WIRTSCHAFTSUNIVERSITÄT WIEN
Vienna University of Economics and Business

An Introduction to Web Application Development – Combining Jakarta Server Pages with Programs Written in Scripting Languages

Bachelor Thesis by Dimitry-J. Lux

Supervised by ao.Univ.Prof. Dr. Rony G. Flatscher

Abstract: This thesis aims to communicate all knowledge necessary to enable the reader to develop web applications quickly and efficiently. To achieve this goal, three key tools are used: The Open Object Rexx scripting language, the Bean Scripting Framework for Open Object Rexx and the Apache Tomcat Software. Tag libraries are used to combine these components. After discussing the main technological components, nutshell examples with increasing complexity are used to guide the reader.

Methodology: This thesis commences with a review of the technologies used. Given the nature of the topic, most information has been gathered from online sources, mainly documentations, tutorials as well as selected scientific papers. These core components were then utilized to create nutshell examples, demonstrating possible implementations.

Contents

Figures

Listings

Glossary

API	Application Programming Interface
ASF	Apache Software Foundation
BSF	Bean Scripting Framework
BSF4ooRexx	Bean Scripting Framework for Open Object Rexx
CGI	Common Gateway Interface
CSS	Cascading Style Sheets
DOM	Document Object Model
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IP address	Internet Protocol Address
Jakarta EE	Jakarta Enterprise Edition
JAR	Java Archive
Java EE	Java Enterprise Edition

JDBC	Java Database Connectivity
JDK	Java Development Kit
JNDI	Java Naming and Directory Interface
JRE	Java Runtime Environment
JSP	Jakarta Server Pages
JSR	Java Specification Request
MIME	Multipurpose Internet Mail Extensions
ooRexx	Open Object Rexx
OWASP	Open Web Application Security Project
SMTP	Simple Mail Transmission Protocol
SMTPS	Simple Mail Transmission Protocol Secure
SQL	Structured Query Language
SSL	Secure Socket Layer
Taglib	Tag Library
TLD	Tag Library Descriptor
TLS	Transport Layer Security
Tomcat	Apache Tomcat Software
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
WAR	Web Application Archive
Webapp	Web Application
Windows	Microsoft Windows

1. Introduction

The World Wide Web was invented by Sir Tim Berners-Lee at the European Organization for Nuclear Research. In 1990 he created the first web client and server, as well as the specifications for Uniform Resource Identifiers (URI), the Hypertext Transfer Protocol (HTTP) and the Hypertext Markup Language (HTML) [1]. With HTTP being a stateless protocol [2, p. 1], in the beginning web pages were simple, static sources of information. Today users of the internet are used to performing complex tasks all from within their web browser.

While the task of developing such web applications might seem daunting at first, the tools used in this thesis will allow beginners to quickly create their own dynamic web pages. The Apache Tomcat Software (Tomcat) offers the necessary infrastructure, most notably, the Jakarta Server Pages (JSP) technology. By interlacing HTML code with the human-oriented [3] Open Object Rexx (ooRexx) programming language, programs can be created with minimal prior knowledge. The Sourceforge site of the Bean Scripting Framework for ooRexx (BSF4ooRexx) includes Tag Libraries (taglibs), which can be used to accomplish this. In addition, BSF4ooRexx allows the usage of countless internal and external Java classes, all from within ooRexx. By using these tools for web development, the only limiting factor to creating web applications is one's imagination.

This thesis was written with the ooRexx programming language in mind. While the development of web applications is being covered from the very beginning, basic knowledge of ooRexx and HTML is recommended. Nonetheless, the components used support a multitude of other existing scripting languages.

To begin with, `2. Technologies` will discuss the underlying technologies of the internet and Java web applications. From `4. Introducing Web Applications /helloworld` onwards, nutshell examples are used to demonstrate the concepts discussed. Put together, these examples will form a functioning shopping website. In case the reader would like to use any of the shown code, be it in full or a fragment, the author encourages such usage, in hope it will help.

For a collection of hyperlinks to all the software mentioned and the nutshell examples, please refer to the appendix: `A.1. Software Required to Begin`. This thesis should be accompanied by a `.zip` archive, containing the previously mentioned nutshell examples and necessary support files; the directory `ZIP_ARCHIVE\` is used to reference its contents.

2. Technologies

A Java web application is built upon a Java Runtime Environment provided by a web server and a combination of components such as JSPs, Jakarta Servlets, JavaBeans, and static pages like HTML [4, Sec. 1.2.].

This section will introduce the core technologies used for web development. First the programming languages to create the programs are discussed, followed by the infrastructure that enables them to be accessed over the internet. Should the reader be familiar with these topics, she might wish to jump directly to: 3. Apache Tomcat Fundamentals

2.1. System Programming Languages and Scripting Programming Languages

In 1998 Ousterhout predicted that: "*scripting languages will handle many of the programming tasks in the next century better than system programming languages*" [5, p. 23].

System programming languages were designed to abstract from assembly languages to make the development process faster. While statements of assembly languages correspond directly to machine instructions, system languages require a compiler which translates the source code into binary instructions. Scripting languages abstract even further, with power and ease of use in mind [5, pp. 23-24]. After compilation, a program can be executed multiple times with different input data [6].

Machine language code consists of strings of 1's and 0's which represent the numeric codes for operations that a computer can execute. These binary digits, also called bits, are difficult to read and write for humans and differ between various computer architectures [7].

"*System programming languages were designed for building data structures and algorithms from scratch, starting from the most primitive computer elements such as words of memory. In contrast, scripting languages are designed for gluing: They assume the existence of a set of powerful components and are intended primarily for connecting components*" [5, p. 23].

Scripting languages use interpreters instead of compilers. The translation does not happen all at once, but instruction by instruction [6]. This allows a quicker development process without compile times. Additionally, programmers are more flexible since the applications are programmed at runtime [5, p. 26]. In contrast to system languages, scripting languages are usually kept in source form [8].

"*Scripting languages are higher level than system programming languages in the sense that a single statement does more work on average*" [5, p. 26].

While system programming languages are strongly typed, scripting languages do not share this trait. Typing refers to variables being declared a particular type such as integer or string. A strongly typed language offers performance gains since the compiler only needs to load specific instructions. While potential errors are detected during compile time, errors in scripting languages occur when a value is used. Scripting languages are generally typeless; variables can freely switch data types. This results in the

interchangeability of code and data, easing the process of combining different components. Overall, strongly typed languages are more restrictive and less flexible, yet more performant [5, pp. 24-27].

With the increase of computing power, this performance difference is increasingly negligible. Nonetheless, in case of an application where performance is crucial, a system programming language might be the better choice. This is particularly the case for programs that are slow to change. On the other hand, scripting languages are particularly useful for programs implementing a graphical user interface, connect through the internet or utilize component frameworks like Java Beans [5, pp. 27-28]. Real life enterprise systems are usually made up of many programs working together, like web servers and database servers. Therefore, system administration, web applications and document processing are areas where the application of scripting languages is preferred [8].

2.2. Java

Even though, the Java system programming language is not being directly used for the creation of the example web applications, it is still used as an underlying component throughout this work. Therefore, a basic understanding of its architecture is required.

The main feature of the Java programming language is its architecture-neutral approach. Instead of machine code its compiler creates so called bytecode that runs on the Java Virtual Machine [9, Ch. 4]. The platform is software only and runs on top of hardware or software environments. In addition to the Java Virtual Machine, the Java platform also includes the Java Application Programming Interface (API) [10].

"*The API is a large collection of ready-made software components that provide many useful capabilities. It is grouped into libraries of related classes and interfaces; these libraries are known as packages*" [10].

After source code is written and saved with the `.java` extension, it needs to be compiled into a `.class` file by the `javac` compiler. A `.class` file contains bytecode which then gets read and executed by the Java Virtual Machine [10].

