Projet - ArchiLog

NOMS DES PARTICIPANTS :

ESTEVES Gabriel 203
LENOUVEL Louis 203

Table des matières

Côté serveur

Pour le projet serveur médiathèque nous avons décidé de couper notre projet java serveur en plusieurs packages afin d'optimiser au mieux la structuration de notre code. Nous avons un package bd, documentAbstract, documents, médiathèque, services. Cette organisation nous a permis de rendre compréhensible et structuré notre code tout en gardant notre code stable.

Tout d'abord, nous avons décidé de créer un package dédié à la base de données, représentant le lien entre l'application et la BDD, (package bd) afin de protéger la connexion bd. Pour cela nous lui avons attribué une visibilité package de sorte que seules les requêtes bd présentes dans le package puissent appeler la connexion. Chaque requête peut être appelée de manière statique dans les autres classes afin de pouvoir donner un accès constant à la base de données à nos services.

Ensuite, nous avons le package documentAbstract qui comprend les classes ConcurrentDocuement et Document. Dans ce package, nous nous occupons des différents points critiques de notre application. Chaque document qui est instancié sera instancié dans un document concurrent afin de pouvoir ajouter des blocs synchronized. Nous avons ajouté ces derniers sur toutes les méthodes susceptibles de provoquer une erreur de concurrence. Ainsi, nous avons pu assurer le thread safety de notre application. Ces classes ne sont pas des types concrets, c'est pourquoi nous les avons séparés du package document.

Ainsi, dans notre package document ne se trouve que les classes concrètes de document comme les DVD par exemple. Nous pourrions très bien ajouter d'autres documents dans ce package, car nous avons implémenté la médiathèque et ses différents services de sorte qu'ils ne manipulent que des IDocument, ainsi notre code est facile à maintenir et à développer.

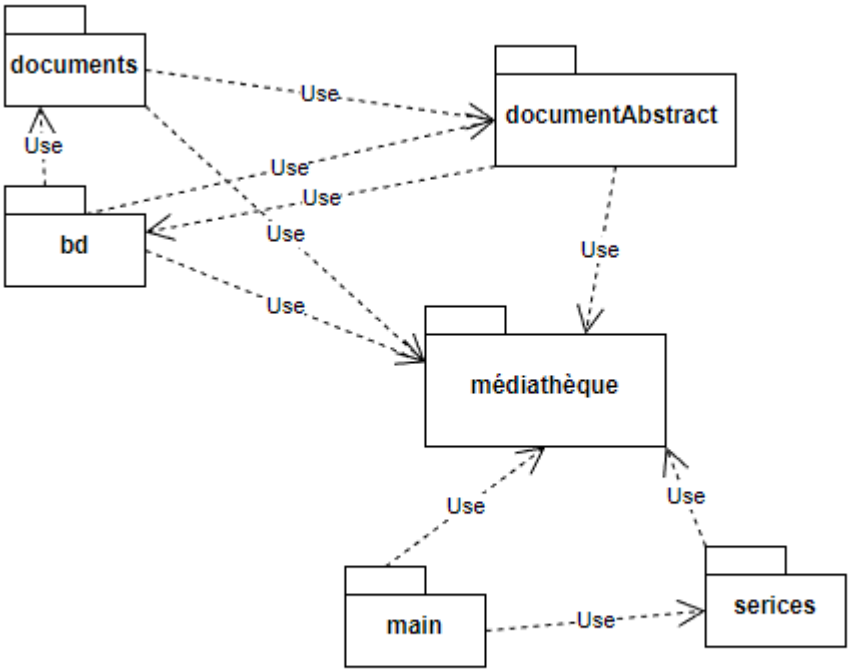
Vient ensuite le package médiathèque, dans ce dernier se trouve toutes les classes stables de notre application. Il y a notamment la classe Abonné qui ne manipule aucun document, mais qui est utilisée par les autres classes du package. Nous avons aussi l'interface IDocument qui représente les documents dans ce package et qui nous évite les sur-dépendances. Cela nous a permis aussi de réduire la complexité de notre code et ainsi un bon

découplage. Dans ce package, se trouve la classe médiathèque qui manipule les documents et les abonnés. Nous avons aussi rajouté la restrictionException et une autre exception exception pour les abonnés bannis, car elles sont propres aux classes se trouvant dans ce package.

