HEPTATHLON

Architecture client - serveur

Résumé

L'objectif de ce projet est de développer un système informatique pour la gestion du stock de marchandises et la préparation des factures de ventes pour l'entreprise Heptathlon, spécialisée dans la vente d'articles de sport et de loisir. Le système doit être architecturé selon le modèle client-serveur et permettre une gestion persistante des informations de stock et de facturation.

Théo BONTEMPS, Dorian DESCAMPS

theo.bontemps@etud.u-picardie.fr, dorian.descamps@etud.u-picardie.fr

1. Table des matières

2. Cahier des charges 3
2.1. Fonctionnalités du système 3
2.1.1. Gestion des données 3
2.1.2. Opérations sur les données 3
2.2. Architecture du système 3
2.3. Gestion des données 4
2.4. Gestion des échanges client/serveur 4
2.5. Technologies à utiliser 4
2.6. Consignes spécifiques 4
3. Architecture 5
3.1. Architecture logique 5
3.2. MCD 5
4. Backend 7
4.1. RMI 7
4.1.1. Technologie 7
4.1.2. Bonnes pratiques 7
4.2. Bases de données 7
4.2.1. Technologie 7
4.2.2. Synchronisation 8
4.2.3. Bonnes pratiques 8
5. Frontend 9
5.1. Technologies 9
5.2. Bonnes pratiques 9
5.3. Présentation du produit (& captures d'écran) 10
6. Procédure d'utilisation 15
6.1. Prérequis 15
6.2. Lancement des bases de données 15
6.3. Arguments des programmes 15
6.3.1. Serveur principal 15
6.3.2. Serveur client 15
6.3.3. Client final 16
6.4. Lancement des programmes 16
6.4.1. MacOS 16
6.4.2. Windows / Linux 17

6.5. Ajout de serveurs clients 17
6.5.1. Base de données 17
6.5.2. Serveur 17
6.5.3. Dossier ressources 17
7. Conclusion 18

2. Cahier des charges

2.1. Fonctionnalités du système

2.1.1. Gestion des données

2.1.1.1. Gestion du stock

Le système doit gérer une base de données contenant les informations suivantes pour chaque article :

- Référence précise de l'article
- Famille de l'article
- Prix unitaire de l'article
- Nombre total d'exemplaires en stock

2.1.1.2. Facturation des clients

Le système doit gérer les factures clients avec les informations suivantes :

- Total de la facture pour un client
- Détail des articles achetés
- Mode de paiement
- Date de facturation

2.1.2. Opérations sur les données

Le système doit permettre les opérations suivantes :

- Consulter le stock d'un article : récupérer les informations d'un article (quantité en stock, prix unitaire, etc.) à partir de sa référence.
- Rechercher un article : récupérer toutes les références des articles d'une famille donnée dont le stock n'est pas nul.
- Acheter un article : un client doit pouvoir acheter un article en stock.
- Payer une facture : un client peut payer ce qu'il doit au magasin.
- Consulter une facture : possibilité de voir la facture.
- Calculer le chiffre d'affaires : à une date donnée en fonction des factures de cette date.
- Ajouter un produit : ajouter des exemplaires d'un produit existant dans le catalogue.

Les prix des articles sont mis à jour tous les matins par le serveur principal, et les factures sont sauvegardées chaque soir sur ce serveur.

2.2. Architecture du système

Le système doit comporter les éléments suivants :

- Serveur principal au siège de l'entreprise.
- Serveur client dans chaque magasin.
- Clients (caisses) dans chaque magasin.

Le processus de communication client-serveur se déroule ainsi :

1. L'utilisateur saisit une requête sur le poste client.
2. La requête est envoyée au serveur.
3. Le serveur reçoit et traite la requête.

4. Le serveur effectue le traitement et envoie le résultat au client.
5. Le client reçoit le résultat et peut effectuer une nouvelle requête.

2.3.Gestion des données

Le serveur du magasin est responsable de la gestion des données de la boutique (stock et facturation) et utilise un SGBD (MySQL) pour stocker ces données.

2.4.Gestion des échanges client/serveur

Les échanges entre les clients et le serveur doivent suivre un protocole défini pour assurer la compréhension mutuelle des requêtes et des résultats. Les échanges doivent se faire au niveau de l'application.

2.5.Technologies à utiliser

- Langage de programmation : Java
- Middleware : RMI (Remote Method Invocation)
- SGBD : MySQL

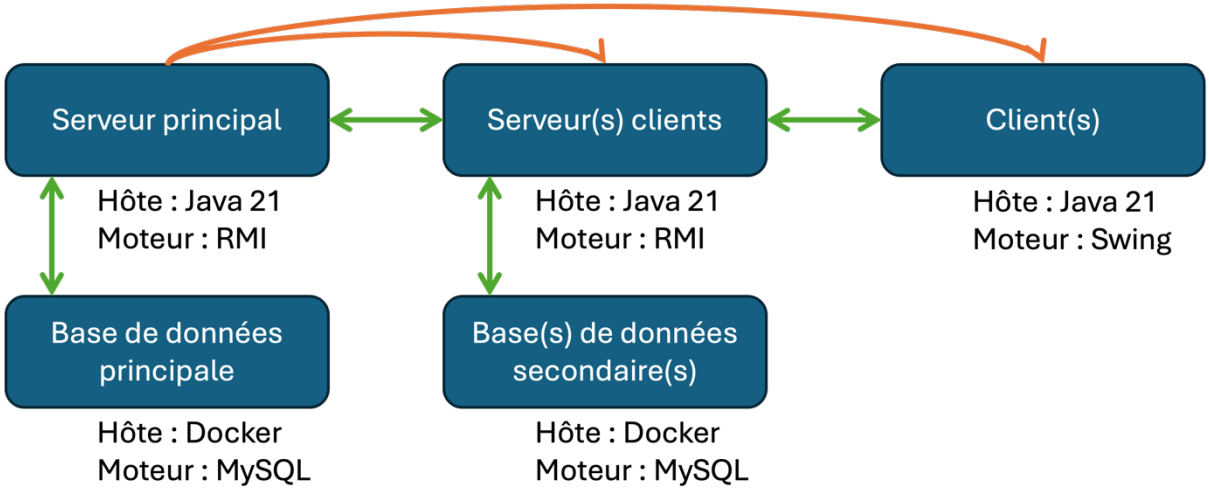
2.6.Consignes spécifiques

- Le système doit être robuste, sécurisé et capable de gérer des échanges en temps réel entre les différents composants (clients et serveurs).
- Une interface utilisateur intuitive et efficace doit être développée pour les postes clients.

