Integer findCartaoValido(Integer codigoSegurancaCartao, Integer nroCartao);

    @Query("select count(obj.id) from Cartao obj where obj.codigoSegurancaCartao = ?1 and obj.nroCartao = ?2 and obj.valorCredito >= ?3")
    Integer isSaldoSuficiente(Integer codigoSegurancaCartao, Integer nroCartao, BigDecimal valorCompra);

    @Query("from Cartao obj where obj.codigoSegurancaCartao = ?1 and obj.nroCartao = ?2")
    Cartao findCartao(Integer codigoSegurancaCartao, Integer nroCartao);

    @Modifying
    @Query("update Cartao set valorCredito = (valorCredito - ?3) where codigoSegurancaCartao = ?1 and nroCartao = ?2 ")
    void atualizarSaldo(Integer codigoSegurancaCartao, Integer nroCartao, BigDecimal valorCompra);

}
```

# Vamos agora criar nossa regra de negócio

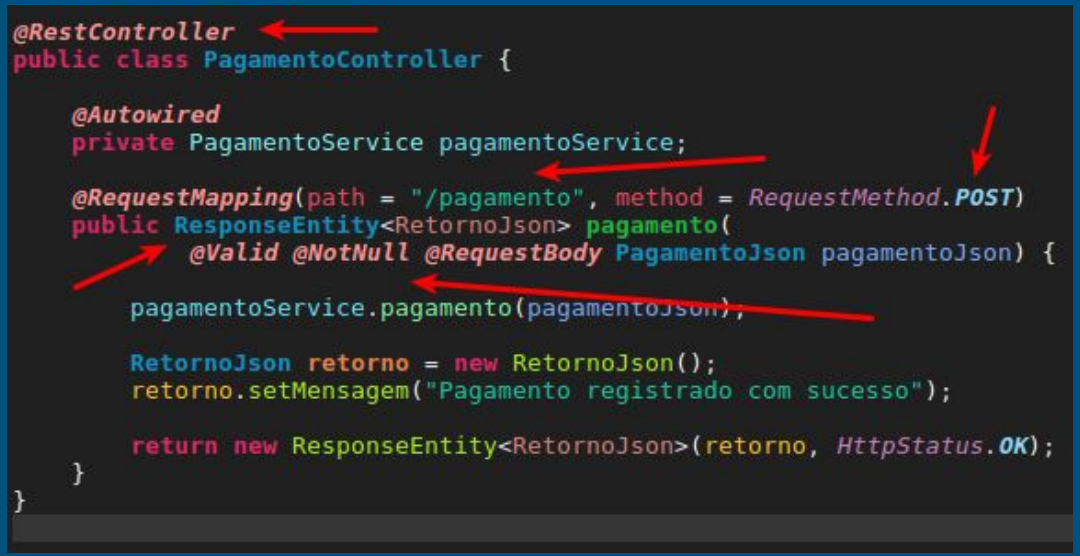
5) criar interface CartaoService; (bank-11-CartaoService.java)
6) criar classe CartaoServiceImpl; (bank-12-CartaoServiceImpl.java)

```
@Transactional
@Override
public void atualizarSaldo(Integer codigoSegurancaCartao,
        Integer nroCartao, BigDecimal valorCompra) {
    cartaoRepository.atualizarSaldo(codigoSegurancaCartao, nroCartao, valorCompra);
}
```

7) noe pacote com.br.choppcloud.bank.pagamento
8) criar classe Pagamento; (bank-13-Pagamento.java)

# Vamos agora criar nossa regra de negócio

9) alterar a classe PagamentoController; (bank-14-PagamentoController.java)



```
@RestController
public class PagamentoController {

    @Autowired
    private PagamentoService pagamentoService;

    @RequestMapping(path = "/pagamento", method = RequestMethod.POST)
    public ResponseEntity<RetornoJson> pagamento(
            @Valid @NotNull @RequestBody PagamentoJson pagamentoJson) {

        pagamentoService.pagamento(pagamentoJson);

        RetornoJson retorno = new RetornoJson();
        retorno.setMensagem("Pagamento registrado com sucesso");

        return new ResponseEntity<RetornoJson>(retorno, HttpStatus.OK);
    }
}
```

# Vamos agora criar nossa regra de negócio

10. criar classe PagamentoException; (bank-15-PagamentoException.java)
11. criar interface PagamentoRepository; (bank-16-PagamentoRepository.java)
12. criar interface PagamentoService; (bank-17-PagamentoService.java)
13. criar classe PagamentoServiceImpl; (bank-18-PagamentoServiceImpl.java)
14. alterar classe PagamentoJson; (bank-19-PagamentoJson.java)
15. alterar classe RetornoJson; (bank-20-RetornoJson.java)

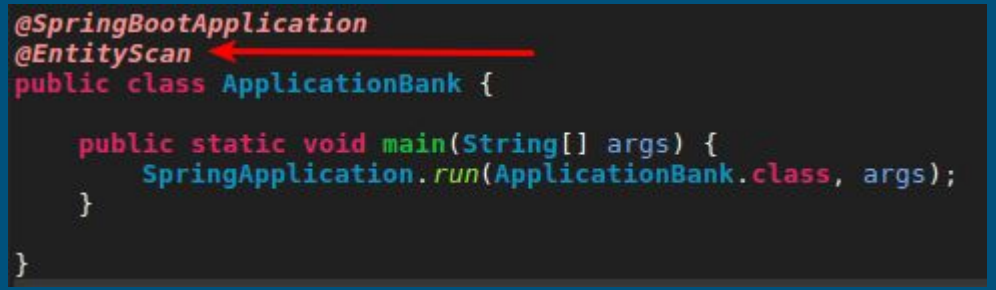
# Vamos agora criar nossa regra de negócio

16. criar no pacote com.br.choppcloud.bank, a classe ExceptionHandlerController; (bank-21-ExceptionHandlerController.java)

```java
@ControllerAdvice
public class ExceptionHandlerController {

    @ExceptionHandler(PagamentoException.class)
    @ResponseStatus(value= HttpStatus.BAD_REQUEST)
    @ResponseBody
    public RetornoJson process(RuntimeException ex) {
        return new RetornoJson(ex.getMessage());
    }

}
```

# Vamos agora criar nossa regra de negócio

17. altere a classe ApplicationBank para ficar igual (bank-22-ApplicationBank.java)




18. Na pasta resources, add o arquivo inserts_basic.sql. (bank-23-inserts_basic.sql)

# Banco MySQL

Se tem instalado o MYSQL na própria máquina, basta criar um DB com o mesmo nome que está no application.properties.

spring.datasource.url=jdbc:mysql://localhost/banco

create database banco;

Caso não tenha o banco, coloque o {IP} e me passe o nome do banco que vai ser usado.

# Hora de brindar

1) ***@EntityScan***
   Configura os pacotes base usados pela auto-configuração ao verificar classes de entidade.
2) ***Repository***
   Não precisa de implementação, pode ser executado HQL e SQL Native.

