

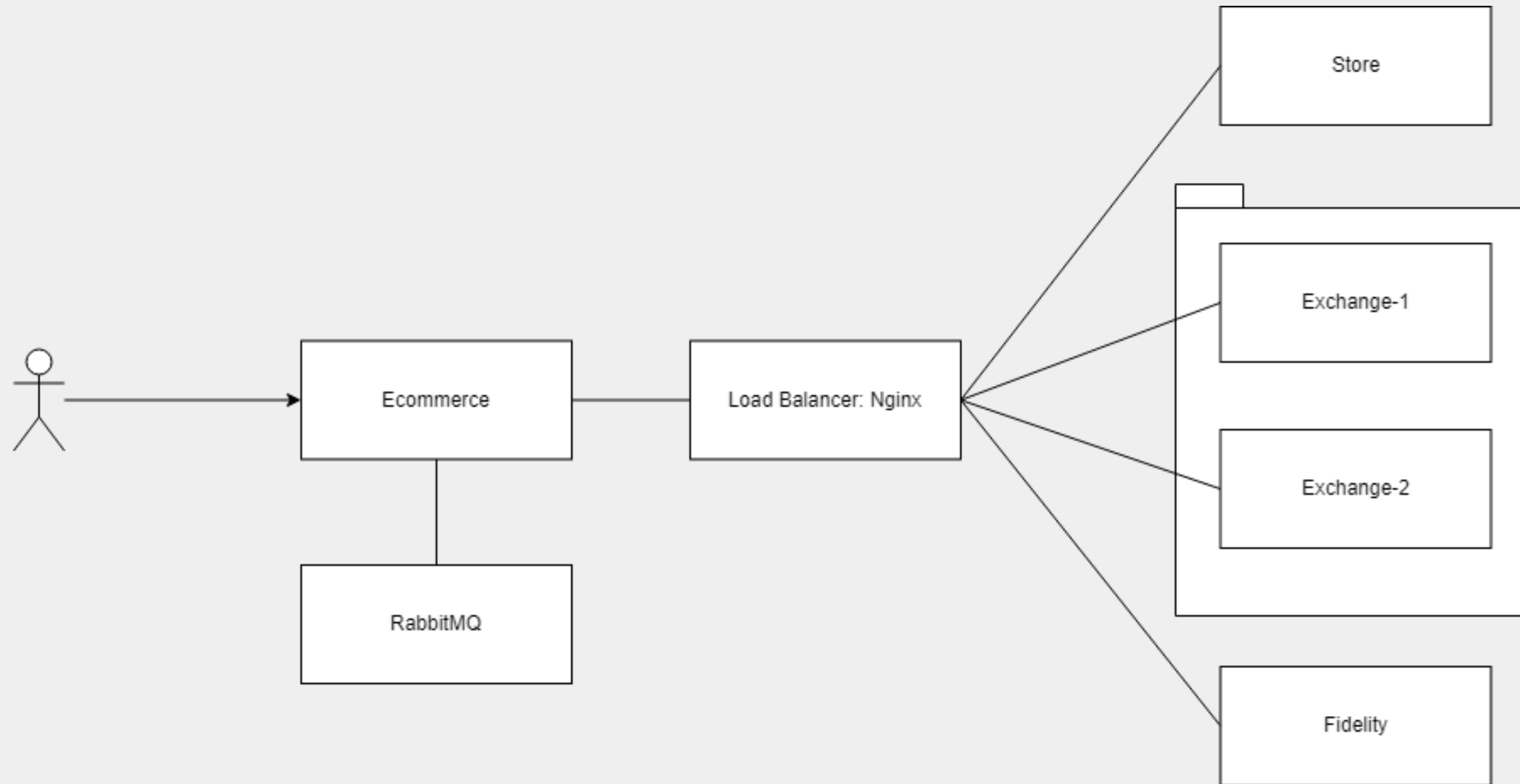
# **Tópicos Especiais Em Engenharia de Software IV**

Alunos:

Ian Jerônimo Nobre Barreto

Victor Gabriel Sousa de Castro

# Containers da aplicação:



# Tecnologias Utilizadas

- Implementação do Ecommerce e dos micro serviços utilizando **Java** com o framework **Spring**;
- Utilização do **Docker** para empacotamento dos serviços em contêineres;
- **Spring Cloud Open Feign** para realizar a comunicação entre os serviços;
- **NGINX** utilizado atuando como load balancer para gerenciar as requisições.



