ADAMS

Advanced **D**ata mining **A**nd **M**achine learning **S**ystem

Module: adams-webservice



Peter Reutemann

June 23, 2015

DRAFT

©2012-2013



THE UNIVERSITY OF
WAIKATO
Te Whare Wānanga o Waikato



Except where otherwise noted, this work is licensed under
http://creativecommons.org/licenses/by-sa/3.0/

Contents

1 Introduction **7**

2 Creating a web-service **9**
2.1 Defining the WSDL 9
2.2 Creating a new module 9
2.3 Configuring code-generation 9
2.4 Implementing the web-service 10
2.4.1 Dummy 10
2.4.2 Server 11
2.4.3 Client 12
2.4.4 Callable transformer support 13

3 Flow **15**

Bibliography **17**

List of Figures

1.1 Webservice schema. 7

Chapter 1

Introduction

The power of web-services [2] lies in the fact that it decouples software frameworks and it is possible to communicate without having to worry about implementation or technology that each of the frameworks uses (see Figure 1.1).

The *adams-webservice* provides SOAP [5] web-service support using the Apache CXF framework [6]. ADAMS uses the approach to generate code on-the-fly based on a WSDL file [3]. This is also called *WSDL first*.

If you already have existing code, then you can use Apache CXF as well to generate the WSDL from your code. See [8] on how to do this.

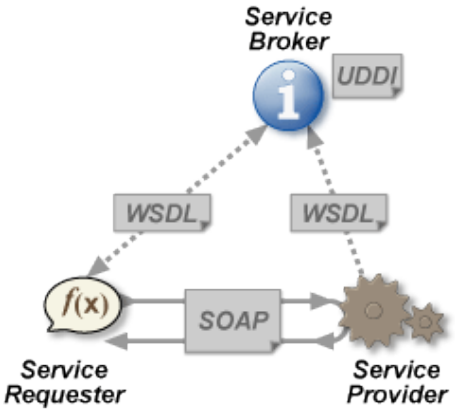


Figure 1.1: Webservice schema.

Chapter 2

Creating a web-service

Creating a web-service involves the following steps:

1. Define the WSDL for the web-service.
2. Create a new ADAMS module, add the *adams-webservice* artifacts as dependencies.
3. Place your WSDL in the `src/main/resources/wsdl` directory (best to create a sub-directory there, e.g., `mywsdl`).
4. Configure the code-generation using the WSDL as basis.
5. Implementing the web-service functionaliy that processes the data (server and client).

2.1 Defining the WSDL

If you start from scratch with a WSDL definition, check out the tutorial at [4]. On the other hand, if you already have existing code, then refer to [8] on how to generate a WSDL from your code.

2.2 Creating a new module

For more details on this, please refer to the manual of the *adams-core* module. In the *Developing with ADAMS* part, see section *Creating a new module*.

2.3 Configuring code-generation

Now you need to configure your *pom.xml* file to generate code from the WSDL on-the-fly. You need to configure the *cxf-codegen-plugin* build plugin, i.e., where the WSDL is located and what code to generate (you might have a bindings XML file as well).

The following `pom.xml` snippet shows how to do this for the Apache CXF *customer service* example (see "wsdl_first" example in the CXF download). The WSDL `CustomerService.wsdl` is located in `src/main/resources/wsdl/customerservice` and also comes with a bindings file `CustomerService-binding.xml` in the same directory. We want to to create *JAX-WS 2.1* compatible code out of it, which we define using the *-frontend* parameter. And since we want to create Java code from the WSDL, we need to use goal *wsdl2java*.

```
<plugin>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxf-codegen-plugin</artifactId>
  <executions>
    <execution>
      <id>generate-sources</id>
      <phase>generate-sources</phase>
      <configuration>
        <wsdlOptions>
          <wsdlOption>
            <wsdl>src/main/resources/wsdl/customerservice/CustomerService.wsdl</wsdl>
            <wsdlLocation>classpath:wsdl/customerservice/CustomerService.wsdl</wsdlLocation>
            <bindingFiles>
              <bindingFile>src/main/resources/wsdl/customerservice/CustomerService-binding.xml</bindingFile>
            </bindingFiles>
          </wsdlOption>
        </wsdlOptions>
        <extraargs>
          <extraarg>-frontend</extraarg>
          <extraarg>jaxws21</extraarg>
        </extraargs>
      </configuration>
      <goals>
        <goal>wsdl2java</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

2.4 Implementing the web-service

Depending on your web-service you might have to implement the following functionality:

- Server – processes requests and sends back responses
- Client – queries a server and processes the response

The following sections explain each in detail.

2.4.1 Dummy

In order to get a dummy implementation generated, you can use the `wsdl2java` tool that comes with the *Apache CXF* binary distribution. The dummy implementation is generated using the `-impl` parameter. Here is an example of generating JAX-WS 2.1 compatible output for the customer service example WSDL:

```
./apache-cxf-2.6.3/bin/wsdl2java
    -impl
    -server
    -client
    -fe jaxws21
    -wsdlLocation "classpath:wsdl/customerservice/CustomerService.wsdl"
    CustomerService.wsdl
```

The `-wsdlLocation` parameter indicates where the Apache CXF code will find the WSDL at runtime. This gets added to the `wsdlLocation` attribute of the `javax.jws.WebService` annotation.

Three generated files are basically of interest here:

- *CustomerService_CustomerServicePort_Client.java* – client code for that shows how to query the webservice.
- *CustomerService_CustomerServicePort_Server.java* – server code for starting the web-service, makes use of *CustomerServiceImpl.java*.

- *CustomerServiceImpl.java* – the dummy implementation that processes the incoming request from the client and generates a response.

Parts of this example code will be used in implementing plugins for ADAMS.

2.4.2 Server

Your server component needs to implement the following interface:

`adams.flow.webservice.WebServiceProvider`

It is easiest to implement your class in the same package, otherwise you need to tell ADAMS where to find your classes (see section *Dynamic class discovery* in the *adams-core* manual).

For convenience, you can simply sub-class the following abstract superclass:

`adams.flow.webservice.AbstractWebServiceProvider`

If you need to process the query data with a sub-flow, see section 2.4.4.

Using the example code generated earlier, we can now implement out server component. You only need to specify the default URL that the web-service gets published under (`getDefaultURL()`), how to start the web-service (`doStart()`) and how to stop it again (`doStop()`).

Here is a basic implementation[1]:

```
public class CustomerServiceWS extends AbstractWebServiceProvider {

  protected EndpointImpl m_Endpoint;

  @Override
  public String globalInfo() {
    return "Provides a customer service webservice.";
  }

  @Override
  public String getDefaultURL() {
    return "http://localhost:9090/CustomerServicePort";
  }

  @Override
  protected void doStart() throws Exception {
    m_Endpoint  = (EndpointImpl) Endpoint.publish(getURL(), new CustomerServiceImpl(this));
  }

  @Override
  protected void doStop() throws Exception {
    if (m_Endpoint != null) {
      m_Endpoint.getServer().stop();
      m_Endpoint = null;
    }
  }
}
```

As you can see, we make use of the dummy implementation created by Apache CXF in the `doStart()` method:

```
new CustomerServiceImpl(this)
```

By default, the "dummy implementation" class (*CustomerServiceImpl*) only has a default constructor. In order to give it access to the webservice and integrate it with ADAMS, we need to add a custom constructor, to link it to the webservice that it uses:

[1] com.example.customerservice.flow.CustomerServiceWS, adams-webservice-server.flow

```
/** the ADAMS owner. */
protected CustomerServiceWS m_Owner;

/** Initializes the service. */
public CustomerServiceImpl(CustomerServiceWS owner) {
  super();
  m_Owner = owner;
}
```

2.4.3 Client

For the client component, you need to implement this interface:

`adams.flow.webservice.WebServiceClient`

Depending on whether your client can take input or generates output, you need to implement the following interfaces as well:

- *WebServiceClientConsumer* – accepts input, which will get forwarded to the web-service.
- *WebServiceClientProducer* – generates output based on the webs-service response.

In order to make development of new web-service components faster, a bunch of abstract classes is already available:

- *AbstractWebServiceClientSource* – for a client that outputs the web-service reponse[2].
- *AbstractWebServiceClientTransformer* – the client queries the web-service with data it receives and outputs the web-service reponse[3].
- *AbstractWebServiceClientSink* – sends the incoming data to the web-service and generates no output[4].

If some of the data requires pre- or post-processing with a sub-flow, check out section 2.4.4 for more details on how to do this.

Using the previous customer service example again, we can now generate a transformer client that queries the web-service with a customer name and then outputs the results. You need to implement the following methods:

- `getWsdlLocation()` – where to find the WSDL.
- `setRequestData(T)` – for setting the query data.
- `doQuery()` – how to perform the query.
- `hasResponseData()` – whether there is any response data available.
- `getResponseData()` – for obtaining the response data.
- `accepts()` – what type of input data the client accepts.
- `generates()` – what type of output data the client generates.

Here is an example implementation[5]:

[2] adams-webservice-client_as_source.flow
[3] adams-webservice-client_as_transformer.flow
[4] adams-webservice-client_as_sink.flow
[5] com.example.customerservice.flow.CustomersByName

