Seminar paper

Tomcat Web Server:

CGI vs. Servlet

Author: Lukas Hubmaier
Matriculation No.: 0965587

Class Title: Projektseminar aus Wirtschaftsinformatik (Schiseminar)
Instructor: ao.Univ.Prof. Mag. Dr. Rony G. Flatscher
Term: Winter Term 2017/18
Vienna University of Economics and Business

List of Figures

1. Introduction

While everyone of us interacts with web servers on a daily basis as we browse through the internet, most people don't know the term "web server" not to mention understand it. The idea of web servers is quite simple. A web server is a computer system or just a software on that system that oversees handling HTTP requests from users visiting a certain website [1]. In the easiest case the content of a website is static and every user gets the same results, when visiting the page. We call this type of websites: static websites or websites with static content. Even though this type of websites may be suitable for some purposes, it is too limited for many other cases. This, consequently, leads to dynamic websites that are able to react on user input and respond with different websites for every user. The applications of this attempt can reach from simple form processing on webpages to sophisticated session management and individual page creation. As this paper serves as introduction to web servers, the focus lies on the basic concepts. While static websites are always written in plain HTML files with additional CSS files for design and images, dynamic content on websites can be achieved by various manners. In the following, terms like "Client-Side-Scripting", "Server-Side-Scripting", "CGIs" and "Servlets" are introduced, explained and illustrated, including instructions on how to use the scripting language Rexx or rather BSF4ooRexx for this purpose. While there is a multitude of different web servers available nowadays, this paper uses Tomcat (or Apache Tomcat) and gives some insights about the structure and mechanisms of this specific web server.

2. Static Content

Before diving into the specifics of CGIs and Servlets it is important to understand the fundamental process of visiting a website. Figure 1 illustrates the simplified process of sending a request to a web server. When entering an URL into the address bar of the client's web browser, the browser sends a HTTP request to the web server of the specified URL and expects a response in form of a html file. In addition to the HTML file representing the elements of the website, images, stylesheets other documents can be attached to the response. This chapter deals with the name resolution of URLs, URL mapping on websites and HTTP methods.

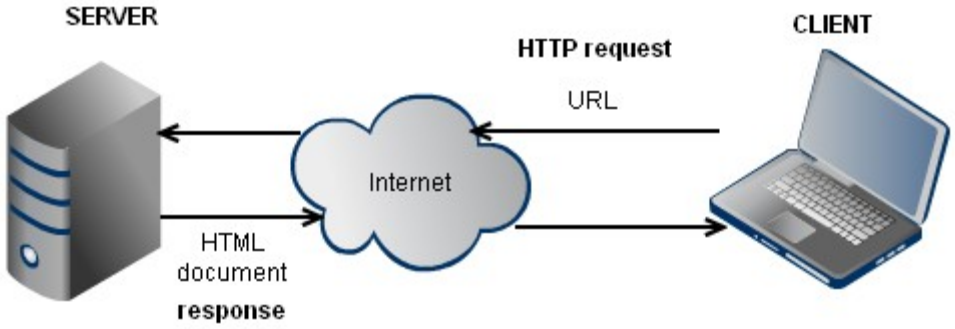


Figure 1 - Server/Client relation with static content [2]

2.1. The Route of an URL

A Uniform Resource Locator (URL) is used when resources are requested over the internet and are therefore used when communicating with a web server. A URL consists is structured based on this structure [3]:

```
scheme://host:port/path?query
```

- **scheme**: defines the protocol (HTTP, HTTPS, FTP, etc.)
- **host**: identifies the system, that wants to be reached
- **port**: (optional) defines the entry point of the system (most web servers assume standard ports, based on the protocol
- **path**: (optional) specifies the location of the resource at the host, which can directly point to a file or be redirected by URL-mapping of the web server

- **query**: (optional) contains additional information that can be used by the requested resource

Since the internet, and network communication in general, is based on the Internet Protocol Suite or TCP/IP, it is important to translate human readable names of hosts into IP addresses for the computer to understand it. This is realized with Domain Name Servers (DNS), which contain entries mapping names to IP addresses and answer requests to resolve names. This way a TCP connection can be established from the client to the host and a request can be sent.

2.2. Hypertext Transfer Protocol (HTTP)

The Hypertext Transfer Protocol is the foundation of the World Wide Web and provides tools for data communication over the internet [4]. It basically serves as a stateless request-response protocol between client and server, which means that the server does not save information about consecutive requests. To overcome this problem, many web servers use server-side sessions, cookies or hidden variables in web forms. Even though Tomcat is capable of all these actions, they won't be discussed in detail here.

2.2.1. Request

According to the definition of the HTTP Protocol, a request contains the following elements:

- request line
- request header
- an empty line
- optional message body

```
GET http://localhost:8080/BasicExamples/index.html?username=user1 HTTP/1.1
Host: localhost:8080
Connection: keep-alive
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/62.0.3202.94 Safari/537.36
Upgrade-Insecure-Requests: 1
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.
8
DNT: 1
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.9,de;q=0.8
```

Figure 2 - HTTP GET request

The request line, which is highlighted in red, contains the applied method (=GET), the resource requested from the server (requested URI) and HTTP-Version. The following lines are request header fields, which give more information about the specification of the request, such as the Host, the used browser or even cookies. Every request is entitled to have an optional body containing more data, which has to be separated by an empty line. In case of a GET request a body is not necessary.

In order to comprehend web servers, it is essential to understand the provided request methods of the HTTP protocol. As defined by the [5], the following request methods are available: GET, POST, HEAD, PUT, DELETE, CONNECT, OPTIONS and TRACE. For transmitting user data on websites, the methods GET and POST are most relevant, as they can be used to transmit user data of forms. Nevertheless, they are differently implemented and understood by web servers, which makes it crucial to investigate them in more detail.

The request in Figure 2 is an example for a GET request, as the leading word indicates. The URL `http://localhost:8080/BasicExamples/index.html` is requested, which which is interpreted as described in section 2.1.

If we look at a simple example of a web form and make use of different methods for transmitting data, we can see the difference in the request that is sent out. Figure 3 and 4 illustrate the code and the displayed result of the web form. To understand HTML Tags, it is advisable to study the fundamentals at [6]. At this point we are focusing on the used method, which can be either GET or POST.

```
<html>
<head></head>
<body>
<form action="evaluation.html" method="post">
Enter a username: <input type="text" name="username"></input><br>
<input type="radio" name="whatToDo" value="create">Create<br>
<input type="radio" name="whatToDo" value="remove" checked="checked">Remove<br>
<input type="submit">
<form>
</body>
</html>
```

Figure 3 - HTML Code of Web Form

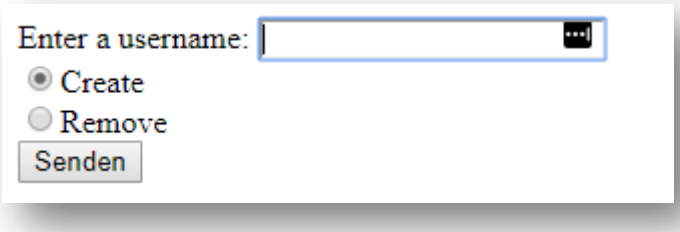


Figure 4 - Display of Web Form

When entering “Lukas” as username, selecting the radio button “Create” and pressing submit, a request is generated targeted for the resource “evaluation.html”. Depending on the specified method the information of the variables username and whatToDo are differently transmitted. While Figure 5 illustrates the request generated when using the GET method, Figure 6 shows the sent request with the use of the POST method. The parts highlighted in red in both requests define the passed parameters. As you can see, GET requests simply expand the URL by a list of parameters separated by a “&” symbol and adding a leading “?” symbol to indicate the beginning of the parameter list. The approach in POST requests is different, as the parameters are added to the request’s body (without a leading “?” symbol) and additionally the Header Field: Content-Length is set. This Field is necessary for the browser and the web server that is interpreting the body of the content, since most web servers don’t send an End of File (EoF) and would continue looking for more content in the body, which results in a stuck reading process [7].

```
GET
http://localhost:8080/BasicExamples/evaluation.html?username=Lukas&whatToDo=create
HTTP/1.1
Host: localhost:8080
Connection: keep-alive
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/62.0.3202.94 Safari/537.36
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.
8
DNT: 1
Referer: http://localhost:8080/BasicExamples/form.html
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.9,de;q=0.8
```

Figure 5 - HTTP GET Request with 2 parameters

```
POST http://localhost:8080/BasicExamples/evaluation.html HTTP/1.1
Host: localhost:8080
Connection: keep-alive
Content-Length: 30
Cache-Control: max-age=0
Origin: http://localhost:8080
Upgrade-Insecure-Requests: 1
Content-Type: application/x-www-form-urlencoded
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/62.0.3202.94 Safari/537.36
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.
8
DNT: 1
Referer: http://localhost:8080/BasicExamples/form.html
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.9,de;q=0.8

username=Lukas&whatToDo=create
```

Figure 6 - HTTP POST Request with 2 parameters

These different methods have advantages and disadvantages, which are discussed as follows:

- Allowed Characters: While GET requests only allow ASCII characters without some reserved characters "`:/?#[]@!$&'()*+,;=`" [8], POST requests are not subject to this restriction and also allow binary data for transmission.
- Visibility: GET requests add the parameters to the URL, which is visible and changeable by every user, while POST requests only send the data in the body. Manipulation of data is still possible, but not as obvious.
- Sensitive Data: For the same reason, as mentioned above, sensitive data must always be sent via POST, since the user does not want to see the password in the URL.
- Security: Appended information in the URL are therefore a security risk, as they are saved in the browser history and may be revealed if the user's system gets compromised.

More information about the differences of the HTTP Methods are available at [9].

2.2.2. Response

```
HTTP/1.1 200
Accept-Ranges: bytes
ETag: W/"80-1512689676768"
Last-Modified: Thu, 01 Dec 2017 23:34:36 GMT
Content-Type: text/html
Content-Length: 80
Date: Thu, 01 Dec 2017 23:35:28 GMT

<html>
<head></head>
<body>
<h1> Hello World! </h1>
</body>
</html>
```

Figure 7 - Example Reponse for HelloWorld.html

A HTTP response is the answer of the web server on a request and contains a header and a body, just like the response. While the body may contain the website, which is displayed at the user's screen, it can also be empty, depending on the status of the response. Message Status- Line, which contains the message code is written in line 1 together with the HTTP Version. These message codes can be used indicate the type of response to help browsers react properly. A list of all message codes is available at [10], but the most relevant groups of codes are the successful responses with code 2xx and the failed responses with code 4xx. Another important component for responses containing a DOM structure in form of HTML tags is the header field *Content-type: text/html*, as it tells the recipient how to interpret the response and if this type of content is accepted, which is defined in the *Accept* header field of the request.

3. Dynamic Content

To expand the features of webpages and meet today's standards of user experience for web browsing, the concept of the HTTP protocol remains, but is enriched by a dynamic aspect. This dynamic adaptation of webpages can be achieved either by scripts running on the client's side or on the server's side. Client-side scripts are executed by the web browser of the user and are executed after the client receives the response from the server. The most used scripting language on web pages is JavaScript but there are many other scripts that can be used instead. In general, client-side scripting is used to change interfaces or show and hide information to make the site more interactive. Even though performance may be better for client-side scripts, as you don't have a delay from the server for waiting for the response, it is not suitable for loading user specific content. Scripts like JavaScript can be adjusted by the user, since they are executed on the client's browser, which makes it not applicable for sensitive data. Also, user specific content is normally stored in databases, which are only accessible from the server.

Server-side scripts, on the other hand, contain all techniques that are used to adapt the response on the server, depending on the request and the users. They are mainly used to generate content or handle user sessions to keep track of their path between the pages. Suitable languages for this are all scripts and programs that are available on the server. Most common languages in this area are C#, Go, Java, Node.js (JavaScript), Python, PHP or Ruby [11], but also other languages like Rexx can be used to perform the core functions that are expected from a web server. One of the main advantages of server-side program execution is the independence from the client's system. The web programmer must not be concerned what script languages are available for the client, but rather delivers only the resulting response of the processed request. This behavior also allows the source code of the web application to be invisible for the user, which prevents others to copy your code. Furthermore, many use cases require the execution of server-side scripts or programs, since only the server is connected with the underlying database, that is storing all user information. This information could be user data, recent purchases in case of a web shop or even sensitive payment information of users, which should be protected especially well and far away from the clients.

