Bulletin of the *Transilvania* University of Braşov • Vol 7(56), No. 1 - 2014
Series III: Mathematics, Informatics, Physics, 81-94

RESTEasy JAX-RS LOGIN WEB SERVICE AND ANDROID CLIENT

Constantin Lucian ALDEA[1]

Abstract

In this paper the steps made to send requests and receive responses between an Android client application and a RESTEasy JAX-RS[2] login web service which is hosted on the JBoss application server are presented. A simple architecture for an enterprise application is used and an authentication mechanism is demonstrated. The client application (installed on an Android device) sends messages to the RESTful web service. The web service verifies the parameters against the database and prepares the response. The client application retrieves the response and performs the login or shows the failure message. The RESTful web service is deployed on the JBoss application server. The parameters and results exchanged between client and server application are wrapped into JSON[3] telegrams.

2000 *Mathematics Subject Classification:* 68N19.
Key words: restful web service, mobile device, security.

1 Introduction

Applications that offer mobile clients implement the next use case. A new or an existing application has a web user management module. By using the user management module users are created, modified, activated or deactivated. This module is also able to manage the user sessions and to keep track of user actions. Using an existing user the customer has access to the all functionalities (e.g. reports) of the application. In the classical way the customer accesses the web page and uses its username and password, logs into the application and uses the application functionalities [1]. Some functionality (reports, news, etc) must be ported to be accessible on the mobile devices. While the web interface of the enterprise application cannot be easily customized and used on the mobile

[1]Faculty of Mathematics and Informatics, *Transilvania* University of Braşov, Romania, e-mail: costel.aldea@unitbv.ro

[2]Java API for JBoss RESTful Web Services

[3]JavaScript Object Notation - readable texts used for data exchange between client and server applications/modules

devices, new client applications are required on the mobile device to intermediate the access to the application modules.

The web services allow the clients with limited resources to access remote resources in the context of their own application.

The mobile devices have limited resources and they use remote resources for providing end user services.

In this paper a service oriented strategy for communication between the mobile device client and the application deployed on the server is presented. Moreover, the mobile client application accesses the data stored into the application database by using the RESTful web services. By using the RESTful web services the client application can access the back-end database tables and other application modules in a controlled and secure way.

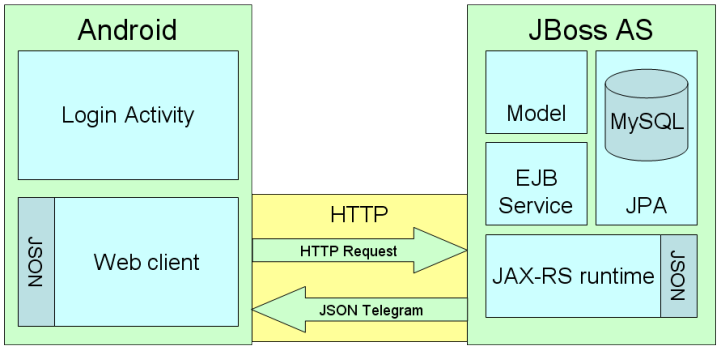

Figure 1: Application architecture

The structure of the logical architecture used for the communication between Android mobile client device and the enterprise application hosted on the JBoss application server is drawn in figure 1. The following components were identified:

- application server - it runs and publishes the web service.
- web service - JAX-RS login web service - waits for connections (POST and GET) and triggers the requested actions providing the answers (JSON formatted).
- model - module used for ORM[4] data representation.
- service - module that implements methods for managing the data persistence.
- JPA[5] - provides relational persistence of data - in this article the connection is made with the MySQL database server.
- Android - mobile device client.

[4]Object-relational mapping
[5]Java Persistence API

- Android Login activity - implemented Android application's life cycle component.
- Android WebClient - represents a client library for dealing with web requests (extends the standard AsyncTask class).

2 Main results

2.1 Prerequisites

To implement the proposed architecture the following prerequisite installation steps should be done:

1. Java installation - Java Platform (JDK) 7u45 [7].

   The environment variables $JAVA\_HOME$ and JRE_HOME must be created and concatenated to the environment variable $PATH$.

2. Eclipse installation - download and unzip (e.g. Kepler). Maven plugin will be installed by using *Eclipse marketplace* menu option.

3. Android sdk - download and setup [4].

4. Installation of the eclipse android plugin [5].

5. Apache maven - build automation tool for Java projects [10]. Note that for maven the local repository must be set otherwise the user profile folder will be used.