```java
public interface CartaoRepository extends Repository<Cartao, Long>{

    @Query("select count(obj.id) from Cartao obj where obj.codigoSegurancaCartao = ?1 and obj.nroCartao = ?2")
    Integer findCartaoValido(Integer codigoSegurancaCartao, Integer nroCartao);

    @Query("select count(obj.id) from Cartao obj where obj.codigoSegurancaCartao = ?1 and obj.nroCartao = ?2 and obj.valorCredito >= ?3")
    Integer isSaldoSuficiente(Integer codigoSegurancaCartao, Integer nroCartao, BigDecimal valorCompra);

    @Query("from Cartao obj where obj.codigoSegurancaCartao = ?1 and obj.nroCartao = ?2")
    Cartao findCartao(Integer codigoSegurancaCartao, Integer nroCartao);

    @Modifying
    @Query("update Cartao set valorCredito = (valorCredito - ?3) where codigoSegurancaCartao = ?1 and nroCartao = ?2 ")
    void atualizarSaldo(Integer codigoSegurancaCartao, Integer nroCartao, BigDecimal valorCompra);

}
```

# Hora de brindar

3) ***@Transactional***

No Service, usar a anotação @Transactional para os métodos que forem alterar dados no banco.

```
@Service
public class CartaoServiceImpl implements CartaoService{

    @Autowired
    private CartaoRepository cartaoRepository;

    @Override
    public boolean isValido(Integer codigoSegurancaCartao, Integer nroCartao) {
        return cartaoRepository.findCartaoValido(codigoSegurancaCartao, nroCartao) > 0;
    }

    @Override
    public boolean isSaldoSuficiente(Integer codigoSegurancaCartao,
            Integer nroCartao, BigDecimal valorCompra) {
        return cartaoRepository.isSaldoSuficiente(codigoSegurancaCartao, nroCartao, valorCompra) > 0;
    }

    @Override
    public Cartao getCartao(Integer codigoSegurancaCartao, Integer nroCartao) {
        return cartaoRepository.findCartao(codigoSegurancaCartao, nroCartao);
    }

    @Transactional
    @Override
    public void atualizarSaldo(Integer codigoSegurancaCartao,
            Integer nroCartao, BigDecimal valorCompra) {
        cartaoRepository.atualizarSaldo(codigoSegurancaCartao, nroCartao, valorCompra);
    }

}
```

# Hora de brindar

4) ***Handler***

Pode ser criado vários @ControllerAdvice, para interceptar as exceptions.

```
@ControllerAdvice
public class ExceptionHandlerController {

    @ExceptionHandler(PagamentoException.class)
    @ResponseStatus(value= HttpStatus.BAD_REQUEST)
    @ResponseBody
    public RetornoJson process(RuntimeException ex) {
        return new RetornoJson(ex.getMessage());
    }

}
```

# Testar

1) Suba a aplicação;
2) faz uma requisição POST para o link
   http://localhost:8090/pagamento
   header: Content-Type=application/json
   body: bank-24-json-teste.txt

   ***Retorno esperado:***
   {"mensagem":"Cartão inválido."}

# Comprar Passagem

O sistema ***Buytrip*** vai ser o sistema que vai iniciar a compra da passagem.

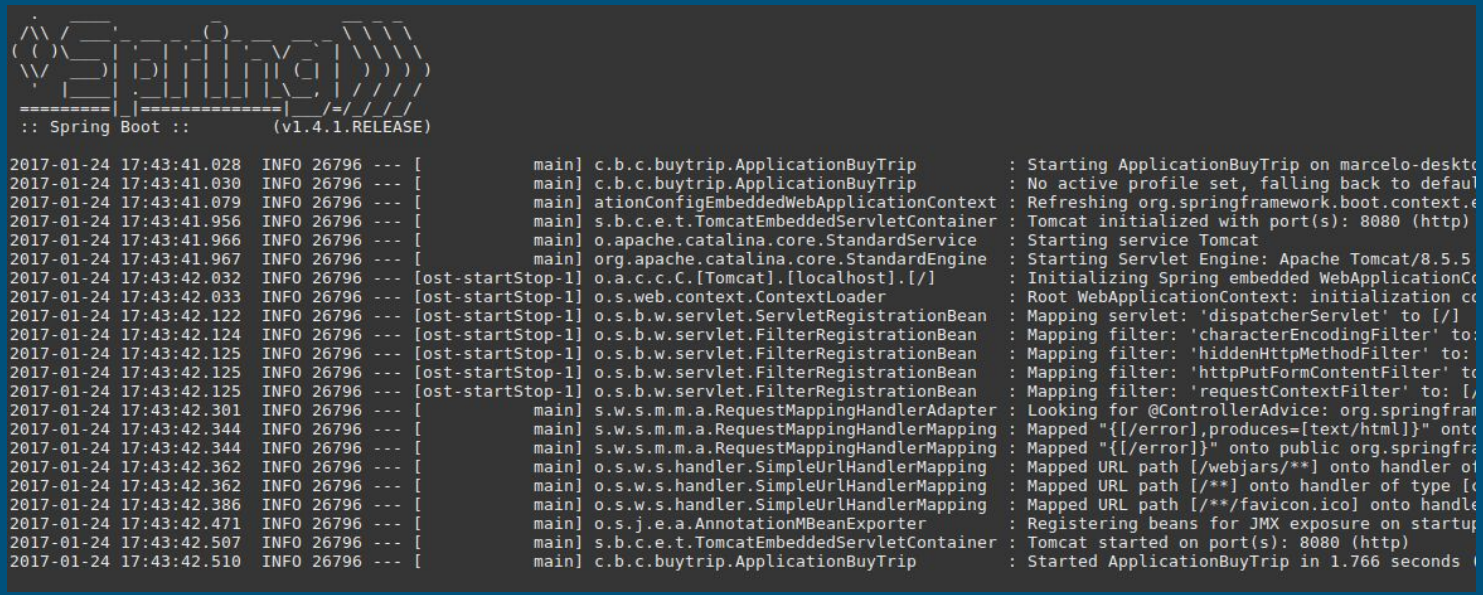
Vamos criar um endpoint para ser registrado o pedido da compra. Vai receber o código da passagem, os dados do cartão e o valor da passagem e devolver um chave para possíveis consultas posteriormente do status da compra.

# Vamos começar

1) criar uma pasta, choppcloud-buytrip;
2) criar o pom.xml; (buytrip-1-pom.xml)
3) criar as pastas
   src/main/java
   src/main/resources
4) add <module> no pom do projeto pai;
   <module>choppcloud-buytrip</module>
5) reimportar o projeto;
6) criar pacote com.br.choppcloud.buytrip
7) criar classe ApplicationBuyTrip; (buytrip-2-ApplicationBuyTrip.java)

# Inicie a aplicação

Execute o ApplicationBuyTrip para ver se o servidor vai subir corretamente.



# Vamos agora criar o endpoint

1) criar o pacote com.br.choppcloud.buytrip.compra;
2) criar classe CompraJson; (buytrip-3-CompraJson.java)
3) criar classe CompraChaveJson; (buytrip-4-CompraChaveJson.java)
4) criar classe RetornoJson; (buytrip-5-RetornoJson.java)
5) criar classe CompraController; (buytrip-6-CompraController.java)

```java
@RestController
public class CompraController {

    @RequestMapping(path = "/", method = RequestMethod.POST)
    public ResponseEntity<RetornoJson> pagamento(
            @Valid @NotNull @RequestBody CompraJson compraJson) {

