EclipseCON 2009 - March 23rd, 2009

Towards Integrated SOA Development with Eclipse STP and Swordfish

-

SCA Tutorial

Vincent Zurczak – EBM WebSourcing

Table of Contents

1. Introduction

This tutorial aims at introducing the SCA Tools project of Eclipse STP through the development of an SCA application. This application is a simple weather forecast application. It will be designed with the STP SCA tools and deployed on the Apache Tuscany platform.

1.1 - Reminder about SCA

SCA (Service Component Architecture) is a recent set of specifications supported by many Software companies, including some major ones (IBM, Sun, BEA...). The first specifications were produced by the OSOA consortium. And it is now about to be standardized by the OASIS organization.

SCA deals with the creation of applications.
It may be seen as a mix between components and services approaches. SCA applications are made up of components and exposed as services. One could say « components inside, services outside ».

From the user point of view, SCA provides easy and powerful features to develop distributed applications and/or to reuse existing applications (either because they are exposed as services or because legacy code can be used in SCA applications). This is possible because SCA deals with several communication protocols (SOAP, HTTP, JMS...) and several programming languages (Java, C++, Cobol...). In fact, SCA aims at being technology-independent.

1.2 - The weather forecast application

The application you are going to create is an SCA application that provides, for a given location, the current weather conditions and the weather forecast for the next days. This application will be accessible by a client application and will use an existing web service.

Until some months ago, there was a free web service on the Internet that provided the (real) weather conditions for a given city. In this tutorial, we will do as if this service was still online. In fact, we will provide you an SCA application exposed a web service that will mimic that defunct web service.

Our application will complete the web service feature by providing the weather forecast.

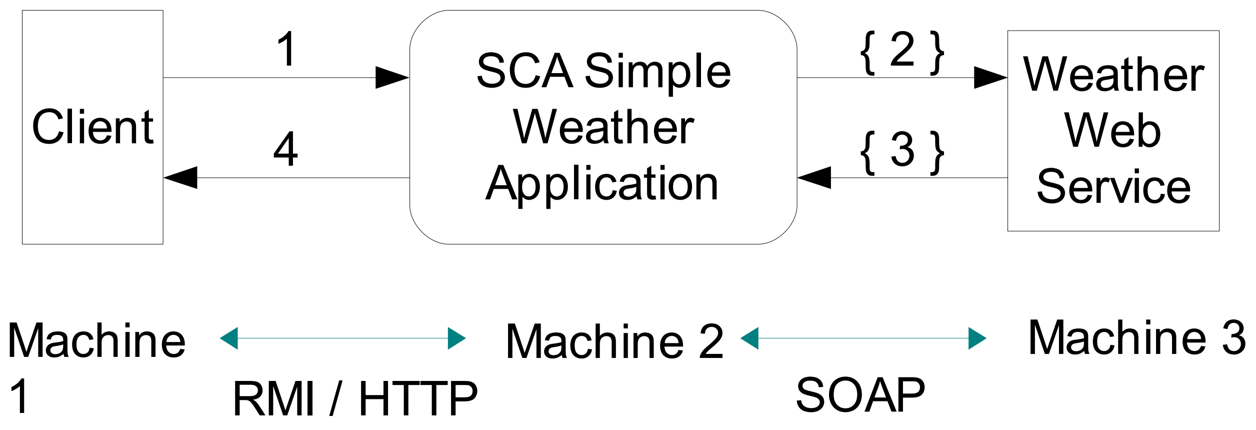


Fig. 1: use context of the application

Features:

The client handles interaction with the user.
The weather web service provide the current weather conditions.
The SCA application provides the weather forecast and aggregates it with the current conditions.

Messages:

1: The client program calls our SCA application.
2: The SCA application may (or not) call the web service to deliver its service.
3: If the web service was called, the application gets the result.
4: The application returns the result to the client.

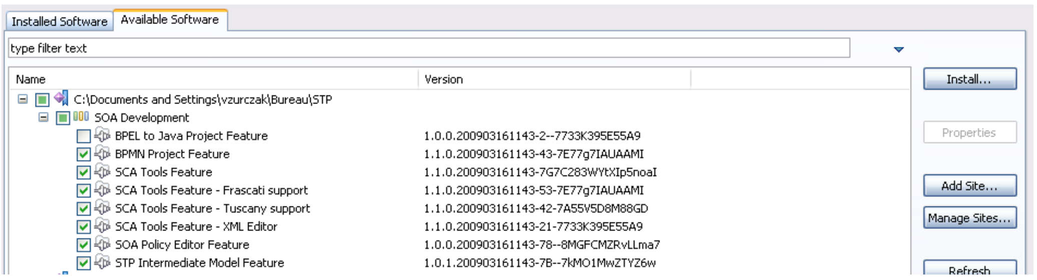
In theory, this application could work on three separate machines.
In the scope of this tutorial, there will be only one machine, yours.

2. Create the SCA weather application

2.1 - Set up the environement

First, install the SCA Tools project (if it is not already done).

- Unzip the given update site on your hard-drive.
- Select **Help > Software updates...**
- Open the **Available software** tab and click **Add site...**
- Click **Local** and select the update site root folder on your hard-drive. Click **OK**.
- Select all the SCA features (you can even select all the STP features) and click **Install...**


Installed Software | Available Software

type filter text

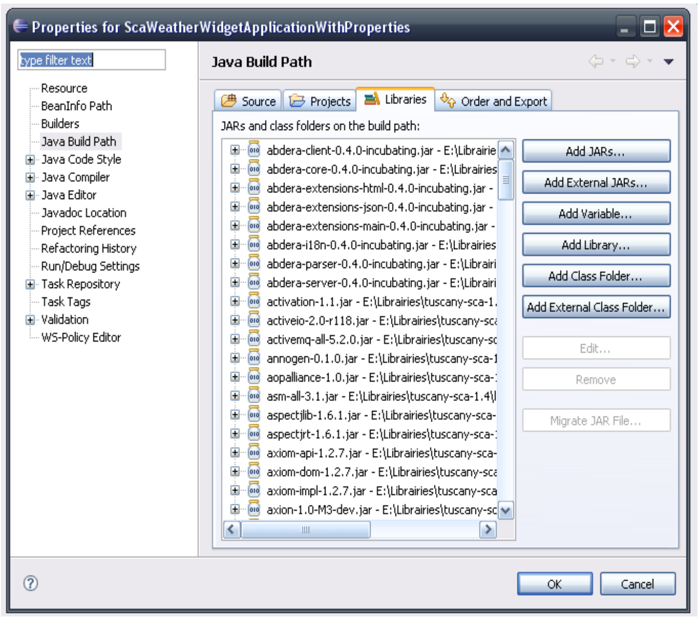
Name	Version
C:\Documents and Settings\vzurczak\Bureau\STP	
SOA Development	
☐ BPEL to Java Project Feature	1.0.0.200903161143-2--7733K395E55A9
✓ BPMN Project Feature	1.1.0.200903161143-43-7E77g7IAUAAMI
✓ SCA Tools Feature	1.1.0.200903161143-7G7C283WYtXIp5noaI
✓ SCA Tools Feature - Frascati support	1.1.0.200903161143-53-7E77g7IAUAAMI
✓ SCA Tools Feature - Tuscany support	1.1.0.200903161143-42-7A55V5D8M88GD
✓ SCA Tools Feature - XML Editor	1.1.0.200903161143-21-7733K395E55A9
✓ SOA Policy Editor Feature	1.0.0.200903161143-78--8MGFCMZRvLLma7
✓ STP Intermediate Model Feature	1.0.1.200903161143-7B--7kMO1MwZTYZ6w

Then, install Apache Tuscany. Unzip the given Tuscany archive (in the resources) on your hard-drive. Usually, you would go on their website to download the binary distribution (*http://incubator.apache.org/tuscany/sca-java-releases.html*).

Eventually, let's prepare the SCA application that will act as the weather web service.
Select **File > Import...** and then **General > Existing projects in the workspace**.
Browse the provided resource files and select the root folder. You should get proposed the project “SimpleWeatherApplication”.

Only select this project and make sure **Copy projects in the workspace** is checked. Click **OK**.
Once the project is in the workspace, add the Tuscany libraries to its build path.

