Processando mensagens com Spring Cloud Stream e Kafka

Henrique Schmidt

@henriquels25

Agenda

- Arquitetura orientada a eventos
- Spring Cloud Stream
- Cenário de negócio
- Arquitetura do Software
- Bindings - consumindo, transformando e enviando eventos
- Tratamento de erros

Arquitetura orientada a eventos

Utilização de eventos para comunicação entre serviços desacoplados

O que é um evento?

Representa alguma alteração no estado do sistema.

Exemplos:

- Item adicionado a um carrinho de compras;
- Solicitação de transferência de dinheiro entre contas;
- Compra entregue para destinatário.

Componentes principais da arquitetura

1. Produtores de eventos
2. Event routers (roteadores de eventos)
3. Consumidores de eventos

Produtores de eventos

O produtor “sente” o evento e publica para o sistema.

Produtores de eventos

Exemplos:

- Sistema de recebimento de pedidos publica evento de criação de pedido;
- Sistema de recuperação de crédito publica evento de divida paga;

Consumidores de eventos

O consumidor recebe o evento e reage conforme sua regra de negócio.

Consumidores de eventos

Exemplos:

- Sistema de envio de notificações recebe evento de compra efetuada com cartão de crédito;
- Sistema de análise de fraudes recebe evento de transferência efetuada.

Produtores e Consumidores de eventos

Tipicamente, são microserviços.

Uma mesma aplicação pode ser tanto um produtor e um consumidor para diferentes eventos.

Consumidores de eventos

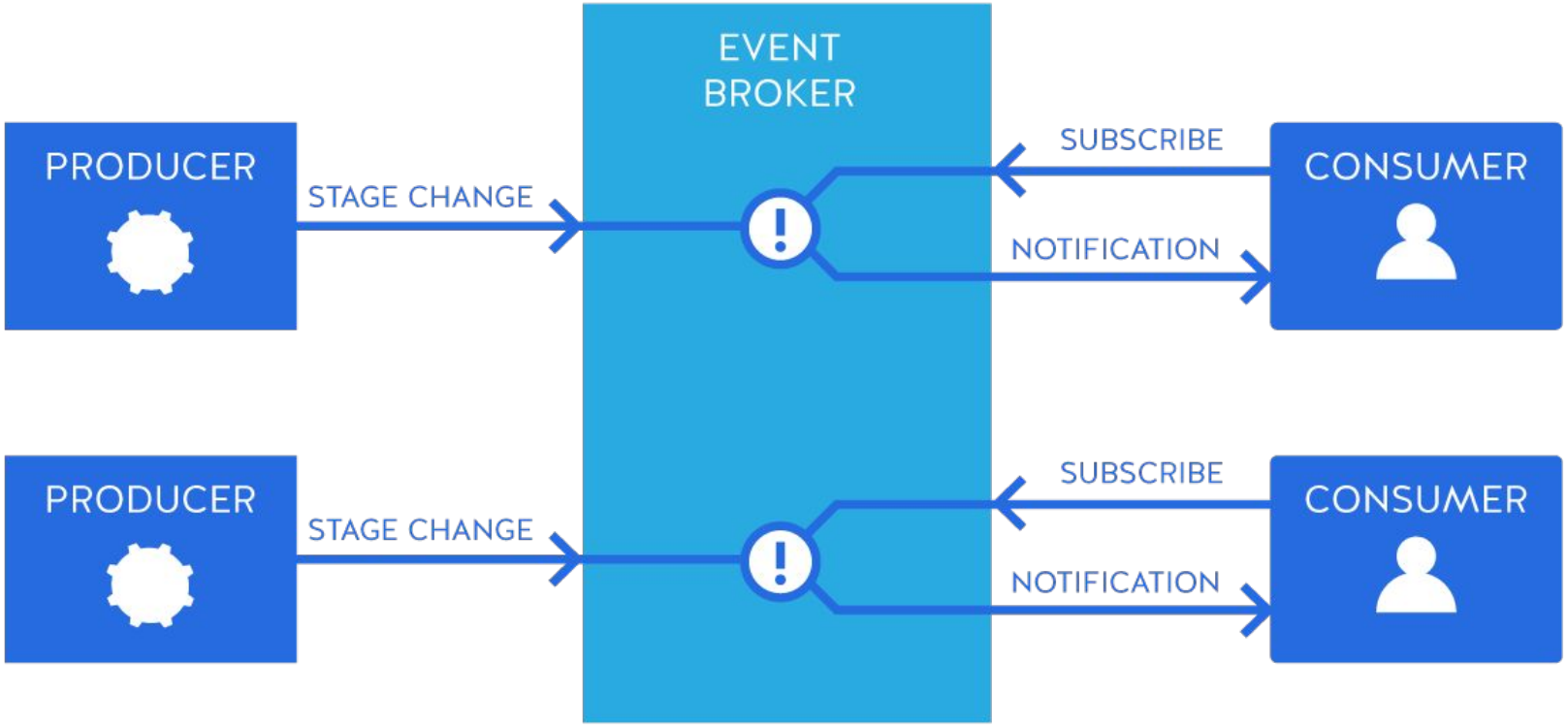
Produtores e Consumidores **NÃO** se conhecem.

Event routers (roteadores de eventos)

Como os consumidores e produtores dos eventos não se conhecem, deve haver algo entre eles, um **middleware.**

Também são chamados de **event brokers.**

Event routers (roteadores de eventos)



Apache Kafka

O Kafka é um broker de mensagens muito utilizado.

Cada evento é enviado para um **tópico** e consumidores se inscrevem para receberem eventos.

Stream

Uma **Stream** é uma sequência ordenada de eventos infinita.

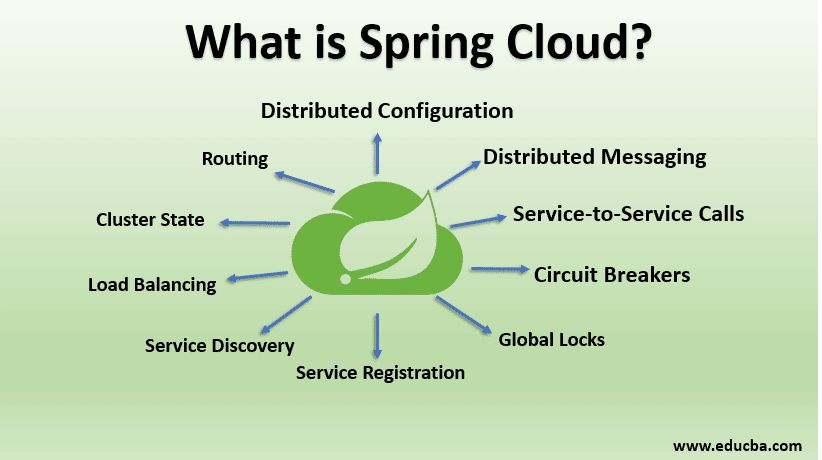
Spring Cloud Stream

O que é?

