(BSF4)ooRexx and Java Web Servers

Rony G. Flatscher (Rony.Flatscher@wu.ac.at), WU Vienna
"The 2020 International Rexx Symposium", Web-based Conference
September 29th – October 1st , 2020

Abstract. BSF4ooRexx is a bi-directional Java bridge for the dynamic scripting language ooRexx which implements the Java scripting framework and can therefore be employed as a scripting language in many Java contexts. Java based web servers like Apache Tomcat implement the Java2EE/JakartaEE JSP (Java server pages) specifications which allow among other things to extend the JSP's functionality by providing tag libraries ("taglibs"). This article introduces such a taglib ("ScriptTagLibs") that allows any Java scripting language – in this article ooRexx [12] – to be employed for servlet programming and creating HTML content that gets returned as a result of a client's request and how to deploy it on Java based web servers in form of a war ("web application archive") file.

1 Introduction

Java web servers have been defined in the form of standardized services that constitute the package "J2EE" ("Java enterprise edition") which has been handed over to the Eclipse foundation by Oracle in the fall of 2017. Eclipse continues to maintain and develop the specifications under the new name "Jakarta EE" ("Jakarta enterprise edition") [1] [1]. The implementation of the standardized Jakarta EE services by different companies and organisations should allow the creators of web applications to deploy them using the "web application archive" ("war") format [2] unchanged on the different Java web servers.

The Apache Software Foundation (ASF, [3]) Tomcat Java web server project[2] [4] maintains a list of links to the Java/Jakarta specifications at [5].

There is a fundamentally important specification named "Java/Jakarta Servlet"

[1] The Wikipedia article refers to the Oracle trademark for "Java" which Oracle uses to force Eclipse and others to use another term than Java, in the case of web servers (web containers) Eclipse uses the term "Jakarta" instead. This trademark issue also forced the top-level package name to be changed from `javax.` to `jakarta.` instead by Eclipse. It is to be expected that for a few years this naming issue will cause confusion and difficulties during the transition period. In this article "Java/Jakarta" will indicate this transitional name change from "Java EE" to "Jakarta EE" in the context of web server software.

[2] Apache Tomcat will be used in the context of this article as a proof-of-concept Java based web server for creating web applications using Rexx. Tomcat versions 9 or earlier are using the original "javax." package name in its implementations of the standardized services for servlet, JSP and TagLib implementations, whereas starting with Tomcat version 10 the new package name "jakarta." gets used instead.

which defines how client requests supplying an URL get mapped on the web server to Java programs serving the request and how such Java programs need to be structured.

Another important standardized service in this context is called "Java/Jakarta Server Pages (JSP) and Expression Language (EL)" which allows mixing HTML/XML text with Java code that will create additional HTML/XML text to be returned to the requesting client. In addition this specification allows for creating and implementing so called "tag libraries" in Java to extend this functionality, a feature that gets exploited in the "*ScriptTagLibs*"[3] developed by the author in this context in order to allow Java script languages to be deployable within JSP pages. As BSF4ooRexx implements a Java script engine for ooRexx [13], one can use ooRexx to implement the logic at the web server side.

This article will employ the open-source Apache Tomcat Java-based web server [4] and describes the directory layout of the installation (home) directory of the Tomcat 9 and 10 servers at the time of writing. The directory named "webapps" is used to deploy web applications, that themselves need to follow a specific subdirectory layout which will get briefly described.

Then a simple Java servlet will get demonstrated, which implements all logic in a single Java class, thereby introducing the important configuration file "*WEB-INF/web.xml*" for allowing the Tomcat server to determine which Java class needs to be loaded and executed depending on the web client request. This very Java servlet example will then be implemented in form of a "Java Server Page (JSP)" which in essence is an HTML/XML text file that contains the markup for the client's browser, interspersed with Java code snippets. This JSP file gets turned into a Java class that contains the interspersed Java code snippets together with Java output statements that create the markup found in the JSP file which then gets compiled on the fly. Whenever a JSP file's content gets changed, the Java based web server will reprocess the JSP file and recompile the resulting Java class and use that version from then on.

The article then introduces the "*ScriptTagLibs*" taglib library, developed for the purpose of allowing any Java based scripting language to be used in JSPs instead of

[3] At the time of writing the "ScriptTagLibs" are offered as `"javax.ScriptTagLibs.jar"` and `"jakarta.ScriptTagLibs.jar"` depending on the namespace used in the hosting Java web server. Each jar file includes two tag libraries, one for running scripts using Apache BSF ("Bean Scripting Framework"), one for running scripts using the newer standard Java scripting framework in the package "javax.script" (a.k.a. "JSR-223") introduced with Java 6.

Java, empowering anyone who knows, e.g., Rexx to create Web server applications with JSPs that contain Rexx code instead of Java code!

2 The Apache Tomcat Java Based Web Server

The free and open-source Apache Tomcat Java based web server is one of the best maintained, very powerful, yet easy to install, to configure, to deploy and to use Java based web servers.

2.1 A Brief Overview of the Tomcat Directory Layout

The Apache Tomcat Java based servers have a simple, easy to use directory structure they create upon installation. At the time of this writing the Tomcat 9 and 10 Java based web servers create the following directories in their installation (home) directory[4]:

- `bin`: this directory includes all scripts and binaries (jar – Java archive files) needed to control the Tomcat server, including scripts to start and stop the Tomcat server from the command line.
- `conf`: this directory contains all of the Tomcat global configurations, mostly in form of XML encoded text files. E.g., the file `server.xml` defines the server's characteristics including the TCP/IP port[5] at which the server listens for client requests. Another important file is `tomcat-users.xml` which allows for defining Tomcat users with their passwords and roles, e.g., for accessing the Tomcat manager console.
- `lib`: this directory contains all *jar* (Java archives) files that should be shared among all web applications.
- `logs`: this directory contains all log files that Tomcat creates at runtime. E.g., output to *stdout* (standard output file) or to *stderr* (standard error file) will be logged and stored in the respective text files in this directory.
- `temp`: this directory is Tomcat's temporary directory.
- `webapps`: this directory serves as the home directory for any web application

[4] The Tomcat installation refers to its installation (home) directory with the shell variable named `CATALINA_HOME`.

[5] In this article the Tomcat server is configured to listen to port *8080*. While developing and testing web applications at the server itself, the server name will be "*localhost*", such that all the URLs in this article are always led in by "*localhost:8080/*".

that gets deployed. Tomcat creates the three subdirectories

  - `docs`: serves the Tomcat documentation.
  - `manager`: supplies the Tomcat manager console.
  - `ROOT`: serves as the default Apache Tomcat interface, with links to Tomcat documentation, Tomcat wiki, Tomcat mailing lists and the like.
- `work`: this directory gets usually used by Tomcat whenever a JSP file needs to be turned into a compiled Java class.

