

DOCUMENTATION TECHNIQUE PROJET SPINALTOOLBOX WEB DANS LE CADRE DU COURS INF4990

Version 3 : 10 Décembre 2014

Sommaire

[Introduction 3](#_Toc405935139)

[Partie 1 4](#_Toc405935140)

[1.1- Serveur Tomcat. 4](#_Toc405935141)

[1.2- CONFIGURATION DE L’ENVIRONNEMENT DE DEVELLOPEMENT INTELLIJ 5](#_Toc405935142)

[1.3- Déploiement du site en ligne : 12](#_Toc405935143)

[1.4- Architecture de l’application : 13](#_Toc405935144)

[Partie 2 17](#_Toc405935145)

[2.1- Fonctionnement général de l’UI 17](#_Toc405935146)

[2.2- Upload d’une image IRM vers le serveur distant : 17](#_Toc405935147)

[2.3- Génération dynamiques des formulaires des Process : 18](#_Toc405935148)

[2.4- Gestion des fichiers/ dossiers/ volumes affichés : 19](#_Toc405935149)

[2.5- Suppression fichier côté UI et côté serveur : 20](#_Toc405935150)

[2.6- Gestion des volumes affichables de brainbrowser : 21](#_Toc405935151)

[2.7- Suppression du répertoire de l’upload sur le serveur : 22](#_Toc405935152)

[Partie 3 23](#_Toc405935153)

[Accélération du site – idées : 23](#_Toc405935154)

# Introduction

Le présent document a pour objectif d’éclaircir la structure générale du projet et de faciliter des futures modifications et expliquer la procédure de déploiement.

Dans un premier temps, nous expliquerons le fonctionnement du serveur tomcat, puis nous verrons la configuration de l’environnement, le déploiement et la mise en ligne du site. Nous terminerons avec le fonctionnement général de l’interface utilisateur.

# Partie 1

## Serveur Tomcat.

Tomcat est un programme open source qui permet de répondre aux requêtes https. Sa caractéristique est d’implémenter les servlets java et les pages java serveur.

Il est composé de trois principaux modules : Jasper qui est le moteur de fichiers JSP, Catalina qui est le conteneur de servlet et Coyote qui est le connecteur http de Tomcat.

L'installation par défaut de Tomcat comprend les répertoires suivants :

* bin : Scripts et exécutables pour différentes tâches : démarrage (startup), arrêt, etc. notamment le fichier catalina.sh qui permet le lancement et l'arrêt du serveur tomcat
* common : Classes communes que Catalina et les applications Web utilisent ;
* conf : Fichiers de configuration au format XML et les DTD que ces fichiers XML utilisent ;
* lib : Le répertoire contenant les bibliothèques logicielles (fichiers .jar
* logs : Journaux des applications Web et de Catalina : notamment le fichier catalina.out
* server : Classes utilisées seulement par Catalina ;
* shared : Classes partagées par toutes les applications Web ;
* webapps : Répertoire contenant les .war);
* work : Fichiers et répertoires temporaires (le cache). [[1]](#footnote-1)

Le site web déployé doit se trouver dans le dossier webapps qui comme le nom l’indique convient les applications WEB.

## CONFIGURATION DE L’ENVIRONNEMENT DE DEVELLOPEMENT INTELLIJ

C’est le logiciel utilisé pour ce projet en vue de son efficacité par rapport à Eclipse.

Pour rendre en mode développement :

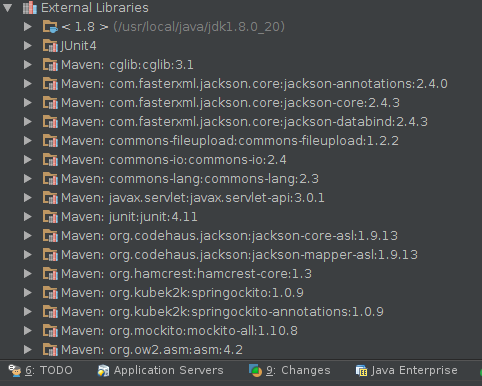
- Télécharger le code

- Lancer IntelliJ et faire un *Open Project* sur le dossier (pas sur le fichier .iml)

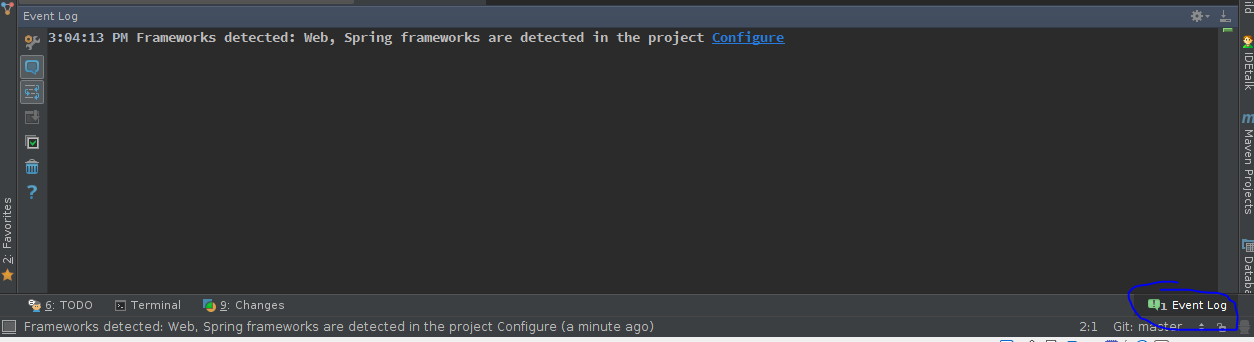
- Automatique *Maven* télécharge les librairies définies dans le fichier *pom.xml*.

