



Lesson 16

Consuming Web Services Using SOAP and REST Apps

Victor Matos

Cleveland State University

Portions of this page are reproduced from work created and [shared by Google](#) and used according to terms described in the [Creative Commons 3.0 Attribution License](#).

Android & WebServices

Overview

- A **WebService** is a **Consumer_Machine-to-Provider_Machine** collaboration schema that operates over a computer network.
- The data exchanges occur independently of the OS, browser, platform, and programming languages used by the provider and the consumers.
- A provider may expose multiple **EndPoints** (sets of WebServices), each offering any number of typically related functions.
- WebServices expect the computer network to support standard Web protocols such as XML, HTTP, HTTPS, FTP, and SMTP.
- **Example:** Weather information, money exchange rates, world news, stock market quotation are examples of applications that can be modeled around the notion of a remote data-services provider surrounded by countless consumers tapping on the server's resources.

Android & WebServices

Advantages of Using the WebService Architecture

- Under the **WebService** strategy the invoked functions are *implemented once* (in the server) and *called many times* (by the remote users).
- Some advantages of this organization are:
 - Elimination of redundant code,
 - Ability to transparently make changes on the server to update a particular service function without clients having to be informed.
 - Reduced maintenance and production costs.

Android & WebServices

Why should the Android developer learn how to create a Webservice?

- Simple apps are usually self-contained and do not need to collaborate with other parties to obtain additional data or services (for instance, think of a scientific calculator)
- However, there are many cases in which the data needed to work with is very extensive, or changes very often and cannot (should not) be hard-coded into the app. Instead, this kind of data should be requested from a reliable external source (for instance, what is the Euro-to-Dollar rate of change right now?)
- Another class of apps requires a very high computing power perhaps not available in the mobile device (think of problems such as finding the shortest/fastest route between two mapped locations, or best air-fare & route selection for a traveler)
- It is wise for an Android developer to learn how to solve typical problems that exceed the capacities of the handheld devices. Understanding the possibilities offered by the client-server computing model will make the developer be a more complete and better professional.

Android & WebServices

WebService Architecture

An ideal *Webservice* provider is designed around four logical layers which define the ways in which data is to be transported, encoded, exposed and discovered by the users.

Layers	Responsibility
Transport	Move messages through the network, using HTTP, SMTP, FTP, ...
Messaging	Encoding of data to be exchanged (XML)
Description	WSDL (Web Service Desc. Lang) used for describing public methods available from the endpoint
Discovery	UDDI (Universal Description & Discovery Integration) facilitates location and publishing of services through a common registry

Android & WebServices

The Client Side - Consuming WebServices

There are two widely used forms of invoking and consuming WebServices:

Representational State Transfer (REST)

Closely tie to the HTTP protocol by associating its operation to the common methods: **GET, POST, PUT, DELETE** for HTTP/HTTPS.

This model has a simple invocation mode and little overhead. Service calls rely on a URL which may also carry arguments. Sender & receiver must have an understanding of how they pass data items from one another. As an example: Google Maps API uses the REST model.

Remote Procedure Call (RPC).

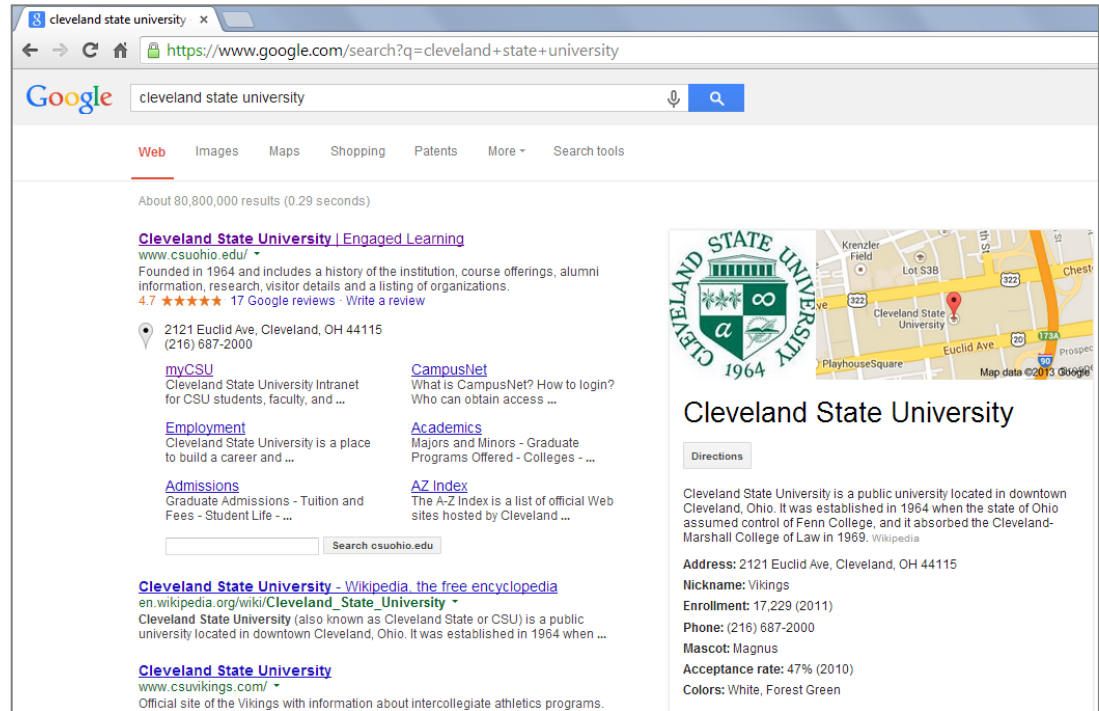
Remote services are seen as coherent collections of discoverable functions (or method calls) stored and exposed by EndPoint providers. Some implementations of this category include: Simple Object Access Protocol (SOAP), Common Object Request Broker Architecture (CORBA), Microsoft's Distributed Component Object Model (DCOM) and Sun Microsystems's Java/Remote Method Invocation (RMI).

Android & WebServices

Consuming WebServices

Example: Using REST.

The following URL is used to make a call to the **Google Search** service asking to provide links to the subject “Cleveland State University”



The screenshot shows a Google search result for 'Cleveland State University'. The search bar contains the text 'cleveland state university'. The results page displays a list of links, including 'Cleveland State University | Engaged Learning', 'myCSU', 'Employment', 'Admissions', 'CampusNet', 'Academics', and 'AZ Index'. On the right side, there is a map of Cleveland State University and a detailed information box for the university, including its address (2121 Euclid Ave, Cleveland, OH 44115), nickname (Vikings), enrollment (17,229 in 2011), phone number, mascot (Magnus), acceptance rate (47% in 2010), and colors (White, Forest Green).

Transport

Provider

Action

Arguments

<https://www.google.com/search?q=cleveland+state+university>

Figure 1. Example of a REST web-service called with a URL that includes arguments

Android & WebServices

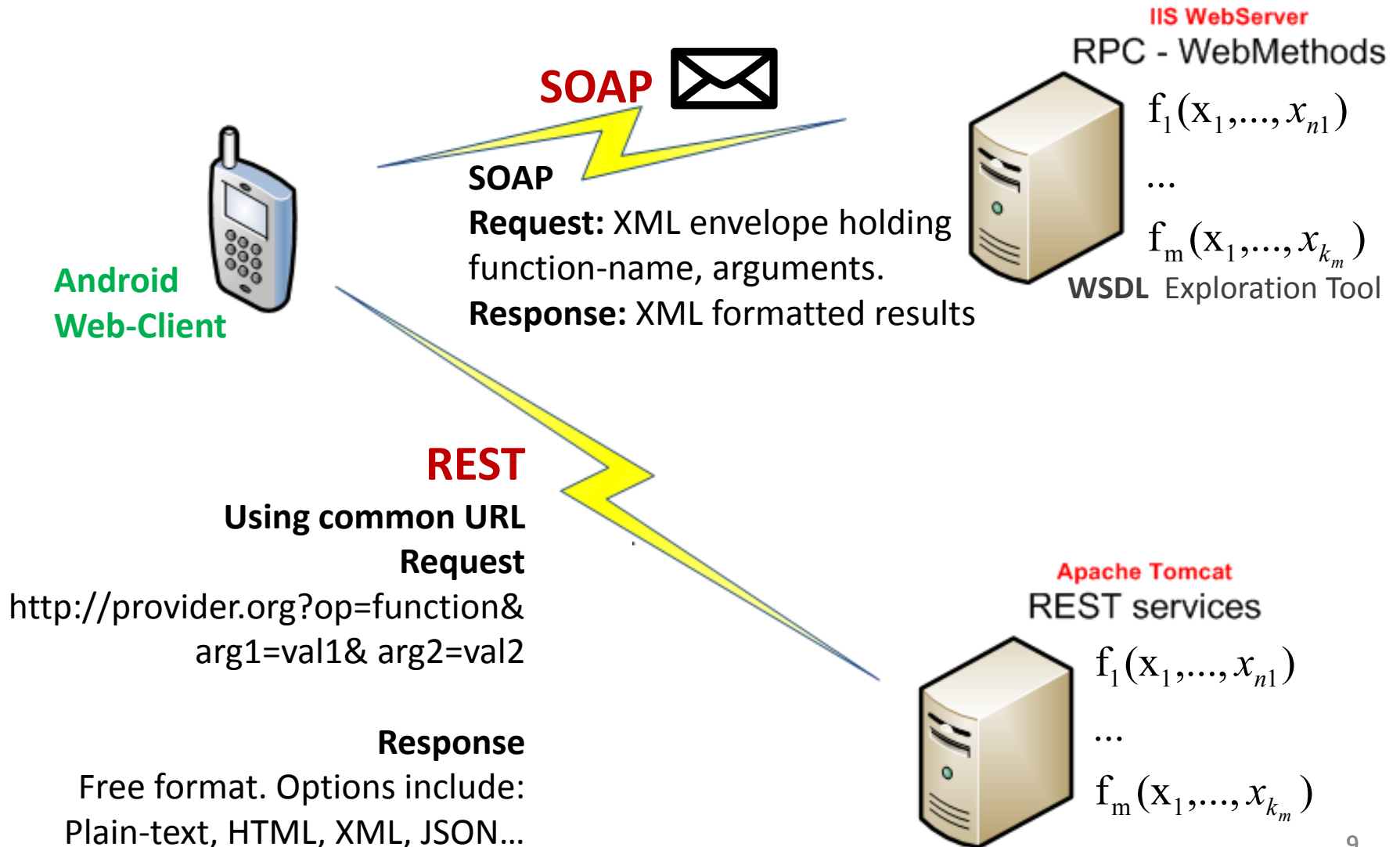
REST vs. SOAP

Although SOAP and REST technologies accomplish the same final goal, that is request and receive a service, there are various differences between them.

- **REST** users refer to their remote services through a conventional **URL** that commonly includes the location of the (stateless) server, the service name, the function to be executed and the parameters needed by the function to operate (if any). Data is transported using HTTP/HTTPS.
- **SOAP** requires some scripting effort to create an XML envelop in which data travels. An additional overhead on the receiving end is needed to extract data from the XML envelope. SOAP accepts a variety of transport mechanisms, among them HTTP, HTTPS, FTP, SMTP, etc.
- SOAP uses **WSDL** (WebService Description Language) for exposing the format and operation of the services. REST lacks an equivalent exploratory tool.

Android & WebServices

Figure 2. A WebClient consuming services using REST & SOAP



Android & WebServices

Examples of Android Apps Using REST and SOAP

In the next sections we will present three examples showing how an Android web-client typically interacts with a remote server requesting and consuming WebServices.

Example 1. SOAP client / .NET provider

An Android app uses a XML KSOAP envelope to call a Windows IIS server. WebServices are implemented as a set of C#.NET functions.