Anyone who wishes to add a new web application needs to create a subdirectory in the *webapps* directory. A web application subdirectory may contain the subdirectory *WEB-INF* with a configuration file named *web.xml* that tells the Tomcat server among other things how to map a specific client request to the appropriate web application service.[6]

A web application can be distributed in form of a *war* (web application archive) file which can be deployed on any Java based web server adhering to the Java based web server standards [5] by simply copying it to the *webapps* subdirectory.[7] The Java based web server will deploy it by creating a subdirectory matching the base name of the *war* file and extracting all of its contents into it. Deleting the *war* file from the *webapps* directory will remove (undeploy) the web application subdirectory from the web server.

2.2 Serving a Client Request with a Java Servlet

In this section a web application will use a single Java program to create the HTML page that gets sent back to the client. The web application directory *rexxla_01_Servlet_Java* will contain the following files:

- `rexxla_01_Servlet_Java/index.html`: as this file is named *index.html* it will be returned to the client by the web server by default. Figure 1 shows the content, Figure 2 shows how it gets rendered in the client's browser.
- `rexxla_01_Servlet_Java/src/ServletJava.java`: the source code of the Java class that creates the HTML text that the client receives from the server.

[6] A web application may omit the *WEB-INF* subdirectory in which case one of the following files should be available: *index.htm*, *index.html* or *index.jsp*.

[7] To create a web application *war* file one merely needs to change into the web application subdirectory and create a zip archive of all files contained in it but use the file extension *war* instead of *zip*.

```
<html>
    <head>
        <title>Servlet (index.html)</title>
    </head>
    <body>
        Please click <a href="runServlet">this link</a> to run the servlet.
    </body>
</html>
```

Figure 1: Content of "rexxla_01_Servlet_Java/index.html".

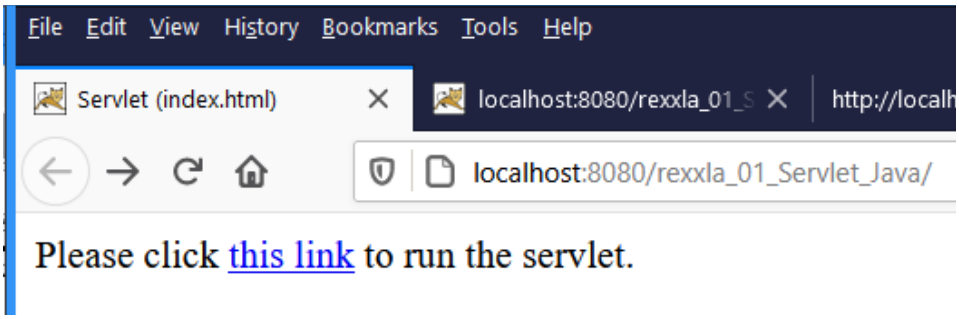

Figure 2: Result of client requesting the web application named "rexxla_01_Servlet_Java".

Figure 3 depicts the Java code, Figure 4 displays the text produced by the Java servlet and Figure 5 shows how this text gets formatted in the client's browser.

```
package org.rexxla;     // define package name for this class
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
// Minimal Java servlet, implements the specific part for HTML body; HttpServlet
// will create, inject and complete HTML to return
public class ServletJava extends HttpServlet {
   public void doGet(HttpServletRequest request, HttpServletResponse response)
     throws ServletException, IOException {
     response.setContentType("text/html");   // set content type
     PrintWriter out = response.getWriter(); // get PrintWriter
     String crlf="\015\012";                 // HTML must use CR-LF as newline
     out.println("<h1>Hello, world (Java Servlet)</h1>" + crlf);
     out.println("<p>This servlet was executed, because of the following URL:" +
                 "<ul><li>URL <em>request.getRequestURL()</em>: <br><code>"+
                 request.getRequestURL()+"</code>" + crlf +   // URI
                 "<li>its URI being <em>request.getRequestURI()</em>: <br><code>"+
                 request.getRequestURI()+"</code></ul></p>"); // URL
   }
}
```

Figure 3: Content of "rexxla_01_Servlet_Java/src/ServletJava.java".

```
<h1>Hello, world (Java Servlet)</h1>

<p>This servlet was executed, because of the following URL:<ul>
<li>URL <em>request.getRequestURL()</em>:
<br><code>http://localhost:8080/rexxla_01_Servlet_Java
/runServlet</code>
<li>its URI being <em>request.getRequestURI()</em>:
<br><code>/rexxla_01_Servlet_Java/runServlet</code></ul></p>

```

Figure 4: The HTML text created by the Java servlet.

Figure 5: Response to client which was created by the Java servlet.

- `rexxla_01_Servlet_Java/WEB-INF/web.xml`: this file gets used to configure this web application, its content is displayed in Figure 6 below. In the *servlet* element it defines a logical name "*ServletJavaHelloWorld*" for the servlet and the fully qualified name of the compiled Java class "*org.rexxla.ServletJava*" that should be executed. The *servlet-mapping* element maps the client request (in this case "*/runServlet*") to the logical name "*ServletJavaHelloWorld*" that will get executed and create the HTML text to be returned to the client.
- `rexxla_01_Servlet_Java/WEB-INF/classes/org/rexxla/ServletJava.class`: the compiled Java class that the server will run and which then creates the HTML text that the client receives from the server.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
                   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                          xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd"
                   version="4.0"
                   metadata-complete="true">
  <display-name>Minimal org.rexxla.ServletJava Demo</display-name>
  <description>
     Welcome to a minimal ServletJava demo
  </description>
  <!-- 'servlet'-element must precede the 'servlet-mapping'-element ! -->
     <servlet>
     <servlet-name>ServletJavaHelloWorld</servlet-name>
     <servlet-class>org.rexxla.ServletJava</servlet-class>
    </servlet>
     <!-- 'servlet-mapping'-element must follow the 'servlet'-element ! -->
    <servlet-mapping>
       <servlet-name>ServletJavaHelloWorld</servlet-name>
       <url-pattern>/runServlet</url-pattern>
    </servlet-mapping>
</web-app>
```

Figure 6: Content of "rexxla_01_Servlet_Java/WEB-INF/web.xml".

The URL gets created using the web server's name "*localhost*", explicitly denoting the port *8080*, the web application's name "*rexxla_01_Servlet_Java*" and the *url-pattern* "*/runServlet*" concatenated with each other to form the URL: "*localhost:8080/rexxla_01_Servlet_Java/runServlet*". The link to the very same Java servlet in the web application's index file "*index.html*" in Figure 1 can be simply denoted as "*runServlet*" which then will be resolved by concatenating it with a slash as a delimiter to the document's base URL "*localhost:8080/rexxla_01_Servlet_Java*".

2.3 Serving a Client Request with a Java Server Page (JSP)

In this section a web application will use a Java server page (JSP) to define the HTML page that gets sent back to the client. The very first time a new or changed JSP file gets requested, the server will create a Java class for it, compile it and run it, which will create the HTML text to be sent back to the requesting client. Successive requests for that same JSP file will cause the immediate execution of the compiled Java class. The web application directory *rexxla_02_JSP* will contain the following files:

- `rexxla_02_JSP/index.html`: as this file is named *index.html* it will be

