tutorial-osgi-camel-part2

Introduction

The purpose of this tutorial is not at all to teach you on SOA but to draw your attention on points that the developer(s)/deployer(s) will be confronted during the design/development and release management phases.

Designing a Service Oriented Architecture seems very obvious for most of us but implies that different parameters are taken into account :

- Identification of the layers of the application,
- Definition of the services,
- Granularity (what are the boundaries of a service, ...),
- Dependency with libraries,
- Testing and debugging strategies,
- Deployment procedure,
- Infrastructure

Some of the points mentioned are particular to SOA world like granularity and definition of service boundaries but others are mostly found in all IT projects. This is really important to keep them in your head because they will impact the project life cycle, quality of the deliverable, efficiency of the team and project duration/cost.

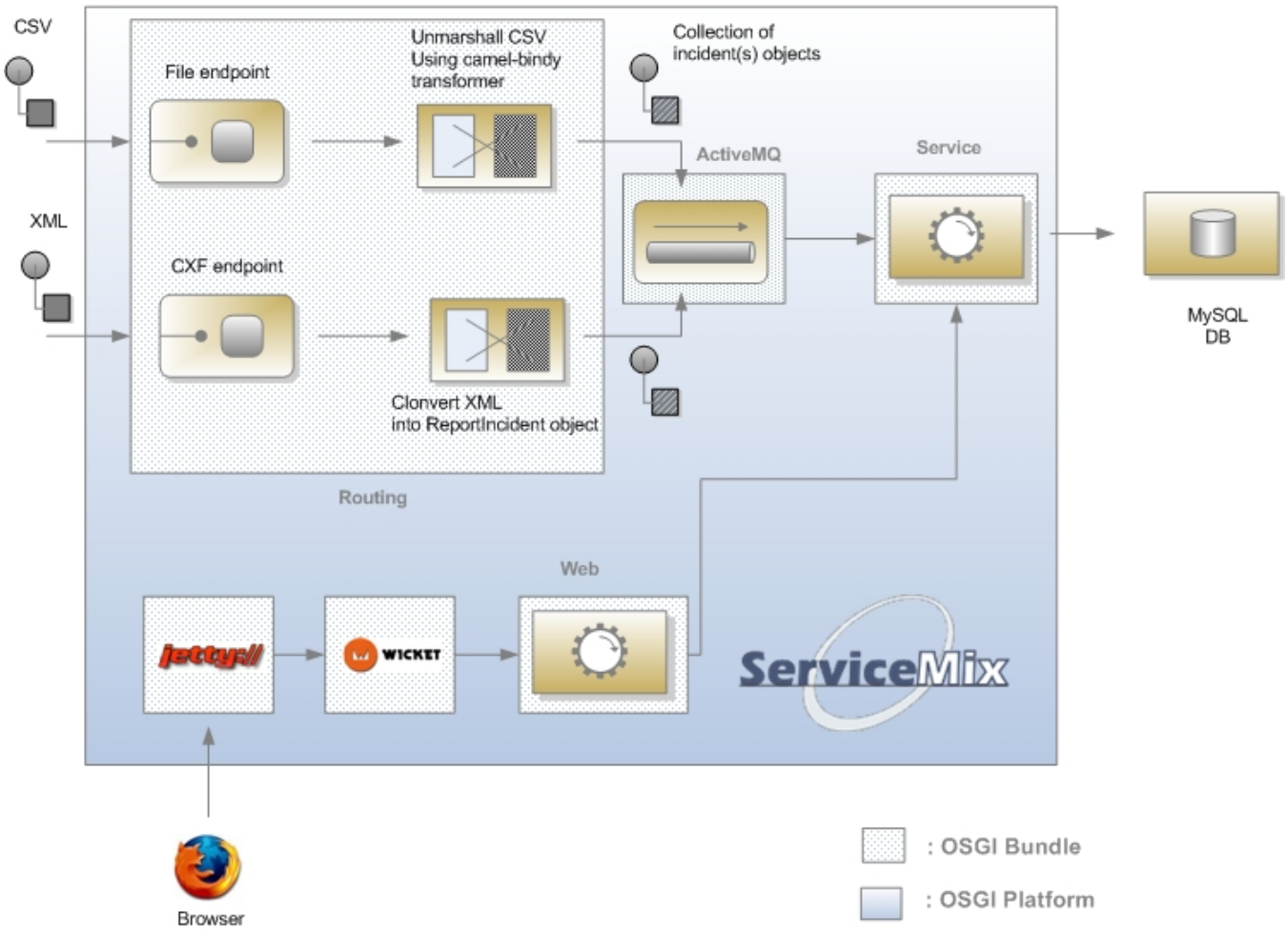
In this second part of the tutorial we will investigate some of the points mentioned and applied them to a real application. The application will be designed around different components (= bundles in the OSGI jargon) and deployed into Apache Felix Karaf (now part of Apache ServiceMix 4).

