Web Services

Table of Contents

[Working Principle 2](#_Toc100617612)

[Core Concepts 3](#_Toc100617613)

[REST 3](#_Toc100617614)

[Architectural Constraints 4](#_Toc100617615)

[Key Elements 5](#_Toc100617616)

[Programming with REST 6](#_Toc100617617)

[Return Types 7](#_Toc100617618)

[Multiple Methods 8](#_Toc100617619)

[Path Parameters and Query Parameters 9](#_Toc100617620)

[SOAP 10](#_Toc100617621)

[Building Blocks 10](#_Toc100617622)

[SOAP vs REST 11](#_Toc100617623)

A **web service** is a standardized medium to allow clients to communicate with server applications. Essentially, a web service is a software module that is designed to perform a certain set of tasks. Clients can search for and invoke the web service, which in turn will provide some functionality to the client.

A web service uses a combination of some other standards, namely XML, SOAP, WSDL and UDDI. XML is used to store the data, SOAP is used to transfer it, WSDL is used to describe the available services and UDDI is used to list the services that are available.

A **Web API** on the other hand, is an improvement over web services which provides a simplistic way to handle the web service without relying on XML based web service protocols like SOAP and WSDL. The specific web API we will be looking into is REST, which is the most common.

## Working Principle

The working principle of a web service is just a client server architecture. The client sends a request to a specific **endpoint** (URL). These requests are made through **remote procedure calls** (RPCs), which are calls made to methods hosted by the relevant web service.

For example, Amazon hosts a web service which provides the prices of all its products. Anyone can build a front-end in any language or platform and give users the ability to use this web service.

## Core Concepts

The main component of a web service is the **data** transferred between client and server, which is usually in XML or JSON format. This provides a **common platform**, freeing us from the issues with different programming languages or platforms being used at the two ends. The data is sent over normal HTTP.

There are two types of web services, SOAP and REST, both of which we will be covering soon. Regardless of which type we use, a few things are common:

1. There will be a **class** which has **methods** which will provide some functionality to the client.
2. The methods will be **exposed** on the network and it will be specified how and where clients can access them.
3. Information will be transferred in a language and platform **independent** manner.

## REST

**Representational State Transfer** (REST) is a software architecture that defines a set of constraints to be used to create web services. Web services that follow this style are said to be RESTful web services. They provide **interoperability** between systems on the internet.

In RESTful implementations, all types of data (records, objects, etc.) are considered to be **resources**. Requesting systems can access and manipulate **textual representations** of these resources (e.g. via JSON, XML, etc.) by using a uniform and predefined set of **stateless operations**.

Unlike SOAP, REST is a style, not a protocol. This means there is no official‘standard’. However, RESTful implementations still use standards, such as HTTP, URI, JSON and XML.

### Architectural Constraints

1. **Client-Server Architecture** – This provides a clear separation of concerns between client logic (the view) and server logic (the business logic).
2. **Statelessness** – REST does not maintain state information for the client. Any required information must be sent by the client.
3. **Cacheability** – Some resources are of a nature that they must be requested repeatedly by the same client. These resources should be marked as cacheable, so that the client knows to cache them and not request them repeatedly.
4. **Layered System** – Adding more layers between a client and a server should not affect the end goal of a request. Essentially, the client should not be aware of anything happening between the request being made and the response arriving.
5. **Uniform Interface** – It is essential to use an interface that can identify resources through requests, manipulate resources through their representations and are self-descriptive. This ensures that the client does not need to know the details of how the resources are implemented to be able to fully use them.

### Key Elements

* **Resources** – In REST, each piece of data is considered to be a resource. These resources are accessed via specific URLs. Suppose we want to access the employee record of the employee with ID 1. The URL for this could be like this: [www.example.com/employee/1](http://www.example.com/employee/1).
* **Request Verbs** – These refer to the request methods, GET, POST, PUT and DELETE.
* **Request Headers** – The headers sent along with the request define the type of response required, authorization details, etc.
* **Request Body** – This contains additional parameters and is mainly used with POST requests.
* **Response Body** – This contains the data requested, mainly in JSON or XML form.
* **Response Status Code** – These are the general response codes sent by HTTP.

## Programming with REST

To create a RESTful application, we will be using the **REST service** as a template and the [WildFly](https://www.wildfly.org/downloads/) application server. The REST service template uses the JAX-RS library. The WildFly application server works in the same way that the Tomcat server does, but it has built in support for REST, something that Tomcat does not.

Every RESTful application has two files to start with, the Application file and the Resource file.

The **Application** file just defines the path at which the RESTful services are to be provided. This is actually a servlet.

import javax.ws.rs.ApplicationPath;  
import javax.ws.rs.core.Application;  
  
@ApplicationPath("/api")  
public class HelloApplication extends Application {  
  
}

JAVA

The **Resource** file defines the path at which a specific method can be accessed.

@Path("/hello-world")  
public class HelloResource {  
 @GET  
 @Produces("text/plain")  
 public String hello() {  
 return "Hello, World!";  
 }  
}

JAVA

This URL is unique to the resource. Thus, we can access the above method by visiting the URL <http://localhost:8080/api/hello-world>. This URL is called an **endpoint**. Clients can thus access a resource using an endpoint (known as **consuming** a resource), to which the server will respond (called **producing** a resource).