   To allow direct calls to the mvn tool the environment variable $maven = pathToMaven$ must be initialized and added to the path variable ($PATH = \%path\%;\%maven\%\backslash bin$).

6. JBoss installation - application server [9]. After unzip the environment variable $JBOSS\_HOME$ must be created and concatenated to the environment variable $PATH$ (e.g. jboss-eap-6.2).

In the configuration file *standalone.xml* of the server the data source must be defined. To define the data source the corresponding database driver module must be installed (e.g. MySQL driver). To create the data source with the name *java:jboss/datasources/butAndroidDS* the following code lines must be added into the standalone.xml configuration file of the application server:

```
<driver name="mysql" module="com.mysql"/>
```

into the section drivers and the following lines into the section datasources:

```
<datasource jta="true" jndi-name="java:jboss/datasources/butAndroidDS"
pool-name="my_pool" enabled="true" use-java-context="true" use-ccm="true">
    <connection-url>jdbc:mysql://localhost:3306/butAndroid</connection-url>
    <driver>mysql</driver>
    <security>
```

```
      <user-name>dbuser</user-name>
      <password>dbpassword</password>
   </security>
   <statement>
      <prepared-statement-cache-size>100</prepared-statement-cache-size>
      <share-prepared-statements>true</share-prepared-statements>
   </statement>
</datasource>
```

The database driver (MySQL) isn't installed by default such that it must be installed as module by making the following steps [11]:

1. download the database driver (MySQL connector mysql-connector-java-5.1.28.zip).
2. create the installation subdirectory $\%JBOSS\_HOME\%\backslash modules\backslash com\backslash mysql\backslash main$ and unzip into it the driver file.
3. create the module.xml file into the same subdirectory with the database connector driver.

```
<?xml version="1.0" encoding="UTF-8"?>
<module xmlns="urn:jboss:module:1.0" name="com.mysql">
   <resources>
      <resource-root path="mysql-connector-java-5.1.28-bin.jar"/>
   </resources>
   <dependencies>
      <module name="javax.api"/>
   </dependencies>
</module>
```

The module was installed successfully when at the start of the application server on the console no errors are shown.

2.2 RESTful login web service

The web services mechanism is presented using Java technologies in [2, 3].

Besides usual network connection (based on communication protocols) two types of web services were defined:

1. web services that use XML with SOAP[6] standard and
2. Representational State Transfer (RESTful) web services.

Both types of web services are accessible inside communication networks through an endpoint and are hosted on servers or devices that have hardware or software components which aren't available on the client devices.

By using a simple network connection (socket communication) the exchange of messages is made without content oriented message checks. The web services theory offers mechanisms for creating, wrapping and transmitting the messages in convenient form by using defined communication standards.

The main advantages and disadvantages of web services are presented in the table 1 and table 2.

[6]Simple Object Access Protocol - defines an XML based message architecture and the messages format

Table 1: Web services disadvantages

SOAP/WSDL	REST
• messages are complex and tools are required to generate the client stubs • while messages are XML they consume more bandwidth • complicated security rules	• the requests (especially GET method) aren't suitable for a large amount of data • some web services standards aren't addressed (Transactions, Security, Addressing, Trust, Coordination)

Table 2: Web services advantages

SOAP/WSDL	REST
• relies on XML Information Set for its message format • used over multiple transport protocols, such as HTTP, SMTP, TCP, or JMS • integrated with any programming model	• REST client are simple and can be accessed from any browser as normal URLs • transported data can be any convenient text or structure so that bandwidth optimizations can be made • application security rules can be setup using the HTTP standards (the administrator / firewall can discern the intent of each message by analyzing the HTTP command used in the request).

The login web service is a function that receives two parameters (username and password). The service checks (using JPA) into the database the rights of the user. If the user is found and the password is valid, a JSON response is created.

The login web service has the general structure of a web service [2]. Besides its scope this web service accesses the entity manager of the application server through the EJB[7] mechanism.