The Java Language Classes, `java.lang`, contain the base types and are always imported into a compilation unit. They include the fundamental classes, such as `object` or the so-called wrapper classes for primitive types like Booleans. The complete Java system also includes the libraries: `java.io`, `java.util` and `java.awt` [9, Ch. 9]. These core libraries enable a huge variety of features, such as network communications, security management or file handling [11].

In addition, external libraries are used to extend the functionality of Java. These reusable bits and pieces can be used to add missing functionality or help a programmer write less code and therefore save time. Beginners can use them to create programs including

features, they would not be able to create by themselves. A library consists of a bundle of packages, which hold Java classes and interface definitions [11].

Notably, a library's application programming interface documentation is one of its most crucial components. `Javadoc` reads the comments in a library's code, to create an API documentation, holding reference information to ease usage [11].

Libraries usually come packaged as `.jar` files [11]. These Java Archives are defined by the `.jar` extension and are based on `.zip` files. They are used to package multiple files together, compressing them to decrease size. While they usually hold multiple `.class` files, whole applications can be packaged this way, also including pictures and audio files [12].

2.3. Java and Scripting Languages

Generally, code written in a scripting language can be compiled into Java bytecode, enabling its execution on the Java Virtual Machine.

2.3.1. JSR-223

The Java Specification Request (JSR) 223 was released at the end of the year 2006 [13]. It enables the embedding of scripts in Java applications and the access of Java objects from within scripts. A script written in compliance with JSR-223 can access the entire standard Java library. Equally important, a Java application written with JSR-223 in mind allows the embedding of scripts without the need to specify a scripting language [14, Sec. 1].

"A program specification describes the results that a program is expected to produce -- its primary purpose is to be understood not executed. Specifications provide the foundation for programming methodology" [15].

The Java Scripting API is defined by JSR-223 and comes included with the Java Standard Edition since version 6 [16]. Its classes and interfaces can be found in the `javax.scipt` package. It contains the `ScriptEngineManager` class, which discovers script engines using the `.jar` file service discovery mechanism. After discovery, a `ScriptEngine` object gets instantiated to perform interpretation [14, Sec. 2]. The `ScriptEngine's eval` method can then be used to execute a script that has been given as input parameter, returning the resulting value [17].

2.3.2. Bean Scripting Framework

"*JavaBeans are classes that encapsulate many objects into a single object (the bean)*" [18].

The Bean Scripting Framework resulted from a research project of Sanjiva Weerawarana at IBM in 1999. Its goal was to access JavaBeans from scripting language environments. The project continued as an open-source project at IBM before it was donated to the Apache Software Foundation at version 2.3 [19].

"*Bean Scripting Framework (BSF) is a set of Java classes which provides scripting language support within Java applications, and access to Java objects and methods from scripting languages*" [20].

The framework's two main classes are the `BSFManager` and the `BSFEngine` [20]. The `BSFManager` class gets instantiated when an application decides to run a script. It is then used to register beans, load script engines, and run scripts. Furthermore, the `BSFManager` registers all available script engines, loads, and unloads them. Each Java Virtual Machine can run multiple `BSFManagers` but each `BSFManager` can only load one engine per language [21]. The `BSFEngine` abstracts a scripting language's capabilities and allows generic handling of script execution and object registration within the execution context of a given language [20].

Releases under the newer version 3.x use the JSR-223 API [22]. The Open Object Rexx programming language is supported with its own BSF engine: BSF4ooRexx.

2.4. Open Object Rexx

All the nutshell examples accompanying this thesis have been created using the ooRexx scripting language. Even if the reader is not familiar with this language, its easily understandable syntax might help with the development of web applications in another language.

Initially developed in 1979 by Mike F. Cowlishaw, the Rexx programming language aimed to make the programming of IBM mainframes easier to understand and more human centric. After gaining popularity in the industry, the language evolved in 1997 by implementing object-oriented features, resulting in the IBM product Object Rex. In 2004 the source code was given to the Rexx Language Association, which released the first open-source version, called Open Object Rexx. This powerful yet extensible language is available for all major operating systems [23, pp. iii-v].

2.5. Bean Scripting Framework for Open Object Rexx

In 2001, the Bean Scripting Framework for Rexx was first introduced at the 12th International Rexx Symposium by Rony G. Flatscher. In this first iteration, based on a seminar paper by Peter Kalender, the Rexx and Object Rexx interpreters were incorporated into the BSF framework. Henceforth, it was possible for Java programs to cooperate with and invoke Rexx and Open Rexx programs [24, p. 5]. Two years later, a further improved version with the ability to start Java from Rexx programs, was presented. It enabled the usage of Java as a Rexx function library [25, p. 5]. In 2009, the first BSF4ooRexx version was released, implementing new features made possible by native ooRexx APIs introduced with ooRexx 4.0 [26, p. 4].

One of BSF4ooRexx's main achievements is to camouflage Java, allowing the ooRexx user to utilize Java class objects without requiring extensive Java knowledge. The user

can send messages to so-called proxy classes, which will be forwarded to the Java object they represent. This is achieved by the Bean Scripting Framework supporting module `BSF.cls`. It constructs an object-oriented interface to the Java Runtime Environment, enabling access to features such as Java arrays [27, pp. 13-20].

2.6. Hypertext Transfer Protocol

The focus now shifts on technologies enabling programs to communicate over the internet. Before a client and a server can start communicating, they need to agree on common rules for data transmission and the information's structure. These rules are established in form of a protocol [28].

"A web server stores and delivers the content for a website – such as text, images, video, and application data – to clients that request it. The most common type of client is a web browser program, which requests data from your website when a user clicks on a link or downloads a document on a page displayed in the browser" [29].

The most popular protocol of the Internet, the Hypertext Transfer Protocol, is an asymmetric, stateless pull protocol, running on the application layer. The client sends a request and gets a response from the server. This request is most often based on a Uniform Resource Locator, which the browser converts to a request [30].

"A URL (Uniform Resource Locator) is used to uniquely identify a resource over the web. URL has the following syntax: protocol://hostname:port/path-and-file-name" [30].

The protocol typically runs over a TCP/IP connection, but also allows other reliable transport methods. Given its stateless nature, requests are not connected, and are not aware of previous communications. The negotiation of data type and representation systems are independent from the way the data is transferred [30].

In addition to the `get` request method, `post` is used to send data to the webserver, while `delete` requests its deletion [30]. A typical HTTP request can be seen in the image below, where `get` is used to request the Tomcat 10 documentation web page.

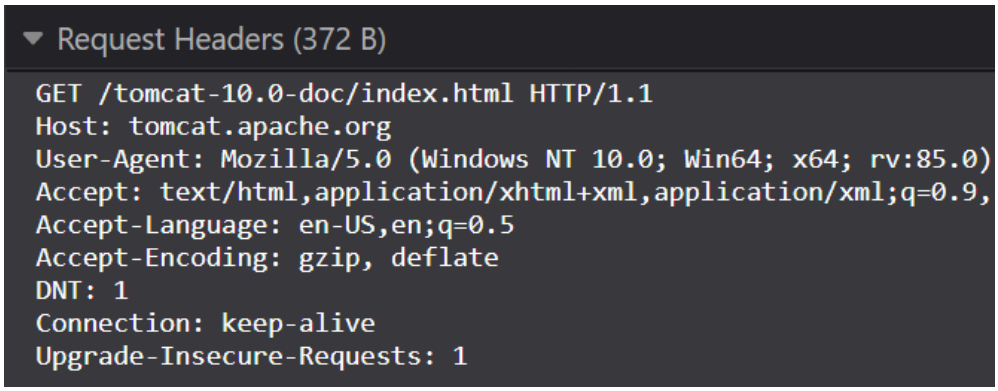

Figure 1: HTTP Request Header

First, the request line specifies the protocol and which resource is requested from the specified host. Afterwards, headers inform the contacted server about what type of files can be received, as well as other information, like the preferred language and the user's browser, which is referred to as User-Agent.

After the request is sent, the server replies with a response message. As can be seen in the figure below, instead of a request line it begins with a status code. For example, the Code 404 means that the requested resource cannot be found [30].

