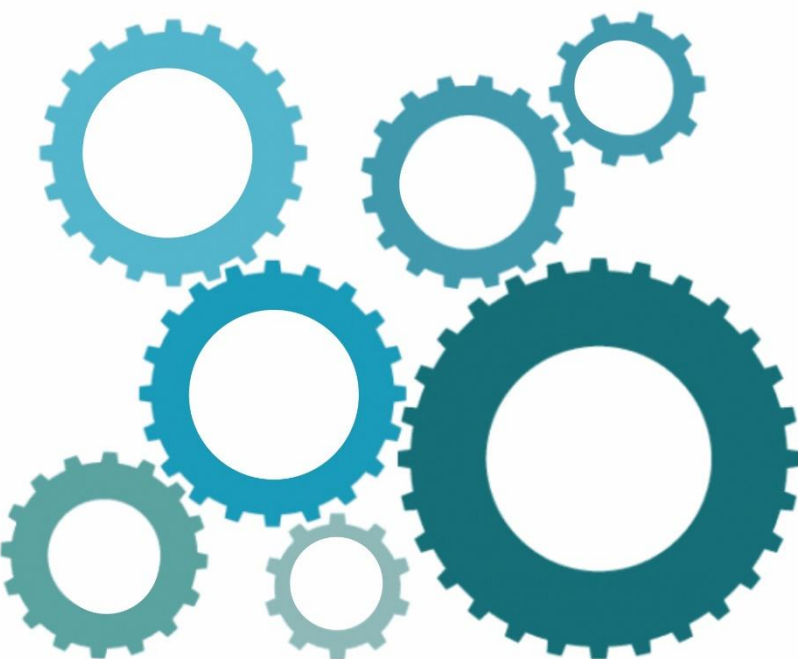








Métodos e Técnicas de Suporte ao Desenvolvimento de Software



DMW
Microservices



David Afonso - 8180011
Michele Freitas - 8180014
Wellington Cunha - 8180012

Índice

Enquadramento teórico	1
Engenharia de Software	1
DDD - Domain Driven Design	1
Bounded Context	2
Aplicação Monolítica e Microsserviços	3
Objetivos	6
Recursos	6
Caso de Estudo - Plataforma de “Loja Online”	7
O Conceito	7
O desenho conceptual	7
Análise do Domínio	7
Definição dos Bounded Contexts	8
Arquitetura técnica	9
Microsserviços	9
Criação da Arquitetura Final	12
UI Inicial	13
UI Final	16
Implementação - Tecnologias	19
 Spring Boot	19
 Postman	20
 Angular	20
 RabbitMQ	20

 Kubernetes.....	21
 GitHub.....	22
Minikube	24
Considerações finais.....	25
Bibliografia	26

Enquadramento teórico

Engenharia de Software

A engenharia de software engloba não só os aspetos tecnológicos da programação de aplicações mas também os vários aspetos sociais com que está relacionada, a conceptualização do funcionamento dos sistemas, os princípios das bases de dados e das redes de computadores, os princípios de gestão, a gestão da equipa, os testes e os ambientes de produção. A engenharia de software é um importante suporte para o desenvolvimento do crescimento da economia mundial eletrónica em que estamos inseridos.

Os processos de desenvolvimento dividem-se em dois tipos funcionais: os clássicos - que visam responder apenas aos requisitos identificados, e os ágeis - que têm a preocupação de avaliar constantemente o que está a ser feito e verificar se está correto.

A tendência tem sido de recorrer aos processos ágeis e incorporar o DevOps que não é mais que a combinação de filosofias culturais, práticas e ferramentas que aumentam a capacidade de uma empresa de distribuir aplicativos e serviços em alta velocidade: otimizando e aperfeiçoando produtos num ritmo mais rápido do que o das empresas que usam processos tradicionais de desenvolvimento de software e gerenciamento de infraestrutura. Essa velocidade permite que as empresas atendam melhor aos seus clientes e compitam de modo mais eficaz no mercado.

DDD - Domain Driven Design

O DDD induz à implementação de um cenário de melhoria contínua, podendo ser uma ferramenta extremamente útil para se desenvolver software de qualidade e que atenda bem as necessidades do cliente.

As técnicas de DDD, que ensinam justamente boas práticas de como modelar seu domínio, além de tornar eficiente a interação entre os vários papéis de pessoas que fazem parte do processo de desenvolvimento de software. DDD pode ser muito útil no trabalho em equipa no desenvolvimento de um sistema complexo.

Domain Driven Design significa Projeto Orientado ao Domínio. Ele veio do título do livro escrito por Eric Evans, dono da Domain Language, uma empresa especializada

em formação e consultoria para desenvolvimento de software. O livro de Evans, boas práticas de programação é um grande catálogo de Padrões, baseados em experiências do autor ao longo de mais de 20 anos desenvolvendo software utilizando técnicas de Orientação a Objetos.

DDD pode ser visto como o regresso da orientação a objetos. Quando se fala em Orientação a Objetos pensa-se logo em classes, heranças, polimorfismo, encapsulamento. Mas a essência da Orientação a Objetos também tem outros elementos como:

- Alinhamento do código com o negócio: o contato dos desenvolvedores com os especialistas do domínio é algo essencial quando se faz DDD;
- Favorecer reutilização: os blocos de construção, facilitam aproveitar um mesmo conceito de domínio ou um mesmo código em vários lugares;
- Mínimo de acoplamento: Com um modelo bem feito, organizado, as várias partes de um sistema interagem sem que haja muita dependência entre módulos ou classes de objetos de conceitos distintos;
- Independência da Tecnologia: DDD não se foca em tecnologia, mas sim em entender as regras de negócio e como elas devem estar refletidas no código e no modelo do domínio.

Para ter um software que atenda perfeitamente a um determinado domínio, é necessário que se estabeleça, em primeiro lugar, uma Linguagem comum, com termos bem definidos, que fazem parte do domínio do negócio e que são usados por todas as pessoas que fazem parte do processo de desenvolvimento de software. Nessa linguagem estão termos que fazem parte das conversas diárias entre especialistas de negócio e equipes de desenvolvimento. Todos devem usar os mesmos termos tanto na linguagem falada quanto no código.

Bounded Context

Usando uma abordagem DDD ajudará a estruturar microsserviços, para que todos os serviços de formulários um ajuste natural para um requisito comercial funcional. Ele pode ajudar a evitar a armadilha de permitir que os limites organizacionais ou opções de tecnologia ditam seu design.

O DDD é uma abordagem para o desenvolvimento de software para necessidades complexas, conectando a implementação a um modelo em evolução.

Antes de escrever qualquer código, tem um panorama geral do sistema a ser criado. O DDD começa com a Modelagem do domínio de negócios e a criação de um modelo de domínio, que é um modelo abstrato do domínio empresarial. Seleciona e organiza os dados de conhecimento do domínio e fornece uma linguagem comum para programadores e especialistas de domínio.

Aplicação Monolítica e Microsserviços

A arquitetura monolítica é a arquitetura de sistema operacional mais comum e antiga, no qual cada componente do S.O. está contido no núcleo do sistema. Ela é uma aplicação formada por vários módulos que são compilados separadamente e depois linkados, formando assim um grande sistema onde os módulos podem interagir.

Ela aumenta a complexidade e o tamanho do código, agrupando tarefas similares em camadas. Dividido em camadas com níveis sobrepostos. Camadas inferiores oferecem funções para camadas superiores. Cada camada comunica-se exclusivamente com as camadas acima e abaixo apenas.

Ela descreve uma única aplicação de software em camadas no qual a interface de usuário e código de acesso aos dados são combinados em um único programa a partir de uma única plataforma. A utilização de Microsserviços é uma abordagem para desenvolver um único aplicativo como um conjunto de pequenos serviços, cada um executando em seu próprio processo e comunicando-se com mecanismos leves, geralmente uma API de recurso HTTP.

