Let's make pizza!

Rapport de projet

Hugo COLLIN,

Reika JACQUOT,

Nathanaël MIESCH,

Gaël BALLOIR

18/05/2025

Table des matières

Introduction

Le projet "Let's make pizza!" visait à développer un système distribué permettant de commander des pizzas, suivre leur préparation, leur cuisson et leur livraison. Ce système met en œuvre une communication asynchrone basée sur des événements via le protocole MQTT.

Dans un contexte de restauration rapide, une gestion efficace des commandes est essentielle. Notre système vise à optimiser ce processus.

Le présent rapport détaille l'architecture, les choix d'implémentation et les fonctionnalités du système développé.

Gestion du projet et organisation

Organisation projet

Nous avons mis en place un Trello (tableau Kanban) et avons divisé le projet en tâches de tailles raisonnables pour constituer les cartes. Les tâches ont ensuite été prises ou attribuées aux membres de l'équipe pour leur exécution.

Nous avons également utilisé un groupe Discord comme moyen de communication de l'équipe durant ce projet.

Outils et pratiques pour le développement

Le projet se présente sous la forme d'un dépôt Git hébergé sur le GitLab de l'Université de Lorraine, et forké (basé) depuis le template fourni.

Le dépôt Git est composé d'une branche `main` sur laquelle a été poussé le travail fonctionnel validé. Le travail en développement a été effectué sur des branches dérivant de `main`, tel `docs`, `dev`, `docs/ab-cd`, `dev/xyz`, etc. Pour être intégré à la branche `main`, ce travail était soumis via une Merge Request (demande de fusion au `main`) et validé par au moins 1 membre de l'équipe. Le cas échéant, nous analysions le problème et collaborions pour le résoudre.

Répartition des tâches

Le travail s’est réparti de la manière suivante :

- Développement
    - Mise en place du dépôt et initialisation des sous-projets : Hugo COLLIN
    - Premiers échanges Client-Pizzeria via broker MQTT : Hugo COLLIN
    - Requête du Client et renvoi par la Pizzeria du menu de pizzas : Hugo COLLIN
    - Ecran de sélection des pizzas avec quantités : Hugo COLLIN
    - Initialisation d'une commande - Requête du client et renvoi par la pizzeria de son état : Gaël BALLOIR
    - Intégration du pizzaiolo à la Pizzeria : Reika JACQUOT
    - Préparation et cuisson des pizzas avec notification des étapes : Gaël BALLOIR
    - Gestion de plusieurs commandes en parallèle : Hugo COLLIN
    - Récupération et affichage des messages Pizzeria côté client : Nathanaël MIESCH
    - Désactivation du mode quantique : Reika JACQUOT
    - Différenciation des écrans suivant la maquette : Gaël BALLOIR, Hugo COLLIN
    - Gérer les commandes incomplètes côté Pizzeria : Reika JACQUOT
    - Projet Maven multi-modules avec code commun et JARs autonomes : Hugo COLLIN
    - Affichage commande incomplète ou annulée côté Client : Nathanaël MIESCH
    - Gestion des problèmes de connexion : Nathanaël MIESCH, Gaël BALLOIR
- Validation :
    - Test manuel et analyse des Merge Requests (MR) : Tous les membres (autre que l’auteur de la MR, minimum 1 validation pour fusionner au main)
- Déploiement :
    - Mise en place de la pipeline GitLab de déploiement des JARs : Hugo COLLIN
    - Release des JARs déployés : Hugo COLLIN

- Documentation :
    - Rédaction du rapport explicatif (ce rapport) : Hugo COLLIN, Reika JACQUOT
    - Documentation technique (READMEs) : Hugo COLLIN
    - Documentation dans le code : Hugo COLLIN, Reika JACQUOT

La signification de ces tâches est explicitée dans la suite de ce document.

Vue globale

Technologies utilisées

Les technologies utilisées dans ce projet sont les suivantes :

- Java en version 21 comme langage de programmation
- Maven comme gestionnaire de paquets et de modules (structure, build…)
- JavaFX pour l'interface graphique du client
- JUnit pour les tests unitaires
- Docker pour lancer un broker MQTT Mosquitto

Java est un langage de programmation multiplateforme, orienté objet, permettant de construire des logiciels localisés et distribués. Java 21 est une version majeure avec un support long-terme, c'est pourquoi nous l'utilisons dans le cadre de ce projet.

Maven est un outil de gestion de projet permettant de diviser l'application en modules afin de centraliser la compilation et la création des JARs exécutables.

JavaFX est un outil de création d'interface utilisateur (UI) pour Java largement utilisé aujourd'hui.

JUnit est le framework de test unitaire pour le langage Java.

Docker permet d'automatiser le déploiement d'un broker MQTT Mosquitto, garantissant une installation rapide, reproductible et indépendante de l'environnement d'exécution.

Projet multi-modules

Notre système se divise en trois composants principaux, chacun ayant un rôle à part entière:

- **Client** : Application JavaFX permettant aux utilisateurs de consulter le menu, passer commande et suivre la livraison, développé sous la forme d'un module Maven ;
- **Broker MQTT** : Serveur Mosquitto déployé via Docker qui gère la communication entre les clients et la pizzeria ;
- **Pizzeria** : Service Java qui reçoit les commandes, gère leur préparation et notifie les clients de l'avancement, développé sous la forme d'un module Maven.