Apache Karaf is a small OSGi based runtime which provides a lightweight container onto which various components and applications can be deployed.

Here is a short list of features supported by the Karaf:

- Hot deployment: Karaf supports hot deployment of OSGi bundles by monitoring jar files inside the home/deploy directory. Each time a jar is copied in this folder, it will be installed inside the runtime. You can then update or delete it and changes will be handled automatically. In addition, the Karaf also supports exploded bundles and custom deployers (blueprint and spring ones are included by default).
- Dynamic configuration: Services are usually configured through the ConfigurationAdmin OSGi service. Such configuration can be defined in Karaf using property files inside the home/etc directory. These configurations are monitored and changes on the properties files will be propagated to the services.
- Logging System: using a centralized logging back end supported by Log4J, Karaf supports a number of different APIs (JDK 1.4, JCL, SLF4J, Avalon, Tomcat, OSGi)
- Provisioning: Provisioning of libraries or applications can be done through a number of different ways, by which they will be downloaded locally, installed and started.
- Native OS integration: Karaf can be integrated into your own Operating System as a service so that the lifecycle will be bound to your Operating System.
- Extensible Shell console: Karaf features a nice text console where you can manage the services, install new applications or libraries and manage their state. This shell is easily extensible by deploying new commands dynamically along with new features or applications.
- Remote access: use any SSH client to connect to Karaf and issue commands in the console
- Security framework based on JAAS
- Managing instances: Karaf provides simple commands for managing multiple instances. You can easily create, delete, start and stop instances of Karaf through the console.
- Supports the latest OSGi 4.2 containers: Apache Felix Framework 2.0.0 and Eclipse Equinox 3.5

Here is a picture of the report incident application that this tutorial will cover :



To summarize, the application is listening for incidents coming from web service or files. According to the origin, the content (= incidents) are transformed into their corresponding objects using for the CSV file, a new camel component : camel-bindy and for the Web Service camel-cxf component. Each message transformed is placed in a queue handled by ActiveMQ engine. All the messages (containing the objects) are next processed by a Bean service who will (with the help of injection of dependency provided by Spring) save the incidents in a DB using Spring and Hibernate frameworks.
A small Apache Wicket web application running in Jetty Web server provide to the users a screen to consult the incidents created.

Remark : A bundle in the OSGI world represents component made by developer. For more info about OSGI, I recommend to have a look on OSGI web site

The project has been cut into the following components :

Maven project name = artifactId	Description	Is it a bundle ?
reportincident.activemq	configuration file of the ActiveMQ engine	yes
reportincident.camelqueueservice	configuration file of the camel-activemq component	yes
reportincident.db	generator of the script DB	no
reportincident.features	features provisioning file containing our bundles dependencies	no
reportincident.model	model layer	yes
reportincident.persistence	hibernate persistence layer; bundle	yes
reportincident.routing	camel routing	yes
reportincident.service	spring service layer	yes
reportincident.web	apache wicket module	yes
reportincident.webservice	CXF web service generator	yes

As you can see, some are considered as OSGI bundles and others no. An important point to mention here concerns the granularity : each layer of our application will be deployed as separate bundle. This will facilitate the maintenance and release management. Of course, you can argue that the granularity is too small. SOA is not an exact science and depending of the size of the application, the team in charge to develop, release management procedure this cutting will be redefined. You can imagine that the parameters used to configure Hibernate and Spring are bundles together instead inside the persistence project. Service bundle could be split into several bundles; one by service type, ... There are no rules of thumb except that the project must be manageable and maintainable.

Prerequisites

This tutorial uses:

- Maven 2.0.9 to setup the projects,
- Eclipse Ganymede 3.4.x,
- Maven eclipse plugin,
- Apache Felix Karaf 1.4.0,
- Dependencies (= jars) used by the tutorial will be downloaded (if not available locally) by Maven

Note: The sample project can be downloaded, see the resources section.