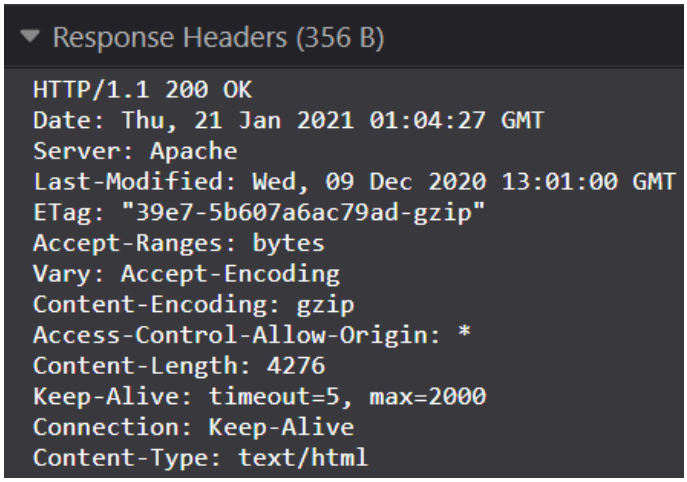

Figure 2: HTTP Response Header

While the response header also includes additional information like the length of the content sent, the message's body contains the requested resource.

2.7. Hypertext Markup Language

While first intended to describe scientific documents, the Hypertext Markup Language soon became the core markup language of the World Wide Web [31, Sec. 1.1.].

HTML is used to define the structure of a web page. A tree of elements, such as `<head>`, `<body>`, or `<p>` is used to describe how a document is to be displayed [32]. Each element consists of a starting and an end tag, with the content displayed in between: `<Tag>Content</Tag>`. Additionally, elements can have attributes placed in their start tag: `<form method="post">` [31, Sec. 1.9.].

It is important to note that: "*HTML documents represent a media-independent description of interactive content. HTML documents might be rendered to a screen, or through a speech synthesizer, or on a braille display*" [31, Sec. 1.9.].

When a web browser parses such a document, it transforms it into a Document Object Model (DOM) tree and stores it in memory. While this representation of a web page is static in nature, scripts can be used to manipulate it [31, Sec. 1.9.].

The presentation of such a document can be altered using Cascading Style Sheets (CSS): *"The CSS1 language is human readable and writable, and expresses style in common*

desktop publishing terminology" [33]. CSS rules can be either applied by an external file, or within the HTML document itself, in form of inline styling.

2.8. Jakarta Servlets

A servlet is a program running inside a web server that creates a customized response for each incoming HTTP request. For example, after a user has filled in a form, a web page will be tailormade according to the input parameters. Another example would be the creation of a user-specific webpage, displaying data from a database or time sensitive information, such as stock quotes. As a result, the response is not the same for each request, but changes dynamically [34, Sec. 1.]. This is achieved by mapping Java programs to client requests that supply an URL. The servlet specification defines how these programs need to be structured [35, p. 2].

"Java servlet is the foundation of the Java server-side technology, JSP (JavaServer Pages), JSF (JavaServer Faces), Struts, Spring, Hibernate, and others, are extensions of the servlet technology" [34, Sec. 1.]. The advantages of the mature Java programming language can be fully accessed: server- and platform-independency, reusability, portability, and high performance [36].

The necessary run time environment for a servlet is provided by a Java Server. Additionally, it handles all networking services and resources necessary, while also managing the life cycle of servlets [37]. For example, it decodes and formats mime-based requests and formats mime-based responses [38, Sec. 1.2.]. The Multipurpose Internet Mail Extension (mime) is used to specify the subtype, encoding and media type of data [39, Sec. 1.]. Security is enhanced by the servlet being part of the web server, and therefore inheriting its security measures [37]. Jakarta Servlets of the Version 5.0 support HTTP/1.1 and HTTP/2.0. It is important to note that the server can modify HTTP requests before and after the processing of the servlet to allow caching [38, Sec. 1.2.].

When creating a servlet, one can either extend the `jakarta.servlet.GenericServlet` interface or the `jakarta.servlet.http.HttpServlet` interface. The more specific `HttpServlet` includes methods supporting the HTTP protocol like `doGet`, which handles HTTP `get` requests [40]. The `init` method represents the beginning of a servlet's lifecycle. It then handles all client calls with the `service` method, before retiring with the `destroy` method [41]. By default, only a single instance is created for each servlet declaration [38, Sec. 2.2.].

2.9. Jakarta Server Pages

"*A JSP page is a text-based document that describes how to process a request to create a response*" [4, Sec. Overview].

Jakarta Server Pages are closely related to servlets and are built based on their specification [42]. The most common application of JSPs is in the form of HTML and XML content. They enable concepts like web applications, servlet contexts, sessions, requests, and responses [4, Sec. Overview]. While HTTP is the default protocol for requests and responses, other protocols are also accepted, if the container supports them [4, Sec. 1.1.].

"*From a coding perspective, the most obvious difference between them is that with servlets you write Java code and then embed client-side markup (like HTML) into that code, whereas with JSP you start with the client-side script or markup, then embed JSP tags to connect your page to the Java backend*" [42].

Before being requested by clients during the request phase, JSPs need to be translated to a servlet class by the container. The result is referred to as a JSP implementation class. Translation is performed once per page and can take place on request or at deployment time. After the class is instantiated at request time, responses are created for corresponding incoming requests. [4, Sec. Overview]. In short, after translation the JSP will be indistinguishable from any other servlet.

Most significantly, JSPs functionality can be extended by Tag Libraries. Within the libraries, tag handlers, implementing the `BodyTag` interface class are found [35, p. 10]. They introduce custom actions, to be used manually or by Java development tools [43].

The Tag Library Descriptor (TLD) is used to describe the Tag Library in form of an XML file and uses the extension `.tld`. It allows JSP containers to interpret pages that use a tag library. Additionally, a `TagLibraryValidator` class can be used to check whether a JSP page is valid according to a set of expected constraints [4, Sec. 7.3.].

JSPs can contain fragments written in a scripting language, which are referred to as scriptlets [44]. While Java is the default scripting language, other languages can be added using two different methods. First, new languages can be declared at the beginning of a JSP. However, some containers only support Java to be used this way. Secondly, a tag library can be used to enable scripting language support by means of a custom action. This approach is beneficial since it is portable between containers, all of which must support the tag extension mechanism. Equally important, this approach allows using multiple different scripting languages on the same page [45, p. 34].

2.10. Apache Tomcat

"*Apache Tomcat is a long-lived, open source Java servlet container that implements several core Java enterprise specs, namely the Java Servlet, JavaServer Pages (JSP), and WebSockets APIs*" [46].

In 1997 the American software developer James Duncan Davidson started to work as an engineer for JavaSoft, which at the time was a part of Sun Microsystems. While working on the Java Web Server he created a reference implementation for the Java Servlet

specification, called the Java Servlet Web Development Kit [47]. In 1999 the project was donated to the Apache Software Foundation and was thereafter called Tomcat. In 2005 Tomcat became a top-level Apache project to be managed by itself [48]. Tomcat refers to multiple components working together, mainly Catalina, Jasper and Coyote.

"*Catalina provides Tomcat's actual implementation of the servlet specification; when you start up your Tomcat server, you're actually starting Catalina*" [49]. This is also the reason why Tomcat's home/installation is often referred to as `CATALINA_HOME` [50]. The Java class `Catalina` not only provides the servlet container's main functionality, but is also responsible for its configuration, security, and logging [49].

The Jasper JSP engine is used to implement the Jakarta Server Pages specification. It compiles JSPs into Java code to be used by Catalina as servlets. It can detect changes at runtime and consecutively recompile a JSP [51]. Therefore, any changes made become immediately visible.

The Coyote HTTP/1.1 Connector enables Tomcat to work as a stand-alone web server. It gets instantiated and listens for connections on a specified TCP port number [52]. The current implementation supports the HTTP/2 protocol, using the class `org.apache.coyote.http2.Http2Protocol` [53]. Without this class, a dedicated HTTP server like the Apache HTTP Server would be required. If needed, Tomcat can be connected to an Apache HTTP server by means of a connection module [54].

