



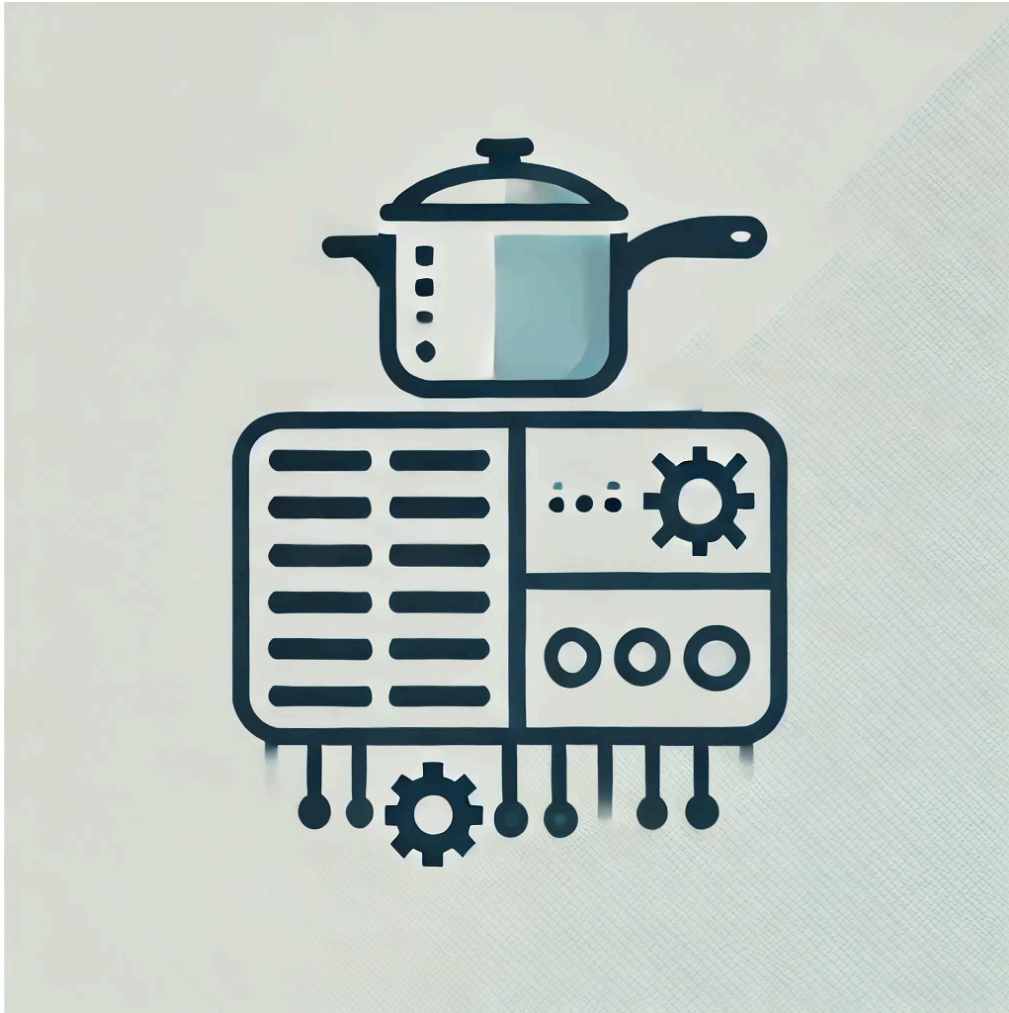
PROJETO DE BLOCO

ASSESSMENT

Professor: Francisco Benjamim Filho

Aluno: Frederico Flores

KitchenSystem



<https://github.com/nagualcode/kithcensystem>

INDICE

1. Introdução.....	4
2. Escolhas Tecnológicas.....	4
2.2 Tecnologias Utilizadas.....	4
3. Estrutura da Aplicação: Design em Camadas.....	6
3.1 Modelagem de Domínios e DDD.....	6
3.1.1 Estrutura dos Domínios.....	6
3.2 Diagrama de Classes.....	7
3.3 Integração com o Frontend (React).....	7
4. Arquitetura Orientada a Eventos com RabbitMQ.....	8
4.1 Vantagens da Arquitetura Orientada a Eventos.....	8
4.2 Diagrama de sequência.....	9
5. Gerenciamento de Banco de Dados com Flyway Docker.....	9
5.1 Estrutura do Banco de Dados.....	9
5.2 Execução das Migrações.....	10
6. Containerização (Docker).....	10
7. Testes Abrangentes (JUnit).....	11
8. Github Actions.....	12
8. Rodando a aplicação.....	12
9. Conclusão.....	14

1.Introdução

KitchenSystem é uma aplicação projetada para gerenciar as operações de uma cozinha de restaurante, abrangendo funcionalidades como criação de pedidos, processamento de pagamentos, gerenciamento de cardápio, e impressão da ordens para a cozinha em papel. O desenvolvimento dessa solução iniciou-se como um Monolito no início do bloco, e evoluiu para uma abordagem de **microsserviços** altamente escalável, utilizando **Spring Boot**.