3.1. Introduction to Tomcat

While there are various web server applications available and in use on the world wide web, they differ in operating system support, openness, security issues, support for dynamic content, user friendliness and other specifications. A list of available Webservers and their support can be found at [12]. Apache Tomcat is an open-source Webserver and container for Java Servlets and Java Server Pages (JSP) and is completely written in Java. Since the software is written in JAVA and runs on a Java Virtual Machine (JVM), it is compatible with every operating system that is supported by Java. Additionally, since BSF4ooRexx, the Bean Scripting Framework for Open Object Rexx [13], is also written in Java, it can be used to set up a web server using the script language Open Object Rexx [14]. In general, Tomcat can be used for two purposes. Primarily, it is in charge of creating dynamic content for the user, which may be achieved with Java Servlets, Java Server Pages or Common Gateway Interfaces (CGIs). Secondarily, it also provides the functionality to act as a general-purpose HTTP server, to handle requests and responses, which are discussed in Section 2. Another benefit of Apache Tomcat is the free use and the available source code of the software.

3.1.1. History

Tomcat was created by the Sun Microsoft software architect, James Duncan Davidson, as a Servlet reference implementation in 1998. With his help, the project was introduced to the open source community in 1999 with version 3.0 [15] Since then, the project was developed further, by adding new features and fixing issues and is at the time of creation of this paper at the stable version of 8.5 (This version will also be used in nutshell examples of this paper).

3.1.2. Components

Due to the object-oriented architecture of the Tomcat, the software can be split into various components. The relationship of the components is illustrated in Figure 8. While the single components are explained as follows, it is important to highlight the relevance of the Catalina Engine, since it is the core of Tomcat and provides the actual implementation of the most recent Servlet specification of the Java Servlet API.

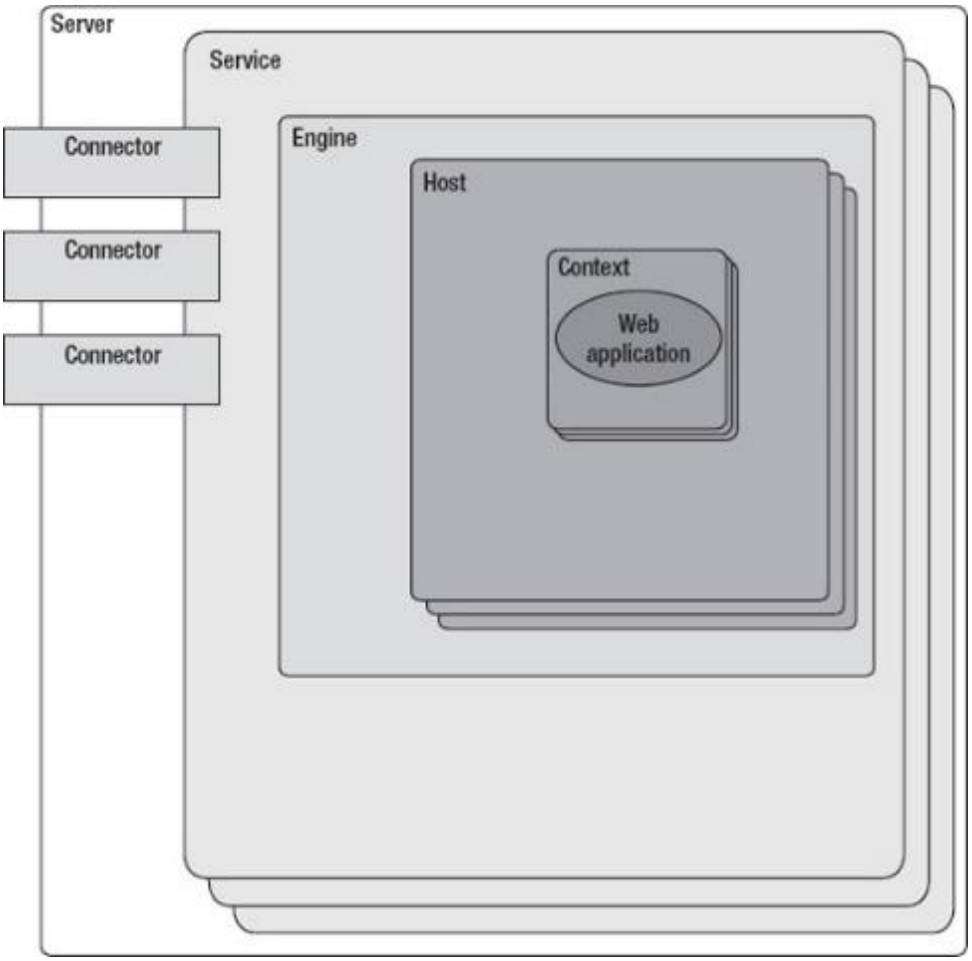


Figure 8 - Architecture of Tomcat Server [16]

- The Server – can exist only once and encapsulates the whole application in a separate JVM to prevent a crash of the whole system, if Tomcat crashes.
- The Service – holds one or more Connectors to establish a connection with one or more web ports.
- The Connector – is responsible for the HTTP request and response handling as discussed in section 2.
- The Engine – is the core of the Servlet container and manages the Servlets. This is where the computation of dynamic content happens.
- The Host – may introduce virtual hosts for different web applications, if it is needed.
- The Context – is the smallest component in the Tomcat architecture and contains a single web application. This is the place to configure specific web applications in regards of security. Each host can have multiple contexts and therefore also multiple web applications.

In addition to this architecture overview, there are many other libraries involved in the Tomcat software. Worth mentioning, is the Jasper 2 JSP Engine, which implements the specification of Java Server Pages. This engine allows to write code directly between HTML Tags, which results in less needed code, but will not be discussed in this paper.

3.1.3. File Structure

The system variable $CATALINA_HOME refers to the installation directory of Tomcat (has to be set) and contains by default the following directories:

Directory	Contents
/bin	Contains the startup and shutdown scripts for both Windows and Linux including essential Jar files for startup.
/conf	Contains the main configuration files for Tomcat, such as server.xml, defining the components discussed in section 3.1.2 and the global web.xml, which can be overwritten for each web application.
/lib	Contains the Tomcat Java Archive (jar) files, shared across all Tomcat components. All web applications deployed to Tomcat can access the libraries stored here. This includes the Servlet API and JSP API libraries.
/logs	Log files for debugging.
/temp	Temporary system files.
/webapps	The directory where all web applications are deployed, and where you place your WAR file when it is ready for deployment.
/work	Tomcat's working directory where Tomcat places all Servlets that are generated from JSPs. If you want to see exactly how a particular JSP is interpreted, look in this directory.

Table 1 - Directory Structure of Tomcat

The most relevant directory is "/webapps", since it is home of every web application. To give an example, the directory $CATALINA_HOME/webapps/examples describes

the web application that is reached with the URL localhost:8080/examples from the host system. The structure of a single web application is also predefined and should contain the following files and directories [17]:

File/Directory	Contents
index.html	This file is shown by default if no specific file name is given in the URL
/WEB-INF/web.xml	This file is the Web Application Deployment Descriptor of the web application, which defines URL Mappings, Servlets and other components that define your web application.
/WEB-INF/lib/	Additional .jar files that are required for your web application and are not part of the global $CATATLINA_HOME/lib folder can be added here.
/WEB-INF/classes	Since Servlets are Java classes, this is the directory where Tomcat will look for the classes to execute them at runtime.
/META-INF/MANIFEST.MF	A short description of your web Application, in a way that is common for every .JAR file.
/META-INF/context.xml	Tomcat Context Descriptor can be used to alter detailed configuration of the specified web application. If this file does not exist, the global Context Descriptor at $CATALINA_HOME/conf/context.xml will be used.

Table 2- File structure of a web application

Even though not these directories are necessary for all web applications, this structure is specified be the Web ARchive (WAR) format.

3.2. CGI

3.2.1. What is a Common Gateway Interface?

Common Gateway is a standard for web servers that defines the exchange of data between a web server and additional, external scripts and programs. These so-called CGI scripts are executed by the web server for every request from a client and generate and return a HTML response depending on received data. CGI scripts typically interact with databases to retrieve more information about the user that sent the request, but since this is just one of many ways to use CGI scripts, we focus on the interface between server and script/program. As it is illustrated in Figure 9, a CGI script is called when the user submits a form in his web browser to the server. The server then passes the information to the CGI script and executes it. The created response is passed back to the server and then forwarded to the user.

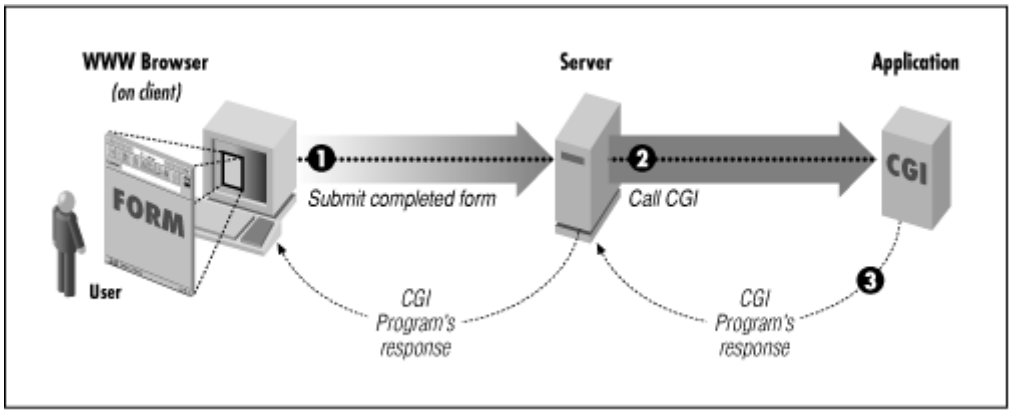


Figure 9 - Dataflow for CGIs [18]

The passing of data from the web server to the CGI happens with the help of environment variables. They are a series of hidden values that are sent to every executed CGI program by the web server and contain information about the requests. [19] gives an overview and explanation of most CGI variables, but a closer look at the RFC for CGI [20] is advisable, since the list is not exhaustive. To get acquainted with CGI programming and get information from forms, the most important variables are QUERY_STRING and CONTENT_LENGTH. These variables can be used by the CGI script read form data and create dynamic content. In addition to the environment variables, the CGI script is also able to read the STDIN stream, which can be used to pass data.

The dataflow from the CGI script back to the web server is established by redirecting the STDOUT stream of the program to the response writer of the web server. This

basically means, that it is possible to use familiar output methods to write the HTML response. A PHP Script would use “ECHO(x)”, perl would use “print(x)” and Rexx would use “SAY x” to write directly to the response.

As you can see in the last example of output functions and the definition of a CGI as a standard, one of the most relevant benefits of this technique is the possibility to use every scripting or programming language you are familiar with, as long as it has access to the STDIN and STDOUT stream to read request information and write to the response. This may be one reason for the wide use of CGI, as there is no need to learn new script or programming languages to start with dynamic web development, resulting in faster and easier realization of projects.

Opposed to client-side scripting, CGIs are only dependent on the operating system and available interpreters and compilers of the web server, and do not rely on the client’s system. Additionally, it is possible to exchange the web server software from Tomcat to Apache HTTP Server or Oracle HTTP Server, since they all use the same interface to communicate with the executed script. See [12] for available Webserver software that supports CGIs.