3. Architecture

3.1. Architecture logique

S'agissant d'une architecture ne nécessitant pas de privilèges, et donc pas d'identification, nous avons choisis de réaliser les traitements du côté client afin de simplifier l'évolutivité de l'interface.
Ainsi, il est possible d'apporter des modifications de logique importantes au niveau client sans avoir la nécessité de modifier l'intégralité de la chaîne.
Ci-après, une figure présentant l'architecture matériel et réseau mise en place.
À noter qu'un serveur client représente le serveur d'un magasin, quand le serveur principal est celui qui regroupe les données de tous les magasins. Un client représente lui une caisse.

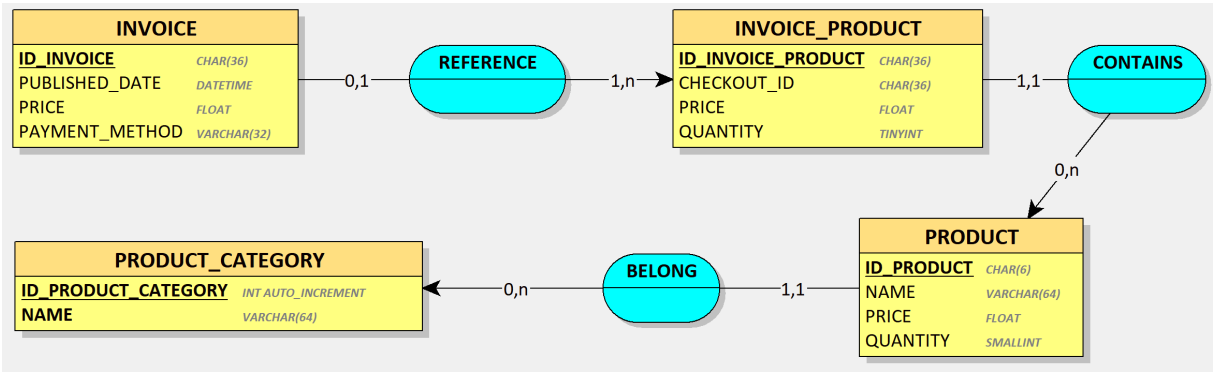


- ❑ **Partage d'interfaces et de classes à la compilation**
- ❑ **Communication réseau permanente**

Les flèches oranges indiquent que des interfaces et des classes sont partagées entre les composants à la compilation. Cela signifie que certains composants doivent avoir des définitions de classes ou d'interfaces communes pour permettre l'interopérabilité via RMI.
Les flèches vertes montrent les canaux de communication réseau permanents entre les différents composants. Les serveurs principaux et clients communiquent en permanence avec leurs bases de données respectives pour les opérations de données, tandis que les serveurs clients ont également une connexion permanente avec le serveur principal pour relayer les requêtes des clients finaux. Le client n'est lui relié qu'à son serveur client respectif.

3.2. MCD

Les différentes bases de données, qu'elles soient relatives au serveur principal, ou à un serveur secondaire utilisent exactement les mêmes tables. L'unique différence réside dans la quantité de chaque produit disponible : celle-ci est définie à -1 sur le serveur principal puisque la gestion des stocks est gérée au niveau du magasin.


INVOICE
ID_INVOICE CHAR(36)
PUBLISHED_DATE DATETIME
PRICE FLOAT
PAYMENT_METHOD VARCHAR(32)
0,1
REFERENCE
1,n
INVOICE_PRODUCT
ID_INVOICE_PRODUCT CHAR(36)
CHECKOUT_ID CHAR(36)
PRICE FLOAT
QUANTITY TINYINT
1,1
CONTAINS
0,n
PRODUCT
ID_PRODUCT CHAR(6)
NAME VARCHAR(64)
PRICE FLOAT
QUANTITY SMALLINT
1,1
BELONG
0,n
PRODUCT_CATEGORY
ID_PRODUCT_CATEGORY INT AUTO_INCREMENT
NAME VARCHAR(64)

4. Backend

4.1. RMI

4.1.1. Technologie

Le cahier des charges imposait RMI comme méthode de communication inter-serveur.

4.1.1.1. Présentation

RMI, ou Remote Method Invocation, est une technologie Java permettant à un programme sur une machine virtuelle Java d'invoquer des méthodes sur un objet situé sur une autre machine virtuelle. Développée par Sun Microsystems (aujourd'hui Oracle Corporation), RMI facilite la communication entre applications distribuées en offrant une abstraction des mécanismes de réseau sous-jacents. Cette technologie utilise les principes d'appel de méthode à distance pour permettre aux développeurs de créer des applications distribuées de manière transparente, en se concentrant sur la logique métier plutôt que sur les détails de la communication réseau.

4.1.1.2. Pertinence

Pour notre projet, qui nécessitait une communication fiable et performante entre différents serveurs, RMI s'est révélé être un choix pertinent. La technologie RMI offre une intégration native avec Java, ce qui simplifie grandement le développement et la maintenance de notre système, étant donné que notre projet est principalement basé sur cette plateforme. De plus, RMI est réputé pour sa facilité de mise en œuvre et ses performances solides, ce qui était crucial pour assurer des échanges rapides et sécurisés entre nos serveurs. Enfin, la robustesse et la fiabilité de RMI garantissent une communication continue et sans faille, essentiel pour le bon fonctionnement et la cohérence de notre projet.