O objetivo principal era criar uma arquitetura distribuída que fosse capaz de lidar com cargas de trabalho variadas, mantendo a simplicidade de manutenção e implementação de novas funcionalidades.

Ao longo deste relatório, detalharei as escolhas tecnológicas, os desafios enfrentados e como as soluções adotadas permitiram atingir os objetivos do projeto.

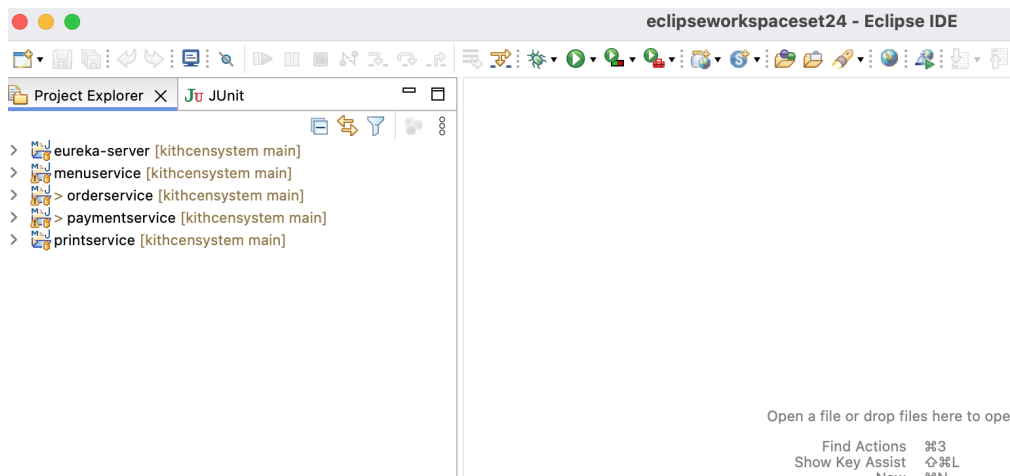
2.Escolhas Tecnológicas

A arquitetura da aplicação foi baseada em uma abordagem de **microsserviços** para garantir flexibilidade e escalabilidade. Cada funcionalidade foi separada em serviços independentes, garantindo que cada componente pudesse ser desenvolvido, testado e implantado de forma isolada. Essa escolha garantiu que o sistema pudesse ser facilmente mantido e escalado conforme a demanda do restaurante aumentasse.

2.2 Tecnologias Utilizadas

- **Spring Boot**: Utilizado para facilitar a criação de microsserviços e permitir a autoconfiguração.
- **Spring MVC**: Para implementar **APIs REST** e facilitar a comunicação entre o frontend e o backend.

- **Spring Data JPA:** Para o mapeamento objeto-relacional com o banco de dados, utilizando anotações JPA.
- **RabbitMQ:** Utilizado como broker de mensagens na arquitetura orientada a eventos.
- **PostgreSQL:** Banco de dados utilizado por cada microsserviço com **schemas** independentes para manter o isolamento dos domínios.
- **FLYWAY:** Para automatizar criação e formatação do banco de dados.
- **React:** Para criar uma interface de usuário que consome as APIs REST e permite interação fluida com o backend.
- **Docker:** Para containerização dos microsserviços e orquestração em ambiente de produção.
- **JUnit e Testcontainers:** Para testes automatizados de unidade e integração.
- **GITHUB:** Para repositório do código.
- **Eclipse IDE:** Para desenvolvimento do código JAVA.



3. Estrutura da Aplicação: Design em Camadas

Ao projetar a arquitetura da aplicação, segui o **design em camadas**, garantindo uma clara separação de responsabilidades:

1. **Camada de Controle:** Lida com as requisições HTTP e utiliza as anotações do **Spring MVC** para expor APIs REST que interagem com o frontend. Esta camada comunica-se diretamente com a camada de serviço.
2. **Camada de Serviço:** Contém a lógica de negócios da aplicação. Aqui, apliquei os princípios **SOLID** para garantir que cada serviço tenha uma única responsabilidade, facilitando a manutenção e expansão futura.
3. **Camada de Repositório:** Utiliza **Spring Data JPA** para o acesso e manipulação de dados no banco de dados. Os repositórios são abstrações da camada de persistência, otimizando o acesso aos dados e garantindo que a lógica de negócios permaneça desacoplada dos detalhes de armazenamento.

