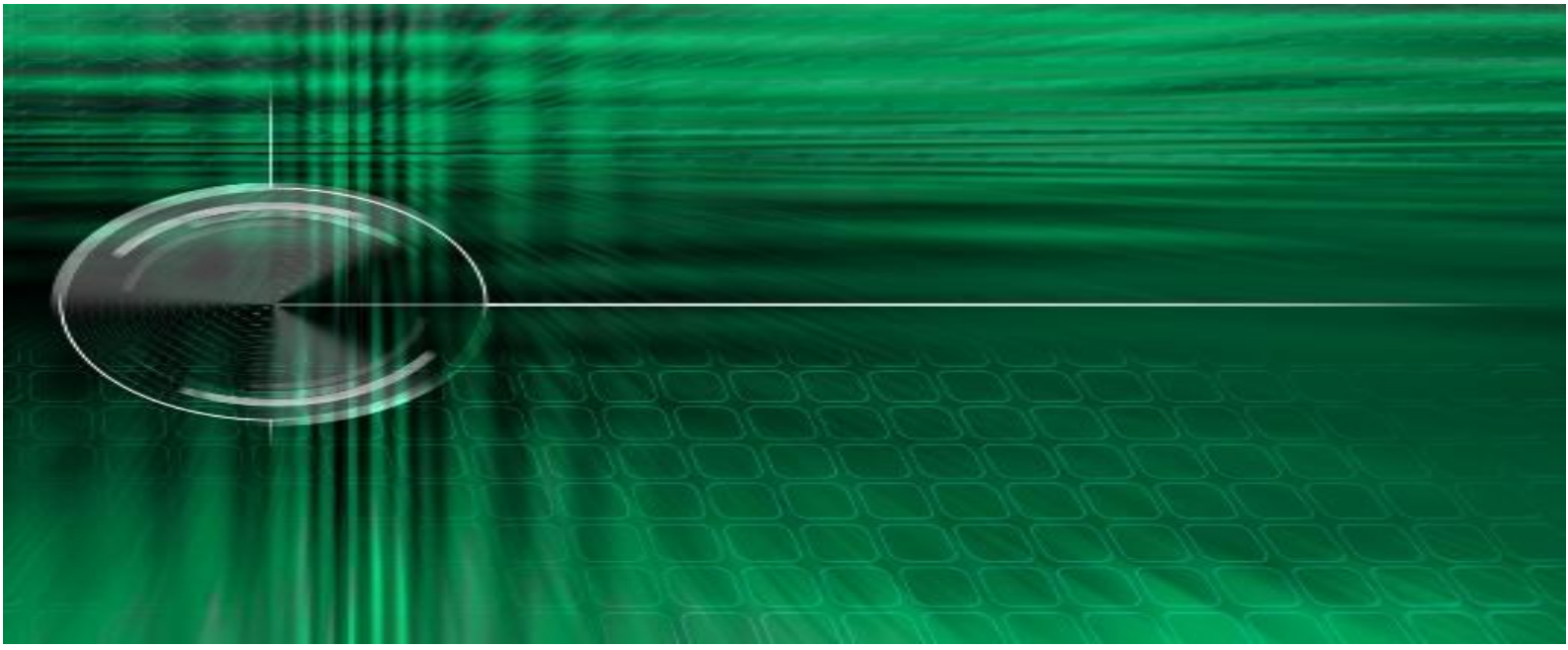
Software Engineering
Competence Center

TUTORIAL

Development and Deployment of REST Web Services in JAVA

An example for Android-based clients



Ahmed Mohamed Gamaleldin
Senior R&D Engineer-SECC
ahmed.gamal.eldin@itida.gov.eg

Abstract

Service-oriented architecture (SOA) is a set of principles and methodologies for designing and developing software in the form of interoperable services. These services are usually representing business functionalities that are built as software components, which can be reused for different purposes.
Web services have taken the concept of services delivered over the web using technologies such as eXtensible Markup Language (XML), Web Services Description Language (WSDL), Simple Object Access Protocol (SOAP), and the Universal Description, Discovery, and Integration (UDDI). In general, web services require an architectural in order to be implemented and used; architectural style is defined as "a set of design rules that identify the kinds of components and connectors that may be used to compose a system or subsystem"[1].

This tutorial presents an introduction to a widely used architectural style for web services, namely, the Representational State Transfer (REST). REST is seen as a simpler alternative to SOAP and Web Services Description Language (WSDL-based) Web services. This tutorial demonstrates the implementation of the REST services for Android platform. The choice of the Android platform is due to the increasing rate of adopting mobile platforms in the services world.

Keywords: Service Oriented Architecture (SOA), Web Services Description Language (WSDL), Simple Object Access Protocol (SOAP), Universal Description, Discovery, and Integration (UDDI), Representational State Transfer (REST)

Table of Contents

1. Introduction

A Service Oriented Architecture (SOA) is a design approach for building business applications as a set of loosely coupled black box components orchestrated to deliver a well-defined level of service by linking together business processes [1]. Moreover, SOA helps to reuse existing software assets where new services can be created from existing applications and IT infrastructures.

The SOA approach is the best way to achieve business agility which is the ability to change the business process quickly in response to the change in the business environment, such as adding a new service to the organization portfolio. The flexibility in SOA comes from the features that SOA paradigm provides [1], these features include:

1. Services can be viewed as software components with well-defined interfaces and implementation-independent. An important aspect of SOA is the separation of the service interface from its implementation. Services are consumed by clients that are not aware of how these services are implemented or concerned with how these services will execute their requests (Figure 1. shows abstract services model).
2. Services are designed to be self-contained and loosely coupled.
3. Services can be dynamically discovered.
4. Composite services can be built from composition of other services.

