RAPPORT PROJET

AYANOU Giovanni – LOHO KOASSIVI

LO54

Olivier RICHARD

PLAN

Table des matières

Introduction

Les tests logiciels sont complexes et prennent beaucoup de temps. Une façon de réduire les efforts associés aux tests consiste à générer automatiquement des données de test. Le test est une partie très importante du développement logiciel. La qualité n'est pas un terme absolu ; c'est une valeur pour une personne. Dans cet esprit, le test ne peut jamais complètement établir l'exactitude des tests de logiciels informatiques. Le test logiciel est une technique d'évaluation des attributs (c'est-à-dire, exactitude, exhaustivité, sécurité, cohérence, absence d'ambiguïté, qualité, etc.) du logiciel et détermine s'il répond ou non à ses fonctionnalités". Le but du test est de découvrir les défauts et les causes et de les corriger le plus tôt possible. Les tests se réfèrent simplement à la validation et à la vérification spécialement pour construire des logiciels de bonne qualité.

Dans ce rapport nous allons mettre en œuvre du Spring MVC (Model Vue Contrôleur) dans une application JavaEE.

I- Qu'est-ce que Spring MVC ?

A- Qu'est-ce que Spring ?

Le Framework Spring est un projet Open Source pour l'environnement JAVA très puissant qui peut ajouter beaucoup de fonctionnalités aux applications Java tel que :

- La gestion transactionnelle
- La gestion des exceptions JDBC
- Framework MVC
- Framework AOP (Programmation Orientée Aspect)

L'un de ses avantages majeurs est que les classes n'ont aucunement besoin d'implémenter une interface pour être prisent en charge cependant, il présente également quelques inconvénients :

- L'application produite est assez lourde
- La mise en œuvre est compliquée à réaliser

Pour faciliter le développement, nous avons opté pour le Framework MVC.

B- Framework MVC

MVC est un pattern de conception visant à découper les applications en couches de bases aux rôles distinctes. (Contrôleur - Dao - Entitée - Service) et des couches additionnelles (Utils et Validators) (à décrire pour gagner plus de place)

En JEE, la vue est classiquement réalisée par des JSP, le contrôleur par une Servlet et le modèle par un JavaBean.

Le Modèle est généralement chargé de :

- Communiquer avec la source de données via des requêtes ;
- S'assurer de l'intégrité des données ;
- De notifier au contrôleur les changements ;

Le contrôleur s'assure de :

- Recevoir les données des différents modèles ;
- Valider ou éditer les formulaires ;
- D'envoyer les données aux vues ;
- De recevoir les données renseignées par l'utilisateur au niveau de la vue ;

La vue bien évidemment joue le rôle de :

- Passerelle de communication entre l'utilisateur et le contrôleur ;
- Interface d'affichage des données à l'utilisateur ;
- Zone de traitement des données par l'utilisateur ;

II- Principe de fonctionnement de Spring MVC

Spring MVC est donc un framework qui permet d'implémenter des applications selon le design pattern MVC. Donc, comme tout autre MVC framework, Spring MVC se base sur le principe décrit par le schéma ci-dessous :

