

# Library Project

---

A Java project using Sockets, Multi-threading, Object-XML conversion and a lot more.

The project is overall well documented, so I'll not explain the code here, if you want to know how a specific part works, just go to the file. Each file has a little tree of method at the beginning.

*I'll show a lil' bit of code when appropriate tho.*

## Project's structure

---

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

- **Client** - Contain `Client.java` , that's it...
- **Server** - Server folder. *duh*. Contain everything the server need to run correctly.
  - **document** - Store the outline of the document and every subtype of document, like `Book` , and can easily be extended.
  - **user** - Store the `User` Class and everything it need.

## Document

---

### Files

```

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

```

To form a new type of `Document` you have to extends `Item` (an abstract Class), who itself implements the `iDocument` interface. Then place it into a sub-folder like the `Book` Class.

### assets and library sub-folders

The `assets` folder contain a bunch of `.txt` files used to form the document (in this case, all of the books). Every file contain one type of Class field, like `ID` or `author` , and then the `FilesToBookIntoXML` Class converts everything into `.xml` and store them in the `library` folder.

## User

---

### Files

```

user
|   FilesToUserIntoXML.java
|   User.java
|   UserDB.java
|
└── assets
    |   .txt files
    |   ...
    |
    └── database
        |   .xml files
        |   ...

```

The `user` Class is very similar to a `document` , it doesn't implements nor extends any Class but still have an `assets` and `library` -like folder ( `database` ).

## How it works

---

We will cover the `client` side first because it's smaller and it simplify the `server` side explanation.

## Client side

When executing the `User` Class, you first have to enter your user's credentials like so :

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

If the authentication is successful, display user's data : (it's more for the programmer's convenience but it's good to have)

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

If not, then print error message and exit.

## Server side

The server work is divided between five threads, each thread then calls a sub-thread to do the work and die.

The `Server.java` file is structured like so :

```

main
|   creating the library and user database
|
|---booking thread
|   |   check for connection
|   |---sub-thread
|   |   |   reserve the book, then die
|
|---borrowing thread
|   |   check for connection
|   |---sub-thread
|   |   |   borrow the book, then die
|
|---returning thread
|   |   check for connection
|   |---sub-thread
|   |   |   return the book, then die
|
|---authentication thread
|   |   check for connection
|   |---sub-thread
|   |   |   check if user exists, then die
|
|---catalogue thread
|   |   check for connection
|   |---sub-thread
|   |   |   send the catalogue, then die

```

## Contractual threads

All the threads are very similar, but the three "book's" ones are almost identical. They check for a connection, create the sub-thread who then ask for the `user` 's and `book` 's ID. With that, check for the `book` in the `library` and then call the concerned method with the correct `user` as parameter (the `UserDB` Class has a method to find it).

```

// find the book
Book b = Objects.requireNonNull(library.getCatalog().stream()
    .filter(book -> book.getID().equals(bookID))
    .findAny().orElse(null));
// execute the method
b.method(userDB.findUserFromID(userID));

```

The `.orElse(null)` is thwart by the `Object.requireNonNull` at the beginning.

## Authentication thread

The authentication thread check just if the user exist, if yes : send it to the `client` , else : send a `null` that

the `Client` Class can understand as an error.

### Catalogue thread

This thread check just for a connection, then the sub-thread can send the catalogue to the `client` . It just call the `toString()` method of the `library` and add a timestamp.

The `.toString()` method of the `library` then call the same method for each `book` , their `.toString()` automatically tells if the `book` is either available, reserved or unavailable like this :

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

## BretteSoft© certifications

---

Please note that we didn't implement those certification, yet. We will only talk about how they can be implemented in our project, with bits of code.

### "Géronimo"

---

#### Description

Some subscribers return books late (sometimes very late), others degrade the books they borrow. A subscriber exceeding a delay of more than 2 weeks and or who deteriorate the book will be prohibited from borrowing for 1 month.

#### Solution

First, we can add a field in the `User` Class to know if the user is prohibited. After that, we still have to check if the user is late.

For that we can use the `LocalDate` and `Period` Classes like so :

```
// setting the deadline to two weeks ahead  
LocalDate deadline = LocalDate.now().plusWeeks(2);  
  
// checking if the deadline is passed  
if (LocalDate.now().isAfter(deadline))  
    /* ban the user */  
else  
    /* tels him to have a good day */
```

Same goes for a ban user :

```
// setting the end of the banning to one month ahead
LocalDate endOfBan = LocalDate.now().plusMonths(1);

// checking if the ban date is passed
if (LocalDate.now().isAfter(endOfBan))
    /* unban the user */
else
    /* tells him he's still ban */
```

## "Cochise"

---

### Description

We add DVDs to the documents of this library (which becomes a media library). Some DVDs are reserved for people over 12 or 16 years old.

### Solution

Almost implemented without knowing, there is a `age` field in the `User` Class.

You just have to override the `booking` and `borrowing` methods in the `DVD` Class, who obviously extends the `Item` Class, like that :

```
@Override
public void method(User user) throws BookingException {
    if (user.getAge() >= this.age)
        /* same as before */
    else
        /* tells the user to come back older */
}
```

Of course we need to add the DVD's library at the start of the `server` , exactly like the books.

## "Sitting bull"

---

### Description

When making a reservation, if the book is not available offer to place an email alert notifying the user when the book has been returned.

### Solution

We can add another field in the `Item` Class, like `nextUser` for example. And then we add this code in the `returning` method :

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

Another solution is to modify the field `user` to a List of `user`. Doing that enable the possibility to queue users. Here's the code of the `returning` method :

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

And we need to modify the `booking` method like so :

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

... and in the `borrowing` method, modify the

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

to

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

You can see here that we no longer need the `reserved` field in the `Item` Class because now multiple peoples can reserve an item. Instead if we need, we can check if the List's length.

## Authors

---

- **Marius Vallas** - *Git management, all the XML conversion crap, Sockets programming and overall refactoring / commenting code*
- **Gabriel Arbane** - *Multi threading and both Client and Server programming*
- **Antoine Dedieu** - *Multi threading and Server programming and user crap*

## License

---

This project is licensed under the MIT License - see the [License](#) file for details

## Acknowledgments

---

Bonjour M. Brette, vous allez bien mdr ?