fr-README.md

# Library Project

Un projet Java avec des Sockets, du Multy-threading, de la conversion vers du XML et plein d'autres trucs.

Le projet est dans l'absolut bien commenté, donc le code ne sera pas expliqué en détail ici, si vous voulez vous renseigner sur une partie du code vous pouvez simplement aller voir le code en question.

*Un peu de code sera expliqué, quand nécessaire.*

## Structure du projet

```
src
├──client
└──server
    ├──document
    │   └──book
    │       ├──assets
    │       └──library
    └──user
        ├──assets
        └──database
```

- Client - Contient `Client.java` , c'est tout...
- Server - Dossier du serveur.*duh*. Contient tout ce dont le serveur a besoin.
  - document - Contient le squelette du type `document` et les sous-type de document, comme la Class `Book` , cela facilite une possible extention.
  - user - Contient la Class `User` et tout ce dont elle dépand.

## Document

### Fichiers

```
document
│   iDocument.java
│   Item.java
└──book
    │   Book.java
    │   FilesToBookIntoXML.java
    │   Library.java
    │
    ├──assets
    │   │   .txt files
    │   ¦   ...
    │
    └──library
        │   .xml files
        ¦   ...
```

Pour former un nouveau type de `document` , vous devez étandre la Class `Item` (qui elle même implémente l'interface `iDocument` ). Placez la ensuite dans un sous-dossier comme pour la Class `Book` .

#### Les sous-dossiers `assets` et `library`

Le dossier `assets` contient juste les fichiers texte utilisés pour former les documents (ici les livres). Tous les fichiers sont formé de champs de Class, par exemple une suite de tous les IDs ou les auteurs, et la Class `FilesToBookIntoXML` les convertie en fichiers XML et les mets dans le dossier `library` .

# User

## Fichiers

```
user
│   FilesToUserIntoXML.java
│   User.java
│   UserDB.java
│
├──assets
│   │   .txt files
│   ┆   ...
│
└──database
    │   .xml files
    ┆   ...
```

La Class `user` ressemble assez à un `document` , elle n'implémente ni étand aucune Class mais elle a elle aussi des dossiers `assets` et `library` ( `database` dans ce cas)

# Comment ça marche

Le coté client est présenté en premier car il est plus court et simplifie un petit peu la comprehantion du coté serveur.

## Coté client

Lors de l'execution, vos identifients son requis :

```
Please, enter your credentials :
"your_ID" "your_password"
```

Si l'autentification réussie, un résumé de l'utilisateur est affiché (c'est plus utile pour le programmeur mais c'est toujours bien de l'avoir).

```
Authentication successful.
User n°0259558340 {
  Pass  : **********
  Name  : Jean-François Brette
  eMail : jean-francois.brette@parisdescartes.fr
  Age   : 47 (j'imagine... ¯\_(ツ)_/¯)
}
```

Sinon, affiche un message d'erreur et ferme le programme.

## Coté serveur

Le foncionnement du serveur est divisé en cinq threads, chacun crée ensuite un sous-thread pour faire le sale boulot.

Le fichier `Server.java` est structuré de cette manière :

```
main
│   crée la library et la base de donnée
│
├──booking thread
│   │   attend une connexion
│   └──sous-thread
│       │   reserve le livre, puis meurt
│
├──borrowing thread
│   │   attend une connexion
│   └──sous-thread
│       │   emprunte le livre, puis meurt
│
├──returning thread
│   │   attend une connexion
```

```
│   └──sous-thread
│       │   rend le livre, puis meurt
│
├──authentication thread
│   │   attend une connexion
│   └──sous-thread
│       │   verifie si l'utilisateur existe, puis meurt
│
└──catalogue thread
    │   attend une connexion
    └──sous-thread
        │   envoie le catalogue, puis meurt
```

#### Threads contractuels

Tous les threads sont plutôt similaires mais les trois concernants les livres se ressembles particulièrement. Ils attendent un connexion, créent leur sous-thread qui à leur tour reçoivent les IDs de l'utilisateur et du livre en question. Avec ça, ils cherchent le livre dans la library et execute la methode avec le bon utilisateur (la Class `UserDB` a une methode pour le trouver).

```java
// trouve le livre
Book b = Objects.requireNonNull(library.getCatalog().stream()
    .filter(book -> book.getID().equals(bookID))
    .findAny().orElse(null));
// execute la methode
b.method(userDB.findUserFromID(userID));
```

Le `.orElse(null)` est contrecaré par le `Object.requireNonNull` au début.