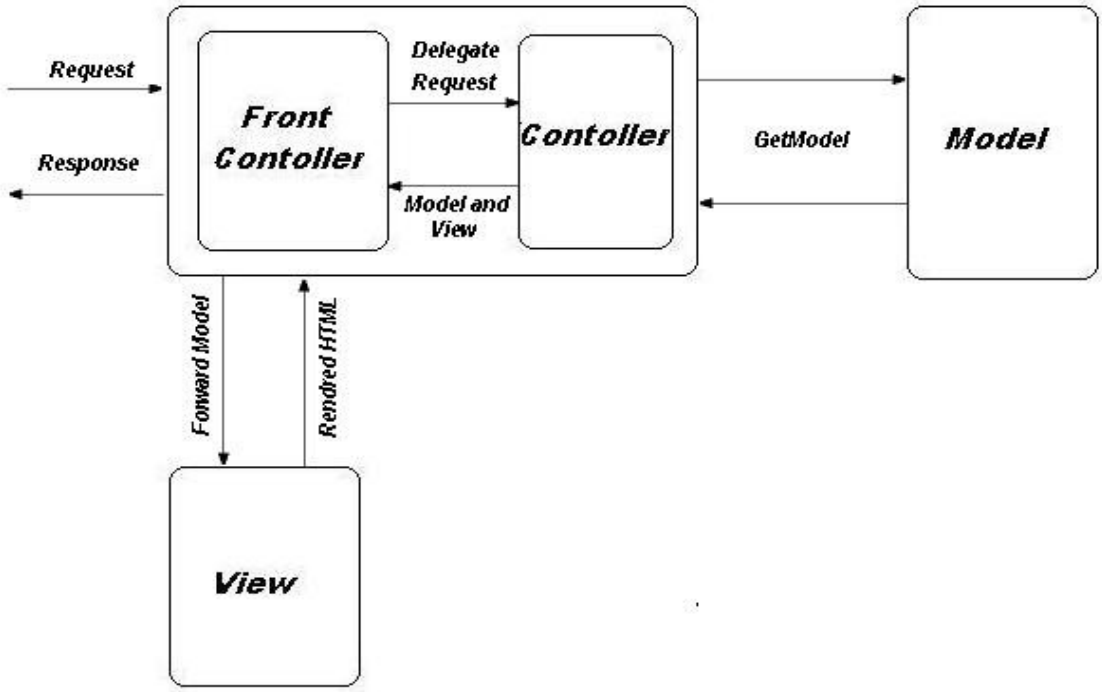

Figure 1: Spring MVC

A- Les étapes de gestion de requête par Spring MVC

Passons maintenant aux détails : ci-dessous la cinématique de la gestion d'une requête par Spring MVC.

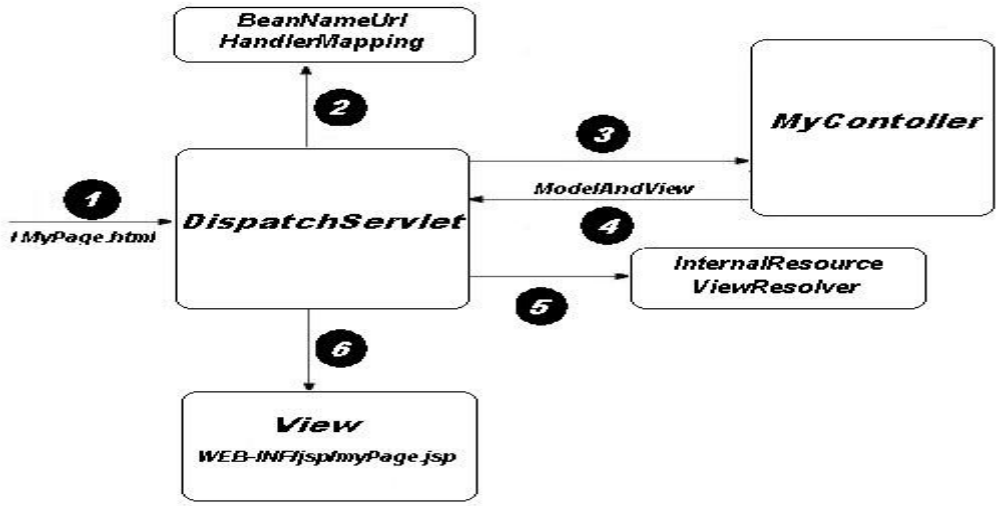

Figure 2: Etapes de Gestion de Requête par Spring MVC

❖ **Description**

1. Le DispatcherServlet reçoit une requête dont l'URI-pattern est '/myPage.html'.

2. Le DispatcherServlet consult son Handler Mapping (Exemple : BeanNameUrlHandlerMapping) pour connaître le contrôleur dont le nom de bean est '/myPage.html'.

3. Le DispatcherServlet dispatche la requête au contrôleur identifié (Exemple : MyController)

4. Le contrôleur retourne au DispatcherServlet un objet de type ModelandView possédant comme paramètre au minimum le nom logique de la vue à renvoyer.

5. Le DispatcherServlet consulte son ViewResolver lui permettant de trouver la vue dont le nom logique est 'exemple. Ici le type de ViewResolver choisit est InternalResourceViewResolver.

6. Le DispatcherServlet expédie la requête à la vue associée ici, la page /WEB-INF/jsp/ myPage.jsp

1) DispatcherServlet

C'est le point d'entrée de l'application, le DispatcherServlet effectue le premier mapping de l'application et distribue les requêtes aux servlets correspondantes.