Since Tomcat is often also referred to as a web server, it is important to establish the difference between a webserver and a web container. A web server is used to store and deliver web pages to clients. To accomplish this, the HTTP protocol is used, which also allows the transmission of information provided by clients [55]. JSPs and servlets are referred to as web components, to be used, they require an environment provided by a web container [4, Sec. 1.1.1.]. Therefore, every web container can be referred to as a web server, while not every web server is a web container. Furthermore, a web container utilizing the Jakarta Servlet API can be referred to as a servlet container.

2.11. Open-Source Software

Most of the software mentioned throughout this thesis is open-source, meaning that its source code is freely available for anyone to inspect, modify and enhance. This approach not only allows control and security but also the creation of a community [56].

"*Open source projects, products, or initiatives embrace and celebrate principles of open exchange, collaborative participation, rapid prototyping, transparency, meritocracy, and community-oriented development*" [56].

Open-source software is usually released under a license, stating the terms of usage, modification, and redistribution. Copyleft licenses, like the GNU General Public License, allow the creation of derivative works, but require them to keep using the same license. In contrast, permissive licenses allow the user to freely use, modify and

redistribute the software [57]. The Apache license is an example for a license that is permissive.

2.11.1. Apache Software Foundation

The Apache Software Foundation is a not-for-profit corporation established in 1999. An all-volunteer board oversees over 350 open-source projects and supports them with a framework for intellectual property and financial contributions [58].

"The mission of the Apache Software Foundation (ASF) is to provide software for the public good" [58].

The Apache Tomcat Software is released under the Apache License version 2.0 [59]. At the time of writing, in January 2021, the current version of the Apache License is 2.0, which was approved in the year 2004. Most importantly, software released under this license can be reproduced and used to create derivates without paying royalties. To offer legal protection, the license is revoked should an entity file a lawsuit claiming patent infringement by the creator or a contributor [60]. All the example programs created for this thesis are licensed under the Apache License 2.0. Therefore, should the reader deem them useful, free usage is encouraged.

2.11.2. Eclipse Foundation, Jakarta Namespace

The Eclipse Foundation is a not-for-profit organization with over 275 members, focusing on open-source projects [61]. It was initially created by IBM in the year 2001 [62].

"The Eclipse Foundation provides our global community of individuals and organizations with a mature, scalable, and business-friendly environment for open-source software collaboration and innovation" [61].

Until 2019, Jakarta Servlets and Jakarta Server Pages were officially known as Java Servlets and Java Server Pages. This name change goes hand in hand with the rebranding of Java Enterprise Edition (Java EE) to Jakarta Enterprise Edition (Jakarta EE), the set of specifications under which the Java Servlet specification was originally released. In the year 2017, the then owner of Java Enterprise Edition, Oracle, decided to donate it to the Eclipse Foundation. These events resulted in the rebranding, using a new name chosen by the community. On September 10, 2019 the first version using the Jakarta namespace was released, Jakarta EE8. The release changed all the included specifications names, including the Java Servlet, which is referred to as Jakarta Servlet from this point onward [63].

While the Standard Edition of the Java Platform enables the development and deployment of desktop and server applications, its Enterprise Edition is focused on large, multi-tiered enterprise applications [64]. "*The focus of the Jakarta EE platform is not to bundle a bunch of unrelated APIs. The purpose of Jakarta EE is to ensure that a*

variety of useful enterprise APIs work in harmony" [65]. Most noteworthy, Java web servers are defined as standardized services in this bundle [35, p. 1].

TomEE is a version of Tomcat, that builds on the standard edition, by adding all the API's composing Jakarta EE. Only a single of those API's is required for the web applications accompanying this thesis, namely Jakarta Mail. To simplify, it has been added to the standard Tomcat version as a standalone `.jar` file [65]. Alternatively, TomEE could have been used instead.

Tomcat 10 was the first version to implement this change in namespace, which is reflected in the naming of all primary packages [66]. To give an example, the cookie class is referred to as `jakarta.servlet.http.Cookie` in Tomcat version 10 and `javax.servlet.http.Cookie` in Tomcat version 9.

This namespace change is directly related to this thesis, since at the time of writing, in early 2021, the Apache Tomcat Software Version 10 was still in its Beta phase. Nonetheless, to future proof this work, it is fully based on Tomcat 10. Should the reader prefer the stable version Tomcat 9, most components should be near identical. The exception being, that, instead of the examples found directly in the zip archive, the ones in the directory `ZIP_ARCHIVE\javax_for_Tomcat09` need to be used instead. They are near identical, the only difference being the above-mentioned name change, realized in the tag library used and the cookie class. For more information on the usage of Tomcat 9 please refer to the appendix: `C. Using Tomcat 9`

2.12. Bringing It All Together

The conventions and structure of Java web applications are used in tandem with the Apache Tomcat Software to provide the necessary infrastructure for web development. The web container handles all incoming HTTP requests and outgoing HTTP responses. Meanwhile, the logic necessary is implemented by JSPs, containing custom tags that enable code written in a scripting language.

These tags originate from one of two tag libraries created by Rony G. Flatscher for BSF400Rexx, using either the BSF or the JSR-223 framework. Originally based on two, now deprecated, tag libraries, they were released in the fall of 2020 in the form of `jakarta.ScriptTagLibs.jar` and `javax.ScriptTagLibs.jar` [35, pp. 10-11]. Scripts that are invoked this way are supplied with the implicit objects normally available to a standard Java Scriptlet inside of a JSP. Most notably `request`, `response` and `out` [45, p. 35]. These objects give programs the ability to interact over the internet.

Although, this approach allows the usage of many different scripting languages, ooRexx has been selected as the programming language of choice. The necessary script engine, `RexxScriptEngine`, is made available by BSF400Rexx [67, p. 5]. Additionally, BSF400Rexx enables the inclusion of countless external Java libraries.

Hence, the script runs on the server, generating dynamic content based on the request sent by the user. This technique is also referred to as server-side scripting. Whereas, the client is not required to support the scripting language used, increased latency might be a disadvantage for some applications [68]. After processing, the user receives a HTML document that gets rendered by a web browser.

3. Apache Tomcat Fundamentals

The following chapter introduces the Apache Tomcat Software, communicating all knowledge required to run the complementary example web applications included with this thesis. At this stage, ooRexx, BSF4ooRexx and Tomcat should be installed. Download links for the first two components can be found in the appendix: `A.1. Software Required to Begin`, as well as a detailed installation guide for Tomcat: `B. Tomcat Installation Guide`

3.1. TOMCAT_HOME

The author uses `TOMCAT_HOME` to describe Tomcat's home/installation directory. The figure below shows its contents.