```
<html>
    <head>
        <title>JSP (index.html)</title>
    </head>
    <body>
        Please click <a href="helloWorld.jsp">helloWorld.jsp</a> to run the JSP.
    </body>
</html>
```

Figure 7: Content of "rexxla_02_JSP/index.html".

returned to the client by the web server by default. Figure 7 shows the content, Figure 8 shows how it gets rendered in the client's browser.

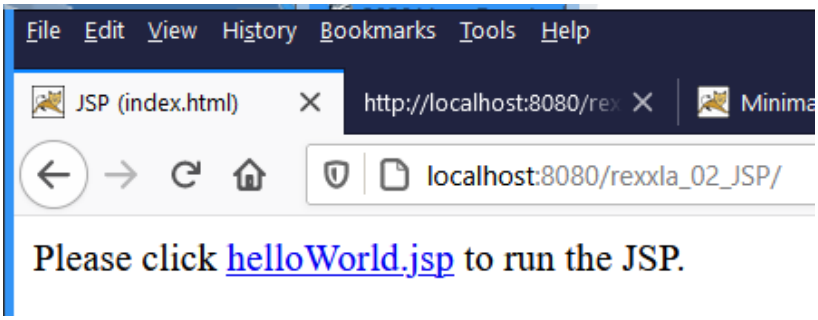

Figure 8: Result of client requesting the web application named "rexxla_02_JSP".

- `rexxla_02_JSP/helloWorld.jsp`: this JSP file includes the same Java statements as the Java servlet in Figure 5 above interspersed with the HTML text. Figure 9 shows the content, Figure 10 shows the result of running this

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Minimal JSP</title>
</head>
<body>
<%
String crlf="\015\012";                    // HTML must use CR-LF as newline
out.println("<h1>Hello, world (JSP)</h1>" + crlf);
out.println("<p>This JSP was executed, because of the following URL:" +
          "<ul><li>URL <em>request.getRequestURL()</em>: <br><code>"+
          request.getRequestURL()+"</code>" + crlf +   // URL
          "<li>its URI being <em>request.getRequestURI()</em>: <br><code>"+
          request.getRequestURI()+"</code></ul></p>"); // URI
%>
</body>
</html>
```

Figure 9: Content of "rexxla_02_JSP/helloWorld.jsp".

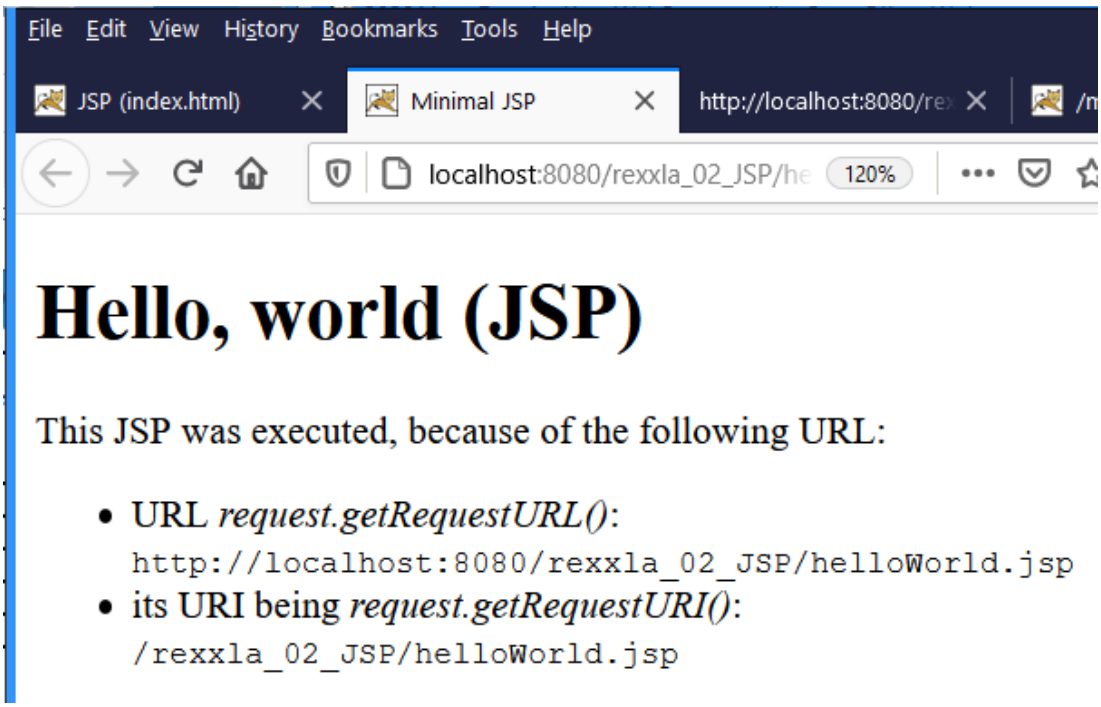

Figure 10: Response to client which was created by the JSP.

JSP in the client's browser.

The sections in Figure 9 with a yellow background highlight the JSP related directive. As can be seen the character sequence "<%" leads-in and "%>" ends a JSP directive. Any text outside of these JSP directives represents plain HTML text that will be sent to the client without any changes. The second JSP section contains the Java code[8] that will create and insert additional

```
1
2  <!DOCTYPE html>
3  <html>
4  <head>
5  <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
6  <title>Minimal JSP</title>
7  </head>
8  <body>
9  <h1>Hello, world (JSP)</h1>
10
11 <p>This JSP was executed, because of the following URL:<ul><li>URL
   <em>request.getRequestURL()</em>: <br><code>http://localhost:8080
   /rexxla_02_JSP/helloWorld.jsp</code>
12 <li>its URI being <em>request.getRequestURI()</em>:
   <br><code>/rexxla_02_JSP/helloWorld.jsp</code></ul></p>
13
14 </body>
15 </html>
```

-Figure 11: The HTML text created by the JSP.

[8] The JSP Java generator will make sure that the following nine variables (referencing the "JSP implicit objects") are always available: `application`, `config`, `exception`, `out` (an instance of `{javax|jakarta}.servlet.jsp.JspWriter`), `page`, `pageContext`, `request`, `response`, `session`.