# Ecommerce - Buy

- Request 0
- Feita por agente externo
- Deve receber no fim do processamento:
  - HTTP Response Code
  - Identificador aleatório gerado no Request 3
- **product:** id do produto a ser comprado;
- **user:** id do usuário que está executando a compra;
- **ft:** parâmetro que indica se a tolerância a falhas será ativada.

```
url = "http://localhost:9090/buy"
params = {
    "product": 11,
    "user": 2,
    "ft": True
}
```

# Store - Product

- **Request 1 (Omission, 0.2s, 0s)**
- **Estratégia implementada:** Retry com limite de tentativas;
- **Implementada** no ProductService do Ecommerce;
- **Limitação:** Não resolve indisponibilidades prolongadas e aumenta o tempo total da resposta devido a múltiplas tentativas.

```
@Autowired
private StoreClient storeClient;

public Product getProductById(Long id, boolean ft) {
    if(ft) {
        return getProductWithRetry(id);
    }

    return getProductWithoutRetry(id);
}

private Product getProductWithRetry(Long id) {
    int maxAttempts = 5;
    int attempt = 0;
    while (attempt < maxAttempts) {
        try {
            return storeClient.product(id);
        } catch (Exception e) {
            attempt++;
            System.out.println("Attempt " + attempt + " failed: " + e.getMessage());
            if (attempt >= maxAttempts) {
                throw new RuntimeException("Max retry attempts reached", e);
            }
        }
    }
    throw new RuntimeException("Unexpected error");
}

private Product getProductWithoutRetry(Long id) {
    return storeClient.product(id);
}
```

# Store - Product

```
private Product getProductWithRetry(Long id) {  
    int maxAttempts = 5;  
    int attempt = 0;  
    while (attempt < maxAttempts) {  
        try {  
            return storeClient.product(id);  
        } catch (Exception e) {  
            attempt++;  
            System.out.println("Attempt " + attempt + " failed: " + e.getMessage());  
            if (attempt ≥ maxAttempts) {  
                throw new RuntimeException("Max retry attempts reached", e);  
            }  
        }  
    }  
    throw new RuntimeException("Unexpected error");  
}
```

# Exchange

- **Request 2: Fail (Crash, 0.1, \_ )**
- **Estratégia implementada:** uso de réplicas e do último valor da taxa de conversão;
- **Implementada** no ExchangeService do Ecommerce;
- **Limitação:** Não resolve quando as duas instâncias caírem, o que pode ser resolvido com o uso de Kubernetes. Sem reinicialização automática do serviço valor ficaria defasado com o tempo.

```
public double getExchangeValue(boolean ft) {
    if (ft) {
        return getExchangeOrLastValue();
    }

    return getExchangeWithoutLastValue();
}

private double getExchangeWithoutLastValue() {
    return exchangeClient.getExchangeRate();
}

private double getExchangeOrLastValue() {

    try {
        double exchangeResponse = exchangeClient.getExchangeRate();
        setExchangeLastValue(exchangeResponse);

        return exchangeResponse;
    } catch (FeignException e) {
        System.out.println("FeignException: " + e.getMessage());

        return getExchangeLastValue();
    } catch (Exception e) {
        System.out.println("Exception: " + e.getMessage());

        return getExchangeLastValue();
    }
}
```



# Store - Sell

- **Request 3: Fail (Error, 0.1, 5s)**
- **Estratégia implementada:** Retry com limite de tentativas combinado a implementação de Circuit Breaker;
- **Implementada** no SellService do Ecommerce;
- **Limitação:** O Circuit Breaker é eficaz para falhas transitórias, mas não resolve falhas permanentes.

```
public String processSellWithRetry(Long id) {
    if (!circuitBreaker.allowRequest()) {
        throw new RuntimeException("Serviço de vendas indisponível. Tente novamente mais tarde.");
    }

    int maxAttempts = 2;
    int attempt = 0;


    while (attempt < maxAttempts) {
        try {
            String response = storeClient.sell(id).getBody();
            circuitBreaker.onSuccess();
            return response;
        }
        catch (Exception e) {
            attempt++;
            circuitBreaker.onFailure();

            try {
                Thread.sleep(SELL_FAILURE_STATE_TIME);
            } catch (InterruptedException ie) {
                Thread.currentThread().interrupt();
                throw new RuntimeException("Erro ao processar a venda. Tente novamente mais tarde.", ie);
            }

            if (attempt >= maxAttempts) {
                throw new RuntimeException("Erro ao processar a venda. Tente novamente mais tarde.", e);
            }
        }
    }
}
```