Um framework para criar microserviços orientados a eventos

Spring Cloud

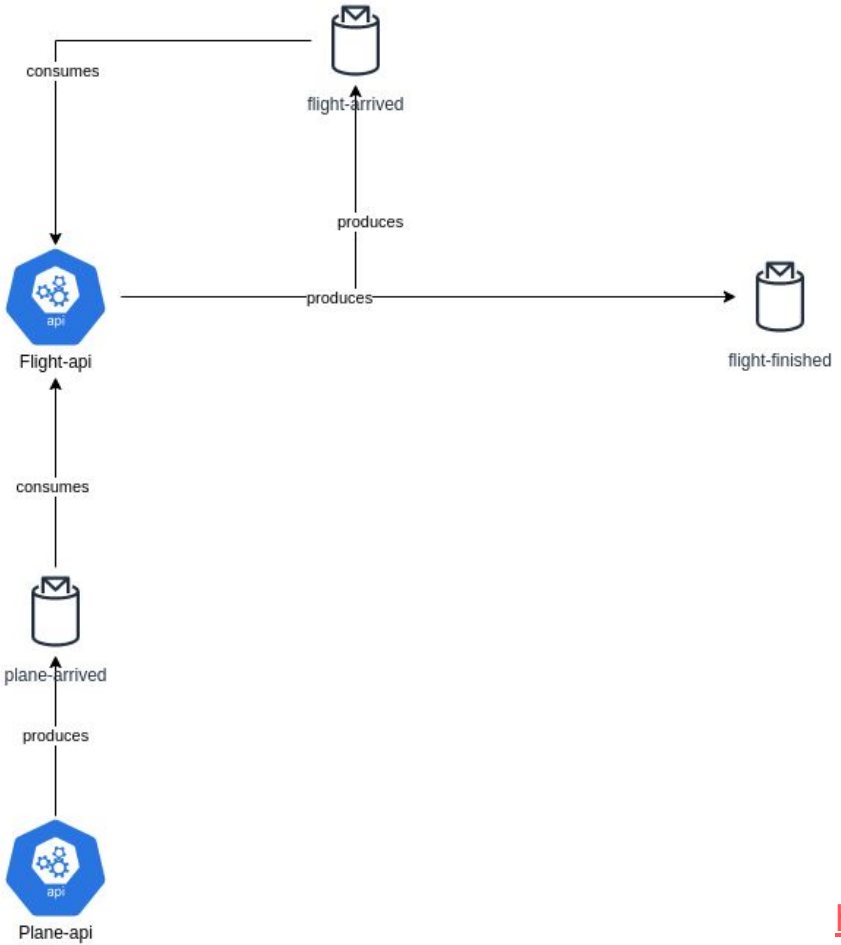
Projeto que gerencia um grupo de ferramentas e frameworks para facilitar a implementação de sistemas distribuídos



Benefícios

- Grande parte do código fonte principal independente do sistema de mensageria;
- Pouquíssimo boilerplate de código;
- Simples de configurar e fácil de usar;
- Bom suporte a testes.

Cenário de Negócio

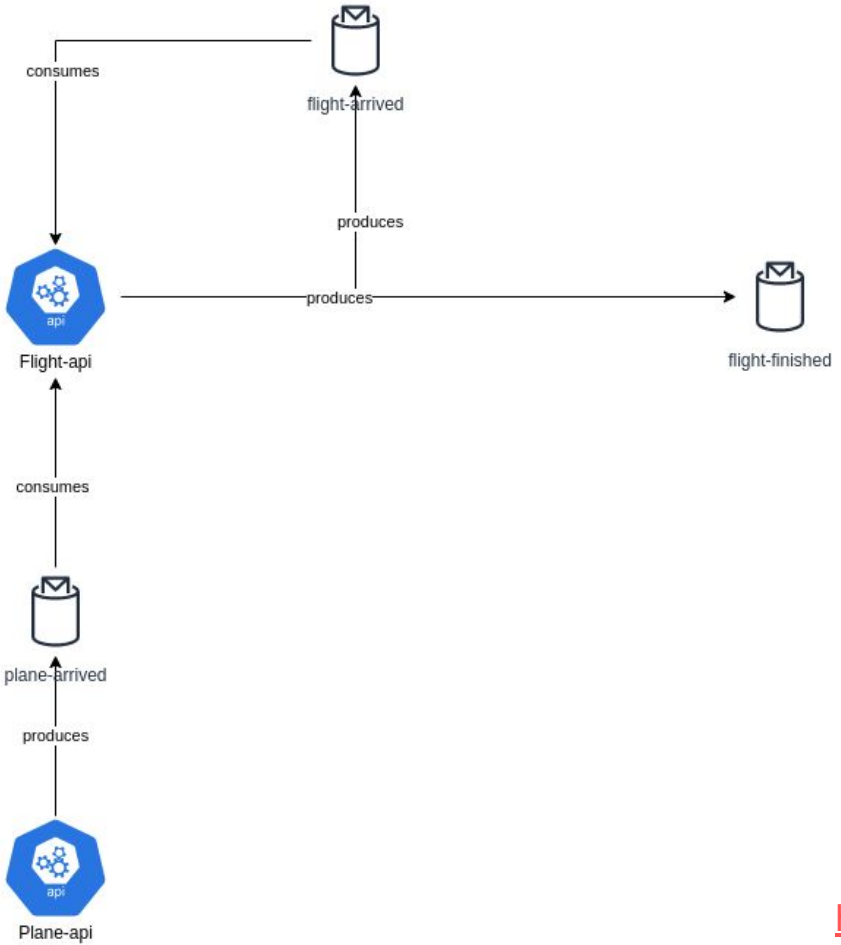


Plane-arrived: Avião pousou

Flight-arrived: Vôo pousou

Flight-finished: Vôo chegou no destino

https://github.com/henriquels25/spring-cloud-stream-sample/



Exemplo:
Voo POA-CNH

Primeiro evento:
Plane-arrived - avião pousou em CNH

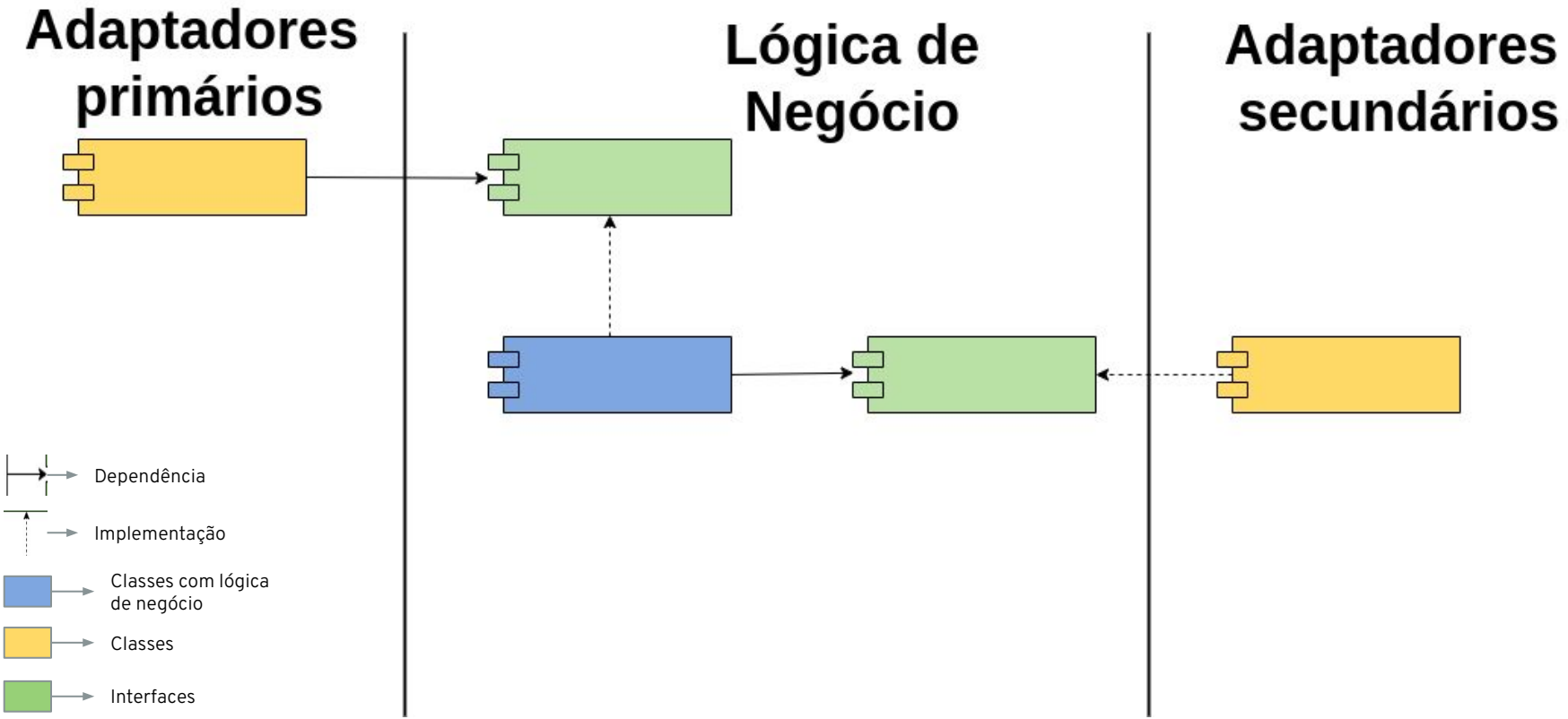
Segundo evento:
Flight-arrived - vôo pousou em CNH

