



[Apache Struts 2 Documentation](#) > [Home](#) > [Tutorials](#) > [Struts 2 + Spring 2 + JPA + AJAX](#)

Apache Struts 2 Documentation
Struts 2 + Spring 2 + JPA + AJAX

 [Edit Page](#)  [Browse Space](#)  [Add Page](#)  [Add News](#)

Added by [Musachy Barroso](#), last edited by [Dave Newton](#) on Feb 16, 2008 ([view change](#)) [show comment](#)

On this tutorial we will demonstrate how to setup Struts 2 in Eclipse, and make it work with Spring, Java Persistence API (using Hibernate) and Struts 2 Ajax tags.

-  Hibernate is licensed under the [LGPL](#)[■], and any application created using Hibernate is subject to the terms of the LGPL.
-  Following this tutorial verbatim will require use of a Struts 2 deployment greater than 2.0.3

- [Prerequisites](#)
 - [Tomcat](#)
 - [MySql](#)
- [Get the code](#)
 - [Show me the code](#)
 - [The maven way](#)
- [Doing it yourself](#)
 - [Create Eclipse project](#)
 - [Dependencies](#)
 - [Domain](#)
 - [Person service.](#)
 - [JPA configuration](#)
 - [Spring](#)
 - [Struts](#)
 - [The pages](#)
 - [Validation](#)
- [Using Toplink Essentials instead of Hibernate](#)
- [References](#)

Prerequisites

- [Tomcat](#)[■]
- [Eclipse](#)[■]
- [MySQL Server](#)[■] ([MySQL licensing policy](#)[■])

Tomcat

Install Tomcat before going forward. See Tomcat's installation guide if you have any problem installing it.

MySql

Install and configure MySQL. Create a database named "quickstart" and run the script below to create the "Person" table. Later, on applicationContext.xml, we'll use 'root' as the user name and password for the database, remember to replace those values with the right ones for your database.

```
CREATE TABLE 'quickstart'.'Person' (
  'id' INTEGER UNSIGNED NOT NULL AUTO_INCREMENT,
  'firstName' VARCHAR(45) NOT NULL,
  'lastName' VARCHAR(45) NOT NULL,
  PRIMARY KEY('id')
)
ENGINE = InnoDB;
```

Get the code

Show me the code

You can just download the [zipped Eclipse project](#)[■], add the required dependencies to the lib folder under the /WebContent/WEB-INF/lib folder (relative to project's root folder) and import it into Eclipse.

The maven way

To run the project this way you will need maven installed.

1. Download the [zipped project](#)
2. Download jta jar from [here](#).
 - Note that the Download Manager may save the file to your root drive, and it may give the file a .ZIP extension. You must rename the file to *jta-1.1-classes.jar*.
 - If a later version is available, update the version references in the next step.
3. Install the jta jar file running:

```
$ mvn install:install-file -DgroupId=javax.transaction -DartifactId=jta -Dversion=1.1 -Dpackaging=jar -Dfile=c:\
```

4. Bear with me, we are almost there
5. cd into quickstart and run:

```
$ mvn jetty:run
```

6. Point your browser to <http://localhost:8080/quickstart>
7. To create an eclipse project run:

```
$ mvn eclipse:eclipse
```

or (to create web project for WTP):

```
mvn eclipse:eclipse -Dwtpversion=1.0
```

Doing it yourself

Create Eclipse project





1. Open Eclipse. Seriously, you need to open Eclipse.
2. Click File -> New -> Project. Under the "Web" folder, select "Dynamic Web Project" and click "Next".
3. Enter the project name, "quickstart" from here on. The project will be running inside Tomcat, so we need to create a server configuration for it.
 1. Under "Target Runtime", click "New", select "Apache Tomcat 5.5" and click next.
 2. Enter Tomcat's installation directory and select an installed JRE (1.5 is required)
4. Now you should be back to the project creation wizard, with Tomcat as your Target Runtime. Click "Next". Select "Dynamic Web Module" and "Java" facets, and click "Finish".