Update the project classpath by adding Tuscany libraries in the project referenced libraries.
Right-click on the project and select **Build path > Configure build path** (or select **Properties** and then **Java Build Path**). In the **Libraries** tab, click **Add external JARs** and go select all the libraries (Ctrl + A) in the Tuscany 'lib' folder you unzipped previously. You should have:



JARs and class folders on the build path:

```
abdera-client-0.4.0-incubating.jar - E:\Librairie
abdera-core-0.4.0-incubating.jar - E:\Librairies
abdera-extensions-html-0.4.0-incubating.jar -
abdera-extensions-json-0.4.0-incubating.jar -
abdera-extensions-main-0.4.0-incubating.jar -
abdera-i18n-0.4.0-incubating.jar - E:\Librairies
abdera-parser-0.4.0-incubating.jar - E:\Librairi
abdera-server-0.4.0-incubating.jar - E:\Librairi
activation-1.1.jar - E:\Librairies\tuscany-sca-1.
activeio-2.0-r118.jar - E:\Librairies\tuscany-sca
activemq-all-5.2.0.jar - E:\Librairies\tuscany-sc
annogen-0.1.0.jar - E:\Librairies\tuscany-sca-1
aopalliance-1.0.jar - E:\Librairies\tuscany-sca-
asm-all-3.1.jar - E:\Librairies\tuscany-sca-1.4\
aspectjlib-1.6.1.jar - E:\Librairies\tuscany-sca-
aspectjrt-1.6.1.jar - E:\Librairies\tuscany-sca-
axiom-api-1.2.7.jar - E:\Librairies\tuscany-sca-
axiom-dom-1.2.7.jar - E:\Librairies\tuscany-sca
axiom-impl-1.2.7.jar - E:\Librairies\tuscany-sca
axion-1.0-M3-dev.jar - E:\Librairies\tuscany-sc
```

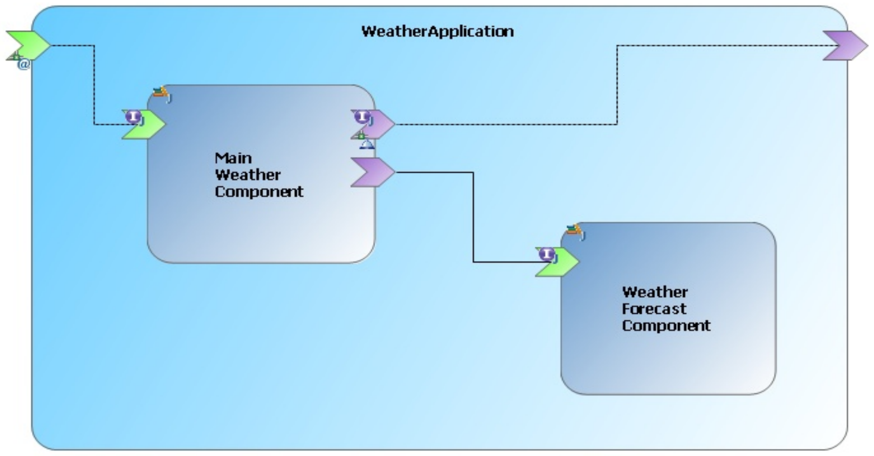
Click **OK**.
The project should now compile without any error.

Start this application by launching the *org.eclipse.stp.sca.demo.simpleweather.client.SimpleWeatherDeployment.java* file. Right-click on it and select **Run As... > Java Application**. Wait few seconds that it starts and open your browser to the address *http://localhost:8080/SimpleWeather?wsdl=SimpleWeather.wsdl*. This is the WSDL interface of the web service. You can also find it in the given project, in the *src* folder.

2.2 - Create the application

The application frame

The following figure shows the SCA assembly of the application that you will create.



This composite, named WeatherApplication, is a composition of two components:

- MainWeatherComponent, which acts as an orchestration component, and
- WeatherForecastComponent, which is in charge of the weather forecast.

First, create an **SCA Java Project** to hold the weather application:

1. Select **File > New > SCA Tools > SCA Java Project**.
2. Set "WeatherApplication" as the **Project name**. For **Project layout**, select **Create separate folders for sources and class files**. Click on the **Next** button.
3. Click on the **Finish** button.

The created project is both a Java project and an SCA project.
In the same way you did it previously, add the Tuscany libraries to the classpath of the project.

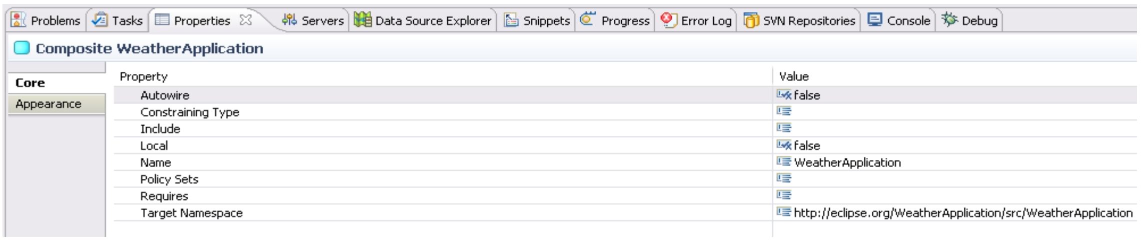
Create a new SCA Composite diagram

To create an SCA diagram:

1. Right-click the project and select **New > Other....**
2. In the New wizard, select **SCA Tools > SCA Composite Diagram** and click **Next**.
3. Choose the *src* folder as container and type a unique name for the diagram in the **File name** field. Click **Finish**.

The new created file is automatically open with the SCA Composite Designer. To open the

Properties view, do a right click on the diagram and the select **Show Properties View**.


Problems | Tasks | Properties | Servers | Data Source Explorer | Snippets | Progress | Error Log | SVN Repositories | Console | Debug

Composite WeatherApplication

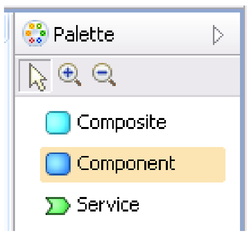
Core
Appearance

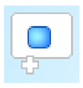
Property	Value
Autowire	false
Constraining Type	
Include	
Local	false
Name	WeatherApplication
Policy Sets	
Requires	
Target Namespace	http://eclipse.org/WeatherApplication/src/WeatherApplication

In accordance with the SCA specifications, the composite and the composite file have the same name. The name of the composite is automatically set and a target namespace was generated.

Components

Add two components named *MainWeatherComponent* and *WeatherForecastComponent*. You can do it with the Component creation tool which is in the palette or using the contextual menu.





Services

Next, add the following services:

- *MainWeatherService* on the MainWeatherComponent,
- *WeatherForecastService* on the WeatherForecastComponent.

References

In the next step, add the following references on the MainWeatherComponent:

- *weatherWSReference*,
- *weatherForecastReference*.

Wires

Now you can wire the services and references. From the palette, you can use :

- the **Wire** creation tool: a "wire" element is added,
- the **Wire target**: target attribute of the "reference" element is used.

Create a wire between *weatherForecastReference* and *WeatherForecastService*.

Promotion

The next step is to handle promotions. Two ways are possible to promote a service (or a reference). You can:

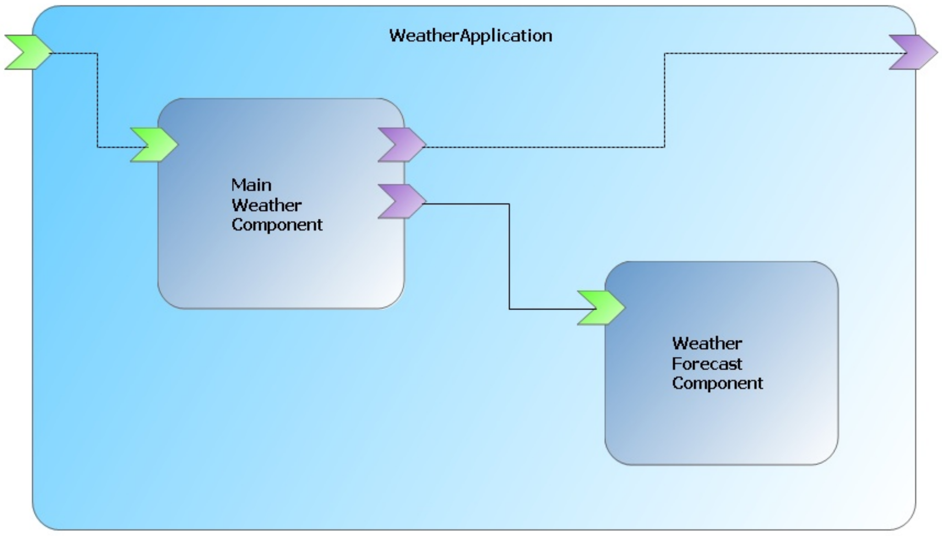
- create a new Service on the composite and then use the **Promote** creation tool from the palette to add a promotion link between the composite service and the promoted component service. Or,

- right-click a component service, and select **Promote** menu item.

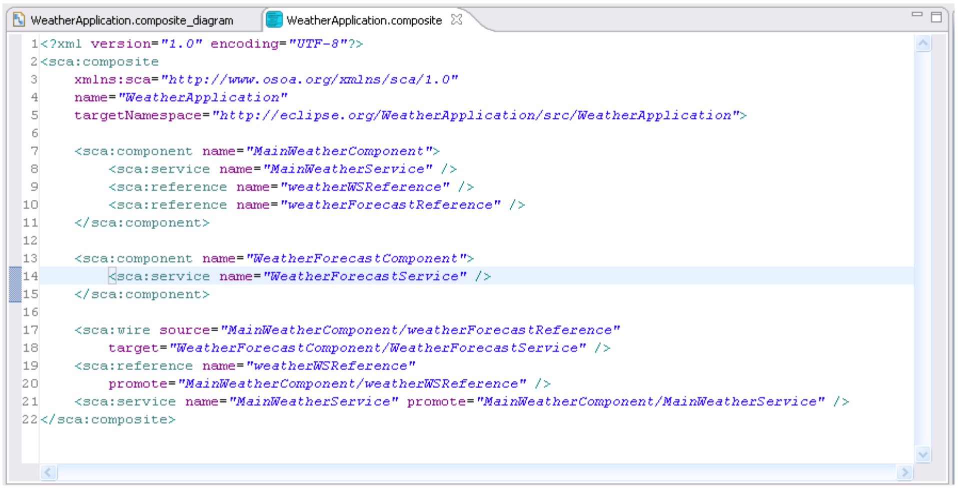
Promote *MainWeatherService* and *weatherWSReference*.