        RetornoJson retorno = new RetornoJson();
        retorno.setMensagem("Compra registrada com sucesso. Aguarda a confirmação do pagamento.");

        return new ResponseEntity<RetornoJson>(retorno, HttpStatus.OK);
    }

}
```

# Para testar endpoint

Fazer uma requisição POST para o link
http://localhost:8080/
header: Content-Type=application/json
body: (json-example.txt)

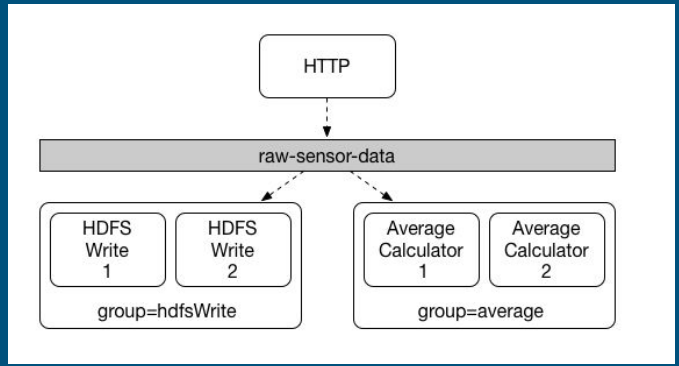
***Retorno esperado:***
{"mensagem":"Compra registrada com sucesso. Aguarda a confirmação do pagamento.","chavePesquisa":null}

# Stream Events

Nosso objetivo é antes de processar a compra, adicionar a informação em alguma fila, para que um outro sistema possa consumir a fila.

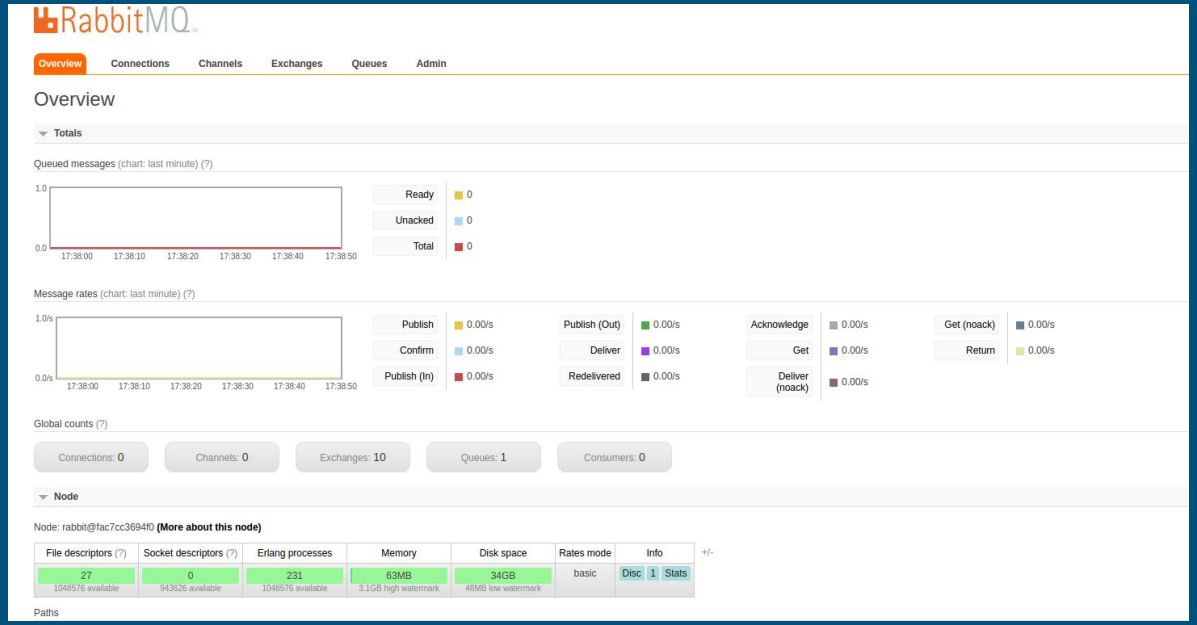
O Buytrip somente vai receber a compra, gerar um código, registrar a compra em uma fila e devolver uma mensagem para o cliente.

Para isso vamos usar o conceito de Stream Events. É um pouco diferente que uma fila simples. A fila por eventos, garante que se escalarmos o sistema que lê a fila, a informação vai chegar em somente 1 sistema.



# Stream Events

Vamos usar como servidor o RabbitMQ.



# Mão na massa

1) Adicionar as chaves no application.properties

```
spring.rabbitmq.host=IP
spring.rabbitmq.username=user
spring.rabbitmq.password=senha
spring.cloud.stream.bindings.input.destination=compra
spring.cloud.stream.bindings.input.group=buyprocess
fila.saida=fila-compras-aguardando
```

# Mão na massa

2) Quem desejar rodar o RabbitMQ na própria máquina, usar o (buytrip-7-docker-compose.yml)

   http://localhost:8881/#/

3) Vamos alterar a classe CompraController para que ela fique igual a (buytrip-8-CompraController.java)
4) vamos criar a fila “fila-compras-aguardando” no RabbitMQ.

# Mão na massa

```java
@RestController
public class CompraController {

    @Autowired
    private RabbitTemplate rabbitTemplate;

    @Value("${fila.saida}")
    private String nomeFila;

    @RequestMapping(path = "/", method = RequestMethod.POST)
    public ResponseEntity<RetornoJson> pagamento(
            @Valid @NotNull @RequestBody CompraJson compraJson) throws Exception  {

        CompraChaveJson compraChaveJson = new CompraChaveJson();
        compraChaveJson.setCompraJson(compraJson);
        compraChaveJson.setChave(UUID.randomUUID().toString());

        String json = new String(new JsonMessageConverter().toMessage(compraChaveJson, null).getBody(), "UTF-8");

        rabbitTemplate.convertAndSend(nomeFila, json);

        RetornoJson retorno = new RetornoJson();
        retorno.setMensagem("Compra registrada com sucesso. Aguarda a confirmação do pagamento.");
        retorno.setChavePesquisa(compraChaveJson.getChave());

        return new ResponseEntity<RetornoJson>(retorno, HttpStatus.OK);
    }

}
```

# Hora de brindar

1) ***RabbitTemplate***

   Injetar o RabbitTemplate para poder enviar a informação para a fila.
   rabbitTemplate.convertAndSend(nomeFila, json);

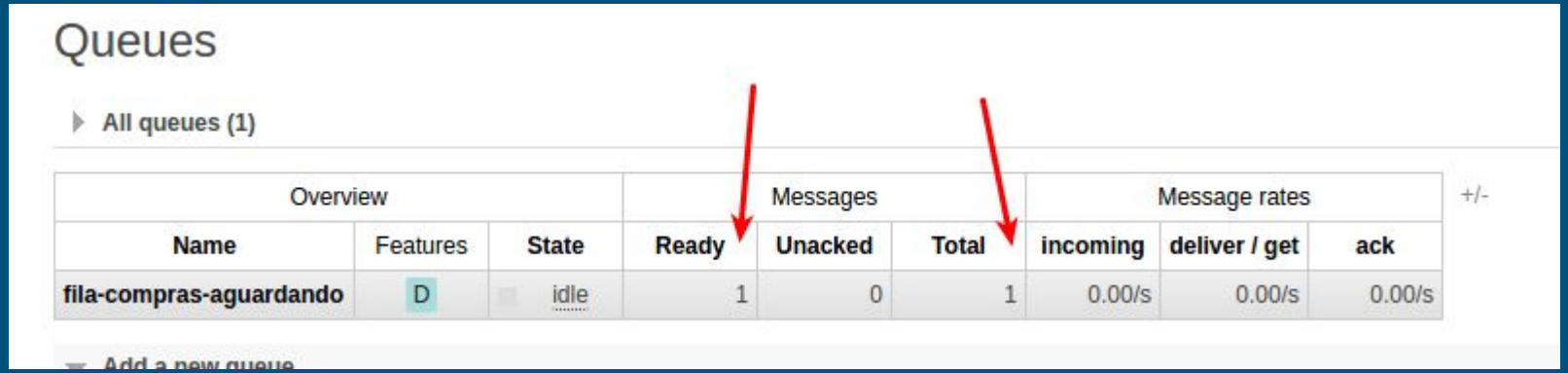
```java
@RestController
public class CompraController {