Dependencies

Your project should contain the folders "src", "build" and "WebContent". We are going to put all the required jars under "/WebContent /WEB-INF/lib". To add files to the "lib" folder, just copy them to `${workspace}\quickstart\WebContent\WEB-INF\lib`, where `${workspace}` is the location of your Eclipse workspace folder.

In the table, the version has been removed from the JAR files, since these may change in future milestone releases. Use whatever version is shipping with the indicated products.

JAR	From	License
xwork.jar	Struts 2	Apache License
struts2-core.jar	Struts 2	
struts2-spring-plugin.jar	Struts 2	
ognl.jar	Struts 2	
freemarker.jar	Struts 2	
commons-logging-api.jar	Struts 2	
mysql-connector-java.jar	MySQL JDBC Driver	MySQL licensing policy
spring.jar	Spring 2.0	Apache License
antlr.jar	Hibernate Core	LGPL
asm.jar	Hibernate Core	
asm-attrs.jar	Hibernate Core	
cglib.jar	Hibernate Core	
dom4j.jar	Hibernate Core	
jdbc2_0-stdext.jar	Hibernate Core	
ehcache.jar	Hibernate Core	
hibernate3.jar	Hibernate Core	
xml-apis.jar	Hibernate Core	
commons-collections.jar	Hibernate Core	

ejb3-persistence.jar	Hibernate Annotations  LGPL 
jta.jar	Hibernate Annotations
hibernate-commons-annotations.jar	Hibernate Annotations
hibernate-annotations.jar	Hibernate Annotations
hibernate-entitymanager.jar	Hibernate Entity Manager  LGPL 
javassist.jar	Hibernate Entity Manager
jboss-archive-browsing.jar	Hibernate Entity Manager

Right click on the project and select "Refresh" (to notify Eclipse of the jars that we just added).

Domain

Our domain model will consist of just a simple "Person" class with a couple of fields.

1. Create a new class named "Person" (File -> New -> Class), and enter "quickstart.model" for the package name.
2. Add the fields "id" (int), "firstName" (String), and lastName ("String") with their setter/getter methods.
3. Mark your class with the "@Entity" annotation, and the "id" field with the annotations "@Id" and "@GeneratedValue".

your class will look like:

Person.java

```
package quickstart.model;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;

@Entity
public class Person {
    @Id
    @GeneratedValue
    private Integer id;
    private String lastName;
    private String firstName;

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }
}
```

@Entity will let the provider know that this class can be persisted. @Id marks the "id" field as the primary key for this class.

@GeneratedValue will cause the id field to be generated by the provider (Hibernate). Classes and fields are by default mapped to tables and columns with the same name, see JPA's documentation for more details.

Person service.

We will now write the class that will take care of CRUD operations on "Person" objects.

1. Create a new interface (File -> New -> Interface), enter "PersonService" for the name, and "quickstart.service" for the namespace. Set its content to:

PersonService.java

```
package quickstart.service;

import java.util.List;
```

```
import quickstart.model.Person;

public interface PersonService {
    public List<Person> findAll();

    public void save(Person person);

    public void remove(int id);

    public Person find(int id);
}
```

1. Create a new class (File -> New -> Class), enter "PersonServiceImpl" for the name and "quickstart.service" for the namespace. Set its content to:

PersonServiceImpl.java

```
package quickstart.service;

import java.util.List;

import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.Query;

import org.springframework.transaction.annotation.Transactional;

import quickstart.model.Person;

@Transactional
public class PersonServiceImpl implements PersonService {
    private EntityManager em;

    @PersistenceContext
    public void setEntityManager(EntityManager em) {
        this.em = em;
    }

    @SuppressWarnings("unchecked")
    public List<Person> findAll() {
        Query query = getEntityManager().createQuery("select p FROM Person p");
        return query.getResultList();
    }

    public void save(Person person) {
        if (person.getId() == null) {
            // new
            em.persist(person);
        } else {
            // update
            em.merge(person);
        }
    }

    public void remove(int id) {
        Person person = find(id);
        if (person != null) {
            em.remove(person);
        }
    }

    private EntityManager getEntityManager() {
        return em;
    }

    public Person find(int id) {
        return em.find(Person.class, id);
    }
}
```

@PersistenceContext will make Spring inject an EntityManager into the service when it is instantiated. The @PersistenceContext annotation can be placed on the field, or on the setter method. If the class is annotated as @Transactional, Spring will make sure that its methods run inside a transaction.