Example 2. REST client / PHP provider

A REST Android client invokes remote PHP services which consult a database on behalf of the client. The response is formatted using JSON.

Example 3. REST client / Servlet provider

Our Android app communicates with an Tomcat Server in which its WebServices are implemented as Java Servlets. As in the previous example, the results of a database query are returned as a JSON string.

Android & WebServices

Windows Communication Foundation (WCF)

BACKGROUND

WCF is a Microsoft technology that provides a framework for writing code to communicate across heterogeneous platforms [1, 2].

1. An IIS WebServer may host various **EndPoints** (WebServices).
2. Each of those EndPoints uses **WSDL** to provide a way of exposing its composition and behavior to clients wishing to find and communicate with the services.
3. Each endpoint includes:
 - **address** (URL - where to send messages),
 - **binding** (how to send messages), and a
 - **contract** (an explanation of what messages contain)

References:

[1] <http://msdn.microsoft.com/en-us/magazine/cc163647.aspx>

[2] [http://msdn.microsoft.com/en-us/library/ms734712\(v=vs.110\).aspx](http://msdn.microsoft.com/en-us/library/ms734712(v=vs.110).aspx)

Android & WebServices

WSDL Service Contracts

Example: The link <http://www.webserviceX.net/uszip.asmx?WSDL> takes us to a WCF EndPoint useful for finding locations in the US based on zip code (only a few lines are shown). This view –written in **WSDL**– is known as the **service contract**.

```
<wsdl:definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" ...
  targetNamespace="http://www.webserviceX.NET">
  <wsdl:types>
  <s:schema elementFormDefault="qualified"
    targetNamespace="http://www.webserviceX.NET">
    <s:element name="GetInfoByZIP"> ← Method's name, Argument, Type
    <s:complexType>
    <s:sequence>
    <s:element minOccurs="0" maxOccurs="1" name="USZip" type="s:string"/>
    </s:sequence>
    </s:complexType>
    </s:element>
    <s:element name="GetInfoByZIPResponse"> ← Returned result
    <s:complexType>
    <s:sequence>
    <s:element minOccurs="0" maxOccurs="1" name="GetInfoByZIPResult">
    <s:complexType mixed="true">
    <s:sequence>
    . . .
```

Android & WebServices

WSDL Service Contracts

Remove the fragment **?WSDL** from the previous link. The shorter URL <http://www.websvc.net/uszip.aspx> exposes the endpoint service functions as shown in the figure below

The screenshot shows a web browser window with the address bar containing the URL `www.websvc.net/uszip.aspx`, which is highlighted with a red rectangular box. Below the address bar, there are several application icons including 'Apps', 'CSU', 'CIS-Matos', 'CSU-Admin#', 'CSU email', 'Amazon', 'Mail', 'Netflix', and 'El Impulso'. The main content area has a dark blue header with the text 'USZip' in white. Below the header, there is a paragraph of text: 'The following operations are supported. For a formal definition, please review the [Service Description](#).' This is followed by a bulleted list of four operations, each with a blue underlined link and a brief description:

- [GetInfoByAreaCode](#)
Get State Code, City, Area Code, Time Zone, Zip Code by Area Code
- [GetInfoByCity](#)
Get State Code, City, Area Code, Time Zone, Zip Code by City
- [GetInfoByState](#)
Get State Code, City, Area Code, Time Zone, Zip Code by state
- [GetInfoByZIP](#)
Get State Code, City, Area Code, Time Zone, Zip Code by Zip Code

A red square with a white arrow points to the 'GetInfoByZIP' operation.

Figure 3. A .NET WebService providing USA ZIP-code information

Figure 4.
**WSDL Service Contracts &
SOAP Envelopes**

← → ↻ ↑

USZip

Click [here](#) for a complete list of operations.

GetInfoByZIP

Get State Code, City, Area Code, Time Zone, Zip Code by Zip Code

Test

To test the operation using the HTTP POST protocol, click the 'Invoke' button.

Parameter	Value
USZip:	<input type="text" value="44114"/>

SOAP 1.1

The following is a sample SOAP 1.1 request and response. The [placeholders](#) shown need to be replaced with the actual values.

```
POST /uszip.asmx HTTP/1.1
Host: www.webserviceX.net
Content-Type: text/xml; charset=utf-8
Content-Length: length
SOAPAction: "http://www.webserviceX.NET/GetInfoByZIP"

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <GetInfoByZIP xmlns="http://www.webserviceX.NET">
      <USZip>string</USZip>
    </GetInfoByZIP>
  </soap:Body>
</soap:Envelope>
```

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <GetInfoByZIPResponse xmlns="http://www.webserviceX.NET">
      <GetInfoByZIPResult>xml</GetInfoByZIPResult>
    </GetInfoByZIPResponse>
  </soap:Body>
</soap:Envelope>
```

Outgoing Envelop
(Request)

Incoming Envelop
(Response)

Android & WebServices

WSDL Service Contracts

Figure 5.

The **Response** data is sent by the WebService to the client as an **XML** encoded string.



Android & WebServices

Example1: Android SOAP Web-Client

This example consists of two parts. First we construct a server-side **EndPoint** offering a set of **Windows IIS web-services**, in the second part we build an Android SOAP client that requests and consumes the previous web-services.

Server Side Software:

- The documents:
(VS2015) [https://msdn.microsoft.com/en-us/library/8wbhsy70\(v=vs.90\).aspx](https://msdn.microsoft.com/en-us/library/8wbhsy70(v=vs.90).aspx)
(VS2010) <http://support.microsoft.com/kb/308359> provide a step-by-step tutorial describing how to create a web-service running on a Windows IIS-Server. We will create three methods: addValues, getPersonList, and rejuvenatePerson.

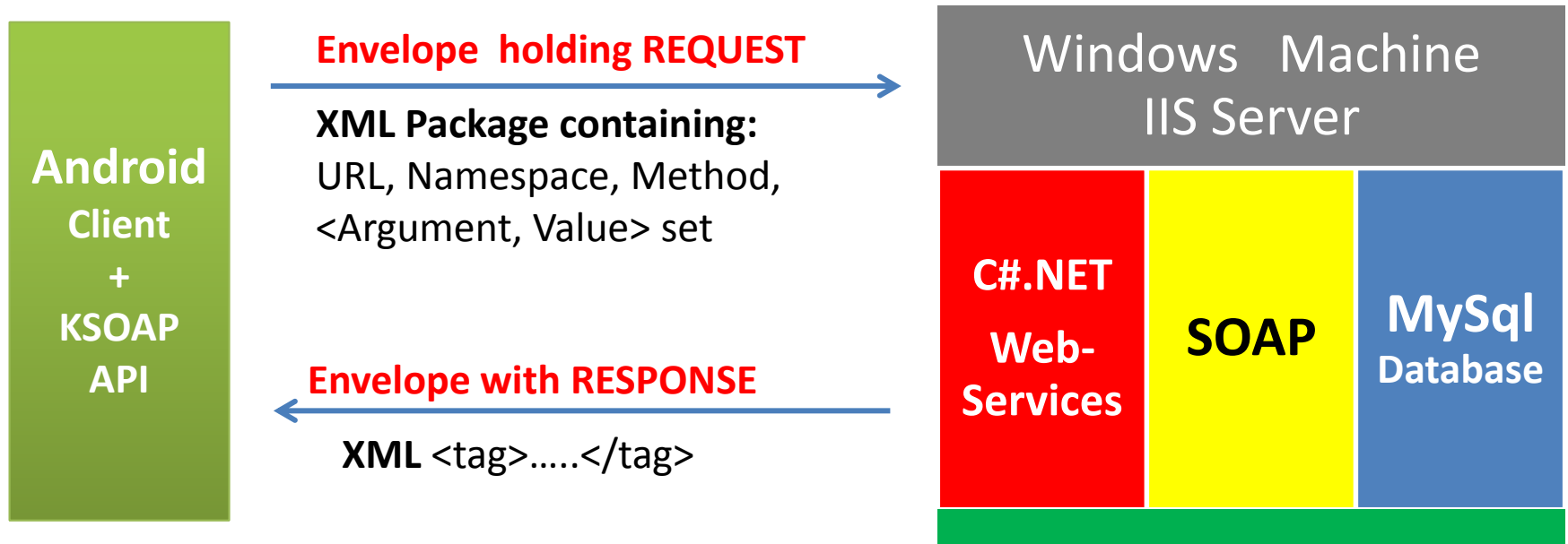
Client Side:

- **KSOAP2** ^[1] facilitates sending requests and receiving results to/from an IIS server.
- KSOAP2 includes various access methods such as: .getProperty(...), .getPropertyAsString(...), .getPropertyCount() to dissect the data tokens returned to the Android app inside the composite **response** object.

[1] KSOAP download link: <http://code.google.com/p/ksoap2-android/>
<http://simpligility.github.io/ksoap2-android/index.html>

Android & WebServices

Example1: Android SOAP Web-Client & .NET WebServices



Android & WebServices

Example1: KSOAP API

BACKGROUND

Android does not supply a native mechanism to handle SOAP exchanges. Consequently we will use an external library such as the **KSOAP2 API**.

- **KSOAP** is designed for limited hardware devices.
- **KSOAP** can exchange simple Java types (**SoapPrimitive**), as well as serialized complex objects (**SoapObject**).

KSOAP API

SoapEnvelope

Holds the encoded object's head and body. Includes a *parse()* method to pull the supplied XML data.

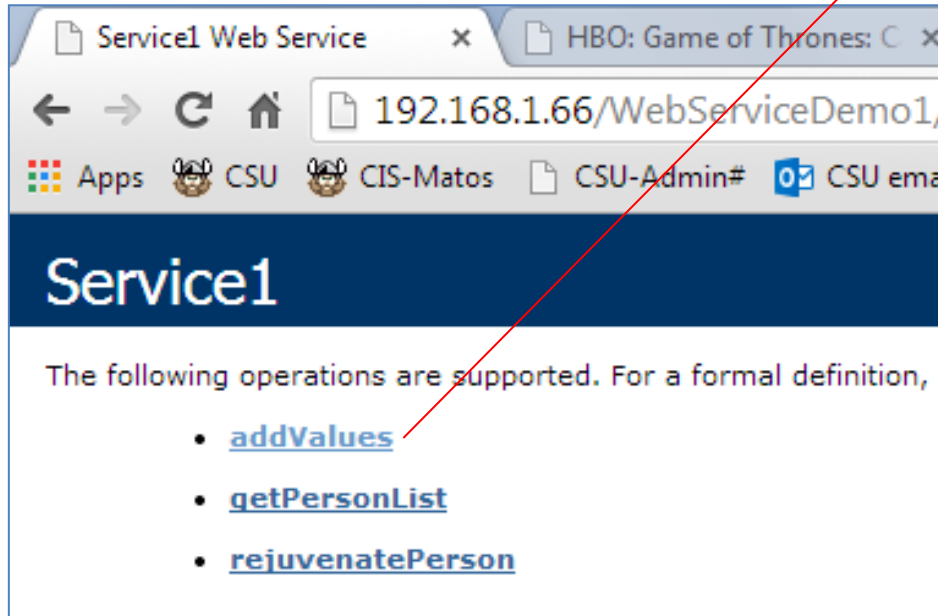
SoapObject

Generic container that travels inside the XML envelope transporting app's data. Custom objects must implement the *KvmSerializable* interface.

HttpTransport

This method uses an URL to place an HTTP call and allow SOAP exchange using the JME generic connection framework.

Example1: Android SOAP Web-Client Consuming .NET WebServices



This Windows IIS EndPoint offers three remote methods. Click on any entry to see contract details regarding its requests & responses

Service1

Click [here](#) for a complete list of operations.

addValues

Test

To test the operation using the HTTP POST protocol, click the 'Invoke' button.

Parameter	Value
var1:	<input type="text" value="111"/>
var2:	<input type="text" value="222"/>

SOAP 1.1

The following is a sample SOAP 1.1 request and response. The placeholders are for the actual request and response.