Step 1 : Initial Project Setup

Different way exist to create maven project. For the basic project like db, we have used the archetype 'simple' with the command followed by `mvn eclipse:eclipse` in the folder created :

```
mvn archetype:generate -DinteractiveMode=false -DgroupId=org.apache.camel.example -DartifactId=reportincident.
model -Dversion=1.0-SNAPSHOT -Dgoals=eclipse:eclipse
```

For the OSGI bundles, different approaches are available depending on the tools that you prefer to use :

- Spring archetype
- Spring bundlor
- PAX maven plugin

But for the purpose of this tutorial, we have used the PAX maven plugin. Why this choice, simply because PAX maven plugin offers a lot of advantages regarding to the one of Spring :

- pom.xml file generated is very simple to use and to understand,
- project can be designed with several modules,
- project can be tested with PAX Exam and launched using PAX runner
- generate all the folders required including also the META-INF,
- manifest file is generated automatically,
- can be imported easily in Eclipse

To create the tutorial projects, you can follow the procedure described here

1) Execute maven command in your Unix/Dos console :

```
mvn org.ops4j:maven-pax-plugin:create-bundle -Dpackage=org.apache.camel.example.reportincident.model
-DbundleGroupId=reportincident.model -DbundleName=reportincident.model -Dversion=1.0-SNAPSHOT
```

2) Move to the folder created and execute the following command :

```
mvn org.ops4j:maven-pax-plugin:eclipse
```

2) Import the generated eclipse project in Eclipse workspace
3) Delete non required files like readme.txt and the folders internal + java class created
4) Enable dependency management (see sonatype site).

Repeat this procedure for the projects :

- reportincident.activemq
- reportincident.camelqueueservice
- reportincident.persistence
- reportincident.routing
- reportincident.service
- reportincident.web
- reportincident.webservice

otherwise import the content of the unzipped file in your workspace. You will gain time.

Step 2 : Develop model layer

It is time now to begin serious things. One of the most important part of a project (if not the most important) concerns the design of the model.
The reportincident model is really simple because it only contains one class that we will use :

- to map information with the database, CSV file,
- to transport information to web screens.

Here is the definition of the incident class that you can create in the reportincident.model project directory `src/main/java/org/apache/camel/example/reportincident/model` or use the code imported

```
import java.io.Serializable;

public class Incident implements Serializable{

        private static final long serialVersionUID = 1L;

        private long incidentId;

        private String incidentRef;

        private Date incidentDate;

        private String givenName;

        private String familyName;

        private String summary;

        private String details;

        private String email;

        private String phone;

        private String creationUser;

        private Date creationDate;


        public long getIncidentId() {
                return incidentId;
        }

        public void setIncidentId(long incidentId) {
                this.incidentId = incidentId;
        }

        public String getIncidentRef() {
                return incidentRef;
        }

        public void setIncidentRef(String incidentRef) {
                this.incidentRef = incidentRef;
        }

        public Date getIncidentDate() {
                return incidentDate;
        }

        public void setIncidentDate(Date incidentDate) {
                this.incidentDate = incidentDate;
        }

        public String getGivenName() {
                return givenName;
        }
```

```
        public void setGivenName(String givenName) {
                this.givenName = givenName;
        }

        public String getFamilyName() {
                return familyName;
        }

        public void setFamilyName(String familyName) {
                this.familyName = familyName;
        }

        public String getSummary() {
                return summary;
        }

        public void setSummary(String summary) {
                this.summary = summary;
        }

        public String getDetails() {
                return details;
        }

        public void setDetails(String details) {
                this.details = details;
        }

        public String getEmail() {
                return email;
        }

        public void setEmail(String email) {
                this.email = email;
        }

        public String getPhone() {
                return phone;
        }

        public void setPhone(String phone) {
                this.phone = phone;
        }

        public String getCreationUser() {
                return creationUser;
        }

        public void setCreationUser(String creationUser) {
                this.creationUser = creationUser;
        }

        public Date getCreationDate() {
                return creationDate;
        }

        public void setCreationDate(Date creationDate) {
                this.creationDate = creationDate;
        }

}
```

Step 3 : Map model layer with CSV file (camel-bindy)

To facilitate the work of the modeler, we will use the incident class not only to persist the information in the database but also to read or generate Comma Separate Value file (CSV). To map the content of the class with a CSV file, we have used a new Camel component : camel-bindy. Like its name suggests, camel-bindy is a binding framework (similar to JAXB2) to map non structured information with Java class using annotations. The current version supports CSV fields and key-value pairs (e.g. Financial FIX messages) but will be extended in the future to support Fixed Length format, ...

