IUT de Paris - Rives de Seine
Université Paris Cité

2023/2024

RAPPORT ARCHI LOGICIELLE

présenté par

Perigault Paul - Rebeller Owen

(209) - (208)

Fait le 09/06/2024

Ressource :

R4.01

Table des matières

Chapitre 1

Côté Serveur

1.1 Structuration du code

Pour notre architecture nous avons structuré le code en trois parties : le client, la configuration et le serveur.

- **Client** : Il gère la communication avec le serveur. Il a aussi son propre *main*.
- **Configuration** : Il contient la classe *Config.java* et le fichier *config.json* pour la configuration des paramètres de l'application. Ce sont les bonnes pratiques de sécurité qui nous ont poussés à faire ceci.
- **Serveur** : Il comprend différents sous-modules :
  - **Base de données** : Ce module gère la connexion à la base de données ainsi que les opérations sur cette dernière.
  - **Éléments** : Défini les entités de l'application telles qu'un *Document* et un *Abonne*.
  - **Exceptions** : Gère les différentes exceptions spécifiques à l'application.
  - **Opérations** : Regroupe les différents services de l'application (Réservation, Emprunt, Retour).
  - **Serveur** : Implémente le protocole de communication utilisé par l'application.
  - **Utilitaire** : Fournit des méthodes utiles à l'application et donc réutilisables.
  - **TimerTask** : Ce module permet de gérer les tâches planifiées.

  Le serveur a aussi son propre *main* afin d'être exécuté indépendamment du client.

1.2 Code Stable

Pour ce projet, nous avons mis en place plusieurs dispositifs pour rendre le code maintenable et stable.

Premièrement, nous avons utilisé des factories pour faciliter la création d'éléments en grande quantité. Nous avons implémenté une factory dédiée aux serveurs ce qui nous permet de générer autant de serveurs que nécessaire. Cela nous offre la flexibilité de remplacer ou mettre à jour un serveur sans affecter le reste du code.

Une factory pour les données a été mise en place pour charger les données depuis la base de données lors de l'initialisation de l'application et de les sauvegarder lors de la fermeture de l'application. Ces factories évitent la répétition de tâches redondantes et utilisent les classes de modèle ou de serveur comme objets, conformément aux notions étudiées en cours. Cela rend les factories plus évolutives et adaptables.

Nous avons également mis en place des interfaces pour assurer la stabilité des classes. De plus, pour certaines parties du code, nous avons factorisé les méthodes et les variables en les regroupant dans une seule classe, permettant ainsi aux autres classes d'hériter de celle-ci.

L'objectif est de faire en sorte que nos classes respectent au mieux les principes SOLID et les design patterns vus en cours de qualité logicielle, tels que le pattern Factory, le pattern Bridge et

autres patterns de conception. Cela permet d'assurer une architecture de code robuste, flexible et facile à maintenir.