HTML text at that location. The resulting HTML text that the requesting client receives from this JSP is shown in Figure 11 above.

- `rexxla_02_JSP/WEB-INF/web.xml`: this file gets depicted in Figure 12 and is not needed for this simple web application. However, its content will be used to describe this web application in the Tomcat manager interface.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
                  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                        xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd"
                  version="4.0"
                  metadata-complete="true">
  <display-name>Minimal JSP-Demo</display-name>
  <description>
     Welcome to a minimal JSP demo
  </description>
</web-app>
```

Figure 12: Content of "rexxla_02_JSP/WEB-INF/web.xml".

2.4 The "ScriptTagLibs" Taglib Library and (BSF4)ooRexx

Java based web servers allow the usage of taglib libraries within JSP pages. A taglib library contains tag handlers that implement the *BodyTag*[9] interface class making it possible to extend the valid tags of a JSP with custom implementations. Tag libraries need to come with a matching TLD (tag library description) text file, which defines the tag names with their possible attributes.

Early on in the ASF there were two taglib libraries created to allow "scriptlets" and "expressions" to be used in JSP pages that allowed for implementing the code in one of the Apache BSF[10] [6] scripting languages. These taglib libraries got deprecated over time, but can still be fetched from [8]. As such they served as the starting point for implementing the "*ScriptTagLibs*" libraries [9] in the fall of 2020

[9] Depending on the targeted Java based web server implementation, the fully qualified name can be either `javax.servlet.jsp.tagext.BodyTag` or `jakarta.servlet.jsp.tagext.BodyTag` (Tomcat 9 or earlier use the *`javax.`* top level package name, starting with Tomcat 10 the *`jakarta.`* top level package name gets used, cf. the "History" section in [1]).

[10] Interestingly, BSF was originally created as an open-source project at IBM for allowing scripting languages to be deployable in Java server pages (JSP)! The code was later donated to ASF. With Java 6 a proper Java scripting framework (package `javax.script` also dubbed "JSR-223" which defines the specifcation, cf. [7]) got introduced into the language, which over time made BSF less relevant. Yet, Apache BSF is still being used in some Java based applications and there may be scripting language implementations for Java that support BSF only.

and which get applied and demonstrated as a proof of concept in this article.

There are two taglib libraries, *javax*.ScriptTagLibs.jar for Tomcat 9 or earlier and *jakarta*.ScriptTagLibs.jar for Tomcat 10, which implement the tags *script* and *expr* identically.

Each tag library comes with two versions, one for deploying script code that gets executed by Apache BSF using the TLD file named `script-bsf.tld` and one where the script code gets executed by the newer and standard Java scripting framework ("JSR-223") named `script-jsr223.tld`.[11] The *script* and *expr* tags are implemented for both, BSF and JSR-223. Note, however, that JSR-223 has additional attributes due to additional features available in the newer and standard Java scripting framework, notably the ability to compile and execute compiled scripts.

Table 1 gives a brief overview of the attributes[12] that can be supplied to the *script* and *expr* tags in JSP.

	Tag Name		
Attribute Name	***script***	***expr***	Comment
type	Must	Must	Scripting language name, e.g., `"rexx"`. *Hint*: In JSR-223 it is possible to supply alternatively a mime-type or file-extension .
arguments	Optional	Optional	Defaults to `"true"`: supply the implicit objects `"request"`, `"response"` and `"out"` as arguments (in the listed order) to the script.
cacheSrc	Optional	Optional	Defaults to `"true"`. Will read the script from the external file once and then reuse the cache. *Advice*: while developing the script logic, set this attribute to `"false"` in order to force always reading from the file system in order to reflect any code changes in the external script file.
compile	Optional	Optional	*JSR-223 only*, defaults to `"true"`. Upon first execution compile the script and execute the compiled version.
debug	Optional	Optional	Defaults to `"false"`: injects debug information at the JSP directive's location in the JSP file.
name	Optional	Optional	Any string, to ease locating exceptions (for debugging).
reflect	Optional	Optional	Defaults to `"false"`: causes an implicit HashMap object

[11] The TLD files get distributed with the *ScriptTagLibs.jar* files and can be found in the jar's *META-INF* directory.

[12] The current values of these attributes can be fetched from the invoked script by accessing the implicit object named *"ScriptTaglibs.Attributes"*, which is a *java.util.Map*.

If using the BSF taglib (*script-bsf.tld*), then the ooRexx programmer may use the function *bsf.lookup(name)*, if using the JSR-223 taglib (*script-jsr223.tld*) then the *javax.script.ScriptContext.getAttribute(name)* method to get access to implicitly registered objects.

	Tag Name		
Attribute Name	***script***	***expr***	Comment
			named "scriptTagLibInfos" to be created that contains taglib and namespace related information as well as the current values of all attributes.
slot	Optional	Optional	Any string that a JSP/script developer can use for any purpose (can be fetched from the invoked script).
src	Optional	Optional	URL of external script code, must reside in the web application directory.
throwException	Optional	Optional	Defaults to "false": controls whether the execution of the JSP page should continue in the case that an exception gets thrown in a script.

Table 1: ScriptTagLibs (Tags and their Attributes).

The web application *rexxla_03_ScriptTagLib* demonstrates usages of the *script* tag from the *ScriptTagLibs* Java archives:

- a simple JSP containing Rexx code in a *script* tag to create part of the HTML text to be sent to the requesting client (`helloWorld-bsf-01.jsp`, `helloWorld-jsr223-01.jsp`),
- a simple JSP using a *script* tag with the *src* attribute to point to the Rexx script stored in the external file named `helloWorld.rex` in the web application's directory which will get loaded and executed (`helloWorld-bsf-02.jsp`, `helloWorld-jsr223-02.jsp`).

  Note: due to the supplying of the implicit objects "`request`", "`response`" and "`out`" as arguments to the invoked scripts by the implementation of the *ScriptTagLibs* taglib library one can regard such scripts to implement the `javax.servlet.Servlet` interface method `service(ServletRequest request, ServletResponse response)`!

The JSP files containing "*bsf*" in their name will use Apache BSF and those with "*jsr223*" will use the newer standard Java scripting framework (JSR-223) to execute the Rexx scripts in the JSP pages. In order to run these examples one needs to install the latest version of *ooRexx 5* (at the time of this writing in beta) from [10] together with the latest version of the ooRexx-Java bridge *BSF4ooRexx* from [11], which supports both, Apache BSF and the newer standard Java scripting framework (JSR-223).

The `bsf4ooRexx-v641-20210207-bin.jar` (or newer) from *BSF4ooRexx* and the *ScriptTagLibs* Java archives `javax.ScriptTagLibs.jar` (Tomcat 9 or earlier) or

`jakarta.ScriptTagLibs.jar` (Tomcat 10 or later) can be copied to Tomcat's home/installation directory ("*CATALINA_HOME*") and there into the subdirectory named *lib*, which causes Tomcat to make them available to all web applications. Alternatively these Java archives can be copied to the web application's `WEB-INF/lib` directory which is searched by Tomcat before Tomcat's *lib* directory in its home/installation directory.

The file `script-bsf.tld` allows one to use BSF to execute the scripts contained in a JSP page, the file `script-jsr223.tld` allows one to use the newer and standard Java scripting framework (JSR-223) to execute the scripts, which is advised. Both files are included in the `ScriptTestLibs.jar` file in the subdirectory named `META-INF`.

The web application named *rexxla_03_ScriptTagLib* demonstrates how to use Apache BSF and execute scripts interspersed in a JSP and an external script, and do exactly the same with the newer standard Java scripting framework (JSR-223). As the reader can see (cf. Figures 15 and 18), the only difference between these two versions of the JSP files is which taglib gets referred to by using the appropriate *uri* value in the JSP directive named *taglib* at the very top of the JSP files.

The web application directory *rexxla_03_ScriptTagLib* will contain the following files:

- `rexxla_03_ScriptTagLib/index.html`: as this file is named *index.html* it will