```
POST /WebServiceDemo1/Service1.asmx HTTP/1.1
Host: 192.168.1.66
Content-Type: text/xml; charset=utf-8
Content-Length: length
SOAPAction: "http://MyHostNameOrIPAddress/addValues"

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <addValues xmlns="http://MyHostNameOrIPAddress/">
      <var1>int</var1>
      <var2>float</var2>
    </addValues>
  </soap:Body>
</soap:Envelope>
```

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <addValuesResponse xmlns="http://MyHostNameOrIPAddress/">
      <addValuesResult>double</addValuesResult>
    </addValuesResponse>
  </soap:Body>
</soap:Envelope>
```



Android & WebServices

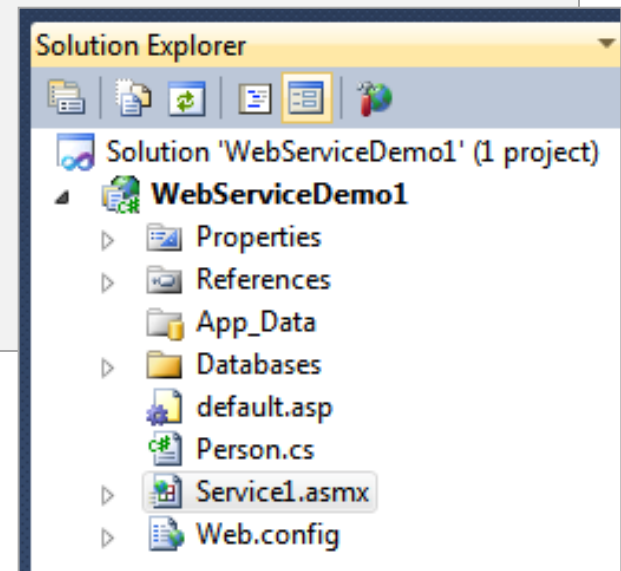
Example1: A Sample of C#.NET Webservices

Page 1 of 3

```
namespace WebServiceDemo1
{
    1 → [WebService(Namespace = "http://MyHostNameOrIPAddress/")]
        [WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
        [System.ComponentModel.ToolboxItem(false)]
        [System.Web.Script.Services.ScriptService]

    public class Service1 : System.Web.Services.WebService
    {
        // add two numbers -----
        → [WebMethod]
        public double addValues(int var1, float var2)
        {
            return (var1 + var2);
        }
    }
}
```

The c#.NET methods acting as webservices are marked with the **[WebMethod]** annotation. Public WebMethods are added to the end-point's WSDL contract, so their input/output structures can be exposed.



Android & WebServices

Example1: A Sample of C#.NET Webservices

Page 2 of 3

```
2 → // Return a list of Person objects -----  
[WebMethod]  
public List<Person> getPersonList(String home)  
{  
    List<Person> lst = new List<Person>();  
    //partial list of the "Winterfell Castle" characters  
    if (home.Equals("Winterfell Castle")) {  
        lst.Add(new Person("Catelyn Stark", 40));  
        lst.Add(new Person("Sansa Stark", 14));  
        lst.Add(new Person("Jon Snow", 22));  
    }  
    return lst;  
}  
  
// accept name & age, return a 'younger' Person object -----  
[WebMethod]  
public Person rejuvenatePerson(string name, int age)  
{  
    return new Person(name, age - 1);  
}  
  
} //class  
} //namespace
```

Android & WeServices

Example1: A Sample of C#.NET Webservices

Page 3 of 3

```
namespace WebServiceDemo1
{
    [Serializable]
    public class Person
    {
        private string _personName;
        private int _personAge;

        public Person( String personNameValue, int personAgeValue )
        {
            this._personName = personNameValue;  this._personAge = personAgeValue;
        }
        public Person( )
        {
            this._personName = "NA";  this._personAge = 0;
        }
        public string personName
        {
            get { return this._personName; }
            set { this._personName = value; }
        }
        public int personAge
        {
            get { return this._personAge; }
            set { this._personAge = value; }
        }
    }
}
```

Person (C#.NET)

+ Person(string, int)
+ personName: String
+ personAge: int

There is an equivalent
Java (POJO) version of
this class in the
Android's app space.

Android & WebServices

Example1: C# Webservices - Comments

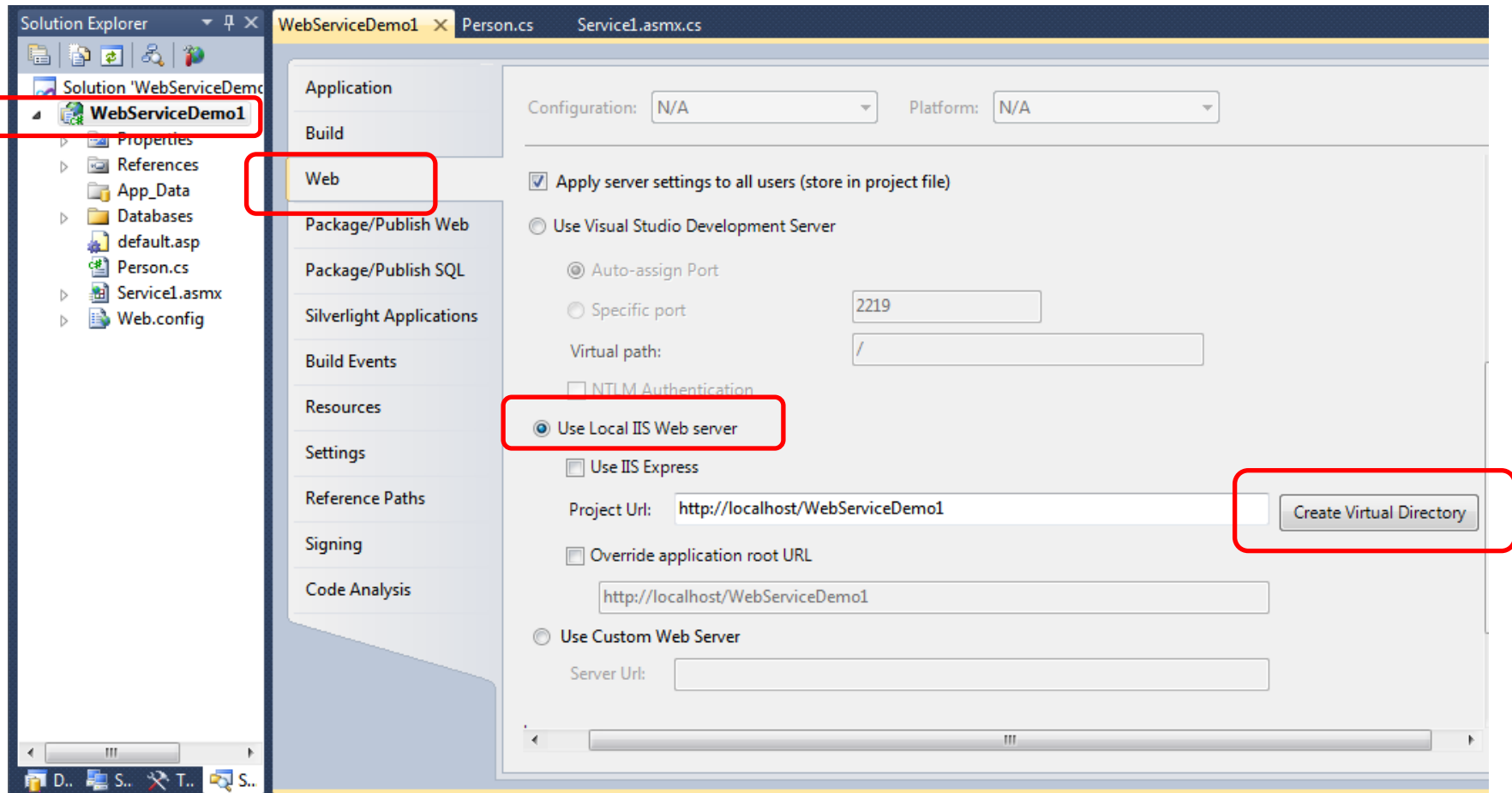
1. Our IIS webservice is implemented in C#.Net; however any .NET language could be used . The entry called **Namespace** identifies the IIS workspace hosting the WebMethods to be called by the Android client app. The literal value of this namespace is important as it will be referenced later by the client app as part of its request object.
2. In our example the .NET service **getPersonList** accepts a string naming a location (such as '*Winterfell Castle*') and returns a list of its fictional inhabitants (**Person** objects made according to the definition shown by Bullet 3). An answer encoded in XML format is sent back to the client. For instance in our example, the returned string is as follows (only a few lines are shown):

```
<ArrayOfPerson xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://MyHostNameOrIPAddress/">
  <Person>
    <personName>Catelyn Stark</personName>
    <personAge>40</personAge>
  </Person>
  <Person>
    <personName>Sansa Stark</personName>
    <personAge>14</personAge>
  </Person>
</ArrayOfPerson>
```

Android & WebServices

Example1: C# Webservices - Comments

- Important Note:** Remember to modify the .NET application's properties as follows: On the 'Solution Explorer' look for the application, Right-Click > Properties > Web > Use Local IIS server > Create Virtual Directory.



Android & WebServices

Example1: Android SOAP Web-Client + .NET Webservices

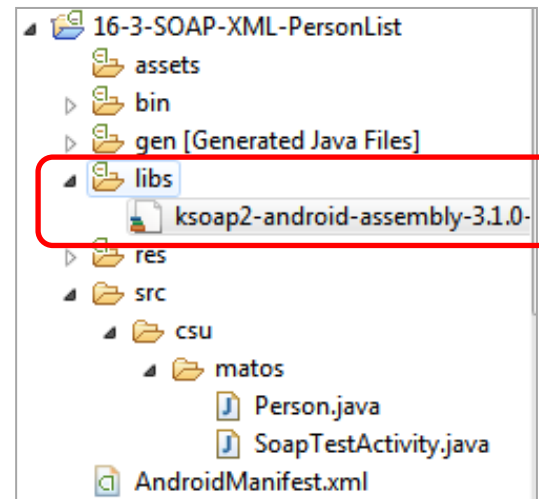
```
Calling .NET web service...

RESPONSE
anyType{Person=anyType{personName=Catelyn Stark;
personAge=40; };
Person=anyType{personName=Sansa Stark;
personAge=14; }; Person=anyType{personName=Jon
Snow; personAge=22; }; }

PERSON-LIST
Row-0
KSOAP
anyType{personName=Catelyn Stark; personAge=40; }
JAVA:
→ Person [name=Catelyn Stark, age=40]
Row-1
KSOAP
anyType{personName=Sansa Stark; personAge=14; }
JAVA:
→ Person [name=Sansa Stark, age=14]
Row-2
KSOAP
anyType{personName=Jon Snow; personAge=22; }
JAVA:
→ Person [name=Jon Snow, age=22]
```

Figure 4. Android WebClient App

The figure shows the decoded response for the request: **getPersonList (home)** which returns a partial list of the fictional inhabitants of a given home location (eg. "*Winterfell Castle*").