Save your diagram. The **WeatherApplication.composite_diagram** contains the graphical part of your SCA assembly and the **WeatherApplication.composite** file contains the XML code that describes your SCA assembly.

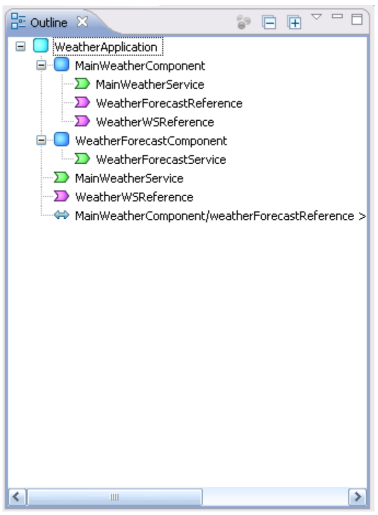
Your diagram should look like:



To view the composite source, double-click the **WeatherApplication.composite** file. It will be opened in the SCA XML editor. To format it in a pretty way, select all the code, right-click on it and select **Source > Format**.


WeatherApplication.composite_diagram
WeatherApplication.composite

```xml
<?xml version="1.0" encoding="UTF-8"?>
<sca:composite
    xmlns:sca="http://www.osoa.org/xmlns/sca/1.0"
    name="WeatherApplication"
    targetNamespace="http://eclipse.org/WeatherApplication/src/WeatherApplication">

    <sca:component name="MainWeatherComponent">
        <sca:service name="MainWeatherService" />
        <sca:reference name="weatherWSReference" />
        <sca:reference name="weatherForecastReference" />
    </sca:component>

    <sca:component name="WeatherForecastComponent">
        <sca:service name="WeatherForecastService" />
    </sca:component>

    <sca:wire source="MainWeatherComponent/weatherForecastReference"
        target="WeatherForecastComponent/WeatherForecastService" />
    <sca:reference name="weatherWSReference"
        promote="MainWeatherComponent/weatherWSReference" />
    <sca:service name="MainWeatherService" promote="MainWeatherComponent/MainWeatherService" />
</sca:composite>
```



To navigate easily in the document, you can use the outline view (that can be started by selecting **Window > Show view > Outline**). Select one element in the outline and it will be highlighted in the editor. Use the **Expand all** action (the right upper '+' button) to discover all the elements.

Define interfaces and implementations

Hint: all the source files are provided as resources for fast copy-paste.

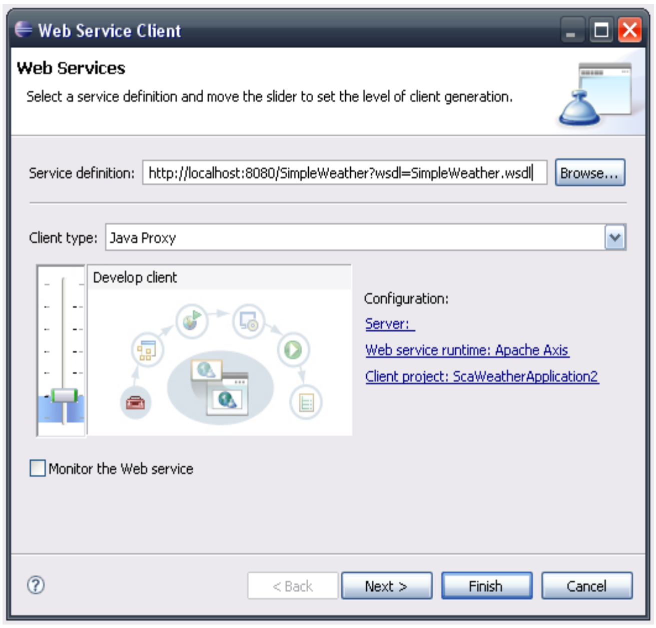
First, in the *src* directory of the WeatherApplication project, create a new package named *org.eclipse.stp.sca.demo.weather* with two sub-packages named *interfaces* and *impl*.

Interfaces

Define the interfaces of the services and references.
The first one we need to get is the one for the referenced web service.

- WeatherWSReference

Right-click on the WeatherApplication project and select **SCA > Create WS Client**. A dialog shows up. Set the left scale to "**Develop client**" and enter the given WSDL URL (*http://localhost:8080/SimpleWeather?wsdl=SimpleWeather.wsdl*) in the **URL** field.


Web Service Client

Web Services

Select a service definition and move the slider to set the level of client generation.

Service definition: http://localhost:8080/SimpleWeather?wsdl=SimpleWeather.wsdl

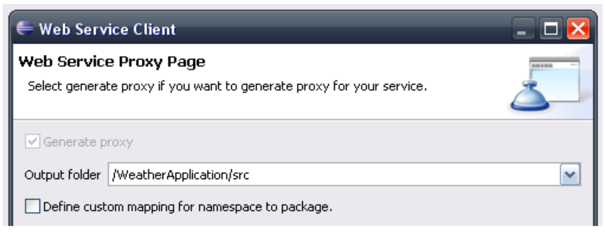
Client type: Java Proxy

Develop client

Configuration:
Server:
Web service runtime: Apache Axis
Client project: ScaWeatherApplication2

Monitor the Web service

Click **Next** once the button is enabled.
Make sure the destination is the project *src* folder and click **OK**.



Web Service Proxy Page

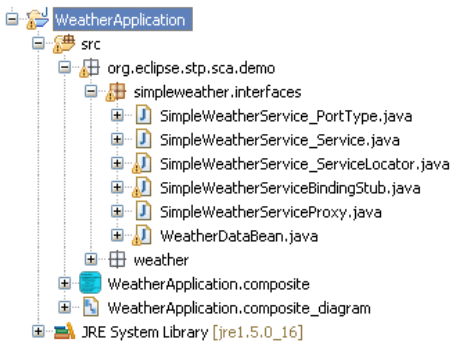
Select generate proxy if you want to generate proxy for your service.

☑ Generate proxy

Output folder /WeatherApplication/src

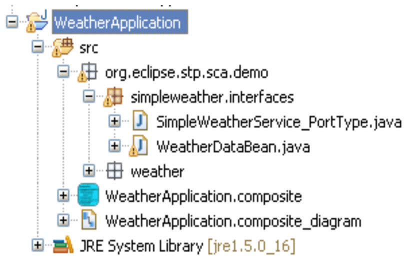
☐ Define custom mapping for namespace to package.

You should now have:



```
WeatherApplication
  src
    org.eclipse.stp.sca.demo
      simpleweather.interfaces
        SimpleWeatherService_PortType.java
        SimpleWeatherService_Service.java
        SimpleWeatherService_ServiceLocator.java
        SimpleWeatherServiceBindingStub.java
        SimpleWeatherServiceProxy.java
        WeatherDataBean.java
      weather
    WeatherApplication.composite
    WeatherApplication.composite_diagram
  JRE System Library [jre1.5.0_16]
```

What we have done here is running a WSDL to Java transformation. The generated code contains some extra-elements we don't need. Right-click on the project and select **SCA > Clean Axis code**. Eventually, you are asked if you want to remove Axis libraries from the classpath. Click **Yes**. You should now have only two classes. The one whose name ends with PortType is the interface of the referenced service.



```
WeatherApplication
  src
    org.eclipse.stp.sca.demo
      simpleweather.interfaces
        SimpleWeatherService_PortType.java
        WeatherDataBean.java
      weather
    WeatherApplication.composite
    WeatherApplication.composite_diagram
  JRE System Library [jre1.5.0_16]
```

Notice:

This approach works when the service operations do not contain specific types (e.g. enum) and use generic types. In complex cases, you will most likely have to use data-binding libraries (e.g. JAX-B, in association with a WSDL to Java library like Apache CXF's one). The given solution has the advantage of embedding a minimal amount of code and libraries in your SCA application.

There is still one thing to clean up manually from the remaining code.
Open the PortType class and remove *implements Remotable* and *throws RMIException* from the code.
Instead, add the *@Remotable* SCA annotation on the class.

You should have:

```java
package org.eclipse.stp.sca.demo.simpleweather.interfaces;

import org.osoa.sca.annotations.Remotable;

@Remotable
public interface SimpleWeatherService_PortType {

	public org.eclipse.stp.sca.demo.simpleweather.interfaces.WeatherDataBean getCurrentWeather(
		java.lang.String city,
		java.lang.String country,
		java.lang.Boolean temperatureInCelsiusDegrees);
}
```

The reference interface is now available.
Let's create the other interfaces (in the ***.weather.interfaces** package).

