MVC Frameworks – STRUTS2

In this tutorial we will learn how to setup a basic STRUTS2 web application. We will also learn how to use an IDE, and an integrated build tool that will simply the setup process.

Do not expect to finish this lab in one session – there is a lot of content to cover! Especially if you've never used an IDE before!

1. Installing IntelliJ IDEA

IntelliJ is a popular Java IDE (Integrated Development Environment). An IDE provides many advanced features over a standard text editor such as code completion and the automatic building of projects. While IntelliJ is not the only Java IDE (e.g. eclipse and netbeans), it will be the preferred IDE for this course.

IntelliJ comes in two editions: the free community editions and paid ultimate edition. The community edition is sufficient for standard Java desktop and console applications. However, for web development we need the ultimate edition. Fortunately, JetBrains (the company behind IntelliJ) provides free student licenses for IntellIJ.

1.1 Firstly we need to sign up for a JetBrains student license. You can sign up here: https://www.jetbrains.com/shop/eform/students

Important: You will need to sign up using your @uon.edu.au student email address.

2.2 Secondly, we need to install IntelliJ IDEA. You can download the IntelliJ installer here: https://www.jetbrains.com/idea/download/

Important: Make sure you select the 'Ultimate' edition!

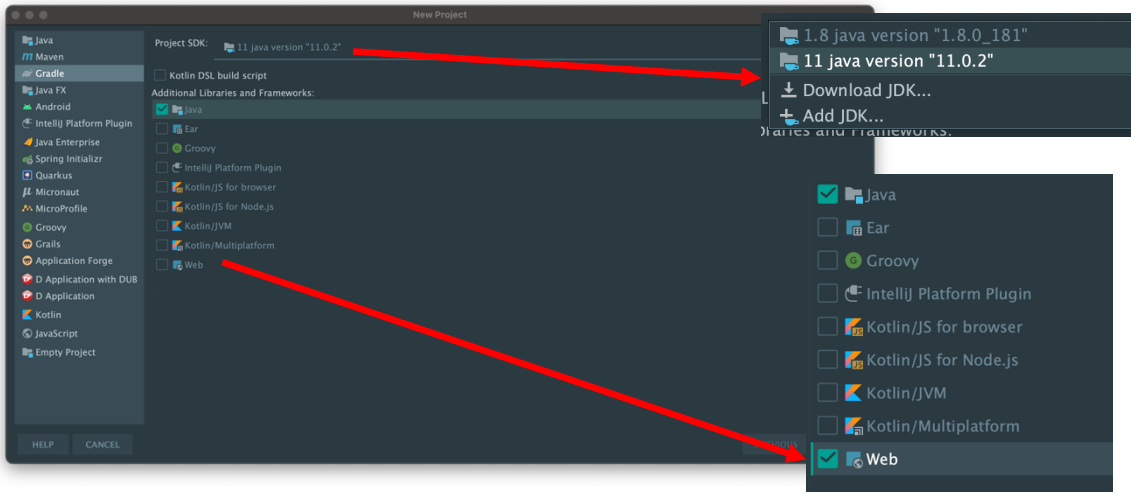
3.3 Open up the installer and follow the prompts to install IntellIJ. On the first-time opening IntelliJ you will be prompted for license details. Ensure your sign in with your newly created JetBrains account.

Important: If you are on a lab PC, make sure you install IntelliJ to your U: drive.