To create the RESTful login web service the following steps need to be done:

1. Create a main maven web project using eclipse (that has the packing type pom).

2. Create two maven modules into the main project.

```
  <groupId>jaxrs-app</groupId>
<artifactId>jaxrs-app</artifactId>
<version>WS</version>
<packaging>pom</packaging>
<name>jaxrs-app</name>
<modules>
 <module>db-ejb</module>
 <module>jaxrs-login</module>
</modules>
```

[7]Enterprise JavaBeans

3. First maven module *db-ejb*: add the dependencies into the maven *pom.xml* file (org.json for working with JSON objects, hibernate-jpa-2.0-api for database persistence, javaee-api for Java EE specification APIs, joda-time for working with dates objects). Set the main project as parent for the *db-ejb* subproject.

```
<parent>
  <groupId>jaxrs-app</groupId>
  <artifactId>jaxrs-app</artifactId>
  <version>WS</version>
</parent>
<modelVersion>4.0.0</modelVersion>
<groupId>db-ejb</groupId>
<artifactId>db-ejb</artifactId>
<version>0.0.1</version>
<packaging>ejb</packaging>
<name>db-ejb</name>
<description>db-ejb</description>
<dependencies>
  <dependency>
      <groupId>org.json</groupId>
      <artifactId>json</artifactId>
      <version>20131018</version>
  </dependency>
  <!-- the other dependencies must be inserted -->
</dependencies>
```

4. First maven module *db-ejb*: create the database model together with the JPQL queries. The class *BaseEntity* contains fields related to the create and update user and timestamps of the records.

```
@NamedQueries({
    @NamedQuery(name = User.JPQL_GET_ALL_USERS, query = "select u from User u"),
    @NamedQuery(name = User.JPQL_GET_USER_BY_USERNAME_AND_PASSWORD, query = "select n
        from User n where n.username = :username and n.password = :password")
})

@Entity
public class User extends BaseEntity implements Serializable{
   @Id
   @GeneratedValue(strategy=GenerationType.AUTO)
   private Integer id;
   @Column(unique = true)
   private String username;
   // ...
```

With the @*Entity* annotation, the connected database knows to format that class into an object and save it to the database. The connection to the valid database is established through the *persistence.xml* file where the class is mapped. The property *hibernate.hbm2ddl.auto* determines the JPA libraries to update or create the database schema.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
      http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
   <persistence-unit name="myDS">
      <jta-data-source>java:jboss/datasources/butAndroidDS</jta-data-source>
      <class>acl.but.model.User</class>
      <properties>
        <property name="hibernate.archive.autodetection" value="class" />
          <property name="hibernate.hbm2ddl.auto" value="update"/>
      </properties>
   </persistence-unit>
</persistence>
```

5. First maven module *db-ejb*: create the stateless EJB service for accessing the database. Add method for searching a user into the database based on username and password (*getUserByUsernameAndPassword*).

```
// imports skipped
@Stateless
public class UserDAO {
   @PersistenceContext
   private EntityManager em;
   @EJB
  private DAO dao;

// other methods
   @TransactionAttribute(TransactionAttributeType.SUPPORTS)
   public User getUserByUsernameAndPassword(String username, String password) {
      try {
         Query qry = em.createNamedQuery(User.JPQL_GET_USER_BY_USERNAME_AND_PASSWORD);
         qry.setParameter("username", username);
         qry.setParameter("password", password);
         return (User) qry.getSingleResult();
      } catch (NoResultException e) {
         return null;
      }
   }
}
```

6. Second maven module *jaxrs-login*: add the dependencies into the maven *pom.xml* file (org.jboss.resteasy for working with RESTful objects, db-ejb for database model, javaee-api for JEE dependencies). Set the main project as parent for the *jaxrs-login* module.

```
  <parent>
    <groupId>jaxrs-app</groupId>
    <artifactId>jaxrs-app</artifactId>
    <version>WS</version>
  </parent>
  <dependencies>
   <dependency>
      <groupId>db-ejb</groupId>
      <artifactId>db-ejb</artifactId>
      <version>0.0.1</version>
      <type>ejb</type>
   </dependency>
      <dependency>
         <groupId>org.jboss.resteasy</groupId>
         <artifactId>resteasy-jaxrs</artifactId>
         <version>3.0.6.Final</version>
      </dependency>
      <!-- the other dependencies must be inserted -->
  </dependencies>
```

7. Second maven module *jaxrs-login*: create the RESTful webservice loader *loaderRESTServices*

```
@ApplicationPath("/")
public class loaderRESTServices extends Application {
   private Set<Object> singletons = new HashSet<Object>();
   private Set<Class<?>> empty = new HashSet<Class<?>>();
   public loaderRESTServices(){
        singletons.add(new LoginRWS());
   }
   @Override
   public Set<Class<?>> getClasses() {
        return empty;
   }
   @Override
   public Set<Object> getSingletons() {
        return singletons;
   }
}
```

8. Second maven module *jaxrs-login*: create of the RESTful web service *LoginRWS* together with the GET (returns the telegram value - is associated with the path */loginRESTSrv/get* and method *getTelegram*) - when previously a successful POST (does the database search and sets the telegram result on *Success* - is associated with the path */loginRESTSrv/* and method *checkUser*) was made the telegram content is different) and POST activities.

