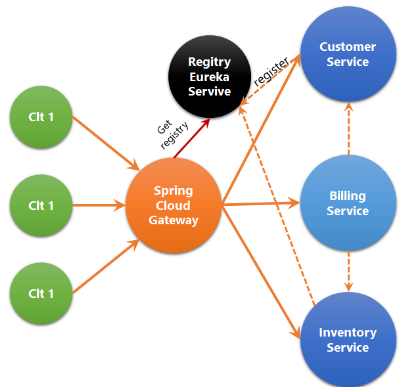
|  |
| --- |
| Travaux Pratiques  FI BDCC S5 : TP de synthèse Systèmes Distribués, Stream et batch processing  Mohamed HAMMANE |

# Introduction et énoncé :

L’objectif est de créer un système distribué basé sur les micro-services permettant de gérer les factures des clients en y intégrant un système de sécurité basé sur Keycloak, Un Bus de messagerie avec KAFKA, un service de Stream processing avec Kafka Streams et un service de Batch Processing avec Spring Batch.

A la fin, nous projetons appliquer pour cette application les patterns CQRS et Event Sourcing.

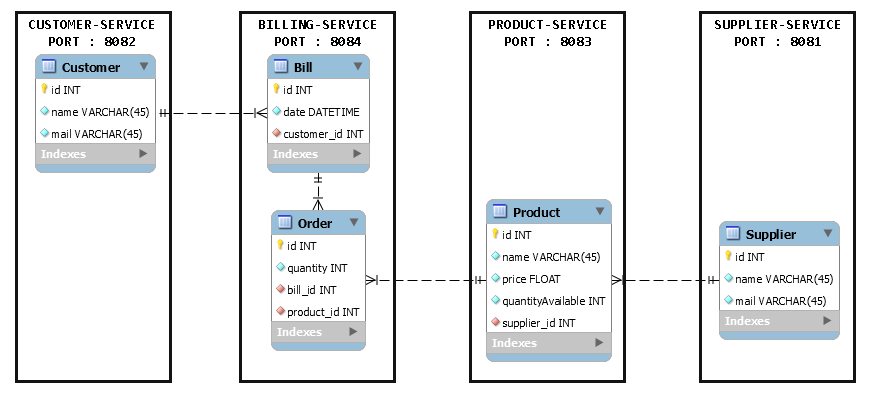


# Travail à faire :

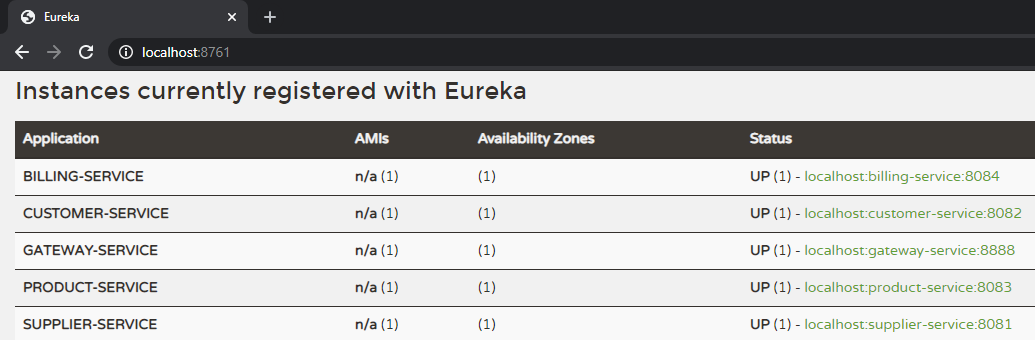
1. Mettre en place les micro-services :
   1. Customer-Service.
   2. Inventory-Service.
   3. Billing-Service.
   4. Eureka Discovery Service.
   5. Spring Cloud Gateway.
2. Mise en place du service de Sécurité avec Keycloak :
   1. Mettre en place le serveur d’authentification OAuth2 Keycloak version 12.0.1.
   2. Créer un Realm.
   3. Le client à sécuriser en mode public client
   4. Créer les rôles (USER, ADMIN, PRODUCT\_MANAGER, CUSTOMER\_MANAGER et BILLING\_MANAGER).
   5. Créer quelques utilisateurs.
   6. Affecter les rôles aux utilisateurs.
   7. Tester l’authentification des utilisateurs en utilisant un client Rest comme ARC :
      1. Authentification avec le mot de passe.
      2. Authentification avec le Refresh Token
   8. Personnaliser le paramétrage des timeouts des tokens.
3. Sécurité l’ensemble des micro-services fonctionnels en mode Bearer-Only en utilisant Spring Security et des adaptateurs Keycloak. On suppose que les micro-services ne sont accessible que pour les utilisateurs authentifiés avec leurs rôles respectifs : PRODUCT\_MANAGER, CUSTOMER\_MANAGER et BILLING\_MANAGER.
4. Développer une application Web Front End qui permet de gérer les produits, les clients et les factures en utilisant le Framework de votre choix : Angulrar, React ou Spring MVC avec Thymeleaf.
5. Sécuriser l’application FrontEnd en mode public client en mettant en place l’adaptateur Keycloak qui instaure un système d’authentification via Keycloak
6. Personnaliser la sécurité de la partie frontend en ajoutant les autres fonctionnalités fournies par Keycloak :
   1. Auto-inscription des utilisateurs
   2. Politique des mots de passe.
   3. Double authentification OTP.
   4. …
7. Mise en place d’une solution de messagerie asynchrone avec le Broker KAFKA :
   1. Mettre en place le Broker KAFKA.
   2. Créer un micro-service Spring Boot qui permet de simuler un Producer KAFKA qui permet d’envoyer à un topic « FACTURATION » à chaque seconde un message contenant le numéro de la facture, le nom du client et le montant de la facture.
   3. Créer un Micro-service Spring Boot qui permet de consommer les messages du Topic « FACTURATION » et de les enregistrer dans sa propre base de données et dans un fichier CSV, avec Une API REST qui permet de consulter les factures.
   4. Créer un Micro-service de Real Time Data Analytics en mode Stream Processing utilisant KAFKA Streams qui permet de traiter en temps réel les messages du Topic « FACTURATION » en produisant des statistiques comme le Total des factures reçus pour les 5 dernières secondes et le total des factures de chaque client.
8. Proposer une solution d’intégration de du BROKER KAFKA dans votre application.
9. Mettre en place un micro-service de batch processing avec Spring Batch permettant de traiter les données du fichier CSV de facturation produit par l’application.