2. Creating a Java Web project in IntelliJ with Gradle

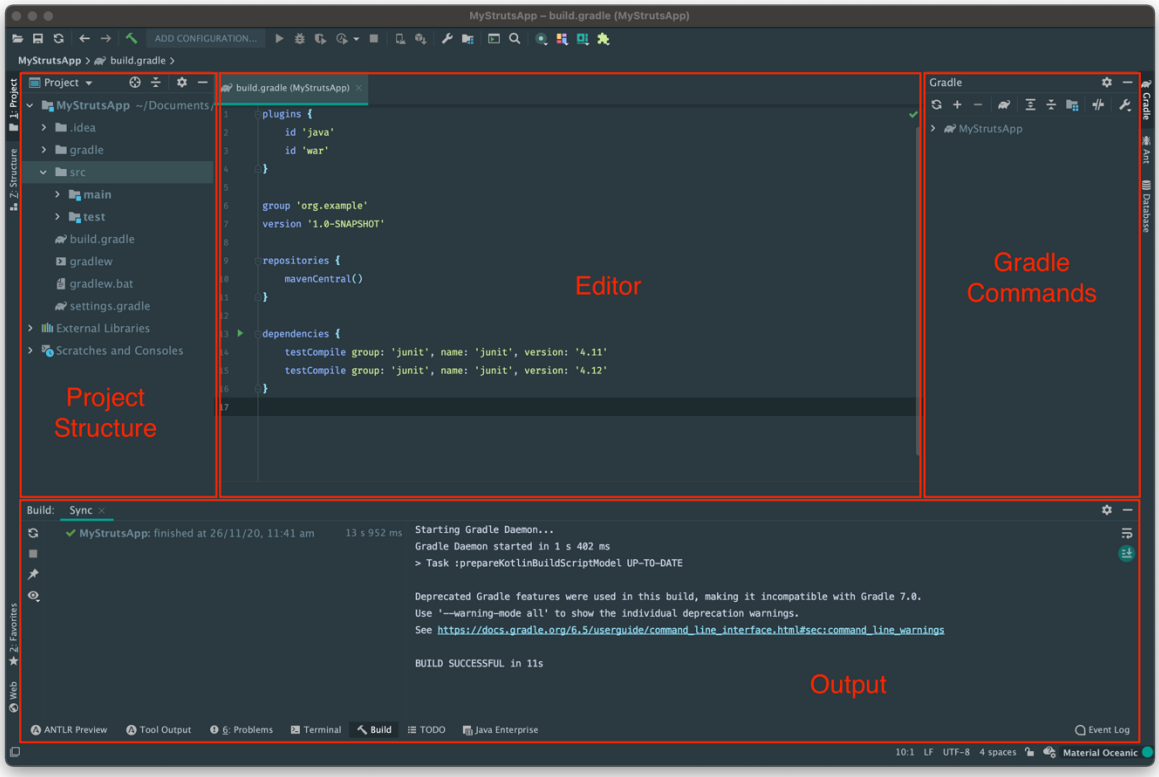
2.1 Start by opening up IntelliJ. At the splash screen click *'New Project'*.

We want to create a 'Gradle' project. Gradle is a Java build tool and dependency manager. This means Gradle will manage compiling and packaging our web application (*i.e. no more javac *.java!)*, but it will also automagically download any framework dependencies. This means we no longer have to manually copy-and-paste .jars into the /WEB-INF/lib directory, nor do we need to update our classpath!



2.2 At the *'New Project'* screen, click 'Gradle' in the sidebar. In the *'Project SDK'* field, select the installation of the Java Development Kit (make sure it is at least Java 11). If no JDK is found, you'll need to add one manually. See lab 01 for the location of the JDK installation. Finally, we need to click the 'web' checkbox. This will tell Gradle to package our application as a web application archive (a 'WAR' file).

2.3 Click next and give the application a name (e.g. MyStrutsApp). Make sure it is saved in your U:/ drive (at UON), and click *'Finish'*. Don't worry about the *'Artifact Coordinates'* fields.



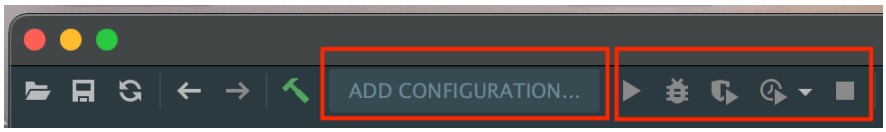
IntelliJ will then open up a new project. It will start by downloading a copy of Gradle, and then proceed to 'index' the project. This may take a few minutes, **wait before continuing.**

On the left we have the project outline. In the middle we have the editor. By default, it will open to the Gradle configuration file. On the right we have the Gradle commands (this may be hidden – click Gradle on the right to view). And at the bottom we have the output.