```
@Path("/loginRESTSrv")
@Stateless
public class LoginRWS {
    @EJB
     private UserDAO userDAO;

    private Telegram telegram = new Telegram("Hello");

    @GET()
    @Path("/get")
    @Produces(MediaType.APPLICATION_JSON)
    public Telegram getTelegram() throws NamingException {
        if (telegram.getTelegram().equalsIgnoreCase("")) {
            telegram.setTelegram("Hello world!");
        }
        return telegram;
    }

    @POST
    @Consumes(MediaType.APPLICATION_FORM_URLENCODED)
    public Telegram checkUser(MultivaluedMap<String, String> appuserParams) throws NamingException {
        String username = appuserParams.getFirst("puser");
        String password = appuserParams.getFirst("ppassword");
        System.out.println("user: " + username + " " + password);
        telegram.setTelegram("Not ok!");
        // when DAO service is not ok search for it -
        Context initialContext = new InitialContext();
        String appName = (String) initialContext.lookup("java:app/AppName");
        System.out.println("AppName: " + appName);
        System.out.println("UserDAO classname: " + UserDAO.class.getSimpleName());
        String jndiUserDAO = "java:global/" + appName + "/" + UserDAO.class.getSimpleName();
        userDAO = (UserDAO)initialContext.lookup(jndiUserDAO);
        // end searching for DAO service - in normal condition not necessarly to do this search
        User user = userDAO.getUserByUsernameAndPassword(username, password);
        if (user != null) {
            String dbusername = user.getUsername();
            String dbpwdHash = user.getPassword();
            if (dbusername.equalsIgnoreCase(username) && dbpwdHash.equalsIgnoreCase(password)) {
                telegram.setTelegram("Success !");
            }
        }
        return telegram;
    }

}
```

9. Second maven module *jaxrs-login*: create the telegram - structure used for transport of the JSON data between client and web service

```
@XmlRootElement
public class Telegram {
    private String telegram;
    public Telegram() { }
    public Telegram(String str) {
        this.telegram = str;
    }
    public String getTelegram() {
        return telegram;
    }
    public void setTelegram(String telegram) {
        this.telegram = telegram;
    }
}
```

10. Deploy the web service. The generated archive (war) will be copied into the *deployments* subdirectory of the JBoss server. The application server auto-

matically detects the war files and deploys or redeploys them.

2.3 Android client application

Within the Android client application activities that manage the user interactions are created. The presented activity has two input fields (username and password). After clicking the login button an asynchronous task is created. In this task the RESTful web service is accessed. To access the web service a http web client is used. While the server path of the application is in the context of the server two calls for the login are made. First call sends the parameters and does the login checks and the second call asks using the web service the status of the login process. The JSON response of the web service is retrieved. A message is shown on the screen indicating whether the login was successful or not. The deployment of the application is made by using the parameters from the file *web.xml*. The */WEB-INF/web.xml* file is the Web Application Deployment Descriptor, an XML file document that defines application data that an application server needs to know about it (e.g. servlets, filters, listeners, initialization parameters, container-managed security constraints, resources, welcome pages, etc.). Using the *web.xml* we notify the server to publish the RESTful web service using the class *acl.but.restws.ws.loaderRESTServices*.

```
<?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"
    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>jaxrs-login</display-name>
  <context-param>
    <param-name>javax.ws.rs.Application</param-name>
    <param-value>acl.but.restws.ws.loaderRESTServices</param-value>
  </context-param>
  <listener>
    <listener-class>org.jboss.resteasy.plugins.server.servlet.ResteasyBootstrap</listener-class>
  </listener>
  <servlet>
    <servlet-name>Resteasy</servlet-name>
    <servlet-class>org.jboss.resteasy.plugins.server.servlet.HttpServletDispatcher</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>Resteasy</servlet-name>
    <url-pattern>/*</url-pattern>
  </servlet-mapping>
</web-app>
```

Steps made by the client application:

1. Create an activity and add buttons and text views to its layout