Exemple : Admettant qu'on a cette configuration dans le web.xml :

```
<servlet>
    <servlet-name>springMVC</servlet-name>
    <servlet-class>
        org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name> springMVC </servlet-name>
    <url-pattern>/myApplication/*</url-pattern>
</servlet-mapping>
```

Avec cette configuration, toutes les urls de notre application commençant par myApplication seront « dispatchées » vers cette servlet.

2) Controller

Le contrôleur (le C dans le modèle MVC) permet d'intercepter la requête et retourner la vue appropriée.

Spring MVC offre plusieurs types de contrôleurs : AbstractController, AbstractController, AbstractCommandController, AbstractFormController, SimpleFormController, AbstractWizardFormController…

L'interface 'Controller' définit le comportement basique d'un contrôleur : intercepter la requête et retourner un 'ModelVue' qui représente le modèle et la vue.

MultiActionController : un contrôleur dont les méthodes seront lancées en fonction des urls d'entrées.

Rappel : avec Spring MVC, vous devez créer vous-même vos Contrôleurs (contrairement à Struts, où il n'y a qu'un controleur : l'ActionServlet).

3) Handler Mappings

Permet de mapper les requêtes HTTP avec le contrôleur correspondant.

4) View Resolvers

Permet de déterminer le nom de la vue.

Exemple : Revenons à notre exemple ci-dessous décrivant la cinématique de la gestion d'une requête par Spring MVC, pour déterminer le nom la page jsp qui correspond au url /myPage nous devons ajouter dans le fichier la configuration ci-dessous :

```
<bean id="jspViewResolver"
    class= "org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="prefix" value="/WEB-INF/jsp/"></property>
    <property name="suffix" value=".jsp"></property>
</bean>
```

Le handler de notre controlller doit retourner un objet de type ModelAndView, celui-ci est déclaré de la manière suivante :

return new ModelAndView("myPage") ;

À partir du nom de vue retourné, myPage par exemple, le viewResolver trouvera la page jsp correspondante qui sera dans le dossier "/WEB-INF/jsp/ » avec le suffixe « .jsp »

III- Implémentation et résultat

Importez le projet dans votre projet IDE (Eclipse, NetBeans, IntelliJ, etc.).

Démarrez le serveur d'applications et lancer votre serveur de base de données.

- Si vous démarrez l'application pour la première fois, avant de lancer l'application, créez la base de données MySql **school_db** puis ajoutez ou décommentez, dans le fichier DispatcherServlet, l'option de création des tables :

  <prop key="hibernate.hbm2ddl.auto">create</prop>

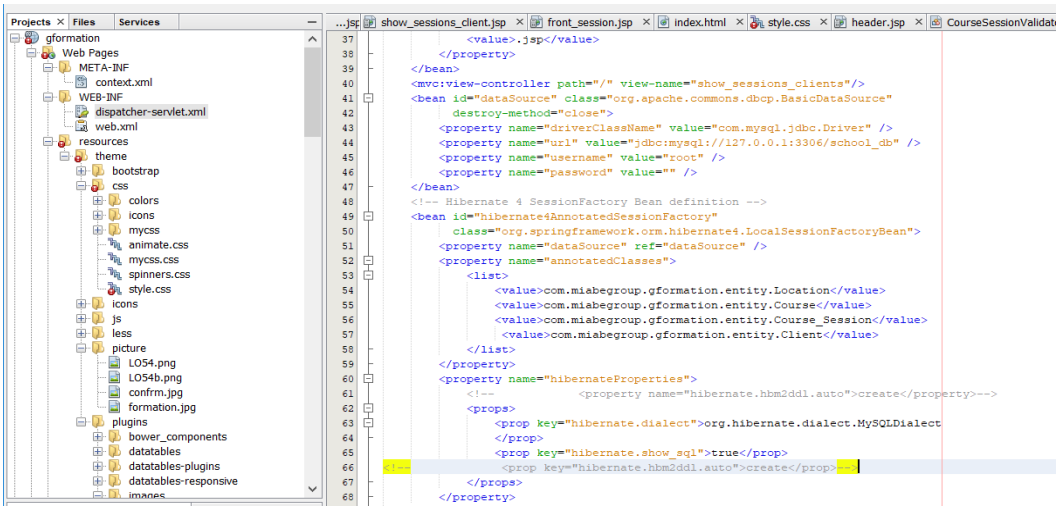

Figure 3: Option Create de Hibernate

- ❖ Sinon lancez directement l'application

Mise en œuvre La mise en œuvre du système est décrite dans les étapes suivantes :