Ce fichier représente nos dépendances. Une fois une dépendance rajoutée, Maven télécharger la librairie Maven de l’entrepôt Maven et la rajoute à notre projet.

Dans la section *External librairies* nous observons les librairies téléchargées par Maven :

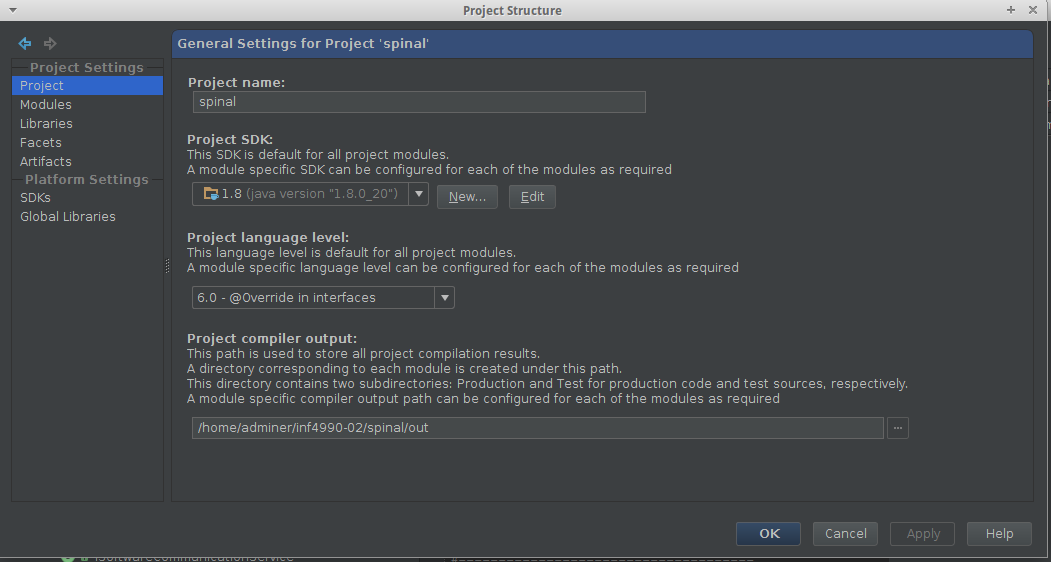


Suite à ceci *IntelliJ* indexe notre projet. Une fois l’indexation terminée, IntelliJ détecte le framework Spring. Une infobulle apparait en haut. Si vous la ratez, il y’a un Event log tout en bas. Cliquer sur configurer.