```
<html>
    <head>
        <title>JSP ScriptTagLibs (index.html)</title>
    </head>
    <body>
        <p>Please click on any of the following minimal Rexx programs:
        <ul>
            <li>Employing the Apache Bean Scripting Framework (BSF):
            <ul>
                <li><a href="helloWorld-bsf-01.jsp">helloWorld-bsf-01.jsp</a>
                <li><a href="helloWorld-bsf-02.jsp">helloWorld-bsf-02.jsp</a>
                    (employing Rexx program <a href="helloWorld.rex">helloWorld.rex</a>)
            </ul>
            <li>Employing the Java Scripting Framework (a.k.a. &quot;jsr-223&quot;):
            <ul>
                <li><a href="helloWorld-jsr223-01.jsp">helloWorld-jsr223-01.jsp</a>
                <li><a href="helloWorld-jsr223-02.jsp">helloWorld-jsr223-02.jsp</a>
                    (employing Rexx program <a href="helloWorld.rex">helloWorld.rex</a>)
            </ul>
        </ul>
    </body>
</html>
```

Figure 13: Content of "rexxla_03_ScriptTagLib/index.html".

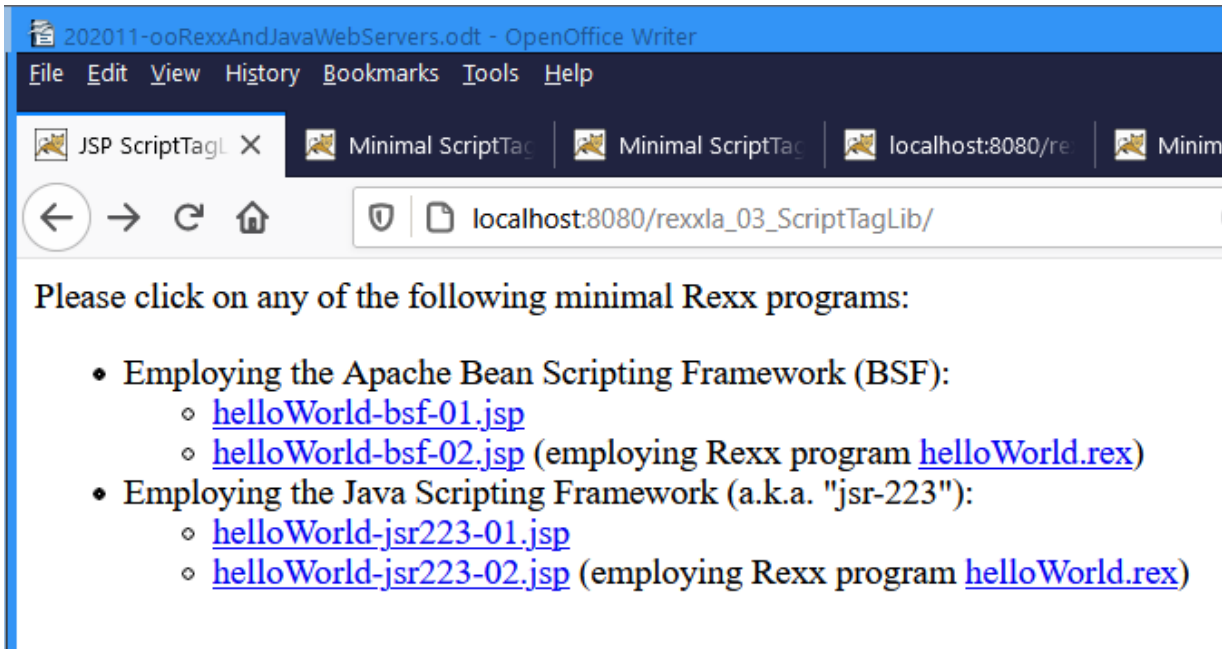

Figure 14: Result of client requesting the web application "rexxla_03_ScriptTagLib".

be returned to the client by the web server by default. It makes the JSPs `helloWorld-bsf-01.jsp`, `helloWorld-bsf-02.jsp`, `helloWorld-jsr223-01.jsp`, and `helloWorld-bsf-02.jsp` available via links and also allows the external Rexx program in `helloWorld.rex` to be displayed directly in a browser by supplying an appropriate link. Figure 13 shows the file's content, Figure 14 shows how it gets rendered in the client's browser.

`rexxla_03_ScriptTagLib/helloWorld-bsf-01.jsp`: this JSP file includes Rexx code in a *script* element which gets executed via BSF. Its content is shown in Figure 15 below. At the top of that JSP page the JSP directive "`<%@ taglib`" (first yellow backround section) defines in its *uri* attribute that BSF should be used ("*http://rexxla.org/taglibs/**bsf***") in the web application and the prefix "*s*" to denote the namespace for it causing the *script* tag to be written as "`<s:script>`" in the JSP.

The second section with yellow background in Figure 15 highlights the *script* element with the *type* attribute having the value "`rexx`" declaring the scripting language Rexx to be used for executing the script's code which creates some HTML text using information from the "`request`" argument. Figure 16 shows the client's browser formatting the returned HTML text which gets depicted in Figure 17.