4.1.2. Bonnes pratiques

Nous avons prêté une attention particulière à limiter au maximum la duplication de code. De cette manière, nous avons créé 3 projets distincts que sont le serveur principal, le serveur secondaire, et l'application client. Cependant, ces 3 projets partagent des dépendances au travers de la compilation (suivant la figure explicative précédemment donnée). Ainsi, il n'est pas nécessaire de dupliquer les classes. Le serveur principal héberge, de ce fait, exactement le même service RMI que le serveur client.
De la même manière, nous avons fait très attention à découper le code au maximum, afin de rendre l'algorithmie la plus lisible possible.
Enfin, des tests en condition réelle ont révélé qu'il était important de créer sa propre usine de sockets RMI, afin de les adapter à nos besoins. Cette usine permet de configurer les connexions RMI de la meilleure des manières, et permet, par exemple, ainsi de désactiver une déconnexion pour inactivité.

4.2. Bases de données

4.2.1. Technologie

Le cahier des charges imposait l'utilisation de MySQL comme SGBD.

4.2.1.1. Présentation

MySQL est un système de gestion de bases de données relationnelles (SGBDR) open source largement utilisé dans le monde de la technologie. Développé par Oracle Corporation, il repose sur le langage SQL pour interagir avec les données de manière efficace et structurée. MySQL se distingue par sa simplicité d'utilisation, sa rapidité, et sa fiabilité. Il est conçu pour être léger, permettant une installation et une configuration faciles.

4.2.1.2. Pertinence

Pour ce projet, dont le schéma de données est relativement simple, et dont la quantité d'information persistante est également faible, MySQL convenait parfaitement. Tout d'abord, son statut de logiciel open source signifie qu'il est gratuit. Ensuite, MySQL offre des performances solides même avec des ressources limitées, assurant des temps de réponse rapides et une gestion efficace des données. Enfin, la sécurité et la fiabilité de MySQL garantissent que nos données sont protégées et accessibles, ce qui a été essentiel pour le bon développement du projet.

4.2.2. Synchronisation

Une synchronisation interbase est réalisée à 2 moments donnés dans le cahier des charges. Il a été à notre charge de définir précisément les heures de synchronisation.

- La mise à jour des prix, à partir du serveur principal, vers les serveurs clients est réalisée à 22h. Celle-ci est ordonnée par un serveur client, en suivant le schéma donné ci-après.
    1. Requête du serveur client pour récupérer les données sur les produits.
    2. Réponse de sa base de données avec les données sur les produits.

    3. Requête du serveur client pour récupérer les données sur les produits.
    4. Requête du serveur principal pour récupérer les données sur les produits.
    5. Réponse de sa base de données avec les données sur les produits.
    6. Réponse du serveur principal avec les données sur les produits.

    7. Lorsqu'il existe une différence :
        - Requête du serveur client pour mettre à jour les données sur le produit.
        - Réponse de la base de données confirmant les modifications.

- La synchronisation des factures est-elle réalisée le soir, à 22h. Elle consiste en 2 opérations distinctes : le transfert des données brutes, puis les fichiers qui y sont liés. Ces opérations sont toutes deux, également demandées par un serveur client et suivent pratiquement le même schéma que celui donné précédemment. La seule différence réside dans l'ajout d'un transfert de fichier via RMI sous la forme d'un tableau de bits.

4.2.3. Bonnes pratiques

Nous avons suivi 2 bonnes pratiques particulièrement notables dans l'implémentation de notre MCD, afin d'assurer l'unicité et la cohérence des données.

- Éviter toute duplication de données.
- Maximiser le nombre de contraintes (indexes primaires ou non et clés étrangères).

5. Frontend

5.1. Technologies

Pour ce qui est de la partie visuelle de l'application, nous avions plusieurs choix a notre disposition, à savoir :

- AWT : Bibliothèque ancienne limitée et dépréciée mais compatible sur tous les OS
- Swing : Simple à prendre en main, avec une prise en charge complète de la conception des fenêtres visuelles dans l'IDE « Jetbrains IntelliJ »
- Java FX : Plus récente, vouée à remplacer Swing, mais plus fastidieuse à prendre en main pour un petit projet comparé a Swing. De plus, pour la conception il faut soit passer par du code, soit un logiciel externe car IntelliJ n'a pas d'éditeur intégré.

Notre choix s'est donc dirigé vers la libraire Swing. Par sa simplicité d'utilisation et elle dispose de tous les éléments graphiques nécessaires à la création du logiciel.

Pour une partie du projet, à savoir le calcul du chiffre d'affaires sur une période donnée, nous avons trouvé une librairie externe qui affiche un calendrier pour sélectionner un jour. Cette librairie est JdatePicker, ancienne par son style graphique, mais nous a évité de passer par des champs de textes qui ne sont pas forcément agréable à utiliser pour des utilisateurs finaux soit recoder une fonctionnalité de calendrier.