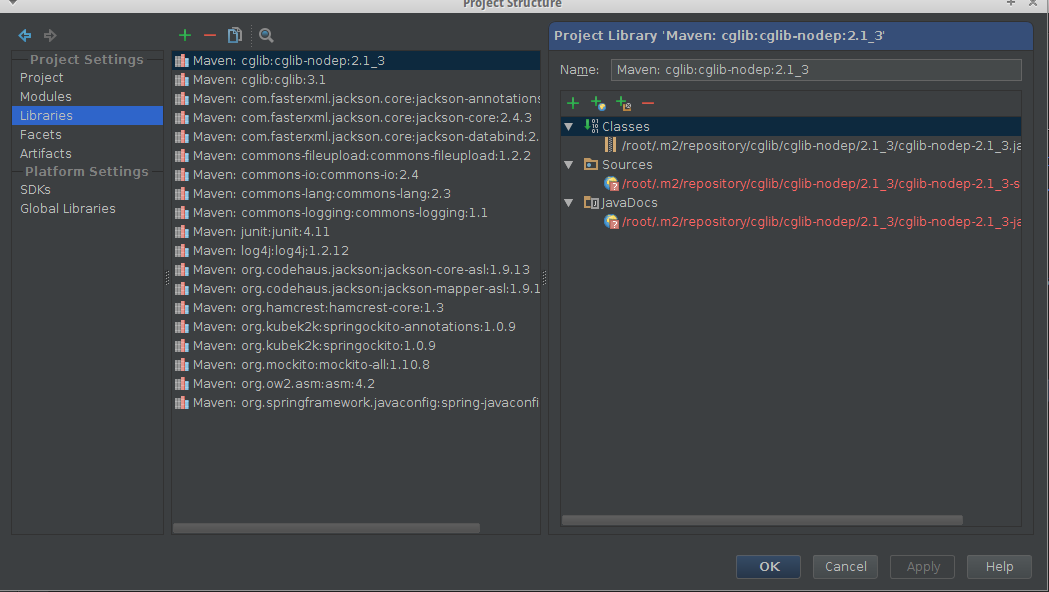


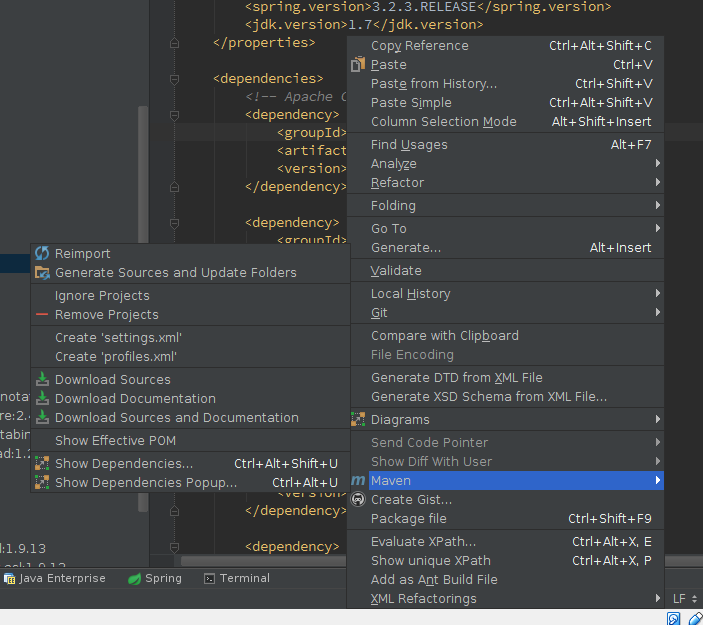
Aller dans File- > Project Structure.

**Dans l’onglet Project,** spécifier la version java SDK utilisée.



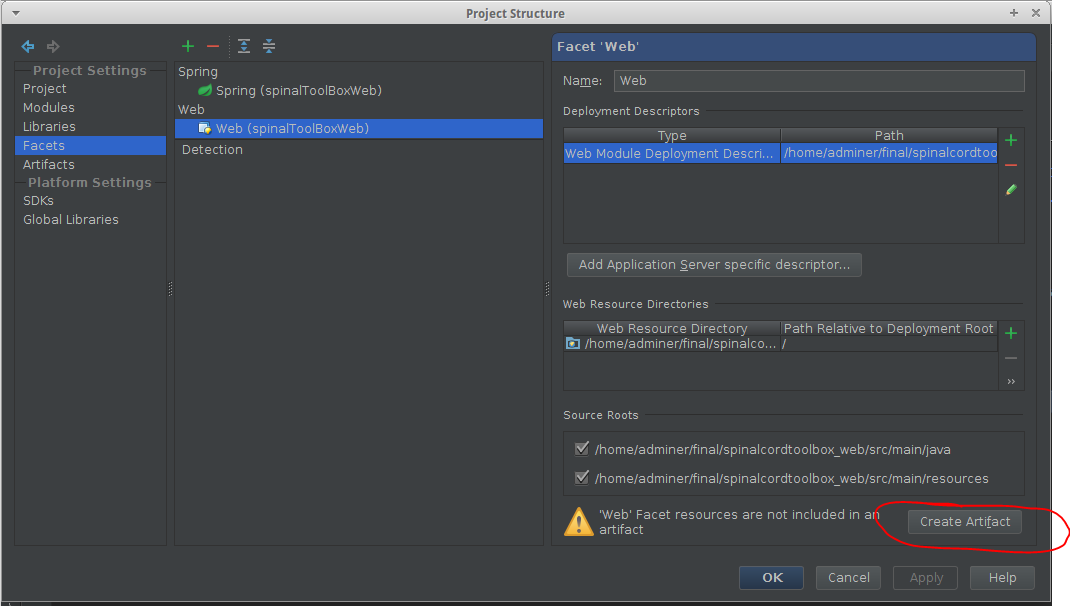
**Dans l’onglet Librairies,** normalement après que Maven ait téléchargé les librairies, vous verrez quelque chose de semblable :



Note : S’il se trouve que Maven ne télécharge pas les librairies, aller dans le fichier pom.xml, click droit -> Maven -> Reimport