S'ajoute à ces 3 modules un module **Common** regroupant le code commun au Client et à la Pizzeria (cf. Communication Client-Pizzeria, section Modèle de données).

Nous avons commencé par développer le Client et la Pizzeria sous la forme de 2 projets Maven distincts communiquant via un bus de données MQTT (un troisième projet Broker) dans le dépôt Git.

Nous les avons par la suite liés ensemble via un pom parent, pour ainsi former un projet Maven multi-modules. Cela permet de centraliser et faciliter le build, et donne la possibilité d'avoir du code commun entre Client et Pizzeria.

Ainsi, la structure du projet est la suivante :

```
lets-make-a-pizza
|- docs
|- broker
|- client
|- common
|- pizzeria
|- pom.xml
|- README.md
|- .gitignore
|- .gitlab-ci.yml
```

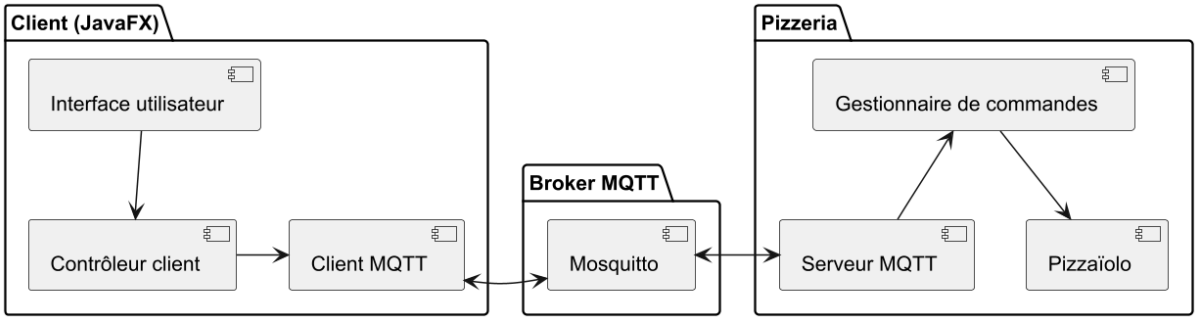
- Chacun des composants `broker`, `client` et `pizzeria` sont développés dans les répertoires correspondants.
- Le répertoire `docs/` contient les documents liés à l'étude.
- Le fichier `pom.xml` contient les informations et paramètres du projet Maven (nom, dépendances, plugins, etc.)
- Le fichier `README.md` contient les informations nécessaires pour comprendre et faire fonctionner le projet.
- Le fichier `.gitignore` demande à Git d'ignorer certains fichiers, notamment suivant leurs formats. Cela permet de ne pas encombrer le dépôt de fichiers inutiles ou reproductibles.
- Le fichier `.gitlab-ci.yml` configure une pipeline GitLab permettant de publier les binaires (JARs) autonomes sur le dépôt.

Communication entre composants

La solution s'appuie sur une architecture Publish/Subscribe qui permet une communication découplée entre les Clients et la Pizzeria. Cette approche offre plusieurs avantages pour ce type d'application :

- Découplage temporel : les clients et la pizzeria n'ont pas besoin d'être actifs simultanément
- Évolutivité : possibilité d'ajouter facilement de nouveaux clients ou pizzerias
- Réactivité : notification en temps réel de l'avancement des commandes

Diagramme de composants illustrant la communication entre composants :



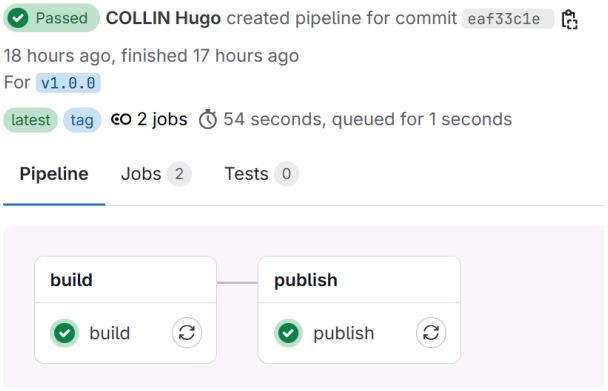
Le scénario de communication typique pour une commande est le suivant :