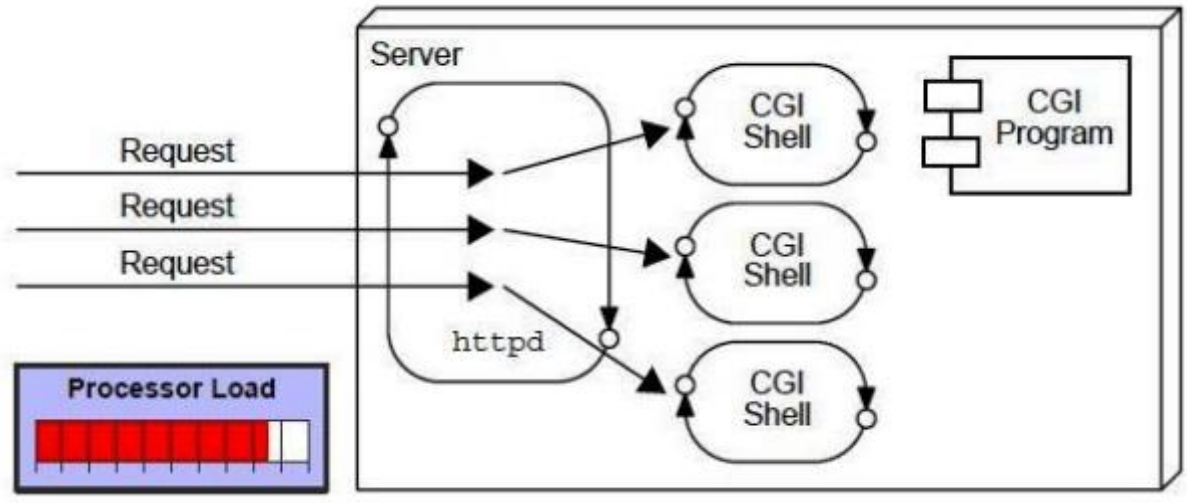


Figure 10 - Performance of CGIs [21]

As you can see in the list of web servers above, there are different types of CGIs, like Simple Common Gateway Interface (SCGI) or Fast Common Gateway Interface (FCGI) etc. These are variations that aim to improve the performance of regular CGIs by reducing the overhead of single requests. This points out one of the disadvantages of CGI programming, as every request leads to the creation of a new process on the server, which can often consume more memory and time as the

execution of the script itself (see Figure 10). High user traffic can easily overwhelm the web server leading to a crash of the system or simply higher response rates. Even though the improved variations of CGIs perform better under high stress, alternative methods, such as Servlets can be favorable.

Another issue that should be concerned with the use of CGIs is the security aspect. It is important to understand, that the web server executes all scripts he is told to even if they include malicious code to delete data or give away sensitive information. Therefore, usually a specified folder – by default "/cgi-bin" – is defined and granted explicit rights to execute programs on the web server, so that the web server software is not even able to execute other resources on the system. Nevertheless, it is important, to sanitize user inputs of forms to prevent unexpected code that is transmitted by a user (Hacker) to be executed.

3.2.2. Execution of Rexx Scripts in Tomcat

An instruction on how to configure Tomcat to enable CGI support can be found in the documentation of the recent Tomcat server [22].

Basically, the installation requires 4 steps:

1. Create a new web application
2. Set up Servlet and Servlet mapping (web.xml)
3. Allow Tomcat to execute external scripts (context.xml)
4. Write script (+ html file with form)

1. To create a web application, called *CGIRexx* we simply create a new folder in the directory *$CATALINA_HOME/webapps* and add the folder *META-INF* and *WEB-INF*. Additionally, we add the folder CGI in the *WEB-INF* directory, which is the home directory of all executable scripts. The expanded *webapps* directory should look like Figure 11. Any request sent to *yourIP:8080/CGIRexx* will now look for HTML files in this folder. If your browser and your web server are on the same device, *yourIP* can be replaced with *localhost*.

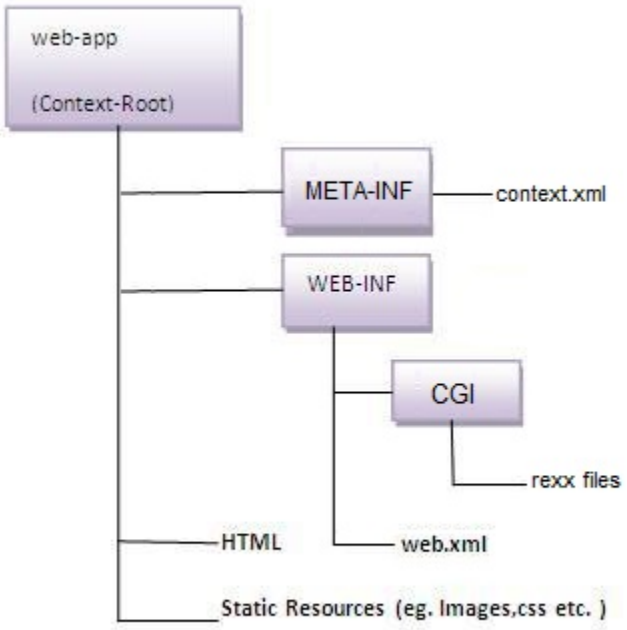


Figure 11 - Web Application Structure for CGI [21] - adapted

2. In the next step we configure the web application and create the file web.xml in the directory WEB-INF. To get the basic structure of an web.xml file you can look at *$CATALINA_HOME/conf/web.xml*, which is the default configuration file, if no application specific file can be found. The relevant information of this file can be found in the tags <Servlet> and <Servlet-mapping>. To understand the file structure, it is important to understand in which order Tomcat processes this information. When a resource X is requested on the web server for the web application *CGIRexx* by sending a request to *localhost:8080/CGIRexx/X*, the web server is looking for a defined Servlet mapping and if none was found, looks for a X directly. In our case, we define the <url-pattern> */cgi-bin/** to load the <Servlet> called *cgi*. This url-pattern is very common in CGI programming, but can be changed, if necessary. To tell Tomcat what the Servlet called *cgi* is, we must define the tag <Servlet> and set the name to *cgi*. The <Servlet-class> is important because it directs the request to the built-in CGIServlet, which is part of the Catalina.jar file in *$CATALINA_HOME/lib*. Now Tomcat can match the URL to the right class to handle the Common Gateway Interface. We can further specify the Servlet by adding <init-param> parameters.

- cgiPathPrefix defines the prefix for the executed script. The recommended value is *WEB-INF/cgi*, since WEB-INF is not accessible for clients.

- Executable defines which executable is used to execute the script file. This can be perl, rexx or every other program that is known to the PATH of the system. Even .exe files can be executed if executable is set to no value.

```
<?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_3_1.xsd"
  version="3.1">

    <display-name>
        Application to demonstrate Rexx with CGI
    </display-name>

    <description>
        This is a simple web application to illustrate the use of CGI and Rexx
    </description>

    <servlet>
        <servlet-name>cgi</servlet-name>
        <servlet-class>org.apache.catalina.servlets.CGIServlet</servlet-class>
        <init-param>
          <param-name>cgiPathPrefix</param-name>
          <param-value>WEB-INF/cgi</param-value>
        </init-param>
        <init-param>
          <param-name>executable</param-name>
          <param-value>rexx</param-value>
        </init-param>
    </servlet>


    <servlet-mapping>
        <servlet-name>cgi</servlet-name>
        <url-pattern>/cgi-bin/*</url-pattern>
    </servlet-mapping>

</web-app>
```

Figure 12 - web.xml of CGIRexx web application

After setting up this file, the URL *localhost:8080/CGIRexx/cgi-bin/helloWorld.rex* results in the execution of the bash command illustrated in Figure 13.

```
C:\Program Files\Apache Software Foundation\Tomcat 8.5\webapps\CGIRexx> rexx WEB-INF/CGI/helloWorld.rex
```

Figure 13 - CGI command

3. Since Tomcat's security configurations prohibit the execution of external programs by default, we have to allow it for this specific web application/context. To do so, a context.xml file has to be placed in the directory *CGIRexx/META-INF* with the <Context privileged="true"> attribute set. See Figure 14:

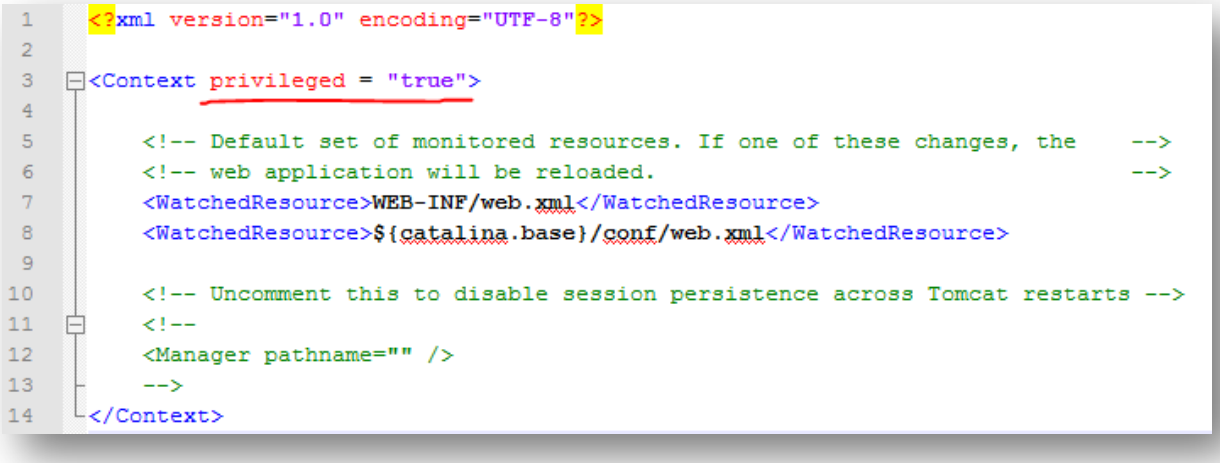


Figure 14 - CGI context.xml

An example of the context.xml file can be found at *$CATALINA_HOME/conf*. It is not advisable to change the context.xml file directly in the *conf* directory, since this would apply additional rights to all web applications on this web server, leading to security issues.

4. For adding Rexx scripts and html files see Section 3.2.3.

3.2.3. Rexx CGI Nutshell Examples

This section gives some examples on how to use Rexx to process CGI Requests and how to create responses that are readable for a browser.

3.2.3.1. HelloWorld.rex

If everything is set up correctly, the obligatory HelloWorld script illustrated in Figure 15 and 16 should be created and placed in the folder CGIRexx/WEB-INF/CGI with the name helloWorld.rex.

Figure 15 - Display of helloWorld.rex

```
Say "Content-type: text/html"
Say
--Say "<html>"
--Say "<head>"
--Say "</head>"
--Say "<body>"
Say "<h1>HELLO WORLD!</h1>"
Say "<h1>It really is a REXX Script</h1>"
--Say "</body>"
--Say "</html>"
```

Figure 16 - helloWorld.rex

Even though the code is not very long, some parts are essential for every CGI script. As mentioned in section 3.2.1. every output from STDOUT is redirected to the response, which enables to simply use *Say*. The first line defines the MIME Type of the document and is required by the browser to understand and interpret the HTML in order to create the appropriate DOM tree. Additionally, the HTTP charset parameter can be set in the same line, as described in [23]. The empty second line is also necessary. Interestingly, it is not necessary to define <html>, <head> and <body> tags in the script, since they are generated automatically if not provided, but if adaptions to the header are needed, they can simply be added to the script.

3.2.3.2. getMethod.rex

This example illustrates how to read parameters from requests using the GET method. To attach parameters to a request, clients usually use forms, but in the case of GET parameters the URL could also just be adapted. For that reason, a HTML file is needed, as it is shown in Figure 17.

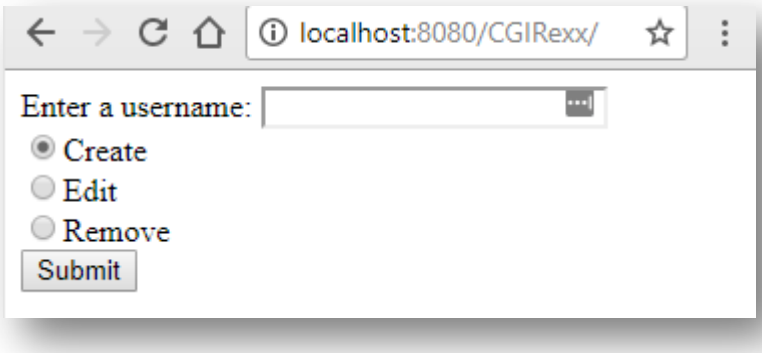


Figure 17 - Display of form

The index.html file, which is illustrated at Figure 18, is placed in the root directory of the web application to be reachable for the client. The URL *localhost:8080/CGIRexx* automatically looks for the index.html file, as it is usual with web servers. This behavior is specified in the web.xml file in Tomcat's configuration directory defines a <welcome-file-list> and can be altered.

```
1  <html>
2   <head></head>
3  <body>
4  <form action="cgi-bin/getMethod.rex" method="get">
5      Enter a username: <input type="text" name="username"></input><br>
6      <input type="radio" name="whatToDo" value="create" checked="true">Create<br>
7      <input type="radio" name="whatToDo" value="edit">Edit<br>
8      <input type="radio" name="whatToDo" value="remove">Remove<br>
9      <input type="submit">
10 <form>
11  </body>
12  </html>
```

Figure 18 - index.html with form

The code includes basic HTML tags, but we focus here on the attributes in the <form> tag. The attribute action defines the recipient of the form-data. In this case a relative path was used, but absolute paths are also possible. Since the method is set to "get" a GET request will be sent to the action URL and parameters will be attached to the URL as described in section 2.2.1. The displayed result with the parameters

- username=”User 1234”
- whatToDo=”create”

is shown in Figure 19.