#### Thread pour l'autentification

Ce thread vérifie si l'utilisateur existe, si oui : le renvoie au client, sinon : renvoie un `null` (que le client interprète bien comme une erreur).

#### Thread pour le catalogue

Ce thread attend juste une connexion. Si connexion il y a, un sous-thread se charge d'envoyer le catalogue et l'heure de mise à jour à l'utilisateur connecté.

La methode `.toString()` de la library appelle la même methode mais pour chaque livre. Leurs methode s'adapte à l'état du livre, precisant donc s'il est disponible, reservé ou indisponible.

```
Book n°6267855912 {
  title  : Children Without Desire
  author : January Wick
} Available
```

# Comment nous avons géré les choses

## Mutli-threading et accès simultanés aux données

Les problèmes que les Threads créent sont innombrables. Comment pouvons nous réduire les risques d'écritures simultanées, ou même comment les faires disparaitres ?

### Ce que nous avons fait

Pour gérer les accès sur une même donnée au même moment, il existe un tas de solutions mais nous avons décidés d'utiliser des `méthodes cazi-pures` . Elles ne sont pas `100% pures` car elles ne répondent pas au second critère d'une fonction pure, à savoir n'avoir aucun effet de bord lors de son execution. Dans notre cas nous voulons modifier des données exterieurs donc nous brisons délibérément cette règle.

### Et comment nous l'avons fait

La logique derrière notre code est la suivante.

Voici ce qui se passe :

```
// at time t
Thread threadA
└──veut modifier Object objX
    |   appelle objX.method

// at the same time t
Thread threadB
└──veut modifier Object objX
    |   appell objX.method
```

et ce qui se passe avec des méthodes pure (directement codées dans l'Object) ressemble à ça :

1. 2 Threads demande l'execution d'une même méthode
2. La JVM "choisie" qui execute en premier
3. Le premier Thread execute la méthode, puis l'Object modifie ses données
4. Le second Thread execute à son tour la méthode mais avec les nouvelles données

Dans notre cas cela donne quelquechose comme ça :

1. Deux personnes veulent prendre le même livre au même moment
2. Ils tirent à la courte paille qui passera en premier au guichet
3. Le "gagnant" peut prendre le livre sans problème
4. Le "perdant" essaye de le l'emprunter mais on luirépond que le livre n'est plus disponible

## Gestion des tâches basées sur le temps

L'action d'emprunter sous entend qu'il faut surveiller les utilisateurs qui empruntent. Que se passe-t'il si quelqu'un reserve un livre mais ne vient pas le chercher ? C'est à nous de trouver une solution.

### Ce que nous avons fait

Pour cette tâche, nous avons utilisé la Class Java `Timer` couplée avec `TimerTask` . On crée un `Timer` au lancement du serveur puis nous ajoutons des `TimerTask` quand nécessaire.

### Et comment nous l'avons fait

Comme expliqué, nous créons d'abord un `Timer` au lancement du serveur.

```
Timer timer = new Timer();
```

Puis les `TimerTask` sont ajoutées de cette manière :

```
/* après que la réservation soit faite */

timer.schedule(() -> {
    if (b.isAvailable()) { // on verifie si le livre est toujours dispo (non emprunté)
        b.reset(); // si oui, alors on le réinitialise
        System.out.println("book n°" + bookID + " has been reset.");
    }
     cancel(); // on annule la tache, cela supprime la TimerTask (libération de mémoire, CPU et tt...)

}, 120000); // délai de 2 minutes

/**
 * Notez que Java 8 ne supporte pas les expressions lambda pour les TimerTasks !
 * IntelliJ le fait mais uniquement de façon esthétique, vous pouvez déployer
 * l'expression pour afficher le code complet de l'instanciation d'une TimerTask.
 */
```

# Certifications BretteSoft©

S'il vous plait, notez que ces certifications n'ont pas été implémentés, pour le moment. À la place nous allons expliquer comment peuvent elles être implémentées dans notre projet, avec un peu d'exemple de code.