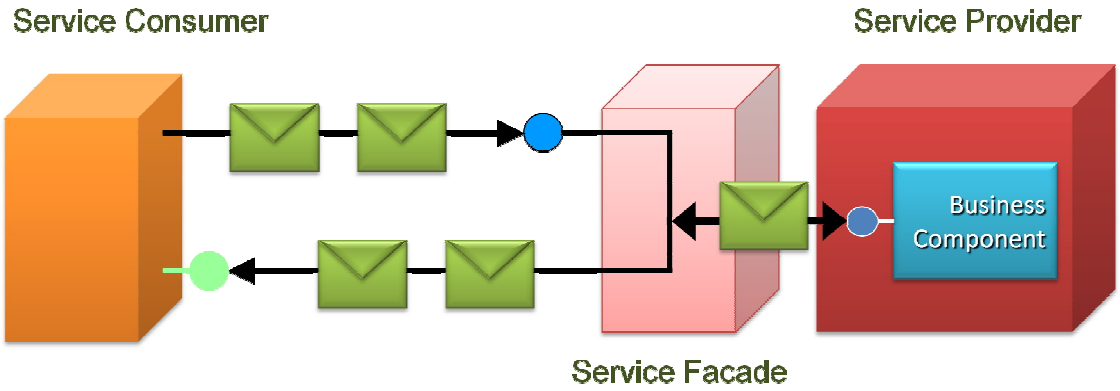


Figure 1. Services abstract model

Web services are software systems that have taken the concept of services delivered over the web to support interoperable machine-to-machine interaction over a network [2]. This interoperability is achieved through a set of XML-based open standards and technologies such as XML, Web Services

Description Language (WSDL), Simple Object Access Protocol (SOAP), and Universal Description, Discovery, and Integration (UDDI). SOA and web services are two different terms; web services are the preferred standard-based way to realize SOA. To define them in clear words, SOA is a paradigm for your design while web services are a practical implementation for the SOA architecture.

Basically, web services requires to follow a software architectural style to be realized, because there's no smart human being on the client end to keep track of every aspect of the service design(initialization, communication protocols, error and exception handling,...etc.). In software engineering, the term software architectural style is defined as "a set of design rules that identify the kinds of components and connectors that may be used to compose a system or subsystem".

This tutorial uses the standard and portable JAX-RS API to simplify the development of RESTful Web services and their clients in Java. We have also covered the development of REST clients for the Android-based platforms.

2. Basics of RESTful Web services

REST stands for **Re**presentational **S**tate **T**ransfer. REST is a technology relies on a *stateless*, *client-server*, *cacheable* communications technology that uses the HTTP protocol [4]. It was first introduced in 2000 by Roy Fielding at the University of California.

REST is considered as *an architectural style* for developing applications that communicate over the network. It defines a set of architectural principles by which you can design web services that focus on the concept of system's resources, including how resource states are addressed and transferred over the HTTP protocol by various client applications written in different programming languages.

In REST, the web services are viewed as resources and can be identified by their URLs (sometimes called URIs). Web service clients that want to use these resources and access a particular representation will need to use a globally defined set of remote methods that describe the action to be performed on the resource [3].

Due to its importance, we need to emphasize on the concepts of resource. Generally, the resources are identified by logical URLs where state, functionality and data could be represented as resources.

They are similar to "methods" or "services" used in Remote Procedure Call (RPC) and SOAP Web Services, respectively. For example, if you want to access the price of a product you do not need to call a "getProductName" and then a "getProductPrice" RPC calls; rather, you can view the product data as a resource by just calling a GET request for this URI, and this resource should contain all the required information or URI links to it.

The revolution made by REST lies in its simplicity as the idea was that, rather than using complex mechanisms or protocols such as CORBA, RPC or SOAP[2] to connect between machines (clients and servers), simple HTTP protocol is used to make calls between machines.

RESTful applications use HTTP requests to post data (create and/or update), read data (like queries), and delete data. Thus, REST uses HTTP for all four CRUD (Create/Read/Update/Delete) operations, so it is lightweight and simple alternative to the complex mechanisms stated above.

REST has emerged in the last few years as a fully featured web service design methodology that is adapted by the major web service providers (e.g. Yahoo, Google, and Facebook).

Figure 2. REST the REST service model in which the client and server communications are done via simple HTTP request and response.

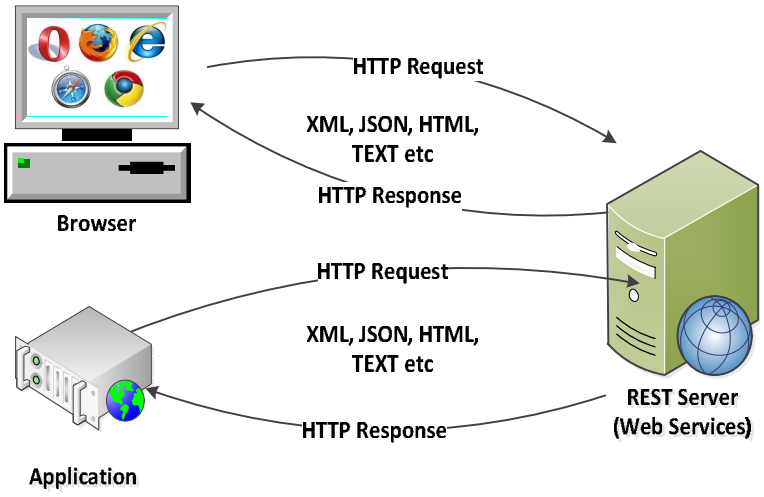


Figure 2. REST communication model

2.1. REST as Lightweight Web Services

Like traditional web services technologies, the REST-based service [3] is:

- Platform-independent (you don't care whether the server is Unix, the client is MacOS, or Windows),
- Language-independent (C# can talk to Java, etc.),
- Standards-based (runs on top of HTTP), and
- Easily used in the presence of firewalls, as it is simply using the popular HTTP protocol that can bypass any firewalls used.