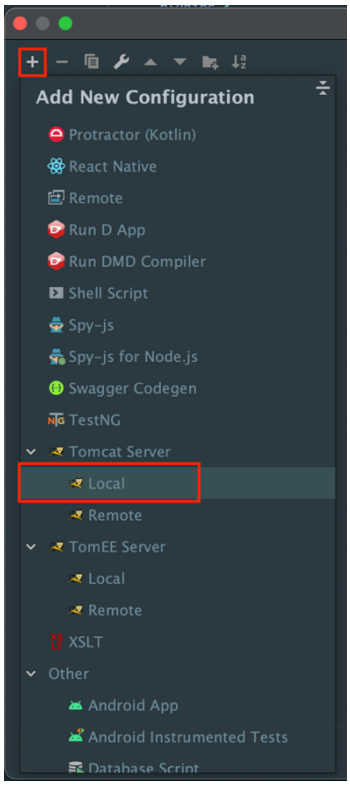
In the project outline, expand the 'src > main' folder. In here we have three content roots: java, resources, and webapp. The java folder is where we place .java files (in packages). Gradle will automatically compile them and place them in the WEB-INF/classes directory. The resources directory is used for Java classpath resources (not web resources). The webapp directory is our web root. It corresponds to the folder we create in the Tomcat webapps directory. Anything in here is 'public', hence we can place static resources and our JSPs here.

2.4 Create new folders in the webapp directory named WEB-INF and META-INF. In here we will create our web.xml, and if needed, our context.xml. But, we NEVER need to place .java or .jar files in the WEB-INF directory – this is handled by Gradle.

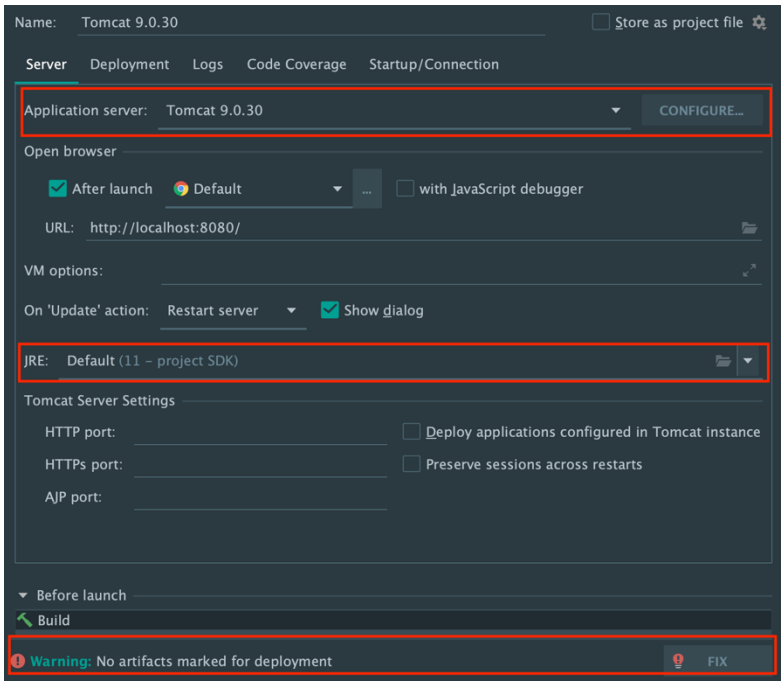
2.5 Finally, we can setup a 'run' target for our web project. A run target will tell IntelliJ how to compile and run our application. As part of this process, we tell IntellJ where tomcat is, how Gradle should build the project, and how it should be deployed.



To create a run target, click *'add configuration'*. A new window will open. Click the plus button in the top-right-hand corner of the screen. A new popup will appear. Scroll down and click *'Tomcat Server > Local'*. We can now start configuring the application deployment.