```
public class CustomersByName extends AbstractWebServiceClientTransformer<String,String> {

  protected String m_CustomerName;  // the name of the customers to look up
  protected String m_ProvidedCustomerName;  // the customer name received by actor
  protected List<Customer> m_Customers;  // the list of customers that were obtained from webservice

  @Override
  public String globalInfo() {
    return "Returns customer names.";
  }

  @Override
  public Class[] accepts() {
    return new Class[]{String.class};
  }

  @Override
  public void setRequestData(String value) {
    m_ProvidedCustomerName = value;
  }

  @Override
  protected URL getWsdlLocation() {
    return getClass().getClassLoader().getResource("wsdl/customerservice/CustomerService.wsdl");
  }

  @Override
  protected void doQuery() throws Exception {
    m_Customers = null;
    CustomerServiceService customerServiceService = new CustomerServiceService(getWsdlLocation());
    CustomerService customerService = customerServiceService.getCustomerServicePort();
    WebserviceUtils.configureClient(customerService, m_ConnectionTimeout, m_ReceiveTimeout);
    m_Customers = customerService.getCustomersByName(m_ProvidedCustomerName);
    m_ProvidedCustomerName = null;
  }

  @Override
  public Class[] generates() {
    return new Class[]{String.class};
  }

  public boolean hasResponseData() {
    return (m_Customers != null) && (m_Customers.size() > 0);
  }

  @Override
  public String getResponseData() {
    String result = m_Customers.get(0).getCustomerId() + ": " + m_Customers.get(0).getName()
        + ", " + Utils.flatten(m_Customers.get(0).getAddress(), " ");
    m_Customers.remove(0);
    return result;
  }
}
```

2.4.4 Callable transformer support

In order to utilize the power of the workflow, you can also process data, in the server and client alike, using a callable transformer. If your component needs to take advantage of such functionality, it needs to implement the `CallableTransformerSupport` interface. For convenience, there are already abstract classes in place:

- *AbstractWebServiceProviderWithCallableTransformer* – if the server needs to process the incoming data with a sub-flow.
- *AbstractWebServiceClientSourceWithCallableTransformer* – a source client that pre-processes the query or post-processes the response.
- *AbstractWebServiceClientTransformerWithCallableTransformer* – a trans-

former client that post-processes the response[6].
- *AbstractWebServiceClientSinkWithCallableTransformer* – a sink client that pre-processes the incoming data before querying the service.

You can use the `applyTransformer` method to transform the data.

[6] adams-webservice-client_as_transformer-callable_transformer.flow

Chapter 3

Flow

This module contains generic actors in which you can simply plug your web-services that you have implemented. In the following a short overview.

The following standalones are available:

- *WSServer* – runs a web-service. waiting for requests[1].

The following sources are available:

- *WSSource* – queries a web-service and forwards the received data[2].

The following transformers are available:

- *WSTransformer* – sends the data it receives to a web-service and forwards the data from the response in turn[3][4].

The following sinks are available:

- *WSSink* – simply sends data to a web-service[5].

[1] adams-webservice-server.flow
[2] adams-webservice-client_as_source.flow
[3] adams-webservice-client_as_transformer.flow
[4] adams-webservice-client_as_transformer-callable_transformer.flow
[5] adams-webservice-client_as_sink.flow

Bibliography

[1] *ADAMS* – Advanced Data mining and Machine learning System
https://adams.cms.waikato.ac.nz/

[2] *Web service* – a method of communication between two electronic devices over the World Wide Web
http://en.wikipedia.org/wiki/Web_service

[3] *WSDL* – Web Services Description Language
http://en.wikipedia.org/wiki/Web_Services_Description_Language

[4] *WSDL Tutorial* – w3c school's tutorial on WSDL
http://www.w3schools.com/wsdl/

[5] *SOAP* – Simple Object Access Protocol
http://en.wikipedia.org/wiki/SOAP

[6] *Apache CXF* – an open source services framework
http://cxf.apache.org/

[7] *Apache CXF* – sample projects (included in the download)
http://cxf.apache.org/docs/sample-projects.html

[8] *Contract first* – how to generate a WSDL from code with Apache CXF
http://cxf.apache.org/docs/defining-contract-first-webservices-with-wsdl-generation-from-java.html

[9] *CXF Code generation* – examples on how to use the Maven *cxf-codegen-plugin*
http://cxf.apache.org/docs/maven-cxf-codegen-plugin-wsdl-to-java.html

[10] *wsdl2java* tool – generates Java code from a WSDL
http://cxf.apache.org/docs/wsdl-to-java.html