Mastering SAP Technologies

SAP HANA Cloud Platform - Workshop Exercise Document

Any questions relating to this tutorial should be addressed to:

Matthias Steiner
mailto:matthias.steiner@sap.com

Author	Matthias Steiner
Version	1.0
Date of Issue	21-05-2015

Table of Contents

Document Control

Version	Section	Description	Editor	Date
0.1		Initial draft	Matthias Steiner	09-03-15
1.0		First release version	Matthias Steiner	21-03-15

This Document

This document contains the detailed exercises for the workshop:

Masterclass: Everything You Need to Know to Get Started With Developing Applications Using SAP HANA Cloud Platform

Contributors

The workshop content is based on various samples provided as part of the SAP HANA Cloud Platform SDK. The UI coding is loosely based on a trimmed-down version of the Master/Details template provided by the OpenUI5 SDK. Special shout-out and thanks to Katharina Seiz for helping me in coming up with the most trimmed-down and lightweight code possible.

The Workshop

This workshop is a hands-on exercise. You'll start from zero and step-by-step develop a fully operational weather application. In total there are 12 chapters/exercises, each building on top of its predecessor. The entire source code of both the final and all intermediate states are available on Github:https://github.com/SAP/cloud-weatherapp

In case you should get lost along the way, you want to skip a specific step or simply want to compare your local version with the expected state you can download (or clone) a specific version using the tagged releases: https://github.com/SAP/cloud-weatherapp/releases

You may also be interested in having a closer look at the commit log as it makes it easy to see the changes applied from one chapter to the next.

Workshop Pre-Requisites

This document assumes that you have a properly installed and configured Eclipse IDE and SAP HANA Cloud Platform SDK (1.x) available. This process is thoroughly explained in the following two tutorials available on the Developer Page of the central SAP HANA Cloud Platform website: http://hcp.sap.com

1. Getting Started with the SAP HANA Cloud Platform Tools for Java
2. Cloning and Running a Maven-based SAP HANA Cloud Platform Project from GitHub

01 – Hello World

Objective

We'll start with a classic "Hello World" project to warm up and to ensure that both Eclipse IDE and the local SAP HANA Cloud Platform (HCP) tooling have been properly installed and configured.

Instructions

1. Create a new dynamic web project by selecting the "**New > Dynamic Web Project**" menu entry.
2. Enter the following information:

Name	weatherapp
Target Runtime	Java Web
Dynamic Web Module Version	2.5

3. Click on <*Next*>.
4. Delete the standard "**src**" **Source folder** and add a new one called "**src/main/java**" to create a project that adheres to the standard Maven Directory Layout.
5. Change the **default output folder** to "**target/classes**".
6. Click on <*Next*>.
7. Change the **Content Directory** from "**WebContent**" to "**src/main/webapp**" (again, to adhere to Maven conventions.)
8. Click on <*Finish*>.
9. Create a new Servlet by selecting the "**New > Servlet**" menu entry.
10. Enter the following information:

Package name	com.sap.hana.cloud.samples.weatherapp.web
Class name	HelloWorldServlet

11. Click on <*Next*>.
12. Change the **URL Mapping** from "**/HelloWorldServlet**" to "**/hello**" to make it a bit easier to memorize.
13. Click on <*Finish*>.

14. Now we need to do our first lines of coding. Navigate to the servlet's "**doGet()**" method and replace the TODO comment with the following line of code:

```
response.getWriter().println("Hello World!");
```

15. Save your changes.

16. Deploy the application to your local server by using the "**Run as > Run on Server**" context menu of the **HelloWorldServlet** node in the **Project Explorer** view.

17. Choose the "**Manually define a new Server**" option and select the "SAP / Java Web Server" option from the server selection. Make sure to select "Java Web Server" as the **server runtime environment**.

18. Click on *<Finish>*. The internal browser is now started and displays the traditional message marking the first step into a new programmer's journey.

02 – Adding a simple HTML page

Objective

The next step is to add a simple index HTML page.