So, we will modify our existing class to add @Annotations required to map its content. This is very trivial to do and will be done in two steps :

1) Add CSVRecord annotation

This annotation will help camel-bindy to discover what is the parent class of the model and which separator is used to separate the fields. If required, you can also use the property 'skipFirstLine' to skip the first line of your CSV file

```
import org.apache.camel.dataformat.bindy.annotation.CsvRecord;

@CsvRecord(separator =",")
public class Incident implements Serializable{
...
}
```

2) Add DataFields annotations

For each of the CSV field that you want to bind with your model, you must add the @DataField annotation with its position. This is not the only property available and you can also add 'pattern' property to by example define the pattern of your Date field.

```
import org.apache.camel.dataformat.bindy.annotation.CsvRecord;
import org.apache.camel.dataformat.bindy.annotation.DataField;

@CsvRecord(separator =",")
public class Incident implements Serializable{

    @DataField(pos = 1)
    private String incidentRef;

    @DataField(pos = 2, pattern = "dd-mm-yyyy")
    private Date incidentDate;

    @DataField(pos = 3)
    private String givenName;

    @DataField(pos = 4)
    private String familyName;

    @DataField(pos = 5)
    private String summary;

    @DataField(pos = 6)
    private String details;

    @DataField(pos = 7)
    private String email;

    @DataField(pos = 8)
    private String phone;

...
}
```

To build the project, simply execute the following maven command in the reportincident.model project

```
mvn clean install
```

Step 4 : Map model layer with DB (Hibernate)

To map our model with the database, we will use the ORM framework Hibernate. Annotation can also be used since the last version of Hibernate but to avoid to overload our class and reduce its readability, we will use the old way using a XML file describing the mapping between the model and the database.

Remark : The ORM uses to persist the information is Hibernate but it can be changed to another existing like iBatis, Apache OpenJPA, ...

```
<hibernate-mapping schema="REPORT">
        <class name="org.apache.camel.example.reportincident.model.Incident" table="T_INCIDENT">
                <meta attribute="extends">Abstract</meta>
                <id name="incidentId" column="INCIDENT_ID"  type="long">
                        <generator class="native" />
                </id>

                <property column="INCIDENT_REF" name="incidentRef" length="55" type="string" />
                <property column="INCIDENT_DATE" lazy="false" length="8" name="incidentDate" type="timestamp" />
                <property column="GIVEN_NAME" length="35" name="givenName" type="string" />
                <property column="FAMILY_NAME" length="35" name="familyName" type="string" />
                <property column="SUMMARY" length="35" name="summary" type="string" />
                <property column="DETAILS" length="255" name="details" type="string" />
                <property column="EMAIL" length="60" name="email" type="string" />
                <property column="PHONE" length="35" name="phone" type="string" />

                <property column="CREATION_DATE" generated="never" lazy="false" name="creationDate" type="
timestamp" />
                <property column="CREATION_USER" generated="never" lazy="false" name="creationUser" type="
string" />
        </class>
</hibernate-mapping>
```

Remark : This file `Incident.hbm.xml` must be created in the directory `src\main\resources\META-INF\org\apache\camel\example\reportincident\model\Incident.hbm.xml` of the project reportincident.model.

Step 6 : Database creation

To create the database, we will use hibernate maven plugin file. The plugin will use the following configuration file to generate the SQL script and create table T_Incident.

Remark : MySQL has been used for the purpose of the tutorial

Here is the content of the hibernate.cfg.xml that you must create in the folder `src/config` of hibernate.db

```
<!-- MySQL DB -->

<hibernate-configuration>
        <session-factory name="reportincident">
                <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
                <property name="hibernate.connection.url">jdbc:mysql:///report</property>
                <property name="hibernate.connection.username">root</property>
                <property name="hibernate.dialect">org.hibernate.dialect.MySQL5Dialect</property>
                <property name="hibernate.connection.password" />
                <property name="hibernate.show_sql">true</property>

        <!-- mapping files -->
        <mapping resource="META-INF/org/apache/camel/example/reportincident/model/Incident.hbm.xml"/>

        </session-factory>
</hibernate-configuration>
```

The pom.xml file of your reportincident.db project must be modified like this :

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
        <modelVersion>4.0.0</modelVersion>
        <groupId>org.apache.camel.example</groupId>
        <artifactId>reportincident.db</artifactId>
        <packaging>jar</packaging>
        <name>Report Incident DB </name>
        <version>1.0-SNAPSHOT</version>

