Web Programming: Struts2, JSTL & Spring

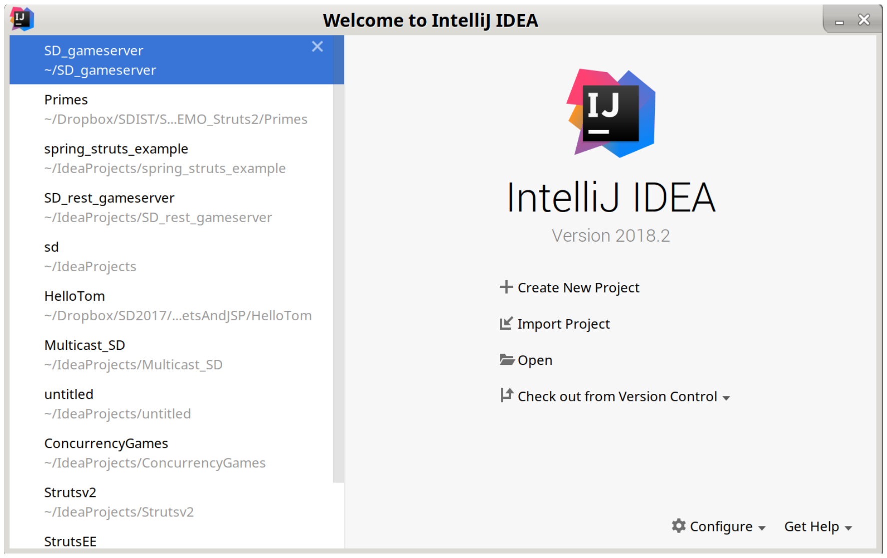
Sistemas Distribuídos 2018/19

This guide comes with four separate projects: `Primes`, `Hey`, `Clock` and `Spring`. Follow the guide and answer all the questions, but before doing that begin with a *quick start* example on how to open and configure projects from the filesystem.

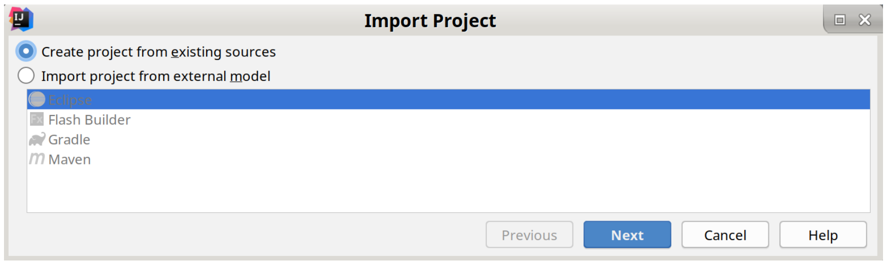
1 Quick start

Web projects can be launched from your IDE (IntelliJ, Eclipse, NetBeans, *etc.*) or executed by Tomcat without any IDE. Usually, you will launch Web applications directly from the IDE by creating a **'web project'** and running it on a server. Occasionally, if you wish to execute a Web application directly in Tomcat you just need to place a WAR file in Tomcat's `webapps` directory. A WAR file is a JAR file containing a Web application.

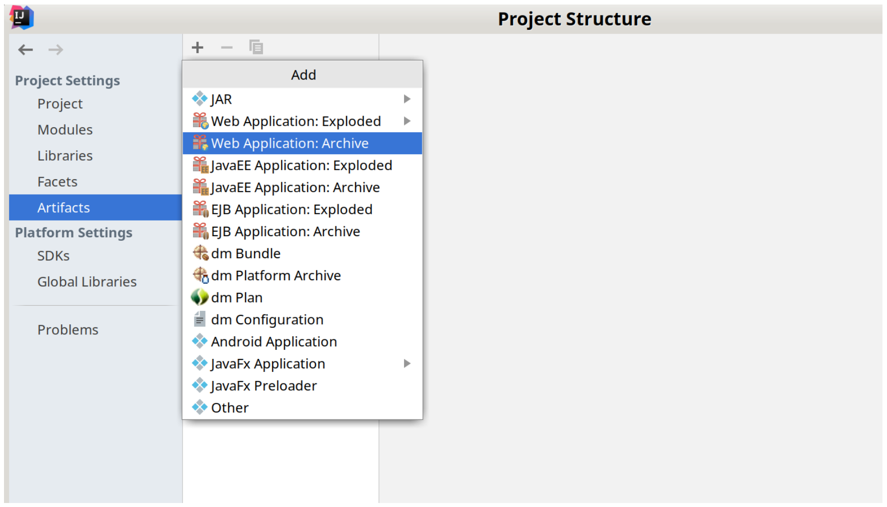
1. From IntelliJ's main screen choose `Import project`.