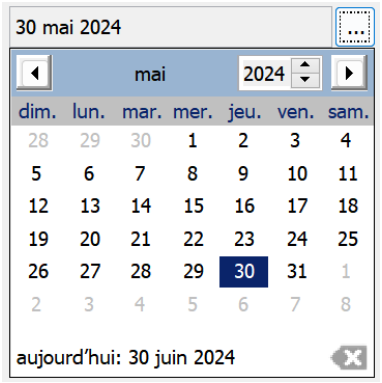


Figure 1 - Sélection de date avec JdatePicker

5.2. Bonnes pratiques

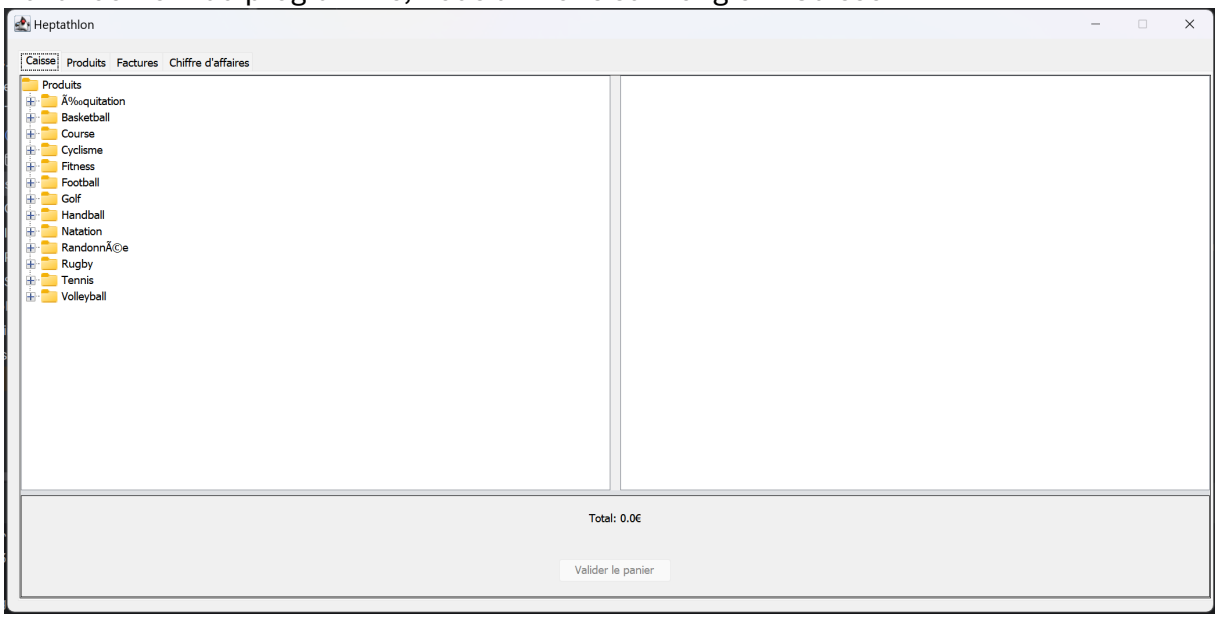
Pour assurer une interface utilisateur simple et efficace, nous avons adopté plusieurs bonnes pratiques. D'abord, nous avons clairement défini le nom des classes et donc séparé les rôles de chaque classe. Nous avons utilisé le concepteur de fenêtres intégré dans a IntelliJ pour faciliter la conception et la prévisualisation.

La gestion des connexions est centralisée dans la classe Main, ce qui permet une maintenance et un débogage simplifiés. L'interface utilisateur est cohérente, avec des méthodes uniformes pour les actions et le chargement des données, comme la méthode refreshData() utilisée dans divers onglets. Les événements utilisateur sont gérés par des listeners liés aux composants graphiques de Swing.

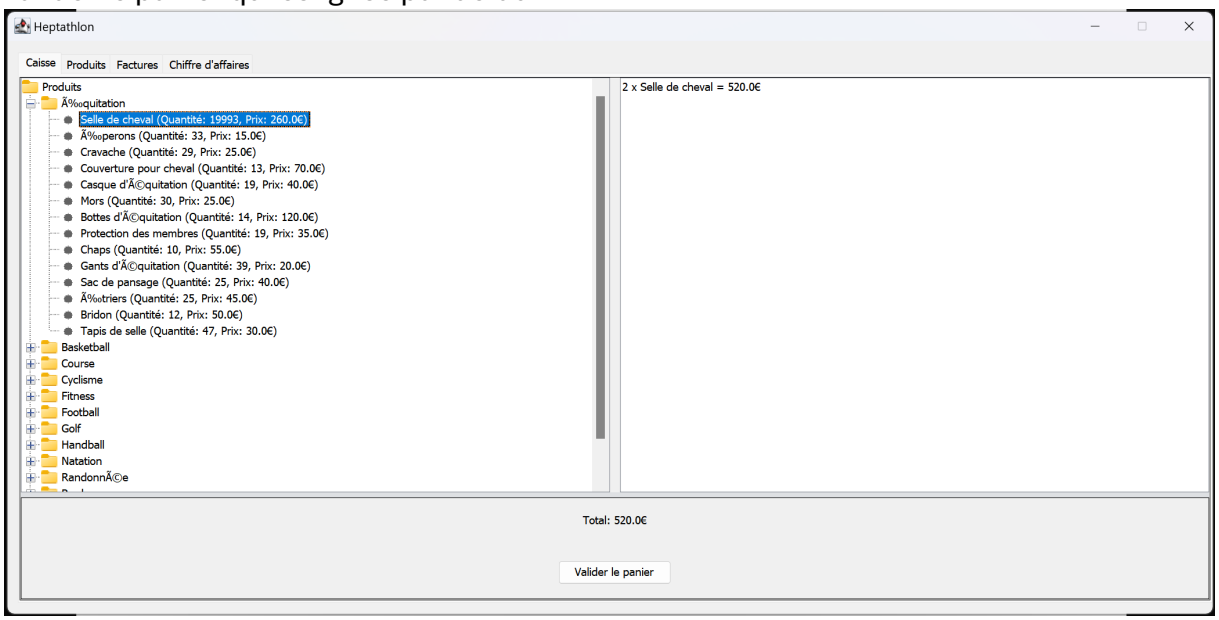
Les composants de l'application sont modulaires et réutilisables, et des dialogues de confirmation (JOptionPane) sont utilisés pour les actions critiques, offrant ainsi un retour visuel aux utilisateurs. Même si l'interface est conçue pour une dimension unique, les composants Swing sont configurés pour être redimensionnables, assurant une bonne lisibilité.