    @Autowired
    private RabbitTemplate rabbitTemplate;

    @Value("${fila.saida}")
    private String nomeFila;

    @RequestMapping(path = "/", method = RequestMethod.POST)
    public ResponseEntity<RetornoJson> pagamento(
            @Valid @NotNull @RequestBody CompraJson compraJson) throws Exception  {

        CompraChaveJson compraChaveJson = new CompraChaveJson();
        compraChaveJson.setCompraJson(compraJson);
        compraChaveJson.setChave(UUID.randomUUID().toString());

        String json = new String(new JsonMessageConverter().toMessage(compraChaveJson, null).getBody(), "UTF-8");

        rabbitTemplate.convertAndSend(nomeFila, json);

        RetornoJson retorno = new RetornoJson();
        retorno.setMensagem("Compra registrada com sucesso. Aguarda a confirmação do pagamento.");
        retorno.setChavePesquisa(compraChaveJson.getChave());

        return new ResponseEntity<RetornoJson>(retorno, HttpStatus.OK);
    }

}
```

# Para testar endpoint

Fazer uma requisição POST para o link
http://localhost:8080/
header: Content-Type=application/json
body: (json-example.txt)

***Retorno esperado:***
{"mensagem":"Compra registrada com sucesso. Aguarda a confirmação do pagamento.","chavePesquisa":"33afc181-288e-45a5-b7f0-a63ed7932e77"}

# Para testar endpoint



Queues

All queues (1)

| Overview | | | Messages | | | Message rates | | | +/- |
|---|---|---|---|---|---|---|---|---|---|
| Name | Features | State | Ready | Unacked | Total | incoming | deliver / get | ack | |
| fila-compras-aguardando | D | idle | 1 | 0 | 1 | 0.00/s | 0.00/s | 0.00/s | |

Add a new queue

# Processar Pagamento

O sistema ***Buyprocess*** vai ser nosso sistema que vai processar o pagamento da passagem.

Este sistema vai ler a fila onde estão as compras e vai ter uma integração com o sistema ***Bank***, para aprovar o pagamento. Após retorno no banco, ele vai adicionar em um nova fila, das passagem já processadas.

***Requisito não funcional***

Caso o sistema Bank esteja fora, o ***Buyprocess*** não pode perder a informação, deve posteriormente re-processar a compra.

# Padrão 2: Design for Failure/Tolerância a Falha

***Tolerância a falhas*** é a propriedade que permite que sistemas continuem a operar adequadamente mesmo após falhas em alguns de seus componentes.

A meta para tudo que devemos projetar em torno de tolerância de falha é minimizar a intervenção humana. Implementar rotinas de falha automática tem que ser parte de cada chamada de serviço que está acontecendo.

# Vamos começar

1) criar uma pasta, choppcloud-buyprocess;
2) criar o pom.xml; (buyprocess-1-pom.xml)
3) criar as pastas
   src/main/java
   src/main/resources
4) add <module> no pom do projeto pai;
   <module>choppcloud-buyprocess</module>
5) reimportar o projeto;
6) criar pacote com.br.choppcloud.buyprocess
7) criar classe ApplicationBuyProcess;
   (buyprocess-2-ApplicationBuyProcess.java)

# Inicie a aplicação

Execute o ApplicationBuyProcess para ver se o servidor vai subir corretamente.

```
in] o.s.i.monitor.IntegrationMBeanExporter   : Located managed bean 'org.springframework.integration:type=
in] o.s.i.monitor.IntegrationMBeanExporter   : Located managed bean 'org.springframework.integration:type=
in] o.s.c.support.DefaultLifecycleProcessor  : Starting beans in phase -2147482648
in] o.s.c.support.DefaultLifecycleProcessor  : Starting beans in phase 0
in] o.s.i.endpoint.EventDrivenConsumer       : Adding {logging-channel-adapter:_org.springframework.integra
in] o.s.i.channel.PublishSubscribeChannel    : Channel 'application:8081.errorChannel' has 1 subscriber(s)
in] o.s.i.endpoint.EventDrivenConsumer       : started _org.springframework.integration.errorLogger
in] o.s.c.support.DefaultLifecycleProcessor  : Starting beans in phase 2147483647
-1] o.s.a.r.c.CachingConnectionFactory        : Created new connection: SimpleConnection@38ccb80f [delegate=
in] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8081 (http)
in] c.b.c.buyprocess.ApplicationBuyProcess   : Started ApplicationBuyProcess in 4.816 seconds (JVM running
```

# Mão na massa

1) Adicionar as chaves no application.properties

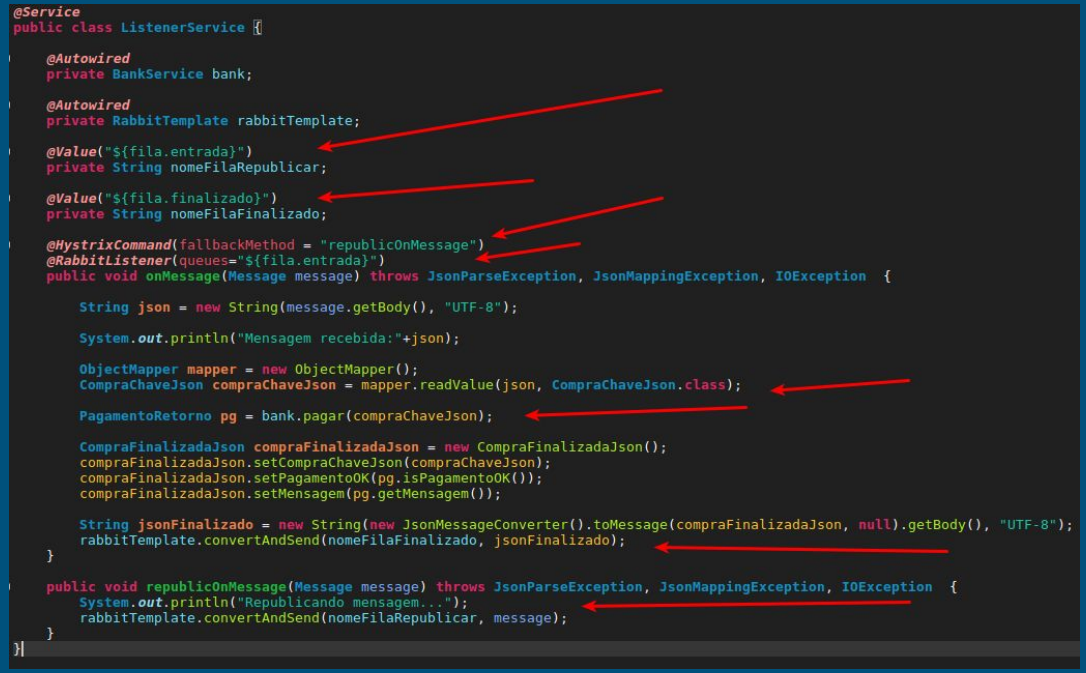
```
server.port=8081
spring.rabbitmq.username=user
spring.rabbitmq.password=senha
spring.cloud.stream.bindings.input.destination=compra
spring.cloud.stream.bindings.input.group=buyprocess
fila.entrada=fila-compras-aguardando
fila.finalizado=fila-compras-finalizado
bank.link=http://localhost:8090/pagamento
```

# Mão na massa

2) criar a pasta com.br.choppcloud.buyprocess.processar
3) criar o classe CompraJson (buyprocess-3-CompraJson.java)
4) criar a classe CompraChaveJson (buyprocess-4-CompraChaveJson.java)
5) criar a classe CompraFinalizadaJson (buyprocess-5-CompraFinalizadaJson.java)

# Mão na massa

6) criar a classe ListenerService (buyprocess-6-ListenerService.java)