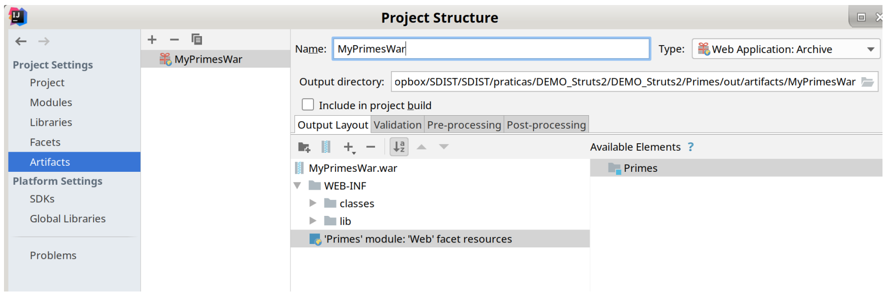
2. Follow the wizard and choose `Create project from existing sources` when prompted.


Import Project
Create project from existing sources
Import project from external model
Eclipse
Flash Builder
Gradle
Maven
Previous
Next
Cancel
Help

3. Open the `Project Structure` window (through the File menu or pressing `Ctrl+Alt+Shift+S`), go to the Artifacts tab, press the `+` icon and choose to add a new `Web Application: Archive`. Change the name of this artifact to something useful ('Primes').



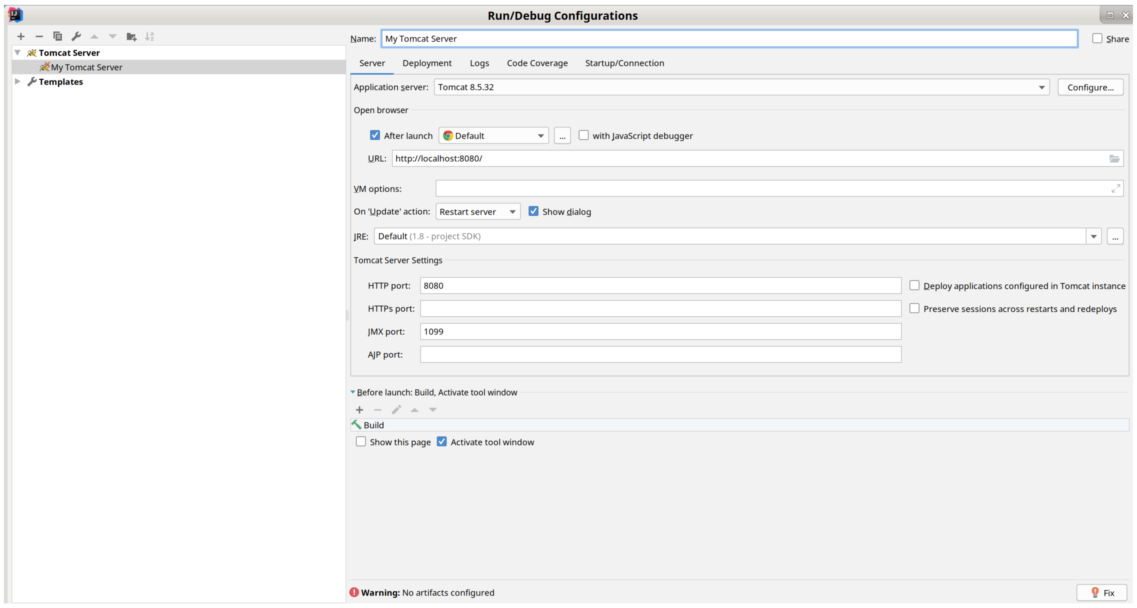
4. From the `Available Elements` pane (on the right) <u>select and double-click all the available items</u>. These will be added to the content of the produced WAR and will be shown on the left side.