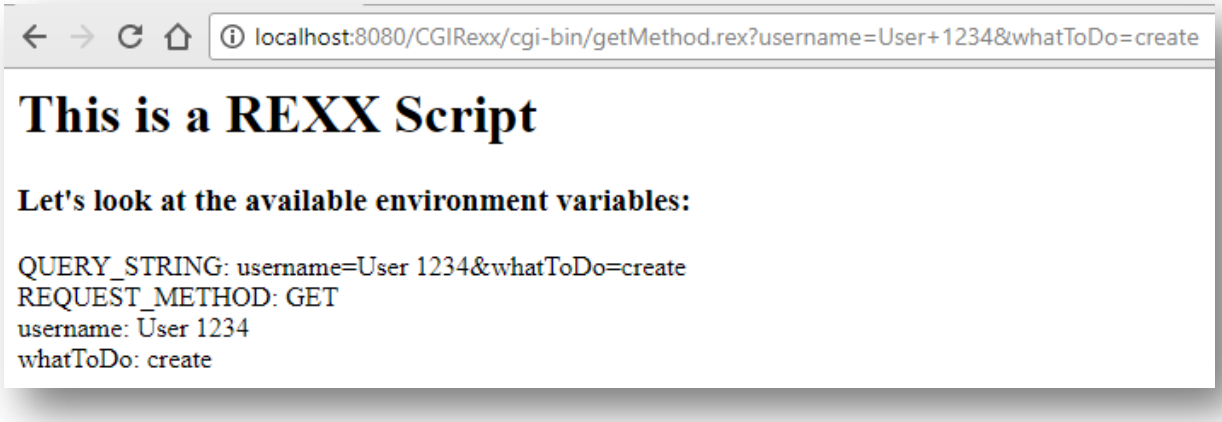


Figure 19 - Display of getMethod.rex

The script only reads and displays parameters but a closer look into the code (Figure 20) is required to understand what is being processed by Rexx. Line 5 and 6 assign the value of the environment variable QUERY_STRING, which was set by the CGI, to the local variable QUERY_STRING. Beside QUERY_STRING and REQUES_METHOD all other environment variables set by the CGI can be accessed in this way. When looking at the URL in more detail, it is noticeable that the CGI translates spaces in variables to ‘+’ symbols, which have to be reverse before parsing the string. This is done in line 8. Additionally, special characters are transcoded to hexadecimal encoding, but this problem is not handled in this example. More details to this parsing process can be found at [24]. The following DO block parses the QUERY_STRING into pairs of names and values and assigns values to the variables. The remaining script just displays the created variables by writing it to the response in form of html code.

```
Say "Content-type: text/html"
Say

-- Accessing Environment Variables set by CGI
QUERY_STRING = value("QUERY_STRING",,"ENVIRONMENT")
REQUEST_METHOD = value("REQUEST_METHOD",,"ENVIRONMENT")

QUERY_STRING = TRANSLATE(QUERY_STRING,' ','+')                  -- replaces '+' with ' '

if REQUEST_METHOD = GET then DO
        input = QUERY_STRING
        DO UNTIL input=''
           PARSE VAR input Name'='Value'&'input                 -- split parameters
           value(Name,Value)                                    -- assign variables
        END
END


Say "<h1>This is a REXX Script</h1>"
Say "<h3>Let's look at the available environment variables: </h3>"
Say "<div>QUERY_STRING: " QUERY_STRING"</div>"
Say "<div>REQUEST_METHOD: " REQUEST_METHOD"</div>"
Say "<div>username: " username"</div>"
Say "<div>whatToDo: " whatToDo"</div>"

~
```

Figure 20 - getMethod.rex

3.2.3.3. postMethod.rex

This example is similar to getMethod.rex, but the data is transmitted and received differently. To send the POST request, we copy the index.html file and name it postForm.html. Additionally, we change the <form> tag as follows:

<form action="cgi-bin/postMethod.rex" method="post">

We can expand the example *postMethod.rex* (shown in Figure 21) by adding an if-else clause, which differentiates the REQUEST_METHOD and creates variables accordingly.

```
Say "Content-type: text/html"
Say

-- Accessing Environment Variables set by CGI
QUERY_STRING = value("QUERY_STRING",,"ENVIRONMENT")
REQUEST_METHOD = value("REQUEST_METHOD",,"ENVIRONMENT")



if REQUEST_METHOD = GET then DO
        QUERY_STRING = TRANSLATE(QUERY_STRING,' ','+')                  -- replaces '+' with ' '
        input = QUERY_STRING
        DO UNTIL input=''
           PARSE VAR input Name'='Value'&'input                         -- split parameters
           value(Name,Value)                                            -- assign variables
        END
END
else DO
        if REQUEST_METHOD = POST then DO
                Parse Pull CONTENT_BODY                                 -- read request body from STDIN
                CONTENT_BODY = TRANSLATE(CONTENT_BODY,' ','+')          -- replaces '+' with ' '
                input = CONTENT_BODY
                DO UNTIL input=''
                   PARSE VAR input Name'='Value'&'input                 -- split parameters
                   value(Name,Value)                                    -- assign variables
                END
        END
END


Say "<h1>This is a REXX Script</h1>"
Say "<h3>Let's look at the available environment variables: </h3>"
Say "<div>REQUEST_METHOD: " REQUEST_METHOD"</div>"
Say "<div>QUERY_STRING: " QUERY_STRING"</div>"
Say "<div>CONTENT_BODY: " CONTENT_BODY"</div>"
Say "<div>username: " username"</div>"
Say "<div>whatToDo: " whatToDo"</div>"
```

Figure 21 - postMethod.rex

In case of POST Requests, parameters are attached to the body of the request, which is automatically directed to the InputStream of the Rexx file. The important line here is `Parse Pull CONTENT_BODY`, which saves the complete input (=body of request) in the variable CONTENT_BODY.

3.3. Servlet

3.3.1. Introduction to Servlets

A Servlet, or also called Java Servlet, is a Java class and is embedded into a Servlet Container. In this case Tomcat or, to be more precise, Catalina is the Servlet Container that calls the Servlet. As opposed to CGIs, Servlets are part of the Web Container and are executed inside the Java VM, that is created by Tomcat, guaranteeing more security. Depending on the version of Tomcat, it supports different Versions of Servlets [25]. Tomcat 8.5, which is used in this context, accepts Servlet Version 3.1. In general, Servlets are waiting for requests, execute some code and finally return a response to the user. Even though this appears to be similar to CGIs, it is the more advanced approach to create dynamic content on web servers.

Since Servlets are programmed in Java, the benefit from the Java infrastructure. This results in operating system independence, in improved error handling, which offers more features than other scripting languages used by CGIs, more security and more robustness, due to the existence of Java's garbage collector which reduces memory leaks etc.

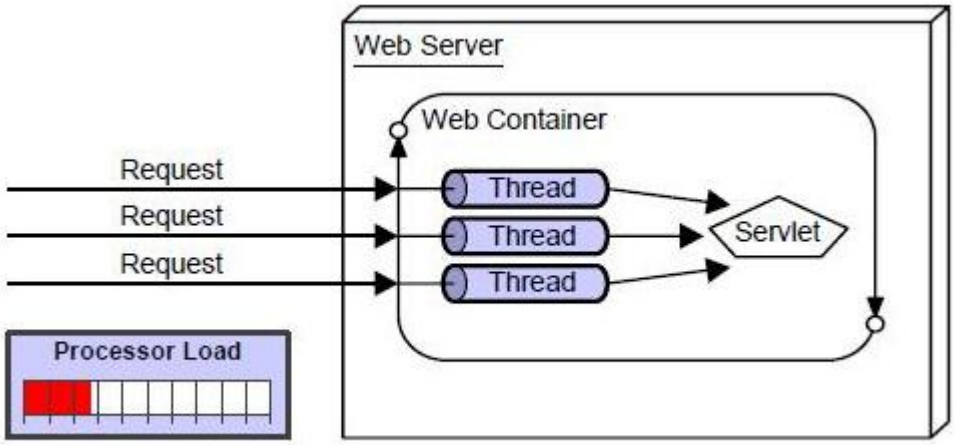


Figure 22 - Performance of Servlets [21]

Unfortunately, the benefits coming from Servlets result in knowledge of Java programming. This paper presents one way to create a Servlet, that uses Rexx Script with the support of BSF4ooRexx, but will be discussed in section 3.3.5. with some nutshell examples.

Another important advantage of Servlets over CGIs is performance. While CGIs create a new process for every request, with Servlets there is only one process for

the whole web container handling all requests and creating new threads. This reduces overhead immensely and is noticeable with high stress on web servers. See Figure 22 in contrast to Figure 10 to notice the difference.

3.3.2. Lifecycle and Methods of Servlets

In general, there is only one requirement for a Java class to be considered a Servlet by Tomcat - the class has to implement the interface javax.Servlet.Servlet. This means, it must support the methods that are defined by the interface. By looking at the documentation of this interface [26] , this only requires the implementation of basic methods but none that are related to HTTP requests. To have access to more methods, it is advisable to create a class that extends the class javax.Servlet.HTTPServlet [27], which itself implements the required interface, but also spits requests into different methods, depending on their used request method.

A typical Servlet makes use of the following methods to handle requests:

- `service(HttpServletRequest req, HttpServletResponse resp)`
- `doGet(HttpServletRequest req, HttpServletResponse resp)`
- `doPost(HttpServletRequest req, HttpServletResponse resp)`
- `doPut (HttpServletRequest req, HttpServletResponse resp)`
- etc.

The method `service()` is called for every request and redirects the call to the corresponding doXXX method. This method should only be overwritten if no differentiation of methods is needed, otherwise the doXXX methods should be implemented directly.

The lifecycle of a Servlet can be summarized to three methods (see Figure 23):

- init() – is started after the constructor of the class was called and can be used to initialize and prepare data, that may be needed for requests.
- service() – is called for every incoming request
- destroy() – is called, once at the end of the life cycle and should be used to terminate pending processes, close connections and general cleanup activities.

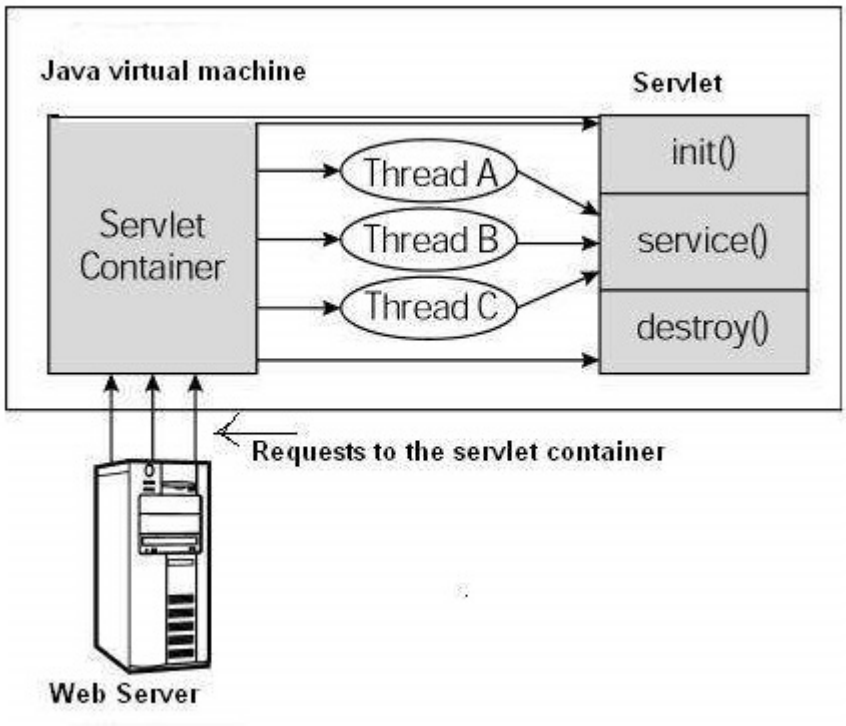


Figure 23 - Life Cycle of Servlets [28]

3.3.3. Configuration of Servlets

As already mentioned in Section 3.2.2., the main configuration of Servlets happens in the web.xml file, located in the WEB-INF directory of the web application. This file can define various Servlets, that are executed with the start of the Tomcat service. The Servlet mapping must be set accordingly to tell the web server which Servlet is responsible for a request and where the class file of the Servlet is located.

