|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | MÉDIATHÈQUE  EN LIGNE | ésultat de recherche d'images pour "bibliothèque iut paris descartes" | | ésultat de recherche d'images pour "tomcat" | ésultat de recherche d'images pour "java jee" | |
| Projet MediaWeb  Développement d’une application web permettant l’ajout, l’emprunt et le retour de documents au sein d’une médiathèque.  ésultat de recherche d'images pour "université paris descartes"  Produit par Raphaël NEVEU & Yannis LE GUEN. |

# Base de données

## Nous avons utilisé MariaDB comme SGBD.

## Notre application ne comporte que 3 tables :

## USER

## DOCUMENT

## EMPRUNT

## 

## Nous avons utilisé un héritage ascendant dans la table DOCUMENT. Ainsi, tous les documents quels que soient leurs types (Livres, CD, DVD...) sont stockés dans cette même table. Les attributs qui ne leur correspondent pas sont simplement mis à NULL (Ex : NbPages pour un CD sera NULL).

## La table USER contient son ID unique, son nom, son mot de passe ainsi que son type symbolisé par un entier.

## La table EMPRUNT ne contient que l’ID d’un DOCUMENT et d’un USER qui sont des clefs étrangères. Une contraintre d’unicité à été mise sur l’idDoc afin d’empêcher plusieurs emprunts simultanés d’un même document.

# Application Java EE

## Découplage vis-à-vis de la persistance et injection de dépendance

## Le but est de pouvoir utiliser n’importe quelle implémentation de la médiathèque sans avoir à changer le code des services, car ceux-ci dépendent du package médiathèque qui est stable. Médiathèque agit comme interface du package persistantData qui communique directement avec la base de donnée.

## Si --> et <-- signifient un couplage, on peut illustrer l’organisation de l’application comme suit :

## Services --> médiathèque <-- persistantData

## La seule fois où services fera référence à persistantData sera lors de l’initialisation de la base (avec Class.forName() dans le service « Connection »). Cependant, ce lien n’est effectué que lors du runtime de l’application, et ne casse donc pas l’injection de dépendance mise en place.

## L’architecture de notre application ressemble donc fortement à une architecture MVC, en effet les JSP qui servent d’interface utilisateur constituent les vues de l’application, le package persistantData traitent toutes les données et fait le lien avec notre base de donnée, et les package mediathèque et services font le lien entre les vues et le modèle et fournissent la logique de base de l’application.

## On peut noter que la plupart du temps quand un service est appelé, une fois son traitement terminé, l’utilisateur est redirigé vers une page JSP grâce aux méthodes de HttpRequest. Ex : req.getRequestDispatcher("/").forward(req, resp);

## Quand ça n’est pas le cas, c’est tout simplement la méthode sendRedirect() qui est utilisée.

## Passage objet-relationnel pour les documents : comment passe-t-on de l’interface Document au schéma relationnel et aux tables Oracle ?

## Maintenance évolutive (prêts de CDs musique, de CD-Jeux, de DVDs) ?

## Comme dit précédemment, nous avons opté pour de l’héritage ascendant au sein de la base de données. Cependant, la philosophie de Java nous pousse à utiliser différentes classes pour les différents types de document (Livres, CD, DVD etc...).

## Nous avons donc créé une classe abstraite « BaseDocument » dans le package persistantData. Cette classe implémente l’interface stable Document du package médiathèque. Les méthodes emprunter() et retour() sont donc codées dans cette classe étant donné que nous n’avons pas besoin de connaître la nature exacte d’un document pour l’emprunter ou le retourner.

## Nous avons egalement créé des classes Livre, CD, et DVD héritant de BaseDocument.

## Ce sont dans ces classes que se retrouvent la attributs spécifiques à chaque document (comme durée pour un CD ou un DVD, nbPages pour un livre). La méthode affiche() est redéfinie ici pour donner ces détails à l’utilisateur.

## Ces différentes classes peuvent donc rendre la médiathèque instable, car si l’on ajoute un nouveau type de document, il est nécéssaire de modifier le code, d’ajouter des new NouveauTypeDocument() selon le type passé en paramètre.

## Une Factory a donc été mise en place afin d’encapsuler ce code instable et d’éviter de répéter le test du type à chaque fois que l’on récupère un document. C’est donc souvent ce rôle qu’aura la factory, elle sert principalement à récupérer un document dans la base afin de le transcrire en objet Java.

## Concernant l’ajout d’un document dans la base, nous avons laissé le traitement dans MediathequeData bien que celui-ci puisse également être instable. En effet, un nouveau type de document est susceptible d’apporter une nouvelle colonne dans la table de la base de données, et il faudra donc modifier cette méthode. Mais encapsuler ce code n’apportait pas vraiment d’intérêt et aurait plutôt compliqué la compréhension générale du code, nous l’avons donc laissé dans MediathequeData.

## A l’avenir donc, si l’on souhaite ajoute un nouveau type de document, il faudra créer la classe de ce nouveau type, modifier la factory, et cette méthode d’ajout dans la base.

## Une difficulté qui s’est fait ressentir est l’absence des paramètres par défaut de Java. En effet, nous avons utilisé des varargs, mais nous ne sommes pas satisfait de cette méthode car il faut s’entendre sur l’ordre des paramètres à passer à la fonction, ce qui peut être source de confusion.

## Variables sessions : la classe Utilisateur et les tables Oracle.

## Lors de la connexion d’un utilisateur à la médiathèque, une session est initialisée. Celle-ci contient alors un object java « Utilisateur » représentant l’utilisateur de l’application.

## La classe Utilisateur contient entre autre son niveau d’autorisation (grâce à son type). Ainsi, un abonné lambda ne pourra qu’emprunter et retourner des documents, alors qu’un bibliothécaire pourra également en ajouter.

## Nous n’avons pas créer de factory ni plusieurs classes java pour les différents types d’utiliateur étant donné qu’il est moins probable que de nouveaux types d’utilisateur arrivent contrairement aux documents. De plus, les JSP doivent également tester le type de l’utilisateur, ce qui aurait été plus compliqué avec plusieurs classes.

## La déconnexion d’un utilisateur est rendue possible en utilisant la méthode invalidate() de la session.

## Concurrence : quels sont les points qui nécessitent une gestion java de la concurrence

## Il est nécéssaire de gérer la concurrence lorsque deux utilisateurs désirent emprunter un document en même temps.

## Cependant, nous n’avons pas géré cette concurrence en java mais grâce à une contrainte d’unicité sur idDoc de la table emprunt dans la base de données.

## Autrement, il aurait fallu soit verrouiller la méthode emprunter() avec un bloc synchronized sur un verrou statique de la classe, mais cela empêchait l’emprunt simultané de documents différents, ce qui pourrait causer des ralentissements à grande échelle.

## Sinon, il aurait fallu stocker les objets java Document dans la MediathequeData et appliquer un verrou sur leur méthode emprunter, mais nous ne stockons pas les objets dans cette application. Il sont créés et détruits à la fin du traitement de chaque services les manipulant.