```
...
<Button
 android:text="@string/txt_login"
 android:id="@+id/btnLogin"
 android:layout_width="fill_parent"
 android:layout_height="wrap_content">
</Button>
...
```

```
...
 <EditText
 android:text=""
 android:id="@+id/txtUserName"
 android:layout_width="fill_parent"
 android:layout_height="wrap_content"
 android:inputType="text">
 </EditText>
...
```

Figure 2: Android client application response

2. Initialize the URL for accessing the login web service. The url contains the JBoss server and port, the application name (from the main module *jaxrs-login-WS*), the root path of the application that published the web service (inside the class *loaderRESTServices* was set @*ApplicationPath*("/")) plus the path declared in the header of the class that implements the service (for the class *LoginRWS* is used the path @*Path*("/*loginRESTSrv*")) plus the path declared for the method (e.g. for the method *getTelegram* was set the path @*Path*("/*get*")). So that the url for accessing the web service is $http://10.22.01.78:8080/jaxrs-login-WS/loginRESTSrv/get$ for the *getTelegram* method.

```
private static final String SERVICE_URL = "http://10.22.01.78:8080/jaxrs-login-WS/loginRESTSrv";
```

3. Implement the login activity and the onclick listener method

```
public OnClickListener ocl = new OnClickListener() {
    @Override
    public void onClick(View v) {
        switch (v.getId()) {
        case R.id.btnLogin:
            System.out.println("login button clicked");
            String username = txtUserName.getText().toString();
            String passwd = txtPassword.getText().toString();
```

```
            passwd = Hashing.md5(passwd);
            callDoLogin(username, passwd);
            break;
        case R.id.btnCancel:
            String string = "";
            txtUserName.setText(string, TextView.BufferType.EDITABLE);
            txtPassword.setText(string, TextView.BufferType.EDITABLE);
            break;
        }
    }
};
```

4. Implement *callDoLogin* method. It can be seen that there are two calls made (one is the login itself and the second is the reading of the login status from the application on the server context).

```
RestClient restclient = new RestClient(RestClient.POST_TASK, this);
restclient.addNameValuePair("puser",username);
restclient.addNameValuePair("ppassword", Hashing.md5(passwd));
restclient.execute(new String[] { SERVICE_URL });

String answerUrl = SERVICE_URL + "/get";
RestClient wstg = new RestClient(RestClient.GET_TASK, this);
wstg.execute(new String[] { answerUrl });
```

5. Implement RestClient class for dealing with the http connections. The Android client must be enabled to send and receive web requests to the RESTful web services. An extended version for the class *RestClient* is presented in [6].

```
private class RestClient extends AsyncTask<String, Integer, String> {

    public static final int POST_TASK = 1;
    public static final int GET_TASK = 2;
    private static final String TAG = "RestClient";
    private static final int CONN_TIMEOUT = 3000;
    private static final int SOCKET_TIMEOUT = 5000;
    private int taskType = GET_TASK;
    private Context context = null;
    private ArrayList<NameValuePair> params = new ArrayList<NameValuePair>();
    public RestClient(int taskType, Context context) {
        this.taskType = taskType;
        this.context = context;
    }
    public void addNameValuePair(String name, String value) {
        params.add(new BasicNameValuePair(name, value));
    }
    private HttpParams getHttpParams() {
        HttpParams htpp = new BasicHttpParams();
        HttpConnectionParams.setConnectionTimeout(htpp, CONN_TIMEOUT);
        HttpConnectionParams.setSoTimeout(htpp, SOCKET_TIMEOUT);
        return htpp;
    }
    private HttpResponse doResponse(String url) {
        HttpClient httpclient = new DefaultHttpClient(new BasicHttpParams());
        HttpResponse response = null;
        try {
            switch (taskType) {
                case POST_TASK:
                    HttpPost httppost = new HttpPost(url);
                    httppost.setEntity(new UrlEncodedFormEntity(params));
                    response = httpclient.execute(httppost);
                    break;
                case GET_TASK:
                    HttpGet httpget = new HttpGet(url);
                    response = httpclient.execute(httpget);
                    break;
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
        return response;
    }
    private String inputStreamToString(InputStream is) {
        String line = "";
```

```
        StringBuilder total = new StringBuilder();
        BufferedReader br = new BufferedReader(new InputStreamReader(is));
        try {
            while ((line = br.readLine()) != null) {total.append(line);}
        } catch (Exception e) {e.printStackTrace();}
        return total.toString();
    }
    @Override
    protected String doInBackground(String... urls) {
        String result = "";
        HttpResponse response = doResponse(urls[0]);
        if (response == null) {
            return result;
        } else {
            try {result = inputStreamToString(response.getEntity().getContent());
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
        return result;
    }
    @Override
    protected void onPostExecute(String response) {
        System.out.println("onPostExecute: " + response);
        handleResponse(response);
    }
}
```

6. Implement handleResponse method for parallel showing of the results.