- WeatherForecastService

```java
package org.eclipse.stp.sca.demo.weather.interfaces;

import java.util.List;
import org.eclipse.stp.sca.demo.simpleweather.interfaces.WeatherDataBean;

public interface WeatherForecastService {
	public List<WeatherDataBean> getWeatherForecast(
			String city,
			String country,
			boolean temperatureInCelsius );
}
```

- MainWeatherService

```java
package org.eclipse.stp.sca.demo.weather.interfaces;

import org.osoa.sca.annotations.Remotable;

@Remotable
public interface MainWeatherService extends WeatherForecastService {
	// nothing
}
```

Implementations

Next, define the implementations (in the ***.weather.impl** package).

- WeatherForecastComponent

```java
package org.eclipse.stp.sca.demo.weather.impl;

import java.util.ArrayList;
import java.util.List;
import java.util.Random;

import org.eclipse.stp.sca.demo.simpleweather.interfaces.WeatherDataBean;
import org.eclipse.stp.sca.demo.weather.interfaces.WeatherForecastService;
import org.osoa.sca.annotations.Service;

@Service(WeatherForecastService.class)
public class WeatherForecastImpl implements WeatherForecastService {

	public List<WeatherDataBean> getWeatherForecast(
			String city,
			String country,
			boolean temperatureInCelsius ) {

		List<WeatherDataBean> result = new ArrayList<WeatherDataBean>( 3 );
		for( int i=0; i<3; i++ )
			result.add( createRandomValue( city, country, temperatureInCelsius ));
```

```
                return result;
        }

        protected WeatherDataBean createRandomValue(
                        String city,
                        String country,
                        boolean temperatureInCelsius ) {

                WeatherDataBean result = new WeatherDataBean();
                Random rand = new Random();

                // -40 °C <= T° <= 40 °C
                int celsiusTemperature = rand.nextInt( 40 );
                if( rand.nextInt( 2 ) == 1 )
                        celsiusTemperature = -celsiusTemperature;

                result.setTemperatureInCelsius( temperatureInCelsius );
                if( temperatureInCelsius )
                     result.setTemperature( celsiusTemperature + " °C" );
                else
                     result.setTemperature((celsiusTemperature * 5 /9 + 32) + " °F" );

                // Wind gust
                int wind = rand.nextInt( 45 );
                result.setWindGust( wind );

                // Humidity
                int humidity = rand.nextInt( 100 );
                result.setHumidity( humidity );

                // Many clouds ?
                if( wind > 20 && humidity > 85 )
                        result.setCloud( "stormy" );
                else if( wind > 20 && celsiusTemperature > 0 )
                        result.setCloud( "windy" );
                else if( celsiusTemperature < -5 )
                        result.setCloud( "snowy" );
                else if( humidity > 85 )
                        result.setCloud( "rainy" );
                else {
                        String[] values = new String[] {
                                        "cloudy",
                                        "foggy",
                                        "rainy",
                                        "slightly_cloudy",
                                        "sunny",
                                        "windy"
                        };
                        int i = rand.nextInt( values.length );
                        result.setCloud( values[ i ].toString());
                }

                // 870 and 1086 hPa are registered extreme measures
                int pressure = 870 + rand.nextInt( 216 );
                result.setPressure( pressure );

                return result;
        }
}
```

- MainWeatherComponent

```
package org.eclipse.stp.sca.demo.weather.impl;

import java.util.List;

import org.eclipse.stp.sca.demo.simpleweather.interfaces.SimpleWeatherService_PortType;
import org.eclipse.stp.sca.demo.simpleweather.interfaces.WeatherDataBean;
import org.eclipse.stp.sca.demo.weather.interfaces.MainWeatherService;
import org.eclipse.stp.sca.demo.weather.interfaces.WeatherForecastService;
import org.osoa.sca.annotations.Reference;
import org.osoa.sca.annotations.Service;

@Service(MainWeatherService.class)
public class MainWeatherImpl implements MainWeatherService {

        /* *******************************************************
```

```
         * Handle SCA references
         */

        private WeatherForecastService weatherForecastReference;
        private SimpleWeatherService_PortType weatherWSReference;

        @Reference
        public void setWeatherForecastReference(
                        WeatherForecastService weatherForecastReference) {
                this.weatherForecastReference = weatherForecastReference;
        }

        @Reference
        public void setSimpleWeatherWSReference(
                        SimpleWeatherService_PortType weatherWSReference ) {
                this. weatherWSReference = weatherWSReference;
        }


        /* *******************************************************
         * Implement the services interfaces (one in this case)
         */

        public List<WeatherDataBean> getWeatherForecast(
                        String city,
                        String country,
                        boolean temperatureInCelsius ) {

                List<WeatherDataBean> result =
                weatherForecastReference.getWeatherForecast( city, country, temperatureInCelsius );

                WeatherDataBean current =
                weatherWSReference.getCurrentWeather( city, country, temperatureInCelsius );
                result.add( 0, current );

                return result;
        }
}
```

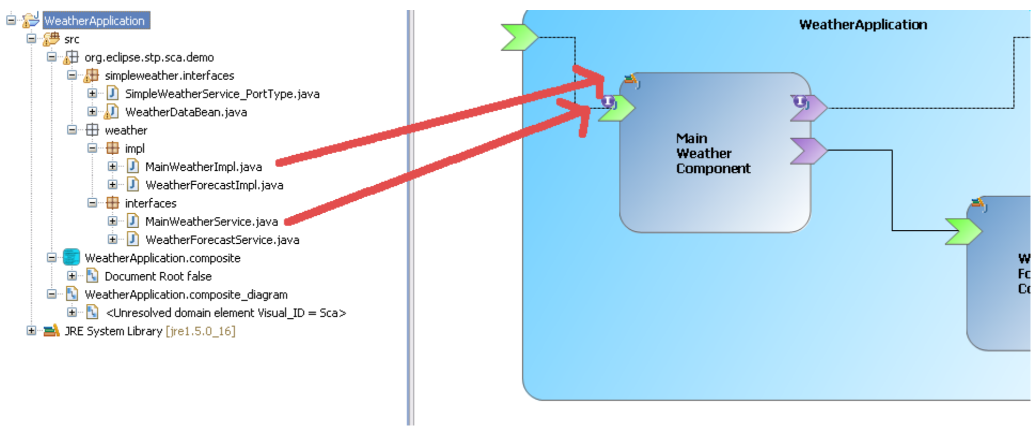
As you can see, in Java, referenced services are manipulated as Java fields.
Service operations are called as Java methods, and data are Java objects (no XML).

Bind the composite with the code

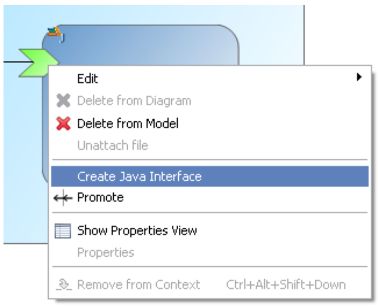
You can now complete you SCA assembly file with the interfaces and implementations that you defined. Open the **WeatherApplication.composite_diagram** file with the SCA Composite Designer.

You have two ways to add interfaces and implementations on your SCA assembly file:

- Drag and drop the Java interface files on the services of your diagram and the Java implementation files on the components. The name of these elements are automatically set.

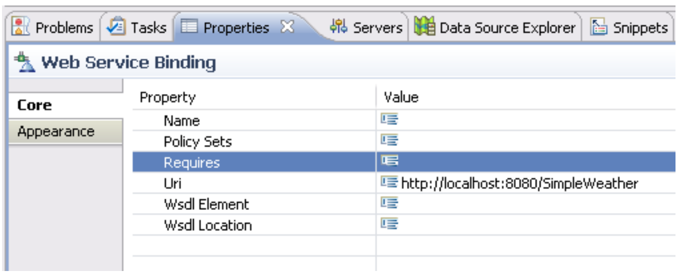


- Right click on a service, a reference or an implementation and select Create Java interface / implementation.



Now, let's configure the bindings (the communication protocols) used by our application.

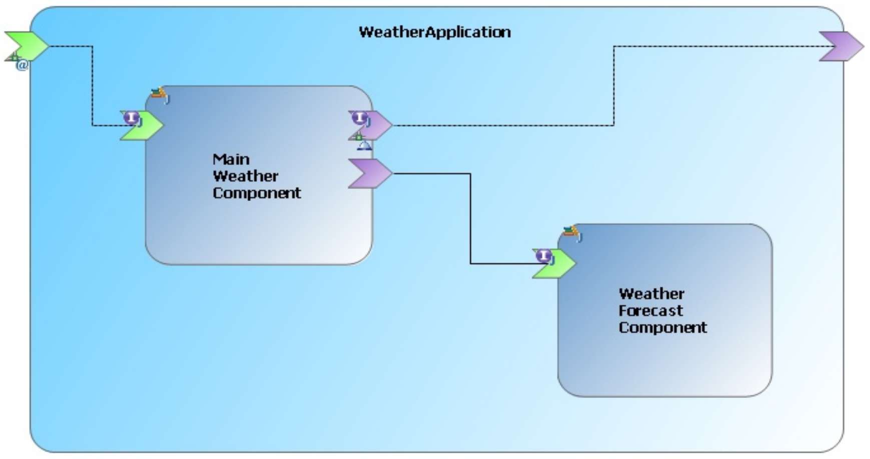
- Add a Web service binding on weatherWSReference.
  - Select it once it is visible on the diagram and check the properties view.
  - In the URI field, type in the web service address: *http://localhost:8080/SimpleWeather* (this is the URL of the web service if our application needs to call it).



