Guide de Migration

Intégration de Spring

Version x.y du 28/03/2008

Etat : xxx

SUIVI DES MODIFICATIONS

Version	Rédaction	Description	Vérification	Date
1.0	G.PICAVET C.ROCHETEAU K.COIFFET	Première version		28/03/08
		Document validé dans sa version xxx		

LISTE DE DIFFUSION

Organisation	Nom	Info	Commentaire	Validation
		☐	☐	☐
		☐	☐	☐
		☐	☐	☐

SOMMAIRE

DOCUMENTS DE REFERENCE

Version	Titre

1 MOTIVATIONS

- Fournir une procédure d'intégration de Spring dans un projet Acube.

  L'intérêt de Spring est de fournir un socle serveur robuste, notamment pour :
  - ➔ centraliser la configuration de la partie serveur d'une application dans un fichier xml normalisé et assembler les objets à partir d'une fabrique unique.
  - ➔ réduire l'adhérence vers le socle technique afin de réutiliser le code dans un environnement différent (tests unitaire)
  - ➔ faciliter l'intégration d'autres librairies (Struts, Spring security, Spring batch, IBatis, ...)
- Préciser les points de contrôle pour vérifier la compatibilité ascendante.

2 COMPATIBILITE

➔ compatibilité avec la couche dao existante et JDBCWrapper.

3 PROCEDURES DE MIGRATION

3.1 REMPLACEMENT DU PATTERN DELEGATE/DAOFACTORY

Le but est de remplacer les classes delegate par des classes de service (au sens SOA) réutilisables. La configuration de l'objet DAO à injecter dans le service ne se fait plus dans dao.properties mais avec Spring.

1. Vérifier la compilation et le bon fonctionnement de l'application à migrer
2. Ajouter les jars suivants de l'application à migrer :
   - spring.jar
   - cglib-nodep-2.1_3.jar

3. Créer un fichier appContext.xml dans Serveur/src :

```
<?xml version="1.0" encoding="UTF-8"?>

<!--
Configuration Spring pour la couche métier.
-->
<beans xmlns="http://www.springframework.org/schema/beans"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xmlns:tx="http://www.springframework.org/schema/tx"
            xsi:schemaLocation="
              http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
              http://www.springframework.org/schema/jee
http://www.springframework.org/schema/jee/spring-jee-2.5.xsd
              http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-2.5.xsd"
    >

        <!-- ========================= couche BUSINESS ========================= -


        <!-- ========================= couche DAO ========================= -->

</beans>
```

4. Dans WEB-INF/web.xml, ajouter les lignes suivantes :

```
<!--  Indique au ContextLoaderListener où sont les fichiers de configuration
Spring -->
<context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>
                classpath*:appContext.xml
```

```
        </param-value>
</context-param>
```

```
<listener>
        <listener-class>
                org.springframework.web.context.ContextLoaderListener
        </listener-class>
</listener>
```

5. Vérifier le démarrage de l'application et le chargement de Spring.

6. dans le fichier appContext.xml, en dessous du commentaire « couche DAO », définir un bean pour chaque clé de DAO définie dans le fichier dao.properties. Exemple :

Si dans dao.properties on a :

```
dao.delegate.Example = \
acube.projet.integration.ExampleDAOJdbc
```

Créer un bean dans appContext.xml :

```
<!-- ========================= couche DAO ========================= -->

        <bean id="exampleDAO" class="acube.projet.integration.ExampleDAOJdbc">
        </bean>
```

<u>Remarque</u> : pour créer un mode « test », il suffira de créer un fichier « appContext-test.xml » référençant les dao de test au lieu des dao jdbc. Ce fichier sera normalement chargé par les classes de test unitaire plutôt que le contexte.