```
<%@ page session="false" pageEncoding="ISO-8859-1" contentType="text/html;
charset=ISO-8859-1" %>
<%@ taglib uri="http://rexxla.org/taglibs/bsf" prefix="s" %>
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Minimal ScriptTagLibs-JSP</title>
</head>
<body>
<s:script type="rexx">
use arg request
say "<h1>Hello, world (ScriptTagLibs JSP)</h1>"
say "<p>This JSP was executed, because of the following URL:"
say "<ul><li>URL <em>request~getRequestURL()</em>: <br>"
say "<code>"request~getRequestURL~toString"</code>"
say "<li>its URI being <em>request~getRequestURI()</em>: <br>"
say "<code>"request~getRequestURI"</code></ul></p>"
</s:script>
</body>
</html>
```

Figure 15: Content of "rexxla_03_ScriptTagLib/helloWorld-bsf-01.jsp".

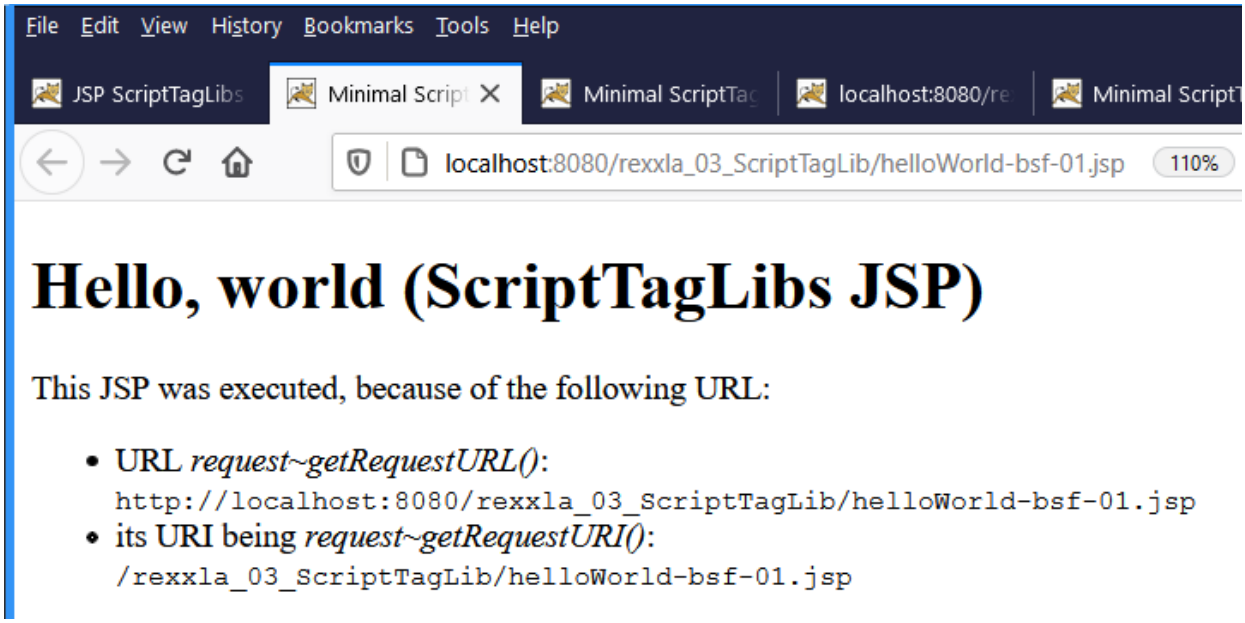

Figure 16: Response to client which was created by "helloWorld-bsf-01.jsp".

```
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Minimal ScriptTagLibs-JSP</title>
</head>
<body>
<h1>Hello, world (ScriptTagLibs JSP)</h1>
<p>This JSP was executed, because of the following URL:
<ul><li>URL <em>request~getRequestURL()</em>: <br>
<code>http://localhost:8080/rexxla_03_ScriptTagLib/helloWorld-bsf-01.jsp</code>
<li>its URI being <em>request~getRequestURI()</em>: <br>
<code>/rexxla_03_ScriptTagLib/helloWorld-bsf-01.jsp</code></ul></p>

</body>
</html>

```

Figure 17: HTML text created and returned by "helloWorld-bsf-01.jsp".

- `rexxla_03_ScriptTagLib/helloWorld-jsr223-01.jsp`: this JSP file includes Rexx code in a *script* element which gets executed via the newer standard Java scripting framework (JSR-223). Its content gets depicted in Figure 18 below, the client's browser presenting the returned HTML text is shown in Figure 19 below.

  Comparing the JSP text (cf. Figure 18) with the one for `helloWorld-bsf-01.jsp` (cf. Figure 15 above) it appears to be identical. The only difference is

```
<%@ page session="false" pageEncoding="ISO-8859-1" contentType="text/html;
charset=ISO-8859-1" %>
<%@ taglib uri="http://rexxla.org/taglibs/jsr223" prefix="s" %>
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Minimal ScriptTagLibs-JSP</title>
</head>
<body>
<s:script type="rexx">
use arg request
say "<h1>Hello, world (ScriptTagLibs JSP)</h1>"
say "<p>This JSP was executed, because of the following URL:"
say "<ul><li>URL <em>request~getRequestURL()</em>: <br>"
say "<code>"request~getRequestURL~toString"</code>"
say "<li>its URI being <em>request~getRequestURI()</em>: <br>"
say "<code>"request~getRequestURI"</code></ul></p>"
</s:script>
</body>
</html>
```

Figure 18: Content of "rexxla_03_ScriptTagLib/helloWorld-jsr223-01.jsp".

Figure 19: Response to client which was created by "helloWorld-jsr223-01.jsp".

the value of the *uri* attribute in the "*taglib*" JSP directive at the top of both JSP files!

- `rexxla_03_ScriptTagLib/helloWorld-bsf-02.jsp`: this JSP file (cf. Figure 20) includes a *script* element that defines the *type* attribute with the value `"rexx"` and refers to an external file containing the code because of the presence of the *src* attribute with the value `"helloWorld.rex"`. Note that the external Rexx program in this case needs to create all of the HTML text that gets returned to the requesting client as the JSP itself has no HTML text defined because it consists of two JSP directives and a single, empty JSP tag (`"<s:script …"`) only. The Rexx code from the external file gets executed using BSF. Its content is shown in Figure 24 at page 20 below. The client's browser rendering the returned HTML data is shown in Figure 21 below.
- `rexxla_03_ScriptTagLib/helloWorld-jsr223-02.jsp`: this JSP file (cf. Figure 22) includes a *script* element that defines the *type* attribute with the value `"rexx"` and refers to an external file containing the code because of the

```
<%@ page session="false" pageEncoding="ISO-8859-1" contentType="text/html;
charset=ISO-8859-1" %>
<%@ taglib uri="http://rexxla.org/taglibs/bsf" prefix="s" %>
<!-- external Rexx script will create all of the HTML! -->
<s:script type="rexx" src="helloWorld.rex" />
```

Figure 20: Content of "rexxla_03_ScriptTagLib/helloWorld-bsf-02.jsp".

Figure 21: Response to client which was created by "helloWorld-bsf-02.jsp".

presence of the *src* attribute with the value `"helloWorld.rex"`. Note that the external Rexx program in this case needs to create all of the HTML text that gets returned to the requesting client as the JSP itself has no HTML text