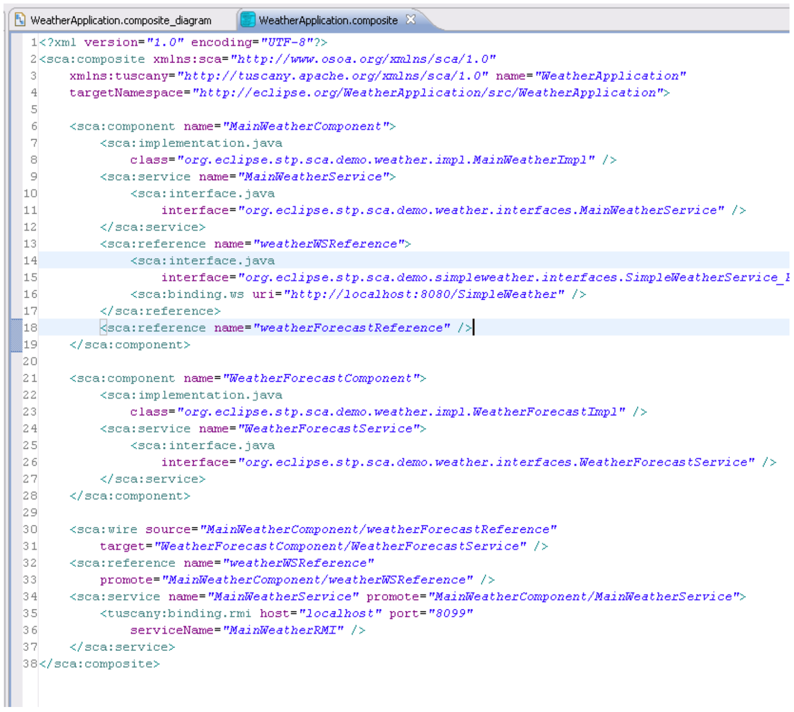
Property	Value
Name	
Policy Sets	
Requires	
Uri	http://localhost:8080/SimpleWeather
Wsdl Element	
Wsdl Location	

- Add a RMI binding on the composite service.
  - Select it once it is visible on the diagram and check the properties view.
  - Set the following fields:
    - Host: localhost
    - Port: 8099
    - ServiceName: WeatherApplicationRMI

Now, your SCA assembly diagram should look like:



and, your SCA assembly file looks like:


WeatherApplication.composite_diagram
WeatherApplication.composite

```xml
<?xml version="1.0" encoding="UTF-8"?>
<sca:composite xmlns:sca="http://www.osoa.org/xmlns/sca/1.0"
    xmlns:tuscany="http://tuscany.apache.org/xmlns/sca/1.0" name="WeatherApplication"
    targetNamespace="http://eclipse.org/WeatherApplication/src/WeatherApplication">

    <sca:component name="MainWeatherComponent">
        <sca:implementation.java
            class="org.eclipse.stp.sca.demo.weather.impl.MainWeatherImpl" />
        <sca:service name="MainWeatherService">
            <sca:interface.java
                interface="org.eclipse.stp.sca.demo.weather.interfaces.MainWeatherService" />
        </sca:service>
        <sca:reference name="weatherWSReference">
            <sca:interface.java
                interface="org.eclipse.stp.sca.demo.simpleweather.interfaces.SimpleWeatherService_
            <sca:binding.ws uri="http://localhost:8080/SimpleWeather" />
        </sca:reference>
        <sca:reference name="weatherForecastReference" />
    </sca:component>

    <sca:component name="WeatherForecastComponent">
        <sca:implementation.java
            class="org.eclipse.stp.sca.demo.weather.impl.WeatherForecastImpl" />
        <sca:service name="WeatherForecastService">
            <sca:interface.java
                interface="org.eclipse.stp.sca.demo.weather.interfaces.WeatherForecastService" />
        </sca:service>
    </sca:component>

    <sca:wire source="MainWeatherComponent/weatherForecastReference"
        target="WeatherForecastComponent/WeatherForecastService" />
    <sca:reference name="weatherWSReference"
        promote="MainWeatherComponent/weatherWSReference" />
    <sca:service name="MainWeatherService" promote="MainWeatherComponent/MainWeatherService">
        <tuscany:binding.rmi host="localhost" port="8099"
            serviceName="MainWeatherRMI" />
    </sca:service>
</sca:composite>
```

There is no action to undertake to validate this application.
The SCA tools automatically make sure that any action you make does not result in an invalid application. When errors are found, errors markers are added on both the files and on the diagram.

If you want to test it, you can, as an example, provide a Java interface as a component implementation. An error should be displayed on the component implementation.

Testing the application

Create the package *org.eclipse.stp.sca.demo.weather.client* and create a class called MainWeatherDeployment.

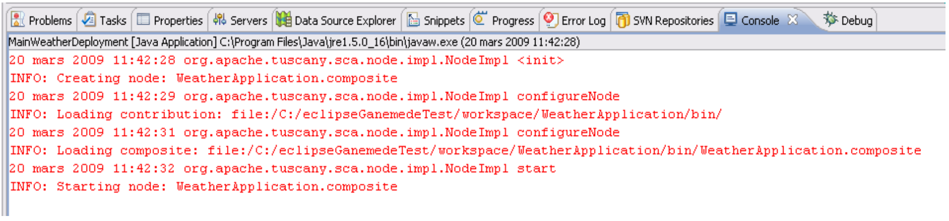
```java
package org.eclipse.stp.sca.demo.weather.client;

import java.io.IOException;
import org.apache.tuscany.sca.host.embedded.SCADomain;

public class MainWeatherDeployment {
        public static void main( String[] args ) {

                SCADomain scaDomain = null;
                try {
                        scaDomain = SCADomain.newInstance( "WeatherApplication.composite" );
                        char c;
                        while(( c = (char) System.in.read()) != 'q' ) {
                                System.out.println( c );
                        }

                } catch (IOException e) {
                        e.printStackTrace();
                } finally {
                        scaDomain.close();
                }

                System.exit( 0 );
        }
}
```

This class will deploy our application on Apache Tuscany.
We also need a client. In the same package, create the class *MainWeatherClient*.

```
package org.eclipse.stp.sca.demo.weather.client;

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.rmi.Naming;
import java.text.SimpleDateFormat;
import java.util.GregorianCalendar;
import java.util.List;

import org.eclipse.stp.sca.demo.simpleweather.interfaces.WeatherDataBean;
import org.eclipse.stp.sca.demo.weather.interfaces.GlobalWeatherService;

public class GlobalWeatherClient {

     public static void main( String[] args ) {
               try {
                      GlobalWeatherService gwService =
                      (GlobalWeatherService) Naming.lookup( "//localhost:8099/MainWeatherRMI" );

                      callWeatherService( gwService );

               } catch( Exception e ) {
                      e.printStackTrace();
               }
       }

       private static String getUserInput() {

               InputStreamReader isr = new InputStreamReader( System.in );
               BufferedReader br = new BufferedReader( isr );
               String s = null;
               try {
                  s = br.readLine();
               }
               catch ( Exception e ) {
                      e.printStackTrace();
               }

               return s;
       }

       private static void callWeatherService( GlobalWeatherService gwService ) {

               try {
                      // Testing web service reference
                      System.out.println( "Weather Conditions and Forecast service - Welcome!\n" );
                      System.out.print( "City: " );
                      String city = getUserInput();
                      System.out.print( "Country: " );
                      String country = getUserInput();
                      System.out.print( "Get the temperature in Celsius or Fahrenheit degrees ? (C/
F) " );

                      char choice;
                      while(( choice = (char) System.in.read()) != 'C' && choice != 'F' )
                             System.out.print(
               "Invalid choice.\nGet the temperature in Celsius or Fahrenheit degrees ? (C/F) " );

                      // Call the service
                      List<WeatherDataBean> beans =
                      gwService.getWeatherForecast( city, country, choice == 'C' );

                      // Display results
                      if( beans == null ) {
                             System.out.println( "An error occurred. Received no data." );
                             return;
                      }

                      System.out.println( "Weather conditions and forecast in "
                                          + city + ", " + country + "\n" );
                      SimpleDateFormat sdf = new SimpleDateFormat( "E" );
                      GregorianCalendar now = new GregorianCalendar();

                      int day = 0;
                      for( WeatherDataBean bean : beans ) {

                             now.add( GregorianCalendar.DAY_OF_YEAR, 1 );
```

```
                              if( day == 0 ) {
                                     System.out.print( "Today: " );
                                     day ++;
                              }
                              else if( day == 1 ) {
                                     System.out.print( "Tomorrow: " );
                                     day ++;
                              }
                              else {
                                     System.out.print( sdf.format( now.getTime()) + ": " );
                              }

                              System.out.println(
                                            bean.getCloud() + ", "
                                            + bean.getTemperature()
                                            + " (" + bean.getPressure() + " hPa)" );
                       }

                } catch( Exception e ) {
                       e.printStackTrace();
                }
        }
}
```