7. Pour chaque classe ayant un stéréotype Delegate :
   ➔ remplacer le code suivant :

```
public class ExampleDelegate {

    /** <code>fichierConfigDAO</code> fichier de configuration des DAO */
    private static Configuration fichierConfigDAO;

    /**
     * <code>daoCle</code>: nom de la cle dans le fichier de properties
     * donnat les classes dao a utiliser
     */
    private static String DAO_CLE = "dao.delegate.Example";

    /** <code>dao</code> dao */
    private ExampleDAO dao = null;

    /**
     * Constructeur
     *
```

```
     * Auto_Generated
     *
     * @throws TechnicalException
     *             en cas de probleme technique
     */
    public ExampleDelegate (ExampleDAO dao) throws TechnicalException {

        this.dao = dao;
        // Récupération du type de DAO à mettre en oeuvre
        try {
            fichierConfigDAO = ConfigurationManager.getInstance("dao");

        } catch (ConfigurationException maere) {

            LabelManager paramLib = new LabelManager();
            paramLib.addParameter("0", "dao.properties");
            logger.error(THIS_CLASS + Label.getLabel("ERR-TEC-01", paramLib));
            throw new TechnicalException(new ErrorVO("ERR-TEC-01", Label
                    .getLabel("ERR-TEC-01", paramLib), maere.getMessage(),
                    true, false, false));
        }

        String dao = Tools.obtenirValeur(fichierConfigDAO, "dao.properties",
                DAO_CLE);

        // Recuperation du DAO
        this.dao = getDAO(dao);
    }

    /**
     * Méthode permettant de créer un DAO en utilisant sa Factory.
     *
     * Auto_Generated
     *
     * @param name :
     *            nom du DAO.
     * @return : un objet de type com.eds.acube.db.schema.Table@131dddcDAO.
     * @throws TechnicalException
     *             en cas de probleme de recuperation du dao
     */
    public ExampleDAO getDAO(String name) throws TechnicalException {

        DAOFactory daoFactory = new DAOFactoryImpl();

        ExampleDAO dao = (ExampleDAO) daoFactory.createDAO(name);
        return dao;
    }
}
```

Par celui-ci :

```
public class ExampleDelegate {

    /** <code>dao</code> dao */
    private ExampleDAO dao = null;

    /**
     * Constructeur
     *
     * Auto_Generated
     *
     */
    public ExampleDelegate (ExampleDAO dao) {
        this.dao = dao;
    }
```

➔ A l'aide d'Eclipse (via refactor/rename), changer le suffixe Delegate en Service

➔ Créer un bean dans appContext.xml :

```
<!-- ========================= couche Business ========================= -->

        <bean id="exampleService" class="acube.projet.business.ExampleService">
                <constructor-arg ref="exampleDAO"/>
        </bean>
```

8. **Intégration aux classes d'action.**
   Les classes d'action référençant le service (anciennement le delegate) ne compilent plus, puisque le constructeur par défaut n'existe plus. Il faut maintenant déléguer l'instanciation du service et la résolution du dao à Spring.

➔ Première méthode : appel direct à la factory de Spring

- Cas d'une action Struts 1 :

```
WebApplicationContext webContext =
        WebApplicationContextUtils
                .getWebApplicationContext(getServlet().getServletContext());

ExampleService delegate =
        (ExampleService) webContext.getBean("exampleService");
```

- Cas d'une action Struts 2 :

```
WebApplicationContext webContext =
        WebApplicationContextUtils
                .getWebApplicationContext(ServletActionContext
```

```
                                .getServletContext());

ExampleService delegate =
     (ExampleService) webContext.getBean("exampleService ");
```

Remarque :il est recommandé de factoriser ce code dans une superclasse Action du projet, afin de réduire la dépendance vers Spring à cette seule classe.

➔ Deuxième méthode : injection automatique de l'objet service dans la classe action. Nécessite le plugin d'intégration de Struts (1 ou 2), qui permet à Struts de déléguer l'instanciation des classes Action à Spring.

- Cas de Struts 1 :

Voir : http://static.springframework.org/spring/docs/2.5.x/reference/webintegration.html#struts

- Cas de Struts 2 :

Voir : http://struts.apache.org/2.x/docs/spring-plugin.html

3.2 GESTION DES DATASOURCES ET TRANSACTIONS (FACULTATIF)

Une version modifiée du JDBCWrapper permet d'utiliser une datasource définie dans le fichier appContext.xml (et non plus dans server.properties). Avec ce système, la gestion des transactions devient également possible au travers de Spring.

