

# Gestion d'un Tetris - Version JSP

Jérémy PERROUAULT – jeremy.perrouault@ascadis.fr

---

ASCADIS © - 2017

## Gestion d'un Tetris en Java EE, JSP

### Description

Page d'accueil et de connexion - /home

Liste des tetriminos disponibles - /tetriminos

Editer et créer un tetrimino

Fonctionnalités à réaliser

Fonctionnalités facultatives

Etapes de développement

Modèle de données

Pages *JSP*

Page /home

Page /tetriminos

Protection des vues *JSP* avec la classe `Rendu`

Protection des URL si l'utilisateur est déconnecté

## Description

Le projet à réaliser permettra, à terme, de jouer à un jeu de Tetris dynamique. Commençons par créer une application capable de gérer un ensemble de Tetriminos.


Pour proposer un design plus moderne, l'application utilisera la bibliothèque CSS *Materialize* (<http://materializecss.com/>).

L'application sera composée de plusieurs pages, dont voici leur rôle.

**A l'exception de la page d'accueil, les pages ne sont pas accessibles par l'utilisateur s'il n'est pas connecté.** Donc, dans notre menu, les liens seront cachés à l'utilisateur non connecté.

## Page d'accueil et de connexion - /home

## Tetris - Bienvenue

 Votre nom

 Votre mot de passe

OK

>

Dans l'illustration ci-dessus, il s'agit de la page d'accueil. Si l'utilisateur n'est pas connecté, cette page affiche un formulaire de connexion. Dans le cas contraire, la page affiche un message de bienvenue.




La barre de navigation doit inclure les autres liens une fois l'utilisateur **connecté** ! (voir l'illustration ci-dessous)

## Tetris - Bienvenue

Accueil Tetrminos


# Bienvenue Jeremy !

SE DÉCONNECTER 

Liste des tetrminos disponibles - /tetrminos

## Tetris - Liste des tetrminos

Accueil Tetrminos

J

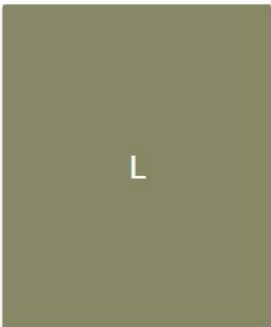
J

:

Détails

RETIRER

EDITER

L


L

:

Détails

RETIRER

EDITER



La page présentée ci-dessus permet de visualiser la liste des tetrminos disponibles. Ces pièces sont communes à toute l'application, et donc à tous les utilisateurs.

Pour modéliser un tetrmino, vous créez une classe `Tetrmino`. Elle possèdera les attributs suivants : `String id`, `String nom`, `String couleur`. Les identifiants pourront être générés avec la classe `UUID` pour garantir la génération d'une chaîne de caractères unique (`UUID.randomUUID().toString()`).

Visuellement et pour chaque tetrmino, 2 boutons sont disponibles :

- **RETIRER** : permet de supprimer un tetrmino de la liste des tetrminos disponibles
- **EDITER** : permet de modifier les paramètres d'un tetrmino

Le bouton + permet de créer un nouveau tetrmino pour tous les utilisateurs.

## Editer et créer un tetrmino



Lorsque l'utilisateur crée un nouveau tetrmino (ou édite un tetrmino existant), il est amené vers la page d'édition, présentée ci-dessus.

Une fois le formulaire rempli et validé, l'utilisateur est redirigé vers la page `/tetrminos`. Les données des tetrminos doivent être mises à jour.

## Fonctionnalités à réaliser

- Accueil : formulaire de connexion et message de bienvenue
- Tetrminos disponibles :
  - a. Afficher la liste des tetrminos
  - b. Implémenter l'ajout d'un tetrmino
  - c. Implémenter la suppression d'un tetrmino

#### d. Implémenter l'édition d'un tetrimino

## Fonctionnalités facultatives

Proposer une page d'inscription permettant d'ajouter des utilisateurs. Vous prendrez en compte cette évolution dans la méthode d'authentification.

## Etapes de développement

Commencez par créer un projet "Dynamic Web" depuis Eclipse.

La correction de ce projet utilisera le serveur d'application **Apache Tomcat**, mais vous êtes libre d'utiliser celui de votre choix.