Instructions

1. Navigate to the "**webapp**" node in the Project Explorer and create a new HTML page by selecting the respective context menu entry: "**New > HTML File**".
2. Choose the name "**index.html**" and click on <*Finish*>.
3. Add the following line of code in between the opening and the closing <body> tag:

```
<p>Hello World!</p>
```

4. Save your changes and publish the update project to the local server by selecting the respective context menu on the server in the **Server** view.
5. Once the publishing is done you can go back to the internal browser window and remove the "/hello" part of the URL and hit <*Enter*>. Now, you should see the same "Hello World!" message for the URL http://localhost:8080/weatherapp/ as well. The reason for this is the <**welcome-file-list**> in the **web.xml** (located in the **WEB-INF** directory underneath the **webapp** folder).
6. For security reasons and for the sake of housekeeping you should remove all **<welcome-file>** entries except for the "**index.html**" one.

03 – Adding Authentication

Objective

In this chapter we'll implement both authentication and authorization.

Note: Please note that HCP adheres to Java standards to manage these aspects!

Instructions

1. In order to activate authentication and establish authorization we have to apply the respective security settings in the **web.xml** configuration file:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://java.sun.com/xml/ns/javaee"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" id="WebApp_ID"
version="2.5">
<display-name>cloud-weatherapp</display-name>
<welcome-file-list>
    <welcome-file>index.html</welcome-file>
</welcome-file-list>
<servlet>
    <display-name>HelloWorldServlet</display-name>
    <servlet-name>HelloWorldServlet</servlet-name>
    <servlet-class>
        com.sap.hana.cloud.samples.weatherapp.web.HelloWorldServlet
    </servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>HelloWorldServlet</servlet-name>
    <url-pattern>/hello</url-pattern>
<servlet-mapping>
<login-config>
    <auth-method>FORM</auth-method>
</login-config>
<security-constraint>
    <web-resource-collection>
        <web-resource-name>Protected Area</web-resource-name>
        <url-pattern>/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
        <!-- Role Everyone will not be assignable -->
        <role-name>Everyone</role-name>
    </auth-constraint>
```

```
</security-constraint>
<security-role>
    <description>All SAP HANA Cloud Platform users</description>
    <role-name>Everyone</role-name>
</security-role>
</web-app>
```

2. With this configuration we have established the rule that everyone needs to authenticate prior to navigating to any of the protected web resources. Since we have applied a wildcard mapping for the pre-defined "**Everyone**" role (which every user has by default) everyone can navigate to any resource of the application.

3. After successful authentication the application can access users principal information using standard servlet APIs. To illustrate that make the following changes to the **HelloWorldServlet**:

```
protected void doGet(HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException
{
    String user = request.getRemoteUser();

    if (user != null)
    {
        response.getWriter().println("Hello, " + user);
    }
    else
    {
        LoginContext loginContext;

         try
        {
            loginContext =  ⏎
LoginContextFactory.createLoginContext("FORM");

            loginContext.login();
            response.getWriter().println("Hello, " +  ⏎
request.getRemoteUser());
        }
        catch (LoginException ex)
        {
             ex.printStackTrace();

         }
    }
}

protected void doPost(HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException
{
    doGet(request, response);
}
```

Note: The reason we also had to implement the "**doPost()**" method is related to specifics of the SAML 2.0 authentication process flow. For more information please refer to the respective parts of the SAP HANA Cloud Platform online documentation.

4. You need to **organize import** statements via the respective context menu ("**Source > Organize imports**") of the main code editor window.
5. Save your changes.
6. Deploy/publish the updated application (you should know the drill by now).
7. Since we are working with a local server so far we need to provide a local user repository to authenticate against. For this purpose, double-click on the local server node in the **Servers** view to open the configuration window.
8. At the bottom of that window there are four tabs: **Overview**, **Connectivity**, **Users** and **Loggers**.
9. Within the **Users** tab you can manage local users. Let's create a simple test user with the user id "test" and a password according to your likings.
10. Save your changes.
11. Now, when you navigate to the HelloWorldServlet with the URL http://localhost:8080/weatherapp/hello you'll first be promted to enter your user credentials before you are forwarded to the requested servlet. If the authentication was successful you should now see a personalized welcome message instead of the dull "Hello World!" we saw earlier.

04 – Introducing Maven

Objective

So far we only did very basic things, yet real-life applications are more complex. In order to prepare for the added complexity – especially in the area of dependency management – we'll convert our project to a Maven-based one.

Instructions

1. Select the "**weatherapp**" node in the project explorer and open the context menu. Select the "**Configure > Convert to Maven Project**" option.

2. Change the group ID from "**weatherapp**" to "**com.sap.hana.cloud.samples**" and click on *<Finish>*. The most noticeable change will be that a "**pom.xml**" file will be created in the root folder of the "**weatherapp**" project.

3. Copy the entire content of the "**pom.xml**" file from Github and use it to replace the existing coding in your local copy.

4. Now, open the context menu on the "**weatherapp**" project in the **Project Explorer** and select the menu entry "**Maven > Update Project…**" (The first time you do this can take a bit longer, as Maven is now downloading all the required build plugins and dependencies specified in the "**pom.xml**".)

5. Next, select the "**Run as > Maven build…**" context menu of the "**weatherapp**" project and enter the following **Goals** (without the quotes): "clean package install".

6. Click on <Run> - the project should build successfully.

05 – RESTful Service Provisioning

Objective

In this chapter, we'll explore how-to expose services in a RESTful manner. For this purpose we are using a library called Apache CXF, which is one of the most often used implementations of the JAX-RS standard.

Instructions

1. First, we need to add the dependency references to Apache CXF to the "**pom.xml**" file:

```
<!-- Apache CXF -->
<dependency>
    <groupId>org.apache.cxf</groupId>
    <artifactId>cxf-rt-frontend-jaxws</artifactId>
    <version>${org.apache.cxf-version}</version>
</dependency>
<dependency>
    <groupId>org.apache.cxf</groupId>
    <artifactId>cxf-rt-frontend-jaxrs</artifactId>
    <version>${org.apache.cxf-version}</version>
</dependency>

<dependency>
    <groupId>javax.ws.rs</groupId>
    <artifactId>javax.ws.rs-api</artifactId>
    <version>2.0</version>
</dependency>
```

2. We also need to specify the corresponding CXF version property at the end of the <properties> tag:

```
    <properties>
        <java-version>1.7</java-version>
        <project.build.sourceEncoding>UTF-
8</project.build.sourceEncoding>
        <org.slf4j-version>1.7.2</org.slf4j-version>
        <com.sap.cloud.neo.api-version>1.76.13</com.sap.cloud.neo.api-
version>

        <org.apache.cxf-version>3.0.0</org.apache.cxf-version>
    </properties>
```

3. Now, let's create a new Class via the corresponding context menu entry "**New > Class**" of the "**weatherapp**" node in the **Project Explorer**. Enter the following information:

Package name	com.sap.hana.cloud.samples.weatherapp.api
Classname	AuthenticationService

4. Click on <*Finish*>.

5. Replace the default coding with the following content:

```
package com.sap.hana.cloud.samples.weatherapp.api;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.SecurityContext;

@Path("/auth")
@Produces({ MediaType.APPLICATION_JSON })
public class AuthenticationService
{
    @GET
    @Path("/")
    @Produces({ MediaType.TEXT_PLAIN })
    public String getRemoteUser(@Context SecurityContext ctx)
    {
        String retVal = "anonymous";
        try
        {
            retVal = ctx.getUserPrincipal().getName();
        }
        catch (Exception ex)
```

```
        {
            ex.printStackTrace(); // lazy
        }
        return retVal;
    }

}
```

6. Save your changes.

7. Open the "**web.xml**" configuration file and copy & paste the following lines of code in between the closing <servlet-mapping> and the opening <login-config> tags:

```
    <servlet>
        <servlet-name>CXFServlet</servlet-name>
        <servlet-class>
          org.apache.cxf.jaxrs.servlet.CXFNonSpringJaxrsServlet
        </servlet-class>
        <init-param>
            <param-name>jaxrs.serviceClasses</param-name>
            <param-value>
com.sap.hana.cloud.samples.weatherapp.api.AuthenticationService
            </param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>CXFServlet</servlet-name>
        <url-pattern>/api/v1/*</url-pattern>
    </servlet-mapping>
```

8. With this, we have registered (Apache) CXF as a Servlet that listens to incoming requests using the **URL-pattern**: **"/api/v1/***". Furthermore, we registered our **AuthenticationService** class as one of the RESTful services. During startup, CXF will introspect the class and use the provided JAX-RS annotations to properly configure our service.
9. Save your changes and publish/deploy your application.
10. Navigate to the following URL: http://localhost:8080/weatherapp/api/v1/auth. After successful authentication you should see your username.

06 – JPA-based persistence

Objective

Now, we will implement a very simple domain model and implement the corresponding persistence layer using JPA. Our domain model only features a single class: **FavoriteCity**, so that we can bookmark/favorite our favorite cities.

Instructions

1. Let's create a base class for our domain model first. That's usually considered best practices as it provides us with a central place to add common functionality to be shared across all domain model objects on later. For that purpose, open the respective context menu entry on the "**weatherapp**" project again and provide the following details:

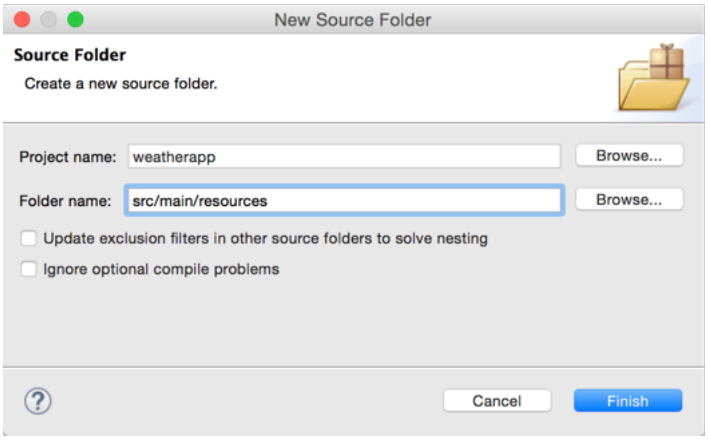
Package name	com.sap.hana.cloud.samples.weatherapp.model
Classname	BaseObject

1. Replace the default coding with this one. Save your changes.
2. Let's create the **FavoriteCity** class respectively**:**

Package name	com.sap.hana.cloud.samples.weatherapp.model
Classname	FavoriteCity

3. Replace the default coding with this one. Save your changes.
4. Next, we need to create a configuration file for our persistence layer. By Maven coventions, these non-source code artifacts should be located in a separate source code folder called: "**src/main/resources**". Hence, let's create that source folder via the corresponding context menu entry on the "**Java Resources**" node in the **Project Explorer**: "**New > Source Folder**".
5. Provide the following information:

Project name	weatherapp
Folder name	src/main/resources



6. Open the context menu of this newly created source folder and choose the "**New > Other**" option and then select the **Folder** option. Name the new folder "**META-INF**" (all capitals!) and click on <*Finish*>.
7. Open the context menu of the newly created "**META-INF**" folder and select "**New > File**". Name the new file "**persistence.xml**" and click on <*Finish*>.
8. Copy & paste the following coding:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
    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_2_0.xsd">

    <persistence-unit name="application" transaction-
type="RESOURCE_LOCAL">
    <provider>
        org.eclipse.persistence.jpa.PersistenceProvider
    </provider>
    <class>
        com.sap.hana.cloud.samples.weatherapp.model.BaseObject
    </class>
    <class>
        com.sap.hana.cloud.samples.weatherapp.model.FavoriteCity
    </class>
        <exclude-unlisted-classes>true</exclude-unlisted-classes>
        <properties>
            <property name="eclipselink.ddl-generation" value="create-
tables"/>
        </properties>
    </persistence-unit>
</persistence>
```

9. Next, we need to add some more dependencies to our "**pom.xml**" file. In this case, the most important dependency is to EclipseLink (our JPA implementation of choice). However, we also need to declare dependencies for the Derby DB and Jackson (a serialization framework needed to convert data into JSON and vice versa.)

```
<!-- EclipseLink (and JPA) -->
<dependency>
        <groupId>org.eclipse.persistence</groupId>
        <artifactId>eclipselink</artifactId>
        <version>2.5.0</version>
    </dependency>
     <dependency>
        <groupId>org.eclipse.persistence</groupId>
        <artifactId>javax.persistence</artifactId>
        <version>2.1.0</version>
    </dependency>
    <!-- Derby -->
    <dependency>
        <groupId>org.apache.derby</groupId>
          <artifactId>derbyclient</artifactId>
          <version>10.9.1.0</version>
    </dependency>
    <dependency>
        <groupId>org.apache.derby</groupId>
        <artifactId>derby</artifactId>
        <version>10.9.1.0</version>
    </dependency>

    <dependency>
        <groupId>javax.ws.rs</groupId>
        <artifactId>javax.ws.rs-api</artifactId>
        <version>2.0</version>
    </dependency>

    <dependency>
        <groupId>org.codehaus.jackson</groupId>
        <artifactId>jackson-jaxrs</artifactId>
        <version>${org.codehaus.jackson-version}</version>
    </dependency>
```