NOTE: You need to place a copy of the KSOAP jar file in the app's **/libs** folder. The **KSOAP** download link is:

<http://code.google.com/p/ksoap2-android/>

Android & WebServices

Example1: Android SOAP-Client Consuming .NET Webservices

```
public class SoapTestActivity extends Activity {
    TextView result;

    // use handler to keep GUI update on behalf of background tasks
    Handler handler = new Handler() {
        @Override
        public void handleMessage(Message msg) {
            // TODO Auto-generated method stub
            super.handleMessage(msg);
            String text = (String) msg.obj;
            result.append("\n" + text);
        }
    };

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_soap_test);
        result = (TextView) findViewById(R.id.result);
    }
}
```

Android & WebServices

Example1: Android SOAP-Client Consuming .NET Webservices

```
//do slow calls to remote server in a background thread
Thread slowJob = new Thread() {
    @Override
    public void run() {
        // IP address at home
        final String URL = "http://192.168.1.66/WebServiceDemo1/Service1.asmx";
        final String NAMESPACE = "http://MyHostNameOrIPAddress/";
        final String METHOD_NAME = "getPersonList";
        String resultValue = "";

        try {
            //prepare SOAP REQUEST (namespace, method, arguments)
            SoapObject request = new SoapObject(NAMESPACE, METHOD_NAME);
            //passing primitive (simple) input parameters
            request.addProperty("home", "Winterfell Castle");

            //prepare ENVELOPE put request inside
            SoapSerializationEnvelope envelope = new SoapSerializationEnvelope(
                SoapEnvelope.VER11);

            envelope.dotNet = true;
            envelope.setOutputSoapObject(request);
            //tell the type of complex object to be returned by service
            envelope.addMapping(NAMESPACE, METHOD_NAME,
                new ArrayList<Person>().getClass());
```

2 →

3 →

Android & WebServices

Example1: Android SOAP-Client Consuming .NET Webservices

```
// TRANSPORT envelope to destination set by URL (call & wait)
HttpTransportSE androidHttpTransport = new HttpTransportSE( URL);
androidHttpTransport.call(NAMESPACE + METHOD_NAME, envelope);

// receiving a complex response object (list of Person objects)
SoapObject response = (SoapObject) envelope.getResponse();

if (response == null) {
    resultValue = "NULL response received";
} else {
    // get ready to show rows arriving from the server
    resultValue = "RESPONSE\n" + response.toString();

    resultValue += "\n\nPERSON-LIST";

    //use KSOAP access methods to parse and extract data from response
    for (int i = 0; i < response.getPropertyCount(); i++) {
        resultValue += "\nRow-" + i;
        resultValue += "\n\tKSOAP\n\t" + response.getProperty(i);

        SoapObject personObj = (SoapObject)response.getProperty(i);
        Person p = new Person(personObj);

        resultValue += "\n\tJAVA:\n\t" + p.toString();
    }
}
```

4 →

5 →

6 →

Android & WebServices

Example1: Android SOAP-Client Consuming .NET Webservices

```
    } catch (Exception e) {
        resultValue = "\nERROR: " + e.getMessage();
    }
    // send message to handler so it updates GUI
    Message msg = handler.obtainMessage();
    msg.obj = (String) resultValue;
    handler.sendMessage(msg);

}
};
slowJob.start();

} // onCreate
}
```

7 →

Android & WebServices

Example1: Java Person Class & KSOAP Serialization

```
import org.ksoap2.serialization.SoapObject;

public class Person {
    private String name;
    private int age;
    // constructors
    public Person() {
        this.name = "na"; this.age = -1;
    }
    public Person(String name, int age) {
        this.name = name; this.age = age;
    }
    // create a local Java Person object using the KSOAP response (C#) object
    public Person(SoapObject obj) {
        this.name = obj.getProperty("personName").toString();
        this.age = Integer.parseInt(obj.getProperty("personAge").toString());
    }
    // accessors (get/set) omitted for brevity...

    @Override
    public String toString() {
        return "Person [name=" + name + ", age=" + age + "]";
    }
}
```

Person

- name: String

- age: integer

+ Person(SoapObject)

+ toString(): String

8

Android & WebServices

Example1: Android SOAP Client App - Comments

1. The Android webclient uses a background **Thread** to offload the task of communicating with the (possibly slow) webserver while keeping the app's GUI responsive. Updates to the app's GUI are made by the main thread from the messages sent by the worker thread through a **Handler** object.
2. The client prepares its **request** SoapObject indicating the Namespace and WebMethod in that workspace that needs to be executed. Then, each parameter (and its value) sent to the invoked method is added with the **.addProperty** clause.
3. A **SoapSerializationEnvelope** is made to carry the user's **request** to the server. The envelope's **dotNet** property indicates that it expects results to be generated by the Windows server. The **.addMapping** method is used to identify the Java type of the data returning inside the SOAP envelope. In this example, the clause
new ArrayList<Person>().getClass()
tells the response consists of a list of user-defined **Person** objects.

Android & WebServices

Example1: Android SOAP Client App - Comments

4. The SOAP envelope is finally transported to the server designated by the given **URL**. After that, the Android app waits for a response.
5. The **envelope.getResponse()** clause eventually sends to the Android client a collection of Person objects encoded in **XML** format (the formal structure of the XML dataset is available in the endpoint's WSDL contract). The statement **response.toString()** converts the XML string into equivalent KSOAP notation. For instance, in our example we obtain:

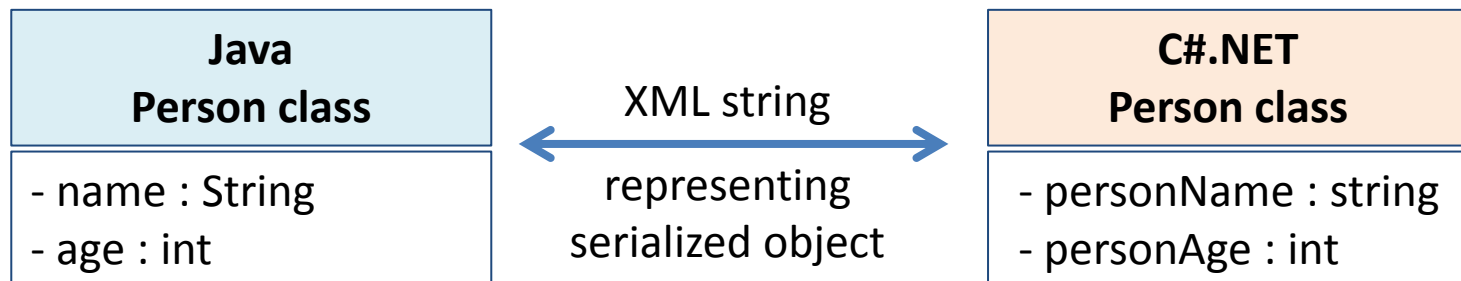
```
anyType{ Person=anyType{personName=Catelyn Stark; personAge=40; };  
         Person=anyType{personName=Sansa Stark; personAge=14; };  
         Person=anyType{personName=Jon Snow; personAge=22; }; }
```

6. The method **response.getPropertyCount()** is used to determine the size of the returned array. The **response.getProperty(i)** extracts the i-th row from the response object. Each row (representing a .NET serialized person) is passed to the local Java Person-class constructor (see Bullet 8).

Android & WebServices

Example1: Android SOAP Client App - Comments

7. A message is sent from the background worker thread to the main app so its GUI could be updated with the data received from the webservice.
8. The Java Person constructor is given (Bullet 5) a **SoapObject** holding encoded Person data, for instance, **obj** looks like the following string:
`Person=anyType{personName=Catelyn Stark; personAge=40; }`
This string is dissected using the KSOAP parsing methods **obj.getProperty("personName")** and **obj.getProperty("personAge")**. These values are then used to make the equivalent Java object
`Person[name=Catelyn Stark, age=40]`

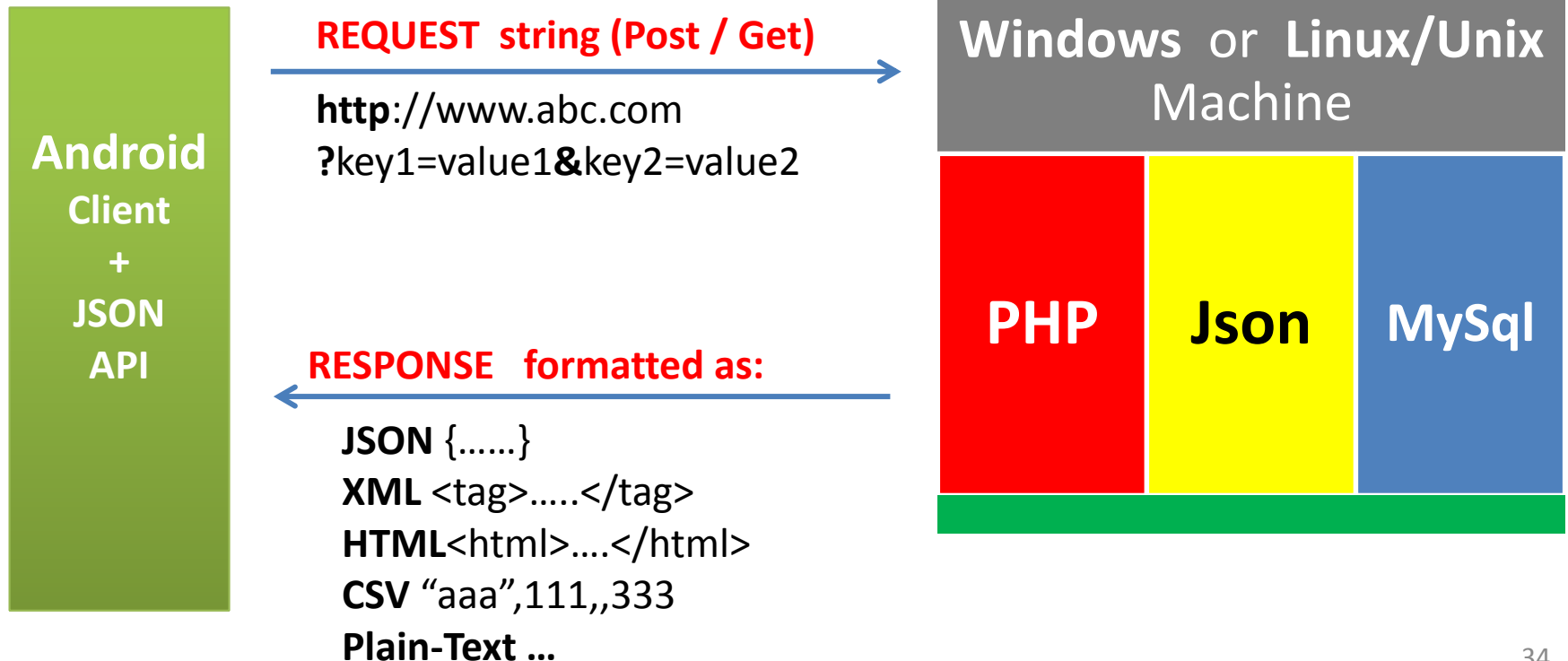


Android & WebServices

Example 2:

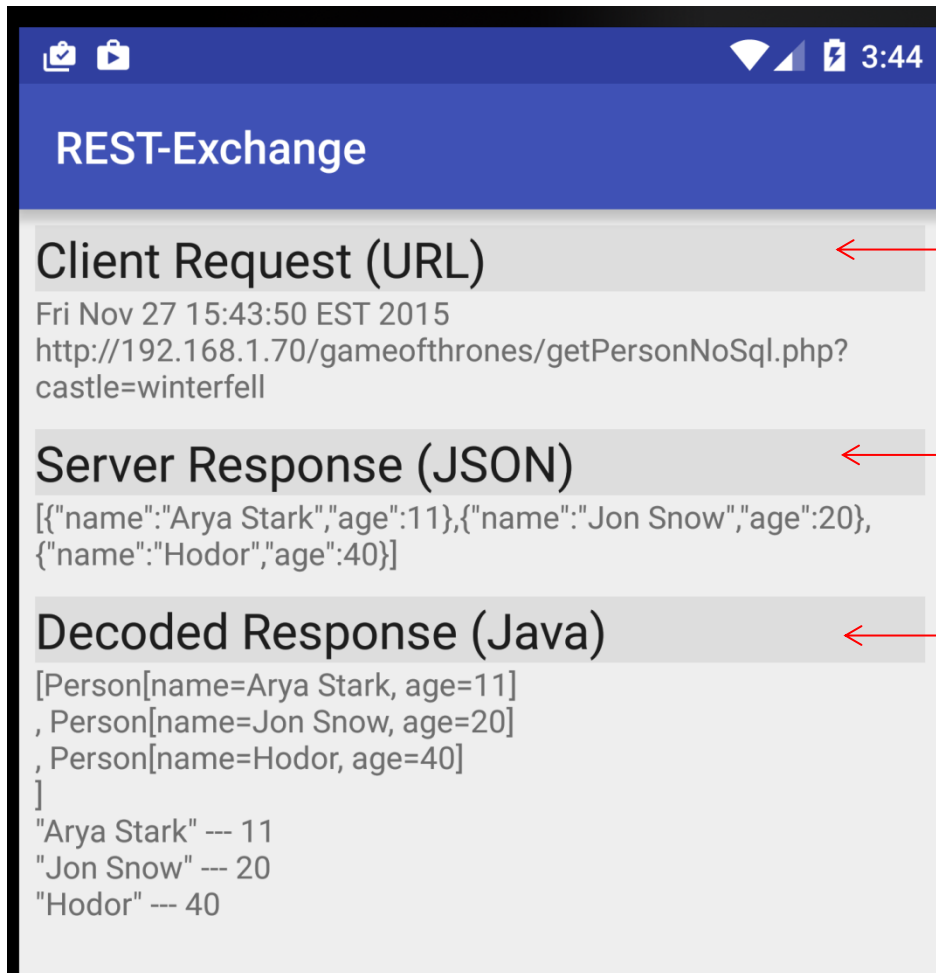
Using an Android REST-based Client + PHP Webservices

In this second example an Android client uses **REST** protocol to interact with a **MS-Windows IIS Server**. WebServices are implemented as a set of **PHP** programs. We use **JSON** encoding for the client-server data exchange.



Android & WebServices

Example 2: Using an Android REST-based Client + PHP services



The screenshot shows an Android application interface with a blue header titled "REST-Exchange". The main content area is divided into three sections, each with a red arrow pointing to it from a text box on the right:

- Client Request (URL)**: Shows the date and time "Fri Nov 27 15:43:50 EST 2015" and the URL "http://192.168.1.70/gameofthrones/getPersonNoSql.php?castle=winterfell".
- Server Response (JSON)**: Shows the JSON response: [{"name": "Arya Stark", "age": 11}, {"name": "Jon Snow", "age": 20}, {"name": "Hodor", "age": 40}].
- Decoded Response (Java)**: Shows the Java representation of the response: [Person[name=Arya Stark, age=11], Person[name=Jon Snow, age=20], Person[name=Hodor, age=40]] and a list of name-age pairs: "Arya Stark" --- 11, "Jon Snow" --- 20, "Hodor" --- 40.

The remote PHP Webservice allows the retrieving of people from a given castle

Response from the server arrives in **JSON** notation, later it is translated to an equivalent Java collection

Android & WebServices

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="7dp"    android:orientation="vertical" >
    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Client Request (URL)" />
    <TextView
        android:id="@+id/txtRequestUrl"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="URL goes here..." />
    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginTop="10dp"
        android:text="Server Response (JSON)" />
    <TextView
        android:id="@+id/txtResponseJson"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Wait - JSON string goes here ..." />
    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginTop="10dp"
        android:text="Decoded Response (Java)" />
    <TextView
        android:id="@+id/txtResponseJava"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Wait - decoded JSON here ..." />
</LinearLayout>
```

Example 2: App's Layout

 REST-JSON

Client Request (URL)

http://informatos.org/gameofthrones/
getPersonMySql.php?castle=king's+Handing

Server Response (JSON)

```
[{"name": "Tiryion Lannister", "age": 30}, {"name": "Cersei  
Baratheon", "age": 32}]
```

Decoded Response (Java)

```
[Person[name=Tiryion Lannister, age=30]  
, Person[name=Cersei Baratheon, age=32]  
]  
"Tiryion Lannister" -- 30  
"Cersei Baratheon" -- 32
```

Android & WebServices

Example 2: Android REST-based Client

```
import android.app.Activity;
import android.app.ProgressDialog;
import android.os.AsyncTask;
import android.os.Bundle;
import android.util.Log;
import android.widget.TextView;

import com.google.gson.Gson; ←
import com.google.gson.JsonArray;
import com.google.gson.JsonElement;
import com.google.gson.JsonObject;
import com.google.gson.JsonParser;
import com.google.gson.JsonSyntaxException;
import com.google.gson.reflect.TypeToken;

import java.net.HttpURLConnection;
import java.net.URI;
import java.net.URL;
import java.io.BufferedReader;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.lang.reflect.Type;
import java.net.URISyntaxException;
import java.util.ArrayList;
import java.util.Date;
```

This app needs to import a support library. For details on how to setup external JARS, see Lesson 14 - Example 7.

Gson downloaded from

<http://mvnrepository.com/artifact/com.google.code.gson/gson>

Android & WebServices

Example 2: Android REST-based Client

```
public class MainActivity extends Activity {
    TextView txtRequestUrl, txtResponseJson, txtResponseJava;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        txtRequestUrl = (TextView) findViewById(R.id.txtRequestUrl);
        txtResponseJson = (TextView) findViewById(R.id.txtResponseJson);
        txtResponseJava = (TextView) findViewById(R.id.txtResponseJava);

        // REQUEST object consists of "URL?ARGUMENTS" (spaces replaced by +)
        // each argument is a KEY=VALUE pair.

        // -----
        // SAMPLE1: Calling a webserver offering services implemented as PHP functions
        // arguments offered as      ?castle=value
        // where possible castle values are: Winterfell, Kings_Landing, DragonStone
        // -----

        // Testing on local Windows machine: IIS + PHP
        // You should get your machine's real IP address using c> ipconfig /all
        String SERVER_URL =
            "http://192.168.1.70/gameofthrones/getPersonNoSql.php?castle=winterfell";
    }
}
```

1



Android & WebServices

Example 2: Android REST-based Client

```
// String SERVER_URL =
// "http://192.168.1.70/gameofthrones/getPersonMySql.php?castle=kings_landing";

// Host is a Unix machine at CSU + PHP services
// String SERVER_URL =
// "http://grail.csuohio.edu/~matos/gameofthrones/getPersonNoSql.php"
// "?castle=Winterfell";

// -----
// Host is a commercial UNIX site running PHP services + MySQL databases
// String SERVER_URL =
// "http://www.informatos.org/gameofthrones/getPersonNoSql.php?castle=winterfell";

// String SERVER_URL =
// "http://informatos.org/gameofthrones/getPersonMySql.php?castle=kings_landing";

// -----
// SAMPLE2: This solution is based on Java code running on the server and the
// client. Java-servlet service running on a local Tomcat server(port :8080)
// String SERVER_URL =
// "http://192.168.1.70:8080/GameOfThrones/GetHeroes?castle=winterfell";
// String SERVER_URL =
// "http://192.168.1.70:8080/GameOfThrones/GetHeroes?castle=dragonstone";
```

Android & WebServices

Example 2: Android REST-based Client

```
txtRequestUrl.setText(new Date() + "\n" + SERVER_URL);
```

```
// Use AsyncTask to execute potential slow task without freezing GUI
```

```
new LongOperation().execute(SERVER_URL);
```

```
}// main
```

```
private class LongOperation extends AsyncTask<String, Void, Void> {
```

```
private String jsonResponse;
```

```
private ProgressDialog dialog = new ProgressDialog(MainActivity.this);
```

```
protected void onPreExecute() {  
    dialog.setMessage("Please wait..");  
    dialog.show();  
}
```

```
2 → protected void doInBackground(String... urls) {
```

```
    try {
```

```
        // WARNING -----  
        // You must use actual IP addresses, do not enter "localhost:8080/..."  
        // try something like: http://192.168.1.70:8080/Service/Method?Args  
        // solution uses Java.Net class (Apache.HttpClient is now deprecated)
```


Android & WebServices

Example 2: Android REST-based Client

```
// STEP1. Create a HttpURLConnection object releasing REQUEST to given site
URL url = new URL(urls[0]); //argument supplied in the call to AsyncTask
HttpURLConnection urlConnection = (HttpURLConnection)url.openConnection();
urlConnection.setRequestProperty("User-Agent", "");
urlConnection.setRequestMethod("POST");
urlConnection.setDoInput(true);
urlConnection.connect();

// STEP2. wait for incoming RESPONSE stream, place data in a buffer
InputStream isResponse = urlConnection.getInputStream();
BufferedReader responseBuffer = new BufferedReader(new
    InputStreamReader(isResponse));

// STEP3. Arriving JSON fragments are concatenate into a StringBuilder
String myLine = "";
StringBuilder strBuilder = new StringBuilder();
while ((myLine = responseBuffer.readLine()) != null) {
    strBuilder.append(myLine);
}
//show response (JSON encoded data)
jsonResponse = strBuilder.toString();
Log.e("RESPONSE", jsonResponse);
} catch (Exception e) { Log.e("RESPONSE Error", e.getMessage()); }

return null; // needed to gracefully terminate Void method
}
```

3 →

Android & WebServices

Example 2: Android REST-based Client

```
protected void onPostExecute(Void unused) {

    try {
        dialog.dismiss();
        // update GUI with JSON Response
        txtResponseJson.setText(jsonResponse);

        // Step4. Convert JSON list into a Java collection of Person objects
        // prepare to decode JSON response and create Java list
        Gson gson = new Gson();
        Log.e("PostExecute", "content: " + jsonResponse);

        // set (host) Java type of encoded JSON response
        Type listType = new TypeToken<ArrayList<Person>>() { }.getType();
        Log.e("PostExecute", "arrayType: " + listType.toString());

        // decode JSON string into appropriate Java container
        ArrayList<Person> personList = gson.fromJson(jsonResponse, listType);
        Log.e("PostExecute", "OutputData: " + personList.toString());

        // Step5. Show results (update GUI with Java version of retrieved list)
        txtResponseJava.setText(personList.toString());

        // OPTIONAL.The following strategy shows an alternative mechanism to
        // interpret the returned JSON response. Here you parse the nodes of the
```

4 →

Android & WebServices

Example 2: Android REST-based Client

```
// underlying data structure using GSON element classes. A JsonElement could
// be a: JsonObject, JsonArray, or JsonPrimitive
String result = "\n";
try {

    JsonElement jelement = new JsonParser().parse(jsonResponse);
    JsonArray jarray = jelement.getAsJsonArray();
    for (int i = 0; i < jarray.size(); i++) {
        JsonObject jobject = jarray.get(i).getAsJsonObject();
        result += jobject.get("name").toString() + " --- "
            + jobject.get("age").toString() + "\n";
    }

} catch (Exception e) {
    Log.e("PARSING", e.getMessage());
}
txtResponseJava.append(result);