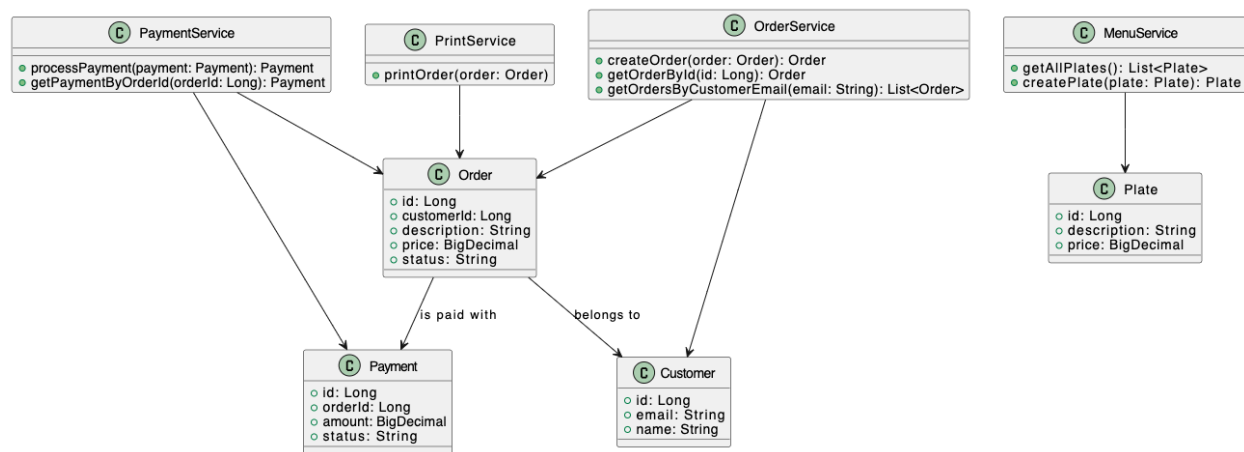
3.1 Modelagem de Domínios e DDD

Para garantir que o sistema fosse facilmente escalável e refletisse adequadamente as regras de negócio, adotei os conceitos do **Domain-Driven Design (DDD)**. O sistema foi modelado em torno de **domínios** e **subdomínios**, cada um representado como um microserviço independente. O isolamento de cada microserviço é garantido por meio de **bounded contexts**, permitindo que cada serviço evolua sem interferir nos outros.

3.1.1 Estrutura dos Domínios

- **OrderService:** Gerencia os pedidos criados pelos clientes. Cada pedido é salvo no banco de dados, e um evento é enviado ao RabbitMQ quando o pedido é criado.
- **PaymentService:** Gerencia os pagamentos dos pedidos. Salva os dados de pagamento e envia mensagens de "pagamento realizado" via RabbitMQ.
- **MenuService:** Gerencia o cardápio do restaurante, permitindo a criação, leitura e remoção de itens de menu.
- **PrintService:** Escuta mensagens de "pagamento realizado" no RabbitMQ e imprime os detalhes do pedido.

3.2 Diagrama de Classes



3.3 Integração com o Frontend (React)

Desenvolvi uma interface de usuário utilizando **React** que permite que os usuários interajam com a aplicação de forma intuitiva. A interface consome as APIs REST para criar novos pedidos.

Order created successfully!

Name

Select Plates

Garlic Bread - \$4.99

Grilled Salmon - \$18.5 Remove

Garlic Bread - \$4.99 Remove

Create Order

My Orders


Order ID	Plates	Total Price	Status
1	Spaghetti Bolognese - \$12.99 Chicken Alfredo - \$14.5	\$27.49	Pending

4. Arquitetura Orientada a Eventos com RabbitMQ

Um dos maiores desafios foi implementar uma **arquitetura orientada a eventos** usando o **RabbitMQ**. A arquitetura orientada a eventos foi crucial para garantir que os microsserviços fossem desacoplados e pudessem se comunicar de forma assíncrona.