5.3. Présentation du produit (& captures d'écran)

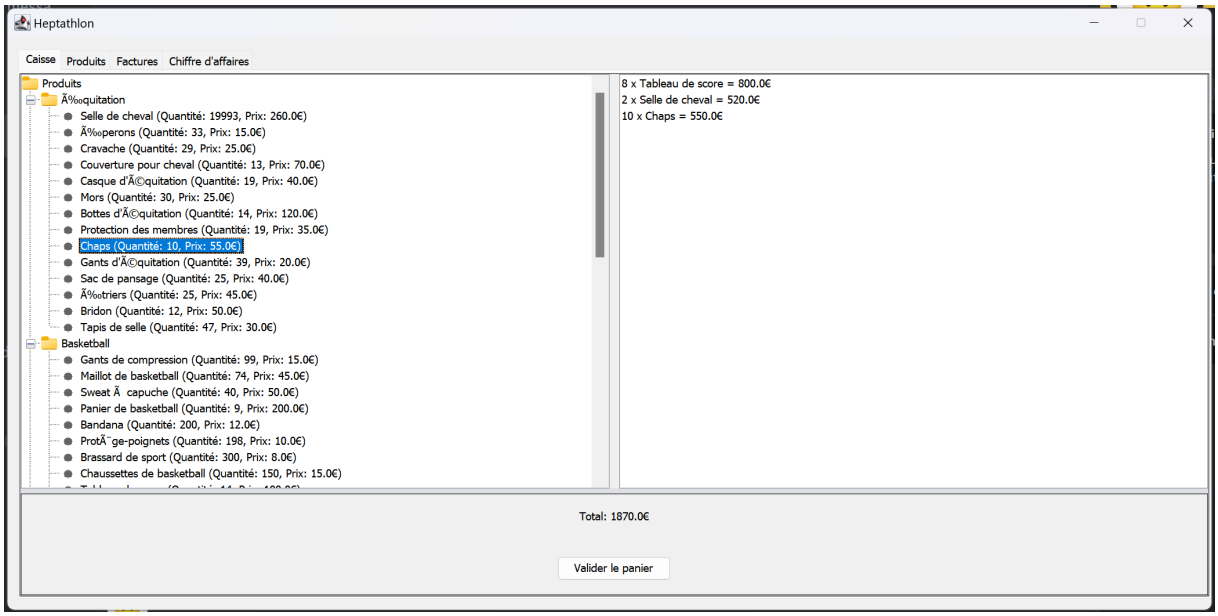
Au lancement du programme, nous arrivons sur l'onglet « Caisse » :



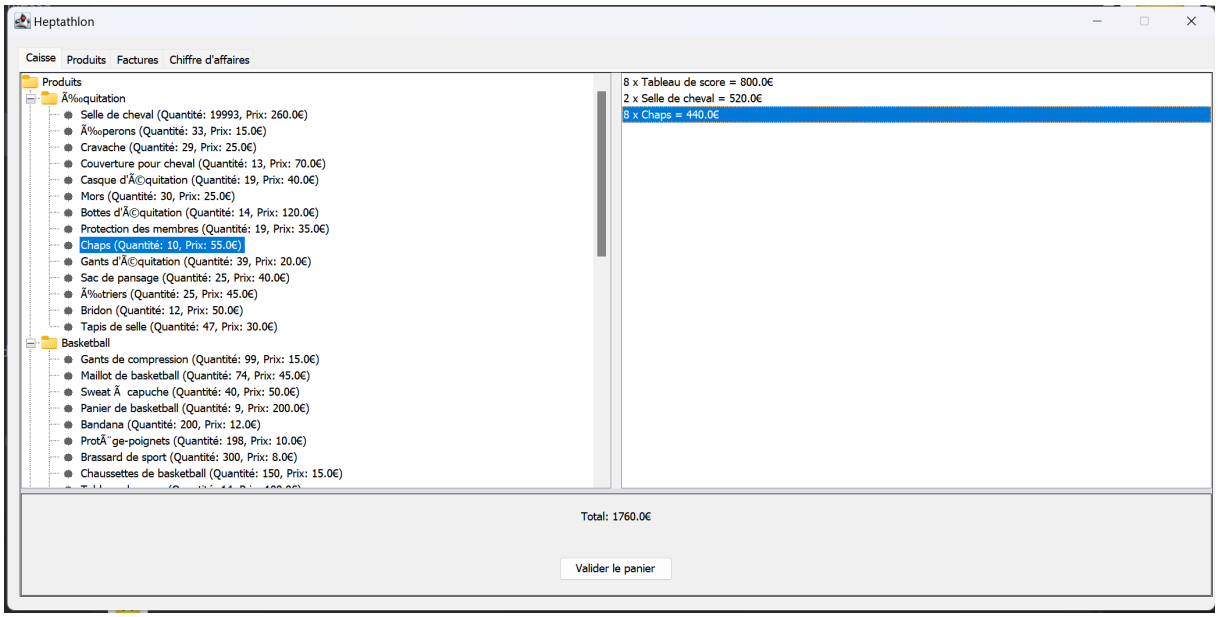
Nous avons a notre disposition 3 parties, celle de gauche qui liste les articles disponibles dans le magasin triés par catégories a droite une zone vide et en bas un total et un bouton valider le panier qui est grisé par défaut.



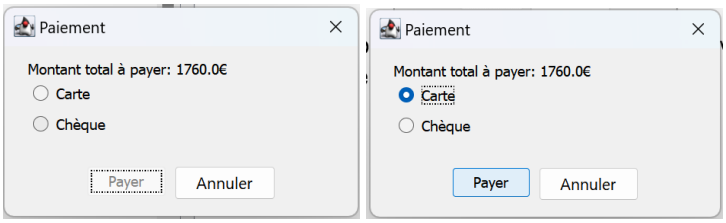
Nous avons ajouté au panier 2 selles de cheval nous voyons que nous en avons pour 520€ de selles de cheval et 520€ au total, et le bouton valider le panier s'est activé.



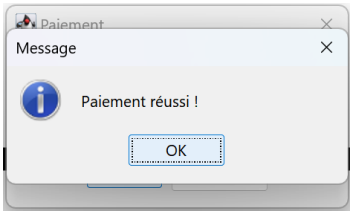
Nous avons ajouté plusieurs produits, et nous avons atteint la limite de 10 « Chaps » disponible en stock, nous ne pouvons pas en ajouter plus dans le panier.



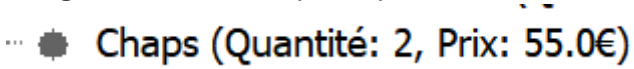
On peut supprimer une quantité d'un produit ajouté au panier, en cliquant sur un élément de la liste de droite. Le prix total s'actualise bien, ainsi que le coût unitaire multiplié par le nombre de produits ajouté au panier.



A la validation du panier, un popup nous demande de payer, avec un choix de la méthode de paiement. Le bouton payer est grisé tant que le choix n'est pas fait.