Although REST doesn't offer any security features, encryption or QoS guarantees, these features can be added on top of the HTTP protocol. For instance, for security, username/password tokens are often sent with the REST service request for authorization purposes. For encryption, REST can be used on top of secure sockets links like HTTPS.

2.2. How simple is REST?

To better understand the concepts of REST let us consider a simple web service as an example: the following service is querying a database application for the details of a given user. All we have is the user's ID as input.

Using SOAP technology, the request (SOAP message) would look like the following:

```
<?xml version="1.0"?>
 <soap:Envelope
  xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
  soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
  <soap:body pb=" http://www.secc.org.eg/database">
   <pb:GetUserDetails>
    <pb:UserID>123</pb:UserID>
   </pb:GetUserDetails>
  </soap:Body>
 </soap:Envelope>
```

The entire SOAP message will be sent (using an HTTP POST request) to the server. The result could be an XML file, but it will be embedded, as the "payload", inside a SOAP response message. On the other hand, with REST; the query will simply look like this: http://www.secc.org.eg/database/UserDetails/123. This URL is sent to the server using a simpler GET request, and the HTTP reply will be the raw result data - not encapsulated in a header or envelope, just the data you need directly.

2.3. ROA vs. SOA and REST vs. SOAP

ROA (REST Oriented Architecture) is just a fancy name for a SOA (Service Based Architecture) that is based on REST services [1].

The main advantage of SOAP-based SOA over ROA is the more mature tool support and the security features that could be applied on the SOAP messages.
On the other hand, the main advantage of ROA is ease of implementation, agility of the design, and the lightweight nature. Whenever you need something up-and-running quickly with good performance and low overhead, it is often better to use REST and ROA. As REST matures, it is expected to become better understood and more popular even in more conservative industries.

2.4. Implementation of REST web services

The implementation of a REST web service must follow four basic design principles [4]:

- Use the same HTTP protocol methods.
- Stateless.
- Represent everything with URIs.
- Multiple representations support

2.4.1. Principle 1: Use the same HTTP protocol methods

One of the key characteristics of a RESTful Web service is the explicit use of HTTP methods in a way that follows the protocol as defined by RFC 2616. According to this mapping:

- To create a resource on the server, use POST.
- To retrieve a resource, use GET.
- To change the state of a resource or to update it, use PUT.
- To remove or delete a resource, use DELETE.

2.4.2. Principle 2: Stateless

REST web services are called very frequently by different clients, thus they need to scale to meet the increasingly high performance demands. The HTTP requests could be forwarded from one server to the other as needed to decrease the overall response time of a web service call. Using intermediary servers to improve the scaling require REST web service clients to send complete and independent requests with all data needed so that the components in the intermediary servers may forward, route, and load-balance the different requests without any state being held locally in between.

2.4.3. Principle 3: Represent everything with URIs

As the philosophy of REST is to expose everything with URIs, the structure of a URI should be straight forward, predictable, and easily understood.

One way to achieve this level of simplicity is to define directory-structure-like URIs. This type of URIs is hierarchical, started at a root path, and branching from it are sub-paths, e.g. http://www.secc.org.eg/root/topics/....

URIs should also be static which means that when the resource changes or the implementation of the service changes, the link stays the same. This allows bookmarking this URI for reuse.

2.4.4. Principle 4: Multiple representations support

The resources could be represented in different formats to give client applications the ability to request a specific content type that's best suited for them.

In the service side it is easy to make use of the built-in HTTP accept headers in order to determine the date types consumed or produced by the service. The value of the header is a MIME type that determines the required representation of the resource. Some common MIME types used by RESTful services are shown in Table 1.

MIME-Type	Content-Type
JSON	application/json
XML	application/xml
XHTML	application/xhtml+xml

Table 1. Common MIME types used by RESTful services

This allows for the service to be used by a variety of clients written in different languages and running on different platforms.

3. When to use REST?

Architects and developers need to decide when REST is an appropriate choice for their applications. Many experts in the REST services summarized the most common cases in which the RESTFul design is more appropriate:

1- The web services are completely stateless; to test this you can check whether the interaction can survive after a restart of the server.

2- The service producer and service consumer have a mutual understanding of the context and content being exchanged. On contrast to SOAP or WSDL-based services, there is no formal way to describe the web services

interface, so both producer and consumer must agree upon the schemas that describe the data being exchanged. In the real world, most commercial applications that expose services as RESTful implementations also provide so-called value-added toolkits that describe the REST services interfaces to developers in popular programming languages [3].

3- REST is particularly useful for limited-bandwidth devices such as PDAs and mobile phones as the overhead of headers and additional layers of SOAP elements on the XML payload will not be appropriate in this case [1].
4- Consuming the services in a pre-developed web applications will be much easier with the REST services.

The SOAP-based design may be much more appropriate when:

1- A formal contract must be established to describe the interface that the web service offers. The Web Services Description Language (WSDL) describes the details such as messages, operations, bindings, and location of the web service.
2- The application need to address some complex nonfunctional requirements like transactions, security, trust, and so on as most real-world applications go beyond simple CRUD operations. With the RESTful approach, developers must build these requirements into the application layer themselves [3].
3- The architecture needs to handle asynchronous processing and invocation [3].

4. RESTful Support in Java

The Java API for XML Web Services (JAX-WS) provides full support for building and deploying RESTful web services. The API was developed through the Java community process program as JSR 224. It is tightly integrated with the Java architecture for XML Binding (JAXB) for binding XML to Java technology data and is included in the Java Platform, Standard Edition (Java SE) and the Java Platform, Enterprise Edition (Java EE) [5].