10. We also need to add the Jackson version as a property in the properties section (as we have done in the previous chapter):

```
<org.codehaus.jackson-version>1.9.9</org.codehaus.jackson-version>
```

11. Save your changes.

12. Now we need to create the respective CRUD service. (Well, in our case we do not really need an update operation, but let's not be too picky!). For that purpose, create a new class with the following details:

Package name	com.sap.hana.cloud.samples.weatherapp.api
Classname	FavoriteCityService

13. Replace the default coding with this one. Save your changes.
14. Last thing we need to do is to register our RESTful service implementation in the "**web.xml**" configuration file (remember?). So, let's add the full-qualified classname of our **FavoriteCityService** class to the (comma-separated) list of "**jaxrs.serviceClasses**":

```
<init-param>
            <param-name>jaxrs.serviceClasses</param-name>
      <param-value>

com.sap.hana.cloud.samples.weatherapp.api.AuthenticationService,

com.sap.hana.cloud.samples.weatherapp.api.FavoriteCityService
            </param-value>
</init-param>
```

15. Underneath the before-mentioned <init-param> tag we need to add another <init-parameter> for JSON de-/serialization as follows:

```
<init-param>
        <param-name>jaxrs.providers</param-name>
        <param-
value>org.codehaus.jackson.jaxrs.JacksonJsonProvider</param-value>
</init-param>
```

16. And while we are at it, we also need to we need to define a so-called DataSource within the "**web.xml**" in order to connect to the underlying database. As such, copy & paste the following code snippet after the closing <welcome-file-list> tag:

```
<resource-ref>
    <res-ref-name>jdbc/DefaultDB</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
</resource-ref>
```

17. Save your changes and deploy/publish your updated application.

18. If you now point your browser to http://localhost:8080/weatherapp/api/v1/cities you'd see two empty brackets "[]" (indicating an empty array) after successful authentication. Don't worry, we haven't saved any cities as favorites yet, so that's just what we would expect.
19. In order to properly test our RESTful service we need a REST tool (e.g. Postman) that allows you to execute HTTP calls in a convenient manner.

07 – Multi-tenancy

Objective

So far, all the users of our weather app would have to share a single favorite list – not very convenient. In this chapter we'll enhance the persistence layer with a multi-tenancy feature.

Instructions

1. First, we'll add the necessary annotations to the persistence **BaseObject** class. Open it and add the following two annotations:

```
@MappedSuperclass
@Multitenant
@TenantDiscriminatorColumn(name = "TENANT_ID",
contextProperty="tenant.id")
public abstract class BaseObject
```

2. Furthermore, we need to slightly adjust the way we obtain a reference to the **EntityManager** within the **FavoriteCityService** as we now need to pass the current tenant ID (in our case the user ID). The following code snippet illustrates the concept:

```
@SuppressWarnings("unchecked")
@GET
@Path("/")
public List<FavoriteCity> getFavoriteCities(
                @Context SecurityContext ctx)
{
        List<FavoriteCity> retVal = null;

        String userName = (ctx.getUserPrincipal() != null) ?
ctx.getUserPrincipal().getName() : "anonymous";

        Map<String,String> props = new HashMap<String,String>();
        props.put("tenant.id", userName);

        EntityManager em =
this.getEntityManagerFactory().createEntityManager(props);

        retVal =
em.createNamedQuery("FavoriteCities").getResultList();

        return retVal;
}
```

3. Save your changes. Also, please change the remaining methods in the **FavoriteCityService** respectively and save all the changes.
4. Deploy/publish your changes. Please explicitly stop and start the server, as we have updated the persistence model!