Enfin, nous avons le package service où se trouvent tous les services disponibles dans la médiathèque, c'est-à-dire réservation, emprunt et retour. Nous les avons placés dans un package à part pour des questions de propreté du code et de bonne structuration.

Pour finir, nous avons le package main qui contient la classe ServeurMediatheque, main de notre serveur. Lui aussi se trouve dans un package à part pour des questions de stabilités de code.

Graphe de dépendances inter-packages :



Pour la mise en œuvre-utilisation de bibliothèques logicielles, nous avons d'abord décidé de créer notre propre bserveur, seriveAbstract, bttp afin de le rendre le plus fiable possible pour notre projet. Une fois ceci fait, nous avons externalisé ces classes en jar et de les réimporter afin de gagner du temps et de réduire la complexité du développement de notre application. En plus de ces bibliothèques que nous avons nous-même construit, nous avons fait appel à des bibliothèques déjà existantes. Nous avons notamment utilisé la bibliothèque jdbc pour la connexion à la base de données et les bibliothèques javax.mail ainsi que activation afin de réaliser la certification Sitting Bull.

Echanges client/serveur

Pour les échanges entre clients-serveurs, nous avons décidé d'utiliser le protocole BTTP 2.0. Ainsi, nous avons une bibliothèque logicielle BTTP2.0 qui contient 2 méthodes encodées et décoder.
Ces méthodes permettent de mettre en place un système de cryptage pour que le bufferedReader.readLine() puisse envoyer des messages contenant des “\n”. Nous implémentons ce système via une méthode de la classe String (replace). Cela nous a permis de faire un affichage plus esthétique des messages lors de l’échange client/serveur.

Ces méthodes sont utilisées dans la classe Client contenant une unique méthode main, au lancement de celui-ci il l’utilisateur doit saisir l’adresse ip et le port qu’il veut utiliser pour choisir son service.
La classe Client repose ensuite sur le système ping pong implémenté grâce à une boucle while tout en utilisant les méthodes de bttp. C'est-à-dire qu’il reçoit un string, il l'affiche. Il saisit un string, il l'envoie.
Ainsi, nous avons rendu totalement indépendant notre client des services. Les échanges commencent par un envoi d’un message du serveur vers le client. Si ce dernier repère un point d'interrogation, il sait alors que c’est une question du serveur et qu’il faut répondre. Si, il n’y a pas de point d’interrogation dans la chaîne de caractères envoyé par le service, ceci implique que c’est une réponse finale. Une fois la réponse envoyée au serveur, elle est traitée par le service. Si la réponse ne correspond pas à ce qui est attendu, alors le serveur met fin à la communication.

Du côté des services, le schéma de questions-réponses est spécifique à chaque service. Les services utilisent également la bibliothèque logicielle bttp2.0 pour encoder à l’envoi et décoder à la réception. En fonction des saisies d’utilisateur, une réponse est systématiquement envoyée qu'elle soit finale ou non. Si la réponse nécessite une réponse du client, elle contient un point d'interrogation, sinon elle n’en contient pas.

PS : Nous avons mis en place ce système avec les points d'interrogation, car quand nous fermions la socket du côté du service. Le client devait saisir une réponse alors que la socket du service était déjà fermé. De cette façon nos échanges clients-serveurs ont pu fonctionnées de manière optimale.