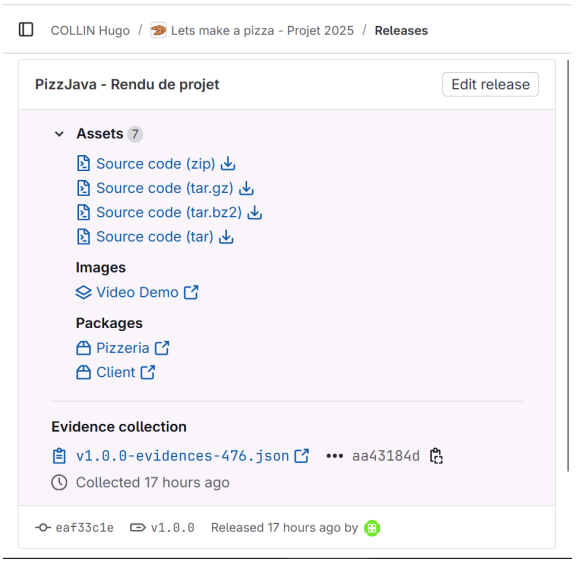
- Le client demande la liste des pizzas disponibles (`bcast/i_am_ungry`)
- La pizzeria répond avec le menu (`bcast/menu`)
- Le client passe commande (`orders/<xxx>`)
- La pizzeria valide la commande et notifie le client des différentes étapes (`orders/<xxx>/status/<status>`), ou annule la commande (`orders/<xxx>/cancelled`)
- La pizzeria notifie la livraison (`orders/xxx/delivery`) et indique si certaines pizzas n’ont pas pu être livrées.

Déploiement

Les JARs Client et Pizzeria sont compilés puis hébergés sur le dépôt via une pipeline de CI/CD GitLab, déclenchée par l'envoi d'un tag Git. Cette pipeline est configurée dans le fichier `.gitlab-ci.yml` .

Une fois hébergés, une Release (lancement de version) manuelle pointant vers ces JARs peut être créée.





Communication Client-Pizzeria

Broker MQTT

Le bus MQTT sert d'intermédiaire entre les clients et la pizzeria : il reçoit les commandes des clients et les transmet à la pizzeria. Notre projet utilise Eclipse Mosquitto, un broker MQTT Open Source connu, léger et efficace.

La structure du module est la suivante :

```
broker
|- docker-compose.yml
|- mosquitto.conf
|- README.md
```

Ce composant prend la forme d'une image Docker que nous paramétrons via une configuration docker-compose. La configuration du broker Mosquitto, elle, est définie dans le fichier `mosquitto.conf`.

Notre configuration :

- Expose le port standard MQTT (1883)
- Permet les connexions anonymes (pour simplifier le développement)
- Active la persistance des messages pour éviter les pertes en cas de redémarrage

Modèle de données commun

Le client et la pizzeria partagent un même modèle de données. Il s'agit de classes Java communes permettant la coordination des informations entre Clients et Pizzeria :

- Pizza : représente les informations pour une pizza
- Order : représente les informations pour une commande

La structure du module est la suivante :

```
common
|- src
|  |- main
|     |- java
|        |- com
|        |  |- pizza
|        |     |- adapter
|        |     |  |- IngredientAdapter.java
|        |     |- Order.java
|        |     |- Pizza.java
|        |- module-info.java

|- pom.xml
|- README.md
```

Dans un premier temps, ces classes ont simplement été dupliquées côté Client et côté Pizzeria. Conscients qu'il s'agit d'une mauvaise pratique de développement, nous avons par la suite mis en place un module commun (qui a nécessité mise en place du projet multi-modules) ce qui a permis de déclarer ces classes une seule fois tout en les partageant pour la compilation de Client et de Pizzeria. Ce code commun au build permet d'éviter la duplication du code dans les 2 modules.

Cette mise en commun a notamment soulevé un défi technique : auparavant, la classe Pizza côté Client prenait une liste de chaînes pour les ingrédients, et la classe Pizza Pizzeria une liste de `Pizzaiolo.Ingredient`.

Nous avons donc mis en place le patron de conception Adaptateur via une interface commune et son implémentation côté Pizzeria, ce qui permet de convertir les `Pizzaiolo.Ingredient` en chaîne pour l'utilisation du `List<String>` dans la classe commune Pizza.

Format des données sérialisées

Nous avons mis en œuvre un format simple pour sérialiser les pizzas.

- Chaque pizza est représentée par une chaîne de caractères avec le format suivant : `nom|ingrédient1,ingrédient2,...|prix`
- Et la liste complète des pizzas (le menu) est transmise comme une série de pizzas séparées par des points-virgules : `pizza1|ing1,ing2|prix1;pizza2|ing1,ing2,ing3|prix2;...`

  Par exemple, un menu avec deux pizzas pourrait ressembler à :

  `margherita|sauce tomate,mozarella,basilic|800;napoli|sauce tomate,mozarella,anchois,olives|950`

- Les commandes sont sérialisées avec un format similaire, où chaque pizza est associée à sa quantité : `pizza1:quantité1,pizza2:quantité2,...`

  Par exemple, une commande de 2 Margherita et 1 Napolitaine serait sérialisée comme : `margherita:2,napoli:1`

Pour éviter les problèmes liés aux caractères spéciaux qui pourraient interférer avec notre format de sérialisation, nous avons implémenté une méthode `sanitize` qui nettoie les chaînes de caractères. Cette méthode est appliquée aux noms de pizzas et aux ingrédients pour garantir l'intégrité de nos données sérialisées.

Ce format de sérialisation était facile à implémenter et à analyser, tout en restant lisible.

Interface de commande Client

L'interface client est une application JavaFX qui permet aux utilisateurs de consulter le menu de pizzas, passer commande et suivre l'état de leur commande en temps réel.