} catch (JsonSyntaxException e) {
    Log.e("POST-Execute", e.getMessage());
}
}
} // AsyncTask
} // class
```

Android & WebServices

Example 2: Android REST-based Client - Comments

1. The client app uses an **AsyncTask** object to call the remote WebService. This practice is functionally equivalent to running a background thread but offers more options (such as showing progress messages).
2. **doInBackground** takes the supplied URL to reach the server and request its assistance. The **Java.Net** class is responsible for establishing the asynchronous HTTP client-server exchange. In our example an HTTP GET operation is invoked using the URL:

<http://192.168.1.66/gameofthrones/getPersonMySQL.php?castle=King's+Landing>

the called PHP method **getPersonMySQL** extracts its result from a MySQL database. Observe that URL arguments appear after the **?** symbol. You may also replace *spaces* with **+** symbols as in **castle=King's+Landing**

3. The client app collects the **JSON** encoded result in a **StringBuilder**. In our example the returned string is: [{"name":"Tiryion Lannister", "age":30}, {"name":"Cersei Baratheon", "age":32}]

Android & WebServices

Example 2: Android REST-based Client - Comments

4. The **onPostExecute** portion of the AsyncTask decodes the JSON response string into a Java ArrayList<Person> collection.

We have chosen **Google's GSON API** for processing the incoming JSON string. The example shows two approaches for decoding JSON data:

- (a) The first invokes the **fromJson(data, type)** method,
- (b) the second alternative **parses** the JSON string looking for the occurrence of JsonElement, JsonArray, and JsonObject tokens.

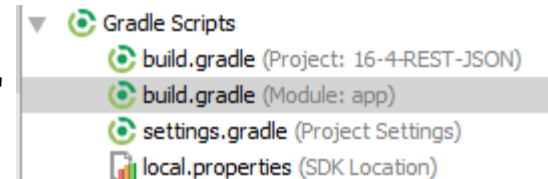
For more details on this issue see Lesson 14

Android & WebServices

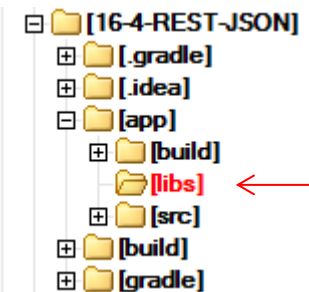
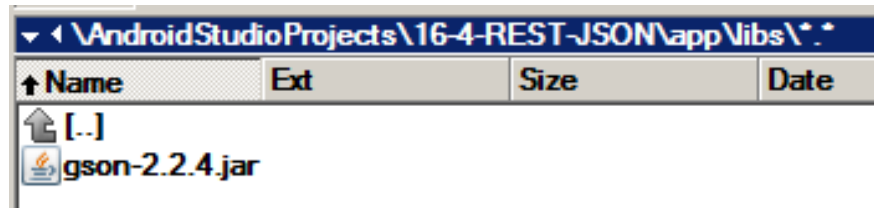
Example 2: Android REST-based Client - Comments

5. **Final details:** Your Gradle Scripts/build.gradle(Module:app) file needs to be modified to include references to the GSON API. In our example we have:

```
...
dependencies {
    compile 'com.android.support:support-v4:19.1.0'
    → compile files('libs/gson-2.2.4.jar')
}
```



6. Imported JARS should be stored in the **apps/lib** folder



7. The Manifest must include request to use **Internet**.

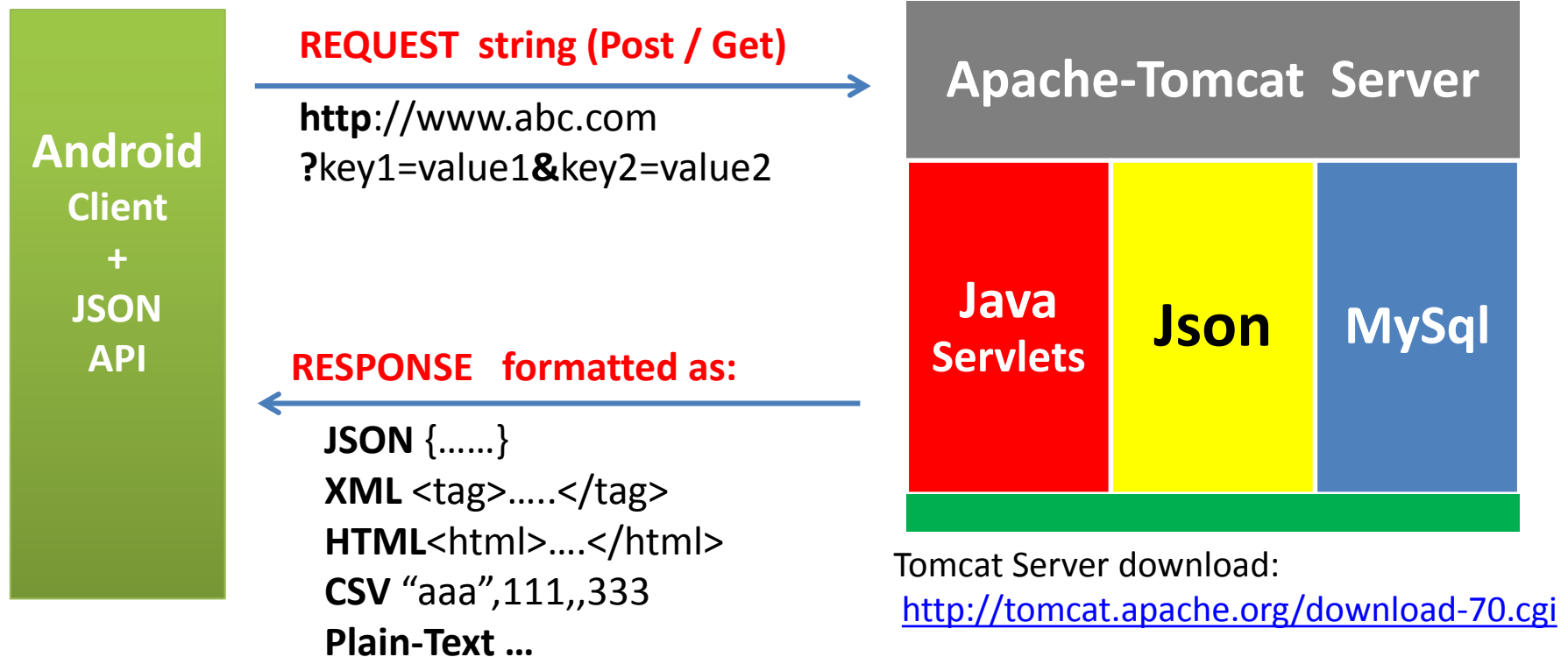
```
<uses-permission android:name="android.permission.INTERNET"/>
```

Android & WebServices

Example 3: Android REST consuming Servlet Webservices

This problem is similar to Example-2. An Android REST-based client supplies a location and a webservice retrieves all inhabitants of that place.

As before we divide the problem in two parts, first we discuss step-by-step how to create the **Tomcat hosted Servlet**, then we build an Android client using REST to communicate with it.



Android & WebServices

Example 3: Android REST consuming Servlet Services

- An **HttpServlet** is a Java class used to extend the functionality of a web server. It is commonly used to provide dynamic web content in a manner similar to the ASP, .NET and PHP technologies.
- A **Servlet** primarily operates on two distinct objects:
 - **HttpServletRequest:** Represents a client's request. This object gives a servlet access to incoming information such as HTML form data and HTTP request headers.
 - **HttpServletResponse:** Represents the servlet's response. The servlet uses this object to return data to the client such as HTTP errors, response headers (Content-Type, Set-Cookie, and others), and output data by writing to the response's output stream or output writer.

Reference:

<http://download.oracle.com/javase/6/api/javax/servlet/http/HttpServletRequest.html>

Android & WebServices

Example 3: Create a Servlet

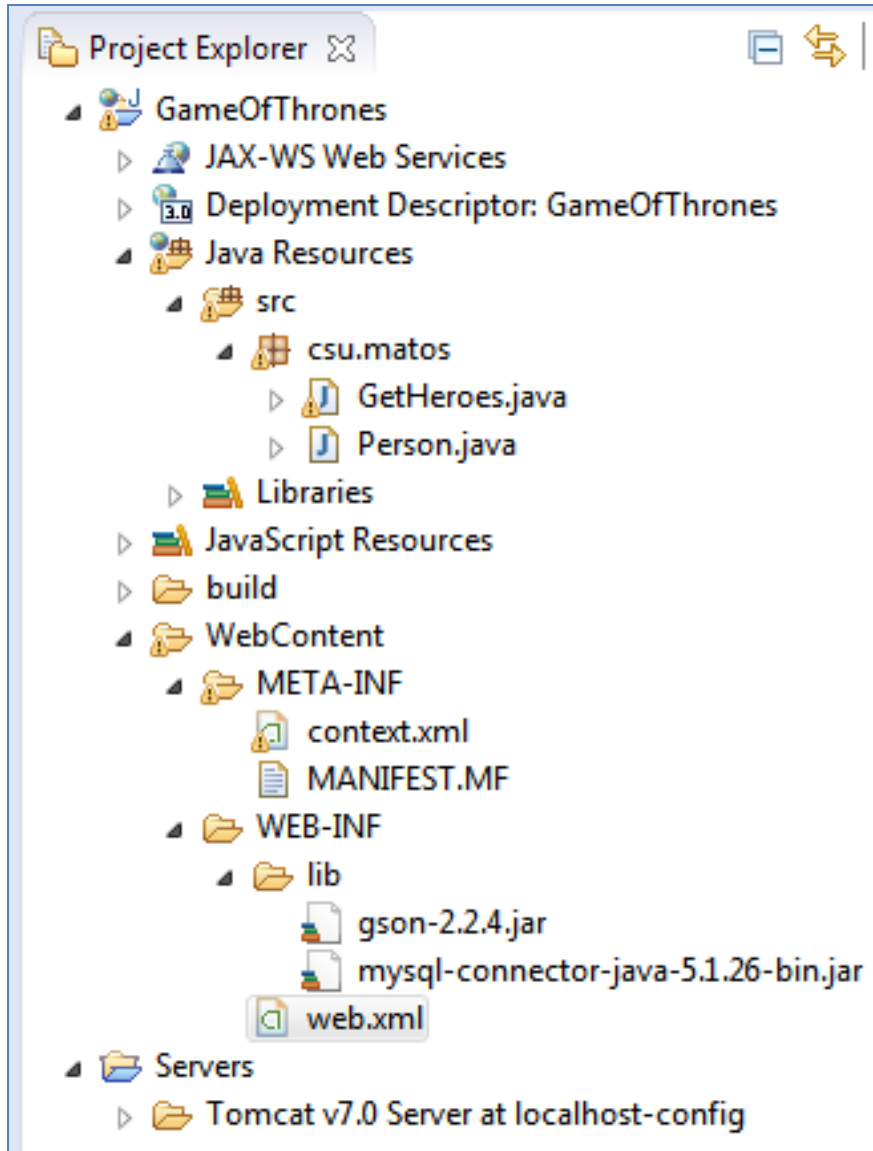
We assume you are using **Eclipse EE** and Apache **Tomcat 7.0** (or newer).

1. From **Eclipse** - create a new *Dynamic Web project*.
 - Set project name to: **GameOfThrones**.
 - On 'Target runtime' pane, click 'New runtime'. From the drop-down list choose **Tomcat Server v7.0** (or a newer version). Click 'Next' button.
 - In textbox 'Tomcat Installation directory' enter the location where your Tomcat software is placed (usually: C:\Program Files\Apache Software Foundation\Tomcat 7.0). Click button 'Finish' > 'Finish'.
2. Add new package **csu.yourlastname** to the folder **/JavaResources/src**
3. Place inside the package a new **Servlet**. In the 'Class name' box enter: **GetHeroes**. Click the button 'Finish'.
4. Add to the package the **Person** class .
5. Copy to the folder: **WebContent/WEB-INF/lib** the **GSON** and **MySQL-JConnector** jars. They are needed to support JSON encoding and access to a MySql database.



Android & WebServices

Example 3: Create a Servlet



6. Create inside the folder **WebContent/META-INF/** a XML file called **context.xml** indicating how to access your MySQL database (*code will be shown shortly*).
7. Create inside the folder **WebContent/WEB-INF** a XML file called **web.xml** explaining the parts of the service and its mapping of symbolic names to Java classes (*shown ahead*).

The structure of your Java Servlet should be as indicated in the figure. Add pending code, and test your app. *Make sure that only the Eclipse developer's web server is running.*

Android & WebServices

Example 3: Create a Servlet - Test Results

<http://localhost:8080/GameOfThrones/GetHeroes?castle=winterfel>