Como tudo em desenvolvimento de software, existem vantagens e desvantagens nos sistemas monolíticos.

Por outro lado, temos um sistema cujo deploy é fácil de ser feito, já que o banco de dados facilmente evoluirá junto para todas as funcionalidades e há apenas um ponto onde o deploy precisa ser feito. Além disso, não há duplicidade de código e classes necessárias entre os diferentes módulos, já que todas elas fazem parte da mesma unidade.

Um dos principais pontos negativos é que você tem um grande ponto único de falha, como mostra a figura 1 abaixo, que significa que se houver algum erro no cadastro de funcionários que deixe o sistema fora do ar, isso vai levar junto todo o sistema, incluindo funcionalidades que não possuem nenhuma relação com essa funcionalidade.

Outro ponto negativo é a base de código, que se torna muito extensa, podendo deixar novos membros do projeto menos produtivos por algum tempo, já que a complexidade do código é bem maior.

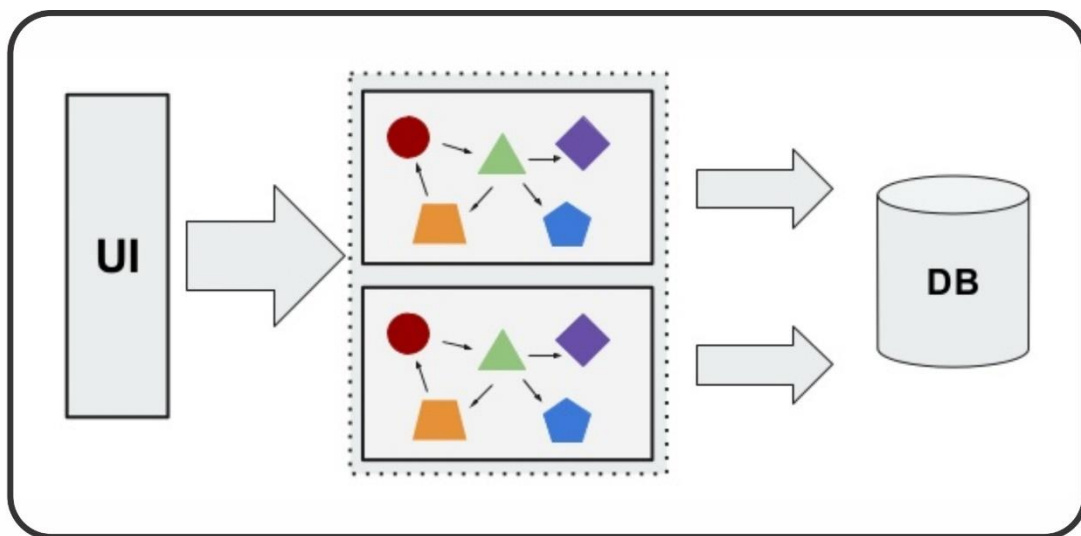


Figura 1 - Exemplo desenho arquitetura monolítica

Esses serviços são criados com base nos recursos de negócios e implementados de maneira independente por um mecanismo de implantação totalmente automatizado.

Há um mínimo de gerenciamento centralizado desses serviços, que pode ser escrito em diferentes linguagens de programação e usar diferentes tecnologias de armazenamento de dados.

Microsserviço é um estilo arquitetônico, o que significa que é uma especialização de tipos de elementos e relações, juntamente com restrições e como elas podem ser usadas.

O principal benefício dos microsserviços é a agilidade, reduzindo muito o tempo de produção por exemplo numa aplicação muito grande. Permitindo dividir o aplicativo num conjunto de componentes desacoplados, como mostra a figura 2, que fornecem serviços definidos (meios definidos com uma interface ou API conhecida).

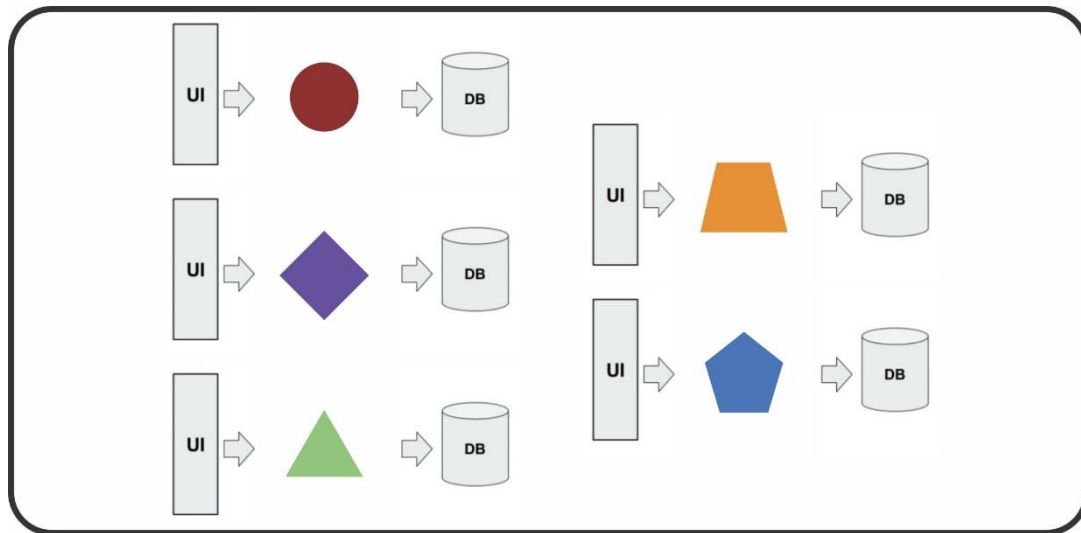


Figura 2 - Exemplo desenho arquitetura microserviços

Com isso os componentes comunicam entre si através de um qualquer protocolo escolhido, geralmente REST, mas não necessariamente. Podendo os componentes usar linguagens e tecnologias que quiserem, sendo desenvolvidos, lançados e implantados de forma independente de modo que a orquestração de toda a aplicação seja reduzida ao mínimo.

Objetivos

- Conceber a arquitetura técnica de um projeto segundo o paradigma microsserviços;
- Desenvolver e implementar uma solução tecnológica de acordo com as práticas e padrões associados a aplicações orientadas a microsserviços;
- Argumentar decisões de arquitetura e justificar opções de implementação no contexto de desenvolvimento de arquitetura baseadas em microsserviços.

Recursos

- Minikube
- Kubernetes
- GIT Server (qualquer)
- Google Cloud Platform (ou alternativa semelhante disponível publicamente)
- Opções para o desenvolvimento dos microsserviços (Vert.x, SeedStack, Spring Boot, JHypster, etc)

Caso de Estudo

Plataforma de “Loja Online”

O Conceito

Para esse projeto com intenção de atingir os objetivos propostos, desenvolvemos uma loja online.

Atualmente, a internet é um canal de vendas tão consolidado quanto as lojas físicas. Ao fazer compras online, os consumidores também desfrutam da comodidade e da economia de tempo. Grande parte das pessoas que compram pela web já conhecem o produto que estão comprando. Por isso, não sentem a necessidade de ter um contato físico com o artigo.

Ao comprar em num comércio eletrônico, o cliente não precisa enfrentar trânsito para ir até à loja física, nem se preocupar com estacionamento.

O processo é fácil e rápido. Inicia-se com a vontade de um cliente em comprar um produto. Acendendo à plataforma da loja, navega pela lista de produtos e seleciona quais itens deseja comprar. Após escolher os produtos pretendidos, clica no botão “Checkout to Cart”, sendo então redirecionado para a página do carrinho para finalizar suas compras.

A compra despoleta a confirmação dos produtos pretendidos, detalhando a identificação da compra, o valor de cada item, a data da compra e o valor total.

Após a compra o cliente finaliza no botão “Go to Order” e já tem o sucesso do seu pedido.