JPA configuration

1. Create a folder named "META-INF" under the "src" folder.
2. Create a file named "persistence.xml" under the "META-INF" folder and set its content to:

persistence.xml

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```

    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/ns/persistence/persistence_1_0
    version="1.0">
    <persistence-unit name="punit">
    </persistence-unit>
</persistence>

```

JPA configuration can be set on this file. On this example it will be empty because the datasource configuration will be in the Spring configuration file.

Spring

1. Update the content of web.xml under /WebContent/WEB-INF/web.xml to:

web.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app id="person" version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
    <display-name>person</display-name>

    <!-- Include this if you are using Hibernate -->
    <filter>
        <filter-name>Spring OpenEntityManagerInViewFilter</filter-name>
        <filter-class>
            org.springframework.orm.jpa.support.OpenEntityManagerInViewFilter
        </filter-class>
    </filter>

    <filter-mapping>
        <filter-name>Spring OpenEntityManagerInViewFilter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>

    <filter>
        <filter-name>struts2</filter-name>
        <filter-class>
            org.apache.struts2.dispatcher.FilterDispatcher
        </filter-class>
    </filter>

    <filter-mapping>
        <filter-name>struts2</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>

    <welcome-file-list>
        <welcome-file>index.jsp</welcome-file>
    </welcome-file-list>

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

```

This will make the container redirect all requests to Struts "FilterDispatcher" class. "index.jsp" is set as the home page, and Spring's "ContextLoaderListener" is configured as a listener.

1. Create a file named "applicationContext.xml" under /WebContent/WEB-INF, and set its content to:

applicationContext.xml

```

<?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:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
        http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-2.0.xsd
        http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-2.0.xsd">

    <bean
        class="org.springframework.orm.jpa.support.PersistenceAnnotationBeanPostProcessor" />

    <bean id="personService" class="quickstart.service.PersonServiceImpl" />

    <bean id="entityManagerFactory"
        class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
        <property name="dataSource" ref="dataSource" />
        <property name="jpaVendorAdapter">
            <bean

```

```

        class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
        <property name="database" value="MYSQL" />
        <property name="showSql" value="true" />
    </bean>
</property>
</bean>

<bean id="dataSource"
    class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="com.mysql.jdbc.Driver" />
    <property name="url" value="jdbc:mysql://localhost/quickstart" />
    <property name="username" value="root" />
    <property name="password" value="root" />
</bean>

<bean id="transactionManager"
    class="org.springframework.orm.jpa.JpaTransactionManager">
    <property name="entityManagerFactory" ref="entityManagerFactory" />
</bean>

<tx:annotation-driven transaction-manager="transactionManager" />

<bean id="personAction" scope="prototype"
    class="quickstart.action.PersonAction">
    <constructor-arg ref="personService" />
</bean>
</beans>

```

Note that the "class" attribute of the bean "personAction" is set to the name of the action class, and the "personService" bean will be passed as a parameter to the action constructor. Change the "url", "username" and "password" in the "dataSource" bean to the appropriate values for your database. For more details on the rest of the beans on this file, see Spring's documentation. The "scope" attribute is new in Spring 2, and it means that Spring will create a new PersonAction object every time an object of that type is requested. In Struts 2 a new action object is created to serve each request, that's why we need scope="prototype".

Struts

We will now create a simple Struts action that wraps PersonServices methods, and we will configure Struts to use Spring as the object factory.

1. Open the new class dialog (File -> New -> Class) and enter "PersonAction" for the classname, and "quickstart.action" for the namespace. Set its content to:

PersonAction.java

```

package quickstart.action;

import java.util.List;

import quickstart.model.Person;
import quickstart.service.PersonService;

import com.opensymphony.xwork2.Action;
import com.opensymphony.xwork2.Preparable;

public class PersonAction implements Preparable {
    private PersonService service;
    private List<Person> persons;
    private Person person;
    private Integer id;

    public PersonAction(PersonService service) {
        this.service = service;
    }

    public String execute() {
        this.persons = service.findAll();
        return Action.SUCCESS;
    }

    public String save() {
        this.service.save(person);
        this.person = new Person();
        return execute();
    }

    public String remove() {
        service.remove(id);
        return execute();
    }

    public List<Person> getPersons() {
        return persons;
    }
}

```

```

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public void prepare() throws Exception {
        if (id != null)
            person = service.find(id);
    }

    public Person getPerson() {
        return person;
    }

    public void setPerson(Person person) {
        this.person = person;
    }
}

```

Look mom my action is a simple POJO!