Vue globale du module

Le module client est structuré selon le modèle MVC (Modèle-Vue-Contrôleur) :

```
client
|- src
|  |- main
|     |- java
|        |- com
|        |- pizza
|           |- ClientApplication.java
|           |- ClientLauncher.java
|           |- MQTTClient.java
|           |- TimeoutTimer.java
|           |- controllers
|              |- OrderController.java
|              |- WaitingController.java
|              |- WelcomeController.java
|     |- resources
|        |- com
|           |- pizza
|              |- assets
|              |  |- pizza.png
|              |- views
|                 |- welcome-view.fxml
|                 |- order-view.fxml
|                 |- waiting-view.fxml
|- pom.xml
|- README.md
```

L'application est divisée en trois écrans principaux, chacun avec sa propre vue FXML et son contrôleur associé :

- Écran d'accueil (`welcome-view.fxml`) : Page d'accueil avec logo et bouton pour commencer la commande ;
- Écran de commande (`order-view.fxml`) : Affichage du menu et sélection des pizzas avec quantités ;
- Écran d'attente (`waiting-view.fxml`) : Suivi en temps réel de l'état de la commande.

La classe `ClientApplication` sert de point d'entrée à l'application et gère les transitions entre les différents écrans. La classe `MQTTClient` contient toute la logique de communication avec le serveur Pizzeria via le protocole MQTT.

Configuration et point d’entrée

Le fichier `pom.xml` configure le projet avec les dépendances nécessaires, notamment :

- JavaFX pour l'interface graphique
- MQTT pour la communication
- Le module commun pour les modèles de données partagés

Deux plugins Maven sont configurés :

- `javafx-maven-plugin` pour le développement et les tests
- `maven-assembly-plugin` pour créer un JAR exécutable autonome

La classe `ClientLauncher` sert de point d'entrée au JAR exécutable et délègue à `ClientApplication`, qui est la classe principale JavaFX. Cette approche permet de créer un JAR qui inclut JavaFX et peut être lancé de manière autonome sur différentes plateformes.

Connexion à une Pizzeria via MQTT

La classe `MQTTClient` gère toute la communication avec le serveur Pizzeria via le protocole MQTT. Elle offre plusieurs fonctionnalités clés :

- Établir de la connexion au broker MQTT et à une pizzeria ;
- Envoyer une demande de menu dans le broker (via la méthode `requestMenu()` qui retourne un CompletableFuture complété une fois le menu reçu) ;
- S’abonner à la réception du menu et recevoir (via la méthode `handleMenuResponse`, qui désérialise les données et complète la `Future`) ;
- Envoyer une commande au serveur (via la méthode `sendOrder`) ;
- Recevoir les notifications pour les étapes (s'abonne aux topics de suivi avec `sendOrder`) ;
- Interagir avec les différents contrôleurs via des Runnable/Consumer et accept() ;
- Gérer les erreurs de connexion (timeouts). Sans réponse du serveur dans les délais, `TimeoutTimer` renvoie une erreur affichée sous forme de popup dans l’interface, ce qui évite de rester bloqué indéfiniment.

Mise en place des vues

L'interface utilisateur est construite selon le pattern MVC (Modèle-Vue-Contrôleur) :

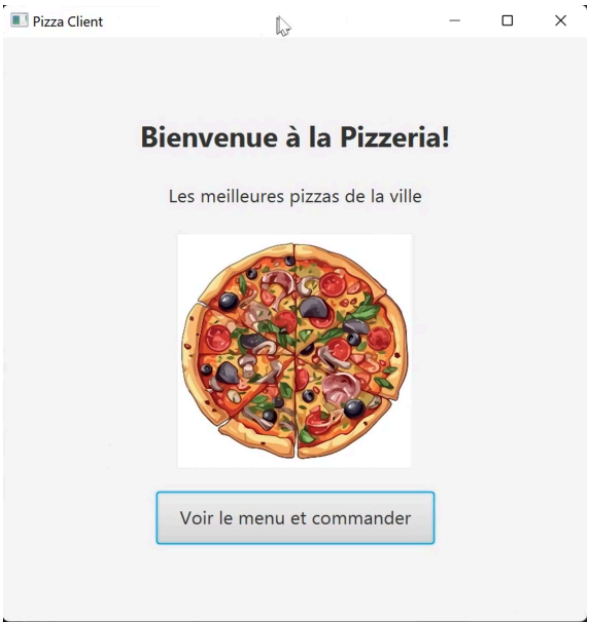
- Les vues sont définies dans des fichiers FXML
- Les contrôleurs gèrent la logique d'interaction avec ces vues
- Les modèles (`Pizza` et `Order` du module `common`) représentent les données

Nous avons d'abord développé une seule vue et un seul contrôleur pour les différentes étapes d'utilisation de l'application (accueil, menu, commande et livraison). Par la suite, nous les avons séparé en 3 scènes avec leurs contrôleurs respectifs pour coller à la maquette d'interface fournie.