O desenho conceptual

Análise do Domínio

Tendo como ponto de partida a aplicação de uma metodologia de desenvolvimento de software orientada ao Domínio segundo o DDD - Domain driven Design, o primeiro passo é a definição do Domínio. Assim sendo, como o âmbito principal da plataforma é a disponibilização de uma lista de produtos que podem ser comprados, definimos que o domínio principal é a compra de produtos.

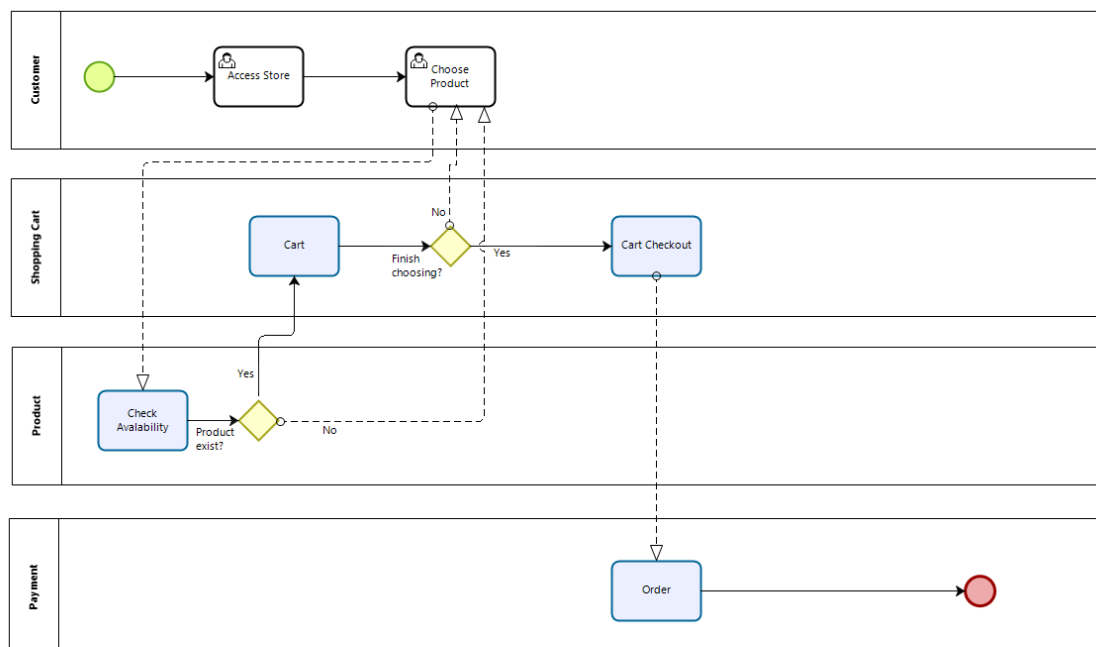
O passo seguinte passa pela definição dos bounded contexts.

Definição dos Bounded Contexts

Definimos que a plataforma de “Loja Online” permite:

- Acesso a página principal da loja;
- Consulta da lista de Produtos disponíveis;
- Possibilidade de escolha de produtos;
- Aquisição dos produtos;
- Finalização da compra

Utilizamos o Bizagi para fazer o bpm da arquitetura inicial do projeto. Nele definimos os diagramas do passo a passo do ciclo do nosso projeto, desde da fase do cliente aceder a loja online, acessar a lista de produtos e escolher todos os produtos desejados. Após terminar de seleccionar e clicar no botão checkout que redireciona para a página do carrinho de compra, onde confere os itens comprados e finaliza a compra no botão go to order.

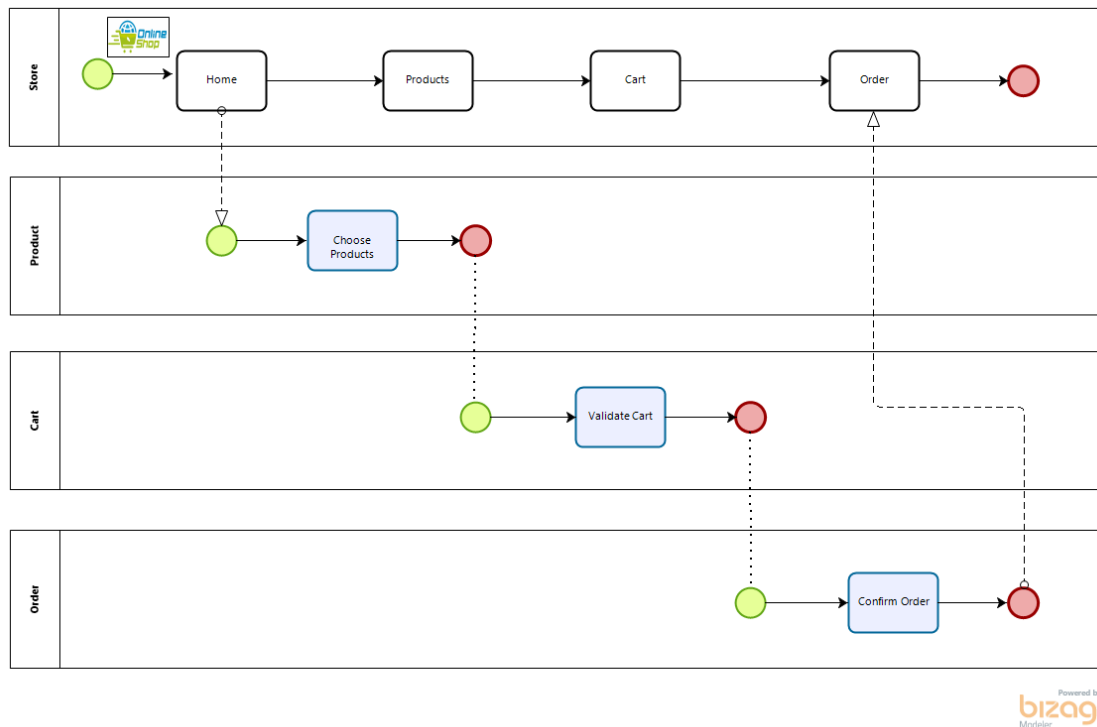


Powered by
bizagi
Modeler

Figura 5– Arquitetura inicial do trabalho

Arquitetura técnica

A loja está disponível numa solução tecnológica que permite o seu desenvolvimento continuado e sua integração sem tempos de indisponibilidade, como mostramos em nossa arquitetura abaixo .



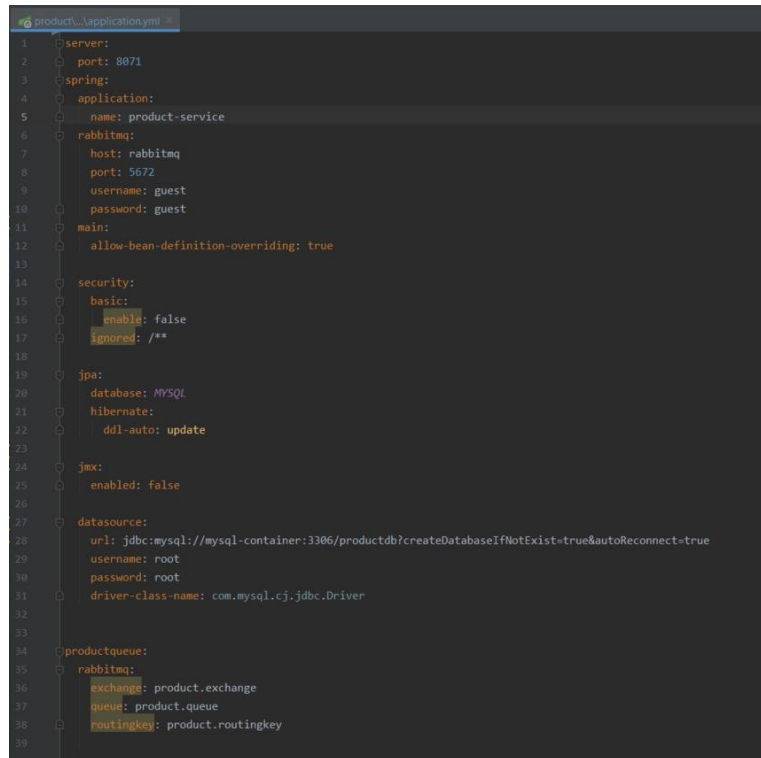
Microserviços

Existe três microserviços diferenciados:

- **Products:** Carrega a lista de todos os produtos disponíveis que vem do microserviço do backend com as suas especificações (nome, preço e a imagem correspondente).
- **Shopping cart:** O array dos produtos comprados é enviado para product microservice (backend) e de lá foi enviado pelo rabbitmq para a productqueue.rabbitmq.queue = product.queuePossui gerando no cart a identificação da compra, a data que foi realizada, os itens selecionados, o valor de cada um e o total da compra.