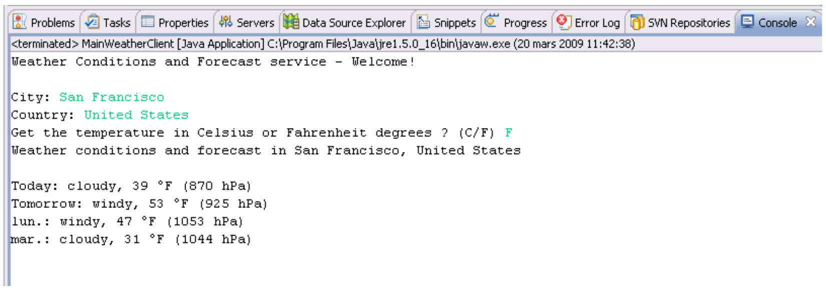
Launch the deployment of the application.
Right-click on WeatherApplicationDeployment.java and select **Run as > Java application**.



```
MainWeatherDeployment [Java Application] C:\Program Files\Java\jre1.5.0_16\bin\javaw.exe (20 mars 2009 11:42:28)
20 mars 2009 11:42:28 org.apache.tuscany.sca.node.impl.NodeImpl <init>
INFO: Creating node: WeatherApplication.composite
20 mars 2009 11:42:29 org.apache.tuscany.sca.node.impl.NodeImpl configureNode
INFO: Loading contribution: file:/C:/eclipseGanemedeTest/workspace/WeatherApplication/bin/
20 mars 2009 11:42:31 org.apache.tuscany.sca.node.impl.NodeImpl configureNode
INFO: Loading composite: file:/C:/eclipseGanemedeTest/workspace/WeatherApplication/bin/WeatherApplication.composite
20 mars 2009 11:42:32 org.apache.tuscany.sca.node.impl.NodeImpl start
INFO: Starting node: WeatherApplication.composite
```

Now, launch the client in the same way.

You should get something like that.



```
<terminated> MainWeatherClient [Java Application] C:\Program Files\Java\jre1.5.0_16\bin\javaw.exe (20 mars 2009 11:42:38)
Weather Conditions and Forecast service - Welcome!

City: San Francisco
Country: United States
Get the temperature in Celsius or Fahrenheit degrees ? (C/F) F
Weather conditions and forecast in San Francisco, United States

Today: cloudy, 39 °F (870 hPa)
Tomorrow: windy, 53 °F (925 hPa)
lun.: windy, 47 °F (1053 hPa)
mar.: cloudy, 31 °F (1044 hPa)
```

To stop the application, type 'q' and 'enter' in the console.

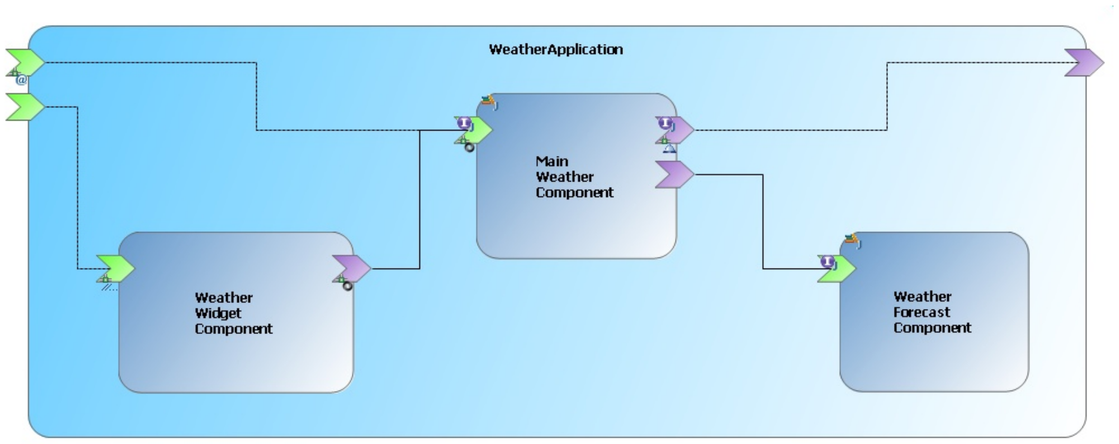
3. Going further: the Weather Web Application

The weather application is nice. But with a single RMI client, nobody is going to use it.
We will modify our application to make it available from a simple Web browser. To achieve this, we will use a nice implementation type which is available in Apache Tuscany: the widget implementation.

3.1 - Modify the composite

- Add a component called *WeatherWidgetComponent*.
- Add it one service called *Widget*, and promote it to the composite.
- Add it one reference called *mainWeatherReference*.
- Add a wire between *mainWeatherReference* and *MainWeatherService*.
- Add a JSON-RPC binding on both *mainWeatherReference* and *MainWeatherService*.
  - Set the *URI* field of *MainWeatherService* to *http://localhost:8081/MainWeather*
  - Do not modify the second JSON-RPC binding.
- Add a HTTP binding on the *Widget* service.
  - Set the *URI* field to *http://localhost:8081/WeatherApplication*

You should have the following diagram:



3.2 - Create the implementation.widget

The widget implementation is a web page which embeds Javascript code to update the page contents. This javascript is also in charge of calling the component references and getting the result in case of callbacks.

In our case, the main component will return the same objects that we received in RMI. Therefore, the javascript will expect to receive elements after reference operations are called.

Create a new source folder and call it *web* (right-click on the project and select **New > Source Folder**). In this folder, create a file called WeatherForecast.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<meta http-equiv="Pragma" content="no-cache">
<title>Weather Forecast</title>
```

```
<style type="text/css" >@IMPORT url("style.css");</style>
<script type="text/javascript" src="WeatherForecast.js"></script>
<script language="JavaScript">

        //@Reference
        var mainWeatherReference = new Reference( "mainWeatherReference" );

        function getWeather() {
                var city = document.getElementById( "cityField" ).value;
                var country = document.getElementById( "countryField" ).value;
                mainWeatherReference.getWeatherForecast( city, country, false, displayWeatherData );

        }


        function displayWeatherData( weatherBeans ) {

                var table = document.getElementById( "Table" );
                table.style.visility = "visible";

                // First row
                var fields = Array( ' - ', 'Clouds', 'Temperature', 'Wind', 'Humidity',
'Pressure' );
                tr = document.createElement( 'tr' );
                for( var i=0; i<fields.length; i++ ) {
                        td = document.createElement ('th');
                        text = document.createTextNode( fields[ i ]);
                        td.appendChild( text );
                        tr.appendChild( td );
                }
                table.appendChild( tr );

                // Weather per day
                var day = new Date().getDay();
                var days = Array( 'Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday',
'Saturday' );
                var list = weatherBeans.list;

                for( var i=0; i<list.length; i++ ) {
                        var currentDay;
                        if( i == 0 )
                                currentDay = "Today";
                        else if( i == 1 )
                                currentDay = "Tomorrow";
                        else
                                currentDay = days[ (day + i) % 7 ];
                        addTableRow( currentDay, list[ i ], table );
                }


                if( table.hasChildNodes()) {
                    while( table.childNodes.length > 5 ) {
                        table.removeChild( table.firstChild );
                    }
                }
        }


        function addTableRow( day, value, parent ) {

                tr = document.createElement( 'tr' );
                td = document.createElement ('th');
                text = document.createTextNode( day );
                td.appendChild( text );
                tr.appendChild( td );

                td = document.createElement ('td');
                img = document.createElement( 'img' );
                img.setAttribute( 'src', 'images/' + value.cloud + ".gif" );
                img.setAttribute( 'alt', value.cloud );
                img.setAttribute( 'title', value.cloud );
                td.appendChild( img );
                tr.appendChild( td );

                td = document.createElement ('td');
                text = document.createTextNode( value.temperature );
                td.appendChild( text );
                tr.appendChild( td );

                td = document.createElement ('td');
                text = document.createTextNode( value.windGust + ' km/h' );
                td.appendChild( text );
```

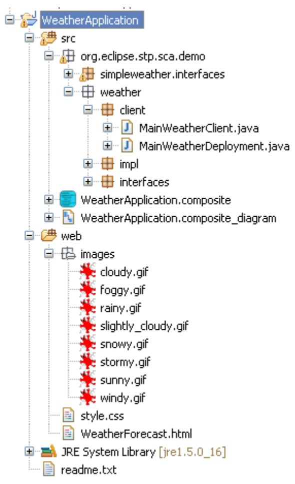
```
               tr.appendChild( td );

               td = document.createElement ('td');
               text = document.createTextNode( value.humidity + ' %' );
               td.appendChild( text );
               tr.appendChild( td );

               td = document.createElement ('td');
               text = document.createTextNode( value.pressure + ' hPa' );
               td.appendChild( text );
               tr.appendChild( td );

               parent.appendChild( tr );
        }
</script>
</head>
<body>
        <div id="container">
               <div id="header" />
               <div id="Search">
                      <p>Get weather forecast for</p>
                      <span>City: </span>
                      <input type="text" id="cityField" value="Your city" />
                      <span>Country: </span>
                      <input type="text" id="countryField" value="The country" />
                      <input type="button" value="Go" onClick="getWeather();" />
               </div>

               <table id="Table" style="visility: hidden;" />
        </div>
</body>
</html>
```

In the same folder, create a file called “style.css”.