```java
@Service
public class ListenerService {

    @Autowired
    private BankService bank;

    @Autowired
    private RabbitTemplate rabbitTemplate;

    @Value("${fila.entrada}")
    private String nomeFilaRepublicar;

    @Value("${fila.finalizado}")
    private String nomeFilaFinalizado;

    @HystrixCommand(fallbackMethod = "republicOnMessage")
    @RabbitListener(queues="${fila.entrada}")
    public void onMessage(Message message) throws JsonParseException, JsonMappingException, IOException  {

        String json = new String(message.getBody(), "UTF-8");

        System.out.println("Mensagem recebida:"+json);

        ObjectMapper mapper = new ObjectMapper();
        CompraChaveJson compraChaveJson = mapper.readValue(json, CompraChaveJson.class);

        PagamentoRetorno pg = bank.pagar(compraChaveJson);

        CompraFinalizadaJson compraFinalizadaJson = new CompraFinalizadaJson();
        compraFinalizadaJson.setCompraChaveJson(compraChaveJson);
        compraFinalizadaJson.setPagamentoOK(pg.isPagamentoOK());
        compraFinalizadaJson.setMensagem(pg.getMensagem());

        String jsonFinalizado = new String(new JsonMessageConverter().toMessage(compraFinalizadaJson, null).getBody(), "UTF-8");
        rabbitTemplate.convertAndSend(nomeFilaFinalizado, jsonFinalizado);
    }

    public void republicOnMessage(Message message) throws JsonParseException, JsonMappingException, IOException  {
        System.out.println("Republicando mensagem...");
        rabbitTemplate.convertAndSend(nomeFilaRepublicar, message);
    }
}
```

# Mão na massa

7) criar o pacote com.br.choppcloud.buyprocess.bank
8) criar a classe BankRetornoJson (buyprocess-7-BankRetornoJson.java)
9) criar a classe PagamentoJson (buyprocess-8-PagamentoJson.java)
10) criar a classe PagamentoRetorno (buyprocess-9-PagamentoRetorno.java)

# Mão na massa

11) criar a classe BankService (buyprocess-10-BankService.java)

```java
@Service
public class BankService {

    @Value("${bank.link}")
    private String link;

    @Autowired
    private RabbitTemplate rabbitTemplate;

    public PagamentoRetorno pagar(CompraChaveJson compraChaveJson) throws IOException {

        PagamentoJson json = new PagamentoJson();
        json.setNroCartao(compraChaveJson.getCompraJson().getNroCartao());
        json.setCodigoSegurancaCartao(compraChaveJson.getCompraJson().getCodigoSegurancaCartao());
        json.setValorCompra(compraChaveJson.getCompraJson().getValorPassagem());

        RestTemplate restTemplate = new RestTemplate();

        HttpHeaders headers = new HttpHeaders();
        headers.setContentType(MediaType.APPLICATION_JSON);
        HttpEntity<PagamentoJson> entity = new HttpEntity<PagamentoJson>(json, headers);

        try {
            ResponseEntity<BankRetornoJson> bankRetorno = restTemplate.exchange(link, HttpMethod.POST, entity, BankRetornoJson.class);
            return new PagamentoRetorno(bankRetorno.getBody().getMensagem(), true);
        }catch(HttpClientErrorException ex){
            if( ex.getStatusCode() == HttpStatus.BAD_REQUEST ) {
                ObjectMapper mapper = new ObjectMapper();
                BankRetornoJson obj = mapper.readValue(ex.getResponseBodyAsString(), BankRetornoJson.class);
                return new PagamentoRetorno(obj.getMensagem(), false);
            }
            throw ex;
        }catch (RuntimeException ex) {
            throw ex;
        }

    }
}
```

12) criar a fila no RabbitMQ: fila-compras-finalizado

# Hora de brindar

1) ***Hystrix***
   Adiciona as dependências para podermos ativar o circuit breaker.
   @EnableCircuitBreaker

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-hystrix</artifactId>
</dependency>
```

# Hora de brindar

2) O Spring Boot Actuator inclui vários recursos adicionais para ajudá-lo a monitorar e gerenciar seu aplicativo quando for empurrado para a produção. Você pode optar por gerenciar e monitorar seu aplicativo usando pontos de extremidade HTTP, com JMX ou mesmo por shell remoto (SSH ou Telnet). A auditoria, a saúde e a coleta de métricas podem ser aplicadas automaticamente à sua aplicação.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

# Hora de brindar

3) ***@HystrixCommand***
Anotação usada para marcarmos qual o método vamos executar o circuit break. Em FallbackMethod, é adicionado o método que vai ser executado em caso de exception não tratada.

```
@HystrixCommand(fallbackMethod = "republicOnMessage")
@RabbitListener(queues="${fila.entrada}")
public void onMessage(Message message) throws JsonParseException, JsonMappingException, IOException  {
```

```
public void republicOnMessage(Message message) throws JsonParseException, Jsc
    System.out.println("Republicando mensagem...");
    rabbitTemplate.convertAndSend(nomeFilaRepublicar, message);
}
```

# Hora de brindar

4) ***RestTemplate***

   Use o RestTemplate para executar chamada REST.

```
RestTemplate restTemplate = new RestTemplate();

HttpHeaders headers = new HttpHeaders();
headers.setContentType(MediaType.APPLICATION_JSON);
HttpEntity<PagamentoJson> entity = new HttpEntity<PagamentoJson>(json, headers);

try {
    ResponseEntity<BankRetornoJson> bankRetorno = restTemplate.exchange(link, HttpMethod.POST, entity, BankRetornoJson.class);
    return new PagamentoRetorno(bankRetorno.getBody().getMensagem(), true);
}catch(HttpClientErrorException ex){
    if( ex.getStatusCode() == HttpStatus.BAD_REQUEST ) {
        ObjectMapper mapper = new ObjectMapper();
        BankRetornoJson obj = mapper.readValue(ex.getResponseBodyAsString(), BankRetornoJson.class);
        return new PagamentoRetorno(obj.getMensagem(), false);
    }
    throw ex;
}catch (RuntimeException ex) {
    throw ex;
}
```

# Testar

Pronto, com o código dos 3 primeiros sistemas pronto, podemos subir os 3 sistemas.

