B

Développement OSGi dans Eclipse

L'objectif de cette annexe est de venir en complément des chapitres de la partie V afin de décrire la manière d'utiliser l'outil Eclipse afin de développer et de tester des applications OSGi.

L'outil de développement Eclipse est particulièrement approprié afin de réaliser des applications de ce type puisqu'il intègre en natif tous ces mécanismes pour le développement de ses greffons. Eclipse utilise à cet effet la technologie OSGi enrichie avec les points d'extension.

L'outil se fonde sur le conteneur OSGi Equinox aussi bien pour le développement que pour l'exécution. Ainsi, dans le cadre du développement, les composants sont dans un contexte OSGi au niveau de la visibilité des classes importées et exportées. Pour l'exécution, les différents composants souhaités peuvent être associés au conteneur au niveau de son démarrage.

Dans cette annexe, nous allons décrire les différentes étapes à suivre afin de mettre en œuvre deux composants simples avec OSGi et Spring Dynamic Modules, l'un offrant un service retournant des données en dur et un autre les affichant dans une interface Web.

Initialisation de l'environnement

La première étape lors de l'utilisation d'Eclipse afin de réaliser des développements OSGi consiste en la spécification d'une plate-forme d'exécution. En effet, cette dernière permet de spécifier l'ensemble des composants externes à utiliser pour les développements et l'exécution du conteneur Equinox.

Dans ce cadre, les packages des composants spécifiés sont utilisables au niveau des développements et leur visibilité est configurée par l'intermédiaire des en-têtes OSGi présents dans le fichier **MANIFEST.MF**. Les éditeurs de classes Java le supportent en natif puisque la compilation

tient compte de ces éléments et affichent les erreurs relatives si les classes ne peuvent pas être résolues dans les packages visibles.

Plate-forme d'exécution

La configuration de la plate-forme d'exécution se réalise en deux étapes. En effet, par défaut, Eclipse prend en compte tous les composants présents dans le répertoire **plug-ins** de la distribution. Ce dernier en contient un nombre important dont la plupart n'ont aucune utilité dans nos développements, ces composants correspondant à des greffons de l'outil Eclipse lui-même.

Nous allons donc créer une plate-forme d'exécution dédiée à nos développements et contenant tous les composants dont nous aurons besoin. Pour ce faire, nous allons créer un projet Eclipse simple dénommé `plate-forme cible` avec un sous-répertoire nommé `cible`. Nous allons ensuite copier dans ce répertoire tous les fichiers jar des composants que nous souhaitons utiliser. Le tableau B-1 récapitule les différents fichiers jar utilisés dans le cadre de nos besoins et classés par outil.

Tableau B-1 – Liste des composants à copier dans le répertoire cible

Catégorie	Composant
Conteneur Equinox	org.eclipse.osgi (3.3.0)
Traces applicatives (slf4j, commons-logging et log4j)	com.springsource.slf4j.api (1.5.0) com.springsource.slf4j.log4j (1.5.0) com.springsource.slf4j.org.apache.commons.logging (1.5.0) org.springframework.osgi.log4j.osgi (1.2.15.SNAPSHOT)
Framework Spring 2.5.5	org.springframework.bundle.spring.core (2.5.5) org.springframework.bundle.spring.beans (2.5.5) org.springframework.bundle.spring.context (2.5.5) org.springframework.bundle.spring.context.support (2.5.5) org.springframework.bundle.spring.aop (2.5.5) org.springframework.bundle.spring.web (2.5.5) org.springframework.bundle.spring.webmvc (2.5.5)
Framework Spring Dynamic Modules	org.springframework.bundle.osgi.core (1.1.1) org.springframework.bundle.osgi.io (1.1.1) org.springframework.bundle.osgi.extender (1.1.1) org.springframework.bundle.osgi.web (1.1.1) org.springframework.bundle.osgi.web.extender (1.1.1)
Tomcat 5.5.23	org.springframework.osgi.catalina.osgi (5.5.23.SNAPSHOT) org.springframework.osgi.jasper.osgi (5.5.23.SNAPSHOT) org.springframework.osgi.catalina.start.osgi (1.0.0.SNAPSHOT)
Servlets et JSP	com.springsource.javax.servlet.jsp (2.1.0) com.springsource.javax.servlet (2.5.0)
JSTL	com.springsource.javax.servlet.jsp.jstl (1.1.2) com.springsource.javax.el (2.1.0) com.springsource.org.apache.taglibs.standard (1.1.2) org.springframework.osgi.commons-el.osgi (1.0.0.SNAPSHOT)
Dépendances	com.springsource.net.sf.cglib (2.1.3) com.springsource.org.aopalliance (1.0.0) com.springsource.org.apache.commons.digester (1.8.0)

Ce répertoire aurait très bien pu se situer en dehors de l'espace de travail Eclipse mais cette approche permet de voir dans un même espace toutes les ressources utilisées.

Une fois le projet créé et les fichiers copiés dans le sous-répertoire, ce dernier peut être référencé à partir des préférences d'Eclipse dans la rubrique *Développement de greffons – Plate-forme cible*. Une fois cette opération réalisée, la nouvelle liste des composants s'affiche alors dans la fenêtre, comme l'illustre la figure B-1.

