

# ADAMS

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

Module: adams-rest



Peter Reutemann

January 8, 2020

©2018-2019



THE UNIVERSITY OF  
**WAIKATO**  
*Te Whare Wānanga o Waikato*



Except where otherwise noted, this work is licensed under  
<http://creativecommons.org/licenses/by-sa/4.0/>

# Contents

<b>1</b>	<b>Introduction</b>	<b>5</b>
<b>2</b>	<b>Using REST</b>	<b>7</b>
2.1	Client . . . . .	7
2.1.1	Custom code . . . . .	7
2.1.2	Generic flow . . . . .	8
2.2	Server . . . . .	9
2.2.1	Custom class . . . . .	9
2.2.2	Generic server . . . . .	10
2.2.3	Context . . . . .	11
2.2.4	Available plugins . . . . .	11
<b>3</b>	<b>Flow</b>	<b>13</b>
3.1	TLS support . . . . .	13
	<b>Bibliography</b>	<b>15</b>



# Chapter 1

## Introduction

REST webservices ([3]) are a popular variant of webservices, that are quite often easier to implement than full-blown SOAP-based ones ([4]).

ADAMS provides a general framework for accessing and implementing REST webservices using Apache CXF[5].



## Chapter 2

# Using REST

The following sections describe how you can access REST webservices (*client*) and write your own ones (*server*).

### 2.1 Client

There are two options for accessing a webservice: custom code for sending/receiving data or via generic processing in the flow itself.

#### 2.1.1 Custom code

When using custom code, you can use one of the following superclasses to derive your own code from:

- `adams.flow.rest.AbstractRESTClientSource`
- `adams.flow.rest.AbstractRESTClientTransformer`
- `adams.flow.rest.AbstractRESTClientSink`

These classes are generics and require you to supply the input and/or output types, override methods for integrating them in the flow (accepts/generates) and methods for handling input/output data. These superclasses are available in the flow through the following, corresponding actors:

- `adams.flow.source.RESTSource`
- `adams.flow.transformer.RESTTransformer`
- `adams.flow.sink.RESTSink`

The `adams.flow.rest.echo.EchoClientTransformer` class is a simple example that sends a UTF-8 string it receives to an Echo REST server, which simply returns the same data. In the `doQuery` method, a URL is constructed from the URL of the echo server and the URL-encoded string that is to be sent to the server. The actual sending via the `GET` method is handled by the `adams.core.net.HttpRequestHelper` class. The response from the echo server is then decoded from a UTF-8 string and forwarded in the flow.

```

package adams.flow.rest.echo;
import adams.core.base.BaseURL;
import adams.core.net.HttpRequestHelper;
import adams.flow.container.HTMLRequestResult;
import adams.flow.rest.AbstractRESTClientTransformer;
import org.jsoup.Connection.Method;
import java.net.URLDecoder;
import java.net.URLEncoder;

public class EchoClientTransformer
    extends AbstractRESTClientTransformer<String,String> {

    protected String m_RequestData;

    public String globalInfo() { return "Client (transformer) for Echo REST service."; }

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

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

    public void setRequestData(String value) { m_RequestData = value; }

    protected void doQuery() throws Exception {
        String url;
        if (getUseAlternativeURL())
            url = getAlternativeURL();
        else
            url = new EchoServer().getDefaultURL();
        url += "echo/" + URLEncoder.encode(m_RequestData, "UTF-8");
        HTMLRequestResult result = HttpRequestHelper.send(new BaseURL(url), Method.GET, null, null);
        if (result.getValue(HTMLRequestResult.VALUE_STATUSCODE, Integer.class) == 200)
            setResponseData(URLDecoder.decode(result.getValue(HTMLRequestResult.VALUE_BODY, String.class), "UTF-8"));
        else
            m_LastError = result.getValue(HTMLRequestResult.VALUE_STATUSCODE) + ": "
                + result.getValue(HTMLRequestResult.VALUE_STATUSMESSAGE);
    }
}

```

### 2.1.2 Generic flow

Using the above example of accessing an echo server, we can simply use existing component available through the flow<sup>1</sup>:

- The string to be sent to the server would be encoded via the `URLEncode` conversion.
- This string would then be prefixed with the URL of the actual server (e.g., `http://localhost:8080/echo/`) to construct the complete URL.
- Via the `HTTPRequest` source, you can then connect to the complete URL. Choose the correct method for accessing, like `GET` or `POST`. This source actor allows you to attach additional headers and parameters as key-value pairs to your request. Cookies are accessed through storage, expecting a map object.
- Using the `ContainerValuePicker` control actor, the response returned by the echo server can be retrieved.
- With the `URLDecode` conversion, the URL encoded response can be turned into a regular string again.