1.3 Graphe de dépendance

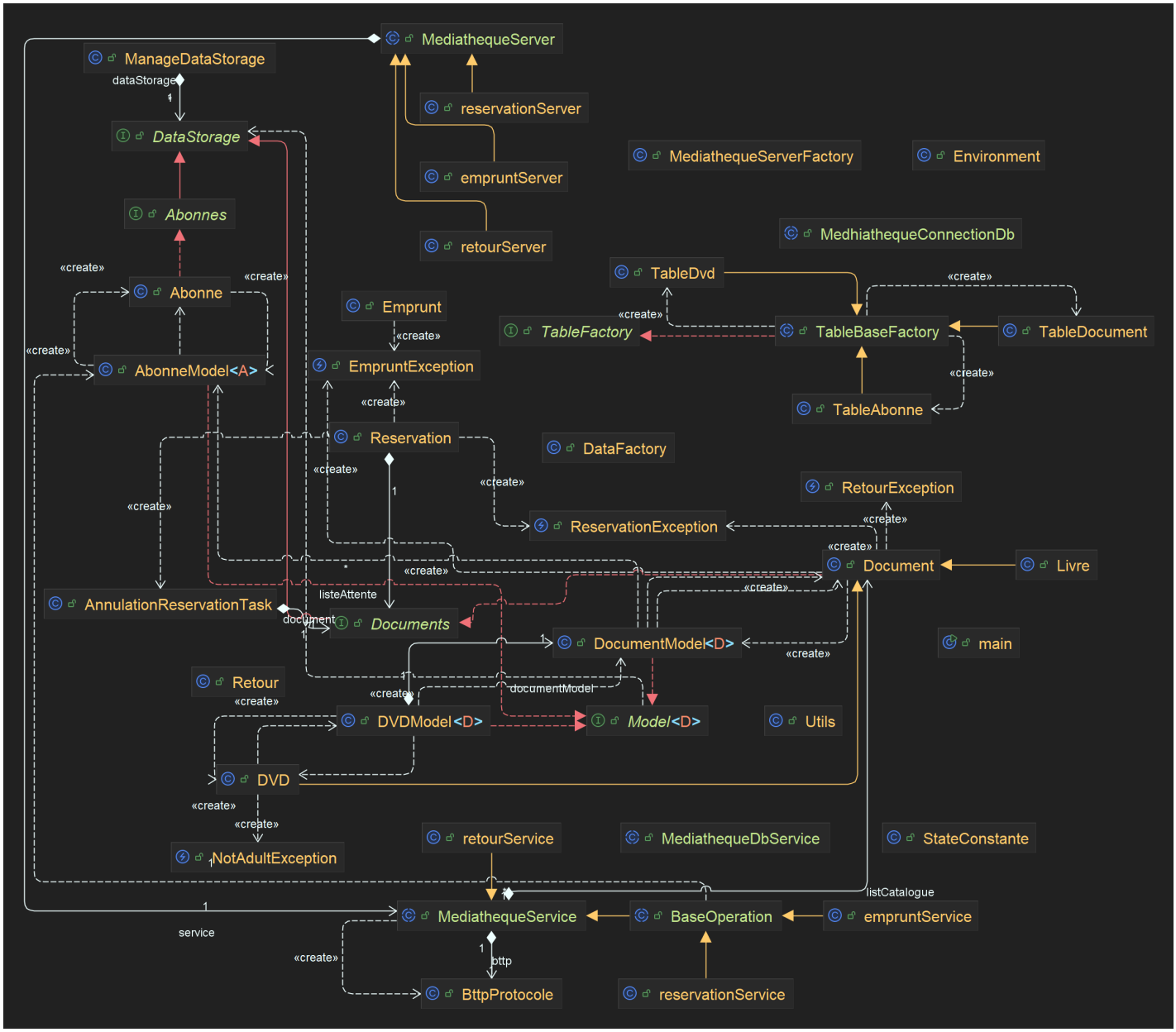

FIGURE 1.1 – *Graphe de dépendance*

1.4 Bibliothèques logicielles

Nous avons utilisé plusieurs bibliothèques logicielles essentielles pour développer le projet. Voici les bibliothèques utilisées et leur utilisation spécifique :

— **JDBC** : Cette bibliothèque a été utilisée pour interagir avec la base de données, permettant d'exécuter des requêtes SQL et de gérer les connexions conformément aux principes vus en cours.

— **Java .net** : Elle a été employée pour créer un protocole de communication via *ServerSocket*, facilitant les échanges entre le client et le serveur.

— **Java .io** : Nous avons utilisé cette bibliothèque pour gérer les opérations d'entrée et de sortie, telles que la lecture et l'écriture de données ainsi que les exceptions associées.

— **Org .json** : Cette bibliothèque a été utilisée pour lire et traiter le fichier *config.json*, ce qui nous permet de gérer les paramètres de configuration de manière sécurisée et flexible.

— **Java .util** : Nous avons largement utilisé cette bibliothèque pour manipuler différentes collections (*Map*, *List*), ainsi que pour les fonctionnalités de *Scanner*, *Date*, *Timer* et *TimerTask*. *TimerTask* a été implémenté afin de limiter la durée maximale d'une réservation pour emprunter un document, évitant ainsi des réservations indéfinies.