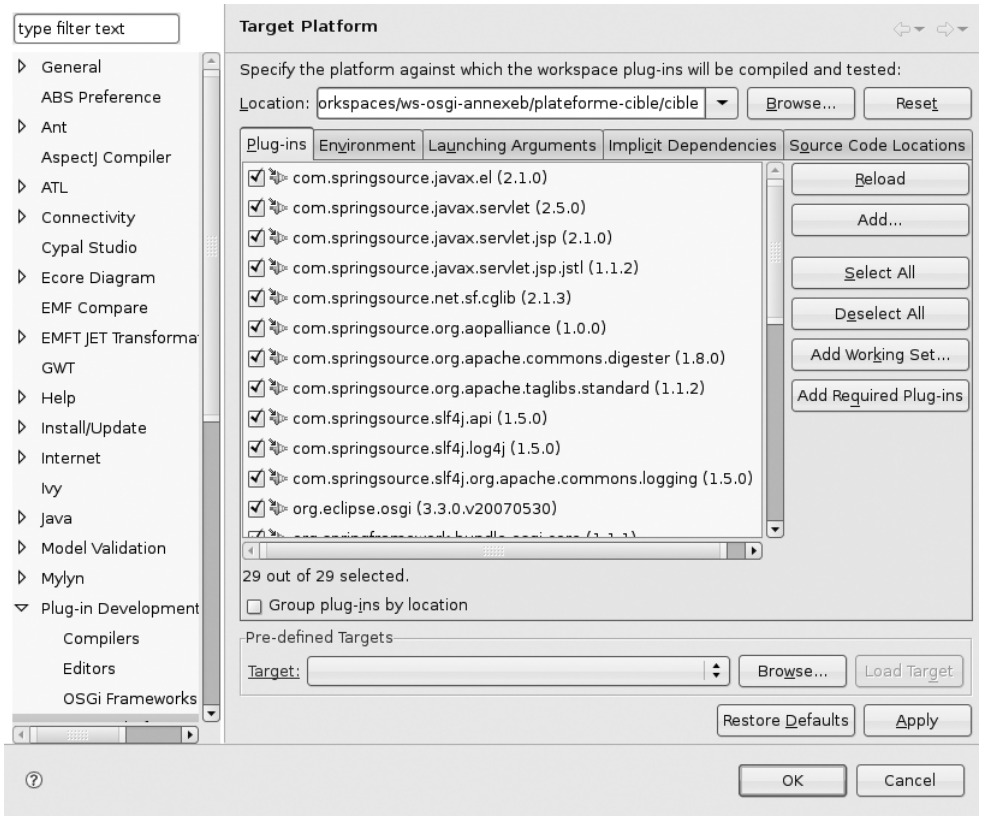


Figure B-1

Fenêtre de configuration de la plate-forme cible

À partir de ce moment, les composants spécifiés sont utilisables aussi bien au niveau du développement de composants ou greffons dans le contexte d'Eclipse que de l'exécution avec le conteneur Equinox.

Il reste néanmoins possible à tout moment de modifier la liste des composants de la plate-forme cible par l'intermédiaire de la précédente fenêtre. Attention de ne pas oublier d'utiliser le bouton « Rafraîchir » afin de mettre à jour la liste des composants en cas d'ajout ou de suppression d'un fichier jar dans le répertoire `cible`.

Développement de composants

Comme le support de la technologie OSGi est intégré en natif dans l'outil de développement Eclipse, toutes les facilités offertes par cet outil afin de développer des greffons sont utilisables dans le développement de composants OSGi.

Cet outillage est désigné dans Eclipse par le terme PDE, abréviation de Plugin Development Environment, et met à disposition des wizards et des éditeurs à cet effet.

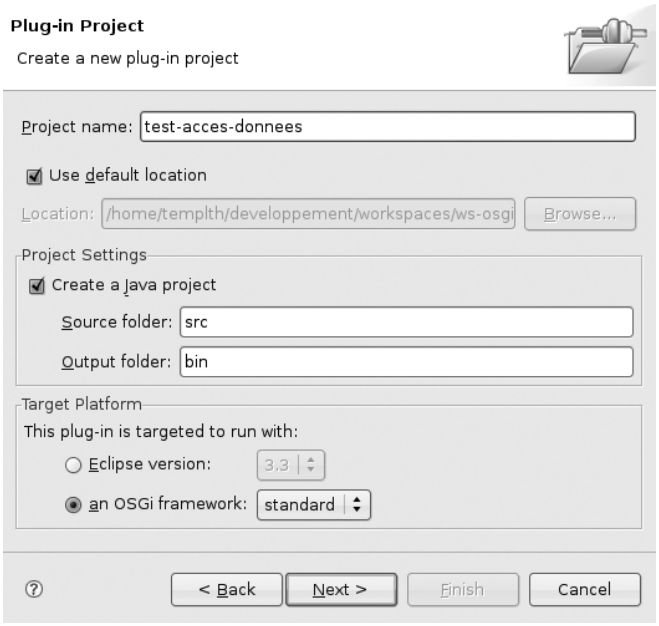
Création de composants

Dans cette annexe, nous allons créer deux composants pour notre application de test. Le premier, se nommant `test-access-donnees`, a en charge de retourner des données en dur par l'intermédiaire d'un service tandis que le second, `test-web`, permet d'afficher ces données en mode Web avec Spring MVC.

La création d'un composant OSGi se réalise dans Eclipse par l'intermédiaire du menu *Nouveau – Autres*. Il suffit par la suite de sélectionner le wizard *Plug-in Project* dans la rubrique *Plug-in Development* de la liste proposée alors. Une fenêtre s'affiche alors afin de saisir les propriétés générales, à savoir le nom du projet relatif ainsi que le type de plate-forme d'exécution. Pour la dernière information, il convient de cocher *standard* dans la zone an *OSGi framework*, comme l'illustre la figure B-2.