---

<sup>1</sup>See example flow: `adams-rest-use_service.flow`



## 2.2 Server

For implementing the server side, you can use two different approaches:

- Custom class to encapsulate complete server
- Creating fine-grained plugins for the **GenericServer**

For both approaches, it is recommended to look at the some of the tutorials listed on the Wikipedia article on JAX-RS[6], the Java API for RESTful Web Services, to get an understanding on how to code for REST.

JAX-RS handles everything, methods (GET/POST), paths and path parameters through Java annotations, which you will see in the following examples.

### 2.2.1 Custom class

In order to make the custom REST service available through the **RESTServer** standalone, it needs to be either sub-classed from **AbstractRESTProvider** or implement **RESTProvider** interface (both are located in `adams.flow.rest`). In the following the use of **AbstractRESTProvider** is discussed, as it simplifies the implementation and supports handling for custom URLs, interceptors, etc.

The `doStart` method configures an **JAXRSServerFactoryBean** (located in package `org.apache.cxf.jaxrs`) instance and returns the **Server** (package `org.apache.cxf.endpoint`) generated from it. The functionality to be offered by the REST service is defined by the service bean that is set via the `setServiceBean` method or the beans that are set via the `setServiceBeans` method. Optional TLS support, i.e., serving content via *https* is configured by calling the `configureTLS` method with the configured factory bean (see 3.1 for more details).

```
package adams.flow.rest;

import adams.core.Utils;
import adams.flow.rest.AbstractRESTProvider;
import org.apache.cxf.endpoint.Server;
import org.apache.cxf.jaxrs.JAXRSServerFactoryBean;
import somewhere.Echo;

public class EchoServer extends AbstractRESTProvider {

    public String globalInfo() {
        return "Simple echo server.";
    }

    public String getDefaultURL() {
        return "http://localhost:8080/";
    }

    protected Server doStart() throws Exception {
        JAXRSServerFactoryBean factory = new JAXRSServerFactoryBean();
        configureInterceptors(factory);
        factory.setServiceBean(new Echo());
        factory.setAddress(getURL());
        configureTLS(factory);
        return factory.create();
    }
}
```

```

package somewhere;

public class Echo {

    @GET
    @Path("/echo/{input}")
    @Produces("text/plain")
    public String ping(@PathParam("input") String input) {
        return input;
    }
}

```

### 2.2.2 Generic server

The light-weight approach of REST and being able to just supply an arbitrary number of service beans, makes it possible to break up monolithic servers into micro-servers and opening up the possibility of given the user the choice in the flow over what functionality the server should have. For that purpose, the **GenericServer** (package `adams.flow.rest`) was developed. In order to add functionality to this REST server, plugins need to be developed. These plugins either just implement the **RESTPlugin** interface or are sub-classed from **AbstractRESTPlugin** (also in package `adams.flow.rest`).

The following code shows the plugin for our echo server, which would be selected as plugin in our **GenericServer**, which in turn is configured through the **RESTServer** standalone.

```

package adams.flow.rest;

import adams.flow.rest.AbstractRESTPlugin;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;

public class Echo extends AbstractRESTPlugin {

    public String globalInfo() {
        return "Simple echo of the input.";
    }

    @GET
    @Path("/echo/{input}")
    @Produces("text/plain")
    public String ping(@PathParam("input") String input) {
        getLogger().info("input: " + input);
        return input;
    }
}

```

There are more abstract superclasses available for plugins, to avoid unnecessary duplication:

- *AbstractRegisteredFlowRESTPlugin* – allows retrieval of registered flows via their ID.
- *AbstractRESTPluginWithDatabaseConnection* – enables database access via its flow context and the available *DatabaseConnection* standalone.
- *AbstractRESTPluginWithFlowContext* – for any plugin that requires flow context, e.g., for accessing configurations.