Terceiro evento:
Flight-finished - vôo pousou no destino

https://github.com/henriquels25/spring-cloud-stream-sample/

Arquitetura do Software

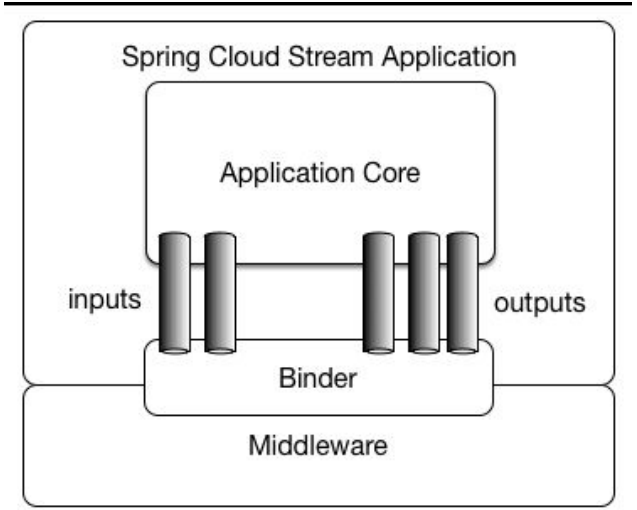
Arquitetura Hexagonal



Bindings

O que são Bindings?

Bindings são pontes que conectam os sistemas de mensageria ao código que implementamos.



Dependência Spring Cloud Stream

Para criarmos bindings no projeto, a dependência do Spring Cloud Stream deve ser adicionada:

```
26      implementation 'org.springframework.cloud:spring-cloud-starter-stream-kafka'
27
```

flight-api/build.gradle

Como definir um binding?

Há duas formas:

- Modelo baseado em anotações **(legado)**;
- Modelo de programação funcional.

Modelo baseado em anotações **(legado)**

```
public interface Source {

    String OUTPUT = "output";

    @Output(Source.OUTPUT)
    MessageChannel output();
}
```

Modelo baseado em anotações **(legado)**

```
public interface Sink {

  String INPUT = "input";

  @Input(Sink.INPUT)
  SubscribableChannel input();

}
```

Modelo de programação funcional

Nas versões atuais o SCS utiliza convenções através das interfaces funcionais do Java:

- Function<I, O>
- Consumer<T>

Consumindo Eventos

Consumer

Para criar um consumidor de um tópico, basta implementar a interface *java.util.Consumer<T>*.

```
@Bean
public Consumer<String> meuConsumer() {
    return System.out::println;
}
```

Classes

Podemos criar consumidores utilizando classes:

```
@Component
class StringConsumer implements Consumer<String> { ...
```

Nome do Binding

Nas propriedades do projeto, sempre o nome do **Spring Bean** que receberá ou enviara mensagens é utilizado.

```
@Bean
public Consumer<?> meuConsumer()

@Bean("consumer1")
public Consumer<?> meuConsumer()
```

Configurando o binding

As propriedades do binding ficam a partir de:

spring.cloud.stream.bindings.<nome-do-binding>.*

Fonte dos dados

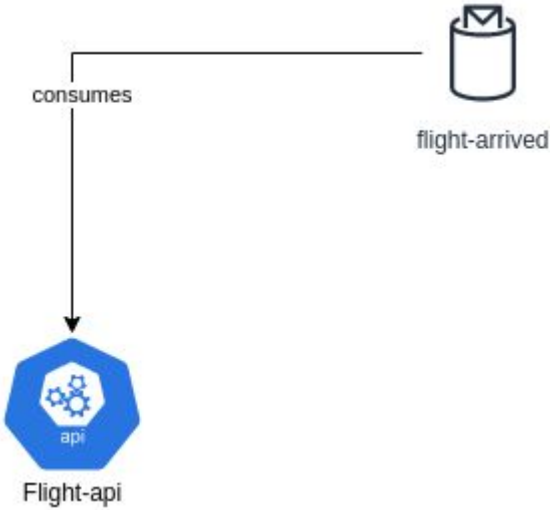
Para especificar de onde vem os dados de binding, é utilizada a configuração **"bindingName-in-<index>.destination"**.

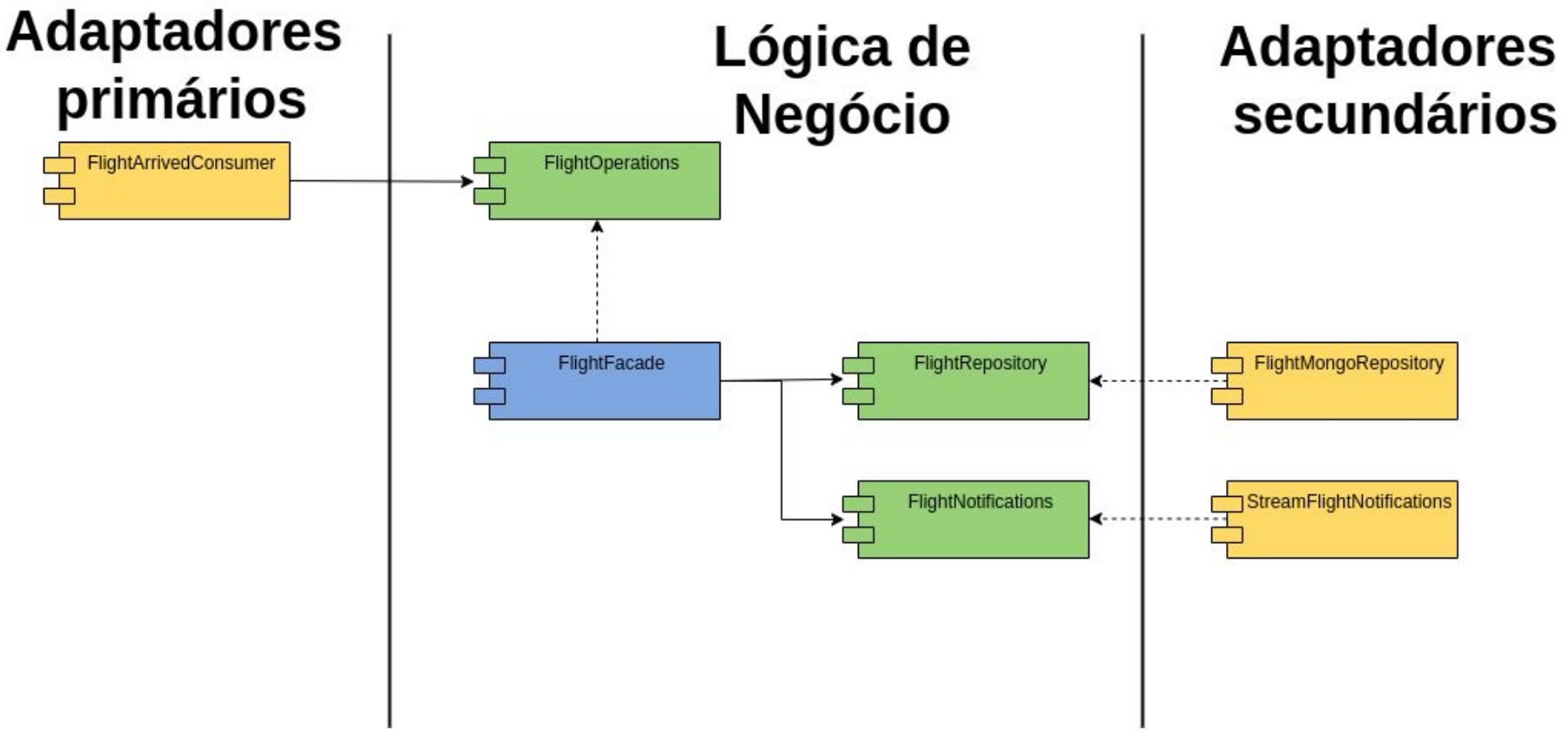
```
spring.cloud.stream.bindings:
  meuConsumer-in-0:
    destination: meuTopicoEntrada
```