# Réalisation :

## Mettre en place les micro-services :

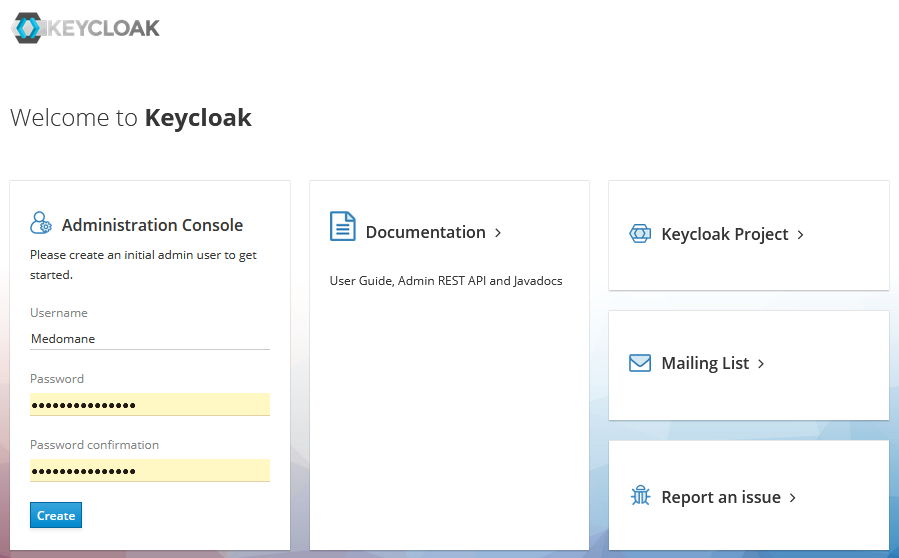


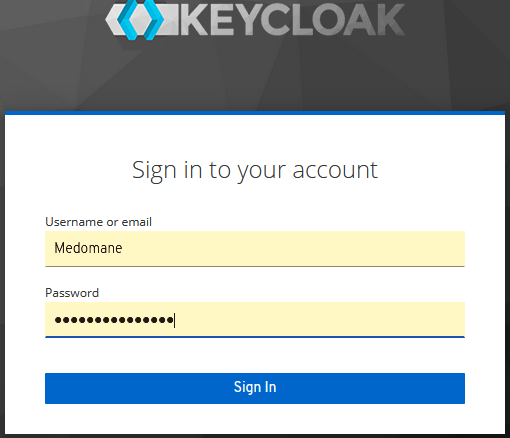
Après avoir démarrer tous les services :



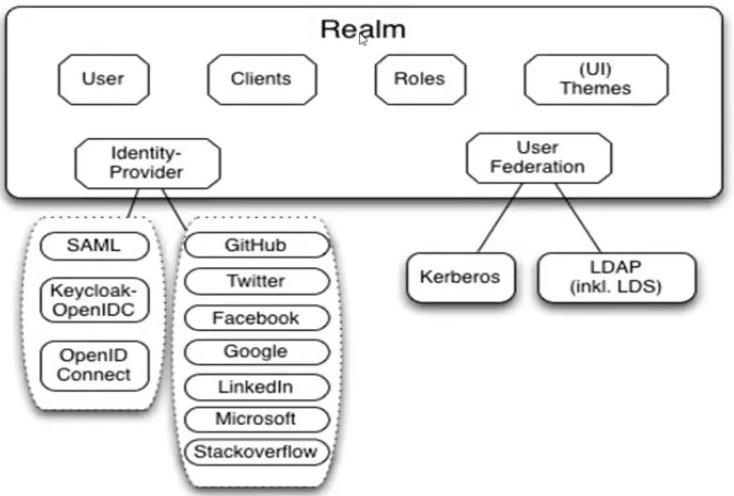
## Mise en place du service de Sécurité avec Keycloak :

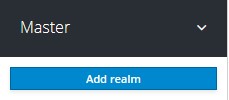
### Mettre en place le serveur d’authentification OAuth2 Keycloak version 12.0.1 :



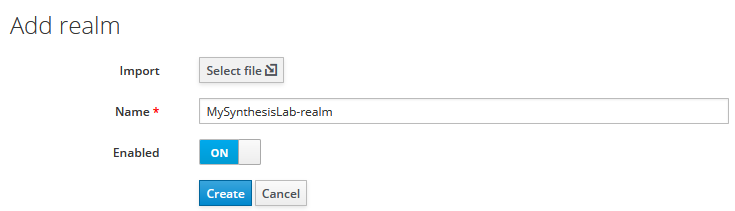


### Créer un Realm :

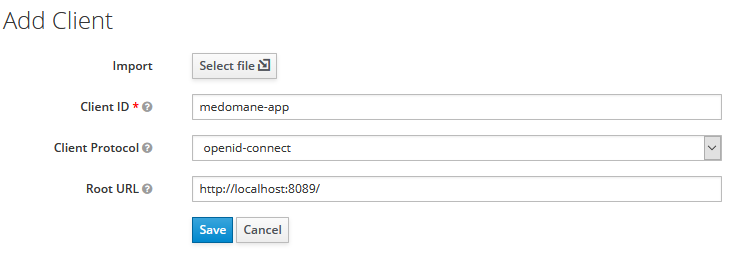


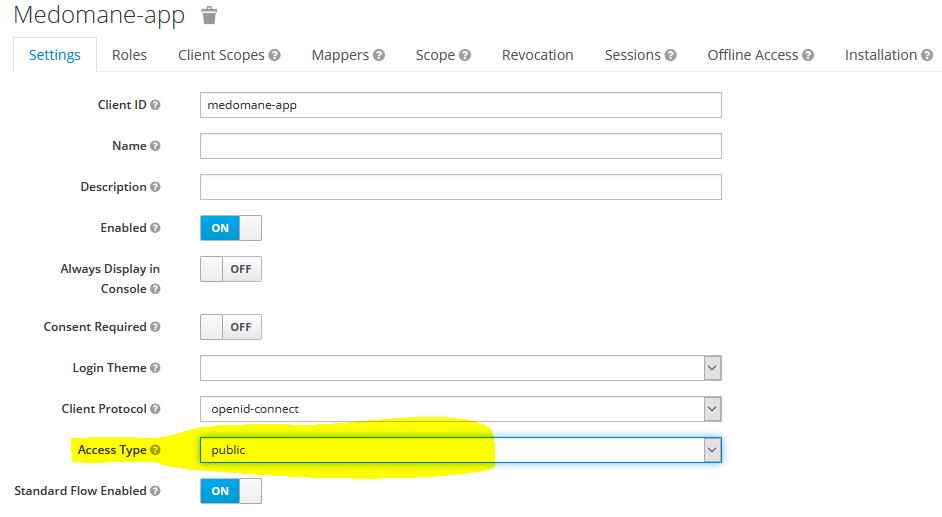


On va nommer notre realm ‘’MySynthesisLab-realm’’ :