Developing RESTful Web services that seamlessly support exposing your data in a variety of representation media types is not an easy task without a good toolkit. In order to simplify development of RESTful Web services and their clients in Java, a standard and portable JAX-RS API has been designed. Jersey RESTful Web services framework is open source, production quality, framework for

developing RESTful web Services in Java that provides support for JAX-RS APIs and serves as a JAX-RS (JSR 311 and JSR 339) Reference Implementation.

Jersey framework [5] is more than the JAX-RS Reference Implementation. Jersey provides its own API that extends the JAX-RS toolkit with additional features and utilities to further simplify RESTful service and client development. Jersey also exposes numerous extension SPIs so that developers may extend Jersey to best suit their needs.

Goals of Jersey project can be summarized in the following points:

- Track the JAX-RS API and provide regular releases of production quality reference implementations that ships with GlassFish.
- Provide APIs to extend Jersey and Build a community of users and developers.
- Make it easy to build RESTful Web services utilizing Java and the Java virtual machine.

For the development purpose, the following steps are needed:

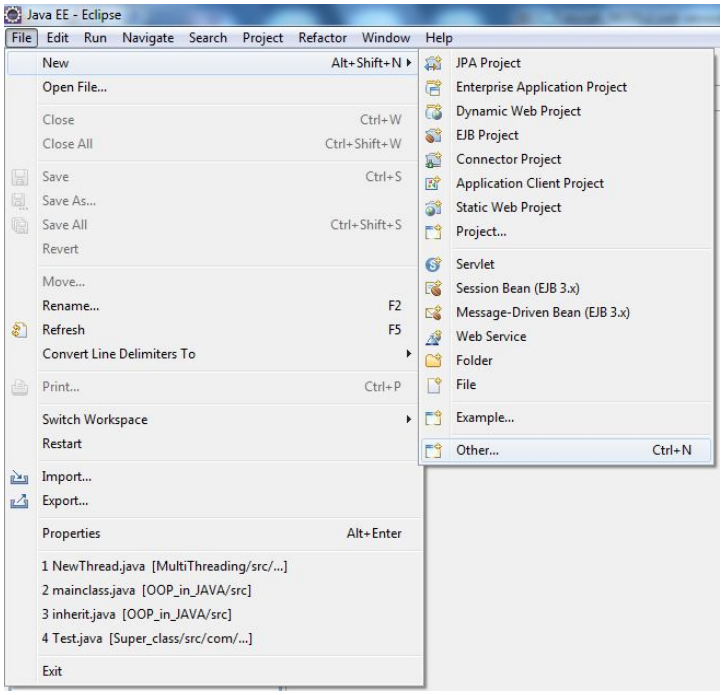
1. Install Java JDK SE Standard Edition 1.6.0_26 or above.
2. Install Eclipse version 3.4 or above (Eclipse Juno for JavaEE is preferred).
3. Download Jersey 1.17.1 ZIP bundle that contains the Jersey jars and core dependencies.
4. Get the Jackson Java JSON-processor libraries form http://jackson.codehaus.org/ (core library is the only needed one).
5. Download Apache Commons Codec from this link http://commons.apache.org/proper/commons-codec/index.html
   These codec provides implementations of common encoders and decoders such as Base64, Hex, Phonetic and URLs.
6. Copy all of the jar files downloaded above into WEB-INF/lib folder in your project (create the lib folder it if not existing).

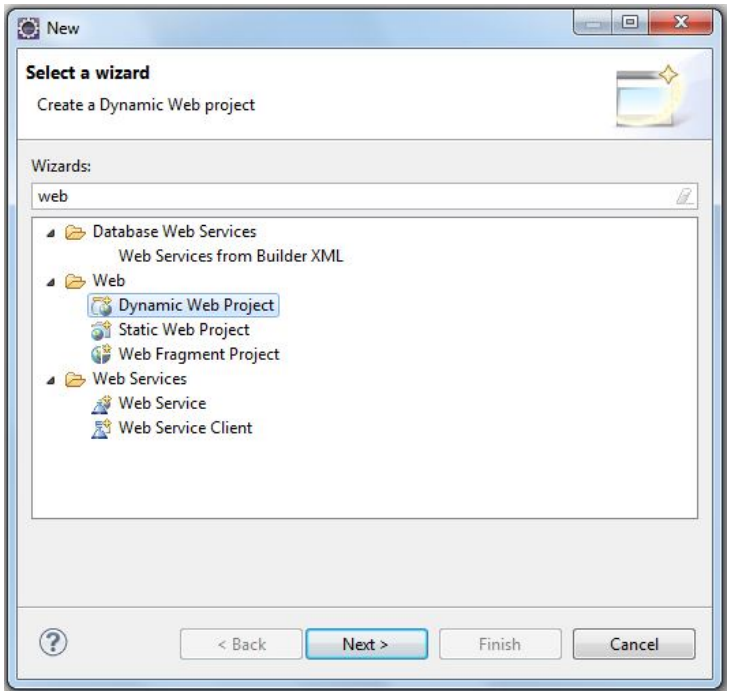
4.1. Hello REST Service Example

In the following part, steps for developing, deploying and creating the client for a very simple Hello service in a clear and systematic way will be explained.