Cada vez que um pedido é criado ou pago, um evento é enviado ao RabbitMQ, que garante que os serviços interessados, como o **PrintService**, recebam essas atualizações. Esse modelo de comunicação assíncrona melhorou a escalabilidade e reduziu o acoplamento entre os serviços.

As dificuldades encontradas no desenvolvimento foram principalmente relacionadas com a organização das diferentes queues no RabbitMQ, a correta serialização das mensagens, bem como o rastreamento das transações distribuídas entre microsserviços, o que dificultou o debug de eventos complexos. No entanto, esse problema foi mitigado com o uso de ferramentas de monitoramento de logs.

 RabbitMQ 3.13.7 Erlang 26.2.5.3

Refreshed 2024-09-19 17:4

OverviewConnectionsChannelsExchangesQueues and StreamsAdmin

Page 1 of 1 Filter: ☐ Regex ?

Displaying :

Overview					Messages			Message rates				
Virtual host	Name	Type	Features	State	Ready	Unacked	Total	incoming	deliver	get	ack	
/	order.print.queue	classic			0	0	0					
/	order.queue	classic			0	0	0	0.00/s	0.00/s	0.00/s		
/	order.update.queue	classic			0	0	0					

Add a new queue

Virtual host: /

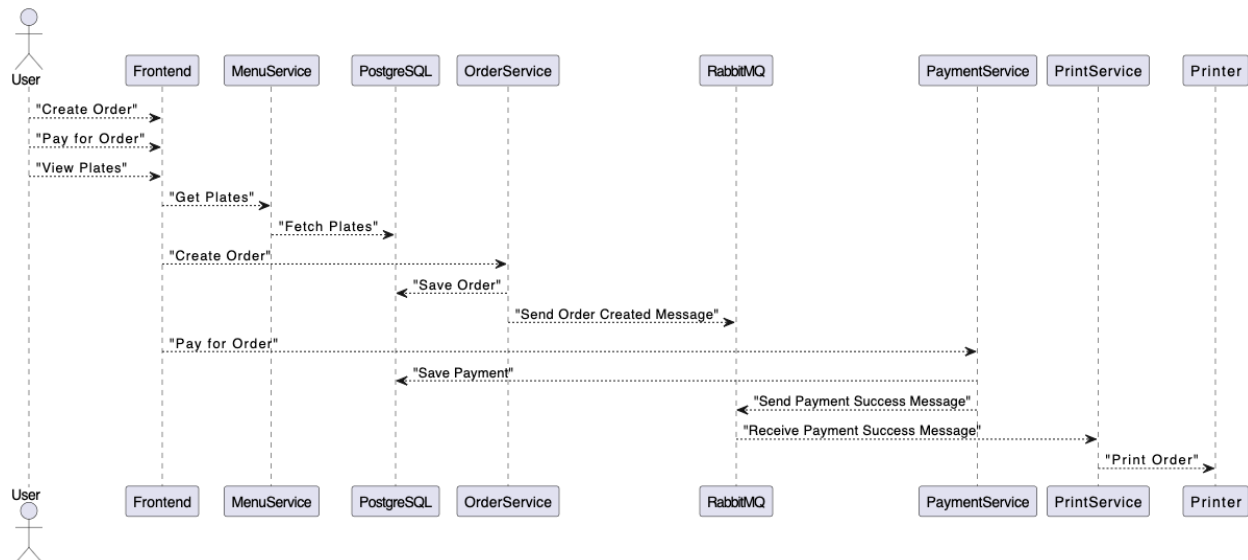
Type: Default for virtual host

Name:

4.1 Vantagens da Arquitetura Orientada a Eventos

- **Escalabilidade:** Os microsserviços podem ser escalados independentemente com base na carga de trabalho, sem afetar outros serviços.
- **Desacoplamento:** Os serviços se comunicam de maneira assíncrona, o que reduz a dependência direta entre eles.
- **Resiliência:** Mensagens são armazenadas no RabbitMQ até serem processadas, garantindo a entrega confiável das informações.

4.2 Diagrama de sequência



A função de cada domínio é bem segregada, o que facilita o desenvolvimento contínuo da aplicação. Permite que o desenvolvimento seja pensado em classes e funções essenciais, sem duplicação de código desnecessária.

5. Gerenciamento de Banco de Dados com Flyway Docker

Neste projeto, utilizamos o Flyway Docker para gerenciar as migrações de banco de dados. Essa abordagem simplifica o processo de migração ao usar um contêiner dedicado do Flyway, que executa os scripts SQL automaticamente durante a inicialização do ambiente Docker Compose.