*Re-import Maven dependencies.*

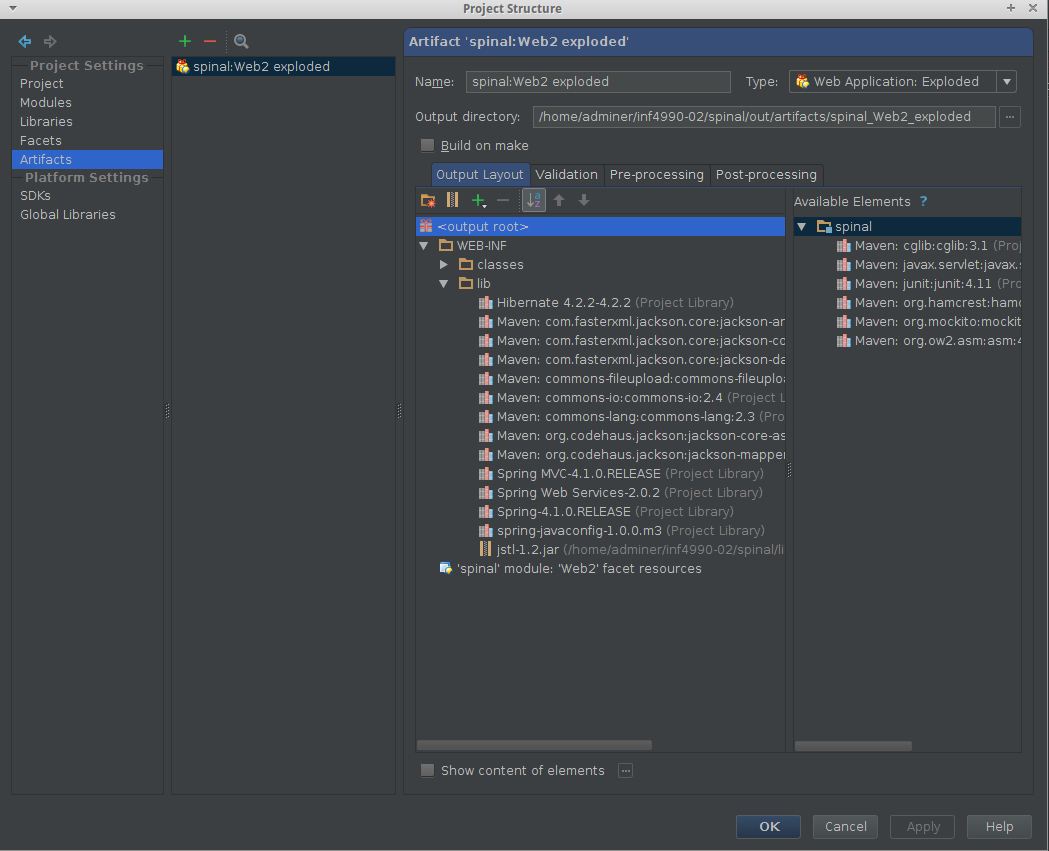
**Dans l’onglet Facet**, cliquez sur Create Artifact. L’artifact représente toute le code et ses ressources.



**Dans l’onglet Artifacts**, vous verrez un artifact de créé. Dans la section output layout, vous verrez à gauche les librairies incluses dans l’artifact et à droite les librairies disponibles.

***Note :*** Il arrive que pendant le développement du code dans IntelliJ qu’aucune erreur n’apparaisse mais que lorsque l’on teste l’application dans le navigateur, il y’ait des erreurs qui apparaissent.

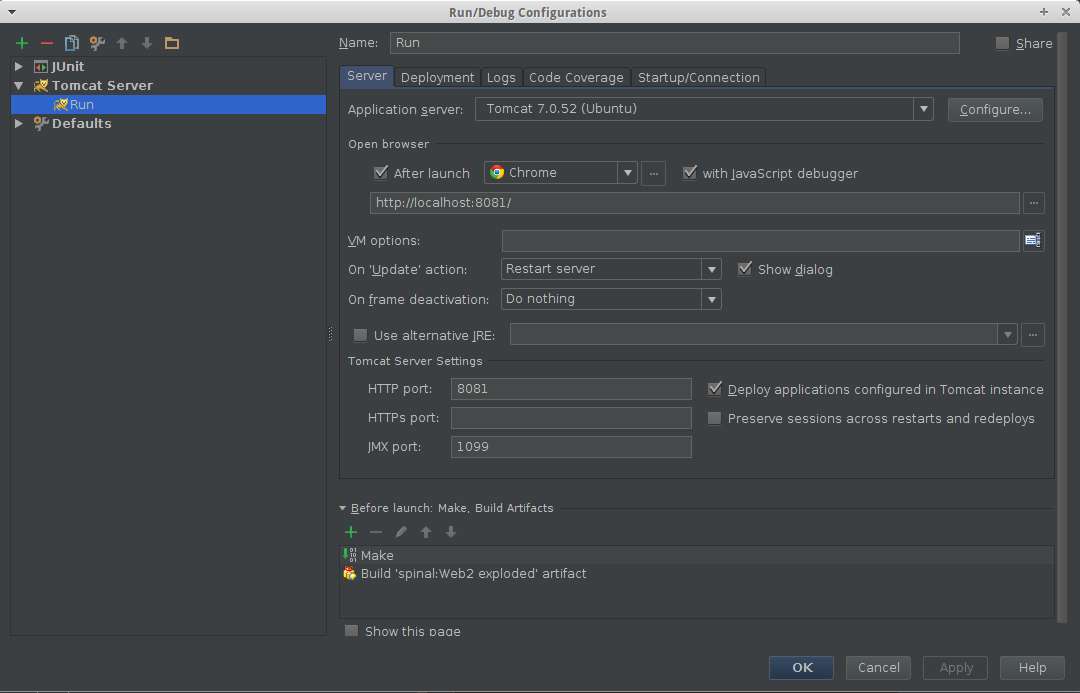
Dans ce cas, il faut prêter attention à la classe qui pose problème et si le cas s’applique, ramener la librairie contenant cette classe de la section librairie disponible à celle incluse dans l’artifact.