Lien entre l'application et la BDD

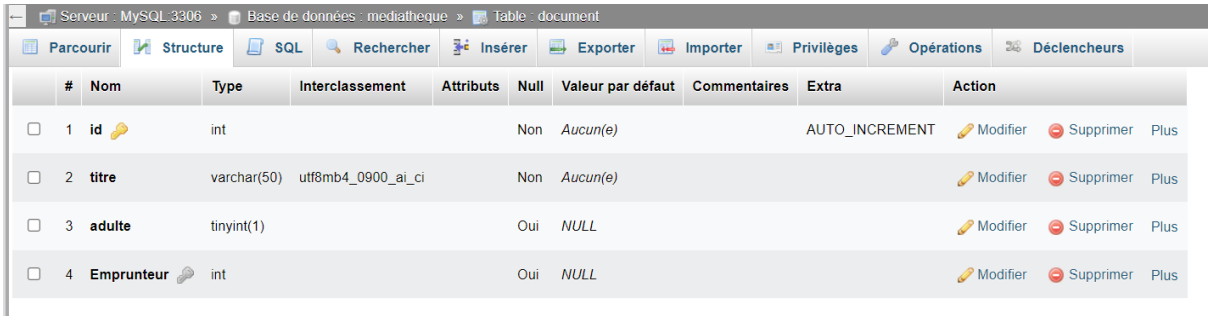
Afin d'optimiser au mieux notre serveur, nous avons créé deux tables dans une base de données dédiée. Une table concernant les abonnés, pour chaque abonnée nous avons son id d'identification, son nom et sa date de naissance.

Table abonne :

#	Nom	Type	Interclassement	Attributs	Null	Valeur par défaut	Commentaires	Extra	Action
1	id	int			Non	*Aucun(e)*		AUTO_INCREMENT	Modifier Supprimer Plus
2	nom	varchar(50)	utf8mb4_0900_ai_ci		Non	*Aucun(e)*			Modifier Supprimer Plus
3	dateNaissance	date			Non	*Aucun(e)*			Modifier Supprimer Plus

Nous avons aussi une table document représentant les documents de la médiathèque, chaque DVD est représenté par son id d'identification, son titre, un boolean adulte qui est à true lorsque le document est à destination des adultes (>16) et inversement, son idEmprunteur qui correspond à l'id de l'abonné qui a emprunté le document. Cette dernière information nous permet de déterminer si le document est déjà réservé lors du démarrage du serveur. Ainsi toutes ces informations sont stockées uniquement dans la base de données, bien entendu d'autres informations sont stockées directement dans le serveur car ce ne sont pas des données persistantes.

Table document :



#	Nom	Type	Interclassement	Attributs	Null	Valeur par défaut	Commentaires	Extra	Action
1	id	int			Non	*Aucun(e)*		AUTO_INCREMENT	Modifier Supprimer Plus
2	titre	varchar(50)	utf8mb4_0900_ai_ci		Non	*Aucun(e)*			Modifier Supprimer Plus
3	adulte	tinyint(1)			Oui	*NULL*			Modifier Supprimer Plus
4	Emprunteur	int			Oui	*NULL*			Modifier Supprimer Plus

En plus d'optimiser la gestion des données, nous avons décidé de mettre en place une connexion constante avec notre base de données. Pour cela, nous avons créé un package bd à part dans le projet avec une classe ConnexionBD et une classe RequetesBD. La classe Connexion BD est composée d'un bloc static qui crée la connexion à notre base de données et qui s'exécute à son premier appel. Elle

dispose aussi d'une méthode getConnexion qui permet de retourner la connexion à la base de données et d’une fonction close qui ferme la connexion. Seule la classe RequetesBD peut utiliser les méthodes de la classe ConnexionBD car elles sont protected, ainsi cela nous a permis de sécuriser la connexion et de ne pas la mettre à la vue de tous. La classe RequetesBD représente toutes fonctions que nous pourrions utiliser afin de manipuler la base de données. Cela comprend les update et les select. Grâce à cette classe nous pouvons modifier ou récupérer des données en temps réel sur notre serveur.

En somme, notre serveur fonctionne de sorte que toutes les données sont récupérées dès son lancement. Ensuite, nous exécutons chaque action sur la base de données en temps réel pour mettre à jour les données persistantes, c'est-à-dire l’emprunteur.