5.1 Estrutura do Banco de Dados

Optamos por utilizar um único banco de dados PostgreSQL para todo o projeto, visando minimizar o consumo de memória, especialmente durante os testes e o desenvolvimento local. Cada microserviço (userservice, paymentservice, kitchenservice, menuservice e orderservice) opera dentro do seu próprio esquema na mesma instância test_db do PostgreSQL. Essa configuração permite que cada serviço mantenha uma separação lógica dos seus dados, enquanto compartilha o mesmo banco de dados físico.

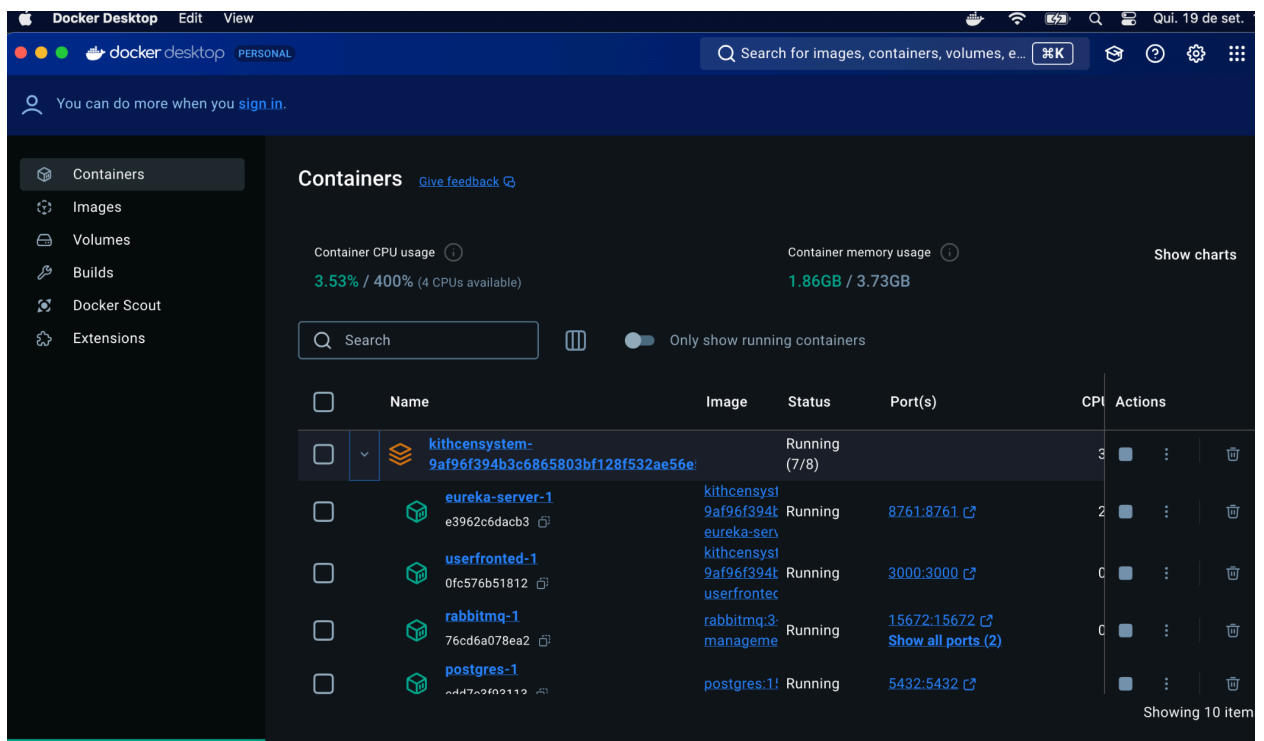
5.2 Execução das Migrações

A imagem Docker do Flyway cuida de executar as migrações de banco de dados. Os arquivos de migração SQL estão localizados na raiz do projeto (por exemplo, V1_CreateAll_Tables.sql). O Flyway executa esses scripts para criar as tabelas e esquemas necessários para cada serviço.

6. Containerização (Docker)

A **containerização** dos microsserviços foi feita utilizando **Docker**, o que facilitou o processo de desenvolvimento e testes locais. Cada microsserviço foi empacotado como um contêiner Docker, permitindo que fossem facilmente implantados em qualquer ambiente.