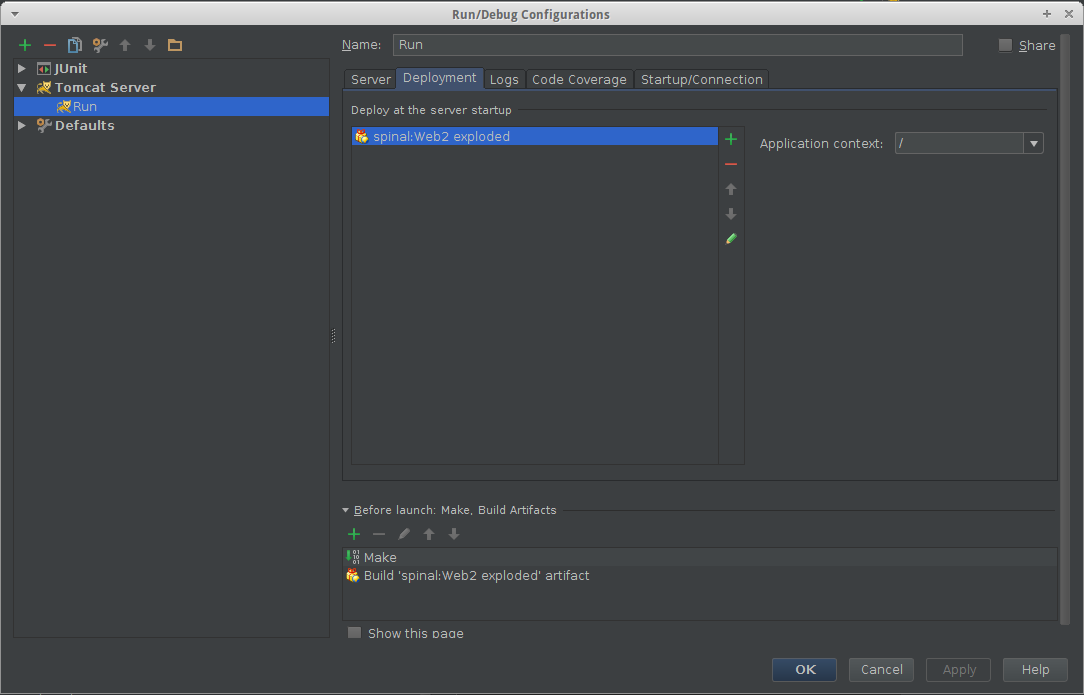


Aller dans *Run -> Edit Configuration* – *cliquer sur +* et choisir *Tomcat Server local*.

Spécifier le chemin du serveur Tomcat, le navigateur pas défaut, quelle action faire à l’update, …

***Important :*** Dans l’onglet déploiement, cliquer sur + et rajouter l’artifact créé précédemment.





## Déploiement du site en ligne :

1- Modifications des paramètres du panneau d’administration du nom de domaine – (voire vidéo sur YouTube) – (pas nécessaire si déjà effectué)

2- Une fois que le nom de domaine pointe sur notre serveur (Tristano dans le cas présent), il faut rediriger les connections entrantes du port 80 vers celui de Tomcat (8080).

Ceci se fait via la commande:

* sudo ipfw add 100 fwd 127.0.0.1,8080 tcp from any to any 80 in
* entrer le mot de passe.
* Redemarrer tomcat :
  + cd /usr/local/apache-tomcat-8/bin/
  + ./restart ( script que nous avons créé fin de faire un shutdown et startup). Une alternative serait de faire ./shutdown suivi de ./startup

Note : A chaque redémarrage, il faut rentrer la commande sudo ipfw … car cette règle s’efface après un redémarrage. Nous avons fait un script pour inscrire cette règle au démarrage mais il ne fonctionne pas.

La version du site disponible est celle située dans le dossier ROOT dans webapps du serveur Tomcat (mentionnée plus haut).

Note : En mode développement il est idéal d’utiliser un autre port autre que 8080 car il ya souvent conflit avec Tomcat signalant que le port est utilisé. Juste spécifier dans Run->Edit Configuration et changer le port.

Ressources : voir les videos concernant Gandi et le déploiement après modifications.

## Architecture de l’application :

Afin de découpler les différents modules, nous avons opté pour une architecture en MVC: Modèle, Vue Contrôleur.