08 – Consuming external services

Objective

In this chapter you'll learn how-to use the Connectivity Service to connect to an external (e.g. backend) system. In our case, we'll consume a RESTful weather service that returns data in JSON format: http://openweathermap.org/api

Instructions

1. First, we create a new service class with the following properties:

Package name	com.sap.hana.cloud.samples.weatherapp.api
Classname	WeatherService

2. Replace the skeleton coding with this content. Save your changes.

3. Now we need to include the full-qualified classname of the **WeatherService** class in the list of JAX-RS services specified in the "**web.xml**" configuration (remember?). The corresponding tag should now look like this:

```
<init-param>
     <param-name>jaxrs.serviceClasses</param-name>
     <param-value>
com.sap.hana.cloud.samples.weatherapp.api.AuthenticationService,
com.sap.hana.cloud.samples.weatherapp.api.FavoriteCityService,
com.sap.hana.cloud.samples.weatherapp.api.WeatherService
      </param-value>
 </init-param>
```

4. Within the coding we are obtaining a reference to a **HttpDestination** with the logical name "**openweathermap-destination**" via JNDI, hence we need to create that destination. For that purpose, double-click on the local server in the **Servers** view. Switch to the "**Connectivity**" tab and click on the green "+" symbol to add a new destination. Enter the following information:

Name	openweathermap-destination
Type	HTTP
URL	http://api.openweathermap.org/data/2.5/weather

5. Similar to what we have done to register the DataSource in the "**web.xml**", we also need to specify the HTTP destination in the web.xml file. Open it and enter the following code snippet underneath the already existing <resource-ref> tag.

```
<resource-ref>
    <res-ref-name>openweathermap-destination</res-ref-name>
    <res-type>com.sap.core.connectivity.api.http.HttpDestination</res-type>
</resource-ref>
```

6. Save your changes and deploy/publish the application again. After successful authenticating yourself, navigate to the following URL: http://localhost:8080/weatherapp/api/v1/weather?id=2158177

7. One more thing: it would actually be nice to be able to traverse the path and query for weather information in a more RESTful manner via a URL pattern like: **/api/v1/cities/{id}/weather**. Let's add a respective method to the **FavoriteCityService**:

```
    @GET
    @Path("/{id}/weather")
    @Produces({ MediaType.APPLICATION_JSON })
    public String getWeatherInformation(@PathParam(value = "id") String
id, @Context SecurityContext ctx)
    {
        WeatherService weatherService = new WeatherService();
        return weatherService.getWeatherInformation(id, null);
    }
```

Note: Since it is a common cause of frustration I want to point it out explicitly here. In case you are developing behind a (corporate) firewall that requires a proxy for outbound communication you need to start your local server with proxy settings as follows:

- In the Servers view, double-click the added server to open the editor.
- Click the Open Launch Configuration link.
- Choose the (x)=Arguments tab page.
- In the VM Arguments box, add the following row:
  `-Dhttp.proxyHost=<your_proxy_host> -Dhttp.proxyPort=<your_proxy_port> -Dhttps.proxyHost=<your_proxy_host> -Dhttps.proxyPort=<your_proxy_port>`
- Choose OK.

09 - UI

Objective

Now that we have all our services ready we need a corresponding (mobile) user interface. For that purpose we will be leveraging OpenUI5.

Instructions

1. First, let's create a "**weather_app"** folder underneath the **webapp** folder.
2. In that **weather_app** folder we need another sub-folder called "**view**".
3. Next create the following Javascript files:
   - /weather_app/view/App.view.xml
   - /weather_app/view/Details.controller.js
   - /weather_app/view/Details.view.xml
   - /weather_app/view/List.controller.js
   - /weather_app/view/List.viw.xml
   - /weather_app/Component.js
4. Copy & paste the content from Github into the respective files and familiarize yourself with the content of the individual files.
5. Also, make sure to update the content of the **index.html** file to match this version.
6. Publish/deploy your updated app and navigate your browser to the root URL: http://localhost:8080/weatherapp . After successful authentication you should see a fully operational UI.

Note: I'll leave it up to you to beautify the UI by applying formatters and respective icons etc.

10 – Deploying to the cloud

Objective

Our last step will be to deploy the application to the cloud.

Instructions

1. Create a new server by opening up the corresponding context menu on the Servers view: **New > Server**.
2. This time, make sure you select the last entry: SAP HANA Cloud Platform.
3. Make sure to change the **landscape host** to "**hanatrial.ondemand.com"**
4. Click <*Next*>.
5. Provide the following information:

Applictaion name	weatherapp
Account name	<*userid*>**trial**
User name	<*userid*>
Password	<*SCN password*>

6. Click on <*Next*>.
7. Select the weatherapp on the list on the left and add it to the (remote) server by clicking on the <*Add*> button. Then, click on <Finish>.
8. Now create the Http destination for the remote server as we have done in chapter 8.
9. Finally publish the application to the (remote) server and test it. Hopefully it'll work like a charm!