### 2.2.3 Context

If your REST service components require a context, e.g., flow context, then you have to use the following approach for setting up the service factory<sup>2</sup>:

```
JAXRSServerFactoryBean sf = new JAXRSServerFactoryBean();
CustomerService cs = new CustomerService();
// HERE: set context in CustomerService service bean
sf.setServiceBean(cs);
sf.setAddress("http://localhost:9080/");
sf.create();
```

By instantiating the *beans* yourself rather than through the factory, you can provide them with context. You then use the *setServiceBean(s)* methods to set one or more beans.

### 2.2.4 Available plugins

The following concrete `RESTPlugin` implementations are available:

- *control.StopFlow* – stops the registered flow with the specified ID
- *echo.Echo* – simply sends back the string received
- *flow.CallableJsonPipeline* – for processing JSON strings with a processing pipeline template (enforces JSON data)
- *flow.CallableTextPipeline* – for processing arbitrary strings with a processing pipeline template
- *flow.CallableJsonTransformer* – for processing JSON strings with a callable transformer (enforces JSON data)
- *flow.CallableTextTransformer* – for processing arbitrary strings with a callable transformer

---

<sup>2</sup><http://cxf.apache.org/docs/jaxrs-services-configuration.html>



## 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 conversions are available:

- *JsonToObject* – maps a JSON string into a Java object (using Jackson’s Databind functionality[7]).
- *ObjectToJson* – turns any Java object into a JSON string (using Jackson’s ObjectMapper[7]).

The following standalones are available:

- *RESTServer* – runs a web-service. waiting for requests<sup>1</sup>.

The following sources are available:

- *RESTSource* – queries a web-service and forwards the received data<sup>2</sup>.

The following transformers are available:

- *RESTTransformer* – sends the data it receives to a web-service and forwards the data from the response in turn<sup>3</sup>.

The following sinks are available:

- *RESTSink* – simply sends data to a web-service<sup>4</sup>.

### 3.1 TLS support

TLS support is automatically configured in case the URL uses `https://` as protocol and the `RESTUtils.configureClient` method is called. For the server side, you need to call the `AbstractRESTProvider.configureTLS` method after the URL has been set for the `JAXRSServerFactoryBean` factory instance. The following standalones have to be present (and configured) to successfully set up TLS support:

---

<sup>1</sup>adams-rest-server.flow

<sup>2</sup>adams-rest-echo\_source.flow

<sup>3</sup>adams-rest-echo\_transformer.flow

<sup>4</sup>adams-rest-echo\_sink.flow

- *KeyManager*
- *TrustManager*
- *SSLContext* – optional, only used to determine the version of the TLS protocol (fallback is *TSL*).

See the following flows for example setups using self-signed certificates:

- *adams-rest-generic\_server-ssl.flow* – echo server via https
- *adams-rest-use\_service-ssl.flow* – access the echo server

The `README.md` file in the `${FLOWS}/restssl` directory contains more information on how these self-signed certificates were generated.

# 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](http://en.wikipedia.org/wiki/Web_service)
- [3] *REST* – Representational state transfer  
[http://en.wikipedia.org/wiki/Representational\\_state\\_transfer](http://en.wikipedia.org/wiki/Representational_state_transfer)
- [4] *SOAP* – Simple Object Access Protocol  
<http://en.wikipedia.org/wiki/SOAP>
- [5] *Apache CXF* – an open source services framework  
<http://cxf.apache.org/>
- [6] *JAX-RS* – Java API for RESTful Web Services  
[https://en.wikipedia.org/wiki/Java\\_API\\_for\\_RESTful\\_Web\\_Services](https://en.wikipedia.org/wiki/Java_API_for_RESTful_Web_Services)
- [7] *Jackson data-binding* – for JSON and other formats  
<https://github.com/FasterXML/jackson-databind>