Figure B-2

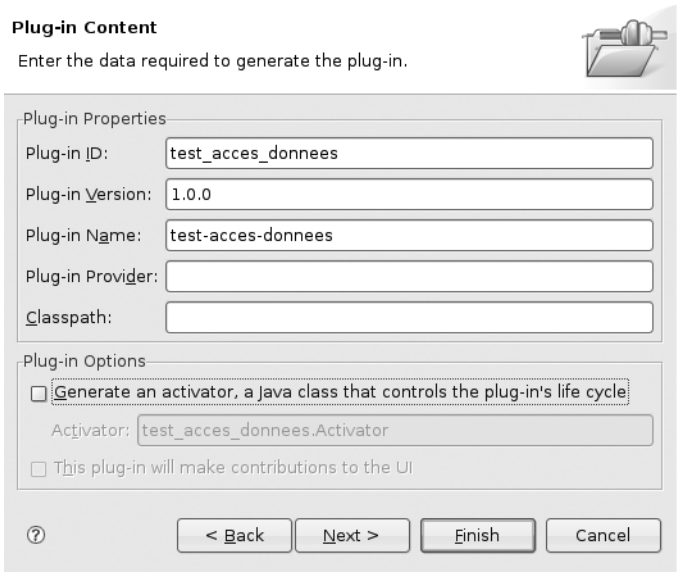
Fenêtre de saisie des propriétés générales du projet OSGi



Une fois ces premières informations saisies, les informations relatives aux propriétés du composant doivent être renseignées. Ces dernières sont utilisées dans la création du fichier **MANIFEST.MF** du composant. Comme nous utilisons Spring Dynamic Modules, il n'est pas nécessaire de générer et d'utiliser une entité d'activation. La figure B-3 illustre les propriétés de cet écran.

Figure B-3

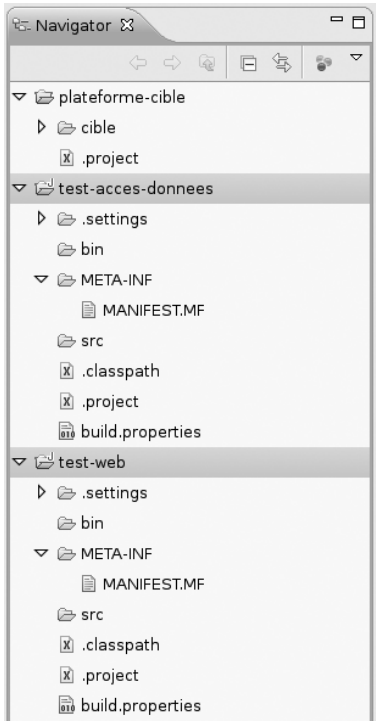
Fenêtre de saisie des propriétés du composant OSGi



Une fois les deux composants créés de cette manière, ils sont visibles dans notre espace de travail, comme l'illustre la figure B-4.

Figure B-4

Espace de travail après la création des composants

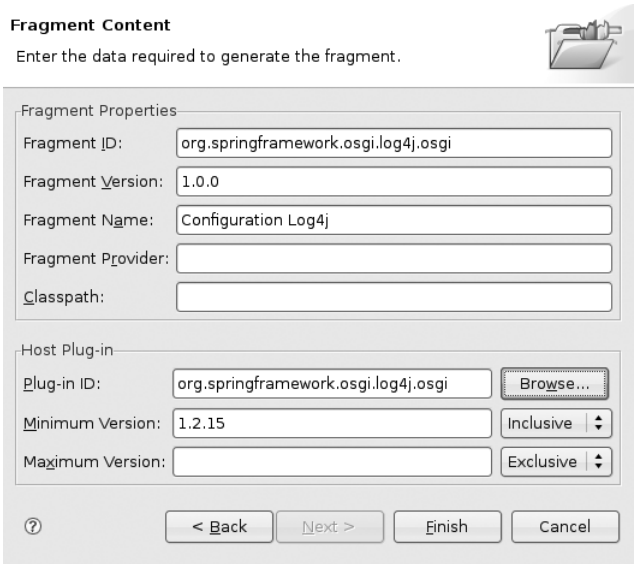


À noter que, pour le composant `test-web`, il convient de positionner `WEB-INF/classes` en tant que destination pour la compilation des classes afin que ces dernières soient visibles par l'application Web par la suite.

Il est à noter qu'un composant de type fragment doit également être créé afin d'avoir accès aux traces applicatives des différents outils. Dans ce cas, la création du fragment se réalise dans Eclipse par l'intermédiaire du menu *Nouveau – Autres* en sélectionnant par la suite le wizard *Fragment Project* dans la rubrique *Plug-in Development* de la liste proposée alors. Une fois le nom renseigné avec la valeur `org.springframework.osgi.log4j.osgi.config` et le champ *Java Project* décoché car il ne contient pas de code Java, il faut référencer le composant OSGi sur lequel le fragment se rapporte, à savoir le composant `org.springframework.osgi.log4j.osgi` dans notre cas, comme l'illustre la figure B-5.

Figure B-5

Sélection du composant référencé lors de la création du fragment



Implémentation de composants

Une fois les composants créés, le développement se réalise sur le même principe que le développement Java, si ce n'est qu'il est nécessaire de réaliser une configuration dans le fichier **MANIFEST.MF** afin de voir des classes et d'en mettre à disposition.

Il est à noter que nous ne décrirons pas ici les différents mécanismes mis en œuvre. Pour plus de précision, veuillez vous reporter au chapitre 15 pour OSGi et Spring Dynamic Modules ainsi qu'au chapitre 6 pour Spring MVC.

Fragment de configuration de Log4j

Ce composant particulier permet de configurer le composant relatif à Log4j afin de spécifier une configuration spécifique à nos besoins par l'intermédiaire d'un fichier **log4j.properties**. Ce fichier doit être créé à la racine du projet du fragment afin d'être pris en compte.

Le contenu suivant peut être utilisé pour ce fichier afin d'afficher les messages d'information de l'outil Spring Dynamic Modules :

```
log4j.rootLogger=INFO, console

log4j.appender.console=org.apache.log4j.ConsoleAppender
log4j.appender.console.layout=org.apache.log4j.PatternLayout
log4j.appender.console.layout.ConversionPattern=%-4r [%t] %-5p %c %x - %m%n

log4j.category.org.springframework.osgi=DEBUG, console
log4j.category.org.springframework.osgi.web=DEBUG, console
```

Avec ce contenu, le démarrage du conteneur Equinox affiche des traces applicatives dans la vue *Console* d'Eclipse.