```
[0] NEW TEST: Wed Feb 19 13:33:56 EST 2014 [2] >>> castle winterfel [3]
PARAMETER castle: winterfel [4] TAKEN FROM MEMORY-DB: [{"name": "Arya
Stark", "age": 11, "castle": "Winterfel Castle"}, {"name": "Jon
Snow", "age": 20, "castle": "Winterfel Castle"}, {"name": "Hodor", "age": 40, "castle": "Winterfel
Castle"}] [5] TAKEN FROM MYSQL-DB: [{"name": "Arya
Stark", "age": 11, "castle": "Winterfel Castle"}, {"name": "Jon
Snow", "age": 20, "castle": "Winterfel Castle"}, {"name": "Hodor", "age": 40, "castle": "Winterfel
Castle"}]
```

<http://localhost:8080/GameOfThrones/GetHeroes?castle=dragonstone>

```
[0] NEW TEST: Wed Feb 19 13:34:59 EST 2014 [2] >>> castle dragonstone [3]
PARAMETER castle: dragonstone [4] TAKEN FROM MEMORY-DB:
[{"name": "Daenerys Targaryen", "age": 18, "castle": "Dragonstone"}] [5] TAKEN FROM
MYSQL-DB: [{"name": "Daenerys Targaryen", "age": 18, "castle": "Dragonstone"}]
```

Android & WebServices

Example 3: Create a Servlet

context.xml

This file indicates how to connect to a MySQL database. It includes the user's name, password, driver name, connection parameters, and database connection string. It must be placed in the **WebContent/META-INF/** folder.

```
<?xml version="1.0" encoding="UTF-8"?>
<Context>
  <!-- comments removed -->

  <Resource    name="jdbc/mysqlresourcegameofthrones"
               auth="Container"
               type="javax.sql.DataSource"
               maxActive="100" maxIdle="30" maxWait="10000"
               username="csuperson"
               password="euclid"
               driverClassName="com.mysql.jdbc.Driver"
               url="jdbc:mysql://192.168.1.66:3306/gameofthrones"/>

</Context>
```

Android & WebServices

Example 3: Create a Servlet

1 of 2

web.xml

This file lists the parts of the servlet, its resources, and indicates how to map the Servlet's java classes to webService names passed in the request object. It must be placed in the **WebContent/WEB-INF** folder.

```
<?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_3_0.xsd" id="WebApp_ID" version="3.0">

  <display-name>GameOfThrones</display-name>

  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
    <welcome-file>index.htm</welcome-file>
  </welcome-file-list>

  <servlet>
    <description></description>
    <display-name>GetHeroes</display-name>
    <servlet-name>GetHeroes</servlet-name>
    <servlet-class>csu.matos.GetHeroes</servlet-class>
  </servlet>
```

Android & WebServices

Example 3: Create a Servlet

2 of 2

web.xml

cont.

```
<servlet-mapping>
  <servlet-name>GetHeroes</servlet-name>
  <url-pattern>/GetHeroes</url-pattern>
</servlet-mapping>

<resource-ref>
  <description>MySQL Datasource - GameOfThrones Example</description>
  <res-ref-name>jdbc/mysqlresourcegameofthrones</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
</resource-ref>

</web-app>
```

Android & WebServices

Example 3: Create a Servlet - GetHeroes Servlet

1 of 5

GetHeores is the actual servlet. It does all its work inside the **doGet** method. The incoming request string is examined looking for the *argument=value* pairs it may carry. Once the *castle* value is know a list of its associated people is assembled. The servlet redundantly extracts its data from two sources, first it uses an in-memory collection of datasets, then it repeats the same type of retrieval querying a mySql database.

```
@WebServlet("/GetHeroes")
public class GetHeroes extends HttpServlet {

    private static final long serialVersionUID = 1L;

    1 → public void doGet(HttpServletRequest request, HttpServletResponse response)
           throws ServletException, IOException {

        PrintWriter output = response.getWriter();
        output.println(" [0] NEW TEST: " + new Date());

        Map<String, String[]> requestMap = request.getParameterMap();
        if (requestMap.isEmpty()) {
            output.println(" [1] <<<< SERVLET CALLED - Empty Request >>>> ");
        }
        for (String key : requestMap.keySet()) {
            String[] value = requestMap.get(key);
            output.println(" [2] >>> " + key + " " + value[0]);
        }
    }
}
```

Android & WebServices

Example 3: Create a Servlet - GetHeroes Servlet

2 of 5

```
// setup memory resident datasets (fake core database)
```

```
ArrayList<Person> winterfellPeople = new ArrayList<Person>();  
winterfellPeople.add(new Person("Arya Stark", 11, "Winterfell Castle"));  
winterfellPeople.add(new Person("Jon Snow", 20, "Winterfell Castle"));  
winterfellPeople.add(new Person("Hodor", 40, "Winterfell Castle"));
```

```
ArrayList<Person> dragonPeople = new ArrayList<Person>();  
dragonPeople.add(new Person("Daenerys Targaryen", 18, "Dragonstone"));
```

```
ArrayList<Person> kingsPeople = new ArrayList<Person>();  
kingsPeople.add(new Person("Tiryon Lannister", 30, "King's Landing"));  
kingsPeople.add(new Person("Cersei Baratheon", 32, "King's Landing"));
```

```
String castle = "";  
castle = request.getParameter("castle");  
output.println(" [3]PARAMETER castle: " + castle);
```

```
// Part1. add to peopleResult data from memory-held lists  
// if no castle is supplied, include all characters
```

```
ArrayList<Person> peopleResult = new ArrayList<Person>();  
if (castle==null){  
    castle = "";  
    peopleResult.addAll(winterfellPeople);  
    peopleResult.addAll(dragonPeople);  
    peopleResult.addAll(kingsPeople);
```


Android & WebServices

Example 3: Create a Servlet - GetHeroes Servlet

3 of 5

```
} else if ("winterfell castle".startsWith(castle)) {
    peopleResult = winterfellPeople;

} else if ("dragonstone".startsWith(castle)) {
    peopleResult = dragonPeople;

} else if ("king's landing".startsWith(castle)) {
    peopleResult = kingsPeople;
}

4 → // prepare to do GSON encoding of selected people
Gson gson = new Gson();
String jsonData = gson.toJson(peopleResult);
output.println(" [4] TAKEN FROM MEMORY-DB: " + jsonData);

// Part2. Redo peopleResult now retrieving from mysql db.
5 → try {
    peopleResult = getDbRecord(castle);
    jsonData = gson.toJson(peopleResult);
    output.println(" [5] TAKEN FROM MYSQL-DB: " + jsonData);
} catch (Exception e) {
    output.println("NO DATA");
}

output.flush();

} // doGet
```

Android & WebServices

Example 3: Create a Servlet - GetHeroes Servlet

4 of 5

```
@Override
protected void doPost(HttpServletRequest request,
                      HttpServletResponse response)
    throws ServletException, IOException {
    doGet(request, response);
}
// Retrieve designated people this time from a mySql database
private ArrayList<Person> getDbRecord(String castle) throws Exception {

ArrayList<Person> result = new ArrayList<Person>();
try
{
    javax.naming.Context initContext = new InitialContext();
    Context envContext = (Context)initContext.lookup("java:/comp/env");

    javax.sql.DataSource ds = (DataSource) envContext.lookup(
        "jdbc/mysqlresourcegameofthrones");
    java.sql.Connection cnn = ds.getConnection();

    String mySQL = " select name, age, castle "
        + " from person where castle LIKE '%" + castle + "%'";

    java.sql.Statement stm = cnn.createStatement();

    java.sql.ResultSet rs = stm.executeQuery(mySQL);
```

6 →

Android & WebServices

Example 3: Create a Servlet - GetHeroes Servlet

5 of 5

```
7 → while ( rs.next() ){
    String pname = rs.getString("name");
    int page = Integer.parseInt(rs.getString("age"));
    String pcastle = rs.getString("castle");

    Person pers = new Person(pname, page, pcastle);
    result.add(pers);
}

}
catch (java.sql.SQLException e) {
    throw new Exception ( " [*[Problems1 ]*] " + e.getMessage());

} catch (javax.naming.NamingException e) {
    throw new Exception ( " [*[Problems2 ]*] " + e.getMessage());

}
return result;

} //getDbRecord

} // class
```

Android & WebServices

Example 3: Create a Servlet - Person class

```
public class Person {  
    private String name;  
    private int age;  
    private String castle;  
  
    public Person(String name, int age, String castle) {  
        this.name = name; this.age = age; this.castle = castle;  
    }  
  
    public Person() {  
        this.name = "na"; this.age = -1; this.castle = "na";  
    }  
  
    public String getName() { return name; }  
    public void setName(String name) { this.name = name; }  
    public int getAge() { return age; }  
    public void setAge(int age) { this.age = age; }  
  