        <dependencies>
                <dependency> (1)
                        <groupId>org.apache.camel.example</groupId>
                        <artifactId>reportincident.model</artifactId>
                        <version>1.0-SNAPSHOT</version>
                </dependency>
        </dependencies>

        <build>
                <plugins>

                        <!-- Hibernate  plugin -->
                        <plugin>
                                <groupId>org.codehaus.mojo</groupId>
                                <artifactId>hibernate3-maven-plugin</artifactId>
                                <version>2.2</version>
                                <configuration>
                                        <components>
                                                <component>
                                                        <name>hbm2ddl</name>
                                                </component>
                                        </components>
                                        <componentProperties>
                                                <drop>true</drop>
                                                <create>true</create>
                                                <format>true</format>
                                                <configurationfile>/src/config/hibernate.cfg.xml<
/configurationfile>
                                                <outputfilename>db_reportincident_create_hsqldb.sql<
/outputfilename>
                                        </componentProperties>
                                </configuration>
                                <dependencies>
                                        <dependency> (2)
                                                <groupId>mysql</groupId>
                                                <artifactId>mysql-connector-java</artifactId>
                                                <version>5.1.8</version>
                                        </dependency>
                                </dependencies>

                                <executions>
                                        <execution>
                                                <phase>process-classes</phase>
                                                <goals>
                                                        <goal>hbm2ddl</goal>
                                                </goals>
                                        </execution>
                                </executions>

                        </plugin>

                </plugins>
        </build>
</project>
```

Remarks :
(1) - Dependency with reportincident.model project must be added because the plugin requires the file `Incident.hbm.xml` to generate the script/db
(2) - If you prefer to use another DB instead of MySql, change the dependency in the pom.xml and hibernate.connection.driver_class and hibernate.connection.url in the cfg file

To create the table + SQL script, simply launch

```
mvn clean install
```

command in the folder of reportincident.db

Step 7 : Add persistence layer and Spring service

Now that the model/db exist, we will create the persistence and layer services. The projects have been designed using the pattern Data Access Object because it allows to change the implementation from a database type to another, between ORM very easily. Moreover interfaces are used as 'contract' between the services and the DAO. This offers the advantage to decouple objects in the application and as you will see later on it will allow us to deploy services, persistence as separate bundles in the OSGI server.

Persistence project

First, we will create the interface declaring the methods that we would like to provide/expose. Create in the folder `src/main/java/org/apache/camel/example/reportincident/dao`, the java class "IncidentDAO" with the following code :

```
package org.apache.camel.example.reportincident.dao;

import java.util.List;

import org.apache.camel.example.reportincident.model.Incident;

public interface IncidentDAO
{

    /**
     * Gets the Incident.
     *
     * @param id the id
     * @return the incident
     */
    public Incident getIncident( long id );

    /**
     * Find all incidents.
     *
     * @return the list<Incident>
     */
    public List<Incident> findIncident();

    /**
     * Find Incident using incident id ref.
     *
     * @param key the key
     * @return the list< order>
     */
    public List<Incident> findIncident( String key );

    /**
     * Save Incident.
     *
     * @param incident the Incident
     */
    public void saveIncident( Incident incident );

    /**
     * Removes the Incident.
     *
     * @param id the id
     */
    public void removeIncident( long id );

}
```

There is nothing particular to mention here as this class is a simple case of Create Read Update Delete implementation. The next class who implements the interface will provide the necessary code to connect to the database using Hibernate framework.

So, create the class `IncidentDAOImpl` in the directory `src/main/java/org/apache/camel/example/reportincident/dao/impl`

```
package org.apache.camel.example.reportincident.dao.impl;


import java.util.List;

import org.apache.camel.example.reportincident.dao.IncidentDAO;
import org.apache.camel.example.reportincident.model.Incident;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.hibernate.HibernateException;
import org.hibernate.Query;
import org.hibernate.SessionFactory;
import org.hibernate.impl.SessionImpl;
```

```
public class IncidentDAOImpl implements IncidentDAO
{

        private static final transient Log LOG = LogFactory.getLog(IncidentDAOImpl.class);

    /** The session factory. */
    private SessionFactory sessionFactory;

    /** The q. */
    private Query q = null;

    /** The Constant findIncidentByReference. */
    private final static String findIncidentByReference =
        "select i from Incident as i where i.incidentRef = :ref";