Figure 24 shows parts of an web.xml example. Tomcat expects a fully qualified class name of the Servlet.class file and scans the directory /yourWebApp/WEB-INF/classes. The directory structure of a web application with Servlets should be composed as illustrated in Figure 25.

```
<servlet>
    <servlet-name>RexxServletName</servlet-name>
    <servlet-class>RexxServlet</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>RexxServletName</servlet-name>
    <url-pattern>/rexx</url-pattern>
</servlet-mapping>
```

Figure 24 - Servlet mapping and class reference

As you can see, the file structure is similar to the example of CGIs, since CGI also make use of Servlets, but we have an additional directory for classes. If additional libraries are required for Tomcat to run a specific Servlet, they can either be added to the lib folder in WEB-INF of your web application or in the $CATALINA_HOME/lib folder, to make the library accessible to all web applications of the server.

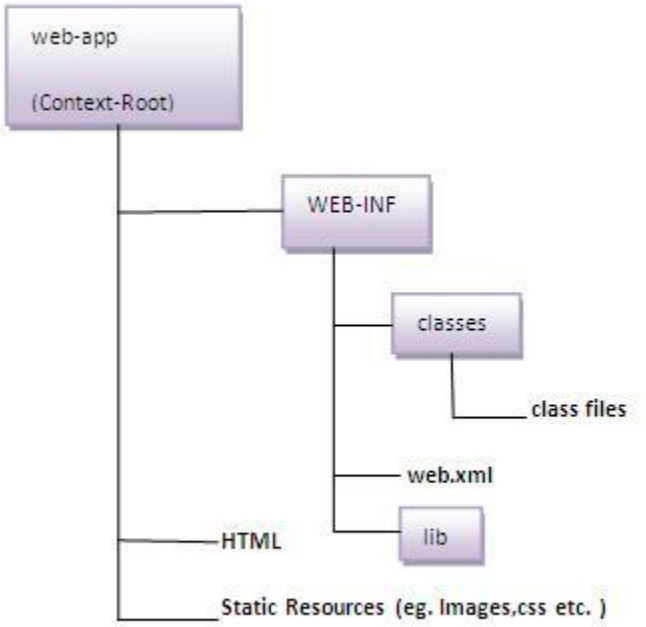


Figure 25 - directory structure for Servlets [21]

3.3.4. Java Servlet Nutshell Examples

This section demonstrates basic examples for Servlets and uses some code from the examples provided by Tomcat. Before creating the first Servlet, it is essential to have set up the environment of your system correctly to compile the Servlets.

1. Install Java SDK 7+ (use the same processor architecture as on other related installations, such as ooRexx and BSF4ooRexx)

2. Set up or edit the following system variables, if they are not already set correctly:

Variable	Value	Reason
JAVA_HOME	C:\Program Files\Java\jdk1.7.x_xxx	To reference java directory
PATH	%JAVA_HOME%\bin	To find javac.exe
CATALINA_HOME	C:\Program Files\Apache Software Foundation\Tomcat 8.5	(optional) if not already done with TomCat installation
CLASSPATH	%CATALINA_HOME%\lib\Servlet-api.jar	To find imports when compiling Servlet
CLASSPATH	%JAVA_HOME%\lib	

3. Compile the HelloWorldServlet.java file with the command:

```
C:\Program Files\Apache Software Foundation\Tomcat 8.5\webapps\ServletExamples
\WEB-INF\classes>javac HelloWorldServlet.java
```

3.3.4.1. HelloWorldExample.java

Since the file HelloWorldServlet.java does not exist yet, we have to create and save it in the WEB-INF/classes directory. Figure 26 illustrates the minimal required code for a Servlet, that responds with “Hello World!” to every incoming request. What we can see in line 9, is that the class extends the HttpServlet class and therefore qualifies for a Servlet. In line 12 the method doGet is overwritten with the arguments request and response. To create the dynamic content, the request variable can be used to receive information and the response variable can be used to write a response. In this example, the request is neglected, but the response is modified. Line 15 and 16 set the content type of the response, analogous to the CGI example. To write directly

to the body of the response, which represents the content of the displayed website, it is necessary to save the PrintWriter object of the response, which is done in line 17. This object can be used to write a String to the response (see line 19).

```
1  import java.io.IOException;
2  import java.io.PrintWriter;
3
4  import javax.servlet.ServletException;
5  import javax.servlet.http.HttpServlet;
6  import javax.servlet.http.HttpServletRequest;
7  import javax.servlet.http.HttpServletResponse;
8
9  public class HelloWorldServlet extends HttpServlet {
10
11     @Override
12     public void doGet(HttpServletRequest request, HttpServletResponse response)
13         throws IOException, ServletException
14     {
15         response.setContentType("text/html");
16         response.setCharacterEncoding("UTF-8");
17         PrintWriter out = response.getWriter();
18
19         out.println("<h1>Hello World!</h1>");
20     }
21 }
```

Figure 26 - HelloWorldServlet.java

Now the file has to be compiled with the java compiler to create a class-file. To do so, execute the command as described in section 3.3.4. Additionally, the web.xml file of the web application has to be configured, as described in Figure 24. If you use the <url-pattern> /helloWorld and the <Servlet-class> HelloWorldServlet, assuming that your application is called ServletExamples, you can send a request to the Servlet using the URL *localhost:8080/ServletExamples/helloWorld*.

3.3.4.2. RequestHeaderServlet.java

The next example is reading information from the request object. To be more specific, all Header Fields of the request are queried and returned to the response in HTML format.

Figure 27 - Display of requestHeaderServlet.java

To display all available Header Fields, we have to call request.getHeaderNames(). Consequently, all values can be read by iterating over this generic list/enumeration and printing the formatted name and value to the response stream.

```
1  import java.io.*;
2  import java.util.*;
3  import javax.servlet.*;
4  import javax.servlet.http.*;
5
6  public class RequestHeaderServlet extends HttpServlet {
7
8      public void doGet(HttpServletRequest request, HttpServletResponse response)
9      throws IOException, ServletException
10     {
11         response.setContentType("text/html");
12         response.setCharacterEncoding("UTF-8");
13         PrintWriter out = response.getWriter();
14
15         Enumeration e = request.getHeaderNames();
16         while (e.hasMoreElements()) {
17             String name = (String)e.nextElement();
18             String value = request.getHeader(name);
19             out.println(name + " = " + value + "<br/>");
20         }
21     }
22 }
```

Figure 28 - requestHeaderServlet.java

3.3.5. Rexx Servlet Nutshell Examples

Since Java programming can be quite challenging, the following examples propose a method to use a predefined Java Servlet that executes a rexx script, which handles the request and creates the response.

The connection of Java Servlets with rexx scripts requires the following additional installed software (same architecture as JAVA JDK):

- Open Object Rexx (ooRexx) [14]
- Bean Scripting Framework for Open Object Rexx (BSF4ooRexx) [13]

The idea of this bridge to Rexx is to create a Java Servlet that reads a Rexx file and executes it with the Scripting Engine of the BSF Manager. Both these classes are part of the BSF4ooRexx .jar file that comes with the installation of the framework. The scripting engine allows to pass the request object (which is a Java object) to the Rexx file and allows to handle the request with Rexx syntax. This syntax then creates a response String in form of HTML tags and returns it to the Servlet. The Servlet again uses this returned object to write the HTTP response.

Even though this initial process might appear to be complex, this bridge has many advantages. First, requests can be handled with the easy readable Rexx syntax and Java programming can be prevented in a large part. Second, once the Servlet is up and running, the script takes effect immediately after alterations occurred. This is not the case for Servlets, as they have to be recompiled. Additionally, Tomcat has to reload the application for every alteration.

3.3.5.1. RexxServlet.java

```java
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
import org.apache.bsf.*;   // BSF support
import java.nio.file.Files;
import java.nio.file.Paths;

public class RexxServlet extends HttpServlet {

    private BSFEngine rexxViaBSF;
    private BSFManager mgr;

    @Override
    public void init(){
        mgr = new BSFManager ();                // get an instance of BSFManager
        try {
          rexxViaBSF = mgr.loadScriptingEngine("rexx");  // load the Rexx engine
        } catch (BSFException e)
        {
            e.printStackTrace();
            mgr.terminate(); // make sure that the Rexx interpreter instance gets terminated!
        }
    }

    @Override
    public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException
    {
        response.setContentType("text/html");
        response.setCharacterEncoding("UTF-8");
        PrintWriter out = response.getWriter();
        try {
            java.util.Vector vArgs = new java.util.Vector();  // Vector for arguments to be passed to Rexx
            vArgs.addElement(request);     // arg # 1

            //read rex script from file
            String path = this.getServletContext().getRealPath("/WEB-INF/doGet.rex");
            String rexxCode = new String(Files.readAllBytes(Paths.get(path)));

            // Execute RexxScript and save return object (in this case String)
            Object rexResponse = rexxViaBSF.apply ("RexxServlet.java", 0, 0, rexxCode, null, vArgs);
            out.println(rexResponse);       // write results from Rexx script to body of response
        }
        catch (BSFException e)
        {
            out.println(e);     //prints Exception in Browser
            e.printStackTrace();    // prints Details of Exception in log-files
        }
    }

    @Override
    public void destroy(){
        mgr.terminate(); // make sure that the Rexx interpreter instance gets terminated!
    }
}
```

Figure 29 - RexxServlet.java

As described above, the key to this approach is the RexxServlet class. To use this class, it is necessary to fulfill two requirements:

1. Make sure your compiler can find the package org.apache.bsf. This package is part of the bsf4ooRexx-xxxxxx.jar file in the installation directory of the bean scripting framework (probably C:\Program Files\BSF4ooRexx). To inform the compiler about the location of this file, it is necessary to be added to the

classPath. This should be automatically done with the installation of bsf4ooRexx, but checking it is important, since it is critical for the Compiler.

2. Since Tomcat needs to have access to the library, it is also necessary to copy the bsf4ooRexx-xxxxxx.jar file into the directory /WEB-INF/lib of your web application. The directory $CATALINA_HOME/lib would also be okay, but it would provide the library to all web applications which is probably not wanted.

When these requirements are fulfilled, the file illustrated in Figure 29 can be compiled in the familiar manner.

The RexxServlet class is built similar to the JavaServlets from section 3.3.4. but is extended with some additional code. In addition to the implementation of the doGet method, the methods init() and destroy() are overwritten, which are part of the Servlet life cycle and are called after starting the web application and before shutting it down. The init() method creates a BSFManager object, which has access to methods of the Bean Scripting Framework for Open Object Rexx. Line 18 loads a Scripting Engine by the name *rexx*, which later will be used to execute the Rexx code. Line 21 and 22 are only executed if an error occurs in the lines above and terminate the Scripting Engine to prevent dead processes or open data streams, which is the same procedure as in destroy(). The doGet() method is extended by line 33-50 to support Rexx files. First, a vector is created and the request it added. Next, the absolute path of the file doGet.rex in the /WEB-INF directory of the web application is generated (line 38) and the file is parsed into a String (line 39). The filename can be changed, but must fit to the name of the Rexx script. For security reasons it is important to locate the Rexx script inside the WEB-INF folder, since it prevents clients to access it directly. Line 42 calls the apply() method of the Script Engine and passes the Rexx Code with the argument vector containing the request object to interpret it. Consequently, the return value of this method is saved to the object variable rexResponse. Since a String is being returned in the Rexx script, the Servlet can simply print this value to the response body, which is displayed in the browser. If any exceptions occur in the Java program or while interpreting the Rexx code, an exception is thrown and printed to the response instead (line 47). This behavior is good for debugging, but should be changed for running systems. The bsf4ooRex provides more examples and applications of interaction with Rexx from Java in the directory /samples/java of the bsf4ooRexx installation.