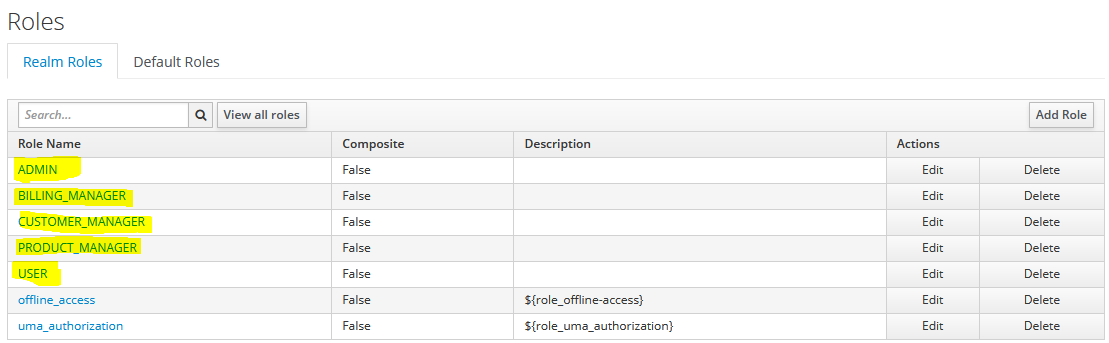


### Le client à sécuriser en mode public client :

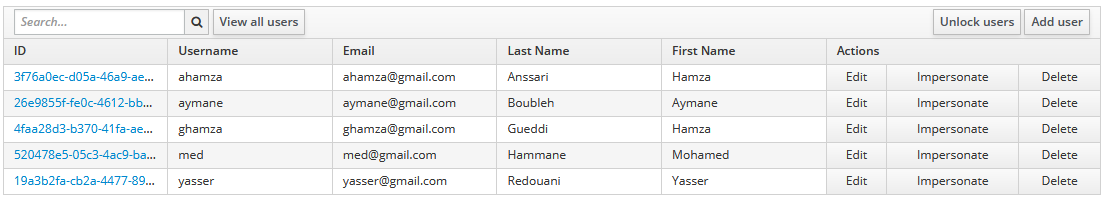




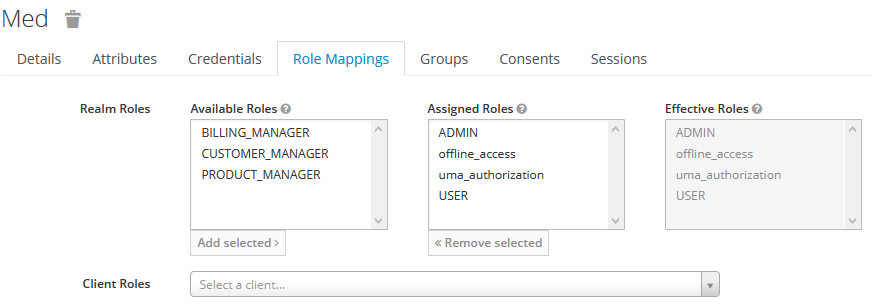
### Créer les rôles (USER, ADMIN, PRODUCT\_MANAGER, CUSTOMER\_MANAGER et BILLING\_MANAGER) :



### Créer quelques utilisateurs :



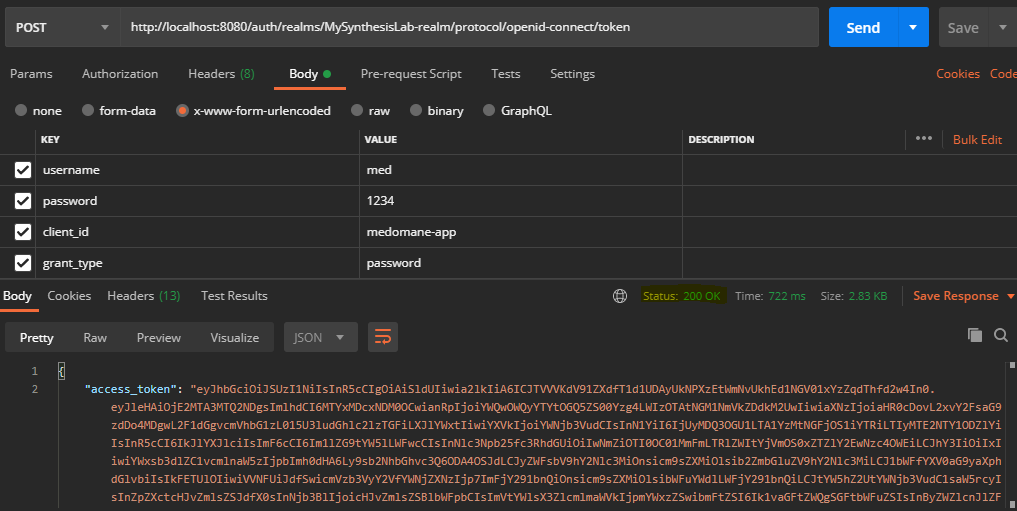
### Affecter les rôles aux utilisateurs :



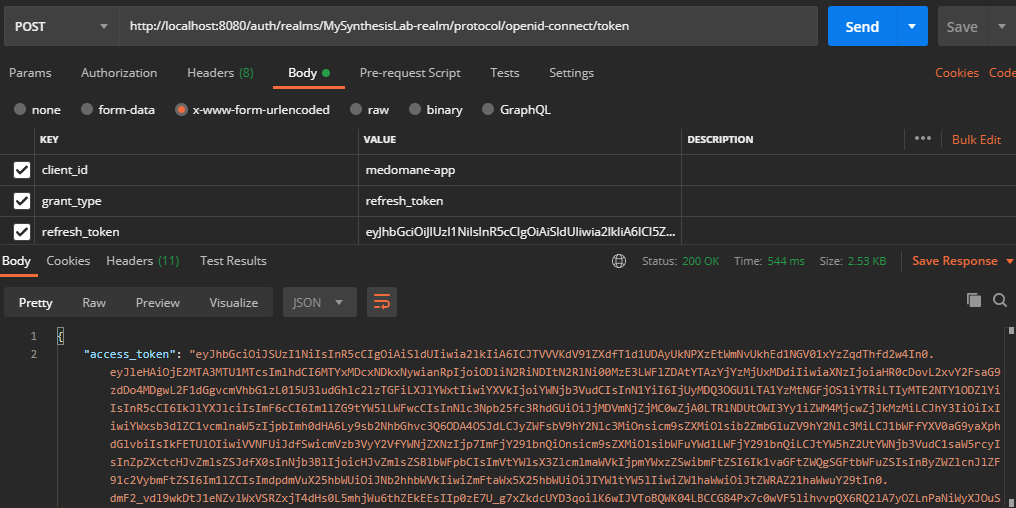
Même chose pour les autres utilisateurs.

### Tester l’authentification des utilisateurs en utilisant un client Rest :

* Authentification avec le mot de passe :



* Authentification avec le Refresh Token :



### Personnaliser le paramétrage des timeouts des tokens :





## Sécurisé l’ensemble des micro-services fonctionnels :

### Supplier service :

@Override  
protected void configure(HttpSecurity http) throws Exception {  
 super.configure(http);  
 http.authorizeRequests()  
 .antMatchers(HttpMethod.*GET*).permitAll()  
 .antMatchers(HttpMethod.*POST*).hasAuthority("SUPPLIER\_MANAGER")  
 .antMatchers(HttpMethod.*DELETE*).hasAuthority("SUPPLIER\_MANAGER");  
 http.csrf().disable();  
}

### Product service :

@Override  
protected void configure(HttpSecurity http) throws Exception {  
 super.configure(http);  
 http.authorizeRequests()  
 .antMatchers(HttpMethod.*GET*).permitAll()  
 .antMatchers(HttpMethod.*POST*).hasAuthority("PRODUCT\_MANAGER")  
 .antMatchers(HttpMethod.*DELETE*).hasAuthority("PRODUCT\_MANAGER");  
 http.csrf().disable();  
}