Verificar se no RabbitMQ tem as 2 filas que vamos usar.

| Overview | | | Messages | | | Message rates | | | +/- |
|---|---|---|---|---|---|---|---|---|---|
| **Name** | Features | **State** | **Ready** | **Unacked** | **Total** | **incoming** | **deliver / get** | **ack** | |
| **fila-compras-aguardando** | D | idle | 0 | 0 | 0 | 0.00/s | 0.00/s | 0.00/s | |
| **fila-compras-finalizado** | D | idle | 7 | 0 | 7 | 0.00/s | 0.00/s | 0.00/s | |

# Teste 1

Suba todos os sistemas e execute:

1) compra com cartão 123456, código de segurança 12, valor 500;
2) compra com cartão 123456, código de segurança 12, valor 5000.51;
3) compra com cartão 9999, código de segurança 12, valor 50.51;
4) compra com cartão 123456, código de segurança 1, valor 50.51;

Esses dados vão gerar 1 compra correta. As outros vão ter alguma crítica.

Repare também que a fila ***fila-compras-finalizado*** está com 4 itens.

# Teste 2

Derrube o sistema buyprocess.

1) compra com cartão 123456, código de segurança 12, valor 500;
2) compra com cartão 123456, código de segurança 12, valor 5000.51;

Repare que a fila ***fila-compras-aguardando*** tem 2 ítens. Como o sistema que processa a compra não está funcionando, os itens continuam na fila.

# Teste 3

Derrube o sistema bank e suba o buyprocess.

Repare que a fila ***fila-compras-aguardando*** continua com os 2 ítens.
Agora o sistema que processa as comprar está funcionando, mas como ele não consegue se comunicar com o ***Bank***, ele recoloca o item na fila para ser processado novamente.

# Feedback

O sistema ***Buyfeedback*** vai ser o sistema responsável por ouvir a ***fila fila-compras-finalizado*** e gravar no Redis os dados finalizados.

Também vamos criar um endpoint que receberá a chave do pagamento e vai retornar os dados processados. Assim vamos poder saber se todos os sistemas estão funcionando corretamente.

# Redis

Redis é uma fonte aberta (BSD licenciado), armazenamento de estrutura de dados na memória, usado como banco de dados, cache e corretor de mensagens. Ele suporta estruturas de dados como seqüências de caracteres, hashes, listas, conjuntos, etc..

Para subir o Redis (buyfeedback-12-docker-compose.yml)

***sudo docker-compose up***

# Vamos começar

1) criar uma pasta, choppcloud-buyfeedback;
2) criar o pom.xml; (buyfeedback-1-pom.xml)
3) criar as pastas
   src/main/java
   src/main/resources
4) add <module> no pom do projeto pai;
   <module>choppcloud-buyfeedback</module>
5) reimportar o projeto;
6) criar pacote com.br.choppcloud.buyfeedback
7) criar classe ApplicationBuyFeedback;
   (buyfeedback-2-ApplicationBuyFeedback.java)

# Inicie a aplicação

Execute o ApplicationBuyFeedback para ver se o servidor vai subir corretamente.

# Mão na massa

1) Adicionar as chaves no application.properties

```
server.port=8082
spring.rabbitmq.username=user
spring.rabbitmq.password=senha
spring.cloud.stream.bindings.input.destination=compra
spring.cloud.stream.bindings.input.group=buyprocess
fila.finalizado=fila-compras-finalizado
```

(buyfeedback-13-application.properties)

# Mão na massa

2) criar pacote com.br.choppcloud.buyfeedback.finalizar
3) criar classe CompraChaveJson (buyfeedback-3-CompraChaveJson.java)
4) criar classe CompraFinalizadaJson (buyfeedback-4-CompraFinalizadaJson.java)
5) criar classe CompraJson (buyfeedback-5-CompraJson.java)

# Mão na massa

6) criar classe CompraRedis (buyfeedback-6-CompraRedis.java)

```java
@RedisHash("compra")
public class CompraRedis {

    @Id
    private String id;
    private String mensagem;

    private Integer codigoPassagem;
    private Integer nroCartao;
    private BigDecimal valorPassagem;

    private boolean pagamentoOK;
```

# Mão na massa

7) criar interface CompraRedisRepository (buyfeedback-7-CompraRedisRepository.java)

```
public interface CompraRedisRepository extends CrudRepository<CompraRedis, String> {

}
```

# Mão na massa

8) criar a classe ListenerService (buyfeedback-8-ListenerService.java)

```java
@Service
public class ListenerService {

    @Autowired
    private CompraRedisRepository compraRedisRepository;

    @RabbitListener(queues="${fila.finalizado}")
    public void onMessage(Message message) throws JsonParseException, JsonMappingException, IOException  {

        String json = new String(message.getBody(), "UTF-8");

        System.out.println("Mensagem recebida:"+json);

        ObjectMapper mapper = new ObjectMapper();
        CompraFinalizadaJson compraChaveJson = mapper.readValue(json, CompraFinalizadaJson.class);

        CompraRedis credis = new CompraRedis();
        credis.setId(compraChaveJson.getCompraChaveJson().getChave());
        credis.setMensagem(compraChaveJson.getMensagem());
        credis.setNroCartao(compraChaveJson.getCompraChaveJson().getCompraJson().getNroCartao());
        credis.setValorPassagem(compraChaveJson.getCompraChaveJson().getCompraJson().getValorPassagem());
        credis.setCodigoPassagem(compraChaveJson.getCompraChaveJson().getCompraJson().getCodigoPassagem());

        System.out.println("Gravando no redis....");
        compraRedisRepository.save(credis);
    }

}
```

# Mão na massa

9) criar a classe CompraController (buyfeedback-9-CompraController.java)

```java
@RestController
public class CompraController {

    @Autowired
    private CompraRedisRepository compraRedisRepository;

    @RequestMapping(path = "/{chave}", method = RequestMethod.GET)
    public CompraRedis status(@PathVariable("chave") String chave){

        CompraRedis compra = compraRedisRepository.findOne(chave);

        if( compra == null ){
            throw new NaoFinalizadoException();
        }

        return compra;
    }
}
```

# Mão na massa

10) criar classe NaoFinalizadoException (buyfeedback-10-NaoFinalizadoException.java)
11) criar classe ExceptionHandlerController no mesmo pacote do ApplicationBuyfeedback (buyfeedback-11-ExceptionHandlerController.java)

```java
@ControllerAdvice
public class ExceptionHandlerController {

    @ExceptionHandler(NaoFinalizadoException.class)
    @ResponseStatus(value= HttpStatus.BAD_REQUEST)
    @ResponseBody
    public String process(NaoFinalizadoException ex) {
        return "Compra ainda não finalizada.";
    }

}
```

# Hora de brindar

1) ***@RedisHash***
   Anotação para definir o objeto que vai ser armazenado no redis.

```
@RedisHash("compra")
public class CompraRedis {

    @Id
    private String id;
    private String mensagem;

    private Integer codigoPassagem;
    private Integer nroCartao;
    private BigDecimal valorPassagem;

    private boolean pagamentoOK;
```

# Teste 1

Suba todos os sistemas e execute:

1) compra com cartão 123456, código de segurança 12, valor 500;
2) compra com cartão 123456, código de segurança 12, valor 5000.51;
3) compra com cartão 9999, código de segurança 12, valor 50.51;
4) compra com cartão 123456, código de segurança 1, valor 50.51;
5) execute a consulta http://localhost:8082/{chave} para saber o resultado da compra já finalizada;