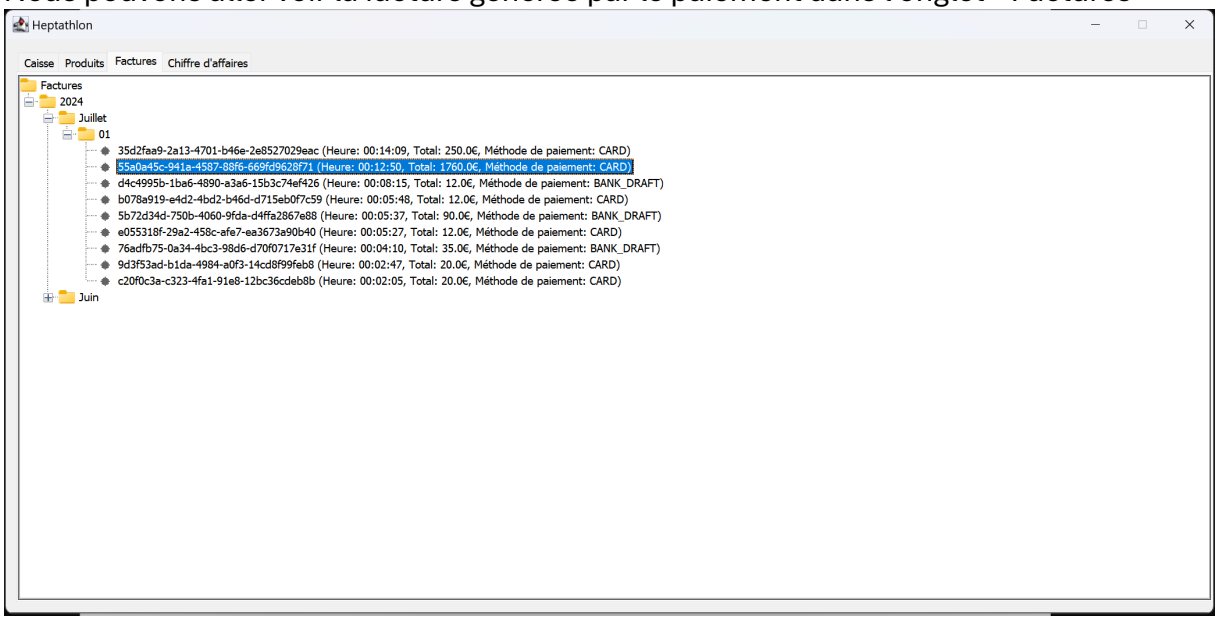


Au paiement, un message nous informe que le paiement c'est bien effectué.

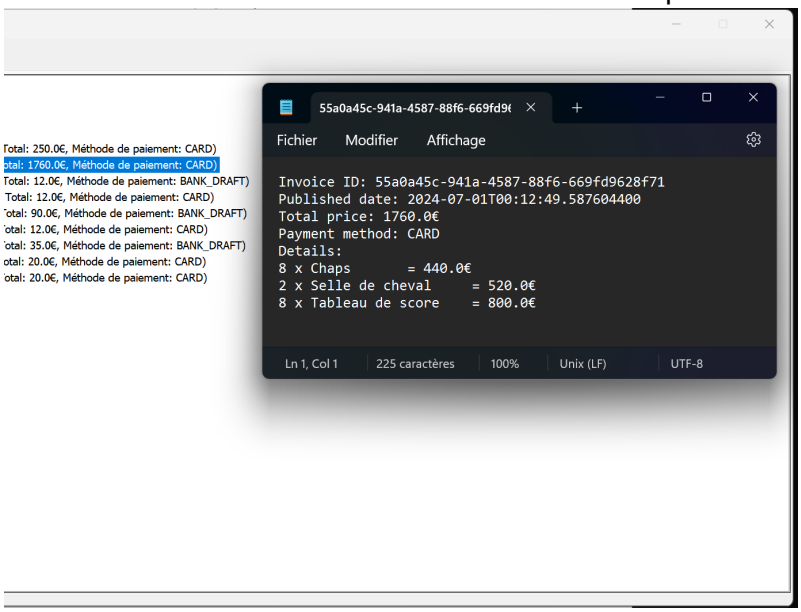


Après paiement nous remarquons que la quantité en stocks des « Chaps » par exemple a diminué (initialement 10, et un client vient d'en payer 8).

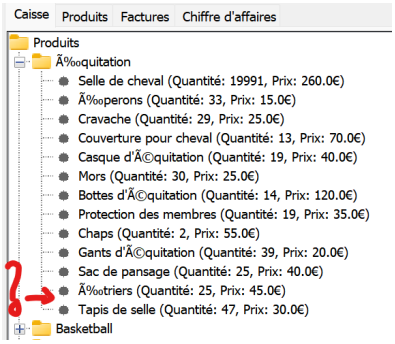
Nous pouvons aller voir la facture générée par le paiement dans l'onglet « Factures »



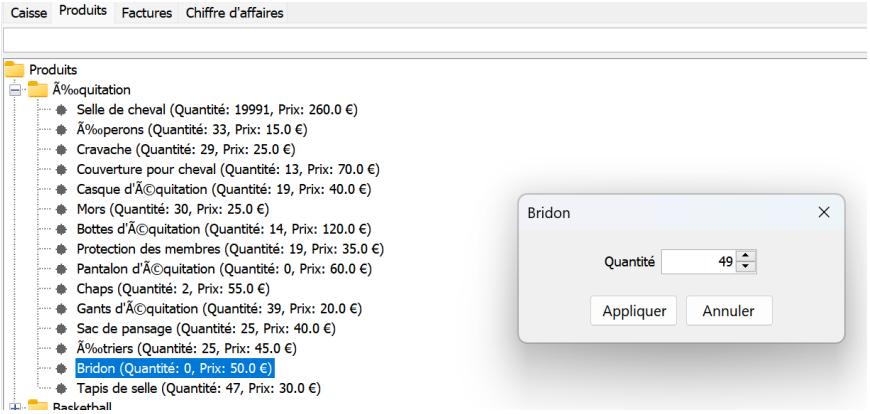
Nous retrouvons bien notre facture d'un total de 1760€ avec un paiement par Carte.