Composant d'accès aux données

La première étape concernant la mise en œuvre de ce composant consiste en la création de l'interface et de l'implémentation du service. Nous pouvons choisir respectivement les packages `tudu.test.service` et `tudu.test.service.impl` pour les entités `TodoListsManager` et `TodoListsManagerImpl`. Ces créations se réalisent de la même manière que pour un projet Java. Pour notre exemple, nous n'allons spécifier qu'une méthode pour ces deux entités, comme l'illustre le code suivant :

```
public interface TodoListsManager {
    TodoList findTodoList(String listId);
}
```

Le code de l'implémentation correspondante contient des traitements très simples et retourne une instance de type `TodoList` avec des valeurs en dur afin de simplifier notre exemple, comme l'illustre le code suivant :

```
public class TodoListsManagerImpl implements TodoListsManager {
    public TodoList findTodoList(String listId) {
        TodoList liste = new TodoList();
        liste.setListId(listId);
        liste.setName("Test");
        return liste;
    }
}
```

La classe `TodoList` quant à elle se trouve dans le package `tudu.test.domain.model` et contient le code simplifié suivant :

```
public class TodoList {
    private String listId;
    private String name;

    public String getListId() {
        return listId;
    }
```

```
    public String getName() {
        return name;
    }

    public void setListId(String listId) {
        this.listId = listId;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

Une fois ces traitements réalisés, il nous reste à configurer le projet aussi bien au niveau de Spring que d'OSGi.

Tout d'abord, la configuration Spring doit se trouver dans un sous-répertoire **spring** du répertoire **META-INF** du composant. Nous l'appelons dans notre cas **applicationContext.xml** mais seule l'extension `xml` compte pour que Spring Dynamic Modules le prenne en compte par défaut. Ce fichier contient simplement la configuration de la classe `TodoListsManager` (❶) ainsi que l'exportation de cette dernière en tant que service OSGi (❷), comme l'illustre le code suivant :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:osgi="http://www.springframework.org/schema/osgi"
xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd
          http://www.springframework.org/schema/osgi
      http://www.springframework.org/schema/osgi/spring-osgi.xsd">
    <!-- Service -->
    <bean id="todoListsManager"→❶
          class="tudu.test.service.impl.TodoListsManagerImpl"/>
    <!-- Service OSGi -->
    <osgi:service ref="todoListsManager"→❷
            interface="tudu.test.service.TodoListsManager"/>
</beans>
```

La configuration du fichier **MANIFEST.MF** du composant est très simple puisqu'il suffit uniquement de spécifier les informations relatives au composant et exporter les packages `tudu.test.service` et `tudu.test.domain.model` pour rendre accessible les classes relatives au service OSGi mis à disposition. Le code suivant illustre le contenu de ce fichier :

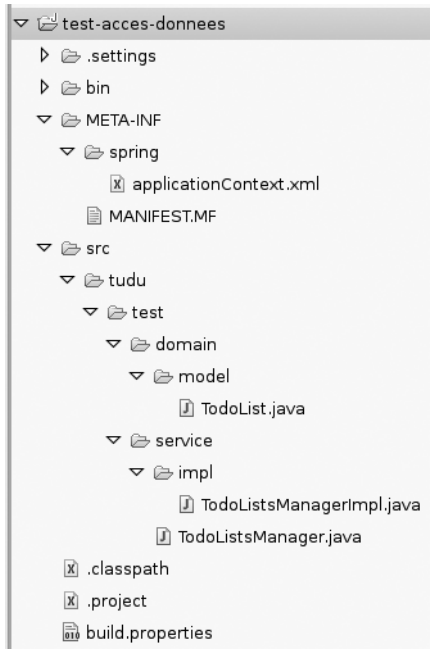
```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: test-acces-donnees
Bundle-SymbolicName: test_acces_donnees
Bundle-Version: 1.0.0
Export-Package: tudu.test.domain.model,
 tudu.test.service
```

Il est à noter qu'un éditeur Eclipse spécifique est mis à disposition afin d'éditer les propriétés des composants OSGi. Cet éditeur enregistre ces informations dans le fichier **MANIFEST.MF**.

La figure B-6 récapitule la structure du projet relatif au composant ainsi que les différents éléments qu'il contient.

Figure B-6

Structure et contenu du composant test-acces-donnees



Composant Web

La mise en œuvre des traitements dans ce composant se réalise de manière similaire à celui décrit dans la section précédente avec, en supplément, des traitements relatifs à la partie Web.

Tout d'abord, puisque nous désirons utiliser le service exporté précédemment, il convient d'importer les packages relatifs (❶) en plus de ceux relatifs aux API Web, à Spring et Spring Dynamic Modules dans le fichier **MANIFEST.MF**. Les composants relatifs à JSTL doivent également être spécifiés en tant que composants nécessaires (❷). Le code suivant illustre le contenu de ce fichier :

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: test-web
Bundle-SymbolicName: test_web
Bundle-Version: 1.0.0
Bundle-ClassPath: WEB-INF/classes
Import-Package: javax.servlet;version="2.5.0",→❶
 javax.servlet.http;version="2.5.0",
 javax.servlet.jsp;version="2.0.0",
```