Concurrence et sections critiques

Tout d'abord pour pouvoir gérer la concurrence nous avons mis en place une classe ConcurrentDocument pour qu'elle puisse gérer la concurrence et les sections critiques de nos IDocuments. Cette classe implémente la classe IDocument et peut être instancié avec un IDocument en paramètre.

Ensuite, dans notre requête qui récupère tous les documents de la base de données. Nous avons créé une liste de IDocument avec des ConcurrentDocuments instanciés avec les DVD de la base de données.

Méthodes initialement dans l'interface du document :

Pour la **méthode getter numéro** comme cette méthode :

```
public interface Document {
    int numero();
```

Le getter du numéro n'est qu'un simple accès en lecture qui est sur un attribut du document qui n'est pas sujet à changement. Nous avons donc décidé de ne pas mettre en place des blocs synchronized car cette méthode ne met pas en péril la Thread-Safety du document.

Ensuite pour les **méthodes de réservation, emprunt et retour** :

```
    @pre ni réservé ni emprunté
    void reservationPour(Abonne ab) ;

    @pre libre ou réservé par l'abonné qui vient emprunter
    void empruntPar(Abonne ab);

    @brief retour d'un document ou annulation d'une réservation
    void retour();
}
```

Ces différentes méthodes modifient les attributs des documents. Ils modifient notamment les attributs réserveur, emprunteur. Il faut donc les protéger des accès concurrents en ajoutant un bloc synchronized. Ainsi, on modifie le parallélisme des threads pendant ces sections critiques pour garantir la thread-safety. Car le Thread courant va prendre le verrou sur la ressource partagé qui est le IDocument que possède la classe ConcurrentDocument.

Méthodes ajoutées pour les certifications :

1- Certification BretteSoft© Sitting Bull

Pour cette **méthode setSendMailTrue();**

```
//attribut à true pour envoyer mail à un utilisateur quand le document sera disponbile
void setSendMailTrue();
```

Elle modifie l'attribut du document et elle le met à true. Il y a donc un changement de l'état du document. Nous avons donc décidé d'ajouter un bloc synchronized pour les mêmes raisons que les méthodes précédentes, afin de garantir la Thread Safety du document.

2- Certification BretteSoft© Géronimo
Pour la **méthode mauvaisEtat();**

```
//attribut à false le boolean bonetat du document
void mauvaisEtat();
```

Elle modifie l'état d'un document et met à false. Il y a donc un changement d'état du document. Nous avons donc rajouté un bloc synchronized afin de garantir la Thread Safety de document.
Pour la **méthode dateRetouremprunt();**

```
//retourne la date de retour attendu pour le document
Date dateRetouremprunt();
```

Cette méthode retourne la date de retour prévu pour le document, or cette date est sujet à changement au fur et à mesure des exécutions. C'est pourquoi nous avons donc rajouté un bloc synchronized afin de garantir la Thread Safety de document.
Pour la méthode renduEnretard

```
//return true si le document est rendu avec un retard de plus de 2 semaines
boolean renduEnretard();
```

Cette méthode utilise la date de rendu initiale du document et lui ajoute deux semaines. Or cet attribut peut changer au cours de l'exécution, nous avons donc décidé de synchronized cette méthode.
Pour la méthode getBonEtat();

```
//return true si le document est en bon etat
boolean getBonEtat();
```

Cette méthode retourne l'état du document, attribut qui est amené à changer de valeur lors de l'exécution. Nous avons donc rajouté un bloc synchronized afin de garantir la Thread Safety de document.