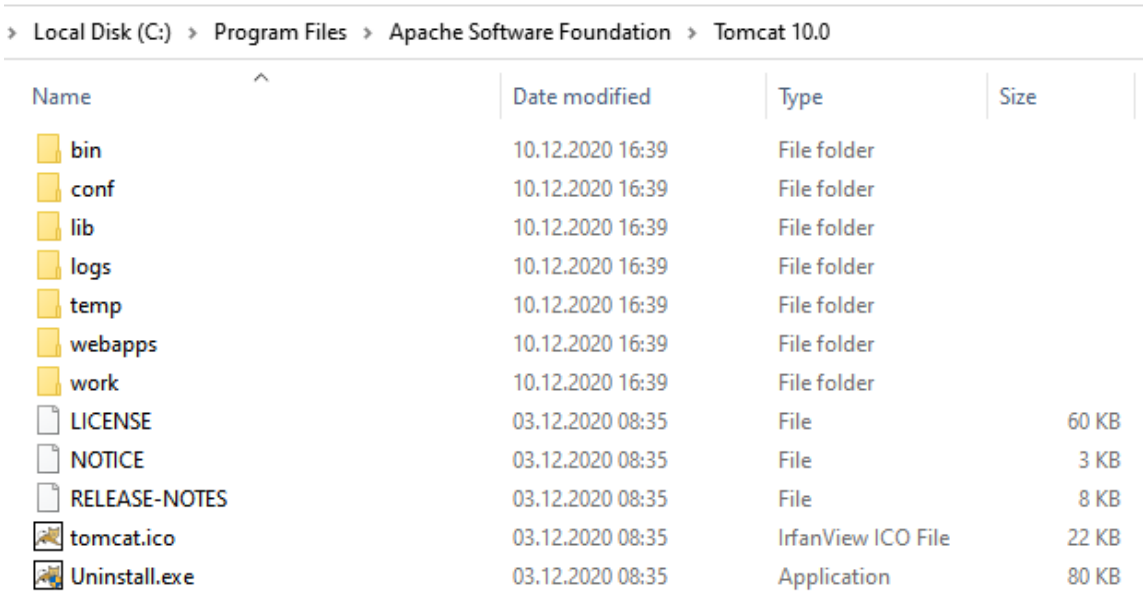

Figure 3: TOMCAT_HOME Directory

`TOMCAT_HOME\bin` contains scripts in the form of `.bat` files. Mainly, `startup.bat` and `shutdown.bat`, which can be used to start and stop the server.

`TOMCAT_HOME\conf` holds multiple files used to configure the software's properties. The `server.xml` file is used to change the initial server configuration on startup, for example it points to external static resources. The file `web.xml` is used to deploy and configure web applications. The files in this folder serve as a default, for certain parameters to be overwritten by a `web.xml` file specific to a web application [69].

`TOMCAT_HOME\lib` contains `.jar` files that are shared among all web applications. The files placed in this directory are not only accessible to all web applications but are also used by Tomcat itself. Files that are necessary for basic functionality, like `catalina.jar` and `jasper.jar` come preinstalled [70].

`TOMCAT_HOME\logs` contains log-files, useful for debugging and testing self-written web applications. Particularly, for each day the server is run, a file called `tomcat10-stderr.yyyy-mm-dd.log` is created, containing error messages. This file is particularly useful to detect the cause of exceptions. Also, it might prove useful to regularly delete old log files to quickly find relevant entries. To make deletion possible, the Tomcat server needs to be shut down. For convenience, all files within the log folder can be deleted since Tomcat will automatically recreate all necessary files during the next startup.

`TOMCAT_HOME\webapps` is the directory where all web applications can be found. Depending on the installation parameters chosen, this folder might already come shipped with default applications.

`TOMCAT_HOME\work` is used by Tomcat for intermediate files during runtime. For example, once a JSP is compiled, the result is placed here [71].

3.2. Deploying Web Applications

"Deploying your application means putting it on a Web server so that it can be used either through the Internet or an intranet" [72].

To begin with, the two demo web application shipped with this work need to be deployed and made accessible. There are two ways to accomplish deployment, either by deploying a web application exploded or in the form of a web archive file.

Web Application Archives use the `.war` file extension and contain all necessary files for a web project. Everything that is needed such JSPs, scripts and the configuration files are contained in a single archive. They are quite like `.jar` files and can be created from the command line with the jar tool included in the Java Development Kit. For example, the command `jar -cvf projectname.war *` will create a web archive from all the files found in a directory [73].

The usage of `.war` files is especially convenient because they use the `.zip` format [74]. Instead of using the command line, it is also possible to create a simple `.zip` archive and giving it the `.war` file extension. To view its contents, `.war` files can also be unpacked by any compression software.

Lesson Learned: When web applications are shipped as `.war` files, all required files are expected to be already included. Special attention needs to be given to `.jar` files.

After placing a `.war` file in `TOMCAT_HOME\webapps`, within a single minute, the software will automatically unpack the files in a new folder of the same name. Afterwards, all files

can be conveniently viewed. Once a `.war` file is unpacked it is considered exploded. Similarly, after deleting a `.war` file, Tomcat will automatically undeploy the corresponding web application.

When web applications are in development, they are usually deployed exploded. A folder in the `webapps` directory is created and the files inside are modified without the need to compress them into a single file. Since any changes made on a JSP are immediately reflected on the corresponding web page, it is possible to edit them while the server is running, and the web application is deployed.

Furthermore, all web servers compliant with Jakarta EE, handle web applications the same way, allowing identical `.war` files to be used with different Java webservers, like IBM WebSphere. They all handle the `.war` files as an independent application, using its directory as a virtual root [75]. Therefore, any concepts used for web application development with Apache Tomcat can be directly transferred to other Java web servers.

While `.war` files contain multiple `.jar` files, `.ear` files contain multiple `.war` files. This format used by the Jakarta EE platform to create application packages [76].

At this point the reader is encouraged to copy the `.war` files `helloworld.war` and `treeshop.war` to `TOMCAT_HOME\webapps`. While the web application called `helloworld` will be the subject of the next sections, `treeshop` will be discussed at a later stage. The files can be found in the root of the complementary archive. In case the reader prefers to use Tomcat 9, the files found in the directory `ZIP_ARCHIVE\javax_for_tomcat9` need to be used instead.

3.3. Running and Stopping Tomcat

There are multiple ways to start the Apache Tomcat software. The previously mentioned scripts `startup.bat` and `shutdown.bat` found in `TOMCAT_HOME\bin` exist for all platforms. On some operating systems, they might have the file extension `.sh`.

On the Microsoft Windows (Windows) operating system the Apache Commons Daemon Service Manager, which creates a taskbar icon, is run from the start menu entry `Monitor Tomcat`. It can be found in the folder `Apache Tomcat 10.0 Tomcat10` and offers a convenient way to configure, start and stop the server.

Figure 4: Apache Commons Daemon Service Manager Taskbar Icon

The reason why Tomcat cannot be started by conventional methods lies in its nature as a service. On Windows, Windows services run in their own Windows session and are used for applications that require long-running functionality, without interfering with other users on the same machine. Additionally, they allow a different security context [77].

Therefore, yet another way to control the status of Tomcat is accessed by typing `services.msc` in the Windows Powershell or the Command Prompt. A list of all services will be displayed. By right-clicking on `Apache Tomcat 10.0 Tomcat10`, the server can be started and stopped. Furthermore, the commands `sc` and `net` can be used to control Windows services from the command line.

Once running, Tomcat can be reached from the URL: `http://localhost:8080`

Localhost is a top-level domain, referring to the current computer and is interchangeable with the Internet Protocol address (IP address) `127.0.0.1`. The number 127 at the beginning of the address triggers a so-called loopback; the request is not forwarded to the internet but handled by the local computer instead. This feature is mainly used by administrators and for testing purposes [78].

Ports are interfaces on a computer to which other devices can connect to, facilitating communication. The ports are numbered starting from 0 to 65535. Ports numbered 0 to 1023 are also called well-known ports, which are reserved for common services like the HTTP protocol, which has the port 80 assigned to it [79]. During the Tomcat installation process, the HTTP connector port got assigned to 8080. Here Tomcat's Coyote component is listening for incoming requests.

In case the reader has defined a different port during installation, the URL needs to be changed accordingly. Any differences concerning the usage of Tomcat 9 are in name only.

3.4. Tomcat Manager

Among other features, the Tomcat Manager gives an overview of all installed web applications and allows them to be deployed, undeployed and reloaded, all without necessitating a restart [80]. This is particularly useful for environments where multiple people work together and for users preferring a graphical interface.

If it has been selected during installation, the Tomcat Manager can be accessed from: `http://localhost:8080/manager`

When started, the application asks for the username and password given during the installation. The users and their passwords are defined in the file `tomcat-users.xml` which can be found in `TOMCAT_HOME\conf`.

4. Introducing Web Applications `/helloworld`

Thus, after introducing the fundamentals of working with Tomcat, the web pages contained in the application `helloworld` will introduce the reader to web application development.