- Order: Envia o valor ou outra informação que decidir enviar, como por exemplo o nome do usuário (não deu tempo de implementar) para o backend (microservice cart) e então é enviado para o microservice order.

Todos são independentes e cada um com seu banco de dados separado



```
1 server:
2   port: 8071
3 spring:
4   application:
5     name: product-service
6   rabbitmq:
7     host: rabbitmq
8     port: 5672
9     username: guest
10    password: guest
11  main:
12    allow-bean-definition-overriding: true
13
14  security:
15    basic:
16      enable: false
17      ignored: /**
18
19  jpa:
20    database: MYSQL
21    hibernate:
22      ddl-auto: update
23
24  jmx:
25    enabled: false
26
27  datasource:
28    url: jdbc:mysql://mysql-container:3306/productdb?createDatabaseIfNotExist=true&autoReconnect=true
29    username: root
30    password: root
31    driver-class-name: com.mysql.cj.jdbc.Driver
32
33
34  productqueue:
35    rabbitmq:
36      exchange: product.exchange
37      queue: product.queue
38      routingkey: product.routingkey
39
```

Figura 3 – Demonstração do application.yml de microservice product

```
1 server:
2   port: 8072
3   spring:
4     application:
5       name: cart-service
6     rabbitmq:
7       host: rabbitmq
8       port: 5672
9       username: guest
10      password: guest
11    main:
12      allow-bean-definition-overriding: true
13    security:
14      basic:
15        enable: false
16        ignored: /**
17
18    jpa:
19      database: MySQL
20      hibernate:
21        ddl-auto: update
22    jmx:
23      enabled: false
24    datasource:
25      url: jdbc:mysql://mysql-container:3306/cartdb?createDatabaseIfNotExist=true&autoReconnect=true
26      username: root
27      password: root
28      driver-class-name: com.mysql.cj.jdbc.Driver
29
30    cartqueue:
31      rabbitmq:
32        exchange: cart.exchange
33        queue: cart.queue
34        routingkey: cart.routingkey
35
36    productqueue:
37      rabbitmq:
38        exchange: product.exchange
39        queue: product.queue
40        routingkey: product.routingkey
```

Figura 4 – Demonstração do application.yml de microservice cart

```
1 server:
2   port: 8073
3   spring:
4     application:
5       name: order-service
6     rabbitmq:
7       host: rabbitmq
8       port: 5672
9       username: guest
10      password: guest
11    main:
12      allow-bean-definition-overriding: true
13    security:
14      basic:
15        enable: false
16        ignored: /**
17
18    jpa:
19      database: MySQL
20      hibernate:
21        ddl-auto: update
22    jmx:
23      enabled: false
24    datasource:
25      url: jdbc:mysql://mysql-container:3306/orderdb?createDatabaseIfNotExist=true&autoReconnect=true
26      username: root
27      password: root
28      driver-class-name: com.mysql.cj.jdbc.Driver
29
30    orderqueue:
31      rabbitmq:
32        exchange: order.exchange
33        queue: order.queue
34        routingkey: order.routingkey
35
36    cartqueue:
37      rabbitmq:
38        exchange: cart.exchange
39        queue: cart.queue
40        routingkey: cart.routingkey
```

Figura 5 – Demonstração application.yml de microservice order

Criação da Arquitetura Final

A arquitetura final, como apresentado na figura abaixo, abordamos todas as tecnologias envolvidas no projeto na criação dos microserviços, desde do backend que utilizamos o spring boot até o front end com o angular, também a base de dados MySQL e o RabbitMQ como servidor mensageiro para garantir a comunicação entre os microserviço. Também o kubernetes para gerenciamento dos nossos containers.

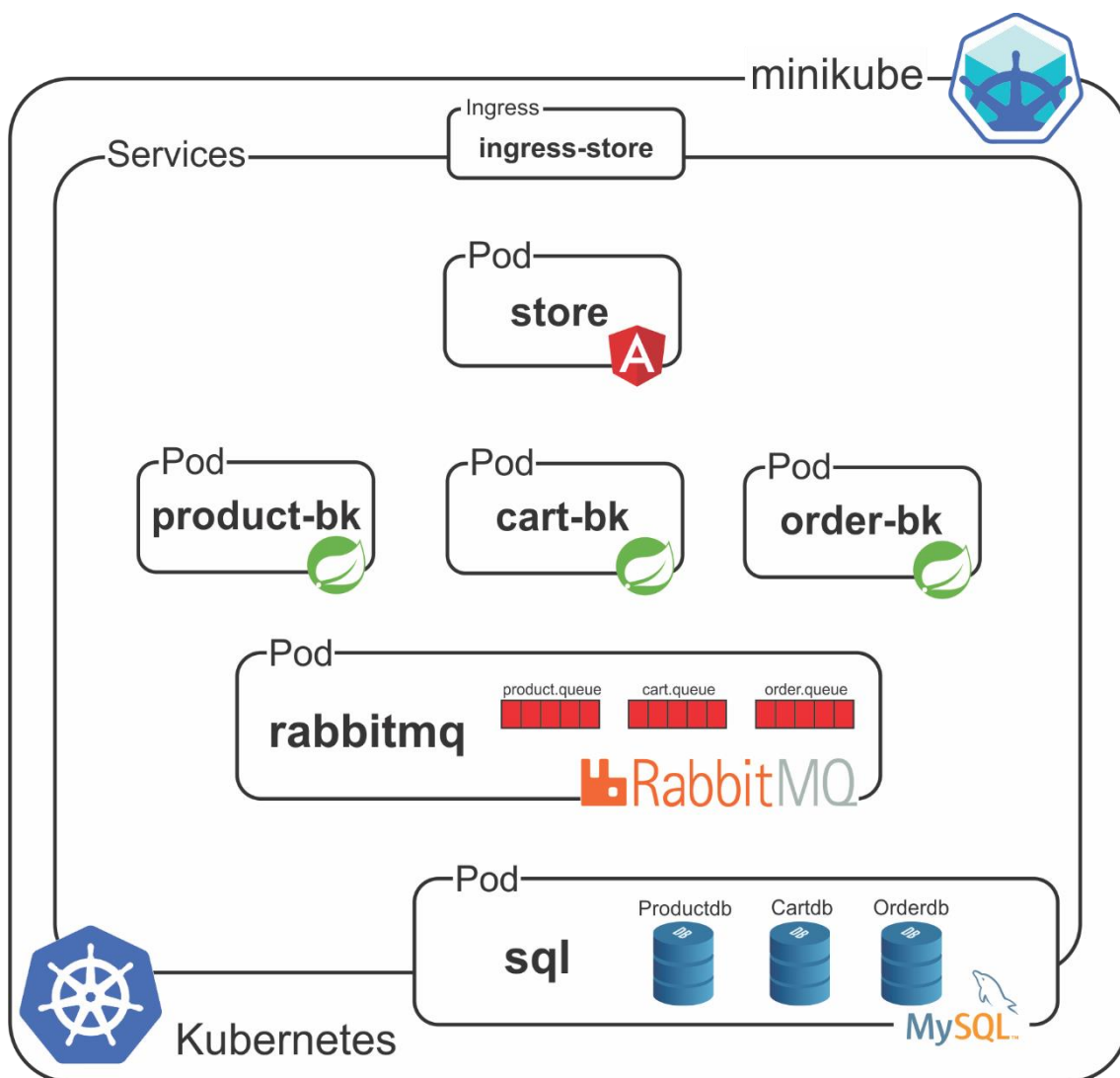


Figura 6 – Arquitetura final do projeto

UI Inicial

Para consumir a API do spring boot inicialmente fizemos o estudo para ver as possibilidades e aprender como deveríamos fazer com o angular. Realizamos pesquisas de diversos artigos para entender o uso do HttpClient.