    /** The Constant findIncident. */
    private final static String findIncident =
        "select i from Incident as i";

    /**
     * Sets the session factory.
     *
     * @param sessionFactory the new session factory
     */
    public void setSessionFactory( SessionFactory sessionFactory )
    {
        this.sessionFactory = sessionFactory;
    }

    /*
     * (non-Javadoc)
     * @see org.apache.camel.example.reportincident.dao.IncidentDAO#findIncident()
     */
    public List<Incident> findIncident()
        throws HibernateException
    {

        // Prepare query
        q = this.sessionFactory.getCurrentSession().createQuery( findIncident );

        // Retrieve the Incidents from database
        List<Incident> list = q.list();

        return list;

    }

    /*
     * (non-Javadoc)
     * @see org.apache.camel.example.reportincident.dao.IncidentDAO#findIncident(java.lang.String)
     */
    public List<Incident> findIncident( String key )
        throws HibernateException
    {
        q = this.sessionFactory.getCurrentSession().createQuery( findIncidentByReference );
        q.setString("ref", key );
        List<Incident> list = q.list();

        return list;
    }

    /*
     * (non-Javadoc)
     * @see org.apache.camel.example.reportincident.dao.IncidentDAO#getIncident(long)
     */
    public Incident getIncident( long id )
    {
        return (Incident) this.sessionFactory.getCurrentSession().get( Incident.class, id );
    }
```

```
    /*
     * (non-Javadoc)
     * @see org.apache.camel.example.reportincident.dao.IncidentDAO#removeIncident(long)
     */
    public void removeIncident( long id )
    {
        Object record = this.sessionFactory.getCurrentSession().load( Incident.class, id );
        this.sessionFactory.getCurrentSession().delete( record );

    }

    /*
     * (non-Javadoc)
     * @see org.apache.camel.example.reportincident.dao.IncidentDAO#saveIncident(org.apache.camel.example.
reportincident.model.Incident)
     */
    public void saveIncident( Incident Incident )
    {
            SessionImpl session = (SessionImpl) this.sessionFactory.getCurrentSession();
        this.sessionFactory.getCurrentSession().saveOrUpdate( Incident );
    }

}
```

The most important point to mention here is that this class to connect to our database and to work with Hibernate needs to have a SessionFactory object. This object is not instantiated by a constructor's class but only declared as a property/field. This is where Spring will help us through its dependency injection.

The injection is defined in the file called `spring-dao-beans.xml` that you will create in the folder `src/main/resources/META-INF/spring` :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="
             http://www.springframework.org/schema/beans
             http://www.springframework.org/schema/beans/spring-beans.xsd">

        <!-- DAO Declarations -->
        <bean id="incidentDAO" class="org.apache.camel.example.reportincident.dao.impl.IncidentDAOImpl">
                <property name="sessionFactory">
                        <ref bean="sessionFactory" />
                </property>
        </bean>

</beans>
```

The sessionFactory object will be created with the help of Spring framework but in order to communicate with the database, information about the data source must be provided.

So realize this goal, you will create the file `spring-datasource-beans.xml` in the same folder directory with the following information :

```
...
        <!--  Hibernate SessionFactory Definition -->
        <bean id="sessionFactory" class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">

                <property name="mappingLocations">
                        <list>
                                <value>classpath*:META-INF/org/apache/camel/example/reportincident/model/*.hbm.
xml</value>
                        </list>
                </property>

                <property name="hibernateProperties">
                        <props>
                                <prop key="hibernate.dialect">org.hibernate.dialect.MySQLDialect</prop>
                                <prop key="hibernate.show_sql">false</prop>
                                <prop key="hibernate.format_sql">true</prop>
                                <prop key="hibernate.cglib.use_reflection_optimizer">true</prop>
                                <prop key="hibernate.jdbc.batch_size">10</prop>
                                <prop key="hibernate.query.factory_class">org.hibernate.hql.classic.
ClassicQueryTranslatorFactory</prop>
                        </props>
                </property>
                <property name="dataSource">
                        <ref bean="dataSource" />
                </property>
...
        <!--  DB connection and persistence layer -->
        <!--  DataSource Definition  -->
        <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
                <property name="driverClassName" value="${driverClassName}" />
                <property name="url" value="${url}" />
                <property name="username" value="${username}" />
                <property name="password" value="${password}" />
        </bean>
```

This file is not complete but we will review later in the tutorial when we will cover specific OSGI stuffs and Spring transaction management. Now, we will design the Spring service part

Spring Service project

In term of design, the service project is very similar to the persistence because we will create an interface and its implementation. Why repeating the interface. The answer is evident; it is for decoupling the service from the DAO implentation to allow you to switch easily from one ORM to another, ...