The "Preparable" interface instructs Struts to call the "prepare" method if the "PrepareInterceptor" is applied to the action (by default, it is). The constructor of the action takes a "PersonService" as a parameter, which Spring will take care of passing when the action is instantiated.

1. Create a new file named "struts.xml" under the "src" folder. And set its content to:

struts.xml

```

<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
    "http://struts.apache.org/dtds/struts-2.0.dtd">
<struts>
    <constant name="struts.objectFactory" value="spring" />
    <constant name="struts.devMode" value="true" />

    <package name="person" extends="struts-default">

        <action name="list" method="execute" class="personAction">
            <result>pages/list.jsp</result>
            <result name="input">pages/list.jsp</result>
        </action>

        <action name="remove" class="personAction" method="remove">
            <result>pages/list.jsp</result>
            <result name="input">pages/list.jsp</result>
        </action>

        <action name="save" class="personAction" method="save">
            <result>pages/list.jsp</result>
            <result name="input">pages/list.jsp</result>
        </action>
    </package>
</struts>

```

Setting "struts.objectFactory" to "spring" will force Struts to instantiate the actions using Spring, injecting all the defined dependencies on applicationContext.xml. The "class" attribute for each action alias is set to "personAction", which is the bean id that we defined on applicationContext.xml for the PersonAction class. This is all that is needed to make Struts work with Spring.

The pages

We only have two pages, "index.jsp" and "list.jsp". "list.jsp" returns a table with a list of the persons on the database. We have this list on a different page because we are going to add some AJAX to spicy it up.

1. Create a new file named "list.jsp" under /WebContent/pages/ and set its content to:

list.jsp

```

<%@ taglib prefix="s" uri="/struts-tags"%>

<p>Persons</p>
<s:if test="persons.size > 0">
    <table>
        <s:iterator value="persons">
            <tr id="row_<s:property value="id"/>">
                <td>
                    <s:property value="firstName" />
                </td>
                <td>
                    <s:property value="lastName" />

```

```

        </td>
        <td>
            <s:url id="removeUrl" action="remove">
                <s:param name="id" value="id" />
            </s:url>
            <s:a href="{removeUrl}" theme="ajax" targets="persons">Remove</s:a>
            <s:a id="a_{id}" theme="ajax" notifyTopics="/edit">Edit</s:a>
        </td>
    </tr>
</s:iterator>
</table>
</s:if>

```

This is going to render a table with each row showing the first and last name of the person, a link to remove the person, and a link to edit. The remove link has the attribute "targets", set to "persons", which means that when the user clicks on it, an asynchronous request will be made to the "remove" action (as configured on struts.xml, "remove" points to the "remove" method in PersonAction), passing the person id as parameter.

When the edit link is clicked on, it will publish the "/edit" topic, which will trigger a javascript function to populate the fields.