```
html, body {
        width: 100%;
        font-weight: bold;
}

#container {
        width: 700px;
        margin: auto;
}

table {
        margin-top: 30px;
        border: 1px solid black;
        border-spacing: 0;
        font-weight: normal;
}

th {
        background-color: #eee;
        border: 1px solid black;
}

td {
        text-align: center;
        border: 1px solid black;
        width: 18%;
}

span {
        padding-left: 20px;
}
```

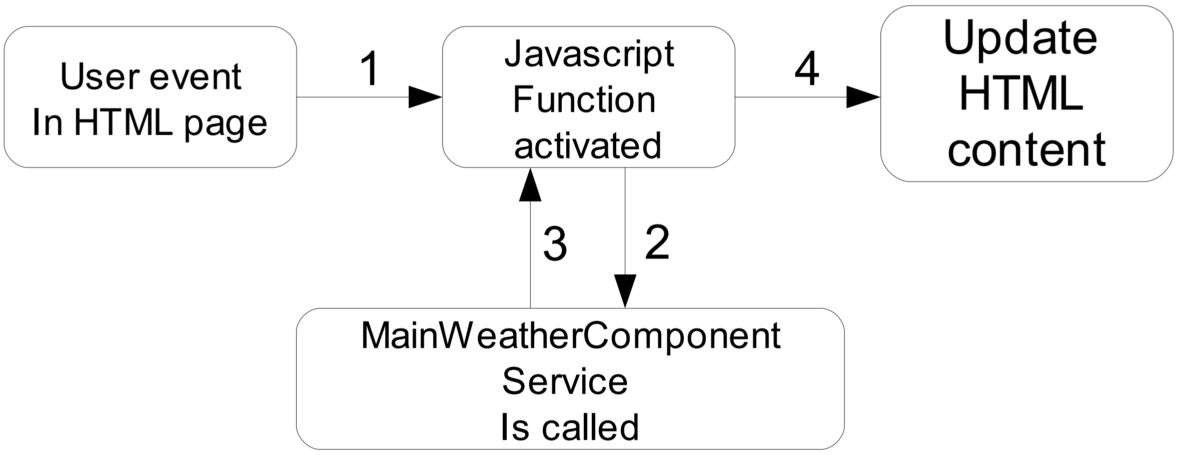
Eventually, create a sub-folder called “images”, and copy-paste the resource images into it.
To do that, you can make a usual copy-paste in the file system and then click **Refresh** on the project in the explorer.

You should have the following structure:



```
WeatherApplication
  src
    org.eclipse.stp.sca.demo
      simpleweather.interfaces
      weather
        client
          MainWeatherClient.java
          MainWeatherDeployment.java
        impl
        interfaces
    WeatherApplication.composite
    WeatherApplication.composite_diagram
  web
    images
      cloudy.gif
      foggy.gif
      rainy.gif
      slightly_cloudy.gif
      snowy.gif
      stormy.gif
      sunny.gif
      windy.gif
    style.css
    WeatherForecast.html
  JRE System Library [jre1.5.0_16]
  readme.txt
```

The web page code is made up of two parts: the HTML skeleton and the javascript code.
The javascript defines the component reference we need (to *MainWeatherComponent*), handles the calls to this reference and the data reception. It then decodes the received data (JSON objects) to display them in the page. It's usual Ajax programming.



1: a user event occurs in the page (e.g. the user click on a button).
2: the MainWeatherComponent is called through the JSON-RPC binding.
3: the called component service returns a list of beans containing weather conditions, one bean/day.
4: the javascript function updates the HTML page with the retrieved content.

The main issue you may encounter when developing this kind of implementation is the navigation in the returned objects. Originally, they are Java objects which are transformed into JSON objects. Navigate in JSON object structures is easy, when you know the structure.

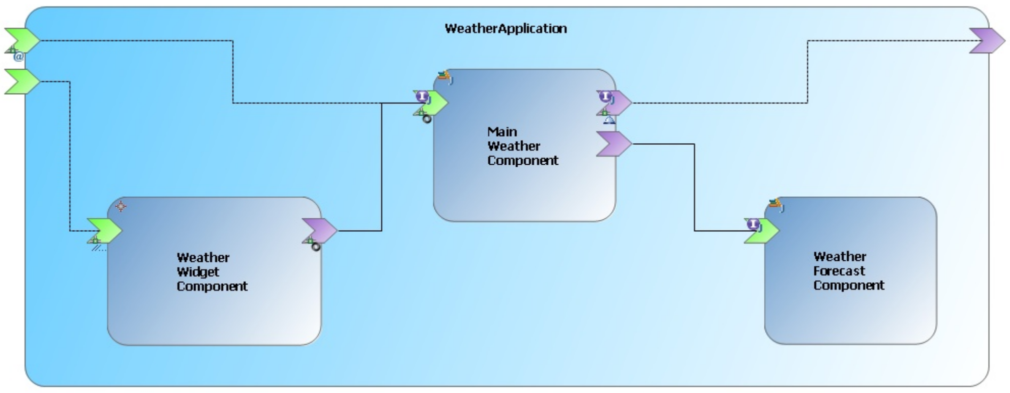
If you have issues to find this structure, you can either use a Javscript debugger, or if you use Mozilla Firefox, you can display the structure and the content of the object by using

window.alert(myJsonObject.toSource());

Now, let's get back to our application. Go to the diagram.

- Add a widget implementation on *WeatherWidgetComponent*.

- Set the *location* attribute value to WeatherForecast.html

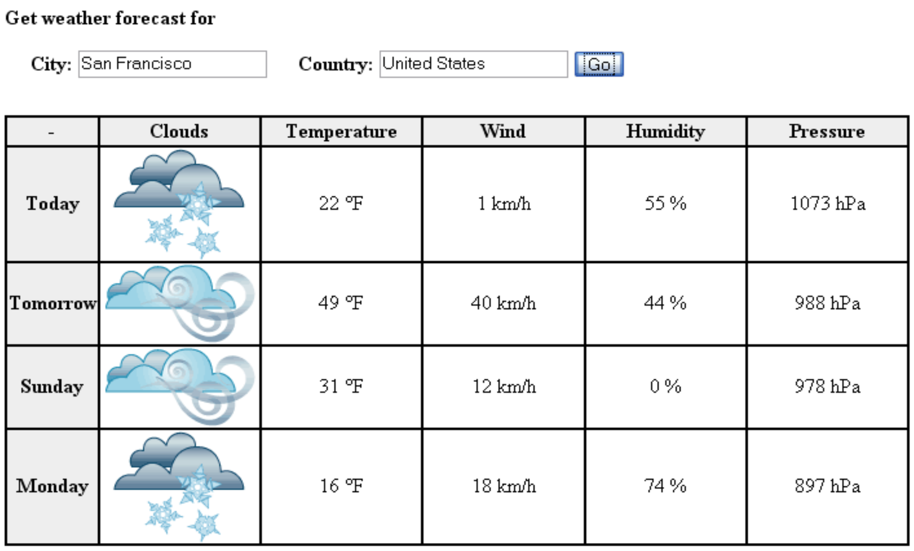


3.3 - Run the application

Launch the deployment of the application.
Right-click on WeatherApplicationDeployment.java and select **Run as > Java application**.

Open your web browser and go to *http://localhost:8081/WeatherApplication/WeatherForecast.html*
Type something in the fields and click “Go”.

You should get something like:



Get weather forecast for

City: San Francisco   Country: United States   Go

-	Clouds	Temperature	Wind	Humidity	Pressure
Today		22 °F	1 km/h	55 %	1073 hPa
Tomorrow		49 °F	40 km/h	44 %	988 hPa
Sunday		31 °F	12 km/h	0 %	978 hPa
Monday		16 °F	18 km/h	74 %	897 hPa

Without modifying or stopping anything, retest the RMI client and make sure it's still working.

Stop the client and the application.
The RMI service is useless now. Delete it. To do that, open the diagram file, right-click on the RMI binding and select **Delete from model**. Do the same thing for the underlying service. Save the file.

4. Going further: using SCA properties

There is one service parameter we did not add in the HTML page: the temperature unit (Celsius or Fahrenheit degrees – actually, it was set to °F by default). Let the user decide which unit to use is a solution. Another one is to force it in the application.

The application implementation deals with both units.
Using an SCA property, we will force it to one value without modifying the code.

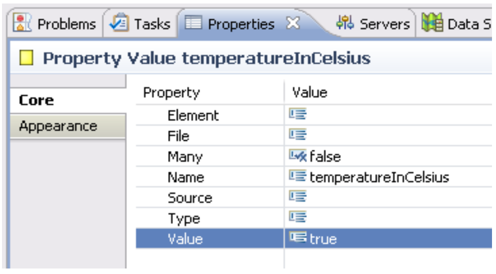
SCA properties

SCA properties are defined as a way to configure an SCA implementation.
Indeed, implementation code could be reusable in several components. A component by itself is just a configuration of this code: how it is exposed (which services), by which protocols can we call it (which bindings), which dependencies do we need (which references)... The component by itself is just a usage instance of its implementation code.

To make code reusable, you must be able to configure it according to the context.
SCA properties are intended for these cases. They allow you to specify one or several configurable aspects. A property can be simple (have a simple type, like string or integer), or complex (a custom class).

Let's add a property for the temperature unit.

- Open the application diagram.
- Create a property called *temperatureInCelsius* on *MainWeatherComponent*.
  - Use either the palette, or use the contextual toolbar.
  - In the properties view, set the Value field to true (indicating we will get Celsius degrees).



Property Value temperatureInCelsius

Property	Value
Element	
File	
Many	false
Name	temperatureInCelsius
Source	
Type	
Value	true

- Create a new interface in the ***.demo.interfaces** package.
  - Call it MainWeatherService2.java

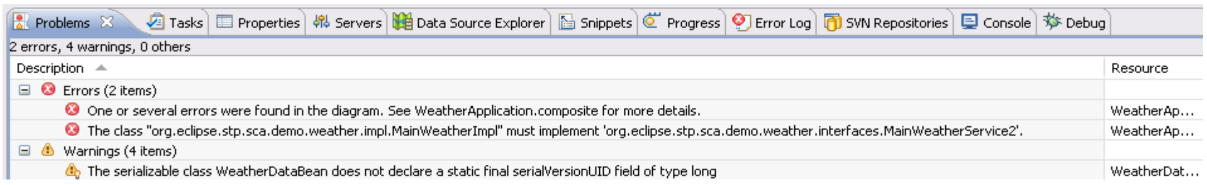
```
package org.eclipse.stp.sca.demo.weather.interfaces;