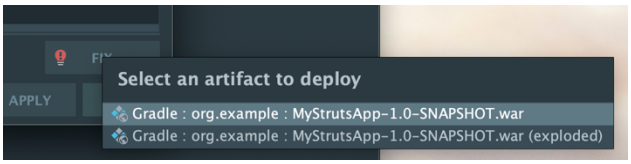


There are a few options we need to configure before it can run.



2.5 We need to tell IntelliJ where our Tomcat installation is. Next to the *'Application Server'* option, click '*configure'.* Click the plus in the top left hand corner to add a new installation. In the popup, click the browse option next to *'Tomcat Home'*, and find your Tomcat folder in the file explorer. This needs to be the root of the Tomcat installation. The warnings will disappear when selected. Click Ok to save the server, and make sure it is selected in the deployment configuration screen.

2.6 Make sure the *'JRE'* option is set to the project SDK. If you have additional Java installations, this may default to a JRE. This can cause problems!



2.7 We need to tell IntelliJ how to deploy our application. When creating the project, we told Gradle to make a web application project. This means Gradle will compile all sources, and place them into a properly formatted Java web application that can be deployed to Tomcat. This is effectively as a Zip file, that gets placed into the Tomcat webapps directory. Start by clicking on the *'Fix'* button at the bottom on the screen. It will ask to *'Select an artifact to deploy'*. Click on the first option. This is the web application archive produced by Gradle.

We also need to tell IntelliJ what *'Application Context'* to deploy the application to. This is the URL prefix we give to our application, based on its name in the webapps directory. By default it will be a hideous mangled name. While still on the *'Deployment'* tab, change the *'Application Context'* to *'/MyStrutsApp'* (or similar). Close the configuration dialogue by clicking 'Ok'.

2.8 Once completed, we are ready to run the application. Deploy the application by clicking the green Play button in the IntelliJ toolbar. IntelliJ will startup Tomcat, and then open your browser to the web application. By default it will show the index.jsp file in the webapp directory (if this is missing – create a dummy index.jsp). This will have a placeholder *'END'.* To stop the application, simply click the red square.

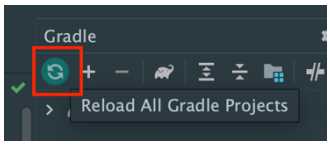
3. A Simple Servlet

To get a feel for using IntelliJ, we can create a test Servlet along with a web.xml file to get our application started. This is to make sure IntelliJ is compiling our sources correctly, and that everything is configured for web development.

The first thing we need to do is tell Gradle what dependencies our project has. Any dependencies we have not provided as part of the JDK need to be explicitly declared as part of the JDK. The first dependency we have for any Java web app is the servlet-api. This is the .jar file we manually add to the CLASSPATH environment variable in lab01.

3.1 Start by opening up the build.gradle file. This file is a Groovy configuration file – don't worry about what this means. All we need to know is that we declare dependencies in the *'dependencies { … }'* block of this file. Update this block to look like the following:

```
dependencies {
    providedCompile group: 'javax.servlet', name: 'javax.servlet-api', version: '4.0.1'
    compile group: 'javax.servlet.jsp.jstl', name: 'jstl-api', version: '1.2'
}
```



This simply declares that our application is dependent upon the servlet API, as well as JSTL. When saved, IntelliJ may tell you that it needs to re-import the project. This means IntelliJ will 'update' what libraries it is aware of that our application uses. In the Gradle sidebar click the reload button to force this.

3.2 We can now create the web.xml file. Right-click the 'webapp > WEB-INF' directory, click 'new > file' and name it web.xml. Copy over the contents from a previous lab exercise.

3.3 Finally, create the Servlet. Right click on the 'main > java' directory and select 'new > Java Class'. Name this class *'SimpleServlet'* and make it extend *HttpServlet*. Add the *@WebServlet* annotation to this class, and map it to '/test'. Override the *doGet()* method and make it print out a simple message.

As you type, notice the auto complete feature. You will not have to type out manual class imports. Simply start typing, find the correct auto complete, and IntelliJ will handle the rest!