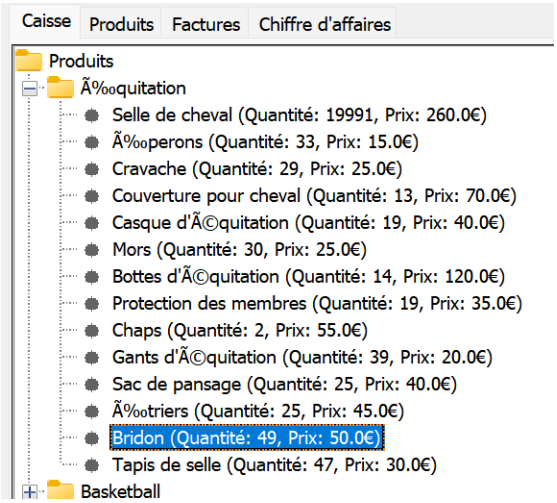
Si on double clic dessus, ça nous ouvre le fichier de facturation qui est en format .txt.



Si un client achète tout le stock de « Bridons » par exemple, ils ne sont plus visibles dans l'onglet caisse car quantité a 0, nous pouvons ajouter de la quantité en stock d'un produit depuis l'onglet « Produits ». On en ajoute 49 en stock :

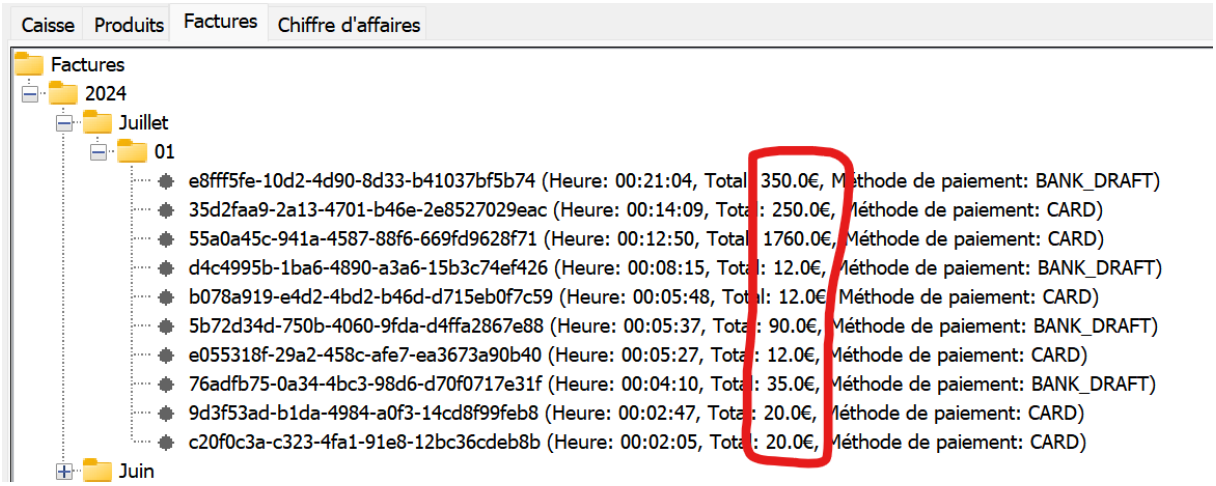


Ils sont disponibles de suite en stock a l'achat



Enfin, le magasin peut calculer son chiffre d'affaires pour une période donnée. Prenons les achats effectués aujourd'hui, nous obtenons un total de 2561€

Ce qui est vérifiable avec l'édition des factures :


Caisse
Produits
Factures
Chiffre d'affaires
Factures
2024
Juillet
01
e8fff5fe-10d2-4d90-8d33-b41037bf5b74 (Heure: 00:21:04, Total: 350.0€, Méthode de paiement: BANK_DRAFT)
35d2faa9-2a13-4701-b46e-2e8527029eac (Heure: 00:14:09, Total: 250.0€, Méthode de paiement: CARD)
55a0a45c-941a-4587-88f6-669fd9628f71 (Heure: 00:12:50, Total: 1760.0€, Méthode de paiement: CARD)
d4c4995b-1ba6-4890-a3a6-15b3c74ef426 (Heure: 00:08:15, Total: 12.0€, Méthode de paiement: BANK_DRAFT)
b078a919-e4d2-4bd2-b46d-d715eb0f7c59 (Heure: 00:05:48, Total: 12.0€, Méthode de paiement: CARD)
5b72d34d-750b-4060-9fda-d4ffa2867e88 (Heure: 00:05:37, Total: 90.0€, Méthode de paiement: BANK_DRAFT)
e055318f-29a2-458c-afe7-ea3673a90b40 (Heure: 00:05:27, Total: 12.0€, Méthode de paiement: CARD)
76adfb75-0a34-4bc3-98d6-d70f0717e31f (Heure: 00:04:10, Total: 35.0€, Méthode de paiement: BANK_DRAFT)
9d3f53ad-b1da-4984-a0f3-14cd8f99feb8 (Heure: 00:02:47, Total: 20.0€, Méthode de paiement: CARD)
c20f0c3a-c323-4fa1-91e8-12bc36cdeb8b (Heure: 00:02:05, Total: 20.0€, Méthode de paiement: CARD)
Juin

6. Procédure d'utilisation

6.1. Prérequis

- Télécharger et installer Docker Desktop.
- Télécharger un JDK (testé 21.0.3 et 22.0.1) au format « Compressed Archive » adapté à votre OS, et à l'architecture de votre processeur (Oracle).
- Télécharger les sources du projet.
- Extrayez ces fichiers en utilisant votre système, ou un outil comme 7zip.

6.2. Lancement des bases de données

Il est possible de lancer les bases de données, configurées par défaut, en exécutant le script suivant.
Il est cependant nécessaire de modifier **<CHEMIN_DES_SOURCES>** par le chemin du dossier correspondant.

```
# Lancement de la base du serveur principal
cd <CHEMIN_DES_SOURCES>/docker/main-server
docker compose down && docker compose up -d

# Lancement d'une base de serveur secondaire
cd <CHEMIN_DES_SOURCES>/docker/client-server
docker compose down && docker compose up -d
```

6.3. Arguments des programmes

Ci-après les arguments à passer aux différents programmes.
Il est à noter que les dossiers ressources correspondent pour chaque machine au dossier local permettant la sauvegarde des factures.