At this point, the file `helloworld.war` should have been placed in `TOMCAT_HOME\webapps`. During deployment of the software, the web archive's contents are exploded

automatically. The folder structure directly influences the path from which pages are accessed: Files in the directory `TOMCAT_HOME\webapps\helloworld`, are accessed from the URL: `http://localhost:8080/helloworld`

The directory `TOMCAT_HOME\webapps\helloworld` is referred to as the application's context path [74]. To allow generalization across different web applications, this thesis uses the path `WEBAPP\` to refer to this directory.

4.1. Web Application Architecture

Some elements are common to all Java based web applications. The directory `WEBAPP\WEB-INF` contains all resources necessary to run an application. Typically it holds `.jars`, `.tlds` and the `web.xml` file. Notably, resources contained in this folder are not made accessible to web users [81].

The `web.xml` file contains the Web Application Deployment Descriptor. It is used by the JSP container to gather general configuration information [4, Sec. 3.1.]. The main `web.xml` file can be found in `TOMCAT_HOME\conf`, while the version specific to a web application is located at `WEBAPP\WEB-INF`. The latter is used in case deviating or additional configuration parameters are required. [82]. For example, it holds information used to name and describe web applications in the Tomcat Manager. At a later stage, this file will be used to add application specific configuration parameters.

Should a tag library be used, matching Tag Library Descriptors are essential to the functioning of a web application. To minimize potential errors and to showcase their interchangeability, both Tag Library Descriptor files for the JSR-223 (`script.jsr223.tld`) as well as the BSF (`script-bsf.tld`) tag library were placed in `helloworld\WEB-INF`.

The directory `WEBAPP\WEB-INF\lib` contains Java `.class` files in `.jar` archives. Like the `web.xml` file, the contained libraries are specific to the web application and take precedence over any classes loaded from `TOMCAT_HOME`.

For the web applications shipped with this thesis to function, two Java Archives are always needed. First, the file `jakarta.ScriptTagLibs.jar` holds the actual BSF and JSR-223 tag libraries. Tomcat 9 users will find the file `javax.ScriptTagLibs.jar` instead. The `bsf4ooRexx-v641-20210205-bin.jar` includes the Bean Scripting Framework, the bridge between Java and ooRexx.

This leaves the question, whether to place the classes necessary for a web application in `TOMCAT_HOME\lib` or `WEBAPP\WEB-INF\lib`. For the application `helloworld`, the author has chosen to package all necessary `.jar` files in `helloworld\WEB-INF\lib`. Therefore, the reader is not required to make any additional changes after deployment.

Generally, the benefit of not requiring the user to modify her Tomcat installation outweighs the redundancy of having multiple identical `.jar` files. As a result, `helloworld`

run effortlessly after being placed in the `webapps` folder. Nonetheless, other factors complicating this issue will be discussed later.

4.2. Introducing Jakarta Server Pages `/helloworld/helloworld.jsp`

The listing below gives an overview of the document `helloworld.jsp`, found in `helloworld\`. At first glance, the Jakarta Server Page is almost identical to a standard HTML page. By interweaving static and dynamic content the JSP gets transformed into a Rexx Server Page.

```
<%@ page session="false" pageEncoding="UTF-8" contentType="text/html; charset=UTF-
8" %>
<%@ taglib uri="/WEB-INF/script-jsr223.tld" prefix="s" %>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>hello, world</title>
</head>
<header>

<s:script type="rexx">
USE ARG request, response, out

greeting = "Hello, world! (Sent from Open Object Rexx)"
out~println(greeting)
</s:script>

</header>
</html>
```

Listing 1: helloworld.jsp

4.2.1. JSP Directives

All JSPs share a common set of characteristics and begin with the so-called directives, containing messages to the JSP container. All directives follow the syntax: `<%@ directive attr="value" %>`. The three existing directive types are `page`, `taglib` and `include` [4, Sec. 1.10.].

The `page` directive is used to communicate page dependent properties to the JSP container. It can occur multiple times and at any position in the document, except for the `pageEncoding` and `contentType` attributes, which are expected to appear at the beginning. Attributes are limited to a single instance, except for `import` and `pageEncoding` [4, Sec. 1.10.1.].

```
<%@ page session="false" pageEncoding="UTF-8" contentType="text/html; charset=UTF-
8" %>
<%@ taglib uri="/WEB-INF/script-jsr223.tld" prefix="s" %>
```

Listing 2: helloworld.jsp JSP Directives

To begin with, the `session` attribute of the `page` directive is used to identify a user across multiple requests. The identification is made possible by means of a cookie, or

alternatively by rewriting the URL [83]. This simple web page, with the only goal of displaying information requires no session.

The `pageEncoding` attribute determines the encoding of the JSP itself, while the `contentType` attribute defines the mime-type and character encoding of the response. Additionally, the character encoding can be defined by `charset` [4, Sec. 1.10.1.].

Encoding is particularly important for webpages since they might contain text in many different languages. Characters on computers are stored as bytes, which need to be mapped to characters using a specific code. The characters in this context are grouped into character sets. Many different character sets exist for different purposes and languages; for the use case of creating a web page, the Unicode `UTF-8` is recommended. `UTF-8` includes a multitude of characters, for almost any possible situation, making it unnecessary to switch or convert between encodings throughout a project [84]. Furthermore, it ensures maximum compatibility with different languages. If the `pageEncoding` is not explicitly declared, `ISO-8859-1` will be used instead [4, Sec. 4.1.1.].

In the second line, the `taglib` directive declares that a tag library is used to extend the page's functionality. The `uri` attribute points to the Tag Library Descriptors exact location in the directory `WEBAPP\WEB-INF`. The declared `prefix` attribute `s` is used to indicate the usage of one of the library's custom actions throughout the document [4, Sec. 1.10.2.].

The `include` directive is used to insert text, data, or code of a specified resource at JSP translation time [4, Sec. 1.10.3.]. In this example the directive has been omitted.

4.2.2. JSP Main Content

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>hello, world</title>
</head>
<header>
```

Listing 3: helloworld.jsp HTML Start Tags

The main contents of the JSP follow the directives. The `<!DOCTYPE html>` declaration informs the browser about the nature of the document and that its author is following HTML 5. Having the `doctype` declaration at the beginning of a web page is good practice and a sign of quality [85]. Similarly, it is a good idea to declare the charset as `UTF-8` once again. The previously declared `charset` attribute, found in the directives, is sent in the HTTP response header. Should the server configuration change or the page gets saved locally, the HTTP response header would be missing [86]. For this situation, the charset gets declared again. Even though, `UTF-8` is the standard charset that will be applied to any HTML5 page in case none is given, the web browser's behavior is not guaranteed, especially for older web browsers [87].

```
<s:script type="rexx">
USE ARG request, response, out

greeting = "Hello, world! (Sent from Open Object Rexx)"
out~println(greeting)
</s:script>
```
Listing 4: helloworld.jsp Script Tag

Afterwards, scripting code is used to display a message in the document's header. The dynamic content starts with the previously declared taglib prefix `s`. The attribute `type` defines the scripting language used, in this case Open Object Rexx. If the JSR-223 taglib is used, this attribute also allows to supply a mime-type or a file-extension [35, p. 11].

Alternatively, many other scripting languages could be used. For example, the addition of the file `jython.jar` would allow the insertion of code written in the Python programming language in place of ooRexx [88, p. 19]. In case needed, this implementation even allows to mix many different scripting languages and Java, all on the same JSP [35, pp. 22-23].

At the beginning of the script, the objects `request`, `response` and `out` are fetched. These objects are part of the implicit objects, nine of which are created by the JSP engine during translation phase [89]. Invoking scripts by means of the tag libraries developed by Rony G. Flatscher, supplies the three mentioned implicit objects automatically as arguments, merely requiring them to be fetched. With ooRexx, this is done with the instruction `USE ARG` [35, p. 23].

The `requests` object provides data the client has transmitted when initially requesting the page, usually it originates from forms. The `response` object modifies or delays the response that is sent in return. The third fetched object `out`, is responsible for writing content to the HTML page the user receives. Furthermore, it enables the formatting of messages [37].

After fetching the implicit objects, the script defines a greeting string and stores it in the variable `greeting`. The `out` object refers to an instance of the Java class `JspWriter`. Next, its `println` method is used to print the characters and terminating the line afterwards [90]. As a result, the greeting previously defined will be displayed in the HTML page header. The closing tags conclude this first script. Since no HTML tags have been given, the `println` method prints the sentence in verbatim without any formatting applied.

The following figure showcases the HTML document the user receives when requesting `helloworld.jsp` from: `http://localhost:8080/helloworld/helloworld.jsp`

Figure 5: hellworld.jsp in Web Browser

As can be seen, the resulting page looks like a standard web page, the parts generated by the script are indistinguishable from the static HTML parts.