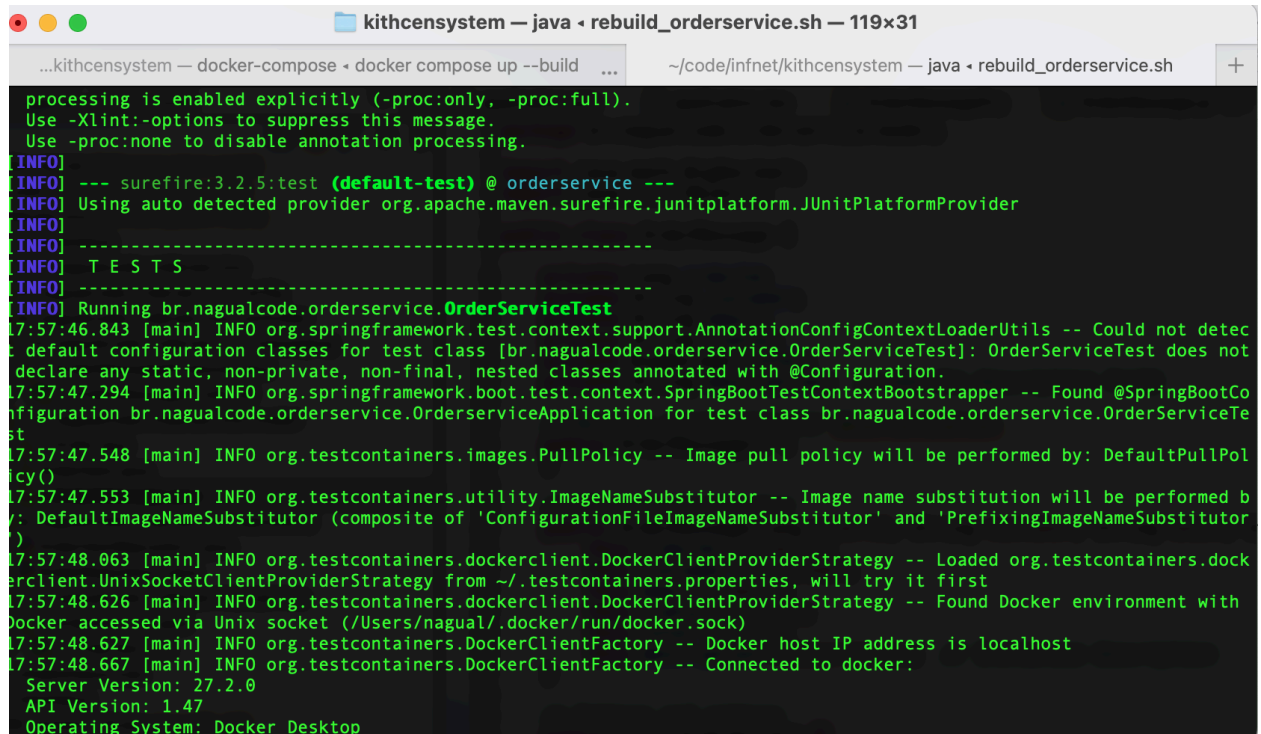
Com o Docker Compose foi possível facilmente organizar a rede virtual interna, e os roteamentos. Também utilizamos o Eureka-Server para a propagação e descoberta dos micro serviços.



7. Testes Abrangentes (JUnit)

Implementei testes unitários e de integração para garantir que cada componente do sistema funcionasse conforme o esperado. Utilizei **JUnit** e **Testcontainers** para simular o ambiente de produção durante os testes, o que aumentou a confiabilidade do sistema antes de sua implantação.

Os testes abrangeram tanto a camada de serviço quanto a comunicação entre os microsserviços, incluindo o envio e recebimento de mensagens via RabbitMQ.