3.3.5.2. doGet.rex (HelloWorld)

As usual, the first program to test a new framework and syntax displays “Hello World!”. The expected output of our Rexx script is shown in Figure 30.

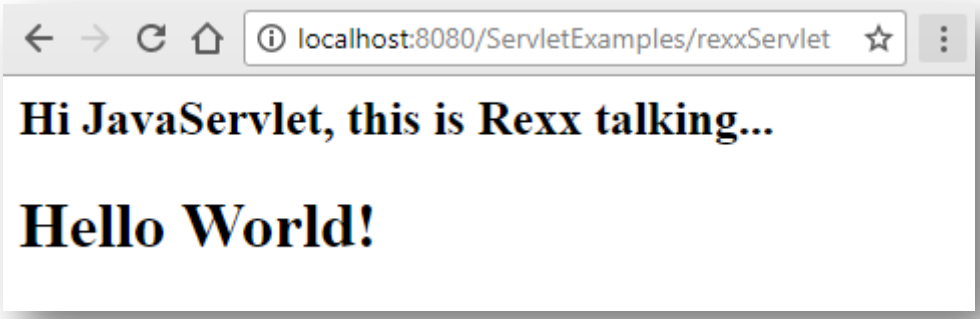


Figure 30 - Display of doGet.rex (Hello World)

```
-- start defining the outputString which will be returned to Servlet
outputString = '<h2> Hi JavaServlet, this is Rexx talking... </h2>'

outputString =  outputString '<h1> Hello World!</h1>'

-- return string to Java Servlet
return outputString
```

Figure 31 - doGet.rex (Hello World)

The script in Figure 31, which has to be located in the WEB-INF directory, is only concerned with writing a String for the response, but shows the working connection between the RexxServlet class and the script.

3.3.5.3. doGet.rex (RequestHeader)

In this example, the request that is passed to the Rexx Scripting Engine is used to read information about the request header. Any other information of the request can be extracted in a similar way. This example delivers comparable results to the Java example in section 3.3.4.2.

To get access to the request, the passed argument vector can be used to assign the Java object to a variable. In fact, the transmitted object is of type `org.apache.catalina.connector.RequestFacade` since is it handled by Tomcat. Additionally, the request headers are scanned and appended to the variable allHeaderFields, which later is added to outputString and displayed.

```
-- read request as argument
jRequest = arg(1)

-- start defining the outputString which will be returned to Servlet
outputString = '<h2> Hi JavaServlet, this is Rexx talking... </h2>'

-- read all header fields
headerNames = jRequest~getHeaderNames
headerFields = ''
allHeaderFields = ''
Do while HeaderNames~hasMoreElements = 1
        headerName = headerNames~nextElement
        headerFields = headerFields headerName ' '
        allHeaderFields = allHeaderFields headerName'=' jRequest~getHeader(headerName)';  <br/>'
END

-- display various information from request
outputString =  outputString 'Method : ' jRequest~getMethod '<br/>'
outputString =  outputString '<br/>'
outputString =  outputString 'HeaderFields : <br/>' headerFields '<br/><br/>'
outputString =  outputString 'allHeaderFields : <br/>' allHeaderFields '<br/>'

-- return string to Java Servlet
return outputString
```

Figure 32 - doGet.rex (Request Header)

Even though these examples only provide a brief insight into the capabilities of web servers, the connection to Rexx is established and java examples and tutorials can be adapted to Rexx. This can include sending form data, reading cookies from clients or handling sessions.

4. Conclusion

After presenting two available techniques to create dynamic web content on the Servlet container and web server Tomcat, both have advantages and disadvantages. While CGIs excel at the ease of use and the acceptance of already known scripting languages, it has issues in various areas compared to Servlets. The most severe problems appear to be security concerned, because of the execution of external programs and bypassing of Java's security mechanism, and performance concerned, due to the creation of unnecessary overhead on every request by starting separate processes. An additional downside of CGIs is the dependency on the operating system, since the according interpreters and compilers have to be compatible with the web server it is running on. The implementation of Servlets, on the other hand, can be problematic for users without any experience in Java or Compilers, which might scare off some web developers. This issue can be reduced by using the presented method of invoking Rex Scripts from Java Servlets.

5. References

[1] Wikipedia, "Web server," [Online]. Available: https://en.wikipedia.org/w/index.php?title=Web_server&oldid=810425625. [Accessed 14 12 2017].

[2] K. László, "Application Development in Web Mapping," 2010. [Online]. Available: http://www.tankonyvtar.hu/en/tartalom/tamop425/0027_ADW1/ch01s02.html. [Accessed 13 12 2017].

[3] IBM, "The components of a URL," [Online]. Available: https://www.ibm.com/support/knowledgecenter/en/SSGMCP_5.2.0/com.ibm.cics.ts.internet.doc/topics/dfhtl_uricomp.html. [Accessed 14 12 2017].

[4] Wikipedia, "Hypertext Transfer Protocol," [Online]. Available: https://en.wikipedia.org/w/index.php?title=Hypertext_Transfer_Protocol&oldid=813782185. [Accessed 14 12 2017].

[5] IETF, "RFC 7231 - Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content," [Online]. Available: https://tools.ietf.org/html/rfc7231. [Accessed 13 12 2017].

[6] W3Schools, "HTML Tutorial," [Online]. Available: https://www.w3schools.com/html. [Accessed 13 12 2017].

[7] J. December and M. Ginsburg, "Writing CGI Scripts," [Online]. Available: http://lnr.irb.hr/ebooks/1575211777/ch30.htm#ASimpleREXXCGIScript. [Accessed 13 12 2012].

[8] IETF, "RFC 3986 - Uniform Resource Identifier (URI): Generic Syntax," [Online]. Available: https://tools.ietf.org/html/rfc3986 . [Accessed 13 12 2017].

[9] w3school, "HTTP Methods: Get vs. Post," [Online]. Available:

https://www.w3schools.com/tags/ref_httpmethods.asp. [Accessed 13 12 2017].

[10] w3school, "HTTP Status Messages," [Online]. Available: https://www.w3schools.com/tags/ref_httpmessages.asp. [Accessed 13 12 2017].

[11] CodeSchool, "Server-side Languages," [Online]. Available: https://www.codeschool.com/beginners-guide-to-web-development/server-side-languages. [Accessed 13 12 2017].

[12] Wikipedia, "Comparison of Web server Software - Wikipedia," [Online]. Available: https://en.wikipedia.org/w/index.php?title=Comparison_of_web_server_software&oldid=813411213. [Accessed 14 12 2017].

[13] R. G. Flatscher et Al., "Bean Scripting Framework for Open Object Rexx," [Online]. Available: https://sourceforge.net/projects/bsf4oorexx/. [Accessed 13 12 2017].

[14] R. G. Flatscher et. Al., "Open Object Rexx," [Online]. Available: http://www.oorexx.org/. [Accessed 13 12 2017].

[15] T. Khare, Apache Tomcat 7 Essentials, Packt Publishing Ltd, 2012.

[16] A. Vukotic and J. Goodwill, Apache Tomcat 7, Berkeley, Calif.: Apress, 2011.

[17] Apache, "Application Develope'r Guide - Deployment," [Online]. Available: http://tomcat.apache.org/tomcat-8.5-doc/appdev/deployment.html. [Accessed 13 12 2017].

[18] S. Guelich, S. Gundavaram and G. Birznieks, CGI programming with Perl, vol. 29, Beijing; Cambridge, Massachusetts: O'Reilly, 2012.

[19] J. Hamilton, "CGI Programming 101," [Online]. Available: http://www.cgi101.com/book/ch3/text.html. [Accessed 13 12 2017].

[20] IETF, "RFC 3875 - The Common Gateway Interface (CGI) Version 1.1," 10 2004. [Online]. Available: http://www.ietf.org/rfc/rfc3875. [Accessed 13 12 2017].

[21] javatpoint, "Learn Servlet Tutorial," [Online]. Available: https://www.javatpoint.com/servlet-tutorial. [Accessed 13 12 2017].

[22] Apache, "Apache Tomcat 8 - CGI How To," [Online]. Available: http://tomcat.apache.org/tomcat-8.5-doc/cgi-howto.html. [Accessed 13 12 2017].

[23] W3C, "Einstellung des HTTP-charset-Parameters," [Online]. Available: https://www.w3.org/International/articles/http-charset/index.de. [Accessed 13 12 2017].

[24] R. L. A. Cottrell, *Web Techniques Magazine,* vol. 1, no. 2, 1996.

[25] Wikipedia, "Java Servlets," [Online]. Available: https://en.wikipedia.org/w/index.php?title=Java_servlet&oldid=811382076. [Accessed 13 12 2017].

[26] Oracle, "Servlet Documentation," [Online]. Available: https://docs.oracle.com/javaee/7/api/javax/servlet/Servlet.html. [Accessed 13 12 2017].

[27] Oracle, "Java Documentation," [Online]. Available: https://docs.oracle.com/javaee/7/api/javax/servlet/http/HttpServlet.html. [Accessed 13 12 2017].

[28] Tutorialspoint, "Servlets Life Cycle," [Online]. Available: https://www.tutorialspoint.com/servlets/servlets-life-cycle.htm. [Accessed 13 12 2017].

[29] IETF, "RFC2616," [Online]. Available: https://www.w3.org/Protocols/rfc2616/rfc2616-sec5.html. [Accessed 13 12 2017].

[30] Tuturialspoint, "Servlet Life Cycle," [Online]. Available: https://www.tutorialspoint.com/Servlets/Servlets-life-cycle.htm. [Accessed 13 12 2017].