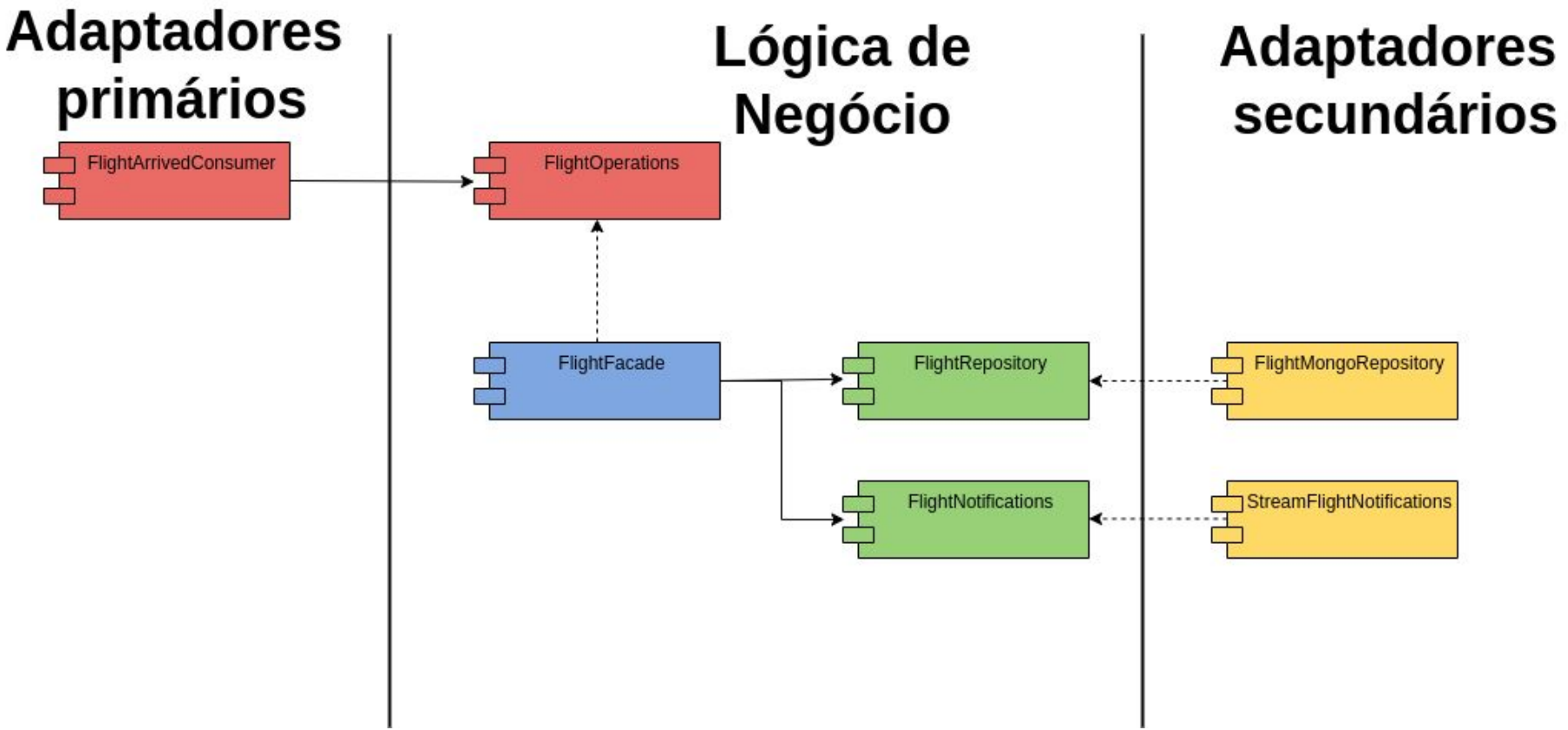
Grupo de consumo

Para especificar de onde vem os dados de binding, é utilizada a configuração **"bindingName-in-<index>.destination"**.

```
spring.cloud.stream.bindings:
  meuConsumer-in-0:
    destination: meuTopicoEntrada
    group: grupoDeConsumo
```


consumes
flight-arrived
api
Flight-api


Adaptadores primários
Lógica de Negócio
Adaptadores secundários
FlightArrivedConsumer
FlightOperations
FlightFacade
FlightRepository
FlightMongoRepository
FlightNotifications
StreamFlightNotifications


Adaptadores primários
Lógica de Negócio
Adaptadores secundários
FlightArrivedConsumer
FlightOperations
FlightFacade
FlightRepository
FlightMongoRepository
FlightNotifications
StreamFlightNotifications

```java
@Component("flightArrivedConsumer")
@AllArgsConstructor
class FlightArrivedConsumer implements Consumer<FlightArrived> {

    private final FlightOperations flightOperations;

    @Override
    public void accept(FlightArrived flightEvent) {
        Airport airport = new Airport(flightEvent.getCurrentAirport());
        flightOperations.flightArrivedIn(flightEvent.getFlightId(), airport);
    }
}
```

flight-api/src/main/java/com/henriquels25/flightapi/flight/infra/stream/FlightArrivedConsumer.java

```yaml
      flightArrivedConsumer-in-0:
        destination: flight-arrived-v1
        group: flight-api
```

flight-api/src/main/resources/application.yml

Transformando eventos

Transformador de eventos

Um transformador de eventos consome e produz uma mensagem ao mesmo tempo.

```
@Bean
public Function<String, String> minhaFuncao() {
    return value -> "Hello, "+ value;
}
```

Classes

Podemos criar transformadores utilizando classes:

```
@Component("toUpperCaseTransformer")
class StringTransformer implements Function<String, String>
{
```

Destino dos dados

Para especificar para onde os dados serão enviados, é utilizada a configuração **"bindingNameout-<index>.destination"**.

```
spring.cloud.stream.bindings:
  meuConsumer-out-0:
    destination: meuTopicoSaida
```

Múltiplos Bindings

Por padrão, só é possível ter um binding em uma aplicação com Spring Cloud Stream. Quando há mais de um binding, eles devem ser informados nas propriedades do projeto:

```
spring.cloud.stream:
  function:
    definition: meuConsumer1;meuConsumer2
```