6.3.1. Serveur principal

Index	Type	Serveur	Valeur par défaut
1	Adresse	Principal	localhost
2	Port	Principal	1099
3	Adresse	Base de données principale	localhost
4	Port	Base de données principale	3308
5	Dossier ressources	-	-

6.3.2. Serveur client

Index	Type	Serveur	Valeur par défaut
1	Adresse	Client	localhost
2	Port	Client	1100
3	Adresse	Base de données cliente	localhost
4	Port	Base de données cliente	3307
5	Dossier ressources	-	-
6	Adresse	Principal	localhost

7	Port	Principal	1099

6.3.3. Client final

Index	Type	Serveur	Valeur par défaut
1	Dossier ressources	-	-
2	Adresse	Client	localhost
3	Port	Client	1100

6.4. Lancement des programmes

Il est possible de lancer les 3 programmes, dans leurs configurations initiales, en exécutant successivement ces 3 commandes dans 3 terminaux différents.
Il est cependant nécessaire de modifier respectivement **<CHEMIN_DU_JDK>** et **<CHEMIN_DES_SOURCES>** par les dossiers précédemment téléchargés et extraits.

Les **<DOSSIER_RESSOURCES>** doivent eux être modifiés par des chemins absolus de dossiers (comprenant le slash final) et doivent être différents pour chaque instance de programme. Il conviendra donc d'utiliser des dossiers préalablement créés. Des dossiers vides d'exemples sont déjà disponible dans le dossier /res/invoices :

- /main-server/
- /client-server/
- /client-front/

6.4.1. MacOS

```
# Lancement du serveur principal
<CHEMIN_DU_JDK>/Contents/Home/bin/java -jar
<CHEMIN_DES_SOURCES>/out/artifacts/main_server_jar/main-server.jar localhost 1099
localhost 3308 <DOSSIER_RESSOURCES>/main-server/

# Lancement d'un serveur secondaire
<CHEMIN_DU_JDK>/Contents/Home/bin/java -jar
<CHEMIN_DES_SOURCES>/out/artifacts/client_server_jar/client-server.jar localhost
1100 localhost 3307 <DOSSIER_RESSOURCES>/client-server/ localhost 1099

# Lancement d'un client (la commande peut être répétée pour plusieurs clients)
<CHEMIN_DU_JDK>/Contents/Home/bin/java -jar
<CHEMIN_DES_SOURCES>/out/artifacts/client_front_jar/client-front.jar
<DOSSIER_RESSOURCES>/client-front/  localhost 1100
```

6.4.2. Windows / Linux

```
# Lancement du serveur principal
<CHEMIN_DU_JDK>/bin/java -jar
<CHEMIN_DES_SOURCES>/out/artifacts/main_server_jar/main-server.jar localhost 1099
localhost 3308 <DOSSIER_RESSOURCES>/main-server/

# Lancement d'un serveur secondaire
<CHEMIN_DU_JDK>/bin/java -jar
<CHEMIN_DES_SOURCES>/out/artifacts/client_server_jar/client-server.jar localhost
1100 localhost 3307 <DOSSIER_RESSOURCES>/client-server/ localhost 1099

# Lancement d'un client (la commande peut être répétée pour plusieurs clients)
<CHEMIN_DU_JDK>/bin/java -jar
<CHEMIN_DES_SOURCES>/out/artifacts/client_front_jar/client-front.jar
<DOSSIER_RESSOURCES>/client-front/ localhost 1100
```

6.5. Ajout de serveurs clients

La procédure précédente, ne permet la création que d'un unique serveur client. Il est toutefois possible de créer N serveurs clients.
Pour se faire, il est nécessaire de suivre la procédure suivante.

6.5.1. Base de données

- Dupliquer le dossier des sources **`./docker/client-server`** sous un nom unique.
- Modifier le fichier **`docker-compose.yml`** de ce nouveau dossier.
  - Le **`container_name`** doit être remplacé par un nom unique.
  - Le premier **`port`** doit être remplacé par un port réseau non utilisé.
- Suivez la procédure précédente pour lancer la nouvelle base de données.

6.5.2. Serveur

Modifier la commande permettant le lancement d'un serveur secondaire en modifiant les arguments de la commande tels qu'indiqué dans la partie 6.3.2. Il est nécessaire d'utiliser un port non utilisé pour le serveur.

6.5.3. Dossier ressources

Chaque client-serveur doit avoir son propre dossier de facturation distinct. À chaque création de client, il faut donc créer un nouveau dossier qui servira pour les factures.

7. Conclusion

Le projet est à ce jour complètement fonctionnel et respecte l'intégralité des points du cahier des charges. Grâce à une conception rigoureuse et une mise en œuvre méthodique, nous avons réussi à développer un système qui gère efficacement les échanges en temps réel entre les différents composants (clients et serveurs).

Les fonctionnalités de gestion des données, de facturation, et de communication client-serveur ont été implémentées avec succès en utilisant les technologies appropriées, notamment Java pour le développement, RMI pour les appels de méthodes à distance, et MySQL pour la gestion des bases de données.

L'architecture du système permet une bonne évolutivité et une maintenance simplifiée grâce à une séparation claire des responsabilités entre les composants. Les bonnes pratiques de développement ont été suivies, garantissant un code propre, lisible, et facilement modifiable. La synchronisation des données entre les serveurs est assurée de manière efficace, garantissant la cohérence et l'actualisation des informations.

La partie frontend, développée en utilisant la bibliothèque Swing, offre une interface utilisateur intuitive et efficace, répondant aux besoins des utilisateurs finaux. Les fonctionnalités comme la gestion du stock, la facturation et le calcul du chiffre d'affaires sont présentées de manière claire et accessible, rendant l'utilisation du système agréable et productive.

En conclusion, ce projet a non seulement atteint ses objectifs initiaux mais a également posé les bases pour d'éventuelles améliorations et extensions futures.