    @Override  
    public String toString() {  
        return "Person [name=" + name + ", age=" + age + ", castle=" + castle + "];"  
    }  
}
```

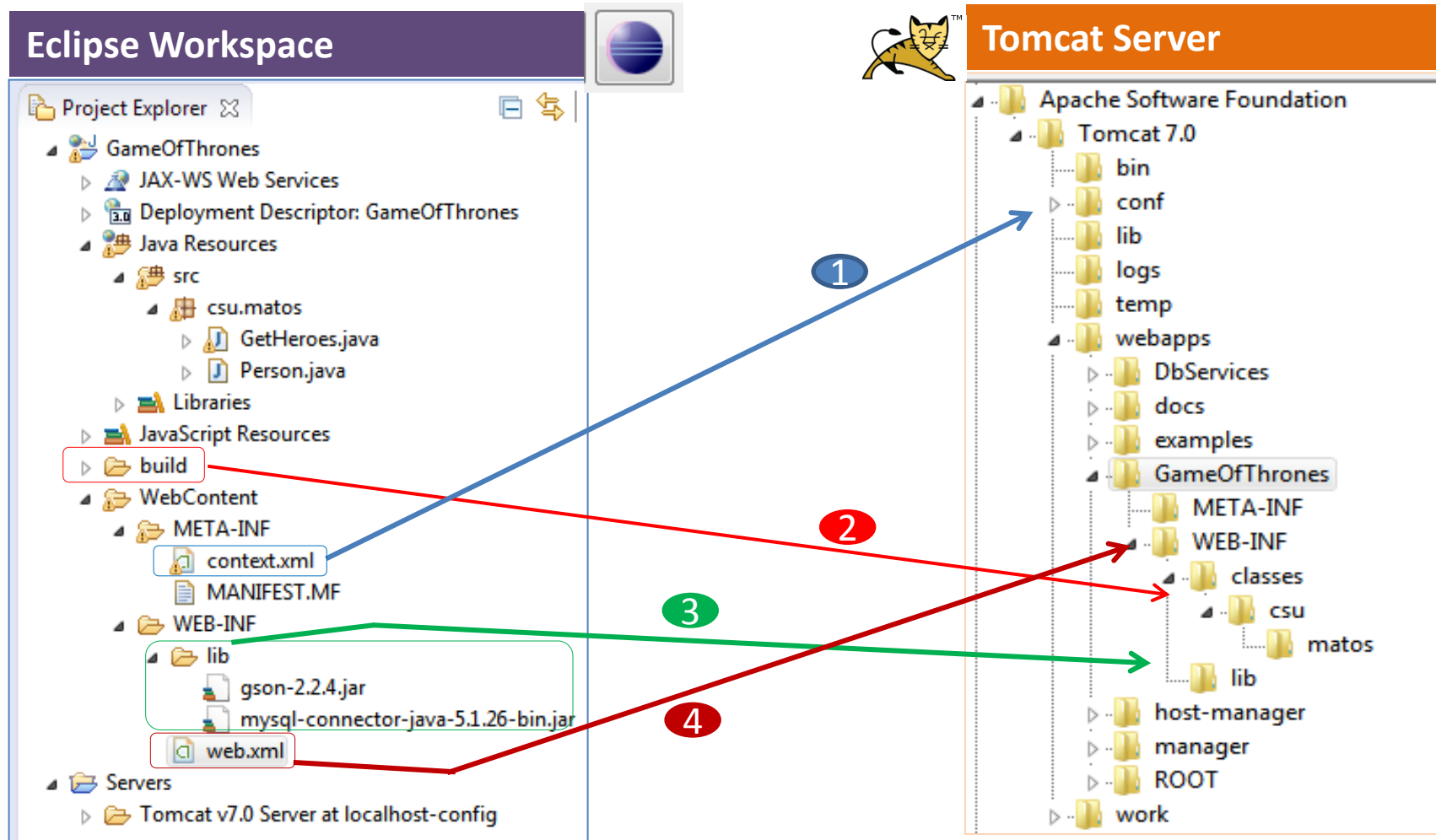
Android & WebServices

Example 3: Servlet - Comments

1. This servlet only implements its **doGet** method in which a **request** URL string is accepted and a **response** object is sent back to the caller.
2. For demonstration purposes the servlet returns twice the result-set. First data is selected from an in-memory (ArrayList-based) collection, the second time data comes from a relational database.
3. The incoming *castle* argument is extracted from the request URL, it indicates what group of people should be retrieved.
4. After a resultset consisting of Person objects held in an ArrayList is assembled, the collection is converted to a JSON string (we use GSON library, see Lecture 14).
5. The method **getDbRecord** redundantly retrieves a version of the resultset from a MySQL database. **DataSource** specs for reaching the database are taken from the global server's **context** file.
6. A select-statement adds the *castle* argument to its *where* clause to find the tuples satisfying the search condition.
7. Database rows are scanned one at the time. From each selected tuple a Person object is created and added to a growing ArrayList. After all rows are processed the ArrayList is returned.

Android & WebServices

Example 3: Transfer your Servlet to the Production Server



Android & WebServices

Example 3: Transfer your Servlet to the Production Server

Preliminary Steps

- Stop the Eclipse Tomcat server instance (no other webserver should be running).
- Locate the Tomcat production server in your system (usually at c:\Program Files\Apache Software Foundation\Tomcat 7.0). If needed create a folder called **/webapps/GameOfThrones**. Add the subdirectories **META-INF** and **WEB-INF**.

Transfer Files From Eclipse Workspace to the Tomcat Production Server

1. Modify the production server's file `\conf\context.xml`. Add to it a copy of the `<resource... />` XML entry from your Eclipse `WebContent\META-INF\context.xml` file. Observe that you may end-up with more than one global `<resource>` entry.
2. Use your *Windows Explorer* to find and copy the Eclipse's `build\classes` folder. Paste this copy in the server's `webapps\GameOfThrone\WEB-INF\` folder.
3. Copy each of the *jars* used by the servlet into the server's `\lib\` folder.
4. Place a copy of the `\WEB-INF\web.xml` file in the corresponding `\WEB-INF\` server's folder.
5. Restart the server.

Android & WebServices

Example 3: Transfer your Servlet to the Production Server

Run/Restart the Tomcat service (Control Panel > Administrative Tools > Services > Locate Apache Tomcat > Start). Run a browser with the following URL:

<http://localhost:8080/GameOfThrones/GetHeroes?castle=winterfell>

The returned page is shown below. Now, you may want to remove all the debugging commentary and only return the encoded JSON data that appears at the end [...]



```
[0] NEW TEST: Wed Feb 19 19:57:53 EST 2014
[2] >>> castle winterfel
[3] PARAMETER castle: winterfel
[4] TAKEN FROM MEMORY-DB: [{"name":"Arya Stark","age":11,"castle":"Winterfel Castle"},
{"name":"Jon Snow","age":20,"castle":"Winterfel Castle"},
{"name":"Hodor","age":40,"castle":"Winterfel Castle"}]
[5] TAKEN FROM MYSQL-DB: [{"name":"Arya Stark","age":11,"castle":"Winterfel Castle"},
{"name":"Jon Snow","age":20,"castle":"Winterfel Castle"},
{"name":"Hodor","age":40,"castle":"Winterfel Castle"}]
```


Android & WebServices

Example 3: Android Client

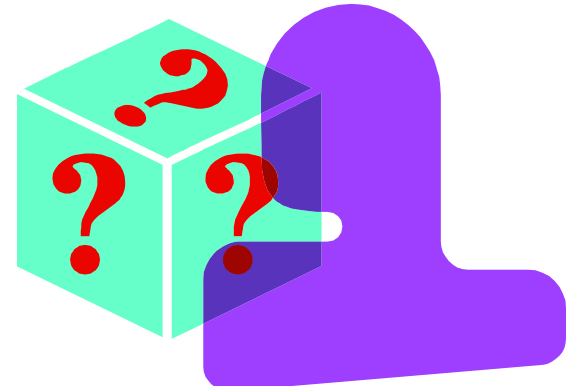
Use the same Android client from the previous example. Change the URL to the desired destination. For this example, test the client-app with the URL values:

<http://localhost:8080/GameOfThrones/GetHeroes?castle=winterfell>

<http://localhost:8080/GameOfThrones/GetHeroes?castle=king's+landing>

<http://localhost:8080/GameOfThrones/GetHeroes?castle=dragonstone>

Producing & Consuming Web Services



< Questions />

Appendix A. Connecting to Oracle DBMS



REST Protocol – Android & Apache's Tomcat Server

This relaces Step 8 in Example 2C. Multitier Application

Add the following **DataSource** to the application's **context.xml** file. Add the file to the **/WebControl/META-INF/** folder of your Eclipse workspace solution (later, this fragment will be copied to the Tomcat's **/conf/context.xml** file)

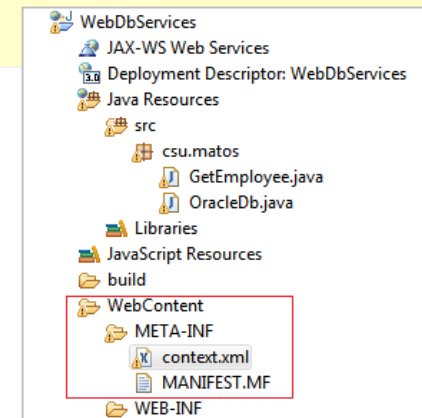
```
<?xml version="1.0" encoding="UTF-8"?>
<context>
  <Resource name="jdbc/myoracle" auth="Container"
    type="javax.sql.DataSource" driverClassName="oracle.jdbc.OracleDriver"
    url="jdbc:oracle:thin:@sancho.csuohio.edu:1521:ORCL"
    username="CSUPERSON" password="EUCLID" maxActive="20" maxIdle="10"
    maxWait="-1"/>
</context>
```

The above *DataSource* helps the JDBC connection identify the involved server and user. Other users could reuse the data-source and provide individual credentials.

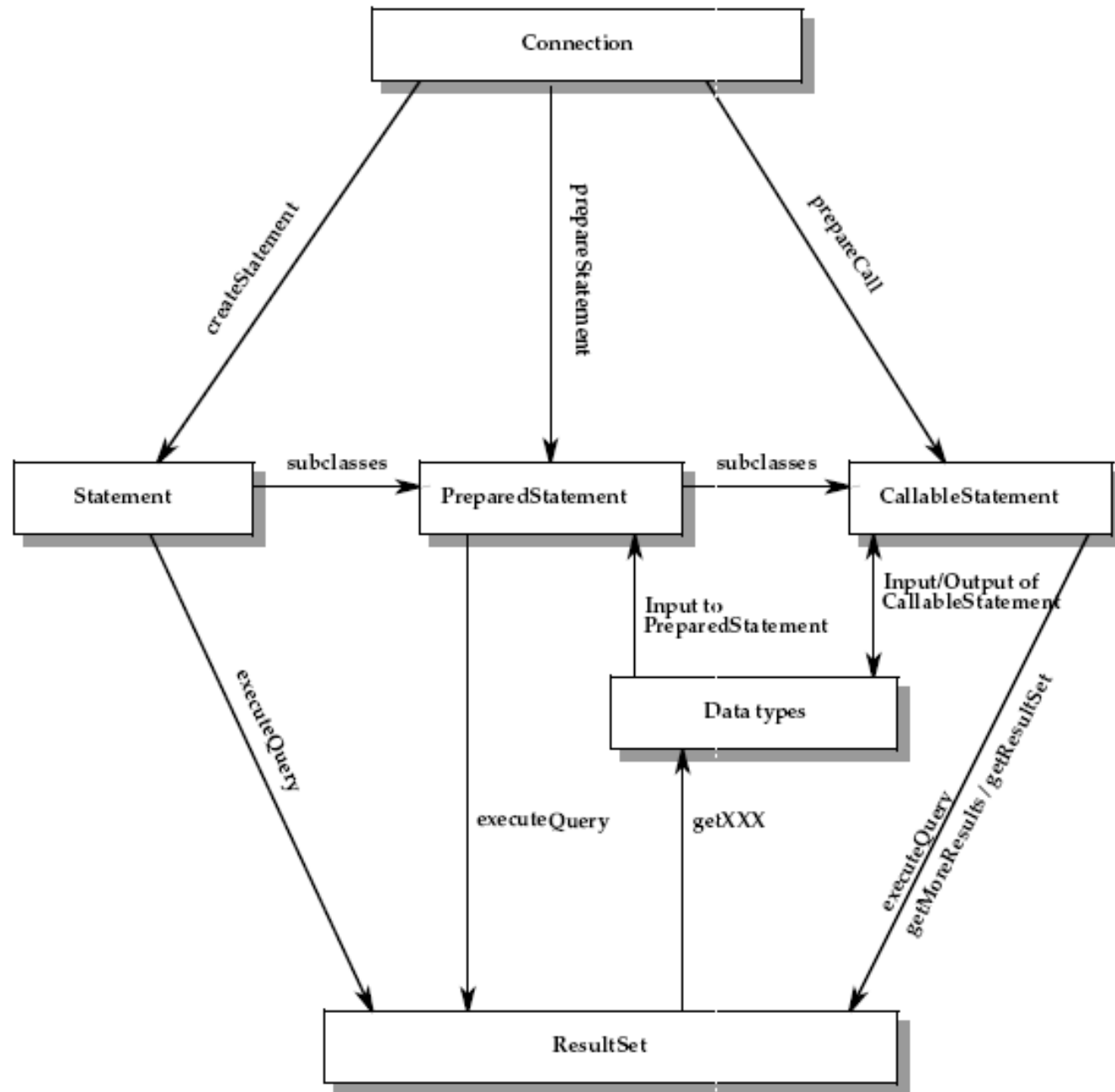
Reference:

<http://tomcat.apache.org/tomcat-5.5-doc/jndi-datasource-examples-howto.html>

<http://tomcat.apache.org/tomcat-5.5-doc/jndi-resources-howto.html>



Appendix B. JDBC Architecture



JDBC™ 4.0 Specification. JSR 221
Sun Microsystems
Lance Andersen, Specification Lead
November 7, 2006

FIGURE 5-1 Relationships between major classes and interface in the java.sql package