— **Java .sql** : Cette bibliothèque a été intégrée pour permettre la communication en SQL avec la base de données, facilitant ainsi les opérations de création, lecture, mise à jour et suppression (CRUD) en utilisant également les exceptions spécifiques de cette bibliothèque.

L'utilisation de ces bibliothèques a permis de structurer notre application de manière efficace, en utilisant des outils et fonctionnalités standards de Java pour assurer maintenabilité et performance.

Chapitre 2

Échanges Client / Serveur

2.1 Client

Pour la section client, nous avons établi plusieurs classes qui faciliteront la division du code en fichiers distincts. Nous disposons donc d'un fichier dédié à la création de socket et au protocole BTTP, tandis qu'une autre classe permettra à l'utilisateur de choisir le service en fonction du port correspondant.

2.2 Serveur

Pour la gestion des entrées et des sorties de données, nous utilisons plusieurs classes :

— **BufferedReader** : Utilisé pour lire les messages du client et du serveur.

— **PrintWriter** : Utilisé pour envoyer des messages dans le flux de sortie.

2.3 Échanges Client / Serveur

Le client se connecte au serveur en utilisant le protocole **BTTP**. Une fois connecté, le client et le serveur peuvent échanger des messages de manière synchrone. La connexion est initiée par le client en spécifiant l'adresse et le port du serveur. Les messages sont échangés en utilisant `BufferedReader` et `PrintWriter`. Le serveur traite les commandes reçues du client et renvoie les réponses appropriées.

Grâce à cette approche, une seule classe client est nécessaire, car la communication via BTTP permet une redirection efficace des ports vers le serveur souhaité pour chaque opération **(emprunt, réservation, retour)**. Le serveur dirige les requêtes vers les services appropriés, garantissant une gestion des différentes opérations possibles à l'heure actuelle et dans le futur.

Cette architecture permet une application performante et facile à maintenir, tout en permettant une extension future des fonctionnalités. Cette approche permet d'éviter la répétition de code en réinitialisant constamment les sockets, améliorant ainsi la performance et la maintenabilité de l'application. Nous avons fait une version basique du BTTP sans implémenter l'encodage/décodage, ce qui réduit le temps de communication mais diminue la sécurité.

Chapitre 3

Lien Application - Base de données

3.1 Modèle

Concernant la liaison entre l'application et le serveur, nous avons utilisé **phpMyAdmin** pour la gestion. Toute base de données MySQL peut quand même être utilisée étant donné que le SQL n'est pas standardisé pour tous les SGBD, ce qui s'assure que l'application soit fonctionnelle pour chaque utilisateur, à condition qu'ils utilisent le même SGBD que nous.

Nous avons mis en place des modèles pour représenter les tables de la base de données sous forme de classes. Cela nous permet de créer un **CRUD (Create, Read, Update, Delete)** afin de réaliser différentes opérations sur la base de données. En outre, nous avons également développé une factory pour générer nos modèles de tables, garantissant ainsi que chaque membre de l'équipe dispose des mêmes modèles. De plus, pour chaque type de modèle, nous avons créé une interface pour rendre les classes flexibles et stables.

Lors de l'initialisation du projet, en exécutant la classe responsable de la création des tables, nous pouvons nous assurer que tous les membres du groupe disposent de la même architecture de base de données.

3.2 Sauvegarde

Pour gérer les différents états des documents, nous avons mis en place plusieurs types de structures pour stocker les documents en fonction de leur état. Par exemple, pour les documents réservés, nous les avons stockés dans une `HashMap` au sein de la classe effectuant l'opération (ex : Réservation), afin d'éviter de les regrouper tous dans une seule structure. Cette approche permet de localiser les documents selon leur état. Il est à noter que tous les documents retournés et donc libres sont stockés dans la structure principale lors de l'initialisation des services.

Lorsque nous fermons l'application correctement, nous nous assurons de stocker les données dans la base de données. Cette approche **minimise les échanges** avec la base de données car il s'agit d'une application `serverSocket` et non d'une application Web. Par conséquent, nous préférons stocker les données dans des variables ou tout autre structure de donnée pour interroger la base de données qu'au lancement et à la fermeture de l'application.