Name: MyPrimesWar
Type: Web Application: Archive
Output directory: opbox/SDIST/SDIST/praticas/DEMO_Struts2/DEMO_Struts2/Primes/out/artifacts/MyPrimesWar
☐ Include in project build

```
MyPrimesWar.war
▼ WEB-INF
  ▶ classes
  ▶ lib
  'Primes' module: 'Web' facet resources
```

5. To run you project choose `Run` on the toolbar, then `Edit configurations...` In this window click on the `+` sign and add a new `Tomcat server (Local)`. Set `Application Server` to be your Tomcat server that you should have already configured (otherwise you will need to download Tomcat from the website, unzip and then click on Configure...).


Run/Debug Configurations

Tomcat Server
My Tomcat Server
Templates

Name: My Tomcat Server
☐ Share

Server | Deployment | Logs | Code Coverage | Startup/Connection

Application server: Tomcat 8.5.32 | Configure...

Open browser
✓ After launch: Default | ☐ with JavaScript debugger
URL: http://localhost:8080/

VM options:
On 'Update' action: Restart server | ✓ Show dialog
JRE: Default (1.8 - project SDK)

Tomcat Server Settings
HTTP port: 8080 | ☐ Deploy applications configured in Tomcat instance
HTTPs port: | ☐ Preserve sessions across restarts and redeploys
JMX port: 1099
AJP port:

Before launch: Build, Activate tool window
Build
☐ Show this page | ✓ Activate tool window

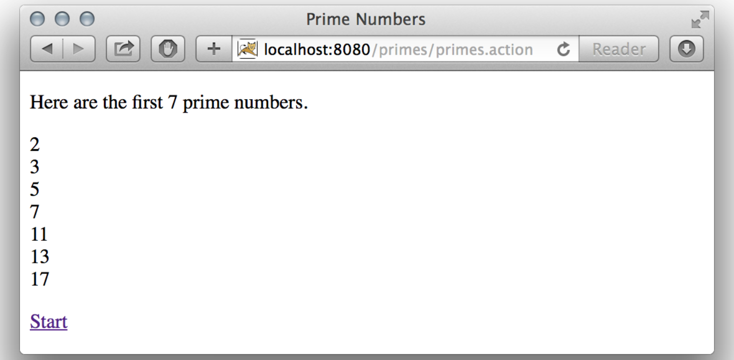
Warning: No artifacts configured | Fix

6. You will see the message ‘Warning: No artifacts configured’ at the bottom, click the `Fix` button. You should now `Run` the project.

2 Running the first Struts2 program (Primes example)

The examples provided with this guide require you to have Apache Tomcat installed somewhere in your computer. Feel free to ask for help integrating the web server with IntelliJ, Eclipse, NetBeans, and so on. It may also help if you are familiar with the Ant build tool or with Maven. In any case, the end result of compiling a Web project is typically a WAR file (Web Application Archive) which is a JAR file containing a Web application.

Start by following the above guide to run the ’Primes’ example in your IDE, and it will be automatically deployed. Open `http://localhost:8080/primes` in your browser and press the submit button to get the following result:


Prime Numbers
localhost:8080/primes/primes.action

Here are the first 7 prime numbers.

2
3
5
7
11
13
17

Start

Note that the URL (`http://localhost:8080/primes`) depends on the name you chose for your project.