## Modèle de données

Voici le modèle de données pour cet exercice :

- la classe `Tetrimino` représente les tetriminos
- les tetriminos disponibles le sont pour toute l'application, ils seront stockés dans le scope (ou la portée) *application*

## Pages JSP

Les pages *JSP* devront ne pas être accessibles aux internautes. Il faudra donc qu'elles soient placées dans le répertoire `/WEB-INF/views/`. Vous les appellerez depuis vos *Servlet* dans ce répertoire.

## Page `/home`

Implémentez une *Servlet* `HomeServlet` pour gérer les pages de connexion et d'accueil.

Donc cette *Servlet* devra vérifier si l'utilisateur est connecté. On utilisera l'attribut du scope *session* `"username"` pour cela. L'utilisateur sera considéré comme connecté si on trouve une chaîne de caractères dans cet attribut de *session*.

Ecrivez ensuite 2 *JSP* pour les 2 vues :

- `login.jsp` : formulaire de connexion
- `accueil.jsp` : accueil

Le formulaire de connexion enverra une requête de type POST vers la *Servlet* `LoginServlet`. C'est elle qui sera chargée de traiter les informations du *login* et de stocker l'information dans la *session*.

Ensuite, vous pourrez créer une *Servlet* `LogoutServlet`, qui se chargera de supprimer la session de l'utilisateur, avec l'instruction `req.getSession().invalidate()`.

## Page /tetrminos

Cette page affichera tous les tetrminos communs de l'application. C'est ici que l'utilisateur pourra effectuer les opérations CRUD (*consulter, ajouter, editer* et *supprimer*).

### Définition

Premièrement, il faudra définir notre classe `Tetrimino`. Implémentez cette classe en lui donnant 3 propriétés : `id`, `nom`, et `couleur`. Implémentez également les *getters* et les *setters*. Implémentez un constructeur prenant des valeurs pour `nom` et `couleur`, et qui génère un identifiant avec la classe `UUID`.



Cette classe implémentant un constructeur avec paramètres n'est pas valide dans la nomenclature POJO. Il faut ajouter un constructeur sans paramètre.

Comme dit plus haut, la liste des tetrminos doit être globale à toute l'application. Pour y accéder depuis n'importe où, créez une classe `TetriminoApplicationDAO` possédant un attribut statique `Map<String, Tetrimino> tetrminos`. Puis implémentez une méthode statique `findAll()` qui permettra de récupérer la liste des tetrminos.

JAVA

```
public class TetriminoApplicationDAO {  
    private static Map<String, Tetrimino> tetrminos = new HashMap<String, Tetrimino>  
    ();  
  
    public static List<Tetrimino> findAll() {  
        return new ArrayList<>(tetrminos.values());  
    }  
}
```

### Consultation

Maintenant que notre modèle de données est établi, implémentez la *Servlet* `TetriminosServlet` et la vue *JSP* associée. La *Servlet* `TetriminosServlet` devra récupérer la liste des tetrminos et la transmettre à la vue *JSP*.

### Suppression

Implémentez cette fonctionnalité dans une nouvelle *Servlet* `DeleteTetriminoServlet`. Cette *Servlet* recevra les requêtes faites sur l'URL `/deleteTetrimino`. L'URL devra avoir comme paramètre l'identifiant du tetrimino à supprimer.

Il faudra que la page des tetriminos génère les liens pointant sur l'URL de suppression de chaque tetrimino.

La *Servlet* `DeleteTetriminoServlet` (comme toutes nos *Servlet* d'action), se contentera de réaliser l'action, puis redirigera le navigateur vers `/tetriminos`.

Pour implémenter la fonctionnalité de suppression, ajoutez une méthode `delete(Tetrimino tetrimino)` dans la classe `TetriminoApplicationDAO`. Vous pourrez utiliser cette méthode depuis la *Servlet* `DeleteTetriminoServlet`.

```
public static void delete(Tetrimino tetrimino) {  
    tetriminos.remove(tetrimino.getId());  
}
```

JAVA

## Edition

Tout comme la fonctionnalité de suppression, l'URL d'édition d'un tetrimino devra prendre comme paramètre l'identifiant du tetrimino correspondant. Elle recevra les requêtes faites sur l'URL `/editTetrimino`.

Il faudra que la page des tetriminos génère les liens pointant sur l'URL d'édition de chaque tetrimino.