```
 org.springframework.beans.factory;version="2.5.5",
 org.springframework.context;version="2.5.5",
 org.springframework.web.bind;version="2.5.5",
 org.springframework.web.context;version="2.5.5",
 org.springframework.web.context.support;version="2.5.5",
 org.springframework.web.servlet;version="2.5.5",
 org.springframework.web.servlet.handler;version="2.5.5",
 org.springframework.web.servlet.mvc;version="2.5.5",
 org.springframework.web.servlet.view;version="2.5.5",
 tudu.test.domain.model,
 tudu.test.service
Require-Bundle: com.springsource.javax.servlet.jsp.jstl,→❷
 com.springsource.org.apache.taglibs.standard
```

De plus, puisque le composant est dédié à une application Web, le répertoire **WEB-INF** doit être créé à la racine du composant. Un sous-répertoire peut être également ajouté afin de contenir notre page JSP. Il est à noter qu'un sous-répertoire `classes` doit être également créé afin de recevoir le résultat de la compilation afin que les classes soient visibles par l'application Web par la suite.

La configuration de l'application Web se réalise dans le fichier **web.xml**, fichier localisé dans le répertoire **WEB-INF**, et dont le contenu est décrit par le code suivant :

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app (...)>
    <display-name>Tudu Web Test</display-name>

    <servlet>
        <servlet-name>tudu-test</servlet-name>
        <servlet-class>
            org.springframework.web.servlet.DispatcherServlet
        </servlet-class>
        <load-on-startup>2</load-on-startup>
        <init-param>
            <param-name>contextClass</param-name>
            <param-value>
                org.springframework.osgi.web.context.support.➥
                            OsgiBundleXmlWebApplicationContext
            </param-value>
        </init-param>
    </servlet>

    <servlet-mapping>
        <servlet-name>tudu-test</servlet-name>
        <url-pattern>*.do</url-pattern>
    </servlet-mapping>
</web-app>
```

Une classe implémentant un contrôleur Spring MVC simple doit être ensuite créée dans le package `tudu.test.web`, cette classe interrogeant simplement le service et mettant à disposition le résultat pour la page JSP, comme l'illustre le code suivant :

```
public class TodoListsControleur extends AbstractController {
    private TodoListsManager todoListsManager;
    protected ModelAndView handleRequestInternal(
                HttpServletRequest request,
                HttpServletResponse response) throws Exception {

        TodoList liste = todoListsManager.findTodoList("test");
        return new ModelAndView("todolist", "liste", liste);
    }

    public void setTodoListsManager(
                    TodoListsManager todoListsManager) {
        this.todoListsManager = todoListsManager;
    }
}
```

Une page JSP simple dénommée **todolist.jsp** est ensuite créée dans le répertoire **WEB-INF/jsp** du composant afin d'afficher les informations contenues dans l'objet passé par le contrôleur précédent :

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ page language="java"
    contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
➥"http://www.w3.org/TR/html4/loose.dtd">

<html>
<body>
    <u>Todolist</u>: <c:out value="${liste.name}"/>
    (<c:out value="${liste.listId}"/>)
</body>
</html>
```

Une fois ces deux entités créées, il convient de les configurer dans Spring par l'intermédiaire du fichier dénommé **tudu-test-servlet.xml**, localisé directement sous le répertoire **WEB-INF** et dont le contenu est décrit ci-dessous :

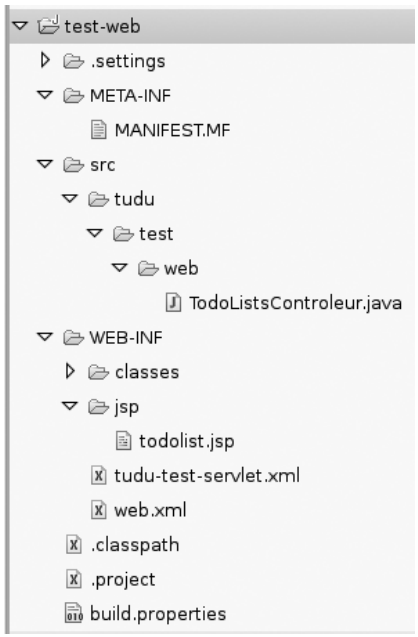
```
<beans (...)>

    <bean id="urlMapping"
          class="org.springframework.web.servlet.handler.➥
                                    SimpleUrlHandlerMapping">
        <property name="mappings">
            <props>
                <prop key="/todolist.do">todolistController</prop>
            </props>
        </property>
    </bean>

    <bean id="todolistController"
          class="fr.argia.osgi.web.TodolistController"
          lazy-init="false">
```

```
        <property name="todoListsManager">
            <osgi:reference
                interface="tudu.test.service.TodoListsManager"
                timeout="5000"/>
        </property>
    </bean>