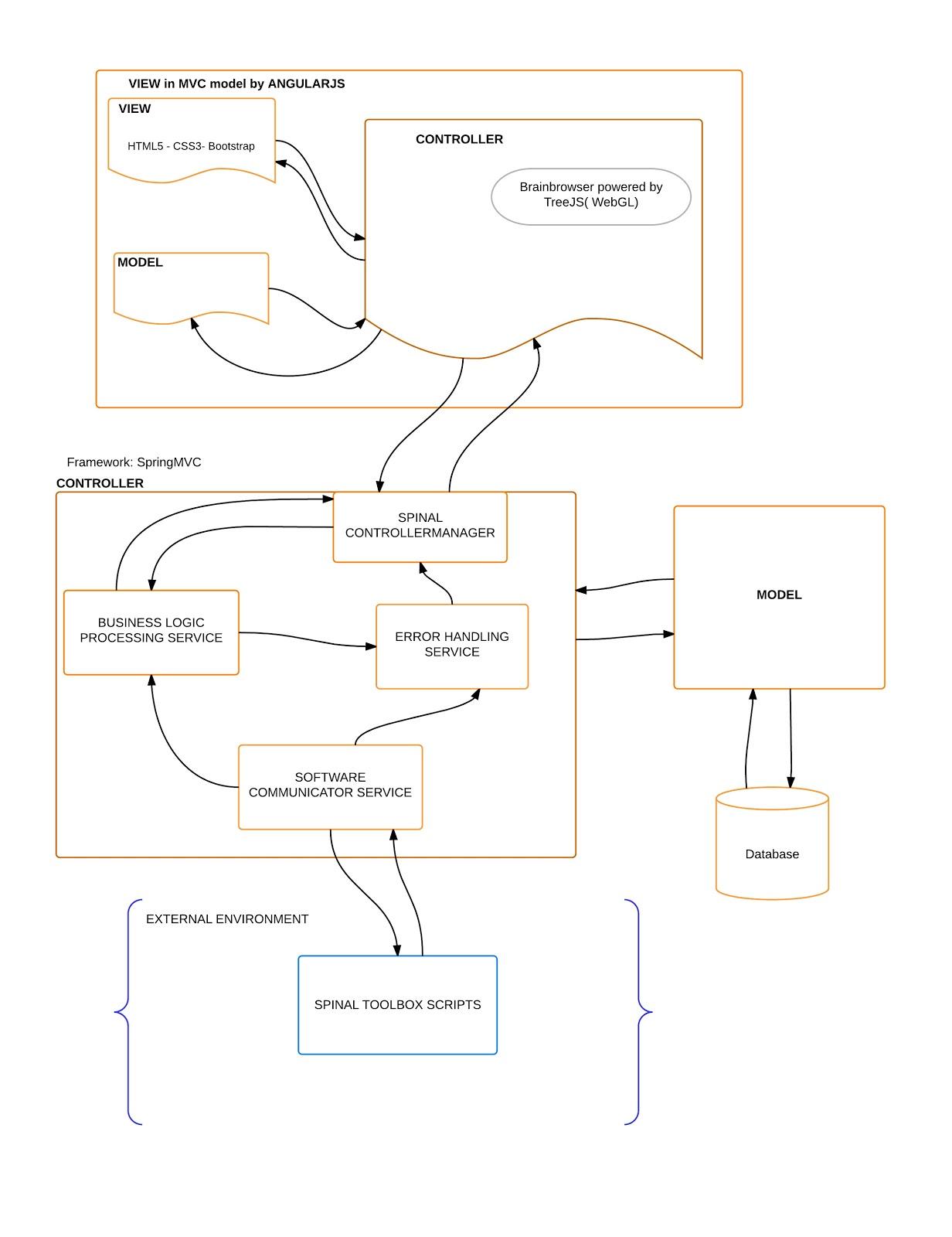


Figure 1- Architecture détaillée du système

La vue représente l’interface graphique de notre site web, la façade entre l’utilisateur et le cœur de notre système.

Le contrôleur, lui se charge de recevoir les requêtes de la vue, de procéder aux traitements des opérations et au besoin, de faire appel au model.

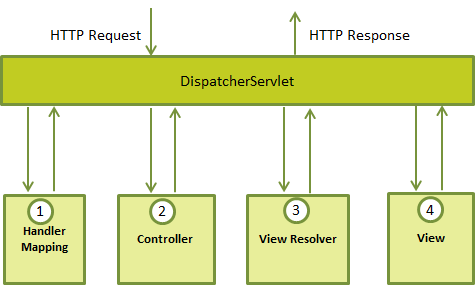


Figure 2- Model-view-controller cadriciel - © Tutorial point

**SpringMVC** nous permet grâce au DispatcherServlet de faciliter le transit des informations. Ce dernier gère les requêtes HTTP venant de la vue et se charge de l’envoi de l’information venant de la vue pour les réponses HTTP.

Une fois une requête http reçue, le *handler mapping* s’occupe de faire la connexion avec l’instance ***bean*** associée à la spécificité de la requête. Ce lien se fait en fonction de l’url envoyée. Une requête effectuée sur le site [www. spinalcordtoolbox.org/](http://www.example.com/listeDesCandidats)neuroPoly fera appel au contrôleur interne neuroPoly tandis qu’une requête effectuée via [www. spinalcordtoolbox.org/](http://www.example.com/tutoriels)inf4990 fera appel au contrôleur interne inf4990.