Re-run your application and navigate to your servlet in the browser (e.g. localhost:8080/MyStrutsApp/test). If you can see your simple message, everything is setup correctly to proceed to configuring STRUTS!

4. Setting Up STRUTS2

Setting up STRUTS is roughly a 4-step process. 1. Declare the STRUTS2 dependency. 2. Apply the STRUTS2 filter. 3. Create the STRUTS2 configuration file. 4. Create your actions.

4.1 STRUTS2 is simple as the core of STRUTS has only has 1 dependency. Add the following to the dependencies block of the build.gradle file:

```
compile group: 'org.apache.struts', name: 'struts2-core', version: '2.5.25'
```

Then reload the Gradle configuration, so IntelliJ knows we have a dependency on STRUTS2.

4.2 In the web.xml, we need to activate the STRUTS2 framework. This is by applying a 'Filter' to our application. Add the following filter and filter-mapping declaration to the web.xml file:

```
<web-app ...>
        ...

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

        <filter-mapping>
                <filter-name>struts2</filter-name>
                <url-pattern>*.action</url-pattern>
        </filter-mapping>
</web-app>
```

This configuration firstly declares the STRUTS2 dispatch filter, and secondly applies this filter to all URLs that end in '.action'. This is a special extension that tells the STRUTS2 filter to try and find an Action to execute.

4.3 Next, we need to create a struts configuration file. In the 'src > main > resources' directory, create a file 'struts.xml'. Add the following content:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC "-//Apache Software Foundation//DTD Struts
Configuration 2.5//EN" "http://struts.apache.org/dtds/struts-2.5.dtd">
<struts>
    <constant name="struts.devMode" value="true"/>
    <constant name="struts.action.extension" value="action" />

    <!-- STRUTS2 groups actions into packages. A package may have a URL prefix -->
    <package name="default" extends="struts-default">

        <!-- Define interceptors here -->

        <!--The default actions which we execute -->
        <default-action-ref name="index" />

        <!-- Define actions here -->

    </package>
</struts>
```

4.4 Finally, we can create a STRUTS2 Action. The Action is the Controller component of the MVC architecture. Later we will create a View and Model.

We will start by creating a simple 'Hello, World' style action displayed on the index page. In 'src > main > java', create a new package called 'app'. Then create a Java class called IndexAction. This class needs to extend *ActionSupport*, and override a method named *execute()*. The *execute()* method will return result code constant named SUCCESS, as below.

```
public class IndexAction extends ActionSupport {
    @Override public String execute() throws Exception {
        return SUCCESS;
    }
}
```

A STRUTS action will control how a response is generated through these result codes. Common result codes include "success", "error" and "input". Success means the action executed successfully, error indicates an error occurred, while input indicates there is either a problem with or missing inputs to the Action. We configure how result codes are transformed to responses later in the struts.xml file.

4.5 Next create a corresponding view to be displayed by this action. In the 'webapp > WEB-INF' directory, create a new folder named 'jsps'. In the jsps folder, create a new jsp file named 'index-success.jsp'. Add some sample text (e.g. a <h1>) to display a message to the user.

With STRUTS its good practice that views reflect the action name and result code they are displayed with. In this case, we have a view for the 'index' action, that is displayed upon the 'success' result code.

4.6 Finally, we need to configure the action mapping. With servlets, we map them to URLs with the @WebServlet annotation. STRTUS2 takes a different approach in using an xml file to

map actions to URLS, and defining what pages to display based on result codes. Open up the struts.xml file and add the following code below the <!-- define actions here --> comment:

```
<action name="index" class="app.IndexAction">
    <result name="success">/WEB-INF/jsps/index-success.jsp</result>
</action>
```

This code configures an action named 'index' (mapped to '/index.action'), backed by class 'app.IndexAction'. When this action returns the 'success' result code, it displays a JSP named 'index-success.jsp'. With this complete, we can now run the application and view the action at 'localhost:8080/MyStrutsApp/index.action'. Note, STRUTS also provides annotation-based configuration as an optional plugin, as seen with Servlets.

4.7 In order to view this page we need to explicitly navigate to /index.action. Even if we delete the overriding index.jsp file (defined as the 'welcome page'), Tomcat will not be able to use the index action as the default page. However, in the index.jsp file, we can use a work around to redirect all traffic from the default page to the index page. Clear out the index.jsp file, and replace *all* content with:

```
<%
  response.sendRedirect(request.getContextPath() + "/index.action");
%>
```

Restart the application. Now, index.action will be displayed as the default page.

4. More Advanced STRUTS Actions

STRUTS2 offers a lot of 'nice' features not present when using plain Sevlets & JSPs. Struts can automatically convert and inject request parameters into your actions or bean models, as well as validating parameters. To demonstrate this, we will create a user registration form.

4.1 Start by creating a new action class named 'RegisterAction', making it extend *ActionSupport* and overriding *execute()*. This action will have the following properties (i.e. fields with corresponding getters and setters):

- String name
- int age
- String email
- String password
- String passwordConfirm

Hint: After declaring the 5 fields, in the editor right-click, click *'Generate..'*, select *'Getters & Setters'*, select all 5 fields, and click ok. This will automagically generate getters and setters!

In the execute method, all we have to do is return the SUCCESS constant. If this was a real registration action that inserted a user into a database, this is where we would implement this feature. *Complete this as a directed task after this tutorial*!

4.2 Next we can create two JSP pages in the 'jsps' directory. One page will show a registration form ('register-input.jsp'), and another will be a success/welcome page ('register-input.jsp').

In register-input.jsp, place the following form:

```
<h1>Register</h1>
<s:form action="register">
    <s:textfield name="name" label="Your Name" />
    <s:textfield name="email" type="email" label="Your eMail" />
    <s:textfield name="age" type="number" label="Your Age" />
    <s:textfield name="password" label="Password" />
    <s:textfield name="passwordConfirm" label="Confirm Password" />
    <s:reset />
    <s:submit />
</s:form>
```

When typing this out, IntelliJ will add the appropriate taglib import. Then, in register-success.jsp, place the following code:

```
<h1>Registered</h1>
<p>
    Welcome, <s:property value="name" default="<unknown>" />!
</p>
```

Notice in both we use special s: prefixed tags. These are the struts tag library that integrate nicely with the Struts framework. In the following sections we will use more of these tags. You may use standard JSTL tags (especially the c: tags), however, these integrate nicely.

The s:property tag is similar to the c:out tag. However, is specifically is used to retrieve values from a STRUTS action. Any field exposed through a getter on the action can be printed out using the s:property tag.

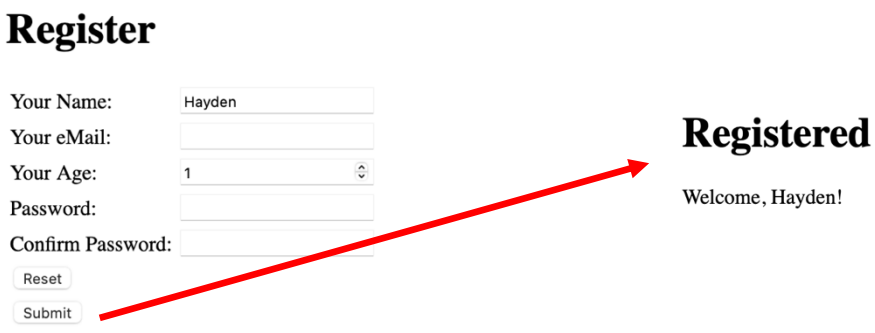
The s:textfield tag automatically creates a text input, with a name corresponding to the actions properties. An action has to expose a corresponding setter to access the passed value.

4.3 Add the following mappings in struts.xml (after the index action):

```
<action name="registration">
    <result name="success">/WEB-INF/jsps/register-input.jsp</result>
</action>