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

</beans>
```

La figure B-7 récapitule la structure du projet relatif au composant ainsi que les différents éléments qu'il contient.

Figure B-7

Structure et contenu du composant test-web



Exécution dans Equinox

Le dernier aspect du support OSGi d'Eclipse consiste en la possibilité de lancer en interne à l'outil un conteneur de ce type, par défaut, le conteneur Equinox étant préconfiguré. Ce conteneur se fonde automatiquement sur les composants présents dans la plate-forme cible ainsi que sur les composants présents dans l'espace de travail, chaque composant pouvant être éventuellement désactivé dans la configuration du conteneur.

Voyons maintenant comment créer et paramétrer un lanceur dédié à Equinox, comment interagir avec le conteneur créé et enfin les techniques permettant de résoudre des erreurs liées aux dépendances des composants.

Lancement du conteneur

Afin de démarrer un conteneur Equinox dans Eclipse, la fenêtre de configuration des lanceurs doit être utilisée. Elle est accessible par l'intermédiaire du menu *Run – Open Run Dialog* ou de la barre d'outils et permet de configurer différents types de lanceur. Le type OSGi Framework est dédié au lancement de conteneurs OSGi.

Un lanceur peut être créé par l'intermédiaire de l'élément *New* du menu contextuel et permet de configurer les différents paramètres de lancement ainsi que les différents composants à ajouter à son démarrage. Les différents composants de la plate-forme cible ainsi que les composants de l'espace de travail sont éligibles et cochés par défaut. Le conteneur les installera et tentera de les démarrer lors de son lancement.

La figure B-8 illustre la fenêtre de configuration d'un lanceur Equinox dénommé *Conteneur OSGi Test* dans Eclipse.

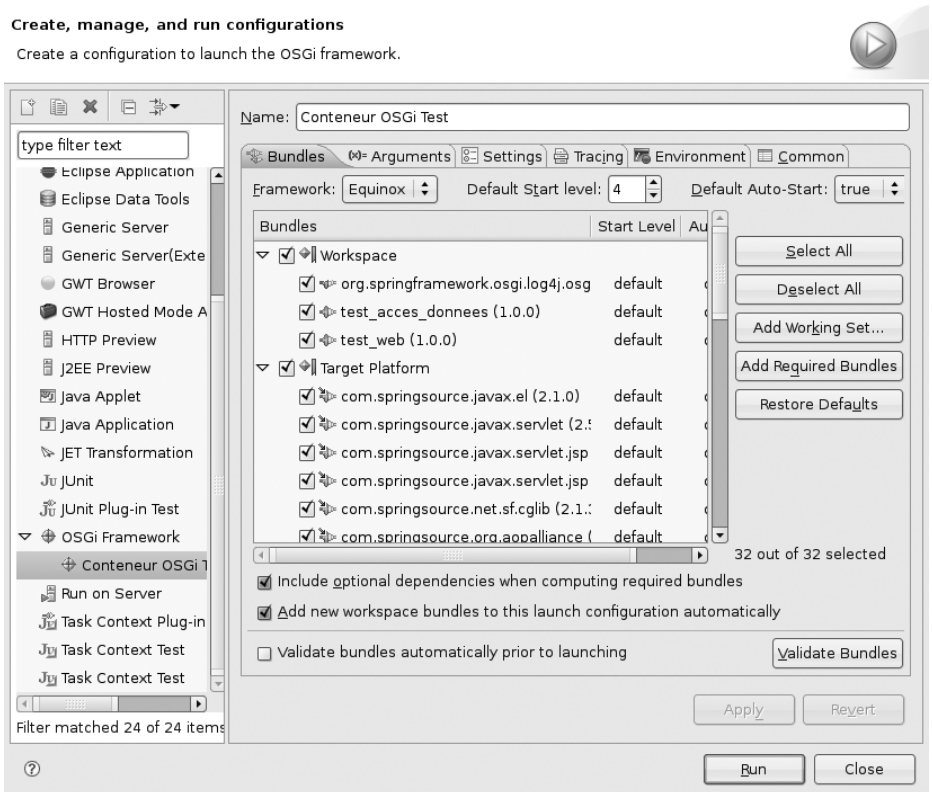


Figure B-8

Fenêtre de configuration d'un lanceur Equinox

En cliquant sur le bouton *Run*, le conteneur se lance et les messages s'affichent dans la vue *Console*. Un outil interactif en ligne de commande est également accessible et utilisable à partir de cette vue.

De plus, il arrive que certains composants doivent être démarrés avant d'autres dans le conteneur. À cet effet, est utilisable la fonctionnalité relative aux niveaux de démarrage.

Dans notre cas, le composant relatif à l'extender web de Spring Dynamic Modules doit être démarré en dernier car il se fonde sur le composant de Tomcat. Pour ce faire, un niveau de démarrage supérieur à celui par défaut, quatre dans notre cas, doit être spécifié. Pour ce composant, un niveau de cinq peut être choisi. Dans la fenêtre de configuration du lanceur, les niveaux de démarrage se paramètrent au niveau de la seconde colonne dans la ligne des composants.

Dans le cas d'un démarrage sans erreur du conteneur Equinox avec les deux composants de test, les traces suivantes sont visibles dans la vue *Console* d'Eclipse :

```
osgi> 0    [Tomcat Catalina Start Thread] INFO
org.springframework.osgi.web.tomcat.internal.Activator  - Starting
Apache Tomcat/5.5.23 ...
(...)
562  [Tomcat Catalina Start Thread] INFO
org.apache.coyote.http11.Http11BaseProtocol  - Starting Coyote
HTTP/1.1 on http-8080
569  [Tomcat Catalina Start Thread] INFO
org.springframework.osgi.web.tomcat.internal.Activator  -
Succesfully started Apache Tomcat/5.5.23 @ Catalina:8080
582  [Start Level Event Dispatcher] INFO
org.springframework.osgi.web.deployer.tomcat.TomcatWarDeployer  -
Found service Catalina
(...)
1168 [Start Level Event Dispatcher] INFO
org.springframework.osgi.web.extender.internal.activator.WarLoaderListener  -
test-web (test_web) is a WAR,scheduling war deployment on context path [/tudu]
(web.xml found at [bundleentry://35/WEB-INF/web.xml])
(...)
2450 [Timer-0] INFO
org.apache.catalina.core.ContainerBase.[Catalina].[localhost].[/tudu]  -
Initializing Spring FrameworkServlet 'tudu-test'
2450 [Timer-0] INFO
org.springframework.web.servlet.DispatcherServlet  -
FrameworkServlet 'tudu-test': initialization started
2625 [Timer-0] INFO
org.springframework.osgi.web.context.support.OsgiBundleXmlWebApplicationContext
- Publishing application context as OSGi service with properties
{org.springframework.context.service.name=test_web, Bundle-SymbolicName=test_web,
Bundle-Version=1.0.0,org.springframework.web.context.namespace=tudu-test-servlet}
2627 [Timer-0] INFO
org.springframework.web.servlet.DispatcherServlet  - FrameworkServlet
'tudu-test': initialization completed in 177 ms
```