Implémentez une nouvelle *Servlet* `EditTetriminoServlet`. Cette *Servlet* analysera le paramètre de la requête HTTP, et utilisera une nouvelle méthode `find(String id)` que vous ajouterez à la classe `TetriminoApplicationDAO`. Cette nouvelle méthode récupérera le tetrimino associé à l'identifiant dans la liste des tetriminos. La *Servlet* alimentera ensuite la vue *JSP* du formulaire d'édition avec l'instance du tetrimino.

```
public static Tetrimino find(String id) {  
    return tetriminos.get(id);  
}
```

JAVA

Le formulaire est écrit dans la *JSP* `editTetrimino.jsp`. Ce formulaire enverra ses données à la *Servlet* `EditTetriminoServlet` en requête POST. La *Servlet* aura donc, encore une fois, besoin de l'identifiant du tetrimino. Vous pourrez utiliser un champ caché `<input type='hidden' name='tetrimino_id' />` pour transmettre l'identifiant.

Lorsque la modification est faite, la *Servlet* redirigera vers `/tetriminos`.

## Création d'un tetrimino

Implémentez cette fonctionnalité en modifiant la classe `EditTetriminoServlet` (ne créez pas de nouvelle *JSP*, ni de nouvelle *Servlet*). Petit indice : Traiter, dans la servlet `EditTetriminoServlet`, le cas où l'identifiant est absent.

Elle utilisera une nouvelle méthode `save(Tetrimino tetrimino)` que vous ajouterez à la classe `TetriminoApplicationDAO`. Cette nouvelle méthode ajoutera le nouveau tetrimino à la liste des tetriminos.

```
public Tetrimino save(Tetrimino tetrimino) {  
    return tetriminos.put(tetrimino.getId(), tetrimino);  
}
```

JAVA

## Protection des vues *JSP* avec la classe `Rendu`

A chaque fois que l'on veut afficher une *JSP*, les étapes suivantes sont déroulées :

- Alimentation de la *JSP* avec les données en les attachant à la requête sous forme d'attributs
- *Forward* du traitement de la requête à la *JSP*
- Exploitation des attributs dans la *JSP*

Et tout ça, de façon non formalisée qui nécessite l'écriture de plusieurs lignes de code. Créez une classe `Rendu` qui contiendra autant de méthodes statiques que de vues *JSP* à afficher.

Par exemple, pour afficher la vue *JSP* de connexion, vous écrirez la méthode suivante :

```
public static void pageLogin(ServletContext context, HttpServletRequest req,  
    HttpServletResponse resp) throws ServletException, IOException  
{  
    RequestDispatcher dispatcher = context.getRequestDispatcher("/WEB-INF/views/login.jsp");  
    dispatcher.forward(req, resp);  
}
```

JAVA

Dans ce cas précis, on ne voit que peu l'intérêt de cette construction. Mais dans le cas d'une vue *JSP* nécessitant des données en entrée :

```

public static void listeTetriminos(String titrePage, List<Tetrimino> tetriminos,
boolean montrerActions, ServletContext context, HttpServletRequest req,
HttpServletRequestResponse resp)
    throws ServletException, IOException
{
    //On alimente les attributs dont la vue JSP a besoin
    req.setAttribute("tetriminos", tetriminos);
    req.setAttribute("montrerActions", montrerActions);

    //On affiche la vue JSP
    RequestDispatcher dispatcher = context.getRequestDispatcher("/WEB-INF/views/tetriminos.jsp");
    dispatcher.forward(req, resp);
}

```



Vous remarquez cependant que vous devez redéfinir la structure *HTML* pour chaque fichier *JSP*. Utilisons la nouvelle classe **Rendu**, qui délèguera vers un nouveau fichier *JSP* **structure.jsp**, qui lui-même inclura la *JSP* de contenu.

```

public static void listeTetriminos(String titrePage, List<Tetrimino> tetriminos,
boolean montrerActions, ServletContext context, HttpServletRequest req,
HttpServletRequestResponse resp) throws ServletException, IOException
{
    req.setAttribute("tetriminos", tetriminos);
    req.setAttribute("montrerActions", montrerActions);

    pagePrincipale(titrePage, "/WEB-INF/views/tetriminos.jsp", context, req, resp);
}

```

```

public static void pagePrincipale(String title, String contentJsp, ServletContext
context, HttpServletRequest req, HttpServletResponse resp) throws ServletException,
IOException
{
    if (title == null)
        title = "Défaut";

    req.setAttribute("pageTitle", title);
    req.setAttribute("contentJsp", contentJsp);

    RequestDispatcher dispatcher = context.getRequestDispatcher("/WEB-INF/views/structure.jsp");
    dispatcher.forward(req, resp);
}