```
<%@ page session="false" pageEncoding="ISO-8859-1" contentType="text/html;
charset=ISO-8859-1" %>
<%@ taglib uri="WEB-INF/script-jsr223.tld" prefix="s" %>
<!-- external Rexx script will create all of the HTML! -->
<s:script type="rexx" src="helloWorld.rex" />
```

Figure 22: Content of "rexxla_03_ScriptTagLib/helloWorld-jsr223-02.jsp".

Figure 23: Response to client which was created by "helloWorld-jsr223-02.jsp".

defined because it consists of two JSP directives and a single, empty JSP tag ("`<s:script …`") only. The Rexx code from the external file gets executed using the newer standard Java scripting framework (JSR-223). Its content is shown in Figure 24. The client's browser rendering the returned HTML data is shown in Figure 23 above

Comparing the JSP text (cf. Figure 22) with the one for `helloWorld-bsf-02.jsp` (cf. Figure 20) it appears to be identical. The only difference is the value of the *uri* attribute in the *taglib* JSP directive at the top of both JSP files which determines whether Apache BSF or the newer and standard Java scripting framework gets used to execute the embedded scripts!

- `rexxla_03_ScriptTagLib/helloWorld.rex`: this file contains the Rexx code referred to in `helloWorld-bsf-02.jsp` and `helloWorld-jsr223-02.jsp`. Figure 24 below depicts its content.

  This Rexx program uses the *resource* directive introduced with ooRexx 5 which allows one to define the different sections of the HTML text verbatimly. The Rexx code makes sure to write these HTML sections interspersed with dynamically acquired information from the Java web server at the time this program runs.

  It is this servlet-like Rexx program that creates the HTML text for both, the `helloWorld-bsf-01.jsp` and `helloWorld-jsr223-01.jsp` JSP files.

- `rexxla_03_ScriptTagLib/WEB-INF/web.xml`: this file gets depicted in Figure 27 and is not needed for this web application. However, its content describes this web application in the Tomcat manager interface.

- The tag library description files `script-bsf.tld` (BSF taglib library) and `script-jsr223.tld` (JSR-223 taglib library) are contained in the *META-INF* directory of the *ScriptTagLibs* Java archives. Both files are almost identical except for the *shortname*, *uri*, *info* and the *tagclass* element (denotes the fully qualified Java class name that must be used to process the respective tag). Figures 25 and Error: Reference source not found depict the content of *"script-bsf.tld"* and highlight the most important definitions in it. The content of *"script-jsr223.tld"* is not shown for brevity, as it is almost identical except for the lines # 9, 10, 11, 14, 48, and having an additional attribute named "*compile*" for the *script* and *expr* tags.

```
use arg request
say .resources~top_part      -- write top HTML block
say "         <code>"request~getRequestURL~toString"</code>"
say .resources~middle_part   -- write middle HTML block
say "         <code>"request~getRequestURI"</code>"
say .resources~bottom_part   -- write bottom HTML block
   -- constant HTML texts
::resource top_part          -- top HTML block
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
      <title>Minimal ScriptTagLibs-JSP</title>
   </head>
   <body>
      <h1>Hello, world (ScriptTagLibs JSP)</h1>
      <p>This JSP was executed, because of the following URL:
      <ul><li>URL <em>request~getRequestURL()</em>: <br>
::END

::resource middle_part       -- middle HTML block
          <li>its URI being <em>request~getRequestURI()</em>: <br>
::END

::resource bottom_part       -- bottom HTML block
      </ul>
   </body>
</html>
::END
```

Figure 24: Content of "helloWorld.rex".

- `rexxla_03_ScriptTagLib/WEB-INF/lib`: this directory may contain the Java archives `bsf4ooRexx-v641-20201217-bin.jar`[13] (or newer) from BSF4ooRexx [11] and the *ScriptTagLibs* Java archive `javax.ScriptTagLibs.jar` (Tomcat 9 or earlier) or `jakarta.ScriptTagLibs.jar` (Tomcat 10 or later). If these archives are present then they get used first when looking for Java classes by Tomcat, otherwise these two Java archives must reside in Tomcat's home/installation directory ("`CATALINA_HOME`") and its immediate subdirectory named `lib`.

[13] Note: if there may be more than one web application that uses BSF4ooRexx, then one must copy `bsf4ooRexx-v641-20201217-bin.jar` (or newer) to the *CATALINA_HOME/lib* directory. The reason being that Tomcat uses different classloaders for different web applications and that the native BSF4ooRexx library caches Java classes for performance reasons. If cached Java classes get used by a classloader that did not load them then Java runtime errors may occur as a result. Placing `bsf4ooRexx-v641-20201217-bin.jar` (or newer) only into the *CATALINA_HOME/lib* directory foregoes this problem.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
  PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
  "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">
<!-- a tag library descriptor -->
<taglib>
  <tlibversion>2.0</tlibversion>
  <jspversion>1.1</jspversion>
  <shortname>BSF JSP Support</shortname>
  <uri>http://rexxla.org/taglibs/bsf</uri>
  <info> Uses Apache BSF to execute scripts and expressions </info>
  <tag>
    <name>script</name>
    <tagclass>org.rexxla.taglibs.bsf.Scriptlet</tagclass>
    <bodycontent>tagdependent</bodycontent>
    <info>Run script</info>
    <attribute>
      <name>type</name>
      <required>true</required>
    </attribute>
    <attribute>
      <name>arguments</name>
      <required>false</required>
    </attribute>
    <attribute>
      <name>cacheSrc</name>
      <required>false</required>
    <attribute>
      <name>debug</name>
      <required>false</required>
    </attribute>
    <attribute>
      <name>name</name>
      <required>false</required>
    </attribute>
    <attribute>
      <name>slot</name>
      <required>false</required>
    </attribute>
    <attribute>
      <name>src</name>
      <required>false</required>
    </attribute>
    </attribute>
    <attribute>
      <name>throwException</name>
      <required>false</required>
    </attribute>
  </tag>

  … continued in Figure26 below …
```

Figure 25: Content of "`script-bsf.tld`", part 1, continued in Figure 26 below …

```
… continued from previous Figure 25 above.

  <tag>
    <name>expr</name>
    <tagclass>org.rexxla.taglibs.bsf.Expression</tagclass>
    <bodycontent>tagdependent</bodycontent>
    <info>Evaluate script expression</info>
    <attribute>
      <name>type</name>
      <required>true</required>
    </attribute>
    <attribute>
      <name>arguments</name>
      <required>false</required>
    </attribute>
    <attribute>
      <name>cacheSrc</name>
      <required>false</required>
    </attribute>
    <attribute>
      <name>debug</name>
      <required>false</required>
    </attribute>
    <attribute>
      <name>name</name>
      <required>false</required>
    </attribute>
    <attribute>
      <name>slot</name>
      <required>false</required>
    </attribute>
    <attribute>
      <name>src</name>
      <required>false</required>
    </attribute>
    <attribute>
      <name>throwException</name>
      <required>false</required>
    </attribute>
  </tag>
</taglib>
```

Figure 26: Content of "script-bsf.tld", part 2, continued from Figure 25 above.