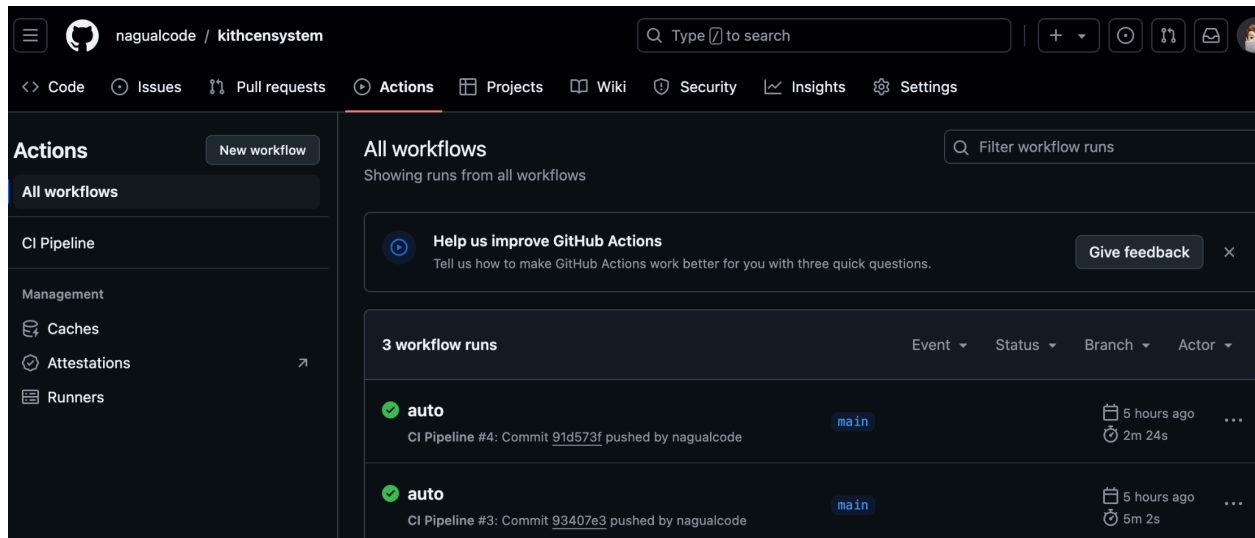


```
kithcensystem — java • rebuild_orderservice.sh — 119x31
...kithcensystem — docker-compose • docker compose up --build ...
~/code/infnet/kithcensystem — java • rebuild_orderservice.sh +

processing is enabled explicitly (-proc:only, -proc:full).
Use -Xlint:options to suppress this message.
Use -proc:none to disable annotation processing.
[INFO] --- surefire:3.2.5:test (default-test) @ orderservice ---
[INFO] Using auto detected provider org.apache.maven.surefire.junitplatform.JUnitPlatformProvider
[INFO] -----
[INFO] T E S T S
[INFO] -----
[INFO] Running br.nagualcode.orderservice.OrderServiceTest
17:57:46.843 [main] INFO org.springframework.test.context.support.AnnotationConfigContextLoaderUtils -- Could not detect
default configuration classes for test class [br.nagualcode.orderservice.OrderServiceTest]: OrderServiceTest does not
declare any static, non-private, non-final, nested classes annotated with @Configuration.
17:57:47.294 [main] INFO org.springframework.boot.test.context.SpringBootTestContextBootstrapper -- Found @SpringBootCo
nfiguration br.nagualcode.orderservice.OrderserviceApplication for test class br.nagualcode.orderservice.OrderServiceTe
st
17:57:47.548 [main] INFO org.testcontainers.images.PullPolicy -- Image pull policy will be performed by: DefaultPullPol
icy()
17:57:47.553 [main] INFO org.testcontainers.utility.ImageNameSubstitutor -- Image name substitution will be performed b
y: DefaultImageNameSubstitutor (composite of 'ConfigurationFileImageNameSubstitutor' and 'PrefixingImageNameSubstitutor
')
17:57:48.063 [main] INFO org.testcontainers.dockerclient.DockerClientProviderStrategy -- Loaded org.testcontainers.dock
erclient.UnixSocketClientProviderStrategy from ~/.testcontainers.properties, will try it first
17:57:48.626 [main] INFO org.testcontainers.dockerclient.DockerClientProviderStrategy -- Found Docker environment with
Docker accessed via Unix socket (/Users/nagual/.docker/run/docker.sock)
17:57:48.627 [main] INFO org.testcontainers.DockerClientFactory -- Docker host IP address is localhost
17:57:48.667 [main] INFO org.testcontainers.DockerClientFactory -- Connected to docker:
  Server Version: 27.2.0
  API Version: 1.47
  Operating System: Docker Desktop
```

8. Github Actions

GitHub Actions é uma poderosa ferramenta para garantir a entrega contínua do projeto, sem que nada deixe de funcionar devido a uma atualização. Em nosso caso isso se dá com a automação de *mvn test* a cada commit para o *main*, com o seguinte workflow:



8. Rodando a aplicação

Enquanto a aplicação ainda se encontra em uma fase conceitual, os testes são feitos com comandos diretos para as endpoints, via cURL. Bem como a simulação dos emails enviados os clientes, e a impressão de ordens na cozinha, aparecem via *system.out.println* no console.