### Customer service :

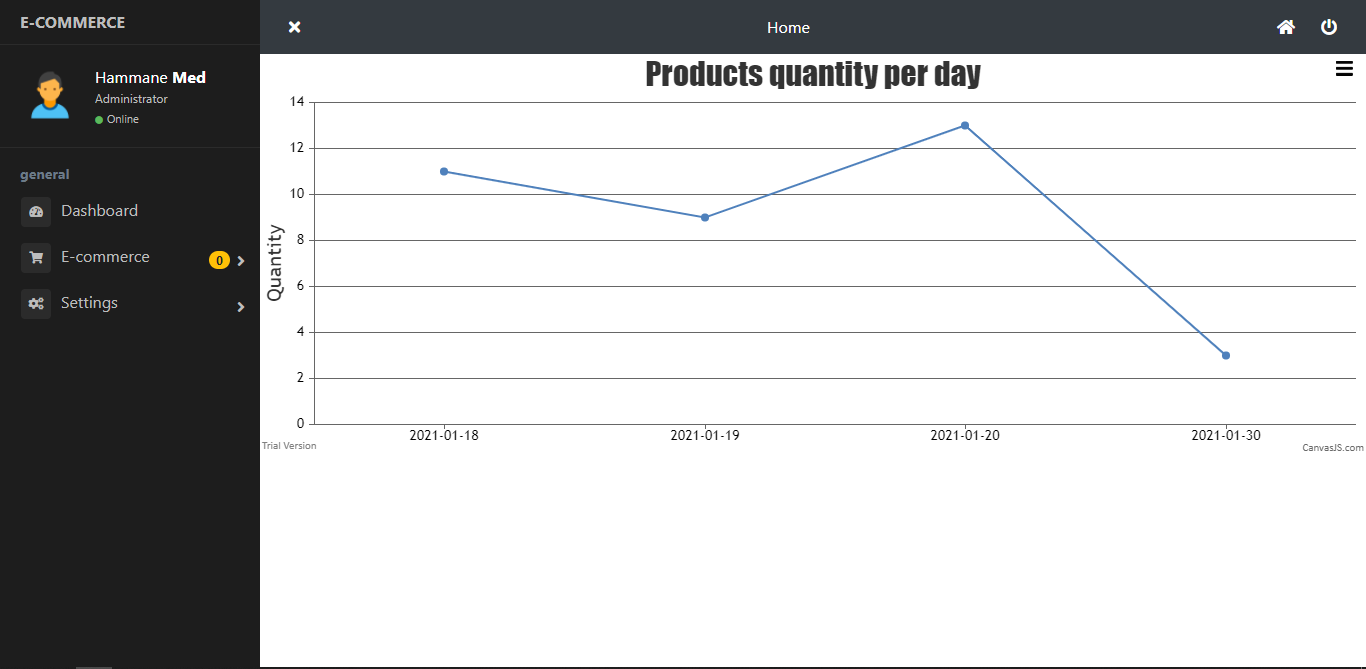
@Override  
protected void configure(HttpSecurity http) throws Exception {  
 super.configure(http);  
 http.authorizeRequests()  
 .antMatchers(HttpMethod.*GET*).permitAll()  
 .antMatchers(HttpMethod.*POST*).hasAuthority("CUSTOMER\_MANAGER")  
 .antMatchers(HttpMethod.*DELETE*).hasAuthority("CUSTOMER\_MANAGER");  
 http.csrf().disable();  
}

### Billing service :

@Override  
protected void configure(HttpSecurity http) throws Exception {  
 super.configure(http);  
 http.authorizeRequests()  
 .antMatchers(HttpMethod.*GET*).permitAll()  
 .antMatchers(HttpMethod.*POST*).authenticated();  
 http.csrf().disable();  
}

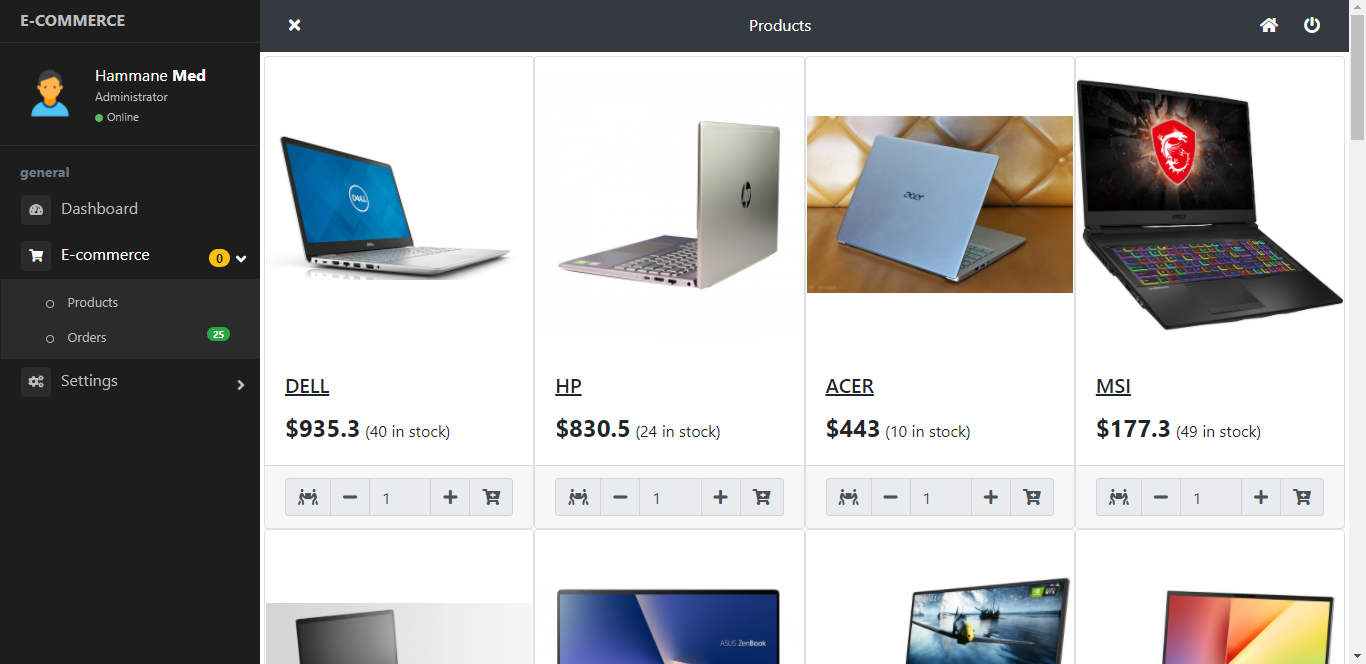
## Développer une application Web Front End qui permet de gérer les produits, les clients et les factures en utilisant Angular :