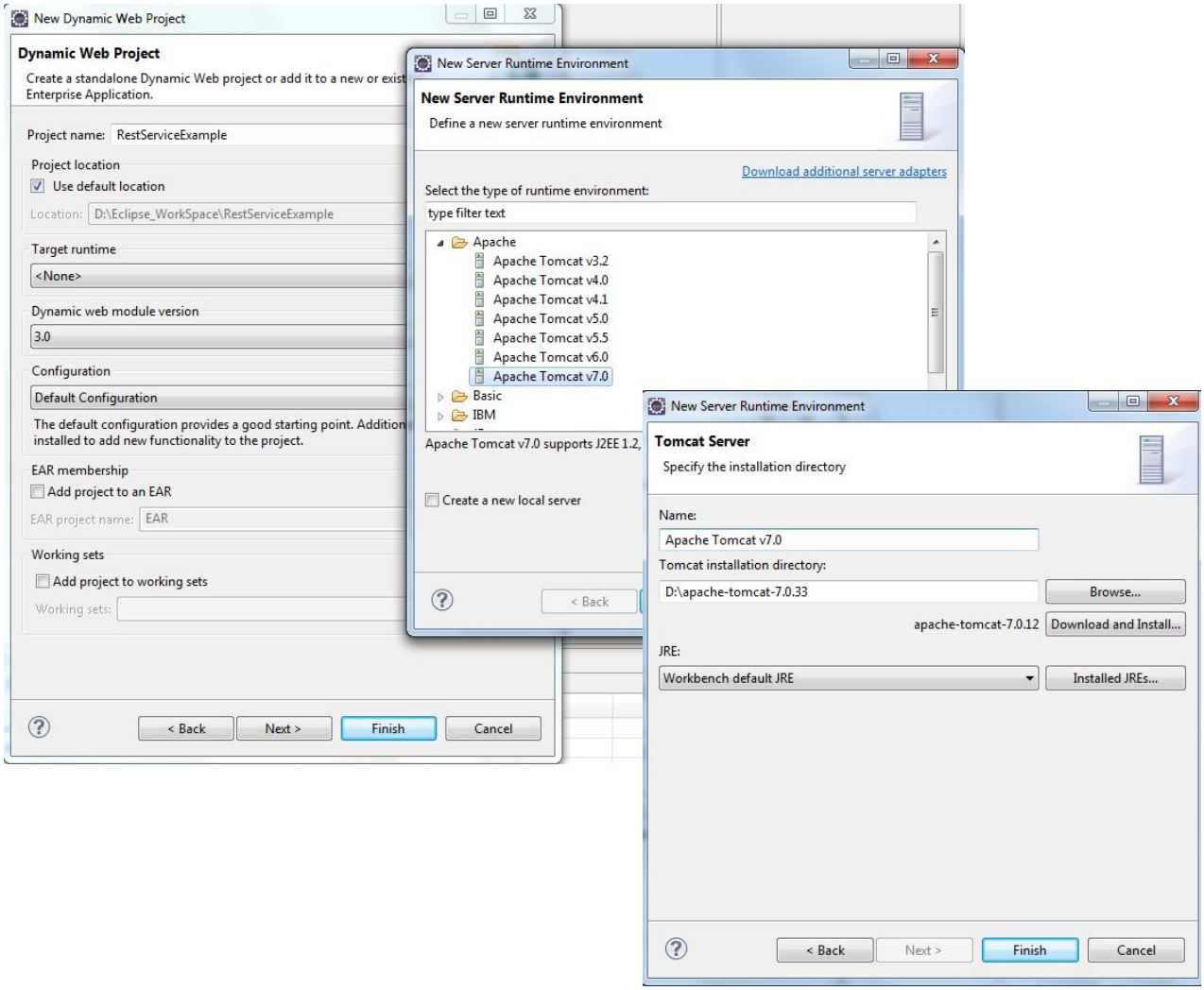
4.1.1. Service Implementation

1. From Eclipse file > new > other.
2. Select web > Dynamic web Project.
3. Click Next and set your project name “RestServiceExample”.





4. Select the target runtime environment (Apache Tomcat v7.0 is used).



5. After the above steps you'll have a new Web project created with the above name and the Apache tomcat server libraries will be included.
6. create a new package with the name "com.secc.test.rest.services" and add a java file "RESTServiceInterface.java" that includes the following:

    i. RESTServiceInterface java class that contains the service operations defined as java methods
    ii. Java annotations supported by the JAX-RS API including:
        a. ***@path***: used to specify the path concatenated to the rest service URL to access certain operations
        b. ***@GET, @POST***: annotations used for determining type of the service, either GET or POST
        c. ***@Produces***: used to specify the data type of service operation output (Text, JSON or any other format)
        d. ***@Consumes:*** used to specify the data type of service operation input (Text, JSON or any other format)
        e. ***@PathParam:*** used to specify operation inputs that could be passed from the URL

```
package com.secc.test.rest.services;

import javax.ws.rs.Consumes;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import org.codehaus.jettison.json.JSONObject;

@Path("/calc")
public class RESTServiceInterface {

	// class constructor;
	public RESTServiceInterface() {
		// TODO Auto-generated constructor stub
	}

	@GET
	@Produces(MediaType.TEXT_PLAIN)
	public String sayPlainTextHello() {
		return "Hello to the REST services world!";
	}

	@GET
	@Path("/add/{a}/{b}")
	@Produces(MediaType.TEXT_PLAIN)
	public String add(@PathParam("a") double a,@PathParam("b") double b) {
		//Start of user code add implementation
		//add your code here
		return "a + b = " + (a+b) ;
//End of user code
```

```
    }

    @GET
    @Path("/sub/{a}/{b}")
    @Produces(MediaType.TEXT_PLAIN)
    public String sub(@PathParam("a") double a,@PathParam("b") double b) {
          //Start of user code sub implementation
          //add your code here
          return "a - b = " + (a-b) ;
          //End of user code
    }

    @Path("/json")
    @POST
    @Consumes(MediaType.APPLICATION_JSON)
    public Response sayPlainTextHello(JSONObject inputJsonObj) throws Exception
    {

        String input = (String) inputJsonObj.get("username");
        String output = "The input you sent is :" + input;
        JSONObject outputJsonObj = new JSONObject();
        outputJsonObj.put("output", output);

        // To return a string saying success or failure

        String result = "!!!!! SUCCESS !!!!   "+output;
          return Response.status(201).entity(result).build();
      }
    }
```