Une fois le contrôleur interne appelé, celui-ci fait appel aux services appropriés qui se chargent du traitement de la requête. Ci-dessous, quelques services présents dans notre système:

* **FileUploadService** qui se charge de recevoir le fichier envoyé via la vue. Ce dernier récupère les *streams* et créé le fichier demandé dans un répertoire spécifié.
* **LogService** se charge d’enregistrer toutes les transactions faites par les contrôleurs et les autres services afin d’avoir une trace sur les opérations effectuées dans le système. Ce service est appelé par les autres services et peut être appelé dans les contrôleurs.
* **SoftwareCommunicationService** se charge de faire appels aux scripts python et d’obtenir la liste des boîtes à outils chargés. Toutes les communications avec le logiciel passent par ce service.

Dans un service, nous avons:

* L’interface du service qui contient toutes les méthodes à exposer.
* Le contrôleur interne au service afin de traiter les différentes opérations.
* Au besoin, un model utilisé par le service.
* Une classe d’exception afin de personnaliser les exceptions aux services.
* Au besoin, une classe de validation pour des mesures de vérification.

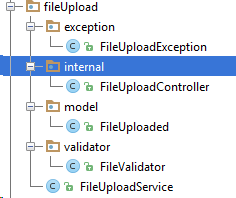


Figure 3- Structure Service

Une fois que le contrôleur termine toutes les opérations, il passe au *DispatcherServlet* le nom de la vue à appeler. Le dispatcher à son tour transmet à la *ViewResolver* le nom de la vue. La *viewResolver*, représentée par la classe *InternalRessourceViewResolver*, possède deux modes de d’opération: un mode par défaut et un mode personnalisé.

Dans le mode par défaut, il faut spécifier le chemin où se trouve la vue ainsi que son nom et aussi l’extension du fichier: par exemple: *WEB-INF/view/index.jsp*

Dans le mode personnalisé, il suffit d’envoyer le nom de la vue.il faut déclarer explicitement dans un **bean**, le préfixe et suffixe. Ce préfixe et suffixe se rajoutent à rajouter au nom de l’index. Cette déclaration est faite juste une fois.

//Resolve view name to jsp

@Bean

ViewResolver viewResolver(){

InternalResourceViewResolver resolver = new InternalResourceViewResolver();

resolver.setPrefix("WEB-INF/view/");

resolver.setSuffix(".jsp");

return resolver;

}

Le résolveur de la vue envoie le chemin relatif du fichier représentant la vue au dispatcher qui renvoie la vue comme réponse à la requête HTTP de l’utilisateur.

En plus d’avoir notre système structuré en MVC, notre vue elle-même constitue un modèle MVC. En effet, une fois le traitement de l’image effectué, le résultat est renvoyé à la vue qui se charge d’afficher un rendu visuel de haute qualité et aussi en 3D grâce à la librairie *Brainbrowser* qui utilise *TreeJs*, librairie fournissant une bonne abstraction du *WebGL*. La structure *MVC* de notre vue est dictée par la libraire angularJs que nous utilisons.

Cette librairie en *JavaScript* récupère les données de l’utilisateur et au besoin opère des opérations sur ces données avant de les envoyer au contrôleur du système.

Un des points forts de notre architecture est la gestion des scripts par des fichiers JSON.

Pour chaque script python, nous avons un fichier JSON permettant de générer dynamiquement dans la page WEB les différents champs et entrées nécessaires à aux paramètres et arguments du script. Voire la documentation technique concernant la création des fichiers JSON.

**Note :** Un aspect important à comprendre est la portée des beans. (Voir livres ressources et aussi Internet).

# Partie 2

## Fonctionnement général de l’UI

Cette présente partie explique le fonctionnement général haut niveau de l’UI, c'est-à-dire les fonctionnalités qu’elle gère et son interaction avec le Backend.

Les fonctionnalités présentes dans l’application côté UI sont listées ci-dessous avec les explications nécessaires.

## Upload d’une image IRM vers le serveur distant :

* Quand l’usager clic sur le bouton upload de l’interface usager, la séquence déclenchée est la suivante :
* Le contrôleur d’angularJS uploadController sous le répertoire upload sera invoqué. Ce dernier appelle la fonction de son modèle pour faire les traitements nécessaires de l’upload.
* La fonction s’assure que :
* L’usager a bien sélectionné une image à uploader
* L’image n’est pas déjà uplodée.
* L’image porte la bonne extension.
* L’usager a fait son premier upload d’images ou non. Dans le cas du premier upload, la fonction génère un token aléatoire qui sera le nom de son répertoire d’upload sur le serveur. Si ce n’est pas le cas, elle garde le token généré à son premier upload.
* Si les conditions ci-dessus sont remplies, la fonction fait un post vers le serveur distant en lui envoyant comme information le fichier à uploadé et le token.
* La fonction reçoit par après la réponse du serveur :
* Si la réponse est positive c'est-à-dire que l’upload s’est bien déroulé côté serveur, alors elle stocke les informations relatives au fichier uploadé tel que le path de son header data, le path de son raw data, le nom du fichier. Ces informations sont utiles par exemple pour faire un GET de l’image côté serveur ou supprimer l’image, etc … Ensuite la fonction ajoute le nom du fichier dans l’arborence des fichiers uploadé pour que l’usager puisse faire des manipulations dessus.
* Si la réponse est négative, un message d’erreur notifiera l’usager.