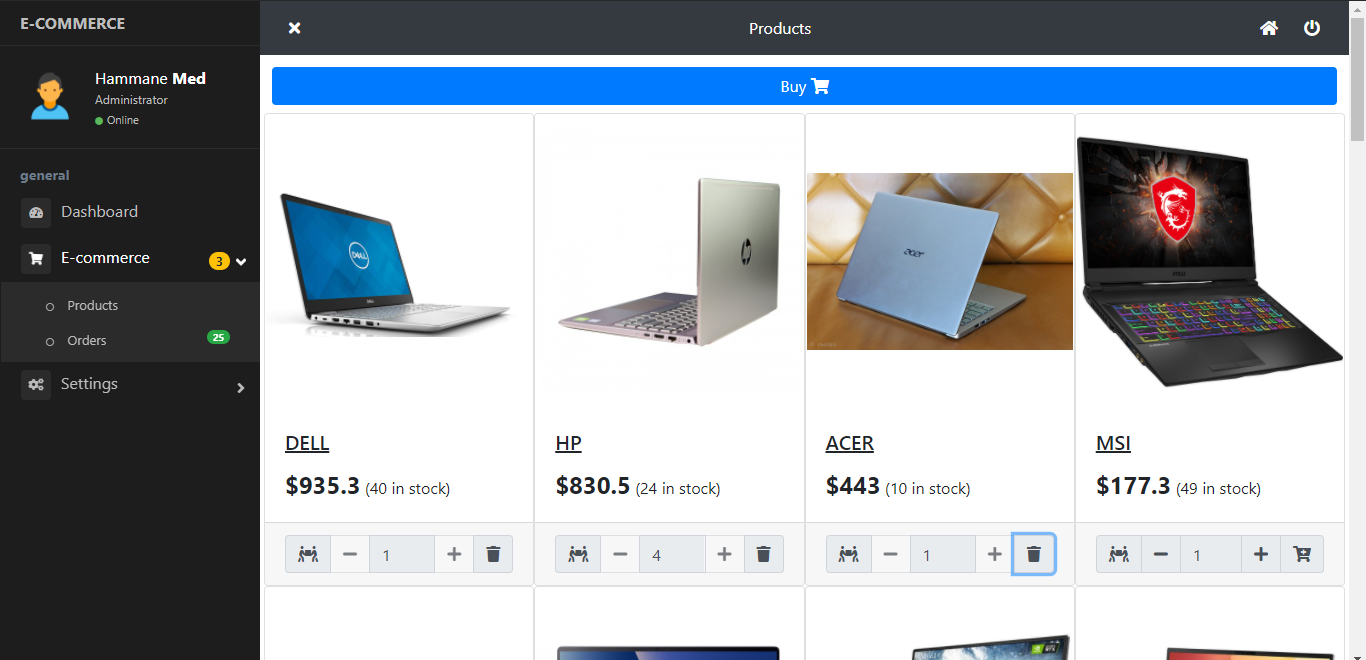
### Dashboard :

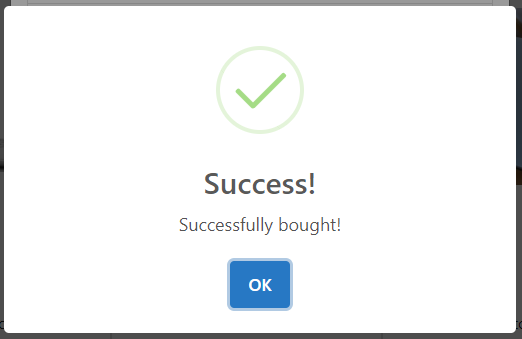
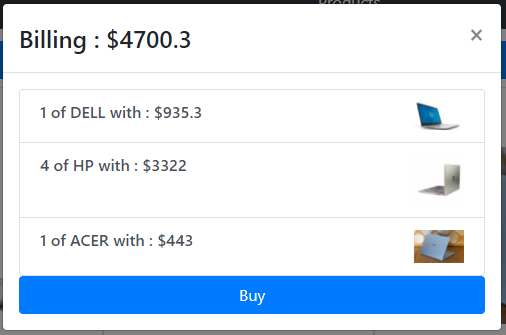


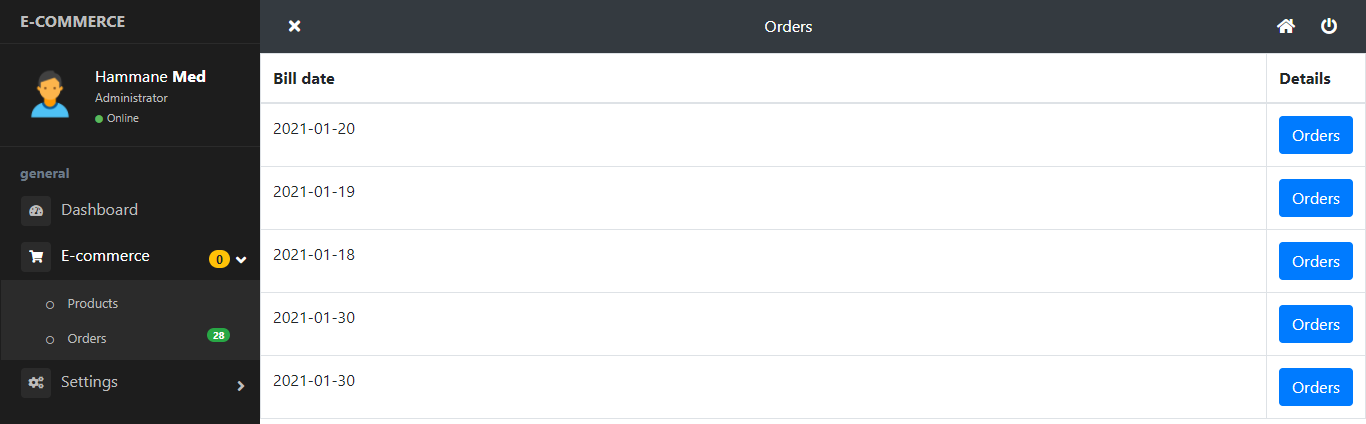
Cette page permet d’afficher le nombre d’article achetés par date.

### E-Commerce :

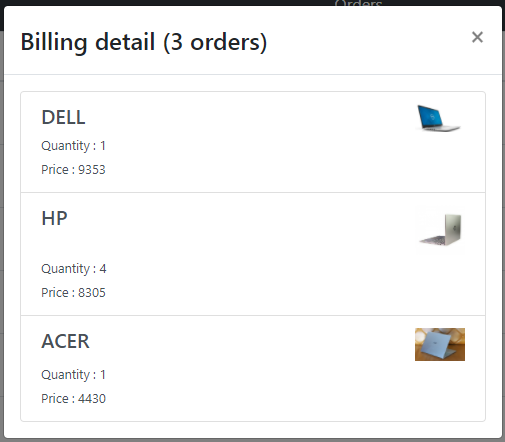
La première partie concerne les produits après l’authentification le client peut acheter des produits en cliquant sur ajouter au panier .après qu’il ajouter des produits au panier

 Il peut les achetés en cliquant sur buy :



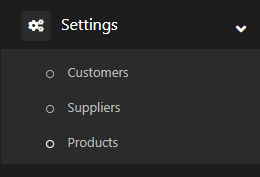
Et la deuxième partie concerne les factures :

Pour chaque facture on peut voir la liste des ordres :