Create the following interface `IncidentService` in the folder `src/main/java/org/apache/camel/example/reportincident/service` with the code :

```
package org.apache.camel.example.reportincident.service;

import java.util.List;

import org.apache.camel.example.reportincident.model.Incident;

public interface IncidentService
{

    /**
     * Gets incident.
     *
     * @param id the id
     * @return the incident
     */
    public Incident getIncident( long id );

    /**
     * Find all Incidents.
     *
     * @return the list<Incident>
     */
    public List<Incident> findIncident();

    /**
     * Find Incident by key ref.
     *
     * @param key the key
     * @return the list< order>
     */
    public List<Incident> findIncident( String key );

    /**
     * Save Incident.
     *
     * @param incident the Incident
     */
    public void saveIncident( Incident incident );

    /**
     * Removes the Incident.
     *
     * @param id the id
     */
    public void removeIncident( long id );

}
```

and its implementation `IncidentServiceImpl` in the folder `src/main/java/org/apache/camel/example/reportincident/service`

```
package org.apache.camel.example.reportincident.service.impl;

import java.util.List;

import org.apache.camel.example.reportincident.model.Incident;
import org.apache.camel.example.reportincident.dao.IncidentDAO;
import org.apache.camel.example.reportincident.service.IncidentService;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

public class IncidentServiceImpl implements IncidentService {

        private static final transient Log LOG = LogFactory.getLog(IncidentServiceImpl.class);

        /** The incident dao. */
        private IncidentDAO incidentDAO;

        public void saveIncident(Incident incident) {

                try {
                        getIncidentDAO().saveIncident(incident);
                } catch (RuntimeException e) {
                        e.printStackTrace();
                }

        }

        public void removeIncident(long id) {
                getIncidentDAO().removeIncident(id);
        }

        public Incident getIncident(long id) {
                return getIncidentDAO().getIncident(id);
        }

        public List<Incident> findIncident() {
                return getIncidentDAO().findIncident();
        }

        public List<Incident> findIncident(String key) {
                return getIncidentDAO().findIncident(key);
        }

        /**
         * Gets the incident dao.
         *
         * @return the incident dao
         */
        public IncidentDAO getIncidentDAO() {
                return incidentDAO;
        }

        /**
         * Sets the incident dao.
         *
         * @param incidentDAO
         *            the new incident dao
         */
        public void setIncidentDAO(IncidentDAO incidentDAO) {
                this.incidentDAO = incidentDAO;
        }

}
```

The same remark as explained previously applies here concerning the DAO injection. So, you will create the following file `spring-service-beans-dao.xml` in the folder `src/main/resources/META-INF/spring` to inject the dependency of the DAO to our service.

```
       <bean id="incidentServiceTarget" class="org.apache.camel.example.reportincident.service.impl.
IncidentServiceImpl">
            <property name="incidentDAO">
                    ...
            </property>
       </bean>
```

Obviously, this file is not complete because the reference of the DAO class is not mentioned except the property name. Don't panic, we will come back later on when we will discuss Spring Blueprint services.

Step 8 : Webservice

This part has already been discussed in detail in the excellent tutorial : Report Incident - This tutorial introduces Camel steadily and is based on a real life integration problem. So we will only explain what we have done specifically for our project.

Compare to the other tutorial, we have packaged the code generated by the CXF framework in a project/bundle separated from the routing/mediation engine. This approach allows you to extend your web services (I mean the methods exposed) without impacting the rest of your application. The question concerning the model is mush more delicate because we have a dependency on the model created to persist information. In our case, we have separated the webservice model (where the fields available are all declared as string) from ours but you can considered to have the same when Types are compatible (e.g. Can I map the Date Time object of my webservice field to my model without any transformation ?).