## Génération dynamiques des formulaires des Process :

Quand l’usager sélectionne un process depuis la liste des process, la séquence déclenchée est la suivante:

* Le contrôleur d’angularJS commandsGeneratorController sous le répertoire commandGeneratorController sera invoqué.
* Ce contrôleur fait une requête HTTP post au serveur pour récupérer le fichier de configuration du process en question.
* Le contrôleur appelle une fonction de son modelé generateDynamicHTML(...) pour générer dynamiquement le formulaire HTML en fonction des paramètres du fichier JSON.

L’usager peut maintenant faire entrer les paramètres de commandes. Au moment où il appuie sur le bouton Run Process, la séquence est la suivante :

* Le contrôleur appelle la fonction verifEmptyMandatoryField(...) de son modelé pour vérifier s’il y’a des champs obligatoires non remplis par l’usager.
* Si tout est correct, le contrôleur fait appel à la fonction generateCommand(...) de son modèle pour associer chaque commande avec l’entrée de l’usager.
* S’il y’a des champs obligatoires non remplis, un message d’erreur notifiera le client.
* Si tout est correct, le contrôleur appelle la fonction sendCommandToServer(...) de son modèle, qui se chargera de faire une requête POST HTTP vers le serveur qui se chargera d’éxécuter la commande en question.
* La fonction reçoit par après les informations de l’exécution de la commande du serveur. S’il y’a des fichiers output genérés côté serveur elle les affiche côté UI dans l’arborescence pour que l’usager fait des traitements avec, en plus elle affiche les output du script exécuté sur le serveur dans l’onglet Output de l’UI. Ceci est utile pour superviser qu’est-ce qui est déroulé réellement comme traitement sur le serveur.
* Les deux modèls du contrôleur commandGeneratorController sont:
* commandOperations : contient les fonctions nécéssaires pour faire toutes sortes de manipulations sur les commandes (i.e envoi au serveur, génération de la commande ...)
* HTMLOperations : contient les fonctions nécessaires pour faire toutes sortes de manipulations HTML (i.e générer le formulaire dynamique, vérifier les champs obligatoires vides du formulaire, initialiser le formulaire ...)

## Gestion des fichiers/ dossiers/ volumes affichés :

Ce module concerne :

* la suppression du nom de l’image uploadé de l’UI et du serveur si l’usager clic sur le bouton de suppression.
* La gestion des volumes d’affichage de brainbrowser en fonction de ce qui est coché comme images dans l’arborescence.

Chacun des deux points précédents est expliqué ci-dessous.

## Suppression fichier côté UI et côté serveur :

Quand l’usager clic sur le bouton de suppression d’une image, la séquence déclenchée est la suivante :

* Le contrôleur d’angularJS filesManagerController sous le repertoire filesManager sera invoqué.
* Le contrôleur appelle la fonction deleteFile(...) de son modèle deletFileOperations qui se charge de:
* enlever le nom du fichier de l’arborescence.
* Appeler la fonction deleteFromServer(...) qui se charge d’envoyer une requête HTTP POST au serveur afin qu’il supprime l’image. La fonction reçoit par après la réponse du serveur pour notifier le client s’il y’a eu des erreurs lors de la suppression.

## Gestion des volumes affichables de brainbrowser :

Ces fonctionnalités sont sous la direction du contrôlleur AngularJS filesManagerController.

Les fonctions du modèle volumesOperations sont invoquées par le contrôleur dans les cas suivants :

* À chaque fois que l’usager décoche une image dans l’arborescence des images téléversées, la fonction displayVolumes(...) du contrôleur appelle la fonction displayVolumes de son modèle pour qu’il mette à jour sa liste d’éléments sélectionnés.

Deux cas de figure se présentent :

* Il n’a pas d’images cochées dans la liste. On fait appel à la fonction generateStartupImageVolume(..) qui se charge de construire le paramètre d’affichage nécessaire à Brainbrowser pour faire sa requête GET côté serveur de l’image vide (fond noir) et l’afficher ensuite dans les panel de BrainBrowser.
* Il y’a des images cochées dans la liste : on fait appel à la fonction generateVolumes(...) qui se charge de créer le paramètre de BrainBrowser contenant les informations nécessaires à ce dernier pour faire les requêtes GET au serveur et afficher notamment ces images par la suite dans les panel de BrainBrowser.

## Suppression du répertoire de l’upload sur le serveur :

Cette action se déclenche au moment où l’usager essaie de fermer l’onglet et confirme sa fermeture.

La fonction du deleteUserRepositoryFromServer() est appelée à partir de la fonction deleteFilesSession() présente dans le fichier sessionClose.js. Ce dernier fichier comporte les fonctionnalités concernant la fermeture de la session de l’usager.

La fonction deleteUserRepositoryFromServer(...) envoi une requête POST au serveur qui se chargera de supprimer le répertoire de l’usager côté serveur.

# Partie 3

## Accélération du site – idées :

- Minifier les fichiers javascripts pour accélérer le chargement des pages. http://jscompress.com/

- Rendre le site non cacheable par le navigateur.

1. Apache Tomcat - http://fr.wikipedia.org/wiki/Apache\_Tomcat [↑](#footnote-ref-1)