Criamos um projeto angular, e fizemos CRUD para nos ajudar de forma simples e básica a manipular os produtos da loja, também criaremos uma API REST fake para simular o back-end, assim focamos no uso do HttpClient. Usamos algumas tecnologias como o Angular CLI, Node.js e o Json-server.

O HttpClient foi usado para fazer a comunicação entre cliente e servidor usando o protocolo HTTP. Ou seja, para consumir dados de uma API externa o HttpClient facilitou essa comunicação, através de muitos métodos disponíveis, como: post, get, put, delete.

Os Benefícios do HttpClient usa a interface XMLHttpRequest que também suporta a navegadores antigos, disponibiliza benefícios, como: Solicitações de request e response interceptadas, Manipulação de erros simplificada, Suporte a api Observable, APIs e tratamentos de erros.

O projeto foi criado usando o angular CLI, instalado em nossa máquina, também instalamos o Node.js com o seguinte comando:

```
sudo npm install -g @angular/cli
```

Para criar o projeto angular 8 com o angular CLI e o Node.js instalado, criamos o nosso projeto angular usando o CLI, com o comando abaixo:

```
ng new angular-http
```

Assim criamos o projeto com os módulos NPM necessários. Após isso, rodamos o projeto angular 8 para verificar se tudo foi criado corretamente com o comando:

```
ng serve --open
```

Com esse comando rodamos o nosso projeto angular, abaixo figura 6.

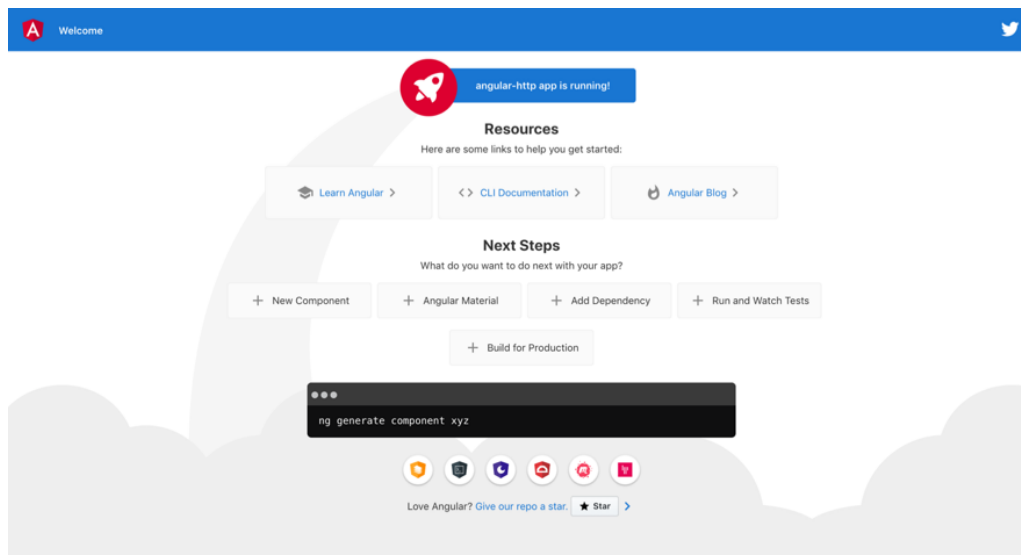


Figura 6 - Bounded contexts

Após criar o projeto, criamos API REST fake para simular o uso do HttpClient. Instamos o json-server, com o seguinte comando:

```
sudo npm install -g json-server
```

Em seguida criamos o db.json com nome products dentro da pasta assets do angular e incluímos o seguinte json, figura 7:

```
dbjson x
src > assets > dbjson > ...
1  {}
2  "products": [
3    {
4      "id": 1,
5      "productName": "TV Set",
6      "price": 300,
7      "pictureUrl": "http://www.pngmart.com/files/1/TV-Transparent-PNG.png"
8    },
9    {
10     "id": 2,
11     "productName": "Game Console",
12     "price": 200,
13     "pictureUrl": "http://www.pngall.com/wp-content/uploads/2/Console-PNG-Transparent-HD-Photo.png"
14   },
15   {
16     "id": 3,
17     "productName": "Sofa",
18     "price": 100,
19     "pictureUrl": "http://www.pngall.com/wp-content/uploads/4/Luxury-Couch-PNG-Free-Image.png"
20   },
21   {
```

Figura 7 – Json com lista de produtos para teste

Rodamos o json-server, figura 8, para simular nossa API REST com um novo terminal e na raiz do projeto executando o seguinte comando:

```
json-server --watch src/assets/data/db.json
```

```
C:\Users\Michele\Documents\estg\FRONT GIT2\productUI>json-server --watch src/assets/db.json

\{^_^}/ hi!

Loading src/assets/db.json
Done

Resources
http://localhost:3000/products

Home
http://localhost:3000

Type s + enter at any time to create a snapshot of the database
Watching...
```

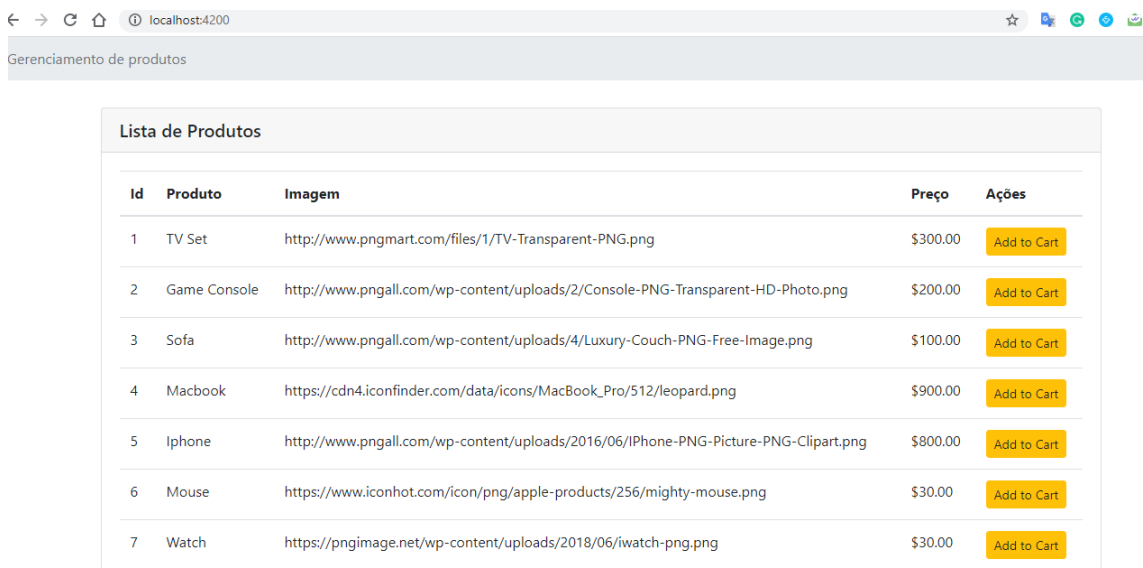
Figura 8– Json Server

Com isso nossa API REST fake ficou exposta no endereço: <http://localhost:3000>.