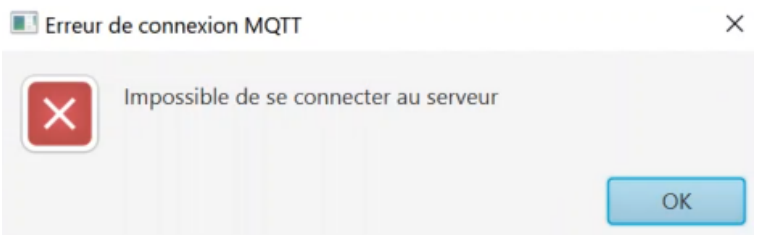
Écran d'accueil

L'écran d'accueil (welcome-view.fxml) est celui qui s'affiche au lancement de l'application.

Il présente un titre, un logo et un bouton pour commencer la commande. Son contrôleur (WelcomeController) permet, lorsque l'utilisateur clique sur ce bouton, de passer vers l'écran de commande.



Si le Client n'arrive pas à se connecter au broker MQTT, un pop-up le notifie à l'utilisateur :

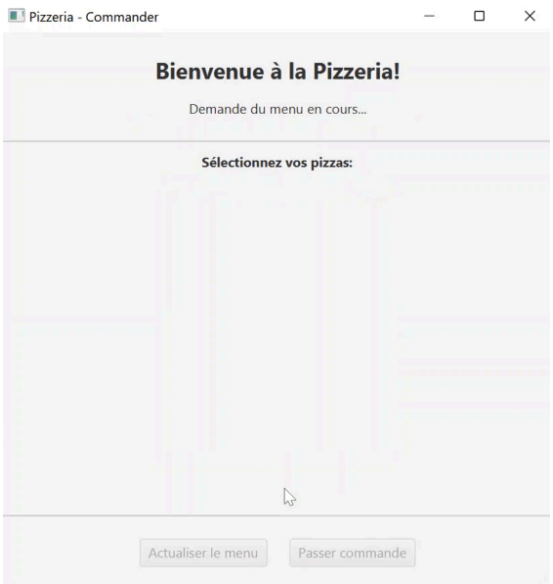


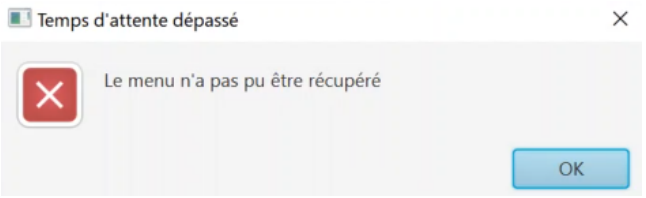
Écran de commande

L'écran de commande (`order-view.fxml`) se compose d'un titre, d'un texte de statut, d'une zone contenant la liste des pizzas et de 2 boutons “Actualiser le menu” et “Passer commande”. Le contrôleur (`OrderController`) gère :

- La demande du menu via MQTT
- L'affichage dynamique des pizzas disponibles
- La création et l'envoi de la commande

La zone listant les pizzas est initialement vide en attendant que le menu soit chargé. Si le menu n'a pas pu être récupéré dans un délai fixé, un pop-up le notifie à l'utilisateur.

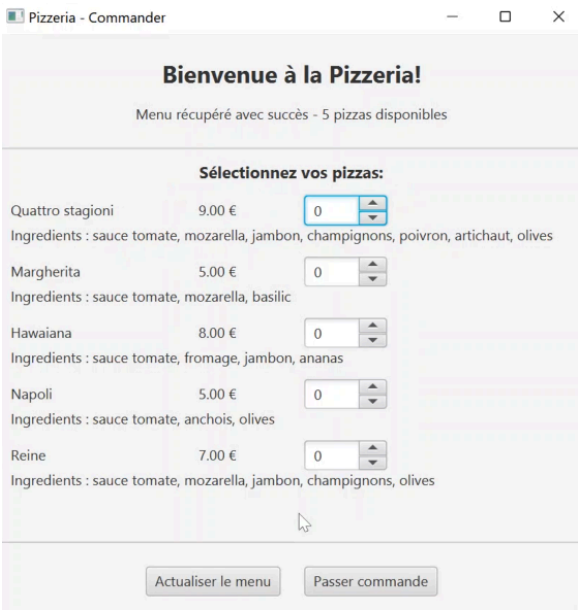




Une fois la liste des pizzas récupérée, le menu des pizzas disponibles s'affiche automatiquement sur la page. En appuyant sur le bouton “Actualiser le menu”, la page se recharge et une nouvelle demande de menu est effectuée.

L'utilisateur peut alors voir les pizzas disponibles ainsi que les ingrédients composant chaque type de pizza ainsi que leur prix. Il peut utiliser les champs de type numérique pour choisir le nombre de pizzas de chaque type qu'il souhaite.

En appuyant sur le bouton “Passer commande”, l'utilisateur envoie sa commande contenant les informations sur les pizzas et la quantité choisie pour chacun des types. Cela a pour effet de passer à la page de suivi de commande. Cette action ne peut s'effectuer que si l'utilisateur a sélectionné au moins une pizza, sans quoi la commande ne sera pas envoyée et un popup l'indiquant sera affiché.