```



Dans l'exemple ci-dessus, `listeTetriminos()` appelle `pagePrincipale()`, qui a le rôle de délégation vers **structure.jsp**. Puis dans **structure.jsp**, le code peut se présenter ainsi.



```
<%@ page import="java.util.List" %>
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8" %>

<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />

    <!-- Materialize -->
    <link href="http://fonts.googleapis.com/icon?family=Material+Icons"
rel="stylesheet">
    <link type="text/css" rel="stylesheet" href="css/materialize.min.css"
media="screen,projection" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <script type="text/javascript" src="https://code.jquery.com/jquery-
2.1.1.min.js"></script>

    <title>${ pageTitle }</title>
  </head>

  <body>
    <jsp:include page="navigation.jsp" />

    <div class="container">
      <jsp:include page="${ contentJsp }" />
    </div>

    <script type="text/javascript" src="js/materialize.min.js"></script>
  </body>
</html>
```

Maintenant et à chaque fois que vous voulez afficher une vue, vous utiliserez la méthode correspondante dans la classe `Rendu`.



Etant donné que chaque méthode a un prototype bien défini, une erreur de compilation sera déclenchée si vous omettez un paramètre, ce qui n'était pas le cas avant la mise en place de cette classe.

De plus, lorsque le projet évoluera, il vous suffira de modifier à **un seul endroit** l'appel à la vue *JSP*.

Cette technique se rapproche du pattern *Facade*, qui permet d'obtenir un contrat simple (les méthodes statiques) à partir d'un contrat plus complexe, ou moins pratique.

### Utilisation des constantes : noms des paramètres et attributs

Donnez-vous les moyens de vous protéger de vos propres erreurs et créez une interface `Constantes` qui contiendra des constantes que vous utiliserez à la place des noms de paramètres et attributs. "userName" n'est pas la même chose que "username".

## Utilisation d'un écouteur

Comme la majorité des outils, framework et bibliothèques, vous avez la possibilité d'insérer des traitements à des points précis du cycle de vie de l'outil que vous manipulez (Java EE dans notre cas). C'est un principe connu sous le nom de *Open Close Principle*.

Nous l'avons vu en cours, Java EE vous permet de déclencher un traitement au moment (par exemple) où Java EE s'initialise, et juste avant qu'il ne commence à prendre en charge les requête HTTP.

Il suffit de créer une classe qui implémente l'interface `javax.servlet.ServletContextListener`, et de la déclarer dans le fichier `web.xml`, ou de l'annoter de `@WebListener` :

```
<listener>
  <listener-
class>fr.ascadis.listener.ApplicationDataInitializationListener</listener-class>
</listener>
```

XML

Lorsque Java EE démarre, la méthode `public void contextInitialized(ServletContextEvent event)` de notre classe est appelée. Et lorsque le serveur s'arrête, il appellera la méthode `contextDestroyed`.

## Préparation au changement du fournisseur de données

Pour le moment, notre projet fonctionne avec une liste stockée en mémoire, sous la forme d'un attribut statique dans la classe `TetriminoApplicationDAO`. Mais d'ici quelques exercices, nous utiliserons une base de données.

Or, notre conception jusqu'ici a un problème : toutes les méthodes de `TetriminoApplicationDAO` sont statiques, et donc chaque appel se fait comme ça :

```
TetriminoApplicationDAO.findAll()
```

JAVA

Le défaut majeur de ce design, c'est que dans la même ligne, nous spécifions la méthode à appeler (ça, c'est correct), mais nous désignons aussi son implémentation (on demande celle de la classe `TetriminoApplicationDAO`). Or, préciser la classe d'implémentation nous empêche de faire évoluer le code vers une autre implémentation (stockage en base de données, gestion différente selon la configuration, ...).

Pour rendre notre code plus modulaire (et par conséquent, plus conforme à la vision *objet*), il faut créer une interface.

Créez une interface `IDA0` qui reprend les méthodes de `TetriminoApplicationDAO`, tout en les rendant non-statiques.

Ensuite, implémentez l'interface `IDA0` dans la classe `TetriminoApplicationDAO`. Toutes les méthodes deviennent elles aussi non-statiques.

Puisque les méthodes de `TetriminoApplicationDAO` ne sont plus statiques, nous avons besoin d'une instance de classe pour accéder à nos données, mais les problèmes suivants interviennent :

- Quand est-ce qu'on doit créer l'instance unique de `TetriminoApplicationDAO` ?
- Où stocker cette instance ?
- Comment retrouver cette instance lorsqu'on en a besoin ?