Configuramos nosso HTTPClient no nosso `app.module.ts` para poder usá-lo adicionando o módulo `HttpClientModule`.

Depois adicionamos os módulos, interface para exibir a lista dos nossos produtos e o serviço responsável pelas requisições. Usamos o `HTTPClient` para criar o método para obter todos os produtos.

Assim que acessamos a tela da aplicação, figura 9, o método `onOnit()` foi disparado chamando o método `getProducts()` exibindo a listagem de produtos do nosso serviço `ProductService`.



Lista de Produtos				
Id	Produto	Imagem	Preço	Ações
1	TV Set	http://www.pngmart.com/files/1/TV-Transparent-PNG.png	\$300.00	<button>Add to Cart</button>
2	Game Console	http://www.pngall.com/wp-content/uploads/2/Console-PNG-Transparent-HD-Photo.png	\$200.00	<button>Add to Cart</button>
3	Sofa	http://www.pngall.com/wp-content/uploads/4/Luxury-Couch-PNG-Free-Image.png	\$100.00	<button>Add to Cart</button>
4	Macbook	https://cdn4.iconfinder.com/data/icons/MacBook_Pro/512/leopard.png	\$900.00	<button>Add to Cart</button>
5	Iphone	http://www.pngall.com/wp-content/uploads/2016/06/iPhone-PNG-Picture-PNG-Clipart.png	\$800.00	<button>Add to Cart</button>
6	Mouse	https://www.iconhot.com/icon/png/apple-products/256/mighty-mouse.png	\$30.00	<button>Add to Cart</button>
7	Watch	https://pngimage.net/wp-content/uploads/2018/06/iwatch-png.png	\$30.00	<button>Add to Cart</button>

Figura 9 – Tela da UI inicial da aplicação

UI Final

Após o entendimento como consumir nossa API, fizemos as adaptações de acordo com as URLs dos nossos microserviços definidas no spring boot e foi possível chamar as informações desenvolvidas no backend para o frontend.

Melhoramos o layout da nossa interface de apresentação e adicionamos rotas ao projeto do angular.

Temos a página home que exibe a apresentação da loja Online Shop, com informações para acessar a aba products, como mostra a figura 10.

Na parte do products temos uma lista de produtos disponíveis na loja, figura 11, com nome, valor e preço. Com um botão add to cart, para o cliente adicionar as compras desejadas.

No final das compras ao clicar em checkout to Cart, ele é redirecionado para a aba Cart, onde apresentamos o Purchase Summary com os itens comprados, os valores de cada item, a data, o id identificador da compra e o Total.

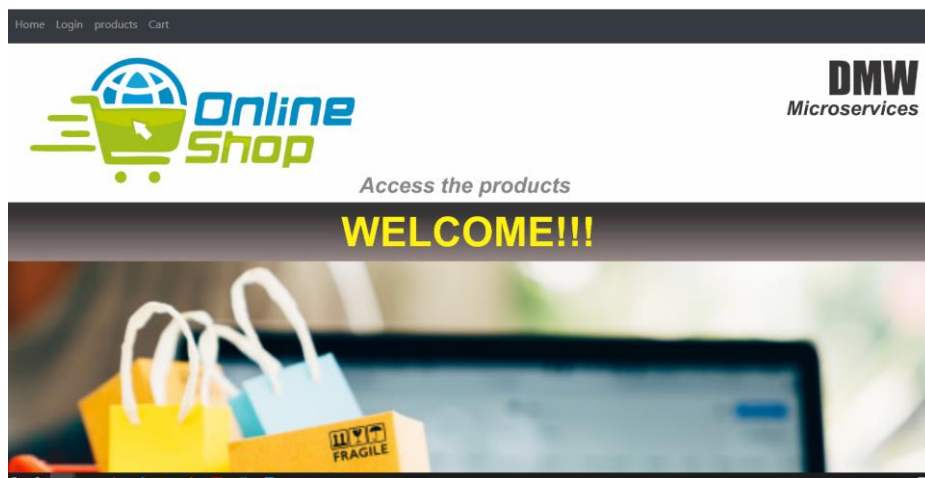


Figura 30 - UI Final Home

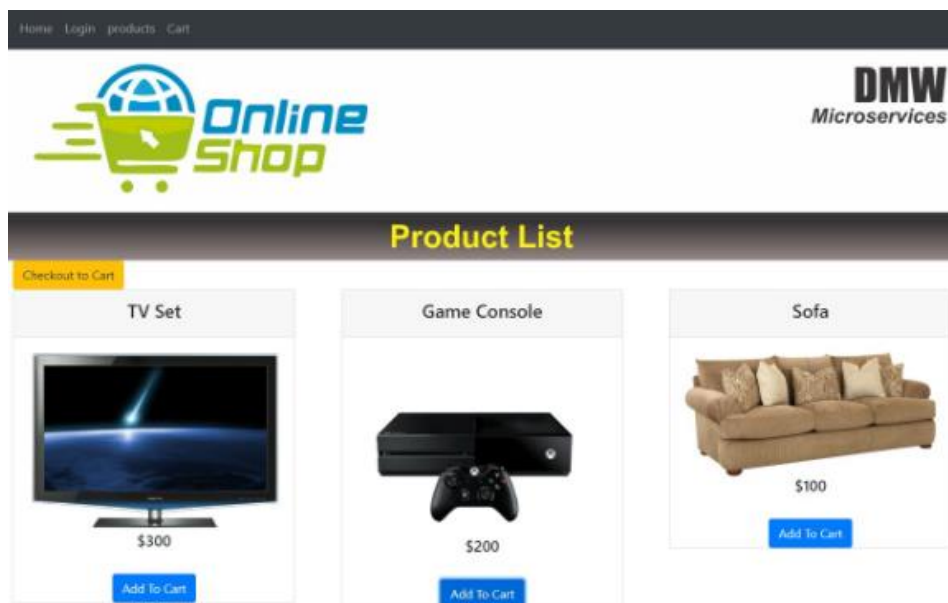


Figura 11 – UI Final Products

A figura 12 exemplifica o funcionamento do "product" para "cart", o microserviço order também tem um listener consumer que ao ser alertado de dado na queue acima, pega o dado e persiste na base de dados.