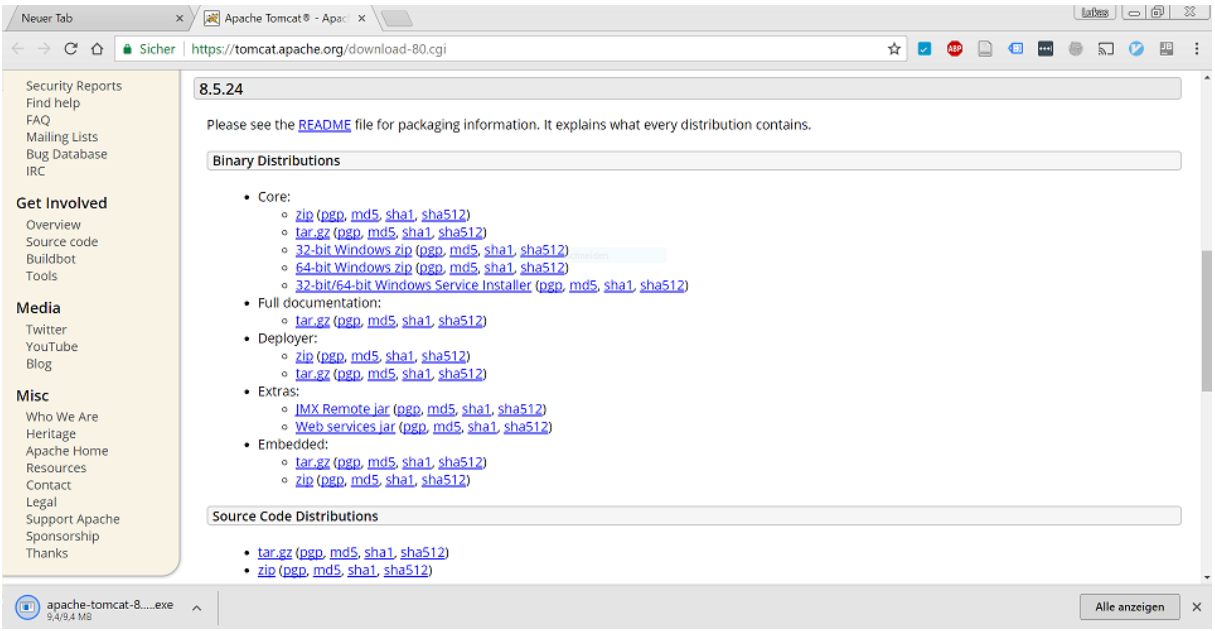
Appendix

a. Tomcat Installation/Configuration Guide for Windows

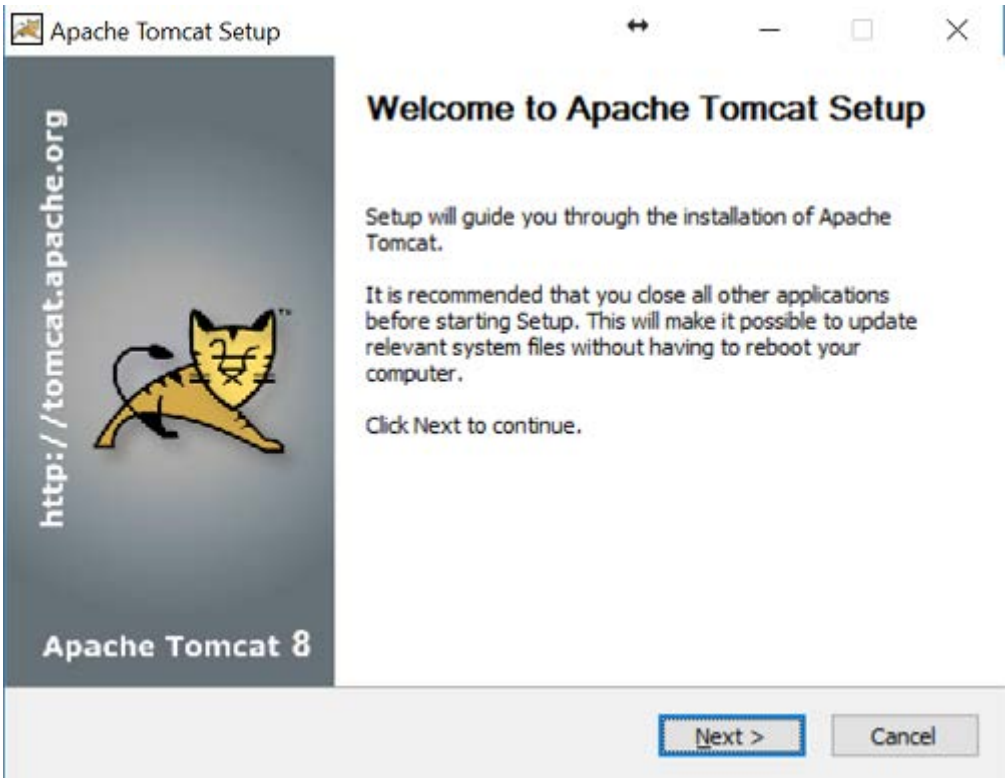
This guide gives a step-by-step instruction on how to install Apache Tomcat on a Windows operating system. A prerequisite for this installation is an installed JRE in a Version suitable to the Tomcat Version.

Go to the website of Tomcat and download the latest version.

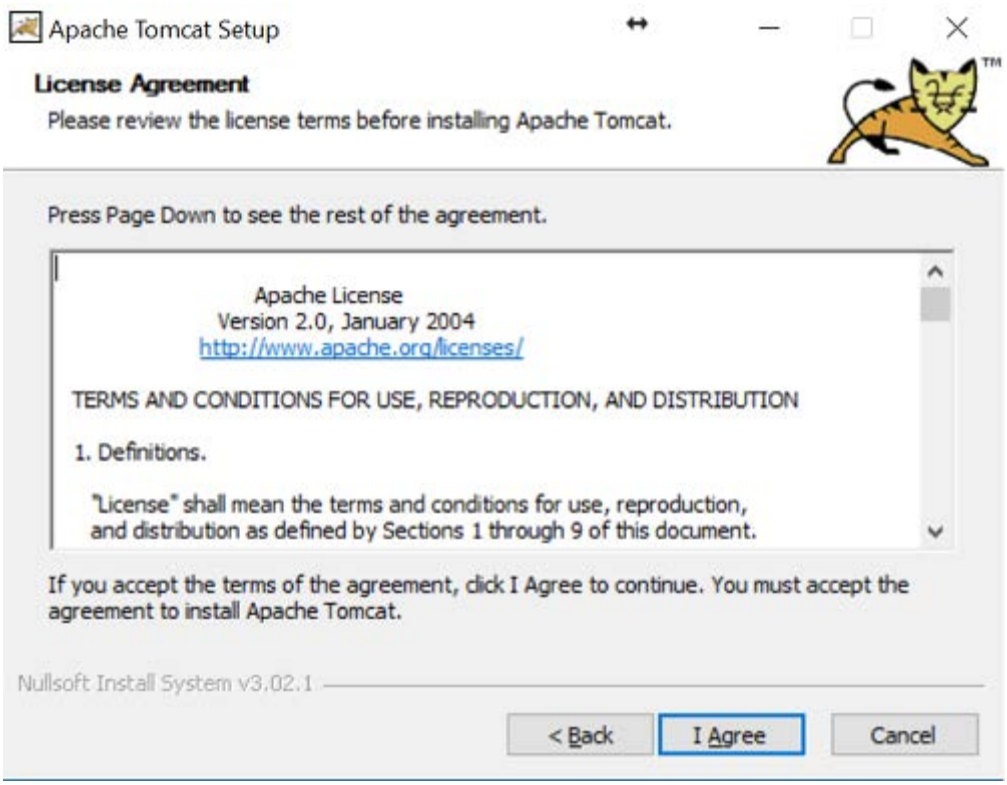
URL: https://tomcat.apache.org/download-80.cgi



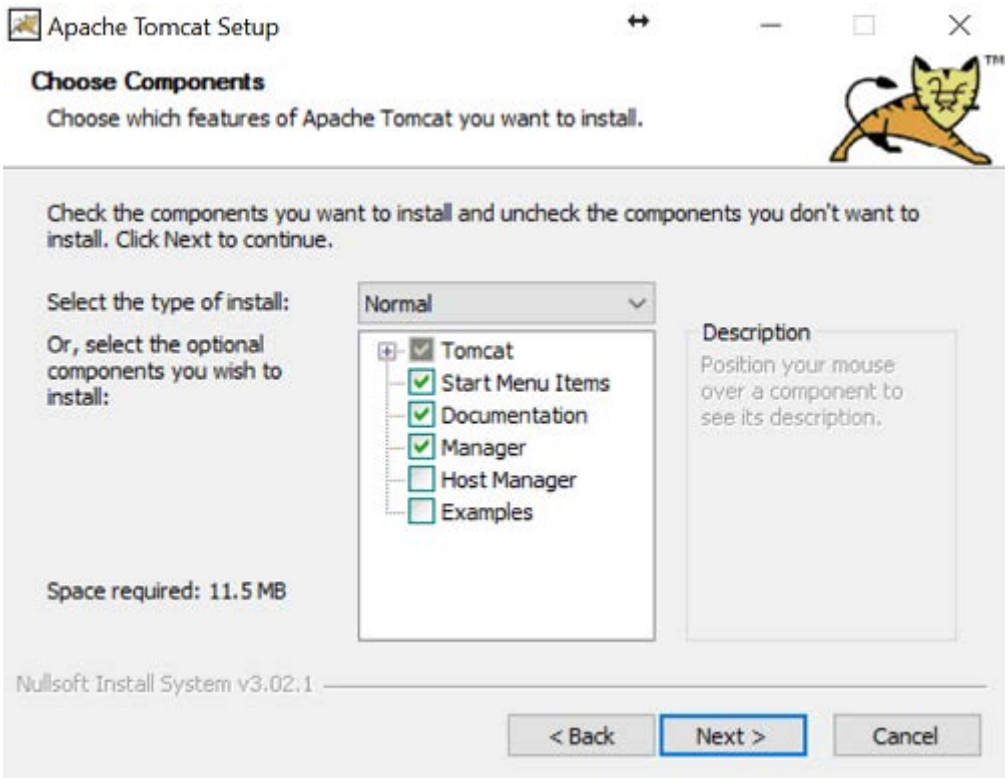
Choose 32-bit/64-bit Windows Service Installer and execute the downloaded .exe file on your system.



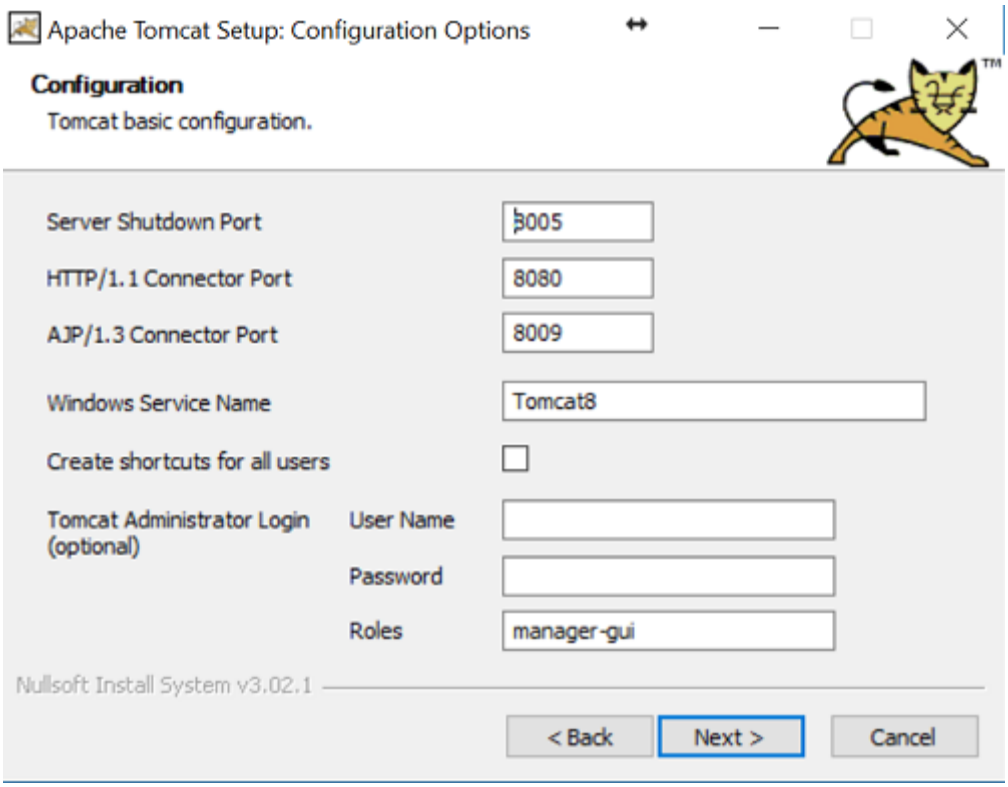
Press Next to start the installation.



Read terms and Agree, if you want to use the software.

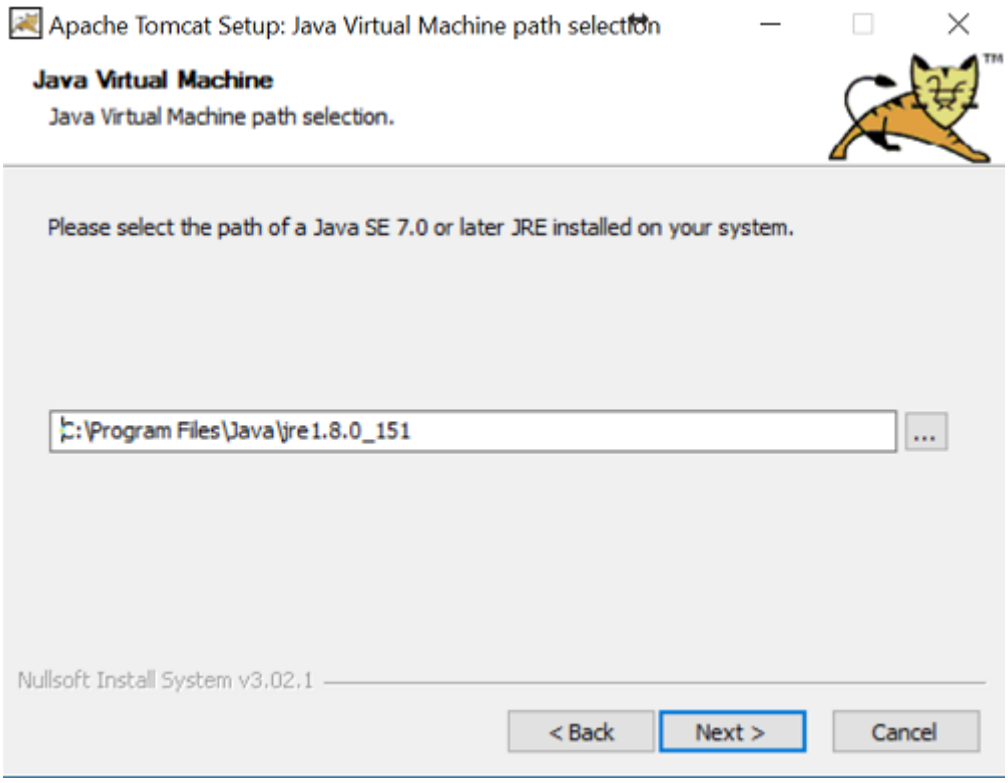


Select the components you want to install. You can leave the default option or check “Examples” to create web application with various examples for Servlets and JSP.