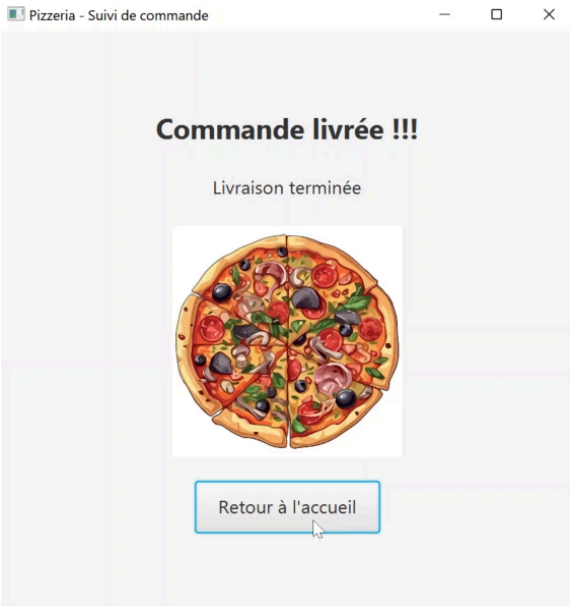
Écran de suivi de commande

L'écran de suivi de commande (`waiting-view.fxml`) affiche l'état de la commande en temps réel. Le contrôleur (`WaitingController`) :

- Reçoit les notifications de statut via MQTT
- Met à jour l'interface en fonction de l'état de la commande
- Gère les animations (logo clignotant pendant l'attente)

Lorsque l’utilisateur arrive sur la page de Suivi de commande, il peut suivre l’état de sa commande en temps réel (préparation, cuisson, en cours de livraison, livrée).

Lorsque la livraison est effectuée, un bouton s'affiche et l'utilisateur peut cliquer dessus pour retourner à l'accueil. Si une des pizzas de la commande n'a pas pu être préparé (notamment les pizzas Hawaiana qui contiennent de l'ananas), la page indique quelles pizzas n'ont pas été livrées.



Liaison FXML et contrôleurs

Les qualificateurs `@FXML` présents dans les contrôleurs permettent de lier des attributs du contrôleur à des éléments JavaFX de la vue FXML ayant le même identifiant. Cette approche permet une séparation claire entre la définition de l'interface (FXML) et sa logique (contrôleur).

Par exemple, dans OrderController :

```
public class OrderController {
    @FXML
    private Label statusLabel;
```

```
<!-- En-tête -->
<top>
    <VBox alignment="CENTER" spacing="10">
        <padding>
            <Insets top="20" right="20" bottom="20" left="20"/>
        </padding>
        <Label text="Bienvenue à la Pizzeria!" style="-fx-font-size: 20px; -fx-font-weight: bold;"/>
        <Label fx:id="statusLabel" text="Prêt à commander"/>
    </VBox>
</top>
```

L’objet status Label de type Label (de la bibliothèque JavaFX) est lié par le qualificateur @FXML au <Label fx:id=”StatusLabel”>, ce qui signifie que l’on peut interagir avec l’élément Label d’id statusLabel depuis l’attribut statusLabel de l’objet OrderController.

Communication entre composants

La communication entre les différents composants de l'application est gérée par :

- Injection de dépendances : Le `MQTTClient` est créé dans `OrderController` et transmis à `WaitingController`
- Callbacks : Le `MQTTClient` utilise des callbacks (`Consumer<T>`) pour notifier les contrôleurs des événements
- `CompletableFuture` : Pour les opérations asynchrones comme la récupération du menu

Cette architecture permet de séparer la logique de communication et la logique d'interface utilisateur, ce qui rend l’application maintenable et évolutive.

Gestion des erreurs

L'application implémente une gestion des erreurs à plusieurs niveaux :

- Timeouts : Détection des problèmes de communication avec le serveur
- Alertes utilisateur : Affichage de messages d'erreur explicites

Traitement des commandes par la Pizzeria

Le module Pizzeria est responsable de la réception, du traitement et du suivi des commandes de pizzas. Il s'agit d'un composant serveur qui interagit avec le client via MQTT et avec le module Pizzaiolo pour la préparation des pizzas.

Vue globale du module

La structure du module Pizzeria est la suivante :

```
pizzeria
|- src
|  |- main
|     |- java
|        |- com
|           |- pizzeria
|              |- App.java
|              |- MQTTServer.java
|              |- PizzaioloIngredientAdapter.java
|- libs
|  |- pizzaiolo.jar
|  |- pizzaiolo-javadoc.jar
|- pom.xml
|- README.md
```

Le module est composé de trois classes principales :

- `App.java` : Point d'entrée de l'application qui initialise et démarre le serveur MQTT
- `MQTTServer.java` : Classe principale qui gère la connexion MQTT, le traitement des commandes et l'interaction avec le Pizzaiolo
- `PizzaioloIngredientAdapter.java` : Adaptateur qui fait le lien entre les ingrédients du Pizzaiolo et le modèle de données

Le Pizzaiolo `pizzaiolo.jar` est une librairie externe utilisée par la classe `MQTTServer`. Sa documentation est contenue dans `pizzaiolo-javadoc.jar`.