Enviando a key da mensagem

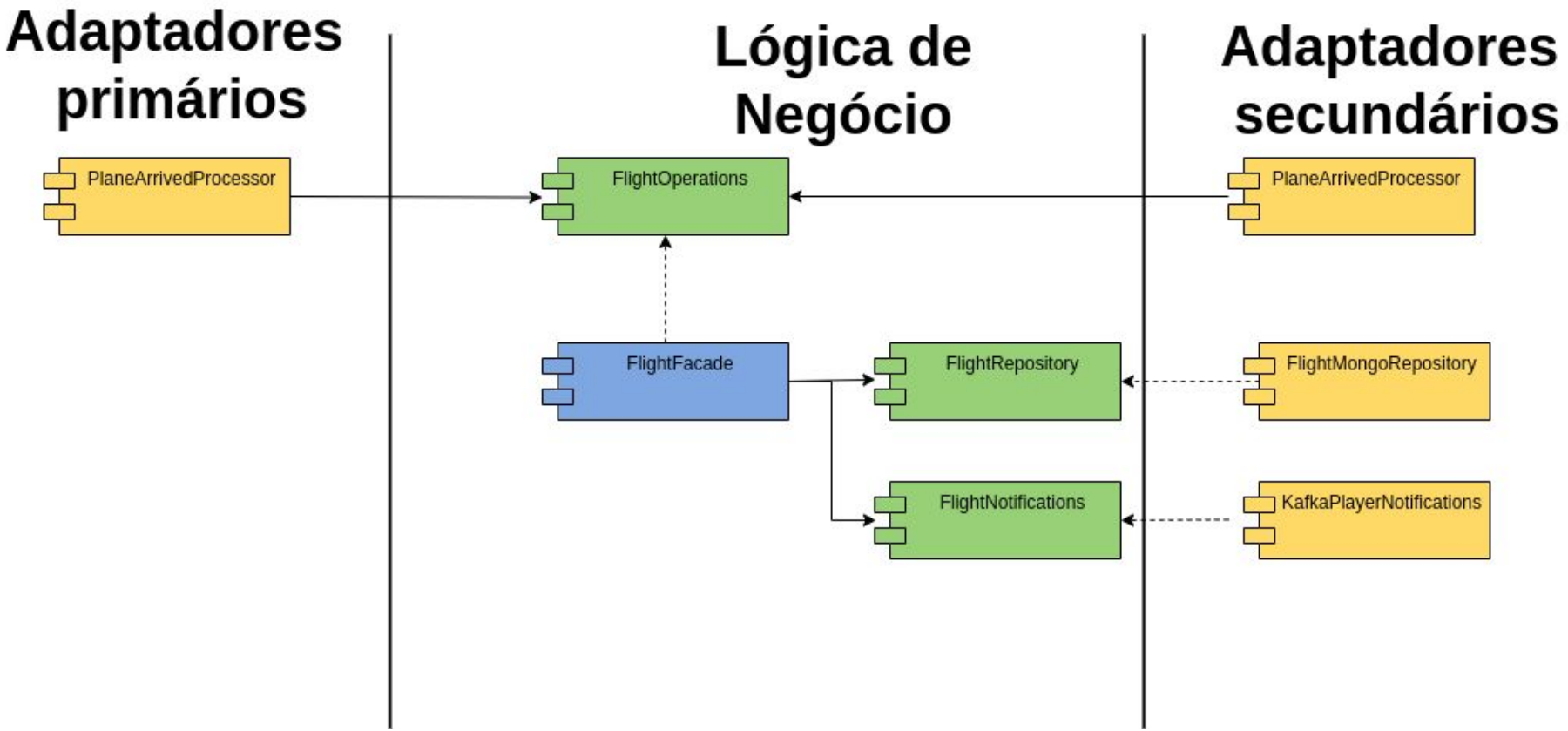
Em muitas situações, é importante configurar a key da mensagem, principalmente para garantir a ordem de consumo dos eventos.

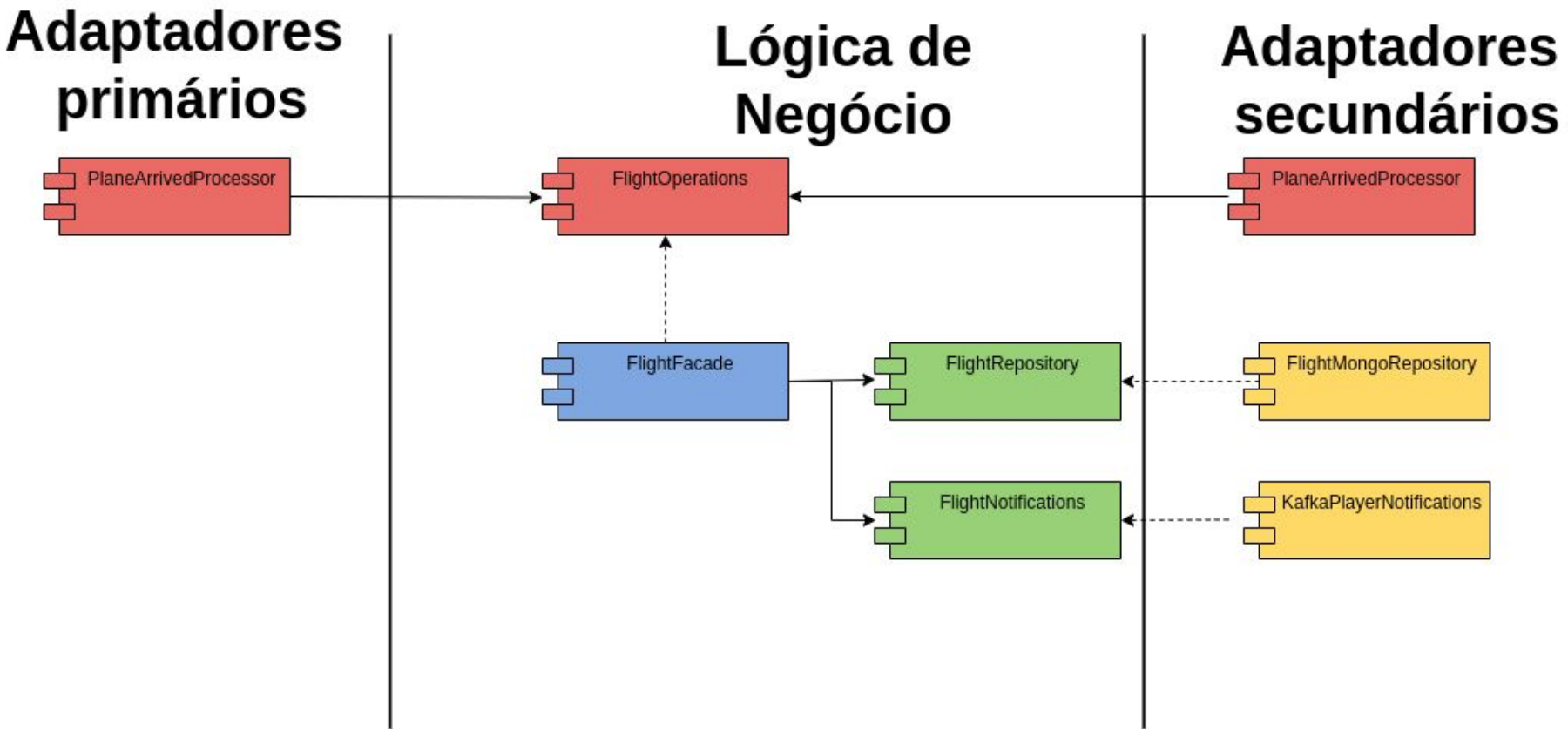
Enviando a key da mensagem

Para fazer isso, uma maneira prática é retornar um objeto do tipo Message na função que produz a mensagem:

```
class PlaneEventProcessor implements Function<PlaneEvent, Message<FlightEvent>> {

    private final FlightOperations flightOperations;
```


api
Plane-api
produces
plane-arrived
consumes
api
Flight-api
produces
flight-arrived


Adaptadores primários
Lógica de Negócio
Adaptadores secundários
PlaneArrivedProcessor
FlightOperations
PlaneArrivedProcessor
FlightFacade
FlightRepository
FlightMongoRepository
FlightNotifications
KafkaPlayerNotifications


Adaptadores primários
Lógica de Negócio
Adaptadores secundários
PlaneArrivedProcessor
FlightOperations
PlaneArrivedProcessor
FlightFacade
FlightRepository
FlightMongoRepository
FlightNotifications
KafkaPlayerNotifications

```java
@Component("planeArrivedProcessor")
@AllArgsConstructor
class PlaneArrivedProcessor implements Function<PlaneArrived, Message<FlightArrived>> {

    private final FlightOperations flightOperations;

    @Override
    public Message<FlightArrived> apply(PlaneArrived planeEvent) {
        String planeId = planeEvent.getPlaneId();
        Flight flight = flightOperations.findConfirmedFlightByPlaneId(planeId)
                .orElseThrow(() -> new NoFlightFoundException(String.format("No flight found for plane id %s", planeId)));
        return MessageBuilder.withPayload(new FlightArrived(flight.getId(), planeEvent.getCurrentAirport()))
                .setHeader(MESSAGE_KEY, flight.getId()).build();
    }
}
```

flight-api/src/main/java/com/henriquels25/flightapi/plane/infra/stream/PlaneArrivedProcessor.java

```yaml
      bindings:
        planeArrivedProcessor-in-0:
          destination: plane-arrived-v1
          group: flight-api
        planeArrivedProcessor-out-0:
          destination: flight-arrived-v1
```

flight-api/src/main/resources/application.yml

```
3     cloud.stream:
4       function.definition: planeArrivedProcessor;flightArrivedConsumer
```

flight-api/src/main/resources/application.yml

Enviando eventos

Stream Bridge

Caso for necessário enviar um evento de forma arbitrária, o Stream Bridge pode ser utilizado.