```
1
2
3  <!DOCTYPE html>
4  <html>
5  <head>
6  <meta charset="UTF-8" />
7  <title>hello, world</title>
8  </head>
9  <header>
10
11 Hello, world! (Sent from Open Object Rexx)
12
13
14 </header>
15 </html>
16
17
18
```

Listing 5: helloworld.jsp Generated HTML Code

Lesson Learned: For URLs, upper- and lower-casing matters, they need to reflect the JSP's exact name.

From this point onwards, the contents of `WEBAPP\WEB-INF`, as well as the page directives and HTML code up to the header can be copied and reused as standard building blocks. The next example page, `helloworld_ext.jsp`, builds on the first.

4.3. BSF Taglib, Expressions, Styling `/helloworld/helloworld_ext.jsp`

```
<%@ page session="false" pageEncoding="UTF-8" contentType="text/html; charset=UTF-
8" %>
<%@ taglib uri="/WEB-INF/script-bsf.tld" prefix="s" %>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<link rel="stylesheet" href="css/treeshop.css">
<title>hello, world</title>
</head>
<header>

<s:script type="rexx">
greeting = "Hello, world! (Sent from Open Object Rexx)"
```

```
SAY '<h1>'greeting'</h1>'
</s:script>

</header>
<body>

<p>The time right now: <s:expr type="rexx">TIME()</s:expr></p>

<s:script type="rexx">
SAY '<p style="color:blue" !important>'pp("This paragraph is made possible by the
BSF taglib")'</p>'

/* Note: when using the BSF framework we need to require BSF.CLS in each script
and expression, if we need to access its public routines or public classes */
::REQUIRES "BSF.CLS"    -- make sure the Java bridge is there
</s:script>

</body>
</html>
```

Listing 6: helloworld_ext.jsp

Compared to the first page, this document's head includes the `link` tag. It is used to extend the document with additional resources. The `rel` attribute, standing for relation, is communicating the nature of the linked resource [91]. A stylesheet of the type `.css`, which is located by the URL: `css/shop.css` has been added. This path is relative, referring to the location of the document requesting the resource. Therefore, the file is found in: `helloworld/css/treeshop.css`. It is also possible to utilize an absolute path, by giving a full URL. However, relative paths are best practice, should the domain or computer change, all absolute paths would need to be changed otherwise [92].

Moving on, the first block has been both simplified and extended at the same time. The simplification is achieved using the Rexx `SAY` instruction which, thanks to the used tag library, results in the same result as the `println` method. The script's standard output file is redirected to the `out` implicit object. Conveniently, the `ScriptTagLib` removes the prompt `REXXout>` which gets prepended by the `RexxScriptEngine`, allowing to write proper HTML content to the JSP [35, p. 22].

```
greeting = "Hello, world! (Sent from Open Object Rexx)"
SAY '<h1>'greeting'</h1>'
```

Listing 7: helloworld_ext.jsp Printing HTML Tags

```
10 <header>
11
12 <h1>Hello, world! (Sent from Open Object Rexx)</h1>
```

Listing 8: helloworld_ext.jsp Generated HTML Code

Additionally, this demonstrates how HTML tags can be used to format outputs of scripting languages. First the tag `<h1>` gets printed in single quotation marks. Afterwards the variable greeting is inserted, followed by a closing tag, once again in single quotation marks. This approach allows weaving together outputs and HTML tags, generating dynamic content that is formatted according to HTML conventions. The

result, that gets sent to the end user, looks like simple, static HTML code, leaving no trace of ever containing scripting content.

Lesson Learned: This approach requires extra care when using quotation marks. Since double quotation marks are needed to specify attributes within tags, single quotation marks are used for `SAY` instructions.

While for other scripting languages the `out` object in combination with the `println` method is a universal way to write HTML content to a JSP, all examples will use the `SAY` instruction from this point onwards.

```
<p>The time right now: <s:expr type="rexx">TIME()</s:expr></p>
```

Listing 9: helloworld_ext.jsp Expression Tag

In addition to `script`, the `expr` tag can be used to fetch the result of an expression defined in a scripting language [93]. In the example, the body of the generated HTML document includes a paragraph outside of the script. It is used to demonstrate how the expression tag is used to intertwine the output of the ooRexx `time()` function with standard HTML code. The function returns a timestamp, which is consecutively displayed on the web page. The output reflects the point in time, at which the page was originally generated. Since HTML is static by nature, expressions allow quick enhancements with dynamic content.

Next, the script tag is used once again, demonstrating how multiple scripting elements can be added to a single JSP.

```
SAY '<p style="color:blue" !important>'pp("This paragraph is made possible by the
BSF taglib")'</p>'
```

Listing 10: helloworld_ext.jsp Inline Styling

The first thing to note is the use of a style attribute, indicating a CSS rule by means of inline styling. Overall, the CSS 2.1 version has over 90 properties, allowing in-depth customization of a web page, including fonts, tables, and backgrounds [94, Sec. 2.1.].

The rule consists of the declaration `color:blue`. When looking at the external stylesheet `shop.css`, the selector for a paragraph, `p`, already exists. The web page `helloworld_ext.jsp` therefore has two different style sources which might conflict with each other. A cascading order is used to determine the applicable value, which also gives the stylesheet its name. Since the declaration in the HTML document is declared `!important` it always takes precedence. Then, the BSF4ooRexx public function `pp` is used to place the text inside square brackets.

Lesson Learned: When creating and changing style elements in an external .css file, the changes might not be immediately reflected on the actual web page. The reason being, that the web browser usually caches front end resources. The hard refresh feature might prove useful by clearing the cache and reloading all resources. For example, a hard fresh can be performed by the button combination `CTRL-Shift-R` when using Firefox on Windows [95].