## "Géronimo"

### Description

Certains abonnés rendent les livres en retard (parfois avec un gros retard) ; d'autres dégradent leslivres qu'ils empruntent ; un abonné, suite à un retard de plus de 2 semaines ou à dégradation delivre constatée au retour, sera interdit d'emprunt pendant 1 mois.

### Solution

En premier, nous pouvons ajouter un champ dans la Class `user` pour savoir si un utilisateur est interdit. On doit ensuite verifier si l'utilisateur est en effet en retard.

Pour ça, les Classes Java `LocalDate` et `Period` sont d'une grande utilité :

```
// mise en place de la deadline à deux semaines plus tard
LocalDate deadline = LocalDate.now().plusWeeks(2);

// verification
if (LocalDate.new().isAfter(deadline))
    /* interdit l'utilisateur */
else
    /* lui souhaite une bonne journée */
```

Pareil pour un utilisateur interdit :

```
// mise en place de la fin d'interdiction à un mois plus tard
LocalDate endOfBan = LocalDate.now().plusMonths(1);

// verification
if (LocalDate.new().isAfter(endOfBan))
    /* enlève l'interdiction */
else
    /* lui dis qu'il est encore interdit */
```

## "Cochise"

### Description

On ajoute des DVDs aux documents de cette bibliothèque (qui devient une médiathèque). Certains DVDs sont réservés aux plus de 12 ou 16 ans.

### Solution

Presque implémentée sans le savoir, il y a un champ `age` dans la Class `user` .

Il suffit juste d'override les methodes `booking` et `borrowing` dans la Class DVD, qui bien évidemment extends la Class `Item` , comme ça :

```
@Override
public void method(User user) throws BookingException {
    if (user.getAge() >= this.age)
        /* comme avant */
    else
        /* dis à l'utilisateur de vieillir un peu */
}
```

Bien sûr il ne faut apas oublier d'ajouter une liste de DVD au lancement du serveur.

## "Sitting bull"

### Description

Lors d'une réservation, si le livre n'est pas disponible, on pourra proposer de placer une alerte mail nous avertissant du retour du livre. La certification suppose l'exploration des bibliothèques de mail java et l'envoi d'un mail-test dans le contexte approprié à l'abonné Brette.

### Solution

Un nouveau champ peut être ajouté dans la Class `User` , par example `nextUser` . Il suffit ensuite de rajouter ce code dans la methode `returning` :

```
@Override
public void returning(User user) throws ReturnException {
    /* comme avant */
    notify(this.nextUser);
}
```

Une autre solution est de remplacer le champ `user` par une `List` de `user` . Faire ça permet en plus de faire une file d'attente pour le document. Le code de la methode `returning` devient :

```
@Override
public void returning(User user) throws ReturnException {
    /* comme avant */
    this.user.remove(0);
    notify(this.user(0));
}
```

Il faut modifier la methode `booking` :

```
@Override
public void booking(User user) {
    assert user != null : "User can't be null."; // panic if user not in database
    this.user.add(user);
}
```

... et de changer dans la methode `borrowing` :

```
if (this.reserved)
```

avec

```
if (this.user.size() == 0)
```

Vous pouvez voir ici que le champ `reserved` n'est plus d'aucune utilité puisque si le document est déjà réservé vous êtes juste mis en file d'attente.À la place, on peut verifier la taille de la liste.

# Auteurs

- **Marius Vallas** - *Gestion du Git, tous les trucs avec les XML, la programmation des `Sockets` et tout le refactoring et la documentation du code.*
- **Gabriel Arbane** - *Multi threading et la partie `Client` et `Server` .*
- **Antoine Dedieu** - *Multi threading, la partie `Server` et la gestion des `user` .*

## License

Ce projet est sous la licence du MIT - voir le fichier License pour plus de détails.

## Remerciement

Bonjour M. Brette, vous allez bien mdr ?