Any dynamic web project needs a web.xml file in which we define some parameters that are needed for the REST service. Here's a sample of the web.xml file used for this simple Hello example.

```
<?xml version="1.0" encoding="UTF-8"?>
    <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
      id="WebApp_ID" version="2.5">
      <display-name>RestServicesExample</display-name>
      <servlet>
        <servlet-name>Jersey REST Service</servlet-name>
        <servlet-
    class>com.sun.jersey.spi.container.servlet.ServletContainer</servlet-class>
        <init-param>
          <param-name>com.sun.jersey.config.property.packages</param-name>
          <param-value>com.secc.test.rest.services</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
      </servlet>
      <servlet-mapping>
        <servlet-name>Jersey REST Service</servlet-name>
```

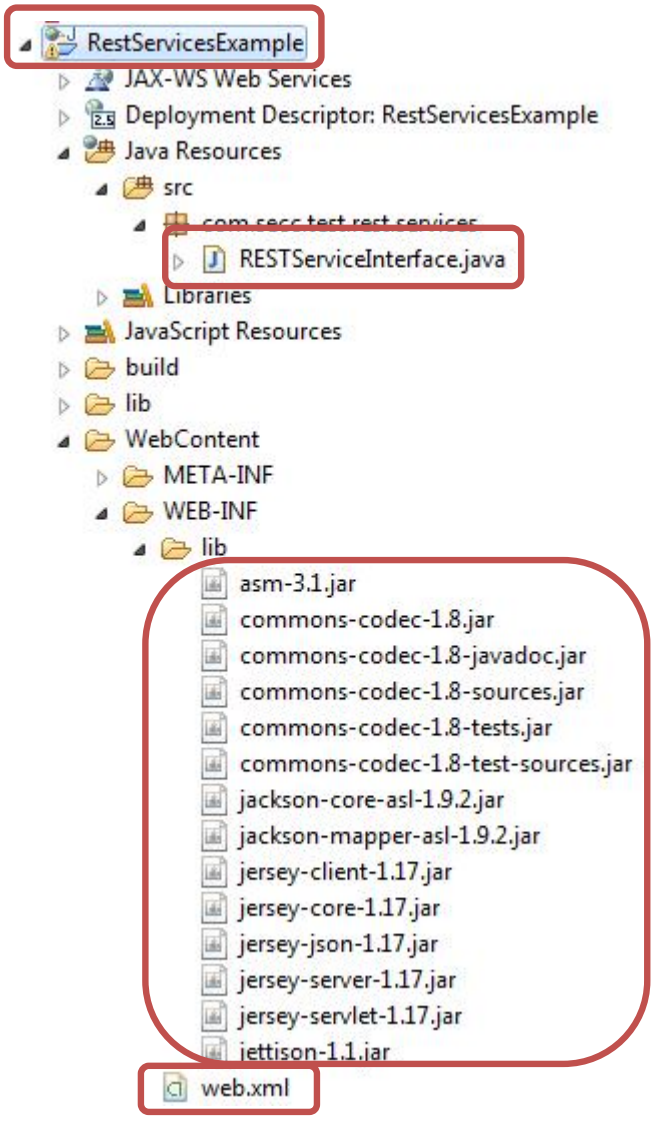
```
    <url-pattern>/test/*</url-pattern>
  </servlet-mapping>

</web-app>
```

Both display-name and url-pattern tags are concatenated to the service URL to form the common URL used to access the REST service. For example, if we want to access the above developed REST service, the common URL pattern will be:

http://localhost:8080/RestServicesExample/test/

display-name url-pattern

In addition, the param name tag in the web.xml file that has the value of *com.sun.jersey.config.property.packages* must point out to the package name in which the REST service is developed. The final project structure will be as shown:

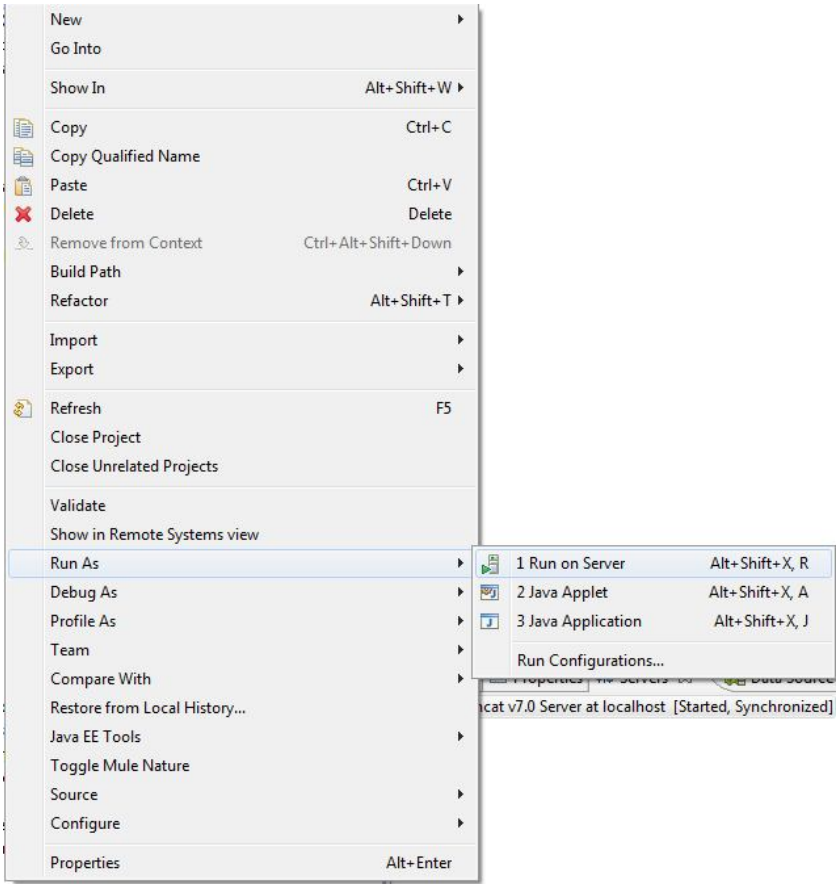


4.1.2. Deployment and running of services

One of the best feature of REST services is that they can be easily deployed in any application server like Tomcat and they can be called directly from web browser (IE, chrome, Firefox, ...etc.).