Comando para mudar o status de uma ordem (criada via interface) como paga, simulando o pagamento bem sucedido de um gateway de pagamentos:

```
curl -X PUT "http://localhost:8082/payments/orderID?status=paid"
```

O processamento dos consumidores das mensagens (PaymentService, OrderService, PrintService) pode ser observado no log de cada microserviço na interface do Docker.

kithcensystem-9af96f394b3c6865803bf128f532ae56e58c0d73-

paymentservice-1

345d173a6785kithcensystem-9af96f394b3c6865803bf128f532ae56e58c0d73-8082:8082

STATUSRunning (19 minutes ago)

LogsInspectBind mountsExecFilesStats

yClient : The response status is 200
2024-09-19 19:26:56 2024-09-19T22:26:56.031Z INFO 1 --- [paymentservice] [rap-executor-%d] c.n.d.s.r.aws.ConfigClusterRes
olver : Resolving eureka endpoints via configuration
2024-09-19 19:31:56 2024-09-19T22:31:56.083Z INFO 1 --- [paymentservice] [rap-executor-%d] c.n.d.s.r.aws.ConfigClusterRes
olver : Resolving eureka endpoints via configuration
2024-09-19 19:35:50 -----
2024-09-19 19:35:50 Sending email to: fred@jardimsonoro.com
2024-09-19 19:35:50 Subject: Your Order Details
2024-09-19 19:35:50 Message:
2024-09-19 19:35:50 Hello Fred,
2024-09-19 19:35:50
2024-09-19 19:35:50 Thank you for your order. Here are the details:
2024-09-19 19:35:50 Order ID: 1
2024-09-19 19:35:50 Total Price: 45.97
2024-09-19 19:35:50
2024-09-19 19:35:50
2024-09-19 19:35:50 -----

Simulação de um email de cobrança enviado para o cliente quando inicia um pedido.

kithcensystem-9af96f394b3c6865803bf128f532ae56e58c0d73-

orderservice-1

b8883f4e03efkithcensystem-9af96f394b3c6865803bf128f532ae56e58c0d73-8085:8085

LogsInspectBind mountsExecFilesStats

2024-09-19 19:37:46 2024-09-19T22:37:46.225Z DEBUG 1 --- [ord
nsumer : Storing delivery for consumerTag: 'amq.ctag-2ZgDNM
tags=[[amq.ctag-2ZgDNM3sr01YtbG3WTTdw]], channel=Cached Rab
n: Proxy@5af850f1 Shared Rabbit Connection: SimpleConnection@
8250], acknowledgeMode=AUTO local queue size=0
2024-09-19 19:37:46 2024-09-19T22:37:46.228Z DEBUG 1 --- [ord
nsumer : Received message: (Body:'[B@8a30ca(byte[71])' Mess
ce.model.OrderMessage}, contentType=application/json, content
ENT, priority=0, redelivered=false, receivedExchange=, receiv
mq.ctag-2ZgDNM3sr01YtbG3WTTdw, consumerQueue=order.update.qu
2024-09-19 19:37:46 2024-09-19T22:37:46.290Z DEBUG 1 --- [ord
rAdapter : Processing [GenericMessage [payload=OrderMessage{o
=null}, headers={amqp_receivedDeliveryMode=PERSISTENT, amqp_r
F-8, amqp_deliveryTag=1, amqp_consumerQueue=order.update.queu
fc0fe, amqp_consumerTag=amq.ctag-2ZgDNM3sr01YtbG3WTTdw, amqp
br.nagualcode.paymentservice.model.OrderMessage, timestamp=17
2024-09-19 19:37:46 Order status updated: 1 -> paid

Envio de menssagem quando o microserviço confirma um pagamento

Order ID	Plates	Total Price	Status
1	Spaghetti Bolognese - \$12.99 Caesar Salad - \$7.99 Ribeye Steak - \$24.99	\$45.97	Paid

Atualização na userinterface (via orderservice).

9. Conclusão

KitchenSystem é um exemplo de como uma arquitetura de microsserviços pode ser aplicada de forma eficiente para gerenciar as operações de um restaurante. A escolha de tecnologias como Spring Boot, RabbitMQ e Docker, conforme vimos em aula, garantiu a escalabilidade e a resiliência da aplicação, enquanto os princípios de **DDD** garantiram a clareza e a manutenibilidade do código.

O projeto, agora documentado, está pronto para ser utilizado como base para futuras expansões, permitindo a adição de novas funcionalidades sem comprometer a integridade da solução.

Futuras implementações (ToDo's) incluem: Integração com gateway de pagamentos, integração com driver de impressão, e criação de perfis de usuarios e administradores bem como a implementação do SpringSecurity.

19/Setembro/2024.