### Return Types

The @Produces annotation is used to define the data type of the returned data. If we wish to avoid using hardcoded strings, we can do this as well.

@Produces(MediaType.*TEXT\_PLAIN*)

JAVA

We can also return other types of data. The most common data types are JSON and XML.

@Produces(MediaType.*APPLICATION\_JSON*)

JAVA

The above code will cause whatever data we return to be converted to the JSON format. This gives us loads of flexibility, since we can return objects and arrays.

If instead we want to take an object as input, we can use the @Consumes annotation.

@Consumes(MediaType.*APPLICATION\_JSON*)  
public void someMethod(*JsonObject* someObject) {}

JAVA

This makes sense if the RESTful endpoint is being accessed from a servlet and that servlet wants to pass a Java object using JSON.

String url = "/path/to/method";  
*Client* client = ClientBuilder.*newClient*();  
*WebTarget* target = client.target(url);  
target.request().post(Entity.*entity*(object, MediaType.*APPLICATION\_JSON*));

JAVA

### Multiple Methods

Above, we saw a single method under a resource. This is the **default method**. We can actually have **multiple methods** under a single resource.

@Path("/resource")  
public class HelloResource {  
 @GET  
 @Produces(MediaType.*TEXT\_PLAIN*)  
 public String hello() {  
 return "Hello, World!";  
 }

@GET  
 @Path("/other-method")  
 @Produces(MediaType.*APPLICATION\_JSON*)  
 public Object someMethod() { return new Object(); }  
}

JAVA

The new method’s URL is <http://www.example.com/api/resource/other-method>.

### Path Parameters and Query Parameters

Parameters can be passed to the server in two ways using a URL, as Path Parameters and as Query Parameters.

We have seen **query parameters** before. They are passed as key value pairs in the format <http://www.example.com/api/resource?key1=value1&key2=value2>. On the server end, we process these like this:

@GET  
@Produces(MediaType.*TEXT\_PLAIN*)  
public String doGreeting(@QueryParam("nameParam") String name) {  
 return "Hello, " + name;  
}

JAVA

**Path parameters** are similar, but they do not use key value pairs. The values are passed directly as <http://www.example.com/api/resource/value1/value2>. On the server end, we process these like this:

@GET  
@Produces(MediaType.*TEXT\_PLAIN*)  
@Path("/{nameParam}")  
public String doGreeting(@PathParam("nameParam") String name) {  
 return "Hello, " + name;  
}

JAVA

Between the two, query parameters are recommended.

## SOAP

The **Simple Object Access Protocol** (SOAP) uses XML to allow language and platform independent data transfer. It is a standard which specifies how to use XML for data exchange. Since this uses XML, it is **lightweight**. SOAP works on HTTP.

### Building Blocks

The web service and the client application exchange **SOAP messages**. When a client makes a request, a SOAP message is generated. The response from the server is also a SOAP message. The SOAP message is just an XML document with the following components:

* **Envelope** – This is the element which identifies the XML as a SOAP message. It is the root element.
* **Header** – This element contains header information like authentication credentials or definitions of complex objects. SOAP messages support both simple parameters like strings and integers and also complex objects.
* **Body** – This element contains the call and response information, i.e. the actual data being transferred.
* **Fault** – This optional element is applicable only for responses. If there is a failure when generating the SOAP response, this element describes the error.

### SOAP vs REST

|  |  |
| --- | --- |
| SOAP stands for Simple Object Access Protocol. | REST stands for Representational State Transfer. |
| SOAP is a protocol. SOAP was designed with a specification. It includes a WSDL file which has the required information on what the web service does in addition to the location of the web service. | REST is an Architectural style in which a web service can only be treated as a RESTful service if it follows the aforementioned architectural constraints. |
| SOAP cannot make use of REST since SOAP is a protocol and REST is an architectural pattern. | REST can make use of SOAP as the underlying protocol for web services, because in the end it is just an architectural pattern. |
| SOAP uses service interfaces to expose its functionality to client applications. In SOAP, the WSDL file provides the client with the necessary information which can be used to understand what services the web service can offer. | REST uses Uniform Service locators to access to the components on the hardware device. For example, if there is an object which represents the data of an employee hosted on a URL as <http://demo.guru99>, some examples of URI that might be used to access them are:  <http://demo.guru99.com/Employee>  <http://demo.guru99.com/Employee/1> |
| SOAP requires more bandwidth for its usage. Since SOAP Messages contain a lot of information inside of it, the amount of data transferred using SOAP is generally a lot.  <?xml version="1.0"?> <SOAP-ENV:Envelope xmlns:SOAP-ENV=<http://www.w3.org/2001/12/soap-envelope> SOAP-ENV:encodingStyle ="http://www.w3.org/2001/12/soap-encoding">  <soap:Body>  <Demo.guru99WebService xmlns="http://tempuri.org/">  <EmployeeID>int</EmployeeID>  </Demo.guru99WebService>  </soap:Body> </SOAP-ENV:Envelope>  XML | REST does not need much bandwidth when requests are sent to the server. REST messages mostly just consist of JSON messages.  {"city":"Mumbai","state":"Maharastra"}  JSON |
| SOAP can only work with the XML format. As seen from SOAP messages, all data passed is in the XML format. | REST permits different data formats such as Plain text, HTML, XML, JSON, etc. The most preferred format for transferring data is JSON. |