To generate the code that our application will use, we will work with following WSDL contract `report_incident.wsdl` that you create in the directory `src/main/resources/META-INF/wsdl`:

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

<wsdl:definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
        xmlns:tns="http://reportincident.example.camel.apache.org"
        xmlns:xs="http://www.w3.org/2001/XMLSchema"
        xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
        targetNamespace="http://reportincident.example.camel.apache.org">

    <!-- Type definitions for input- and output parameters for webservice -->
    <wsdl:types>
        <xs:schema targetNamespace="http://reportincident.example.camel.apache.org">
            <xs:element name="inputReportIncident">
                <xs:complexType name="inputReportIncident">
                    <xs:sequence>
                        <xs:element type="xs:string" name="incidentId"/>
                        <xs:element type="xs:string" name="incidentDate"/>
                        <xs:element type="xs:string" name="givenName"/>
                        <xs:element type="xs:string" name="familyName"/>
                        <xs:element type="xs:string" name="summary"/>
                        <xs:element type="xs:string" name="details"/>
                        <xs:element type="xs:string" name="email"/>
                        <xs:element type="xs:string" name="phone"/>
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
            <xs:element name="outputReportIncident">
                <xs:complexType name="outputReportIncident">
                    <xs:sequence>
                        <xs:element type="xs:string" name="code"/>
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
        </xs:schema>
    </wsdl:types>

    <!-- Message definitions for input and output -->
    <wsdl:message name="inputReportIncident">
        <wsdl:part name="in" element="tns:inputReportIncident"/>
    </wsdl:message>
    <wsdl:message name="outputReportIncident">
        <wsdl:part name="out" element="tns:outputReportIncident"/>
    </wsdl:message>
```

```
    <!-- Port (interface) definitions -->
    <wsdl:portType name="ReportIncidentEndpoint">
        <wsdl:operation name="ReportIncident">
            <wsdl:input message="tns:inputReportIncident"/>
            <wsdl:output message="tns:outputReportIncident"/>
        </wsdl:operation>
    </wsdl:portType>

    <!-- Port bindings to transports and encoding - HTTP, document literal encoding is used -->
    <wsdl:binding name="ReportIncidentBinding" type="tns:ReportIncidentEndpoint">
        <soap:binding transport="http://schemas.xmlsoap.org/soap/http"/>
        <wsdl:operation name="ReportIncident">
            <soap:operation
                    soapAction="http://reportincident.example.camel.apache.org/ReportIncident"
                    style="document"/>
            <wsdl:input>
                <soap:body parts="in" use="literal"/>
            </wsdl:input>
            <wsdl:output>
                <soap:body parts="out" use="literal"/>
            </wsdl:output>
        </wsdl:operation>
    </wsdl:binding>

    <!-- Service definition -->
    <wsdl:service name="ReportIncidentEndpointService">
        <wsdl:port name="ReportIncidentPort" binding="tns:ReportIncidentBinding">
            <soap:address location="http://localhost:8080/camel-example/incident"/>
        </wsdl:port>
    </wsdl:service>

</wsdl:definitions>
```

The code will be generated thanks to a maven plugin : cxf-codegen-plugin.

Add the following line in your `pom.xml` of the project `reportincident.webservice`

```
<!-- CXF wsdl2java generator, will plugin to the compile goal -->
                        <plugin>
                                <groupId>org.apache.cxf</groupId>
                                <artifactId>cxf-codegen-plugin</artifactId>
                                <version>${cxf-version}</version>
                                <executions>
                                        <execution>
                                                <id>generate-sources</id>
                                                <phase>generate-sources</phase>
                                                <configuration>
                                                        <sourceRoot>${basedir}/target/generated/src/main/java<
/sourceRoot>
                                                        <wsdlOptions>
                                                                <wsdlOption>
                                                                        <wsdl>${basedir}/src/main/resources
/META-INF/wsdl/report_incident.wsdl</wsdl>
                                                                </wsdlOption>
                                                        </wsdlOptions>
                                                </configuration>
                                                <goals>
                                                        <goal>wsdl2java</goal>
                                                </goals>
                                        </execution>
                                </executions>

                        </plugin>
```

The code is generated using the maven command :

```
mvn generate-sources
```

Remark : the code is generated in the directory `target/src/main/java`

Conclusion

Everything is in place to integrate the services together except the routing and OSGI stuffs. This is what we will cover in the following sections.

It is time now to have a break, to make some sport exercices, to drink a cup of good 'Java' coffee or to go outside of the building to take a walk with your favorite pets.

Links

- Part 2 : real example, architecture, project setup, database creation
- Part 2a : transform projects in bundles
- Part 2b : add infrastructure and routing
- Part 2c : web and deployment

#Resources

File	Modified
ZIP Archive tutorial-osgi-camel-part2.zip	Apr 01, 2010 by Moulliard Charles