1. Create a new file named "index.jsp" under /WebContent and set its content to:

index.jsp

```

<%@ taglib prefix="s" uri="/struts-tags"%>
<html>
    <head>
        <s:head theme="ajax" debug="true"/>
        <script type="text/javascript">
            dojo.event.topic.subscribe("/save", function(data, type, request) {
                if(type == "load") {
                    dojo.byId("id").value = "";
                    dojo.byId("firstName").value = "";
                    dojo.byId("lastName").value = "";
                }
            });

            dojo.event.topic.subscribe("/edit", function(data, type, request) {
                if(type == "before") {
                    var id = data.split("_")[1];

                    var tr = dojo.byId("row_"+id);
                    var tds = tr.getElementsByTagName("td");

                    dojo.byId("id").value = id;
                    dojo.byId("firstName").value = dojo.string.trim(dojo.dom.textContent(tds[0]));
                    dojo.byId("lastName").value = dojo.string.trim(dojo.dom.textContent(tds[1]));
                }
            });
        </script>
    </head>
    <body>
        <s:url action="list" id="descrsUrl"/>

        <div style="width: 300px;border-style: solid">
            <div style="text-align: right;">
                <s:a theme="ajax" notifyTopics="/refresh">Refresh</s:a>
            </div>
            <s:div id="persons" theme="ajax" href="{descrsUrl}" loadingText="Loading..." listenTopics="/refresh">
            </div>

            <br/>

            <div style="width: 300px;border-style: solid">
                <p>Person Data</p>
                <s:form action="save" validate="true">
                    <s:textfield id="id" name="person.id" cssStyle="display:none"/>
                    <s:textfield id="firstName" label="First Name" name="person.firstName"/>
                    <s:textfield id="lastName" label="Last Name" name="person.lastName"/>
                    <s:submit theme="ajax" targets="persons" notifyTopics="/save"/>
                </s:form>
            </div>
        </body>
    </html>

```

Look mom no page refresh!

The div "persons" will load its content asynchronously, and will show "Loading..." while while the request is on progress (you can use the "indicator" attribute for better progress feedback), you can force it to refresh clicking on the "Refresh" link. The "submit" button, will make an asynchronous request to the action "save" ("save" method on PersonAction), and will publish the topic "/save" to which we subscribed to, using "dojo.event.topic.subscribe", to clear the input fields.

Validation

Because we don't want any John Doe on our database, we will add some basic client side validation to our form. In Struts 2, validation can be placed on xml files with the name pattern `ActionName-validation.xml`, located on the same package as the action. To add validation to an specific alias of an action (like a method), the validation file name follows the pattern `ActionName-alias-validation.xml`, where "alias" is the action alias name (in this case a method name, "save"). Add a file named `"PersonAction-save-validation.xml"` under `/src/quickstart/action`, and set its content to:

```
<!DOCTYPE validators PUBLIC
"-//OpenSymphony Group//XWork Validator 1.0//EN"
"http://www.opensymphony.com/xwork/xwork-validator-1.0.2.dtd">
<validators>
  <field name="person.firstName">
    <field-validator type="requiredstring">
      <message>First name is required!</message>
    </field-validator>
  </field>
  <field name="person.lastName">
    <field-validator type="requiredstring">
      <message>Last name is required!</message>
    </field-validator>
  </field>
</validators>
```

See the Struts documentation for details on existing validators, and how to write, and plug in, your own validators.

To run the project, Right click on your project and Run As -> Run on Server. You can debug it on the same way, Right click on the project and Debug As -> Debug on Server. Download and install Struts 2 Showcase to see more examples.

Using Toplink Essentials instead of Hibernate

1. Add this to pom.xml

```
<repositories>
  <repository>
    <id>java.net</id>
    <url>https://maven-repository.dev.java.net/nonav/repository</url>
    <layout>legacy</layout>
  </repository>
</repositories>
```

2. Add this to the *dependencies* node in pom.xml

```
<dependency>
  <groupId>toplink.essentials</groupId>
  <artifactId>toplink-essentials</artifactId>
  <version>2.0-38</version>
  <exclusions>
    <exclusion>
      <groupId>javax.transaction</groupId>
      <artifactId>jta</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```

3. Replace the `jpaVendorAdapter` element in *applicationContext.xml* with this:

```
<property name="jpaVendorAdapter">
  <bean class="org.springframework.orm.jpa.vendor.TopLinkJpaVendorAdapter">
    <property name="databasePlatform" value="oracle.toplink.essentials.platform.database.MySQL4Platform" />
    <property name="generateDdl" value="true" />
    <property name="showSql" value="true" />
  </bean>
</property>
```

References

[Struts](#)[■]
[Spring JPA Doc](#)[■]
[JPA and Spring Tutorial](#)[■]
[Eclipse Dal](#)[■]

Generated by [Atlassian Confluence](#) (Version: 2.2.9 Build: 527 Sep 07, 2006) [AutoExport Plugin](#) (Version: Unknown - PluginManager Error)