```
2679 [Timer-0] INFO
org.springframework.osgi.web.deployer.tomcat.TomcatWarDeployer  - Successfully
deployed bundle [test-web (test_web)] at [/tudu] on server
org.apache.catalina.core.StandardService/1.0
```

Interroger le conteneur

Comme nous l'avons évoqué précédemment, une fois le conteneur Equinox lancé, le processus met à disposition de base un outil en ligne de commande permettant d'interagir avec le conteneur aussi bien pour accéder à des informations relatives aux composants qu'il contient que pour changer leurs états.

Cet outil offre également la possibilité d'avoir accès à une commande de diagnostics des composants très utile pour détecter d'éventuelles erreurs ou avertissements relatifs à la résolution des erreurs de dépendances.

Le tableau B-2 liste les principales commandes de cet outil regroupées en catégorie.

Tableau B-2 – Principales commandes du conteneur Equinox

Catégorie de commandes	Commandes
Contrôle du conteneur	`launch` : lancement du conteneur ; `shutdown` : arrt du conteneur ; `close` : arrt du conteneur et finalisation du processus ; `init` : désinstallation de tous les composants présents.
Contrôle des composants	`install` : installe un composant en se fondant sur son fichier jar. `uninstall` : désinstalle un composant; `start` : démarre un composant ; `stop` : arrête un composant; `refresh` : rafraîchie un composant ; `update` : met à jour un composant.
Affichage d'informations	`status` : affiche le statut global du conteneur en affichant la liste des composants ainsi que les services présents ; `ss` : affiche la liste des composants avec leurs états respectifs; `services` : affiche la liste des services avec les services les ayant enregistrés et les utilisant ; `packages` : affiche la liste des packages avec les composants les mettant à disposition et les utilisant; `bundles` : affiche la liste des composants avec leurs informations respectives `bundle` : affiche les informations détaillées (identifiant, statut, services et packages) d'un composant `headers` : affiche les en-têtes OSGi présents dans le fichier **MANIFEST.MF** pour un composant précis.
Diagnostics relatifs à l'exécution du conteneur	`diag` : affichage des contraintes non satisfaites pour un composant

Dans le tableau ci-dessus, la commande `ss` correspond à la plus importante car elle permet d'avoir une vue globale rapidement du contenu d'Equinox, aussi bien au niveau des composants présents que de leurs états respectifs. Il est alors possible par la suite avec les autres commandes de modifier leurs états ou d'afficher des informations plus précises sur ces derniers.

La figure B-9 illustre l'utilisation des commandes `ss`, `bundle` et `headers` afin d'avoir accès aux informations des composants présents dans le conteneur.

```
osgi> ss

Framework is launched.

id      State       Bundle
0       ACTIVE      org.eclipse.osgi_3.3.0.v20070530
1       ACTIVE      org.springframework.osgi.catalina.start.osgi_1.0.0.SNAPSHOT
2       ACTIVE      org.springframework.bundle.spring.beans_2.5.5
3       ACTIVE      org.springframework.bundle.osgi.web.extender_1.1.1

30      ACTIVE      org.springframework.bundle.spring.context.support_2.5.5
31      ACTIVE      org.springframework.osgi.log4j.osgi_1.2.15.SNAPSHOT
                    Fragments=27

osgi> bundle 20
initial@reference:file:../../test-web/ [20]
  Id=20, Status=ACTIVE      Data Root=/home/templth/developpement/workspaces/ws-
  No registered services.
  No services in use.
  No exported packages
  No imported packages
  No fragment bundles
  Named class space
    test_web; bundle-version="1.0.0"[provided]
  No required bundles

osgi> headers 20
Bundle headers:
 Bundle-ManifestVersion = 2
 Bundle-Name = test-web
 Bundle-SymbolicName = test_web
 Bundle-Version = 1.0.0
 Manifest-Version = 1.0
```

Figure B-9

Utilisation des commandes ss, bundle et headers d'Equinox

La figure B-10 illustre quant à elle l'utilisation de la commande `services` permettant d'afficher la liste de tous les services enregistrés dans le conteneur. Dans cette figure, nous pouvons remarquer la présence du service correspondant au composant d'accès aux données ainsi que les services relatifs aux contextes applicatifs Spring des composants éligibles par Spring Dynamic Modules.

```
osgi> services
{org.eclipse.osgi.service.runnable.StartupMonitor}={service.ranking=-2147483648, service.id=1}
  Registered by bundle: System Bundle [0]
  No bundles using service.
{org.osgi.service.packageadmin.PackageAdmin}={service.ranking=2147483647, service.pid=0.org.eclipse.osgi.framework.internal.co
  Registered by bundle: System Bundle [0]
  Bundles using service:
    System Bundle [0]
    initial@reference:file:spring-osgi-extender-1.1.1.jar/ [15]
    initial@reference:file:../../test-web/ [35]


    .
{org.apache.catalina.core.StandardService, org.apache.catalina.Service, javax.management.MBeanRegistration, org.apache.catalin
  Registered by bundle: initial@reference:file:catalina.start.osgi-1.0-SNAPSHOT.jar/ [1]
  Bundles using service:
    initial@reference:file:spring-osgi-web-extender-1.1.1.jar/ [3]
{org.springframework.beans.factory.xml.NamespaceHandlerResolver}={service.id=23}
  Registered by bundle: initial@reference:file:spring-osgi-extender-1.1.1.jar/ [15]
  Bundles using service:
    initial@reference:file:../../test-acces-donnees/ [34]
    initial@reference:file:../../test-web/ [35]
{org.xml.sax.EntityResolver}={service.id=24}
  Registered by bundle: initial@reference:file:spring-osgi-extender-1.1.1.jar/ [15]
  Bundles using service:
    initial@reference:file:../../test-acces-donnees/ [34]
    initial@reference:file:../../test-web/ [35]
{tudu.test.service.TodoListsManager}={org.springframework.osgi.bean.name=todoListsManager, Bundle-SymbolicName=test_acces_donn
  Registered by bundle: initial@reference:file:../../test-acces-donnees/ [34]
  Bundles using service:
    initial@reference:file:../../test-web/ [35]
{org.springframework.osgi.context.DelegatedExecutionOsgiBundleApplicationContext, org.springframework.osgi.context.Configurabl
  Registered by bundle: initial@reference:file:../../test-acces-donnees/ [34]
  No bundles using service.
{org.springframework.web.context.ConfigurableWebApplicationContext, org.springframework.ui.context.ThemeSource, org.springfram
  Registered by bundle: initial@reference:file:../../test-web/ [35]
  No bundles using service.
```

Figure B-10

Utilisation de la commande services d'Equinox

Techniques de résolution des erreurs

Nous avons vu dans la précédente section que le conteneur Equinox offrait un outil en ligne de commandes afin de l'interroger. Les différentes commandes mises à disposition offrent un intéressant moyen de récupérer diverses informations afin de déterminer la cause d'erreurs, notamment au niveau de la résolution des dépendances.

Prenons un exemple. Nous allons reprendre l'ensemble des composants présents dans le tableau B-1 mais sans y intégrer le composant `com.springsource.slf4j.org.apache.commons.logging` (1.5.0). Cette omission a pour conséquence d'introduire des erreurs au niveau de la résolution des dépendances.

Une fois le conteneur Equinox lancé comme précédemment mais en décochant le composant précédemment cité, nous remarquons qu'aucune trace relative à l'exécution n'est affichée.

Exécutons la commande `ss` afin de visualiser l'état dans lequel se trouvent les composants. La figure B-11 illustre le résultat de cette commande.