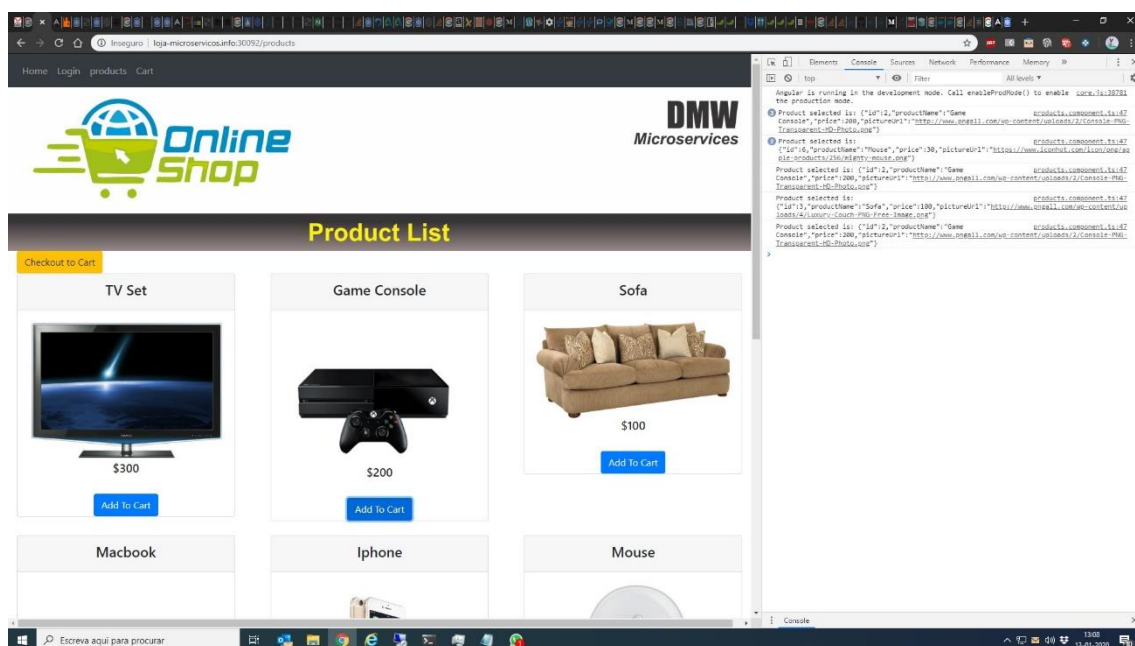


Figura 12 - UI Final Cart

Na figura 13, demonstramos um exemplo do funcionamento da nossa aplicação com o rabbitMQ acabando de vir da lista de produtos depois de fazer checkout. No console é possível ver os produtos que foram selecionados na tela de produtos. Ao clicar

em checkout, o array de produtos foi enviado para product microservice (backend) e de lá foi enviado pelo rabbitmq para a productqueue.rabbitmq.queue = product.queue

Então o listener consumer do rabbitmq no cart service que está escutando a queue acima, vai buscar os dados assim que for colocada lá e no mesmo botão checkout também houve o roteamento para a página cart e está por estar configurada pra ir buscar os dados, alimenta-os logo que é carregada.

O mesmo botão também roteia a página na user interface para order e lá também está configurada para buscar os dados no backend e carregá-los nos seus campos.

O botão Go to Order envia o valor ou outra informação que decidir enviar, como por exemplo o nome do usuário (não deu tempo de implementar) para o backend (microservice cart) e então é enviado para o microservice order por meio do rabbitmq para cartqueue.rabbitmq.queue = cart.queue.

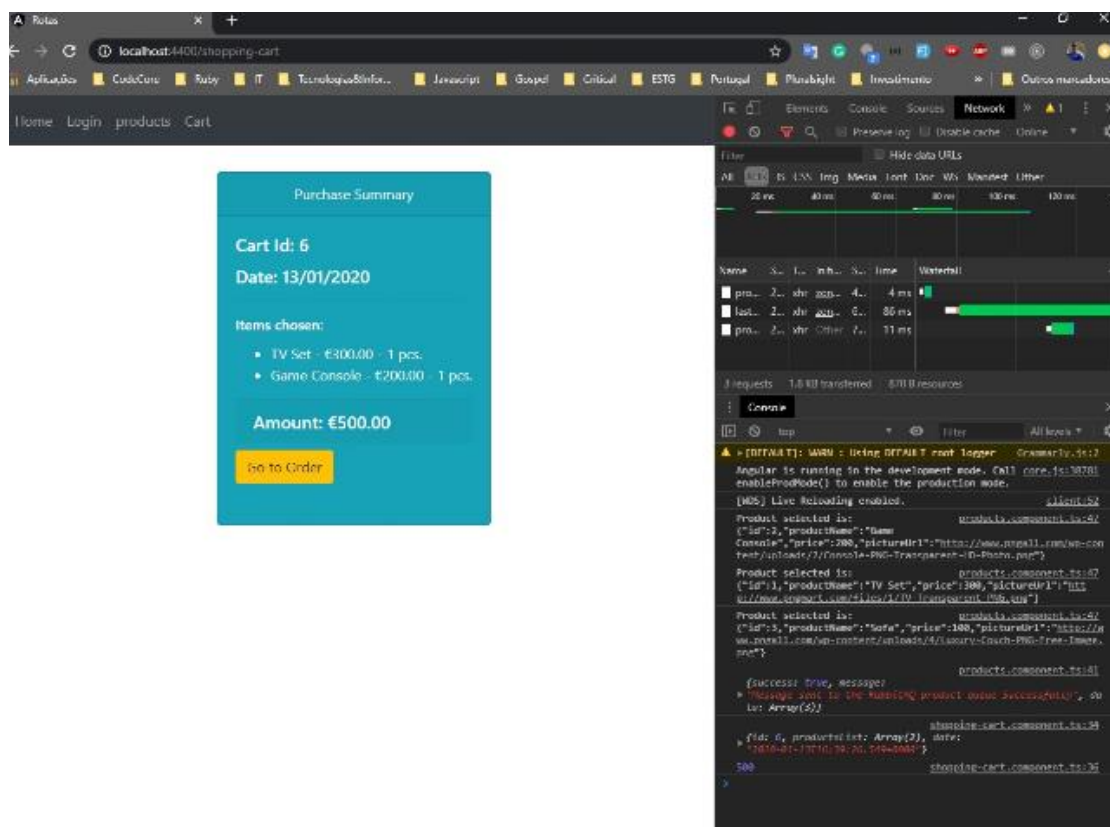


Figura 13 - UI Final Go to Order

Implementação - Tecnologias



Figura 14 – Algumas tecnologias utilizadas

Spring Boot

É um framework que simplifica o desenvolvimento de aplicações em Java, garantindo uma versão do sistema pronto para produção sem a necessidade de fazer inúmeras configurações, otimizações e facilitando o processo de configuração e publicação das aplicações.

Através da escolha dos módulos com os starters que inclui no pom.xml do projeto. Eles, basicamente, são dependências que agrupam outras dependências. Apesar do Spring Boot, através da convenção, já deixar tudo configurado, nada impede que se crie customizações caso sejam necessárias.

Permite ter uma aplicação rodando em produção rapidamente, além de seguir as melhores práticas de design, e com configurações já otimizadas. Ele oferece uma série de templates, ferramentas previamente configuradas, segue convenções ao invés de configurações, ou seja, fornece a maioria dos recursos necessários já prontos para utilização. Para criar projeto é possível utilizar o Spring Boot STS ou outra IDE como o IntelliJ.



No trabalho uma parte do grupo estava utilizando para criar o projeto a IDE do Spring STS no começo do desenvolvimento do backend, mas depois com as divisões das tarefas, definimos migrar para o IntelliJ para todos usarem a mesma plataforma de trabalho.



O Postman é uma ferramenta independente que possui o objetivo de testar serviços RESTful (Web APIs) através do envio de requisições HTTP e da análise do seu retorno.

Com ele é possível consumir facilmente serviços locais e na internet, enviando dados e efetuando testes sobre as respostas das requisições.

No trabalho usamos o postman para auxiliar nos testes do projeto, fazer chamadas Rest para verificar se os seus endpoints estavam reagindo corretamente. Com isso, analisamos o funcionamento dos serviços externos e deixamos ele interagir com nossa API de acordo com as requisições GET, POST, PUT e DELETE.



Para desenvolvimento da user interface do projeto escolhemos o framework Angular e seus componentes, que além de abstrair e facilitar o uso de recursos importantes, como chamadas a APIs REST e controle do fluxo de páginas.

Utilizamos as rotas para o utilizador poder interagir com os serviços da página, desde de acessar a página principal, a lista com os produtos disponível e fazer as compras.



É um message broker, implementado para suportar as mensagens no protocolo denominado Advanced Message Queuing Protocol (AMQP).

O RabbitMQ possibilita lidar com o tráfego de mensagens de forma rápida e confiável, além de ser compatível com diversas linguagens de programação, possuir interface de administração nativa e ser multiplataforma.

Dentre as suas aplicabilidades, estão possibilitar a garantia de assincronicidade entre aplicações, diminuir o acoplamento entre aplicações, distribuir alertas, controlar fila de trabalhos em background.

A dependência do RabbitMQ para o Spring Boot é o `spring-boot-starter-amqp`, que apesar do nome genérico traz funcionalidades do RabbitMQ.