<action name="register" class="app.RegisterAction">
    <result name="success">/WEB-INF/jsps/register-success.jsp</result>
    <result name="input">/WEB-INF/jsps/register-input.jsp</result>
</action>
```

Notice we don't declare a class for the input-register page. Struts can use a default action that always returns success to allow us to map JSPs without a backing Action. Run the application, and navigate to '/registration.action'. This will display the form. Fill it in and click submit. If all is configured correctly, you should see the welcome message.



4.4 So far, this application lets us fill in a form & submit data. But it doesn't include any validation. There is nothing stopping a user from submitting total garbage! Thankfully, Struts2 implements a comprehensive validation framework.

Validation can be configured automatically with annotations, XML, or procedurally in code. In our form, we want to have the following constraints:

- All fields are required
- eMail must be a valid email
- age must be greater than 18, but less than 100
- password and confirmPassword must be equal

The first three can be achieved using validation annotations. Validation annotations are placed on setters, and are used to enforce constraints. If the validation of any field fails, the action will implicitly return the *'input'* result code. Input is a standard struts result indicating the input is invalid. When this occurs on the RegisterAction, we have configured it to display the input form.

On all setters, add the following annotation:

```
@RequiredStringValidator(message = "<field> is required") // Update <field> to the correct name
```

On the *age* setter, add the following annotation:

```
@IntRangeFieldValidator(min = "18", max = "100", message="Age must be between 18 and 100")
```

On the *email* setter, add the following annotation:

```
@EmailValidator(message = "email is not a valid email")
```

With these three annotations, we can achieve 3 of the 4 validation constraints. However, the final constraint is much more complex. It requires comparing two parameters, something not supported through annotations. To get around this, we can procedurally implement some validation logic. If an action extends *ActionSupport*, we can override a method called *validate()*. Validate is called to procedurally validate our actions. After the execute method, add the following code:

```
@Override
public void validate() {
    if (!Objects.equals(this.password, this.passwordConfirm)) {
        this.addFieldError("passwordConfirm", "Password and confirmation do not match");
    }
}
```

Note: Object.equals is a null safe version of using the equals method – it stops us having to check for null before calling equals()!

We can now run the application, and see what happens. If we try to submit valid input, error messages appear! It's not pretty, but can be easily fixed with some CSS! But even more importantly, the field values are carried through from the first request to the error page. This is automatic – we don't have to manually print these values out!

4.5 All of our code works, but it isn't really MVC. Our action has bound parameters, that really belong in a model bean. Fortunately, Struts has first-class support for beans. We can easily update our code to declare that this action depends upon a model bean, and have struts directly inject parameters into it.

Start by declaring a new Java class named RegisterBean. Move the 5 fields and their getters/setters to this bean. In the RegisterAction class, create a new private field:

```
private RegisterBean model = new RegisterBean();
```

Make the RegisterAction implement an interface named ModelDriven

```
public class RegisterAction extends ActionSupport implements ModelDriven<RegisterBean> {
```

Declare a new method *getModel()* that returns the newly created bean (required by the ModelDriven interface):

```
@Override
public RegisterBean getModel() { return model; }
```

And update the validate method to validate the fields on the bean, and not the action:

```
if (!Objects.equals(this.model.getPassword(), this.model.getPasswordConfirm())) {
```

If we try to run the application now, everything will work … but the validation. Validation can only be implemented on the setter methods of an Action – it cannot be implemented on bean setters!

4.6 Fortunately, there is a work-around for this. On our execute method, we can define a @Validations annotation to specify *all* validations for this specific action. Start by removing all validations from the bean fields, and add the following annotation to the *execute* method:

```
@Validations(
        requiredStrings = {
                @RequiredStringValidator(fieldName = "name", message = "name is required"),
                @RequiredStringValidator(fieldName = "age", message = "age is required"),
                @RequiredStringValidator(fieldName = "email", message = "email is required"),
                @RequiredStringValidator(fieldName = "password", message = "password is
required"),
                @RequiredStringValidator(fieldName = "passwordConfirm", message =
"confirmPassword is required")
        },
        intRangeFields = {
                @IntRangeFieldValidator(fieldName = "age", min = "18", max = "100", message =
"age must be between 18 and 100")
        },
        emails = {
                @EmailValidator(fieldName = "email", message = "email is not a valid email")
        }
)
```

Now if we re-run the application, everything will work as before! As a benefit, the validation logic is much more condensed here. It is good practice to place all the validation logic in one place. However, keep in mind, there are always exceptions to the rule, and not all validation can be implemented by annotations!

5. Interceptors

STRUTS 2 allows the definition of interceptors that run before and after the execution of an action. By default, STRUTS includes a comprehensive set of interceptors that provide many of the features we have used so far. For example, STRUTS uses an interceptor to call the 'validate' method, and to perform any annotation-based validation on our actions. Creating a custom interceptor is as simple as creating a new class, extending the *'AbstractInterceptor'* class, and configuring it to run around our actions. We will use an interceptor to implement some basic logging in our application.

5.1 Start by creating a new class called *'LoggingInterceptor'*. Make it extend the *'AbstractInterceptor'* class and implement the required method *'intercept'*. Hint: IntelliJ can automatically implement this method. Hover over the name 'AbstractInterceptor', and IntelliJ will provide an auto-complete option.

5.2 In this method we want to log before and after an action has executed. We will also print out the class name of the Action, and the URL the action is bound to. Replace the contents of the intercept method with the following:

```
@Override
public String intercept(ActionInvocation invocation) throws Exception {
    Object action = invocation.getAction();
    String className = invocation.getAction().getClass().getName();
    String name = invocation.getProxy().getActionName();

    System.out.printf("Action: %s, Name: %s\n", className, name);
    System.out.println("Before invoke");
    String result = invocation.invoke();
    System.out.println("After invoke");

    return result;
}
```

5.3 Finally, we need to configure the application to use the interceptor. Interceptors can be applied to individual actions, or globally to *all* actions. As logging is useful to be applied to all actions, we will apply it globally. Just after the interceptors marker in the struts.xml file, place the following code:

```
<interceptors>
    <interceptor name="myLogging" class="app.LoggingInterceptor"></interceptor>
    <interceptor-stack name="myStack">
        <interceptor-ref name="defaultStack"></interceptor-ref>
        <interceptor-ref name="myLogging"></interceptor-ref>
    </interceptor-stack>
</interceptors>

<default-interceptor-ref name="myStack"></default-interceptor-ref>
```

Within <interceptors>, we define both individual interceptors, and interceptor stacks. An interceptor stack is a reusable set of interceptors. In this example, we define our interceptor

(named myLogging), and then a stack of interceptors, containing the default stack (i.e. all STRUTS provided interceptors, followed by our interceptor. We then define the default set of interceptors to be applied to all actions with the <default-interceptor-ref> tag. If we were to apply our interceptor to an individual action, we can simply place an `<interceptor-ref name="…" />` tag INSIDE an action tag, just before the results.

When we run the application, in the console you should see output similar to:

```
Action: app.IndexAction, Name: index
Before invoke
After invoke
```

This means our interceptor is running around our action. If we place a print statement inside the action, we could see something similar to:

```
Action: app.IndexAction, Name: index
Before invoke
Inside index
After invoke
```

Directed Tasks

1. Currently, when we register, nothing happens. Integrate some JDBC code from the previous lab and insert a new user into a simple database.

2. This is a very simple application. But it is packed with features provided by the Struts framework. How hard would it be to re-implement this application with only servlets JSPs? Try writing out the form validation in standard Java code. How much time did we save?

3. STRUTS interceptors are very powerful. They can be used to implement much more useful features than simple logging. A useful feature could be the injection of a JDBC connection into your action. For example, before invoking the action, retrieve a connection from the JNDI connection pool. After executing the action, you can then close the pool. To inject the connection, declare an interface named 'ConnectionAware' with a single method 'setConnection(Connection)'. The interceptor will check to see if the action implements ConnectionAware, and if so, call setConnection on the action; before closing the connection after forwarding the invocation.