Intérêts :

-remplacement de la classe TransactionManager et des intrications avec le JdbcWrapper ,

-la démarcation des transactions se fait plus simplement au niveau de la couche service (via les annotations ou en xml),

- pas d'exception JDBCWrapper au niveau de la couche service (encapsulation),

-les transactions imbriquées sont gérées,

-possibilité de datasources multiples via JTA,…

1. Ajouter la configuration suivante dans appContext.xml

```
        <!-- ========================= ressources ========================= -->

        <!-- Datasource standalone, sans pool de connexion, afin de fonctionner
avec un hsqlDb en mode In-Process -->
        <bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource" destroy-
method="close">
                <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
```

```
                <property name="url"
value="jdbc:mysql://130.177.222.253:3306/GESTION_CONTACTS"/>
                <property name="username" value="contacts_dbo"/>
                <property name="password" value="contacts_dbo"/>
        </bean>

        <!-- Transaction manager pour une unique DataSource JDBC -->
        <bean id="txManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
                <property name="dataSource" ref="dataSource"/>
        </bean>

        <!--
                Instruct Spring to perform declarative transaction management
                automatically on annotated classes.
        -->
        <tx:annotation-driven transaction-manager="txManager"/>

        <bean id="jdbcWrapper" class="acube.projet.technical.JDBCWrapper">
                <constructor-arg ref="dataSource"/>
        </bean>
```

2. Pour chaque classe DAOJdbc :

   - remplacer le constructeur par défaut par un constructeur avec jdbcwrapper :

```
    private JDBCWrapper jdbcWrapper;

    /**
     * Constructeur
     *
     * Auto_Generated
     *
     * @throws TechnicalException
     *             en cas de probleme technique
     */
    public ExampleDAOJdbc(JDBCWrapper jdbcWrapper) throws TechnicalException {
                this. jdbcWrapper = jdbcWrapper
```

   - dans appContext.xml, modifier le bean dao de manière à injecter le jdbcwrapper :

```
        <bean id="exampleDAO" class="acube.projet.integration.ExampleDAOJdbc">
                <constructor-arg ref="jdbcWrapper"/>
        </bean>
```

3. Supprimer les références au TransactionManager (et probablement le try,catch) et utiliser l'annotation @Transactional autour des méthodes ou classes nécessitant l'utilisation d'une transaction.

Exemple dans une Action :

```
    @Transactional
    protected ArrayList getBeansActions(HttpServletRequest request)
            throws DAOException, FunctionalException, TechnicalException,
            JDBCWrapperException {

                …
                appelService1();
                appelService2();
        }
```

3.3 GESTION DES TESTS UNITAIRES

3.3.1 TEST UNITAIRE VIA ANNOTATIONS

1. créer un répertoire Serveur/lib et copier les jars suivants :
   - junit-4.4.jar
   - spring-test.jar
2. créer un répertoire source Serveur/srctest
3. Créer une classe pour chaque test unitaire, dans le même package que la classe testée.

Exemple de classe de test :

```
1: @RunWith(SpringJUnit4ClassRunner.class)
2: @ContextConfiguration(locations={"/appContext.xml"})
   public final class ExampleServiceTest {

       /** auto-injection via Spring */
3:     @Autowired
       private ExampleService exampleService;

4:     @org.junit.Test
       public void test() throws Exception {
           List<ExampleVO> utils = exampleService.listByRole("utilisateur");
           Assert.assertEquals(1, utils.size());
       }
   }
```

1 : annotations indiquant le runner junit utilisé (ici celui de Spring pour JUnit 4)
2 : annotations indiquant le fichier de configuration de Spring à utiliser
3 : annotations indiquant que la la dépendance sera automatiquement résolu par Spring (la correspondance se fait sur le nom et le type java du bean défini dans appContext.xml)

4 : annotations indiquant que la méthode est un test (excutable par le runner junit)