Chapitre 4

Concurrence

Nous avons utilisé des blocs `synchronized` ainsi que des fonctions pour protéger les sections critiques de notre application, comme pour le paradigme du problème des philosophes afin d'assurer une exécution thread-safe. La concurrence se trouve entre les différentes opérations (réservation, retour, emprunt) car elles ne peuvent être exécutées simultanément sur un même document.

C'est en soi intuitif mais il est important de l'implémenter correctement pour éviter les conditions de course. Les sections synchronisées sont donc exécutées que lorsqu'elles sont disponibles, ceci garantit que l'application soit thread-safe.

Chapitre 5

Maintenance évolutive

5.1 Ajout de nouveaux Documents

L'ajout de nouveaux documents nécessite d'implémenter l'interface correspondant aux documents ainsi que son modèle respectif si l'on veut le stocker dans la base de données. La logique des opérations de réservation, emprunt, et retour ne changera pas. C'est du **Pattern Bridge** que nous nous sommes inspirés pour implémenter les documents.

5.2 Passage en application web

Pour passer sur une application Web, nous devons revoir notre serveur de la médiathèque. Les opérations ne changeront pas mais toute notre partie serveur devra être revue. Nous devons donc optimiser la maintenabilité de l'application afin de dissocier la médiathèque Server et son service du protocole de communication existant.

5.3 Ajout de nouveaux services

La numérotation des ports ainsi que les URL sont gérées par une classe `Config` qui lit un fichier `config.json`. Nous avons utilisé ceci car c'est une bonne pratique en termes de sécurité de ne pas écrire en clair dans le code les URL et ports.

Pour ajouter un nouveau service, il suffit d'attribuer les informations confidentielles à ce service dans la partie configuration, puis d'implémenter la classe du service. Il est important de ne pas oublier d'implémenter l'interface correspondante à la médiathèque, qui est propre à la relation entre le service et le serveur.

Chapitre 6

BretteSoft

Pour la partie BretteSoft, nous avons implémenté une seule fonctionnalité : **Sitting Bull**. Cette fonctionnalité permet de notifier un utilisateur par courriel lorsque ce dernier souhaite emprunter un document déjà réservé.

6.1 Implémentation

Pour implémenter cette fonctionnalité, nous avons intégré dans un fichier existant nommé `Utils` les fonctions nécessaires pour envoyer un courriel à une personne choisie, en l'occurrence le grand Wakan Tanka, à l'adresse suivante : **jean-francois.brette@u-paris.fr**. Cela signifie que lorsqu'une personne souhaite réserver un document, il lui sera demandé si elle souhaite être notifiée. Si elle répond positivement, un courriel lui sera envoyé dès que le document sera disponible.

— **Javax.mail** : Bibliothèque utilisée pour envoyer des mails.

Concernant la sécurité, les paramètres du courriel, notamment l'adresse d'envoi, le port ainsi que le mot de passe, ont été stockés dans un fichier d'environnement.

6.2 Processus

Pour faire fonctionner cette bibliothèque, nous devons d'abord mettre en place certains éléments, notamment le mot de passe pour l'accès aux applications. En effet, pour pouvoir se connecter et ensuite envoyer un courriel à une personne, il est nécessaire de générer un mot de passe pour l'authentification. Auparavant, il suffisait de cocher une case dans la section "Less secure App" de Google, mais cette option a été supprimée depuis le 30 septembre 2024.

Sources :

— **Envoyer mail SMTP** : Gmail SMTP server to send emails for SRM

— **Less secure App** : Google Accounts Answer

6.3 Alternatives d'implémentation

Il existe différentes manières d'implémenter la fonctionnalité BretteSoft. Par exemple, nous aurions pu :

— Créer notre propre serveur SMTP pour envoyer les messages.

— Utiliser d'autres bibliothèques ou services proposant l'envoie de mail.

Cependant, pour des raisons de sécurité et de simplicité, nous avons préféré utiliser un service déjà existant comme Javax.mail. Et d'utiliser le serveur SMTP de Gmail pour envoyer les mails car c'est un service déjà utilisé en entreprise.