Furthermore, the input form is not entirely protected. A negative number will be fine, but try submitting an exclamation mark (!) in the form. Observation: user input must never be trusted.

Question 1. The first **view** that users see is the result of `index.jsp` executing and constructing the following HTML:

```
<body>
    Prime numbers needed?
    <form name="primes" action="/primes/primes.action" method="post">
        <input type="text" name="primesBean.number" value=""/>
        <input type="submit" value="Submit"/>
    </form>
</body>
```

We are using Struts UI Tags to generate the form. This is done by including `<%@ taglib prefix="s" uri="/struts-tags"%>` at the top of the JSP file. Which tags have we used in this example?

Question 2. In the prime numbers example, indicate source files containing one model, one view, and one action. How do the three work together to print the prime numbers?

Question 3. We structured the code in such a way that the **Model** components are JavaBeans, the **View** components are JSPs, and the **Controller** components are Actions. How does Struts2 know how to "glue" everything? For example, how are we telling struts to 1) start by displaying `index.jsp`, 2) update `PrimesBean`, 3) execute `PrimesAction`, and display `primes.jsp` at the end?

Hint: There's an XML file with a suggestive name.

3 Sessions and external data sources (Hey example)

The example in the `Hey` directory uses an external data source, which is an RMI server. If you open the file `HeyBean.java` the constructor sets up the connection to the server. As expected, `HeyBean` is a model which encapsulates access to data. Run the second Struts2 example by following these steps:

1. Start up the `RMIServer`.
2. Configure and run the 'Hey' example in your IDE.
3. Open `http://localhost:8080/hey`

Question 4. Open `LoginAction.java` and observe that the `execute()` method just accepts any user without confirming if the password is correct. Modify the code so that it checks if the password matches the user by accessing the RMI server (through the bean).

Hint: Add a password field to `index.jsp` so that `LoginAction.setPassword()` is called when the submit button is pushed. Then, rewrite the `execute()` method to check if the user matches the password.

Question 5. If you turn off the RMI server, what happens to the application?

Question 6. Why does the class `LoginAction` implement `SessionAware`?

Question 7. Implement a clone of Yo, which will be called Hey! The only operation consists of saying "Hey!" to other users. Whenever you say "Hey!" to someone, he/she just needs to refresh the page to see that you did so.

Hint: The view `hey.jsp` prints the names, and you need to modify it in order to be able to press names (use HTML links, checkboxes or radiobuttons). Then, create a new action called SayHeyAction and don't forget to add it to `struts.xml` so that you can submit requests to it. Then, add two remote methods to the RMI server (one to say Hey! and one to read the incoming Heys!). Finally, modify `hey.jsp` so that it reads incoming Heys! from the bean.

4 Interceptors (Clock example)

Inside the Clock example directory, start by opening the `struts.xml` file. In addition to actions and views, you now see the `<interceptors>` tag. We are adding a new interceptor on top of the existing `defaultStack`, which will execute before *all* actions. This interceptor is specified in `ClockInterceptor.java` and you should examine this code. Build and run the Clock example.

Question 8. Can we use an interceptor to check if a user is logged in? Try it on the example of Question 7, or on your practical assignment.

Hint: Add an interceptor that only executes any action if and only if the user is logged in. Exclude the login action from this interceptor (by using the `defaultStack` *for that action alone).*