The *ScriptTagLib* implementations make sure for Rexx, that each Rexx script's standard output file will be transparently redirected to the supplied implicit "*out*" Java object[14], thereby causing Rexx *SAY* and ooRexx *.output~say* statements to be

[14] The JSR-223 support of ooRexx is realized with the *RexxScriptEngine* [13] that automatically prepends prompt strings when accessing standard files to ease the analysis of Java log files. In the case of accessing the standard output file the prompt "*REXXout>*" gets prepended which is undesired in the context of Java server pages as the requesting client would (unexplainably) get

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
                   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd"
                   version="4.0"
                   metadata-complete="true">
<display-name>Minimal JSP-ScriptTagLibs Demo</display-name>
<description>
   Welcome to a minimal JSP-ScriptTagLibs demo
</description>
</web-app>
```

Figure 27: Content of "rexxla_03_ScriptTagLib/WEB-INF/web.xml".

transparently redirected. As a result it is quite easy for Rexx programmers to create the HTML text to be sent back to the requesting client by merely using the standard Rexx output statements.

3 Roundup and Outlook

This article briefly introduced and discussed Java based web servers, Servlets, JSPs and taglib libraries. The author created a few taglib libraries (*"ScriptTagLibs"*) that allow script code to be deployed via Apache BSF or the newer standard Java scripting framework (a.k.a. JSR-223) as long as there are appropriate scripting engines available for each framework. As BSF4ooRexx creates a bidirectional ooRexx-Java bridge and implements BSF and JSR-223 it becomes possible to exploit Java server pages also for ooRexx where the JSP's code is not implemented in Java, but rather in [oo]Rexx! As a matter of fact, these new taglib libraries allow for mixing multiple programming languages (Java as well as Java scripting languages) in the same JSP page should a need arise. In addition one can mix execution of scripts via Apache BSF and the newer standard Java scripting framework which allows one to deploy older Java scripting languages for which only Apache BSF engine implementations are available.

The article demonstrated with ooRexx nutshell examples the usage of the *script* tag from the BSF and the JSR-223 *ScriptTagLibs*. Using the new ooRexx *resource* directive of *ooRexx 5.0* it becomes possible to easily define static HTML text in the ooRexx program and use it for creating the HTML text that gets returned to the

this prompt as part of the HTML text sent back to it. Therefore the *ScriptTagLib* implementation for JSR-223 makes sure, that *RexxScriptEngine*'s automatic standard output prompt gets removed.

requesting client. Taken together with the *ScriptTagLibs* default behaviour of always supplying the implicit Java objects *request*, *response*, and *out* for each script invocation as arguments one can create external script programs that mimicry the *{javax|jakarta}.servlet.Servlet*'s *service* method.

The script taglibs are available for both namespaces, *javax* (e.g., Tomcat 9 or earlier) and *jakarta* (e.g., Tomcat 10 or later), the distributions [9] carry the top level package names as their first word: ***javax****.ScriptTagLibs.jar* and ***jakarta****.ScriptTagLibs.jar*, respectively.[15]

Acknowledgements

The author wishes to thank DI Walter Pachl for his feedback and proofreading.

4 References

[1] "Jakarta EE", Wikipedia (as of 2020-11-19):
https://en.wikipedia.org/wiki/Jakarta_EE

[2] "WAR (file format)", Wikipedia (as of 2020-11-19):
https://en.wikipedia.org/wiki/WAR_(file_format)

[3] "Apache Software Foundation" (as of 2020-11-19): http://www.apache.org/

[4] "Apache Tomcat" (as of 2020-11-19): https://tomcat.apache.org/

[5] "Specifications" (links to Java/Jakarta EE specifications in the Apache Tomcat project) (as of 2020-11-19):
https://cwiki.apache.org/confluence/display/TOMCAT/Specifications

[6] "Bean Scripting Framework (BSF)" (as of 2020-11-19):
https://commons.apache.org/proper/commons-bsf/

[7] "JSR 223: Scripting for the Java Platform " (as of 2020-11-19):
https://jcp.org/en/jsr/detail?id=223

[8] "Apache BSF Taglib Libraries, Deprecated" (as of 2020-11-19):
https://svn.apache.org/repos/asf/jakarta/taglibs/deprecated/bsf/

[15] ScriptTagLibs [9] offers example web applications that demonstrate the use of *JavaScript*. As of 2021-03-16 an ooRexx webshop with cart, RDBMS, e-mail newsletter, file-upload et.al. got created by a student (a Bachelor thesis at WU Vienna) with the ScriptTagLibs and can be fetched from: http://wi.wu.ac.at/rgf/diplomarbeiten/#bakk_202102; another student's seminar paper experiments with *PHP* (sic!) and *Groovy* as a proof of concept and can be fetched from: http://wi.wu.ac.at/rgf/diplomarbeiten/#sem_202102_01.

[9] "ScriptTagLibs" (as of 2020-11-19):
https://sourceforge.net/projects/bsf4oorexx/files/Sandbox/rgf/taglibs/

[10] "Open Object Rexx (ooRexx)" (as of 2020-11-19):
https://sourceforge.net/projects/oorexx/

[11] "Bean Scripting Framework for ooRexx (BSF4ooRexx), a Bidirectional ooRexx-Java bridge" (as of 2020-11-19):
https://sourceforge.net/projects/bsf4oorexx/

[12] Flatscher R.G.: "Resurrecting REXX, Introducing Object Rexx", "ECOOP (RDL – Revival of Dynamic Languages – Workshop 10)", Nantes, France, July 3rd -7th 2006 (as of 2020-11-19):
http://wi.wu.ac.at/rgf/rexx/misc/ecoop06/ECOOP2006_RDL_Workshop_Flatscher_Paper.pdf

[13] Flatscher R.G.: "'RexxScript' – Rexx Scripts Hosted and Evaluated by Java (Package `javax.script`)", in: Proceedings of the "The 2017 International Rexx Symposium", Amsterdam, The Netherlands, April 9th - 12th 2017. URL (as of 2020-11-19):
http://www.rexxla.org/events/2017/presentations/201704-RexxScript-Article.pdf