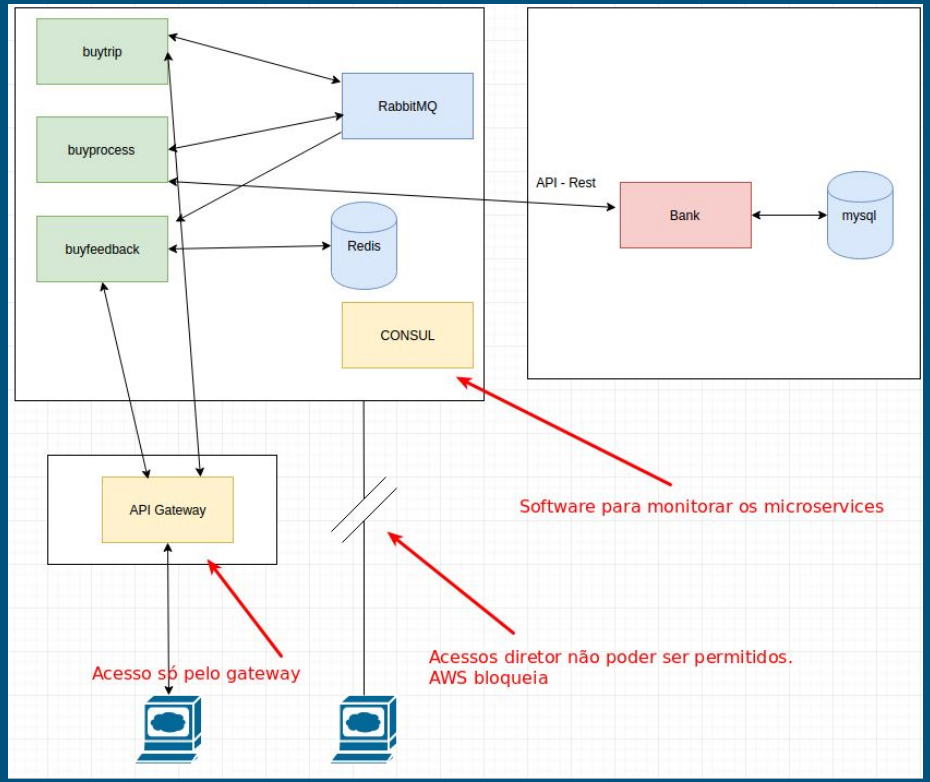
# Observação

Não repararam que em todos os sistemas, precisamos saber a porta da aplicação para poder enviar a compra e depois saber o status?

E se a gente subir duas instâncias do buytrip. Qual porta vamos usar? Vai ser erro porque a porta já vai estar sendo usada?

No nosso projeto temos 4 sistemas integrados, funcionando como um ecossistema, mas ainda não tem todas as características de uma cloud.

# Consul.io + Gateway



# Consul.io

O Consul facilita a inscrição de serviços e a descoberta de outros serviços através de uma interface DNS ou HTTP.

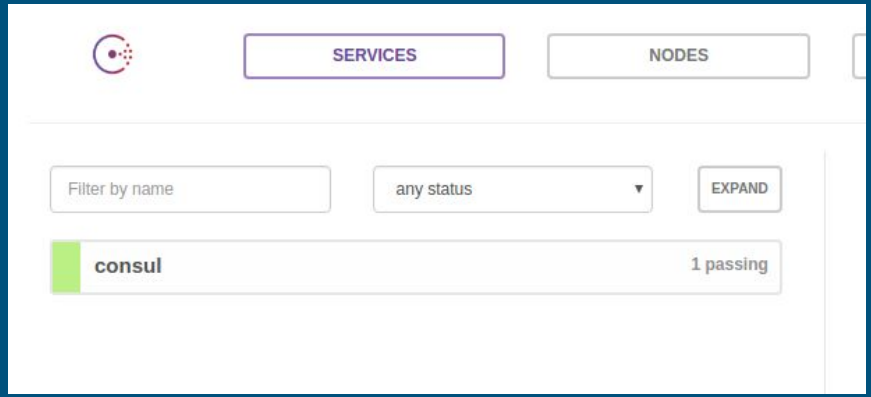
# Iniciar o Consul.io

1) Usar o arquivo infra-1-docker-compose.yml.
   Alterar tag {SEU IP} no arquivo para o seu ip.

2) sudo ps -ef | grep dnsmasq
   Caso apareça algum serviço, dar um kill -9 {codigo}

3) sudo docker-compose up

4) http://localhost:8500/



# Gateway

O Gateway vai ser o sistema que irá interceptar as requisições e redirecionar para os microservices correspondentes.

Para redirecionar vamos usar o ***Zuul***, que integrado ao ***Consul***, consegue saber todos os sistemas disponíveis.

https://github.com/Netflix/zuul

# Vamos começar

1) criar uma pasta, choppcloud-gateway;
2) criar o pom.xml; (infra-2-pom.xml)
3) criar as pastas
   src/main/java
   src/main/resources
4) add <module> no pom do projeto pai;
   <module>choppcloud-gateway</module>
5) reimportar o projeto;
6) criar pacote com.br.choppcloud.gateway
7) criar classe ApplicationGateway; (infra-3-ApplicationGateway.java)
8) criar properties application.properties (infra-5-application.properties)

# Hora de brindar

1) ***Zuul***
   Zuul é um roteador baseado em JVM e balanceador de carga do lado do servidor pela Netflix.
2) ***Consul-discovery***
   Faz com que a aplicação se registre no Consul.io

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-zuul</artifactId>
</dependency>

<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-consul-discovery</artifactId>
</dependency>
```

# Hora de brindar

3) ***@EnableZuulProxy***
   Habilita o Zuul proxy.

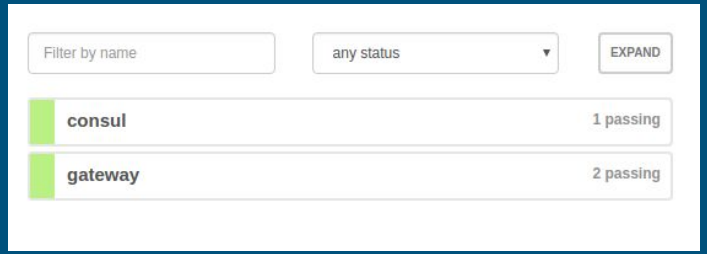
4) ***@EnableDiscoveryClient***
   Habilita o sistema para conectar ao Consul.io

# Inicie a aplicação

Execute o ApplicationGateway para ver se o servidor vai subir corretamente.