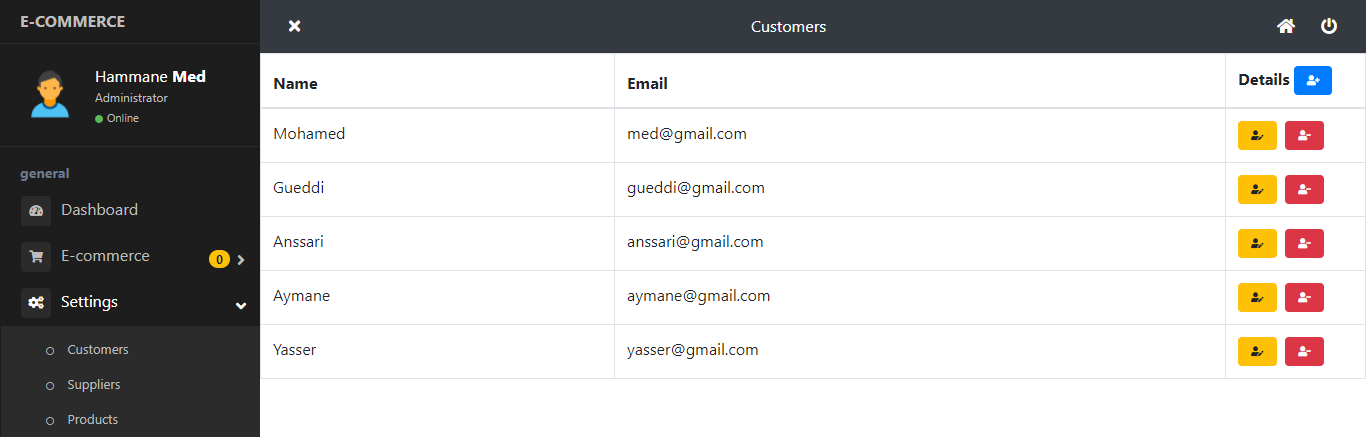


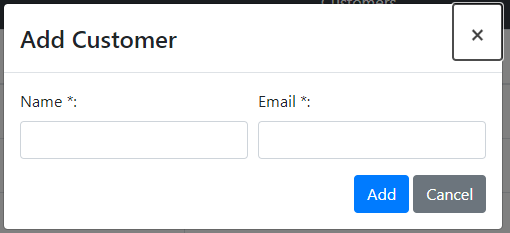
### Paramètres :

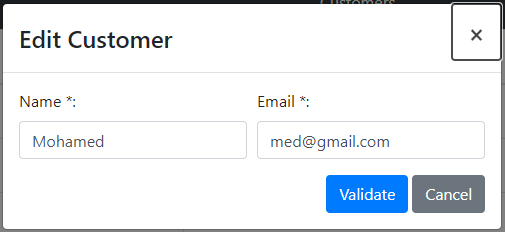
Cette partie concerne les paramètres des produits, fournisseurs et produits :

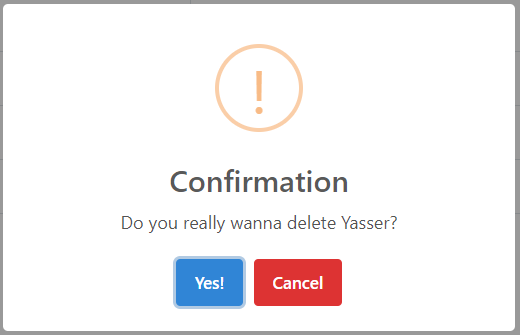


Ici la gestion des clients :

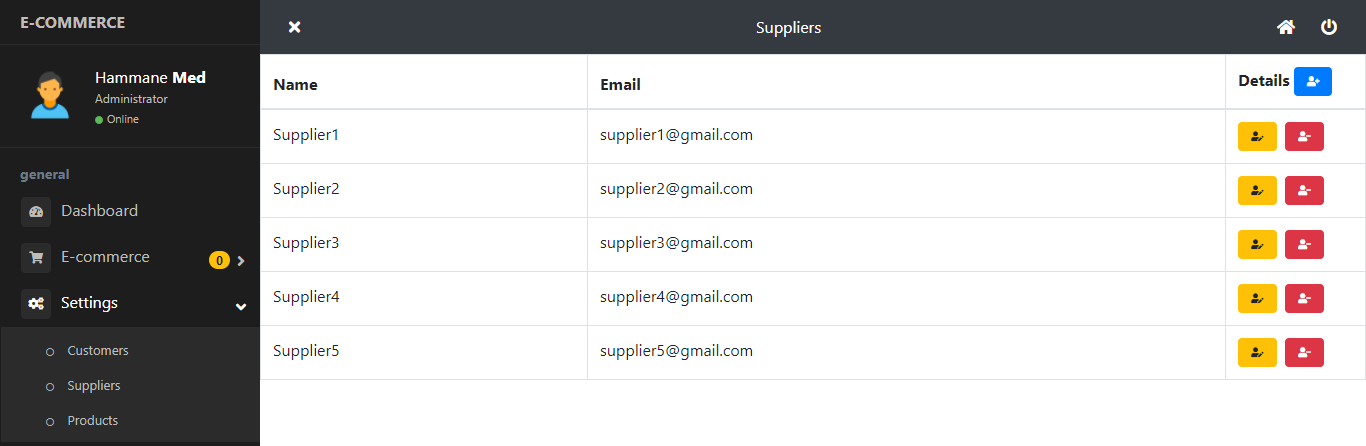




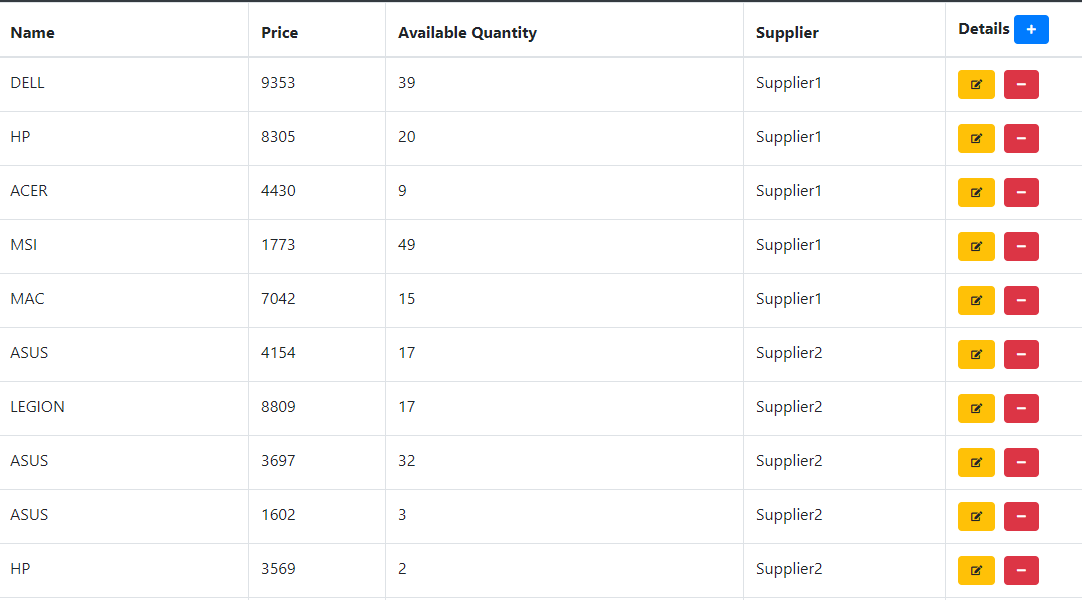


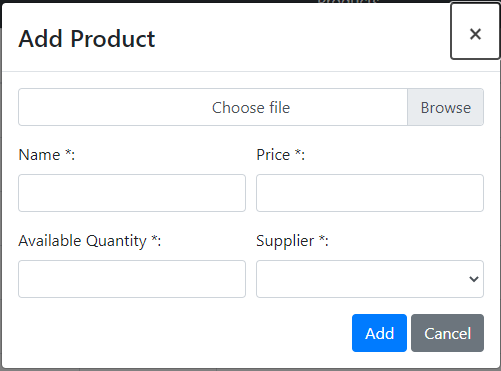


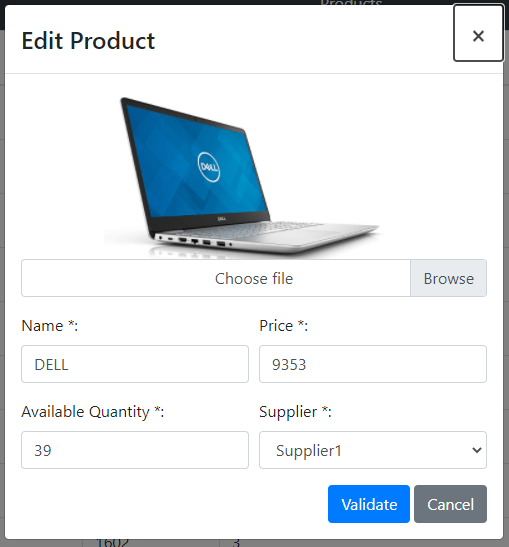
Aussi pour la gestion des fournisseurs :