- Lancement du serveur Apache et du serveur de base de données
- Lancement du Serveur Tomcat
- Build de l'application
- Lancement de l'application
- Accès à l'URL

IV- Envoie de Mail

A- JavaMail

Envoyer un Email, en utilisant une application Jee, est chose simple uniquement en utilisant l'API JavaMail. Nous avons donc intégré la version JavaMail 1.5.5 à nos dépendances.

Pour exploiter cet API nous avons créé deux classes :

sendEmail : Pour la configuration des entêtes de mail, des modalités d'envoie (Format d'envoi, l'adresse affichée au destinataire, l'autorisation de répondre au mail ou non).

simpleEmail: Nous y avons donc configuré:

- Le destinateur et son mot de passe (le serveur de mail qui dans notre cas est une adresse Gmail)
- Les paramètre d'accès à Gmail via le port Smtp (L'hôte de l'adresse et son suffixe, le port de connexion à l'hôte, l'autorisation de s'y authentifier et la validation d'authentification sécurisée)

```
    */
    public static void sendEmail(Session session, String toEmail, String subject, String body) {
        try {
            MimeMessage msg = new MimeMessage(session);
            //set message headers
            msg.addHeader("Content-type", "text/HTML; charset=UTF-8");
            msg.addHeader("format", "flowed");
            msg.addHeader("Content-Transfer-Encoding", "8bit");

            msg.setFrom(new InternetAddress("nepasrepondre@fomation.fr", "NoReply-JD"));

            msg.setReplyTo(InternetAddress.parse("nepasrepondre@fomation.fr", false));

            msg.setSubject(subject, "UTF-8");

            msg.setText(body, "UTF-8");

            msg.setSentDate(new Date());

            msg.setRecipients(Message.RecipientType.TO, InternetAddress.parse(toEmail, false));
            System.out.println("Message is ready");
            Transport.send(msg);

            System.out.println("EMail Sent Successfully!!");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Figure 4: Classe SendEmail

```
16  */
17  public class SimpleEmail {
18
19      public static void sendMail(String toEmail, String msg) {
20          final String fromEmail = "ayanougiovanni@gmail.com"; //requires valid gmail id
21          final String password = "Packardbell.1"; // correct password for gmail id
22          //final String toEmail = "myemail@yahoo.com"; // can be any email id
23
24          System.out.println("TLSEmail Start");
25          Properties props = new Properties();
26          props.put("mail.smtp.host", "smtp.gmail.com"); //SMTP Host
27          props.put("mail.smtp.port", "587"); //TLS Port
28          props.put("mail.smtp.auth", "true"); //enable authentication
            props.put("mail.smtp.starttls.enable", "true"); //enable STARTTLS
30
31          //create Authenticator object to pass in Session.getInstance argument
32          Authenticator auth = new Authenticator() {
33              //override the getPasswordAuthentication method
34              protected PasswordAuthentication getPasswordAuthentication() {
35                  return new PasswordAuthentication(fromEmail, password);
36              }
37          };
38          Session session = Session.getInstance(props, auth);
39
40          EmailUtil.sendEmail(session, toEmail, "Formation : Inscription", msg);
41
42      }
43
44  }
45
```

Figure 5: Classe SimpleEmail

B- Implémentation

Vu que nous n'avons pas de serveur mail dédié, nous avons utilisé une adresse mail google personnelle. Avant de pouvoir utiliser cette adresse dans une application externe à google, vous devez soit donner l'autorisation à cette application d'utiliser votre compte ou désactiver la restriction aux applications étrangères de google.

Conclusion

Au vu de tout ce qui précède ce projet a été pour nous une occasion de se familiariser avec les framework et les tests unitaires. Nous avons pu nous rendre compte davantage de l'importance qu'ils représentent en termes de qualité et fiabilité du code puisqu'il nous était impossible de lancer notre application lorsqu'un de nos tests échouait. Ce projet nous aura donné l'opportunité de mettre en pratique toutes les connaissances en Java EE acquises durant le semestre et d'avoir de nouvelles compétences.