# Store - Sell



```
public final StoreClient storeClient;
private final CircuitBreaker circuitBreaker;
private static final long SELL_FAILURE_STATE_TIME = 5 * 1000;

public SellService(StoreClient storeClient) {
    this.storeClient = storeClient;
    this.circuitBreaker = new CircuitBreaker(3, SELL_FAILURE_STATE_TIME, 3);
}

public String sellProduct(Long id, Boolean ft) {
    if(ft) {
        return processSellWithRetry(id);
    }

    return processSellWithoutRetry(id);
}
```

# Fidelity

- **Request 4 (Time = 2s, 0.1s, 30s)**
- **Estratégia implementada:** Registrar no log e processar quando possível, utilizando RabbitMQ;
- **Implementada** no FidelityService, FidelityProducer e FidelityConsumer do Ecommerce;
- **Limitação:** Re-enfileirar mensagens repetidamente pode causar congestionamento na fila, especialmente em períodos de falhas prolongadas.

```
@Autowired
private FidelityClient fidelityClient;

@Autowired
private FidelityProducer fidelityProducer;

public void addBonus(Long user, int bonus, boolean ft) {
    if(ft) {
        addBonusWithRabbit(user,bonus);
    } else {
        addBonusWithoutRabbit(user,bonus);
    }
}

private void addBonusWithRabbit(Long user, int bonus) {
    try {
        fidelityClient.addBonus(user, bonus);
    } catch (FeignException e) {
        fidelityProducer.sendMessage(user, bonus);
    }
}

private void addBonusWithoutRabbit(Long user, int bonus) {
    fidelityClient.addBonus(user, bonus);
}
```

# Fidelity

## FidelityService

```
@Autowired
private FidelityClient fidelityClient;

@Autowired
private FidelityProducer fidelityProducer;

public void addBonus(Long user, int bonus, boolean ft) {
    if(ft) {
        addBonusWithRabbit(user,bonus);
    } else {
        addBonusWithoutRabbit(user,bonus);
    }
}

private void addBonusWithRabbit(Long user, int bonus) {
    try {
        fidelityClient.addBonus(user, bonus);
    } catch (FeignException e) {
        fidelityProducer.sendMessage(user, bonus);
    }
}

private void addBonusWithoutRabbit(Long user, int bonus) {
    fidelityClient.addBonus(user, bonus);
}
```

## FidelityProducer

```
public void sendMessage(Long user, int bonus) {
    FidelityMessage message = new FidelityMessage(user, bonus);
    rabbitTemplate.convertAndSend(RabbitMQConfig.EXCHANGE_NAME, RabbitMQConfig.ROUTING_KEY, message);
}
```

## FidelityConsumer

```
@RabbitListener(queues = RabbitMQConfig.QUEUE_NAME)
public void receiveMessage(FidelityProducer.FidelityMessage message) {
    try {
        System.out.println("Consumindo mensagem... " + message);
        fidelityClient.addBonus(message.getUser(), message.getBonus());
        System.out.println("Mensagem consumida com sucesso... " + message);
    } catch (FeignException e) {
        System.out.println("Tentativa falha, reenviando mensagem... " + message);
        fidelityProducer.sendMessage(message.getUser(), message.getBonus());
    }
}
```

# Outras Estratégias

- **Manipulador de Erros (Error Handling):** Identifica erros do sistema em tempo real e executa processos previamente configurados para solucionar problemas de forma automática;
- **Failover:** quando um erro for identificado a operação é transferida para um sistema secundário;
- **Reinicialização:** quando um erro irrecuperável for identificado os processos são encerrados de maneira segura e iniciado novamente.

# Como executar o projeto:

Para executar o projeto, basta utilizar o seguinte comando na raiz do projeto:

```
$ docker-compose up --build
```

A aplicação Ecommerce ficará disponível em <http://localhost:9090>

Os outros serviços ficarão disponíveis em <http://localhost:8080>