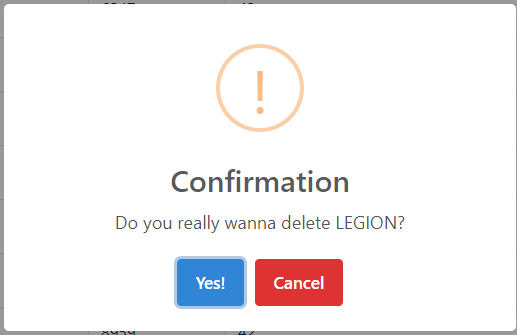


Et finalement la gestion des produits :









## Sécuriser l’application FrontEnd en mode public client en mettant en place l’adaptateur Keycloak qui instaure un système d’authentification via Keycloak :

Après l’installation de la bibliothèque KeyCloak j’ai créé un service pour gérer la sécurité coté client ainsi le système d’authentification.

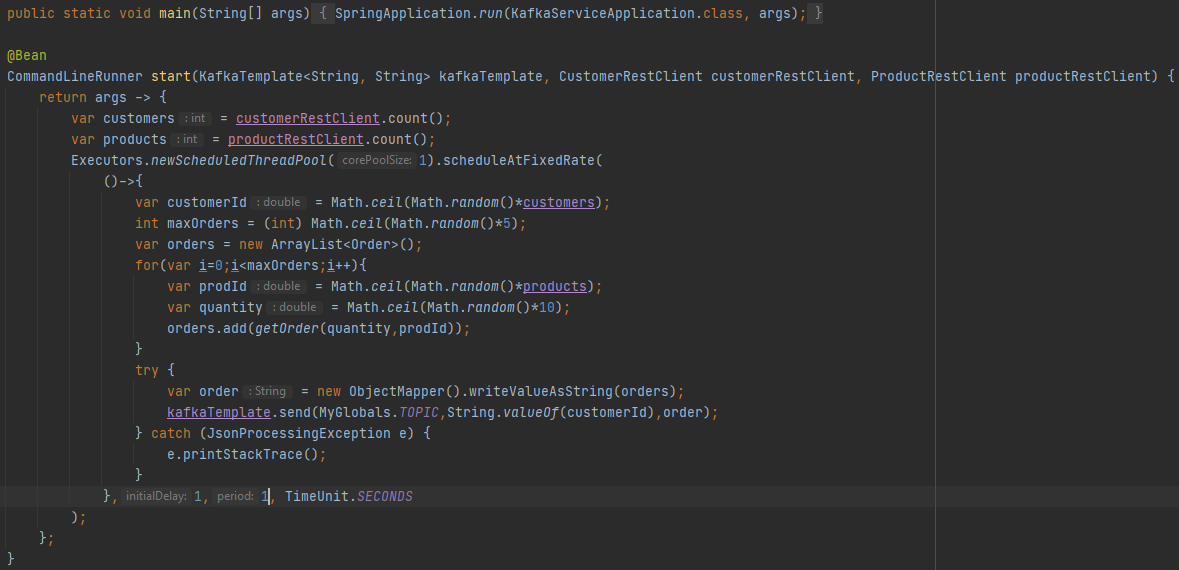
## 

## Fonctionnalités :

Et aussi j’ai ajouté quelques fonctionnalités de keycloak dans la partie front-end.

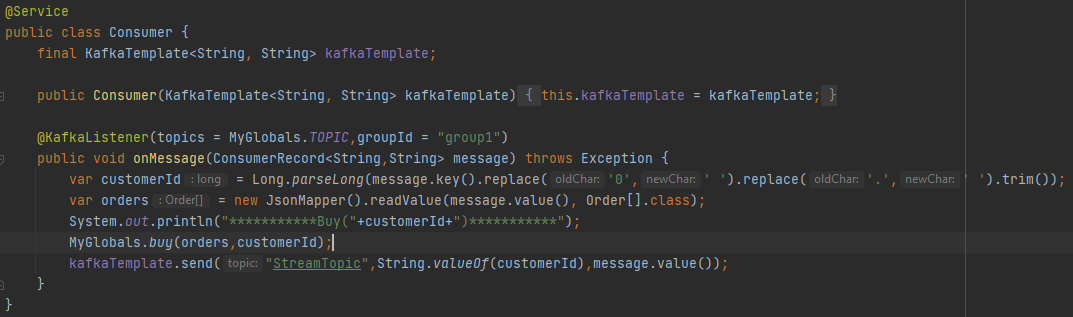
## Solution Kafka :

La première chose j’ai créé un Producer (le micro service kafka-service) qui génère des Orders pour chaque Costumer



Puis il envoie le résultat dans un topic(FACTURATION).

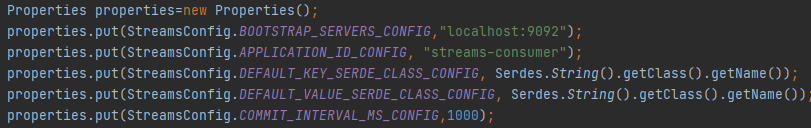
Puis on j’ai créé le spring boot consumer qui effectue des achats et génère des factures dans le topic StreamTopic



Puis on a maintenant notre notre Kafka stream consumer qui va faire real-time analytics de notre data pour l’envoyer par la suite à notre application front-end

Et voici la partie Stream processing :

On a d’abord la partie configuration de notre consumer



Puis la partie du stream processing :



Cette partie se charge de grouper les factures par client puis calculer la somme des quantités dans un window de 5 seconds puis envoyer le résultat dans le topic streamKafkaTopic (topic qu’on va utiliser pour la partie front-end en utilisant webflux websockets avec la bibliothèque Angular NGXS).

Et finalement pour la partie back-end j’ai créé un micro-service (reactivekafka) qui va se charger d’envoyer les messages du topic streamKafkaTopic vers la partie Angular en utilisant webflux.