```java
@SpringBootApplication
@EnableDiscoveryClient
@EnableZuulProxy
public class ApplicationGateway {

    public static void main(String[] args) {
        SpringApplication.run(ApplicationGateway.class, args);
    }
}
```

```
spring.application.name=gateway

spring.cloud.consul.host=localhost
spring.cloud.consul.port=8500
```



# Integrar BuyTrip

1) No arquivo application.properties, (infra-6-integrar-buytrip.txt)
   spring.application.name=comprar
   spring.cloud.consul.host=localhost
   spring.cloud.consul.port=8500
   server.port=0
2) no pom.xml add:
   <dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-starter-consul-discovery</artifactId>
   </dependency>
3) Adicionar a anotação @EnableDiscoveryClient no ApplicationBuyTrip;

# Integrar BuyFeedback

1) No arquivo application.properties, add:
   spring.application.name=status
   spring.cloud.consul.host=localhost
   spring.cloud.consul.port=8500
   server.port=0
2) no pom.xml add:
   <dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-starter-consul-discovery</artifactId>
   </dependency>
3) Adicionar a anotação @EnableDiscoveryClient no ApplicationBuyFeedback;

# Consul.io


comprar
2 passing
consul
1 passing
gateway
2 passing
status
2 passing

# Teste 1

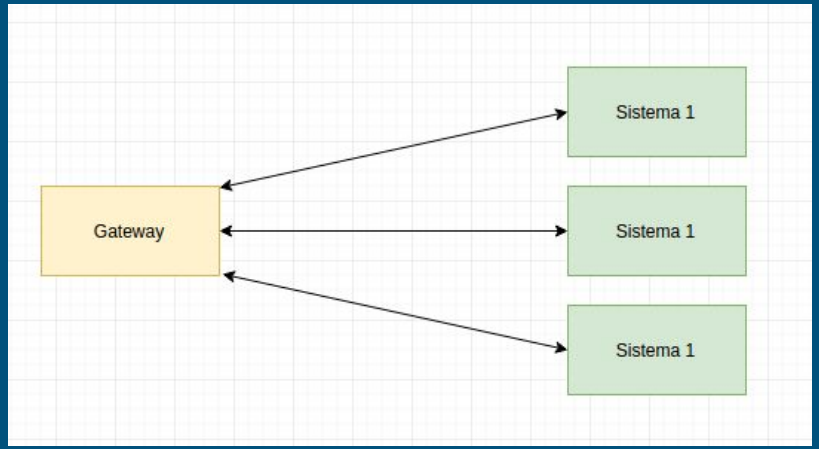
Suba todos os sistemas e execute:

1) Executar uma compra com o link http://localhost:8080/comprar/ compra com cartão 123456, código de segurança 12, valor 500;

2) execute a consulta http://localhost:8080/status/{chave} para saber o resultado da compra já finalizada;

# Loadbalancer

O balanceamento de carga melhora a capacidade de resposta e aumenta a disponibilidade de aplicativos.



# Mão na massa

1) Derrubar o consul
   sudo docker rm -f {id}
2) no docker-compose, alterar a linha “***command***” e add ***-join {IP}***
   command: -server -join 1.1.1.1 agent -bind=2.2.2.2
3) subir o consul
   sudo docker-compose up

# Mão na massa

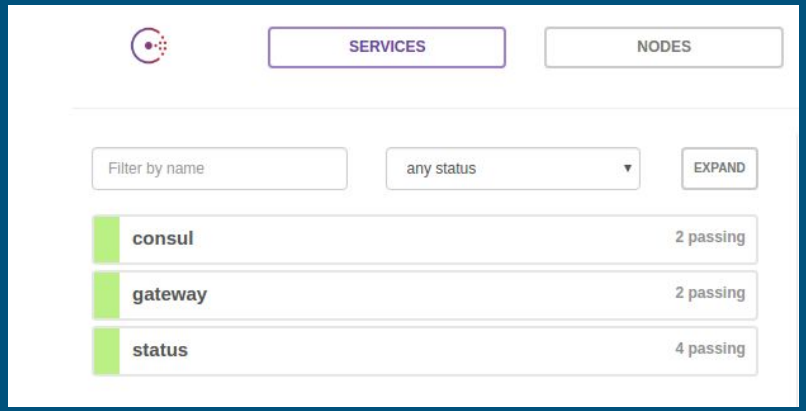
4) No projeto ***Buyfeedback***, alterar a classe CompraController. Adicionar o método status. (infra-4-CompraController.java)

```
@RequestMapping(path = "/meunome", method = RequestMethod.GET)
public String status(){
    return "Estou na máquina do: Marcelo";
}
```

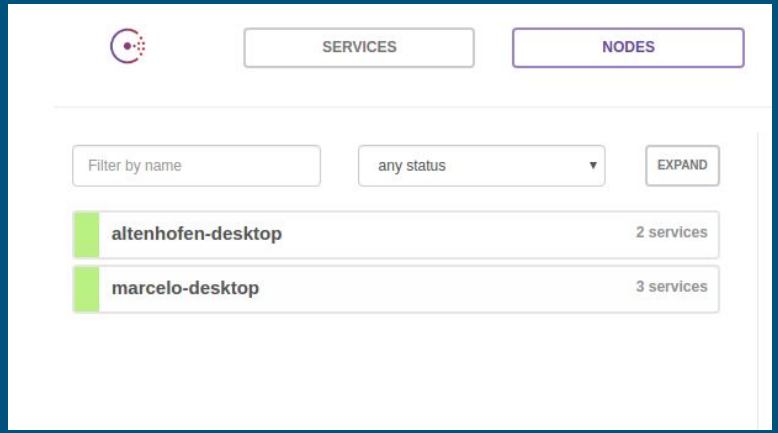
5) Alterar o status para retornar o seu nome.
6) Subir o projeto ***Buyfeedback*.**

# Consul.io / Cluster

Um cluster consiste em computadores vagamente ou fortemente ligados que trabalham em conjunto para que, em muitos aspectos, eles possam ser vistos como um único sistema.




http://localhost:8080/status/meunome
Estou na máquina do: ***Marcelo***



# Monitoramento

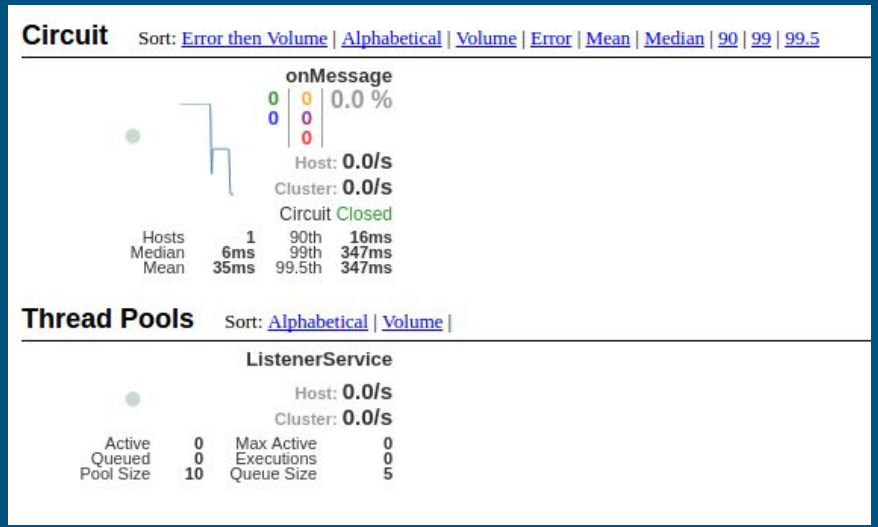
A relevância de um monitoramento de software é identificar pontos estratégicos das funcionalidades mais utilizadas, identificar gargalos e antecipar possíveis erros, agregando valor ao produto

O sistema buyprocess tem um ***Circuit Break*** implementado. Ele foi usado para caso não consiga achar o sistema Bank, ele recoloca o item na fila.

Vamos usar o ***Hystrix Monitor*** para visualizar o que está ocorrendo no nosso sistema em realtime.

# Monitoramento

Acesse:
http://{IP}:8060/hystrix
Add no campo o link do sistema buyprocess.
http://{SEU-IP}:8081/hystrix.stream



# Hora de brindar

# THE END

NÃO ME CHAMA DE IRMÃO, BROTHER

Marcelo de Souza