Escolhemos o RabbitMQ para fazer o gerenciamento das nossas filas pois é open source, escrito em Erlang (confiável), roda em diversos sistemas operacionais e foi compatível com a linguagem utilizada no projeto.

Rabbit possui vários plugins. Tem um painel de gerenciamento que ajudou muito durante o trabalho para controlar as filas, também apresenta um bom desempenho e com possibilidade de configurações para atender alta escalabilidade e por ser o centro de um sistema de microsserviços responsivos.



O Kubernetes é um orquestrador de containers. No Kubernetes temos os pods, onde conseguimos guardar os nossos containers, assim sendo, poderemos ter um ou mais containers dentro de um pod. O mais interessante é que, dentro dos nossos pods poderemos ter, além dos containers, várias configurações, para que se comportem da maneira que desejamos.

Imaginando um cluster, uma central master de kubernetes. Embaixo do Kubernetes temos diversas máquinas. Não sabemos que existam essas máquinas, quando estão embaixo do Kubernetes, ou seja, sabemos somente a quantidade de CPU que temos disponível em nosso cluster. Também, sabemos a o quanto de memória temos disponível. A partir disso, todas as vezes que fizermos um deploy de uma aplicação, o Kubernetes tomará conta de tudo e, somente, subiremos os nossos pods.

Usamos o minikube, o que nos permite executar o Kubernetes localmente. Alguns exemplos nas figuras abaixo 15 e 16.

```
FROM openjdk:11-jdk-alpine
VOLUME /tmp
ARG JAR_FILE
ADD ${JAR_FILE} app.jar
ENTRYPOINT ["java","-Djava.security.egd=file:./target","-jar","/spring-boot-amqp-consumer-javainuse-0.0.1-SNAPSHOT.jar"]
```

Figura 15 - Dockerfile_minikube

```
FROM openjdk:11
COPY ./target/spring-boot-amqp-consumer-javainuse-0.0.1-SNAPSHOT.jar spring-boot-amqp-consumer-javainuse-0.0.1-SNAPSHOT.jar
CMD ["java","-jar","spring-boot-amqp-consumer-javainuse-0.0.1-SNAPSHOT.jar"]
```

Figura 16 - Dockerfile_minikube



Por ser um site e serviço que fornece GIT gratuito que é um sistema de controle de versão distribuído e um sistema de gerenciamento de código fonte.

Como o desenvolvimento do nosso projeto conteve muitas etapas, chegando em um ponto para o grupo colaborar e acompanhar criamos repositórios para ter um histórico do que foi feito e poder até reverter em caso de erros.

Problem	Solution
<p>The bean 'rabbitTemplate', defined in class path resource [org/springframework/boot/autoconfigure/amqp/RabbitAutoConfiguration\$RabbitTemplateConfiguration.class], could not be registered. A bean with that name has already been defined in class path resource [com/example/springbootstarteramqp/config/RabbitMQConfig.class] and overriding is disabled.</p>	<p>Application.properties</p> <pre>spring.main.allow-bean-definition-overriding=true</pre> <p>class SpringBootStarterAmqpJavainuseApplication</p> <pre>@SpringBootApplication(exclude = {DataSourceAutoConfiguration.class, XADataSourceAutoConfiguration.class})</pre>

Figura 17 - Alguns dos problemas e solução que tivemos durante o trabalho

Minikube

```

Microsoft Windows [Version 10.0.18363.535]
(c) 2019 Microsoft Corporation. Todos os direitos reservados.

C:\Users\david.afonso>kubecttl get all

NAME                READY   STATUS    RESTARTS   AGE
pod/cart-bk          1/1     Running   4           113m
pod/order-bk          1/1     Running   0           113m
pod/product-bk        1/1     Running   0           113m
pod/rabbitmq          1/1     Running   0           145m
pod/sql               1/1     Running   0           145m
pod/store-6597dc9bfc-hw98p  1/1     Running   0           14m
pod/store-6597dc9bfc-t2fqw  1/1     Running   0           13m

NAME                TYPE          CLUSTER-IP   EXTERNAL-IP   PORT(S)          AGE
service/cartbk      ClusterIP     10.105.245.177 <none>        8072/TCP          113m
service/cartbk-ext  NodePort      10.104.86.33  <none>        8072:30073/TCP    113m
service/kubernetes  ClusterIP     10.96.0.1     <none>        443/TCP           148m
service/message-queue-external  NodePort      10.97.142.253 <none>        15672:30072/TCP   145m
service/mysql-container  ClusterIP     10.99.145.35  <none>        3306/TCP          145m
service/mysql-ext     NodePort      10.104.10.133 <none>        3306:30306/TCP    145m
service/order-ext     NodePort      10.110.83.147 <none>        8073:30074/TCP    112m
service/orderbk       ClusterIP     10.108.223.178 <none>        8073/TCP          113m
service/product-ui    ClusterIP     10.104.65.60  <none>        8080/TCP          144m
service/productbk     ClusterIP     10.101.115.147 <none>        8071/TCP          113m
service/productbk-ext  NodePort      10.111.117.39 <none>        8071:30071/TCP    113m
service/productui-ext  NodePort      10.104.198.208 <none>        80:30091/TCP,8080:30092/TCP 144m
service/rabbitmq      ClusterIP     10.108.81.6   <none>        5672/TCP,15672/TCP 145m

NAME                READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/store  2/2     2             2           145m

NAME                DESIRED   CURRENT   READY   AGE
replicaset.apps/store-6597dc9bfc  2         2         2       14m
replicaset.apps/store-f49bb9648   0         0         0       144m

C:\Users\david.afonso>kubecttl get ingress

NAME        HOSTS                ADDRESS        PORTS    AGE
ingress-store  loja-microservicos.info  172.17.30.167  80       145m

C:\Users\david.afonso>minikube ip
172.17.30.167

C:\Users\david.afonso>

```

Considerações finais

No desenvolvimento desse projeto foi possível adquirir novas competências no paradigma de desenvolvimento, mas também novas experiências aprendidas na utilização de ferramentas, frameworks, enfim, tecnologias que colaboram muito na implementação da arquitetura com microsserviços.

Especialmente sobre o projeto realizado, tivemos diversos desafios, muito trabalho e graças a Deus superação nos obstáculos que surgiram para obter sucesso nos deveres que foram cumpridos.

Descobrimos que para a equipe trabalhar corretamente é preciso todos estarem conectados na forma como estão a desenvolver e alinhar todas as ferramentas.

Apesar de o desenvolvimento com microsserviços ser mais complexo, entendemos que para a evolução na criação de aplicações e para manutenção é essencial pois são independentes e muito mais rápidas.

Bibliografia

- <https://www.rabbitmq.com/getstarted.html>
- <https://microservices.io/patterns/>
- <https://spring.io/guides/gs/messaging-rabbitmq/>
- <https://microservices.io/patterns/microservices.html>
- <https://www.youtube.com/watch?v=MPHyrZWTais>
- <https://spring.io/guides/gs/messaging-rabbitmq/>
- <https://angular.io/tutorial>
- <https://github.com/DickChesterwood/k8s-fleetman/tree/master/k8s-fleetman-webapp-angular>
- <https://microservices.io/patterns/ui/client-side-ui-composition.html>
- <https://gorillalogic.com/blog/build-and-deploy-a-spring-boot-app-on-kubernetes-minikube/>
- <https://dzone.com/articles/quick-guide-to-microservices-with-kubernetes-spring>
- <https://www.javainuse.com/spring/spring-boot-rabbitmq-hello-world>
- <https://microservices.io/patterns/data/database-per-service.html>
- <https://www.javainuse.com/spring/sprsec>
- <https://microservices.io/patterns/security/access-token.html>
- <https://dzone.com/articles/quick-guide-to-microservices-with-kubernetes-spring>