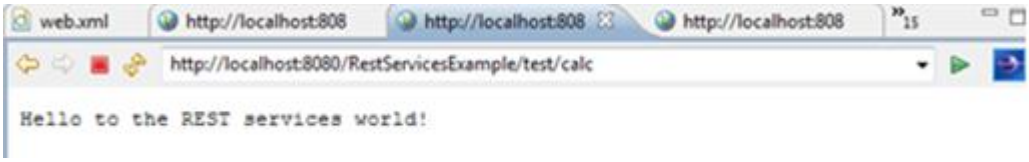
To deploy and run the REST service developed above you can do that by two ways:

1- ***From inside Eclipse:*** using its internal Tomcat application server (for development purposes) as you will right click the web project and select *Run As -> Run on server*. Eclipse will initiate the Tomcat server and opens an internal a web interface to run the service. In the address field, you can simply write the following line:



http://localhost:8080/RestServicesExample/test/calc

You can see the output as follow:



You can also put the same URL in a web browser as shown below:

2- ***Building a war file*** to be deployed in a standalone Tomcat application server (in the practical case)

5. Building Android client for REST services

REST services are well suited for providing content to the limited resources devices like smart phones and tablets. In fact, if you've interacted with any cloud-based APIs recently, you will find that most of them are exposed as REST services.

Figure 3. REST services for Android

There are a lot of methods for developing REST services client for the Android platform. In this tutorial we are focusing on using the asynchronous task concepts in Android to handle the calling process for the REST services. The asynchronous task enables proper and easy use of the UI thread as it allows performing background operations and publishing results on the UI thread without having to manipulate threads and/or handlers [7].

5.1. Development steps for Android REST client

In this section we will explain in a clear and easy way how to build an Android REST client. This was a part of the SALE advertising platform developed by SECC R&D team as one of the RECOCAPE[1] project activities.

[1] http://www.secc.org.eg/Recocape

1- Create an Empty Android project: In this tutorial we are using the ADT bundle for windows as the Android development platform http://developer.android.com/sdk/index.html
2- Create a new package with the name of *com.secc.sale_rest_client* and copy the following code in *RestWebServiceJavaClient.java* file

```
package com.secc.sale_rest_client;

import java.io.IOException;
import java.io.InputStream;
import java.io.UnsupportedEncodingException;
import java.util.concurrent.ExecutionException;

import org.apache.http.HttpEntity;
import org.apache.http.HttpResponse;
import org.apache.http.client.HttpClient;
import org.apache.http.client.methods.HttpGet;
import org.apache.http.client.methods.HttpPost;
import org.apache.http.entity.StringEntity;
import org.apache.http.impl.client.DefaultHttpClient;
import org.apache.http.protocol.BasicHttpContext;
import org.apache.http.protocol.HttpContext;
import org.json.JSONException;
import org.json.JSONObject;
import org.json.JSONTokener;
import com.sale.advertisementclient.MainActivity;

import android.app.Activity;
import android.content.Context;
import android.os.AsyncTask;
import android.os.Bundle;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.EditText;


public class RestWebServiceJavaClient {

       private String servicePath;
       private JSONObject json;
       private boolean secure = true;
    private static Context appContext = null;




public RestWebServiceJavaClient(String servicePath) {
             super();
             this.servicePath = servicePath;
             RestWebServiceJavaClient.appContext = ((MainActivity)
MainActivity.MainActivityInstance).getApplicationContext();
```

```
}
public RestWebServiceJavaClient(String servicePath,JSONObject json) {
	super();
	this.servicePath = servicePath;
	this.json = json;
RestWebServiceJavaClient.appContext = ((MainActivity)
MainActivity.MainActivityInstance).getApplicationContext();

	}

	public RestWebServiceJavaClient(String servicePath, Context context) {
		super();
		this.servicePath = servicePath;
		RestWebServiceJavaClient.appContext = context;

	}

	public RestWebServiceJavaClient(String servicePath,JSONObject json, Context
context) {
		super();
		this.servicePath = servicePath;
		this.json = json;
		RestWebServiceJavaClient.appContext = context;
	}


	public class WebServiceGetText extends AsyncTask <Void, Void, String> {

		protected String getASCIIContentFromEntity(HttpEntity entity) throws
IllegalStateException, IOException {
		InputStream in = entity.getContent();
		StringBuffer out = new StringBuffer();
		int n = 1;
		while (n>0) {
		byte[] b = new byte[4096];
		n =  in.read(b);
		if (n>0) out.append(new String(b, 0, n));
		}
		return out.toString();
		}
		@Override
		protected String doInBackground(Void... params) {

		HttpClient httpClient;

		httpClient = new DefaultHttpClient();

		HttpContext localContext = new BasicHttpContext();
		HttpGet httpGet = new HttpGet(servicePath);

		httpGet.addHeader("accept", "text/plain");
		String text = null;
		try {
		HttpResponse response = httpClient.execute(httpGet, localContext);
		HttpEntity entity = response.getEntity();

		text = getASCIIContentFromEntity(entity);
```

```
	} catch (Exception e) {
				System.out.println(e.getLocalizedMessage());
			}
			System.out.println("!!!!!!!!"+text+"!!!!!!");
		return text;

		}
		protected void onPostExecute(String results) {
		if (results!=null) {
			System.out.println("GETTEXT:: SUCCESS"+results);
		}
	}
}

	public class webServicePostReturnJSON extends AsyncTask <Void, Void, JSONObject>
	{
		protected String getASCIIContentFromEntity(HttpEntity entity) throws
IllegalStateException, IOException {
		InputStream in = entity.getContent();
		StringBuffer out = new StringBuffer();
		int n = 1;
		while (n>0) {
		byte[] b = new byte[4096];
		n =  in.read(b);
		if (n>0) out.append(new String(b, 0, n));
		}
		return out.toString();
		}
		@Override
		protected JSONObject doInBackground(Void... params) {

		HttpClient httpClient;

		httpClient = new DefaultHttpClient();

		HttpContext localContext = new BasicHttpContext();

	HttpPost request = new HttpPost(servicePath);
	StringEntity s = null;
		try {
			s = new StringEntity(json.toString());
		} catch (UnsupportedEncodingException e1) {
			// TODO Auto-generated catch block
			e1.printStackTrace();
		}
	s.setContentEncoding("UTF-8");
	s.setContentType("application/json");

	request.setEntity(s);
	request.addHeader("accept", "application/json");
		String text = null;
		String query = null;
		JSONObject object = null;
		try {
		HttpResponse response = httpClient.execute(request, localContext);
		HttpEntity entity = response.getEntity();
		// if you want to return a string
```

```
        text = getASCIIContentFromEntity(entity);
        // for JSON objects returned form GET methods:
         object = (JSONObject) new JSONTokener(text).nextValue();
 } catch (Exception e) {
              System.out.println(e.getLocalizedMessage());
        }
        return object;
        }
        protected void onPostExecute(String results) {
        if (results!=null) {
              System.out.println("POST:: @@@@@@@@"+results+"@@@@@@@");
        }
 }
}
```