Et voici le service :

@Service  
public class KafkaServiceImpl implements KafkaService {  
 private final Flux<ReceiverRecord<String, String>> testTopicStream;  
 KafkaServiceImpl() {  
  
 Properties kafkaProperties = new Properties();  
 kafkaProperties.put(ConsumerConfig.*BOOTSTRAP\_SERVERS\_CONFIG*,"localhost:9092");  
 kafkaProperties.put(ConsumerConfig.*CLIENT\_ID\_CONFIG*, "reactive-consumer");  
 kafkaProperties.put(ConsumerConfig.*GROUP\_ID\_CONFIG*, "sample-group");  
 kafkaProperties.put(ConsumerConfig.*KEY\_DESERIALIZER\_CLASS\_CONFIG*, StringDeserializer.class);  
 kafkaProperties.put(ConsumerConfig.*VALUE\_DESERIALIZER\_CLASS\_CONFIG*, StringDeserializer.class);  
 kafkaProperties.put(ConsumerConfig.*AUTO\_OFFSET\_RESET\_CONFIG*, "earliest");  
  
 ReceiverOptions<String, String> receiverOptions = ReceiverOptions.*create*(kafkaProperties);  
  
 testTopicStream = createTopicCache(receiverOptions);  
 }  
   
 public Flux<ReceiverRecord<String, String>> getTestTopicFlux() {  
 return testTopicStream;  
 }  
  
 private <T, G> Flux<ReceiverRecord<T, G>> createTopicCache(ReceiverOptions<T, G> receiverOptions) {  
 ReceiverOptions<T, G> options = receiverOptions.subscription(Collections.*singleton*("streamKafkaTopic"));  
 return KafkaReceiver.*create*(options).receive().cache();  
 }  
}

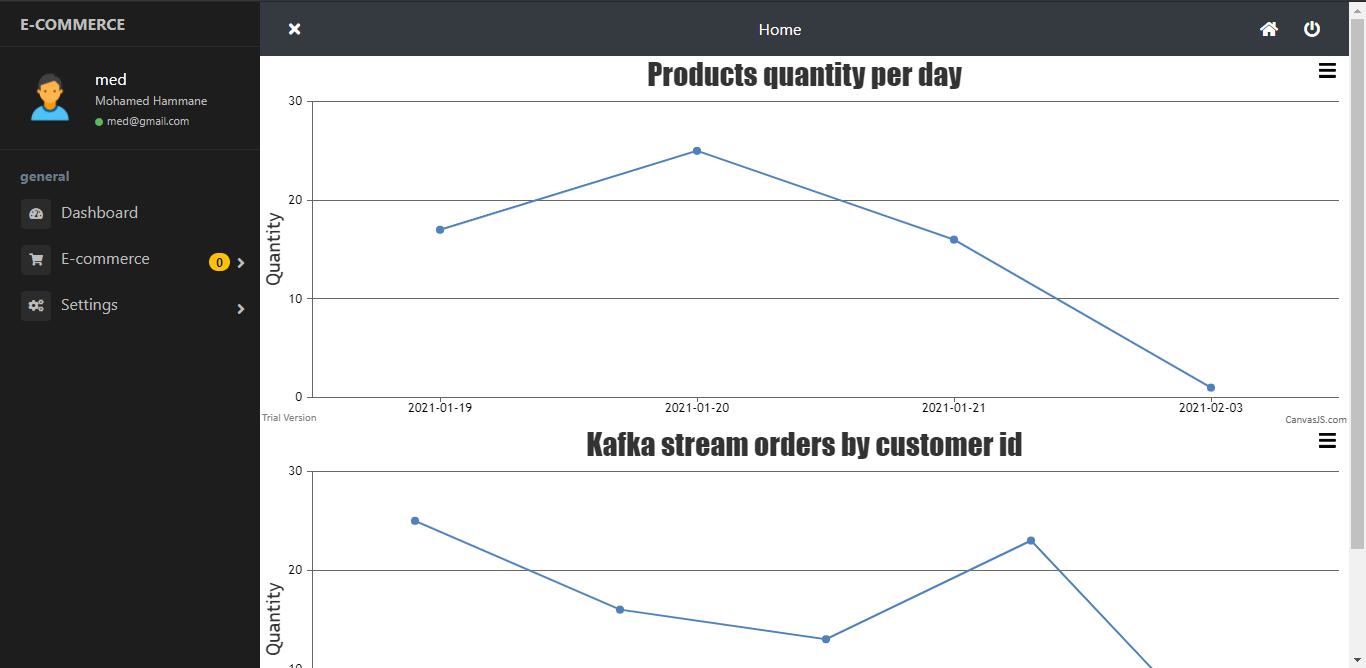
Et la configuration:

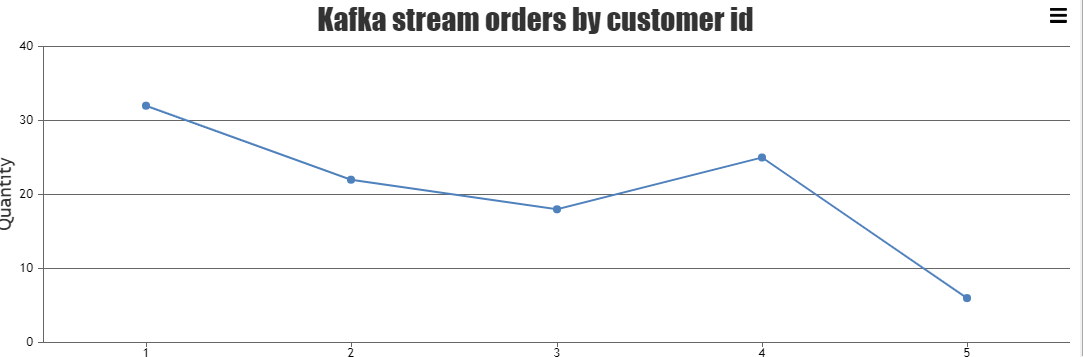
@Configuration  
public class ReactiveWebSocketConfiguration {  
  
 private final WebSocketHandler webSocketHandler;  
  
 public ReactiveWebSocketConfiguration(WebSocketHandler webSocketHandler) {  
 this.webSocketHandler = webSocketHandler;  
 }  
  
 @Bean  
 public HandlerMapping webSocketHandlerMapping() {  
 Map<String, WebSocketHandler> urlMap = new HashMap<>();  
 urlMap.put("/websocket", webSocketHandler);  
 CorsConfiguration corsConfiguration = new CorsConfiguration();  
 corsConfiguration.addAllowedOrigin("http://localhost:4200");  
 SimpleUrlHandlerMapping handlerMapping = new SimpleUrlHandlerMapping();  
 handlerMapping.setOrder(1);  
 handlerMapping.setUrlMap(urlMap);  
 return handlerMapping;  
 }  
  
 @Bean  
 public WebSocketHandlerAdapter handlerAdapter() {  
 return new WebSocketHandlerAdapter();  
 }  
}

Puis le Component:

@Component  
public class ReactiveWebSocketHandler implements WebSocketHandler {  
  
 private static final ObjectMapper *json* = new ObjectMapper();  
  
 private final KafkaService kafkaService;  
  
 public ReactiveWebSocketHandler(KafkaService kafkaService) {  
 this.kafkaService = kafkaService;  
 }  
  
 @Override  
 public Mono<Void> handle(WebSocketSession webSocketSession) {  
 return webSocketSession.send(kafkaService.getTestTopicFlux()  
 .map(record -> {  
 Message message = new Message("kafkaStream", record.key()+"|"+record.value());  
 System.*out*.println(record.key()+" --> "+record.value());  
 try {  
 return *json*.writeValueAsString(message);  
 } catch (JsonProcessingException e) {  
 return "Error while serializing to JSON";  
 }  
 })  
 .map(webSocketSession::textMessage))  
 .and(webSocketSession.receive().map(WebSocketMessage::getPayloadAsText).log());  
 }  
}

Pour la partie front-end avec Angular j’ai utilisé la bibliothèque NGXS puis avoir en temps réelle les données de notre topic streamKafkaTopic, et j’ai ajouter un Chart graphique qui represente ce changement en temps réelle :





# Code source :

<https://github.com/Medomane/SynthesisLab> avec une vidéo démo.