Stream Bridge

Exemplo:

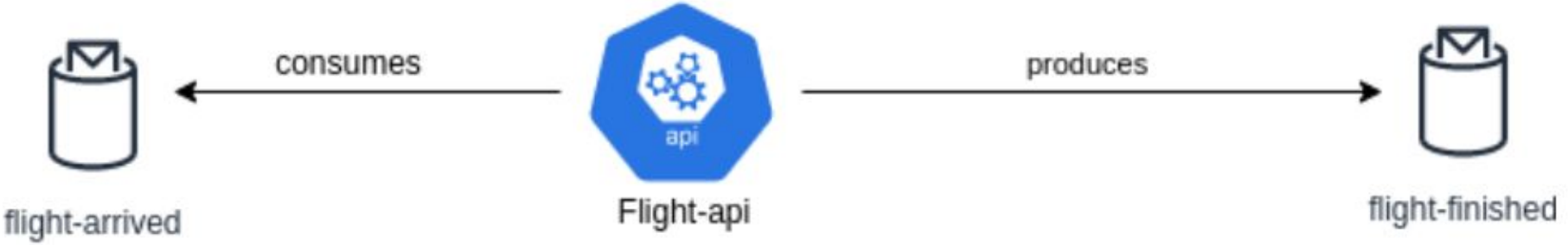
A partir de um endpoint REST, é necessário enviar uma mensagem.

Stream Bridge

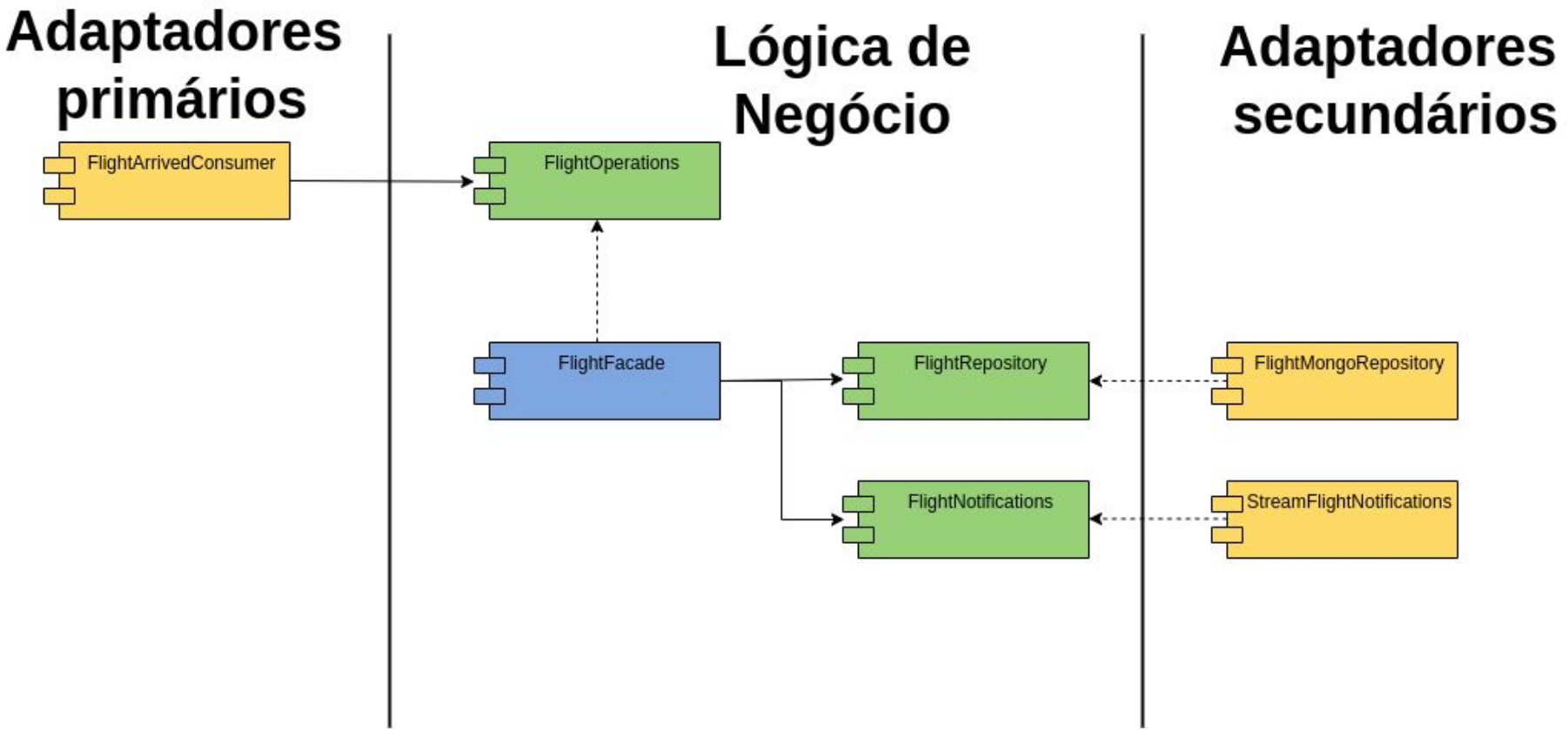
Deve-se apenas injetar o Stream Bridge e configurar um destino nas propriedades.

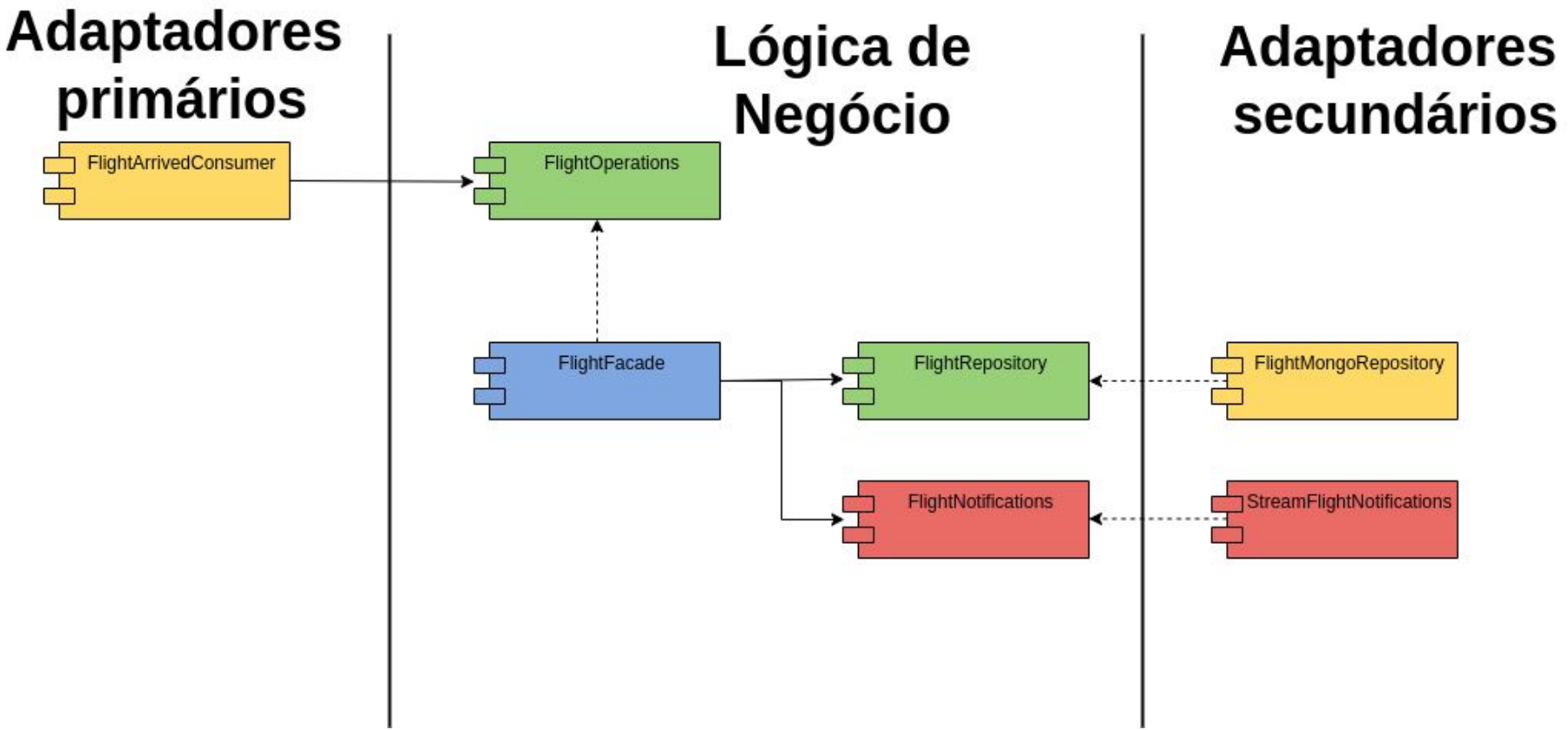
```
@Autowired
private StreamBridge streamBridge;


streamBridge.send("myDestination", body);
```

```
spring.cloud.stream.bindings:
  myDestination-out-0:
    destination: meuTopicoDestino
```