5 Integration with Spring

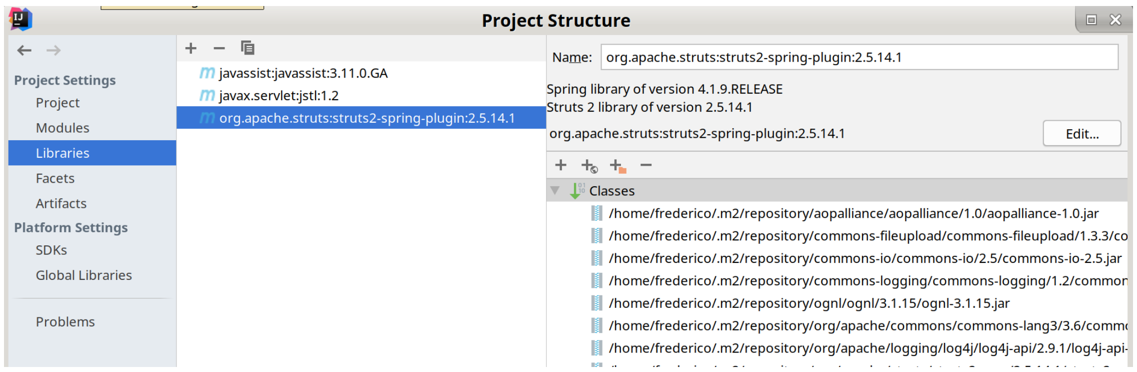
Spring is a popular Java framework that provides a range of useful features for the development of web applications. The most used feature is *'dependency injection'*, which can be described as the programmer giving the responsibility of managing the lifecycle of objects (creation, destruction) to Spring.

When developing a web application that follows the MVC model, it is common to see a series of support classes (*e.g.*, services, utility, validators) declared inside of the controller classes (actions in the case of Struts2), for example:

```
public class AdminLoginAction extends ActionSupport {

  private AuthService authService = new AdminAuthService();

}

public class UserLoginAction extends ActionSupport {

  private AuthService authService = new UserAuthService();

}
```

With Spring, we can use *'dependency injection'* to inject at run-time the desired service (*AdminAuthService* or *UserAuthService*) and only need to write one general action instead of two actions.

The first step to allow your Struts2 project to take advantage of Spring's dependency injection is adding the `struts2-spring-plugin` to your project's libraries, as shown in Figure 1.



javassist:javassist:3.11.0.GA
javax.servlet:jstl:1.2
org.apache.struts:struts2-spring-plugin:2.5.14.1

Name: org.apache.struts:struts2-spring-plugin:2.5.14.1

Spring library of version 4.1.9.RELEASE
Struts 2 library of version 2.5.14.1

org.apache.struts:struts2-spring-plugin:2.5.14.1

```
/home/frederico/.m2/repository/aopalliance/aopalliance/1.0/aopalliance-1.0.jar
/home/frederico/.m2/repository/commons-fileupload/commons-fileupload/1.3.3/co
/home/frederico/.m2/repository/commons-io/commons-io/2.5/commons-io-2.5.jar
/home/frederico/.m2/repository/commons-logging/commons-logging/1.2/commor
/home/frederico/.m2/repository/ognl/ognl/3.1.15/ognl-3.1.15.jar
/home/frederico/.m2/repository/org/apache/commons/commons-lang3/3.6/comm
/home/frederico/.m2/repository/org/apache/logging/log4j/log4j-api/2.9.1/log4j-api-
```

Fig. 1: Project structure that includes 'struts2-spring-plugin'.

After the plugin is installed you need to edit your project's `web.xml` file to add a new listener, as below:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         (...) >

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

The next step is creating a new file with the name `applicationContext.xml` inside of the `WEB-INF` folder. This XML file will declare the *'beans'* known by Spring and which will be used for the dependency injection. Each bean declaration uses the <`bean`> tag and requires an `id` and, usually, the Java `class` associated to the bean (`class` is not a mandatory parameter but is very useful). Below is an example of a possible `applicationContext.xml` for a scenario where there are two services that are used to authenticate normal and administrator users.

```
<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-2.5.xsd">

    <bean id="adminAuthService" class="services.AdminAuthService" />

    <bean id="userAuthService" class="services.UserAuthService" />

    <bean id="userLoginAction" class="actions.GenericLoginAction" scope="prototype">
        <property name="authService" ref="userAuthService" />
    </bean>

    <bean id="adminLoginAction" class="actions.GenericLoginAction" scope="prototype">
        <property name="authService" ref="adminAuthService" />
    </bean>

</beans>
```

Please note that two different beans (*'userLoginAction'* and *'adminLoginAction'*) share the same Java class (*'actions.GenericLoginAction'*) but use a different bean for the authentication service, specified using the <`property`> tag.