Configuration et point d’entrée

Le fichier `pom.xml` configure le module comme un projet Maven avec les dépendances nécessaires :

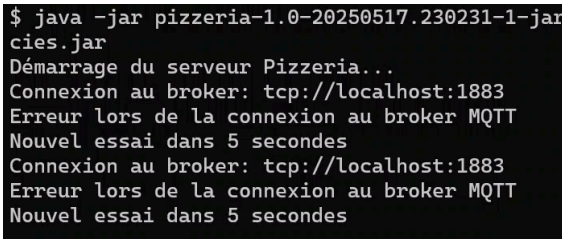
- Le module commun pour partager les modèles de données
- La bibliothèque MQTT pour la communication
- Le JAR du Pizzaiolo comme dépendance système

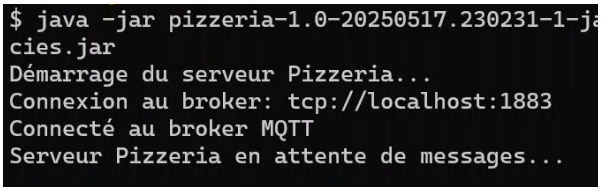
La configuration permet de créer un JAR exécutable avec toutes les dépendances incluses, ce qui facilite le déploiement. Le plugin `maven-antrun-plugin` est utilisé pour extraire le contenu du JAR du Pizzaiolo et l'inclure dans notre JAR final.

La classe `App` sert de point d'entrée à l'application. Elle initialise le serveur MQTT et configure un hook d'arrêt pour assurer une fermeture propre de la connexion lors de l'arrêt de l'application.

Connexion au serveur MQTT

La classe `MQTTServer` gère la connexion au broker MQTT et le traitement des messages. Elle implémente une logique de reconnexion toutes les 5 secondes en cas d'échec.





La communication avec le client se fait via des topics MQTT spécifiques :

- `bcast/i_am_ungry` : Réception des demandes de menu
- `bcast/menu` : Envoi du menu
- `orders/+` : Réception des commandes
- `orders/{id}/status/{status}` : Envoi des mises à jour de statut
- `orders/{id}/delivery` : Notification de livraison
- `orders/{id}/cancelled` : Notification d'annulation

Gestion du catalogue de pizzas

Au démarrage, le serveur initialise un catalogue de pizzas en utilisant les informations fournies par le Pizzaiolo. Cette liste sera ensuite transmise lors d'une demande Client.

```
Demande de menu reçue
Menu envoyé: margherita|sauce tomate,mozarella,basilic|5;qua
ttro stagioni|sauce tomate,mozarella,jambon,champignons,poiv
ron,artichaut,olives|9;reine|sauce tomate,mozarella,jambon,c
hampignons,olives|7;napoli|sauce tomate,anchois,olives|5;haw
aiana|sauce tomate,fromage,jambon,ananas|8
```

La classe `PizzaioloIngredientAdapter` est utilisée pour convertir les ingrédients du format du Pizzaiolo vers la classe `Pizza` de notre modèle de données. Cette approche suit le patron de conception Adaptateur, ce qui permet d'intégrer facilement le Pizzaiolo tout en gardant un modèle commun au Client et à la Pizzeria.

Traitement des commandes

Lorsqu'une commande est reçue, elle est traitée dans un thread séparé pour permettre le traitement parallèle de plusieurs commandes.

La méthode `processOrder` gère le cycle de vie complet d'une commande :

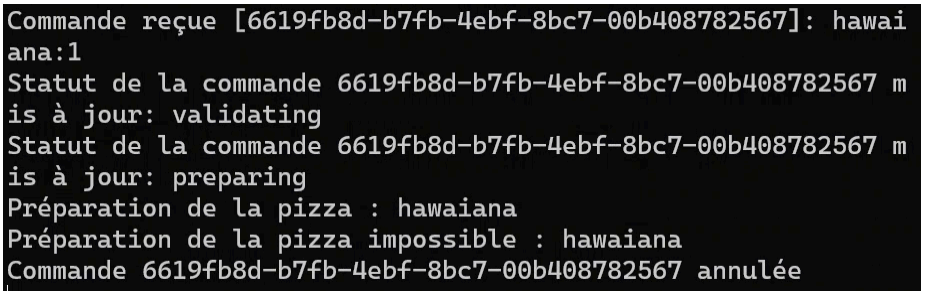
- Validation : Vérification que les pizzas commandées existent et que les quantités sont valides
- Préparation : Pour chaque pizza commandée, demande au Pizzaiolo de la préparer
- Cuisson : Cuisson des pizzas préparées
- Livraison : Simulation d'un temps de livraison et notification au client

À chaque étape, le statut de la commande est mis à jour et publié sur le topic MQTT correspondant pour informer le Client.