```
osgi> ss

Framework is launched.

id      State       Bundle
0       ACTIVE      org.eclipse.osgi_3.3.0.v20070530
1       RESOLVED    org.springframework.osgi.catalina.start.osgi_1.0.0.SNAPSHOT
2       INSTALLED   org.springframework.bundle.spring.beans_2.5.5
3       INSTALLED   org.springframework.bundle.osgi.web.extender_1.1.1
4       INSTALLED   org.springframework.bundle.osgi.core_1.1.1
5       INSTALLED   org.springframework.bundle.osgi.io_1.1.1
6       ACTIVE      com.springsource.org.aopalliance_1.0.0
7       INSTALLED   org.springframework.bundle.spring.context_2.5.5
8       INSTALLED   org.springframework.bundle.spring.core_2.5.5
9       ACTIVE      org.springframework.osgi.commons-el.osgi_1.0.0.SNAPSHOT
10      INSTALLED   org.springframework.bundle.spring.webmvc_2.5.5
11      ACTIVE      com.springsource.slf4j.api_1.5.0
12      ACTIVE      test_acces_donnees_1.0.0
13      ACTIVE      org.springframework.osgi.catalina.osgi_5.5.23.SNAPSHOT
14      ACTIVE      com.springsource.net.sf.cglib_2.1.3
15      INSTALLED   org.springframework.bundle.osgi.extender_1.1.1
16      ACTIVE      com.springsource.javax.servlet.jsp.jstl_1.1.2
17      ACTIVE      com.springsource.org.apache.taglibs.standard_1.1.2
18      INSTALLED   org.springframework.bundle.spring.web_2.5.5
20      ACTIVE      test_web_1.0.0
21      ACTIVE      com.springsource.javax.servlet_2.5.0
22      ACTIVE      org.springframework.osgi.jasper.osgi_5.5.23.SNAPSHOT
23      ACTIVE      com.springsource.slf4j.log4j_1.5.0
24      INSTALLED   com.springsource.org.apache.commons.digester_1.8.0
25      INSTALLED   org.springframework.bundle.osgi.web_1.1.1
26      ACTIVE      com.springsource.javax.el_2.1.0
28      ACTIVE      com.springsource.javax.servlet.jsp_2.1.0
29      INSTALLED   org.springframework.bundle.spring.aop_2.5.5
30      INSTALLED   org.springframework.bundle.spring.context.support_2.5.5
31      ACTIVE      org.springframework.osgi.log4j.osgi_1.2.15.SNAPSHOT
                    Fragments=32
32      RESOLVED    org.springframework.osgi.log4j.osgi.config_1.0.0
                    Master=31
```

Figure B-11

Résultat du lancement de la commande ss avec des erreurs de résolution

Nous remarquons qu'un large ensemble de composants sont dans un état non plus actif, mais installé. Cela signifie qu'ils sont dans l'état précédant la résolution des dépendances et nous pouvons donc raisonnablement considérer qu'un problème se situe au niveau de cette résolution.

Nous remarquons également que la plupart des composants relatifs à Spring et Spring Dynamic Modules sont dans ce cas. Essayons de démarrer, par exemple, le composant `org.springframework.bundle.spring.core` (2.5.5) en se fondant sur la commande `start`. La figure B-12 illustre le résultat de cette commande.

```
osgi> start 2
org.osgi.framework.BundleException: The bundle could not be resolved. Reason: Missing Constraint: Import-Package: org.apache.commons.logging;
        at org.eclipse.osgi.framework.internal.core.BundleHost.startWorker(BundleHost.java:305)
        at org.eclipse.osgi.framework.internal.core.AbstractBundle.start(AbstractBundle.java:260)
        at org.eclipse.osgi.framework.internal.core.AbstractBundle.start(AbstractBundle.java:252)
        at org.eclipse.osgi.framework.internal.core.FrameworkCommandProvider._start(FrameworkCommandProvider.java:260)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
        at java.lang.reflect.Method.invoke(Method.java:597)
        at org.eclipse.osgi.framework.internal.core.FrameworkCommandInterpreter.execute(FrameworkCommandInterpreter.java:145)
        at org.eclipse.osgi.framework.internal.core.FrameworkConsole.docommand(FrameworkConsole.java:291)
        at org.eclipse.osgi.framework.internal.core.FrameworkConsole.console(FrameworkConsole.java:276)
        at org.eclipse.osgi.framework.internal.core.FrameworkConsole.run(FrameworkConsole.java:218)
        at java.lang.Thread.run(Thread.java:619)
```

Figure B-12

Résultat du lancement de la commande start pour le composant Spring Core

L'erreur soulevée par le démarrage du composant fait apparaître que le package `org.apache.commons.logging` n'est pas présent dans le conteneur. En utilisant à nouveau la commande `ss`, nous remarquons que le composant relatif, le composant `com.springsource.slf4j.org.apache.commons.logging` (1.5.0), n'est pas présent dans le conteneur. L'ajout du composant permet alors de résoudre le problème.