The last step is to add these actions to `struts.xml`:

```
        <action name="adminLogin" class="adminLoginAction">
            <result>admin.jsp</result>
        </action>

        <action name="userLogin" class="userLoginAction">
            <result>user.jsp</result>
        </action>
```

You should note that in the `class` parameter of the <`action`> tag we pass the id of the Spring bean instead of passing the Java class name as is usually done.

Inside the `Spring` folder there is a sample project that showcases how Spring can be integrated in Struts2, repeating some of the concepts that have been talked before. The project consists of a search engine of a sales company that allows the client to search for products according to a set of criteria. Spring is used in the action class to allow one single Java class to be used for 3 different types of products (cars, laptops and shoes) by using *dependency injection* to choose the service and model that should be used for each product type.

Open the project and run it, then analyze and understand the code. Answer the following questions, some of which may require visiting the list of useful links available in the next section.

Question 9. How are we able to have 3 different types of search (car, laptop and shoes) using the same action?

Question 10 What is the purpose of the *'inputObject'* field of the SearchAction? Which information will it keep?

Question 11 What is required for dependency injection through Spring? What do you need to have on the Java-side and on the XML-side for dependency injection?

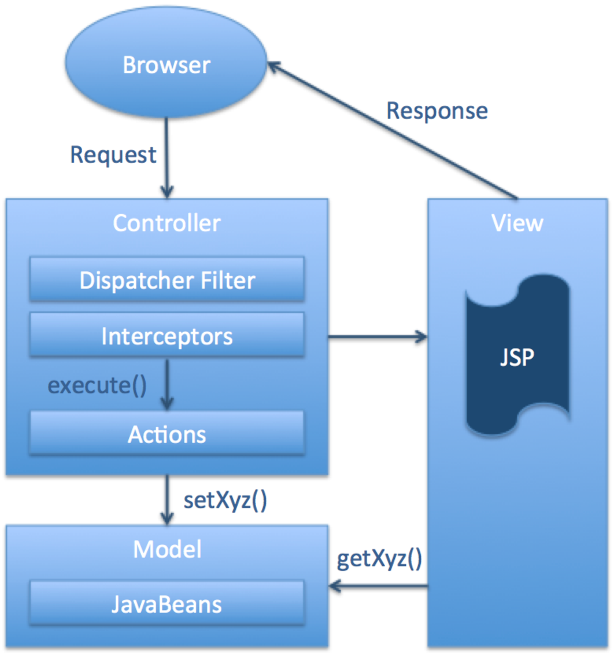
Question 12 What is the difference between a singleton and a prototype scope (of a bean)? What is the default scope used by Spring?

Question 13 What is a problem that could occur if the action and model beans were not declared with a prototype scope? Why is it OK for the service beans to have a singleton scope?

Question 14 Adapt the code provided for the Clock example so that it uses Spring to inject the ClockBean in ClockAction. Run the project to see whether your changes work.

6 Documentation

All the Struts2 examples provided in this guide use the following MVC structure:



In these examples, there is no Java code in any view. We use Struts UI Tags (those starting with `<s:`) and JSTL Tags (starting `<c:`) to execute `if` statements, iterators, etc. Always avoid Java code in JSP files, and use only JSP tags, Struts2 UI tags, or JSTL tags. Read also the supplied documentation:

Struts UI Tags `Docs/struts-2.3.16-docs/ui-tag-reference.html`

JSTL Specification `Docs/jstl-1.2-spec.pdf`

Struts2+Spring `https://struts.apache.org/getting-started/spring.html`

Struts2's Spring plugin `https://struts.apache.org/plugins/spring/`

Spring Framework Overview `https://docs.spring.io/spring/docs/5.1.1.RELEASE/spring-framework-reference/overview.html`