Maintenance évolutive

Ajouter des livres, cds ?
Afin de pouvoir ajouter des livres et des cd, il faudrait leur créer une classe à chacun et les extends de notre classe Document, qui elle-même implémente l'interface IDocument. Nous avons déjà implémenté toutes les méthodes de la classe IDocument. Chaque méthode est implémentée dans Document.
Pour la classe DVD nous avons uniquement spécialisé certaines méthodes de la classe Document qui est extend. Et nous avons appelé le super des méthodes spécialisées pour uniquement ajouter cette contrainte d'âge. Il est donc très simple d'ajouter d'autres types de documents.
De plus, nous avons créé notre serveur médiathèque et ses services de sorte qu'ils puissent ne manipuler uniquement que des IDocument. Ainsi, le serveur fonctionne de manière générique et stable et ne dépend pas de la classe DVD. Les seuls efforts nécessaires pour cette implémentation seraient la création des classes et la création des tables pour chaque classe.
Pour la base de données, il aurait fallu pouvoir mettre de l'héritage en créant une table document et en ajoutant certaines tables avec leurs spécificités. Mais avec MySQL nous n'avons pas pu procéder de cette façon.

Faire passer l'application en application web ?
Pour pouvoir faire passer notre application en application web au lieu de ServerSocket/Socket, il n' y a pas besoin de faire de changement car les données persistantes sont sauvegardées en temps réel dans la base de données de la médiathèque, on met à jour systématiquement l'emprunteur des documents.

Ajouter un nouveau service au port 6000 ?
Pour ajouter un service au port 6000, il suffit de créer une classe qui extends ServiceAbstract de notre bibliothèque logicielle bserveur. Et implémenter la méthode run qui est abstract dans la class ServiceAbstract. Et il faut ensuite créer un thread dans le main du Serveur avec le nouveau service qui a été créé et avec le port 6000. Il est donc très simple d'ajouter un nouveau service grâce à notre bibliothèque logicielle.

BretteSoft

Nous avons essayé d’intégrer la tribu comme guerrier des plaines.

1 Certification BretteSoft© Sitting Bull
Nous avons mis en place la certification Sitting Bull du mieux possible.
Lors du service de réservation, si le document est réservé ou emprunté par un autre abonné que celui qui réserve. Alors, on propose à l’utilisateur s'il souhaite recevoir un mail quand le document sera de nouveau disponible. S’il a répondu oui, alors quand le document sera disponible (réservation annulée ou bien retour du document). Il recevra un mail à la boîte du grand Wakan Tanka (jean-francois.brette@u-paris.fr)

2 Certification BretteSoft© Géronimo
Nous avons réalisé la certification geronimo de la manière la plus optimale. Lorsque l’abonné retourne un document qu’il a emprunté, nous lui demandons si le document est en bon état. (nous comptons sur l’honnêteté de l'abonné.). Si le document lors du retour est en mauvais état alors nous bannissons l’abonné de la tribu pendant 1 mois et le sauvegardons dans un attribut de la classe document. Ainsi, lorsque ce document sera encore emprunté, nous ne demanderons plus son état. Lorsque l’abonné emprunte un document, nous lui laissons une semaine pour le rendre et nous lui indiquons. Nous sauvegardons cette date dans la classe document. Si, lors du rendu, cette date est dépassée de deux semaines, alors l’abonné est banni 1 mois de la tribu en créant une date de bannissement. Aucune addition de bannissement n’est possible. Lors d’une réservation ou un emprunt, nous vérifions donc si l'abonné est banni. Si c’est le cas, nous le bloquons. Sinon nous le laissons communiquer et s'il a été déjà banni nous lui supprimons sa date de bannissement.

Conclusion

Ce projet a été une opportunité stimulante pour notre équipe, car nous avons pu mettre en pratique toutes les compétences que nous avons acquises tout au long de ces deux années à l'IUT, en créant une application complète tout en gérant le multi-threading et la gestion de la concurrence entre ces threads. Ce projet a été une réussite, car il nous a permis de relever un défi ambitieux, tout en nous enseignant de nouvelles notions importantes. Nous sommes fiers du travail accompli et sommes convaincus que ce projet sera un excellent ajout à notre portfolio, démontrant notre capacité à réaliser des projets complets et professionnels.