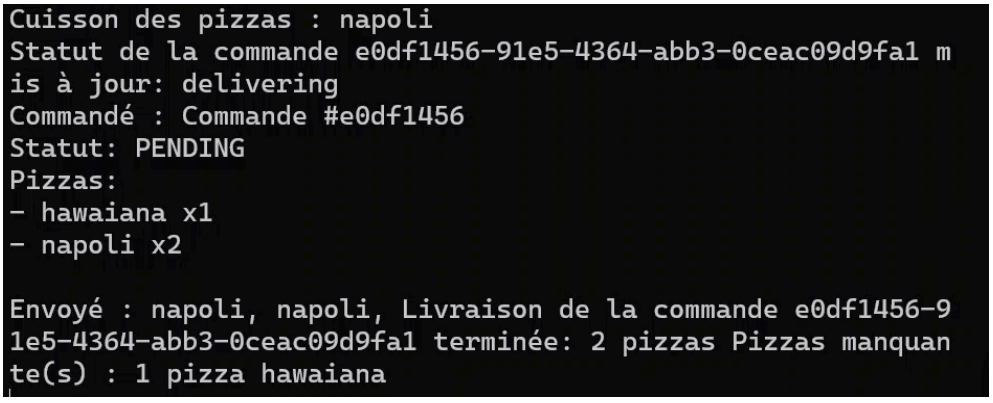
```
Commande reçue [bc1dffed-a05a-4073-b8da-0ec2316b950f]: margh
erita:1,napoli:2
Statut de la commande bc1dffed-a05a-4073-b8da-0ec2316b950f m
is à jour: validating
Statut de la commande bc1dffed-a05a-4073-b8da-0ec2316b950f m
is à jour: preparing
Préparation de la pizza : margherita
Statut de la commande bc1dffed-a05a-4073-b8da-0ec2316b950f m
is à jour: cooking
Cuisson des pizzas : margherita
```

Le système gère également plusieurs cas d'erreur :

- Commande invalide ou vide : Si la validation échoue, ou si aucune pizza ne peut être préparée, la commande est annulée



- Ingrédients indisponibles : Si certaines pizzas ne peuvent pas être préparées, la commande est partiellement livrée avec une notification des pizzas manquantes.



Interaction avec le Pizzaiolo

L'interaction avec le Pizzaiolo est gérée par la méthode `processOrder`. Pour chaque pizza commandée, le serveur :

- Récupère les détails de la pizza dans le catalogue
- Demande au Pizzaiolo de préparer la pizza
- Demande au Pizzaiolo de cuire la pizza

La synchronisation est utilisée pour la cuisson afin d'éviter des problèmes de concurrence avec le Pizzaiolo, qui n'est pas thread-safe.

Extensions du système

Gérer plusieurs commandes en parallèle pour un même client

Actuellement, notre système permet à un client de passer une commande à la fois. Pour permettre à un client de gérer plusieurs commandes en parallèle, plusieurs modifications seraient nécessaires.

Côté Client, au niveau de la communication MQTT :

- S’assurer que chaque instance Client a un identifiant unique et persistant.
- S'abonner à plusieurs topics de suivi de commande simultanément.
- Modifier la structure des callbacks pour router les notifications vers la bonne commande.

Côté Client, au niveau de l’interface utilisateur :

- Ajouter un écran de gestion des commandes (nouvelle vue FXML et contrôleur associé) pour afficher toutes les commandes actives du client.
- Permettre à l'utilisateur de revenir à l'écran de commande sans annuler sa commande en cours.
- Ajouter des badges ou notifications pour indiquer le nombre et l’état des commandes actives.

Côté Pizzeria :

- Associer les commandes à l'identifiant Client pour permettre le suivi.
- Ajuster le pool de threads pour gérer efficacement plusieurs commandes simultanées.

Côté Modèle de données :

- Ajouter un identifiant Client dans la classe `Order`.
- Modifier l'énumération `OrderStatus` pour gérer des états plus complexes.

Gérer plusieurs Pizzerias

Pour étendre notre système à plusieurs Pizzerias, des modifications touchant la structure de communication et à la découverte de services seraient nécessaires.

Au niveau de l'architecture MQTT :

- Ajouter un identifiant de pizzeria dans la structure des topics :
  - `pizzeria/{pizzeriaId}/bcast/menu`
  - `pizzeria/{pizzeriaId}/orders/{orderId}/status/{status}`
- Créer un topic pour la découverte des pizzerias disponibles : `pizzeria/discovery`

Au niveau de l'interface Client :

- Ajouter un écran pour choisir la pizzeria avant de passer commande.

Au niveau du modèle de données :

- Créer une classe pour représenter les informations d'une Pizzeria.
- Ajouter un identifiant de Pizzeria dans la classe `Order`.

Au niveau de la Pizzeria :

- Chaque instance de serveur doit avoir un identifiant unique.
- Chaque Pizzeria pourrait avoir ses propres paramètres (menu, temps de préparation, etc.).
- Développer une logique pour répartir les commandes entre les Pizzerias selon leur charge.

Conclusion

Le projet "Let's make pizza!" nous a permis de développer un système distribué complet de commande et livraison de pizzas basé sur une architecture événementielle.

Ce projet nous a permis de comprendre le fonctionnement de la communication découplée entre plusieurs entités via le protocole MQTT. Nous avons ainsi développé une application robuste et maintenable en utilisant des technologies de développement modernes. De plus, la mise en place d'un projet multi-modules avec Maven a facilité la gestion des dépendances et la réutilisation du code.