In the above code the RestWebServiceJavaClient class includes some constructors and methods to make it easier for calling the REST services as follow:

1- To call the GET service (sayPlainTextHello), you will just need to write the following code:

```
RestWebServiceJavaClient sayHello=new
 RestWebServiceJavaClient("http://localhost:8080/RestServicesExample/test/calc");
WebServiceGetText w1= sayHello.new WebServiceGetText ();
              AsyncTask<Void, Void, String> l1=w1.execute();

              try {
                    String str=l1.get();
                    // str variable that contains the
                    if(str.contains("error")){
                    }
              } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
              } catch (ExecutionException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
              }
```

We defined an asynchronous task in Android and started it execution. The resulted output can be easily displayed in a text area or in a toast

2- To call the GET service (*add*), you will just need to write the following code:

```
RestWebServiceJavaClient sayHello=new
RestWebServiceJavaClient("http://localhost:8080/RestServicesExample/test/calc/add
/3/5");
WebServiceGetText w1= sayHello.new WebServiceGetText ();
```

```
      AsyncTask<Void, Void, String> l1=w1.execute();

            try {
                  String str=l1.get();
                  // str variable that contains the
                  if(str.contains("error")){
                  }
            } catch (InterruptedException e) {
                  // TODO Auto-generated catch block
                  e.printStackTrace();
            } catch (ExecutionException e) {
                  // TODO Auto-generated catch block
                  e.printStackTrace();
}
```

You can do the same for *sub* service

3- To call the POST service (sayPlainTextHello), you will just need to write the following code:

```
JSONObject input = new JSONObject();
try {
      input.put("username", “REST Example”);

} catch (JSONException e) {
      // TODO Auto-generated catch block
      e.printStackTrace();
}

RestWebServiceJavaClient sayPlainTextHello=new
RestWebServiceJavaClient(http://localhost:8080/RestServicesExample/test/calc/json
/sayPlainTextHello,input);
webServicePostJSONReturnTEXT w4= sayPlainTextHello.new
webServicePostJSONReturnTEXT();
AsyncTask<Void, Void, String> l4=w4.execute();

      try {
            String str=l4.get();
            System.out.println("!!!!!! Test plain text post service :: "+ str);
      }

      catch (InterruptedException e) {
                  // TODO Auto-generated catch block
                  e.printStackTrace();
            } catch (ExecutionException e) {
                  // TODO Auto-generated catch block
                  e.printStackTrace(); }
      }
```

This code will simply create a JSON object with name”*username*“and value “*REST Example*”. This JSON object is passed to the POST service *sayPlainTextHello* , The asynchronous task will execute the service and get the following JSON object back {“output”,” The input you sent is :”}

6. Summary

REST is a return to the Web, through its emphasis on the early Internet standards, URI and HTTP. Exposing the resources of the system through a RESTful API is a flexible way to provide different kinds of applications with data formatted in a standard way. Using such technique will help to meet integration requirements that are critical to building systems where data can be easily combined. In this tutorial, we have covered most aspects of REST services including the design and implementation of both services and clients using JAX-RS technology. Android REST client example was demonstrated using a sample source code and detailed development steps.

7. References

[1] Service Oriented Architecture For Dummies, 2nd IBM Limited Edition

[2] Abdallah M., Mahjoub W., " A Quick Introduction to SOA", http://www.secc.org.eg/Recocape/Publications.html

[3] Leonard Richardson, Sam Ruby, "Restful web services book", O'Reilly Media, May 2007

[4] http://www.ibm.com/developerworks/webservices/library/ws-restful/

[5] http://jersey.java.net/

[6] http://maven.apache.org/

[7] http://developer.android.com/reference/android/os/AsyncTask.html

8. Abbreviations

SOA	Service Oriented Architecture
REST	Representational state transfer
HTTP	Hypertext Transfer Protocol
JMS	Java Message Service
URL	Unique Resource Location
URI	Unique Resource Identifier
QoS	Quality of Service
XML	eXtensible Markup Language
WSDL	Web Services Description Language
RPC	Remote Procedure Call