```
public void handleResponse(String response) {
    try {
        JSONObject jso;
        jso = new JSONObject(response);
        responseWS = response;
        System.out.println("handleResponse:" + responseWS);
        loginHandler.post(createUI);
    } catch (JSONException e) {}
}
```

7. The UI result are displayed using a Runnable object

```
final Runnable createUI = new Runnable() {
    public void run() {
        try {
            JSONObject jsonObj = new JSONObject(responseWS);
            String res = "responseWS: " + jsonObj.toString();
            System.out.println(res);
            Toast.makeText(LoginActivity.this, res, Toast.LENGTH_LONG).show();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
};
```

2.4 Remarks

- JBoss server should be started with the option *standalone -b 0.0.0.0* such that the server listens and responds to requests on all network interfaces. The web service is published when the deployment is made and it should be accessible from the Android client application through the network connection.

- The method $md5(String)$ from the class $Hashing$ is used by the Android client application to compute the hash for the given password before sending it through the network to the web service (as POST request parameter).

```
public static String md5(String s)
 {
     MessageDigest digest;
     try
     {
         digest = MessageDigest.getInstance("MD5");
         digest.update(s.getBytes(),0,s.length());
         String hash = new BigInteger(1, digest.digest()).toString(16);
         return hash;
     }
     catch (NoSuchAlgorithmException e) {e.printStackTrace();}
     return "NOHASH";
 }
```

- The class *RestClient* is defined inside the class *LoginActivity* and directly accesses the *handleResponse* method (from *LoginActivity*). For the extended version of the application it is recommended to put the class *RestClient* in a utility package.

- *persistence.xml* must be placed into the web project into the subdirectory $src\backslash main\backslash resources\backslash META-INF$

- For creating the war files using maven command *mvn clean install* is used.

- The Android client application needs to access Internet so that the corresponding permission must be granted

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

3 Conclusion

The web services permit the easy defining of the service oriented architectures. The services and their modularity is exploited by the mobile applications. The mobile clients have access to complex functionalities with less resources. By using the enterprise architecture presented in this article at least two big advantages of the service oriented architecture are envisaged:

- complex modules that support the web services that cannot be run on the mobile devices are accessed and used;

- functionalities are made available for mobile devices allowing the enterprise processes to be accessible using the mobile technologies without need of the modules re-factoring.

The login RESTful web service architecture can be extended and used for publishing all the functionalities of the enterprise application. The modules that are behind the system functionalities can be wrapped in proper RESTful web service calls. The web services are run in the application server context and can be scaled using PaaS[8] clouds with less programming effort [12].

[8]Platform as a service - a service model of cloud computing

References

[1] Aldea, C., L., *Elemente de securitate în reţele de calculatoare*, *Transilvania* University Publishing House, Braşov, 2010.

[2] Aldea, C., Sangeorzan, L., Aldea, A. (2009, September). *Web services and enterprise games*, In I. Rudas, N. Mastorakis (Eds.), WSEAS International Conference. Proceedings. Mathematics and Computers in Science and Engineering (No. 5). WSEAS.

[3] Aldea, C., L., (2013, September). *JAX-WS login web service and Android client*, In Bulletin of the Transilvania University of Brasov Series III **6(55)** (2013), 51-60.

[4] https://developer.android.com/sdk/index.html

[5] http://developer.android.com/sdk/installing/installing-adt.html

[6] http://avilyne.com/?p=105

[7] http://www.oracle.com/technetwork/java/javase/downloads/index.html

[8] http://java.dzone.com/articles/jax-ws-hello-world

[9] http://www.jboss.org/jbossas/downloads/

[10] http://maven.apache.org/download.cgi

[11] https://zorq.net/b/2011/07/12/adding-a-mysql-datasource-to-jboss-as-7/

[12] https://www.openshift.com/