import java.util.List;

import org.eclipse.stp.sca.demo.simpleweather.interfaces.WeatherDataBean;

@Remotable
public interface MainWeatherService2 {
	public List<WeatherDataBean> getWeatherForecast( String city, String country );
}
```

- Copy the MainWeatherImpl.java file in MainWeatherImpl2.java.
  - Make it implement *MainWeatherService2* instead of *MainWeatherService*.
  - Add a property at the end of the code.

```
@Property
public boolean temperatureInCelsius;
```

- Replace the interface of *MainWeatherService*:
  - either right-click on the interface icon and select Set Java interface, or
  - edit the interface property with the properties view, or
  - open the composite file and edit the class name in the XML editor.

- You should now have an error marker on the implementation figure.



```
2 errors, 4 warnings, 0 others
Description                                                                                                                                 Resource
Errors (2 items)
    One or several errors were found in the diagram. See WeatherApplication.composite for more details.                                     WeatherAp...
    The class "org.eclipse.stp.sca.demo.weather.impl.MainWeatherImpl" must implement 'org.eclipse.stp.sca.demo.weather.interfaces.MainWeatherService2'.   WeatherAp...
Warnings (4 items)
    The serializable class WeatherDataBean does not declare a static final serialVersionUID field of type long                              WeatherDat...
```

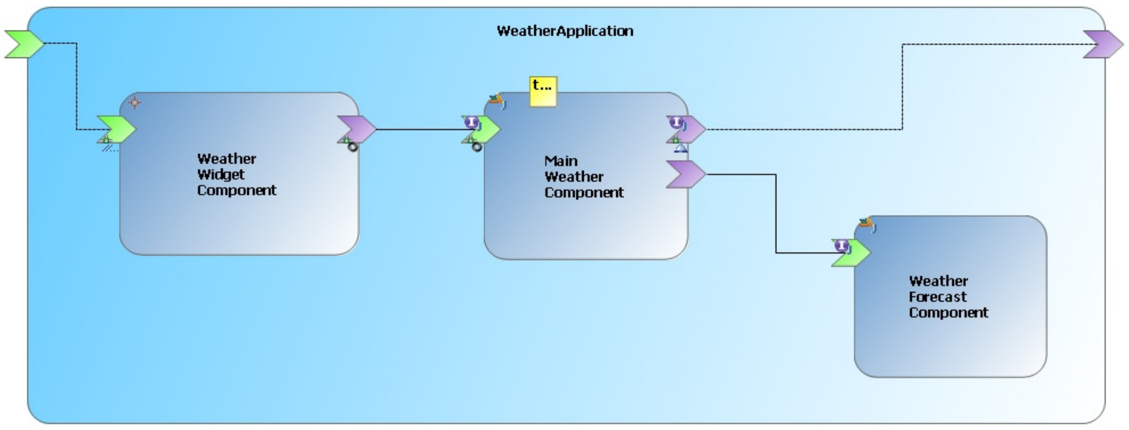
- Update the implementation class of *MainWeatherComponent*.
  - Use *MainWeatherImpl2* instead of *MainWeatherImpl*.
- Rename *MainWeatherService* in *MainWeatherService2* on the composite_diagram.
- And in WeatherForecast.html, replace

```
mainWeatherReference.getWeatherForecast( city, country, false, displayWeatherData )
```

by

```
mainWeatherReference.getWeatherForecast( city, country, displayWeatherData );
```

That's “all”.
The temperature unit will be initialized by the SCA container with the value given in the composite.



Run the application once again.
Right-click on WeatherApplicationDeployment.java and select **Run as > Java application**.

Open your web browser and go to *http://localhost:8081/WeatherApplication/WeatherForecast.html*
Refresh it if necessary and check that the temperature is now in Celsius degrees.

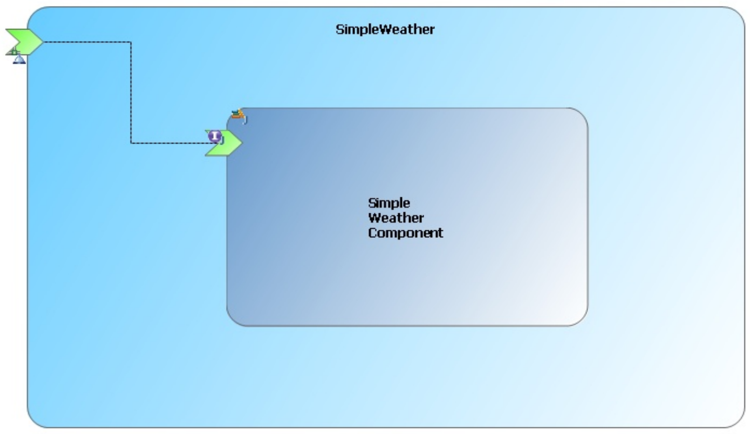
5. Going further: understand the SimpleWeatherApplication

The SimpleWeather application acts as the web service our application uses.
It is a composite with a single component and one service with a web service binding.

Create this application from zero.

- Create a new SCA project and a new SCA diagram.
- Add the component and the service mentioned above.
- Create a Java interface implementing the following contract
  - f : (String city, String country, boolean tempInCelsius) → Weather Java Bean
  - ... where a Weather Java bean is a Java structure you will define yourself.
- Create a Java implementation for the component and implementing this service.
- Add a web service binding on the composite service.
  - Do not forget to specify a URI for the WS binding.
- Bind the Java files with the composite.
- Create a deployment class.
  - Once it runs, open your web browser and go to the WS URI you entered with “?wsdl” appended to this URI: you should see the WSDL service interface.

Feel free to look at the code of the SimpleWeather application if you get in troubles.



6. Links and resources

SCA:

- Open SOA: http://www.osoa.org/display/Main/Home
- SCA by OASIS: http://www.oasis-opencsa.org/sca
- Introducing SCA, by David Chappell
  - http://www.davidchappell.com/articles/Introducing_SCA.pdf
- Apache Tuscany's guide to SCA: http://tuscany.apache.org/quick-guide-to-sca.html

SCA Tools:

- Eclipse STP website: http://www.eclipse.org/stp/
- SCA Tools website: http://www.eclipse.org/stp/sca/index.php
- SCA Tools wiki: http://wiki.eclipse.org/STP/SCA_Component

SCA platforms (or supporting SCA):

- Apache Tuscany: http://tuscany.apache.org/
- OW2 FraSCAti: http://frascati.ow2.org/
- Fabric3: http://www.fabric3.org/
- Newton: http://newton.codecauldron.org/site/index.html
- PEtALS: http://petals.ow2.org/
- Swordfish: http://www.eclipse.org/swordfish/

SCA tutorials:

- First steps with the SCA Composite Designer
  - http://wiki.eclipse.org/STP/SCA_Component/SCA_First_Steps_With_Composite_Designer
- Apache Tuscany - SCA store application
  - http://tuscany.apache.org/getting-started-with-tuscany.html
- Other Apache Tuscany tutorials
  - http://tuscany.apache.org/sca-java-getting-started-guides-1x.html