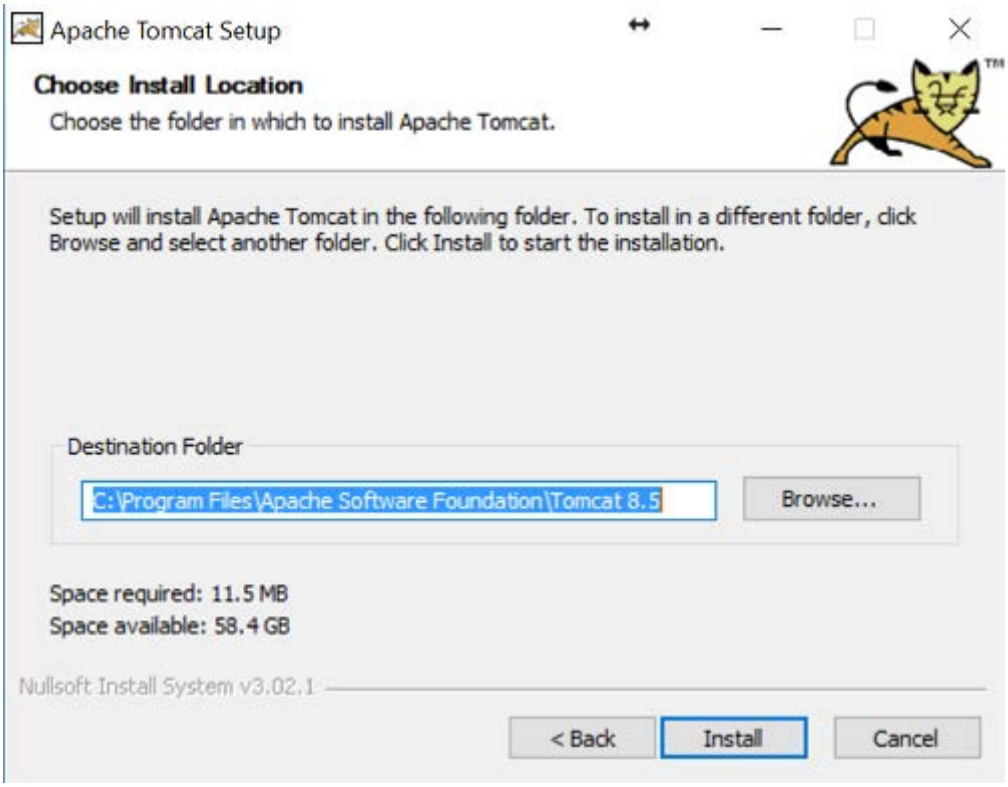


In this window you can define on which ports you want to listen for connections. The most important port number is the HTTP/1.1 Connector Port which is 8080 by default. This is the port that is used to send HTTP Requests to the websever, hence localhost:8080.

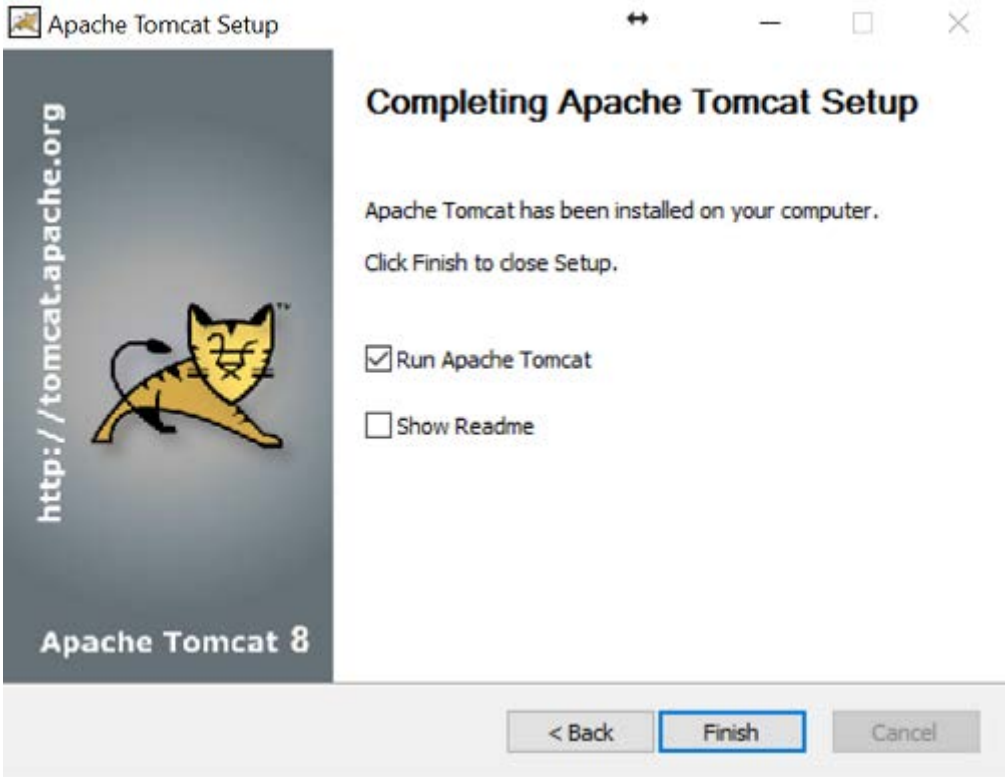
If you want to use the mangager application of Tomcat which allows you to restart web applications and configure settings in the webbrowser, add your credentials in the textfields “User Name” and “Password”. These can be set later aswell, but more complicated.



Choose the location of your Java Runtime Environment to inform Tomcat about the location of the JVM library which is needed to start. This may be as illustrated in the figure above, or inside your JDK directory, such as: JDK 1.x.x_xxx/JRE.

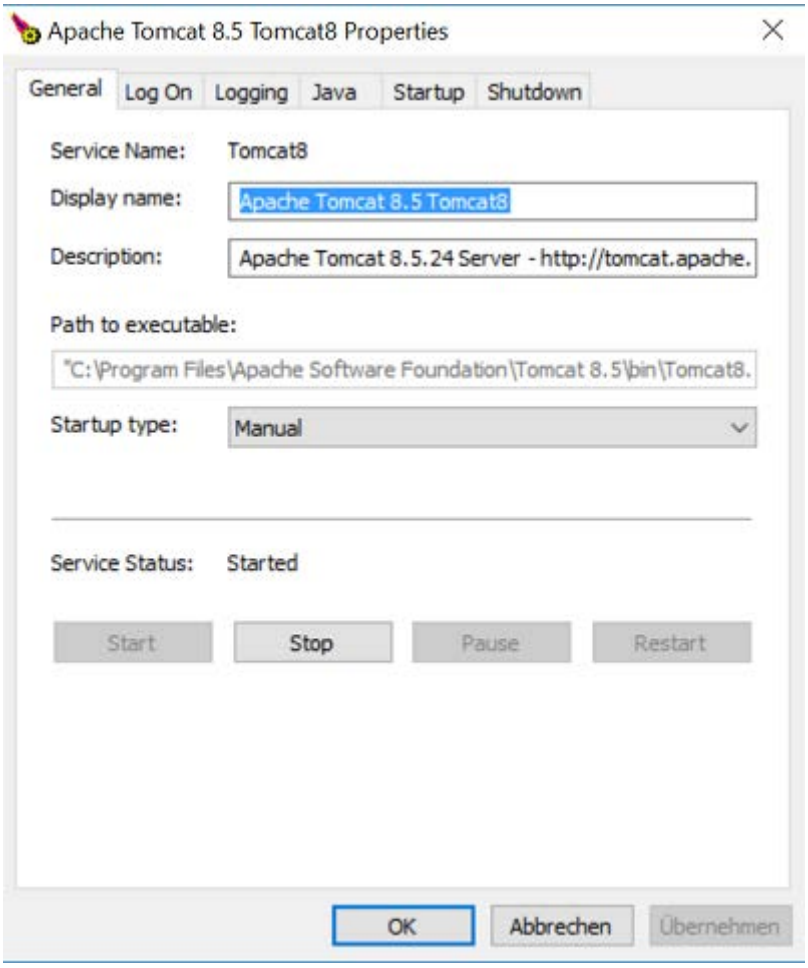


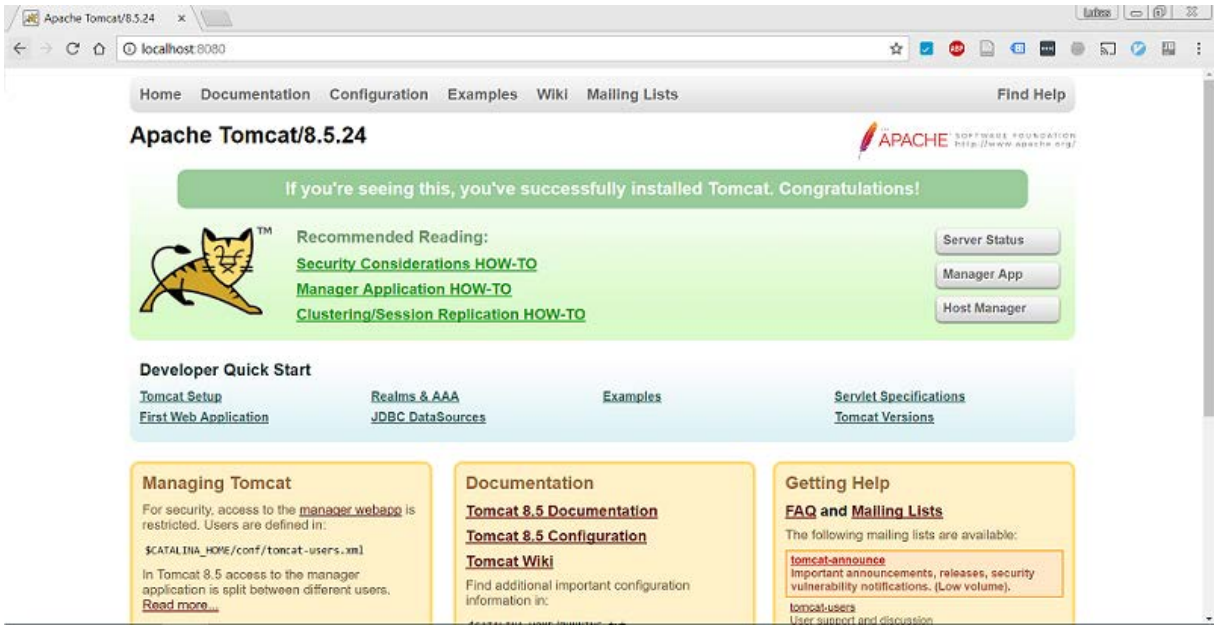
Choose the destination folder of your installation. This folder path is set to the System Variable $CATALINA_HOME. If want to follow the instructions of this paper's examples, set the variable accordingly.



Select “Run Apache Tomcat” and click finish, to start the Tomcat Manager process.

The Tomcat Manager should now be running and as indicated by a symbol in the task bar. A green play button indicates a running Tomcat Service, listening for requests, wheras a red square indicates a stopped status. To change the status or other settings, this can be done in the property windows when double clicking it.





The successful installation can be tested by typing the URL: localhost:8080. This will reveal the ROOT web application of the server, which provides links to informative pages about Examples, Documentation, but also lets you manage servlets. The Manager App can be visited from here using the credentials defined during the installation process.