Nous aurions pu utiliser le pattern Singleton pour répondre à ce problème, mais il est préférable d'utiliser les fonctionnalités offertes et *garanties* par le serveur d'application.

Pour répondre à la première question, regardez la section précédente, sur les écouteurs. Nous allons les utiliser pour initialiser notre instance de la classe `TetriminoApplicationDAO`.

Pour répondre à la deuxième question, nous voulons que cette liste de tetrminos soit disponible dans toute l'application, donc nous pouvons utiliser le *scope application* de Java EE. Il faut utiliser le contexte de *Servlet*.

```
@Override
public void contextInitialized(ServletContextEvent event)
{
    IDA0 tetriminoDAO = new TetriminoApplicationDAO();
    event.getServletContext().setAttribute("tetriminoDAO", tetriminoDAO);
}
```

JAVA

Pour répondre à la troisième question c'est maintenant évident. Pour récupérer notre instance `IDA0`, il nous faut un accès au contexte *Servlet*.

```
IDA0 tetriminoDAO = (IDA0)context.getAttribute("tetriminoDAO");
```

JAVA

Pour éviter d'écrire ce code de façon répétitive, vous le factoriserez dans une méthode statique, par exemple dans une classe `TetriminoDataAccess`.

JAVA

```
public class TetriminoDataAccess
{
    public static IDAO get(ServletContext context) {
        return (IDAO)context.getAttribute("tetriminoDAO");
    }
}
```

Et nous pourrions utiliser ce code depuis nos *Servlet*.

JAVA

```
List<Tetrimino> tetriminos =
TetriminoDataAccess.get(this.getServletContext()).findAll();
```

## Création d'une classe abstraite `DataAccessServlet`

Remarquez qu'il faut toujours passer la valeur `getServletContext()`. Vous pouvez créer une classe abstraite `DataAccessServlet` qui héritera de `HttpServlet` et qui fournira une méthode pour accéder à l'instance IDAO.

JAVA

```
public abstract class DataAccessServlet extends HttpServlet
{
    protected IDAO getTetriminoDAO() {
        return (IDAO)this.getServletContext().getAttribute("tetriminoDAO");
    }
}
```

Nos *Servlet* pourront en hériter (à la place de `HttpServlet`).

JAVA

```
List<Tetrimino> tetriminos = this.getTetriminoDAO().findAll();
```

Exemple complet avec la *Servlet* `TetriminosServlet` ci-dessous.

JAVA

```
@WebServlet("/tetriminos")
public class TetriminosServlet extends DataAccessServlet
{
    private static final long serialVersionUID = 1L;

    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        Rendu.listeTetriminos("Liste des tetriminos", this.getTetriminoDAO().findAll(),
true, this.getServletContext(), req, resp);
    }
}
```

N'est-ce pas plus agréable ?



Les *Injectons de dépendance* (pattern *Inversion de Contrôle*) existent et sont plus évoluées pour utiliser les données et les composants applicatifs au sein du code. Nous le verrons plus tard.

## Protection des URL si l'utilisateur est déconnecté

Cacher le menu lorsque l'utilisateur n'est pas connecté, c'est bien. Mais si celui-ci connaît les URL, il peut toujours accéder aux fonctionnalités de l'application. Vous allez donc empêcher ces actions en utilisant un filtre, qui sera déclenché avant chaque appel à une *Servlet*.



Le filtre que vous allez mettre en place n'est utile que pour ce projet, dans le cadre d'une démonstration des filtres. Vous ne devez **EN AUCUN CAS** utiliser cette technique pour sécuriser des projets de production. La sécurité est un problème très complexe, et nous nous appuyons sur des implémentations matures et correctement testées (*Spring Security* avec *Spring*).

Implémentez, dans un filtre, l'algorithme qui suit :

- Si l'URL demandée concerne un fichier de ressource ("js/", "css/", "font/", ...), laissez le traitement se poursuivre
- Si l'URL demandée est la page d'accueil, laissez le traitement se poursuivre
- Si l'URL demandée est la page de connexion, laissez le traitement se poursuivre
- Dans tous les autres cas
  - a. Si l'utilisateur est connecté, laissez le traitement se poursuivre
  - b. Sinon, redirigez l'utilisateur vers la page de connexion



Utilisez `request.getRequestURI()` pour vérifier l'URL demandée.

Last updated 2017-02-07 22:54:02 Paris, Madrid