O evento apenas é enviado para flight-finished quando o vôo aterrissa no destino


Adaptadores primários
Lógica de Negócio
Adaptadores secundários
FlightArrivedConsumer
FlightOperations
FlightFacade
FlightRepository
FlightMongoRepository
FlightNotifications
StreamFlightNotifications


Adaptadores primários
Lógica de Negócio
Adaptadores secundários
FlightArrivedConsumer
FlightOperations
FlightFacade
FlightRepository
FlightMongoRepository
FlightNotifications
StreamFlightNotifications

```
52          if (updatedFlight.getStatus() == FINISHED) {
53              flightNotifications.flightFinished(flightId);
54          }
```

flight-api/src/main/java/com/henriquels25/flightapi/flight/FlightFacade.java

```java
@Component
@AllArgsConstructor
class StreamFlightNotifications implements FlightNotifications {

    private final StreamBridge streamBridge;

    @Override
    public void flightFinished(String flightId) {
        streamBridge.send("flightFinished-out-0", new FlightFinishedEvent(flightId));
    }
}
```

flight-api/src/main/java/com/henriquels25/flightapi/flight/infra/stream/StreamFlightNotifications.java

```yaml
        flightFinished-out-0:
          destination: flight-finished-v1
```

flight-api/src/main/resources/application.yml

Tratamento de erros

Dead Letters Queue (DLQ)

Quando a mensagem não é processada corretamente por algum problema, ela pode ser enviada para um tópico especial para posterior análise e reprocessamento.

Dead Letters Queue (DLQ)

Com o SCS, é possível enviar uma mensagem para uma DLQ com apenas duas propriedades:

```
spring.cloud.stream:
     kafka:
       bindings:
         meuBinding:
          consumer:
            enableDlq: true
            dlqName: meuTopicoDLQ
```

Retries

O SCS também permite, de maneira prática, configurar retentativas de processamento para uma mensagem. A mensagem será descartada ou enviada para dead letter apenas depois deste reprocessamento.

Retries

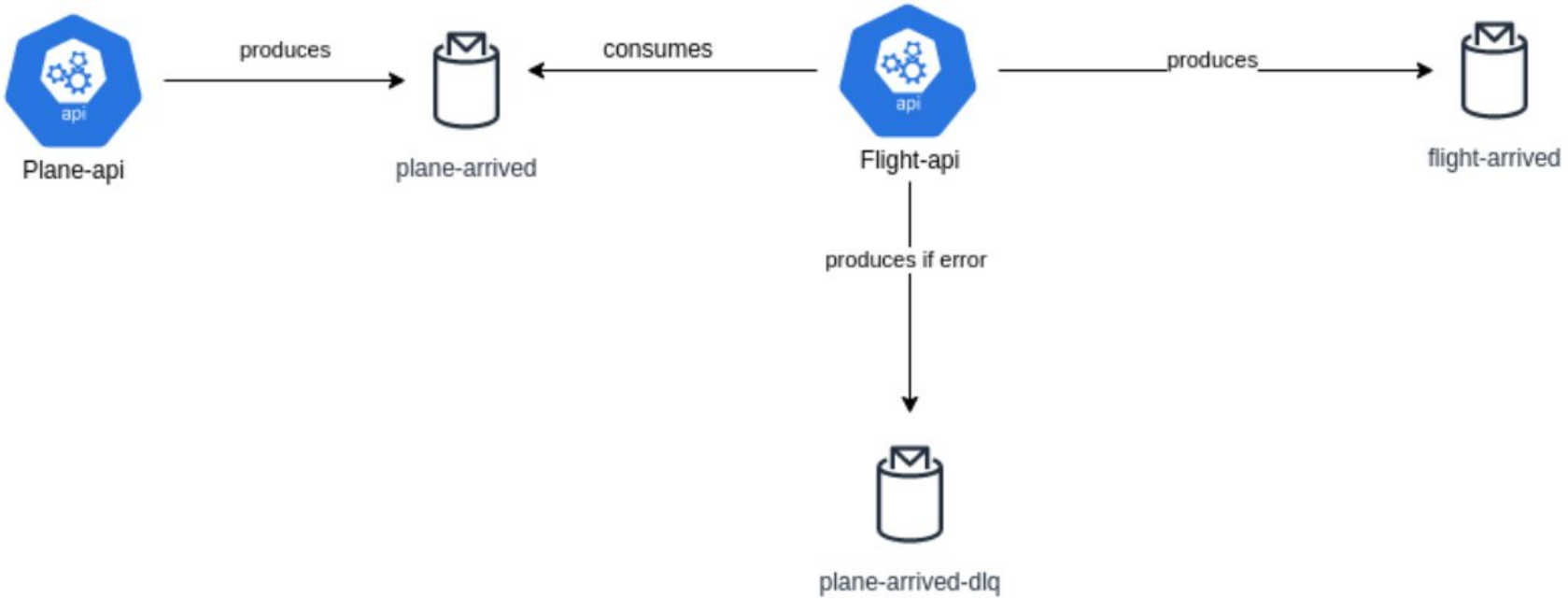
O valor default de retentativas é **3**. Podemos alterar o valor na seguinte propriedade:

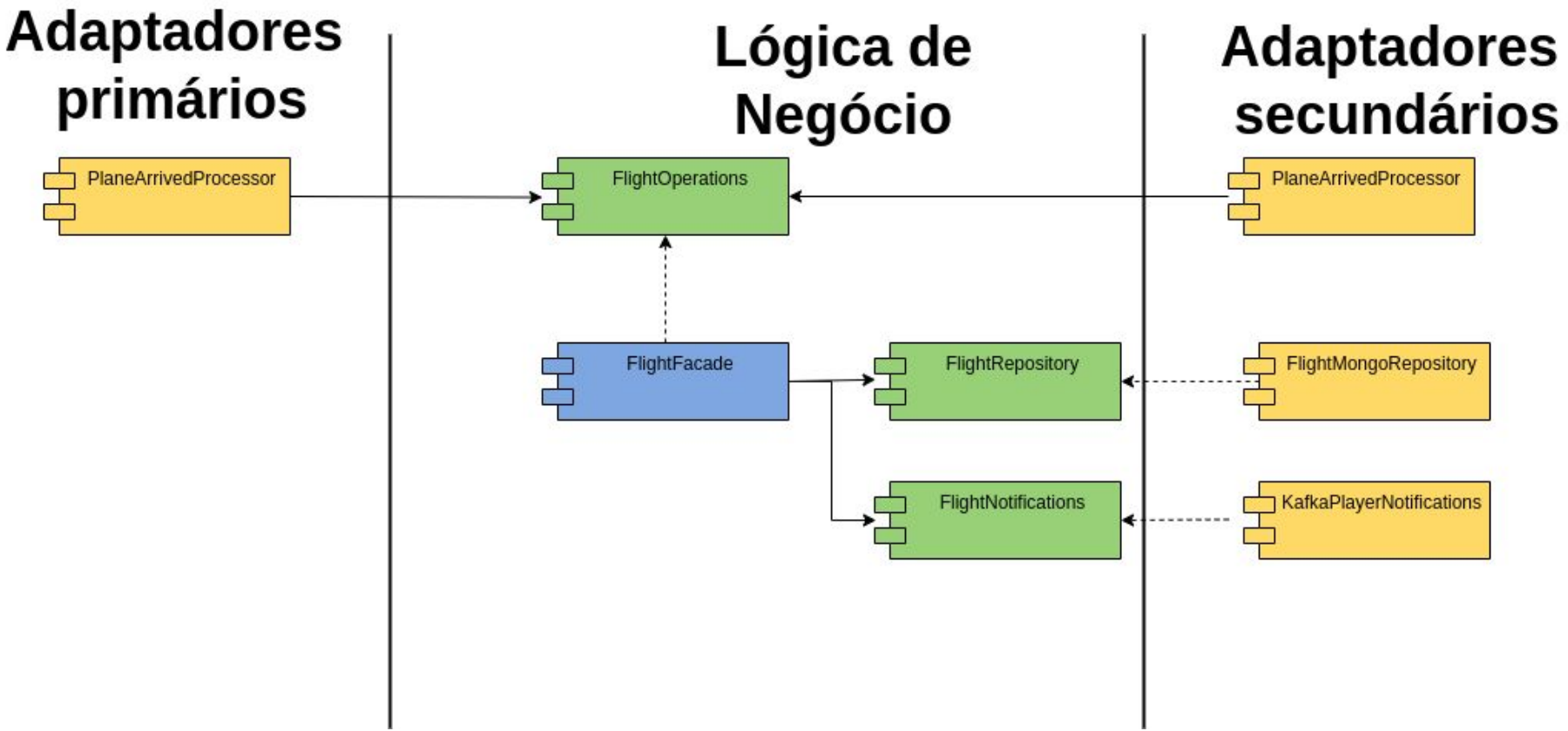
```
spring.cloud.stream:
     bindings:
       meuBinding:
        consumer:
            maxAttempts: 0
```

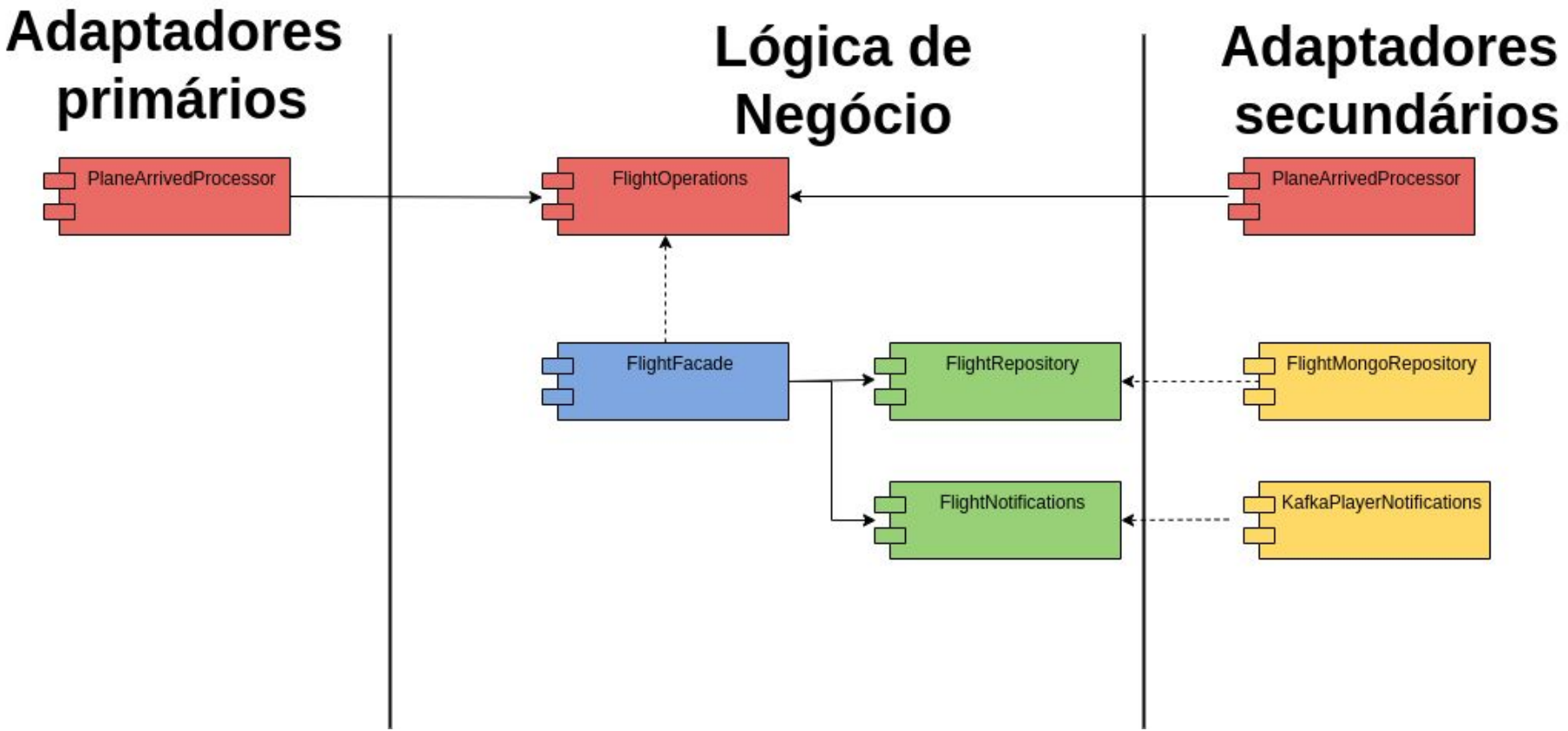
Retries - Exceptions

Por padrão, para todas exceptions são realizadas retentativas de processamento. É possível definir uma lista de exceptions para retentativa.

```
spring.cloud.stream:
     bindings:
       meuBinding:
        consumer:
            retryable-exceptions:
             MinhaException:false
```


api
Plane-api
produces
plane-arrived
consumes
api
Flight-api
produces
flight-arrived
produces if error
plane-arrived-dlq


Adaptadores primários
Lógica de Negócio
Adaptadores secundários
PlaneArrivedProcessor
FlightOperations
PlaneArrivedProcessor
FlightFacade
FlightRepository
FlightMongoRepository
FlightNotifications
KafkaPlayerNotifications


Adaptadores primários
Lógica de Negócio
Adaptadores secundários
PlaneArrivedProcessor
FlightOperations
PlaneArrivedProcessor
FlightFacade
FlightRepository
FlightMongoRepository
FlightNotifications
KafkaPlayerNotifications

```
5       kafka:
6         bindings:
7           planeArrivedProcessor-in-0:
8             consumer:
9               enableDlq: true
10              dlqName: plane-arrived-dlq-v1
```

```
18    bindings:
19      planeArrivedProcessor-in-0:
20        destination: plane-arrived-v1
21        group: flight-api
22        consumer:
23          retryable-exceptions:
24            com.henriquels25.flightapi.plane.infra.stream.NoFlightFoundException: false
```

flight-api/src/main/resources/application.yml

Dúvidas ou comentários?