```
::REQUIRES "BSF.CLS"     -- make sure the Java bridge is there
```
Listing 11: helloworld_ext.jsp BSF Class

In contrast to the first example, the BSF tag library is used. Most importantly, when using this taglib, `::REQUIRES "BSF.CLS"` needs to be included at the end of a script. Without its addition, public routines, and classes of the BSF4ooRexx framework will not be functional. When using the JSR-223 taglib the directive can be omitted. The `BSF.CLS` which is part of BSF4ooRexx is used to define public routines and classes. For example, public routines offer functions such as the creation of Java arrays, while public classes such as `BSF_PROXY` enable sending ooRexx Messages to Java proxy objects [96]. Therefore, to avoid inexplicable errors or missing content, it is good practice to always include this directive when using the BSF taglib.

The choice of taglib hardly matters when ooRexx is used. Nonetheless, both the BSF and the JSR-223 tag library exist to ensure maximum compatibility and the ability to run programs without making changes. Afterall, some applications still use Apache BSF and some scripting language implementations only support the BSF framework [35, pp. 10, 23]. For example, programming languages like Groovy internally prefer the BSF framework, while older languages might not offer JSR-223 support at all.

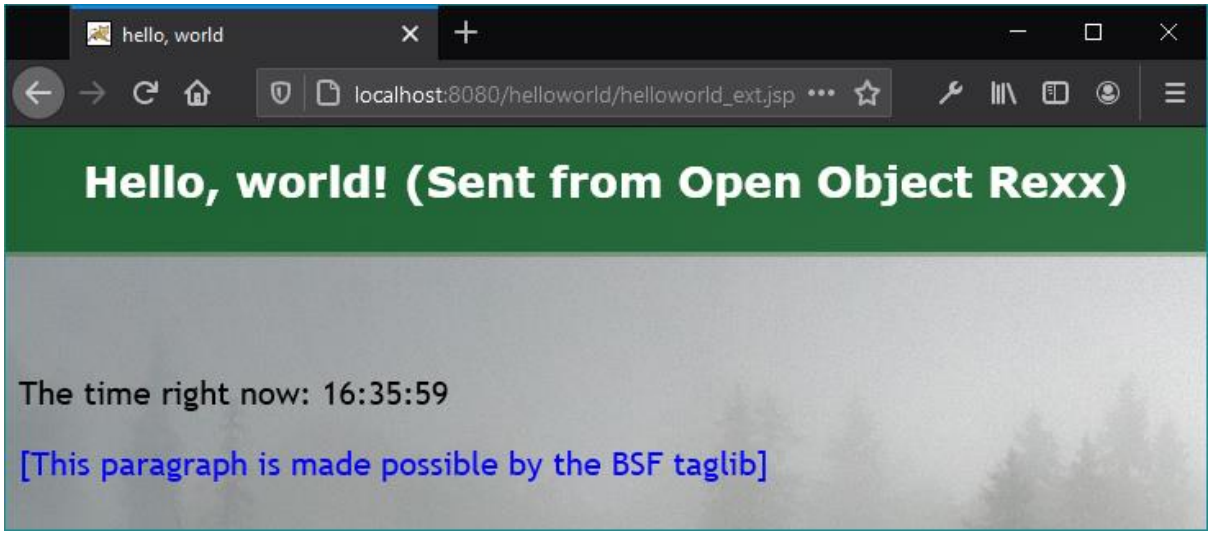

Figure 6: helloworld_ext.jsp in Web Browser

All the differences in the page's look and feel stem from the applied stylesheet.

4.4. Welcome Files `/helloworld/index.html`

This is a good point to mention welcome files. If no specific page is requested, Tomcat redirects the user to a welcome file. For example, the URL: `http://localhost:8080/helloworld` opens the welcome page for the `helloworld` web application. A welcome file can be placed in `WEBAPP\`, as well as in any subfolders. Tomcat will first look for a welcome file declared in the web application's `web.xml` file. It will then check for the existence of the following files: `index.html` → `index.htm` → `index.jsp` [97].

4.5. Introducing Cookies `/helloworld/lastvisit.jsp`

In previous parts of this thesis, the HTTP protocol and its stateless nature have been discussed. Cookies are a solution to the problem that different HTTP requests cannot be related to each other. Most website features that are taken for granted, like shopping carts, are enabled by cookies.

To maintain state, the server sends information in the `Set-Cookie` HTTP response header, to be stored by the client's browser. When the user contacts the same server at a later point in time, the previously received cookie data gets sent back in the `Cookie` HTTP request header. By changing the path and domain attributes, the scope of a cookie can be altered. By default, the cookie only gets sent to exact path from where the web page has been requested from. Each cookie is represented by a cookie-pair consisting `of cookie-name` and `cookie-value`. Should a web browser receive a cookie with the same `cookie-name`, `domain-value` and `path-value` as an existing cookie, the stored data gets replaced with the newly received values [98, Sec. 8.6.].

Furthermore, cookies include a `Max-Age` attribute; after the stated number of seconds has passed, the cookie gets deleted. Similarly, cookies can include an `Expires` attribute, which indicates the time and date at which the cookie expires. Should the cookie have both the `Max-Age` and the `Expires` attribute, the `Max-Age` attribute takes precedence. The cookie's `Domain` attribute indicates the hosts it gets transmitted to [98, Sec. 8.6.].

While the first parts of the page `lastvisit.jsp` are identical to the previous example, the document's body contains code utilizing cookies.

```
<s:script type="rexx">
USE ARG request, response, out

lastVisit = .nil
allCookies = request~getCookies

IF allCookies \= .nil THEN DO singleCookie OVER allCookies -- iterate over cookies
   IF singleCookie~name = "lastVisit" THEN lastVisit = singleCookie~value
END

IF lastVisit == .nil THEN SAY '<p>This is your first visit!</p>'
                     ELSE SAY '<p>Your last visit was at: 'lastVisit'</p>'

/* Create/Overwrite cookie with the current time */
cookie = .bsf~new("jakarta.servlet.http.Cookie","lastVisit",time())
cookie~setMaxAge(60*60*24) -- the cookie will expire after 1 day
response~addCookie(cookie)
</s:script>
```

Listing 12: lastvisit.jsp

The method `getCookies` is used to gather all cookies that are included in the request. It results in an array of all transmitted cookies or `.nil` in case no cookies exist [99]. This array is assigned the variable `allCookies`.

If cookies are present, a `DO OVER` loop iterates over all the cookies contained in the newly created array. It is necessary to first check for the existence of cookies, since a `.nil` value for `allCookies` would otherwise result in an error. If a cookie with the name `lastVisit` is present, its value will be assigned to a variable of the same name. In this case, a short message about the users last visit will be displayed, otherwise she will be informed that this is her first visit.

Afterwards, a cookie is created by utilizing the class `jakarta.servlet.http.Cookie`. A name and value are given with the constructor [100]. In the final step the method `setMaxAge` is used to define the cookie's expiration date. Since this value is given in seconds, a small mathematical operation is used to define a maximum age of one day. After the cookie is created, the `addCookie` method is used to add the cookie named `lastVisit` to the response that gets sent to the client. As value, the built in Open Object Rexx function `time()` is used to store a timestamp, corresponding to the exact moment in time when the code is executed [101]. For all future visits, this timestamp will be displayed and afterwards updated. Unless the visits are further apart than one day, after which the cookie will automatically be deleted.

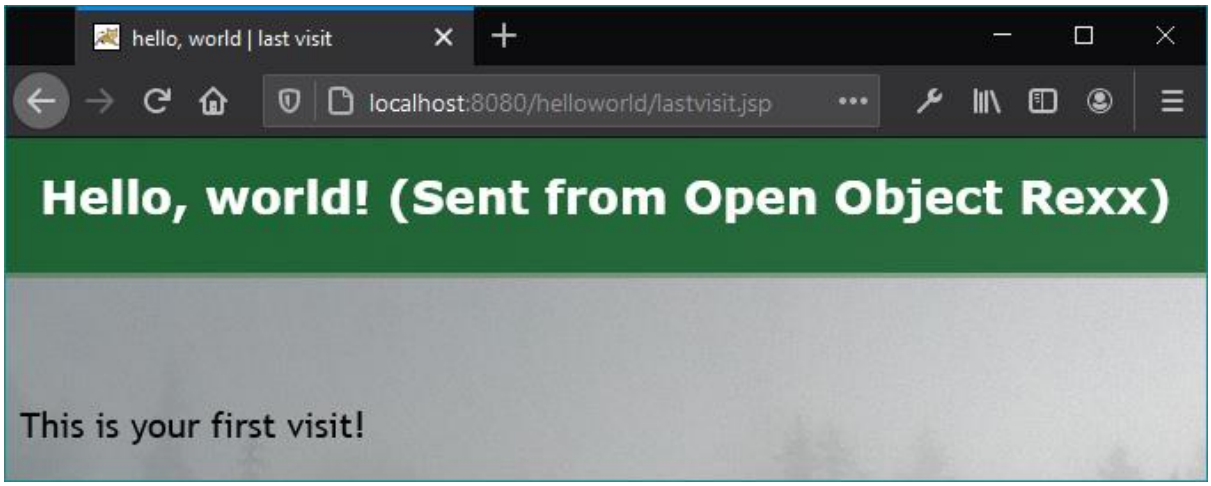

Figure 7: lastvisit.jsp in Web Browser on First Visit

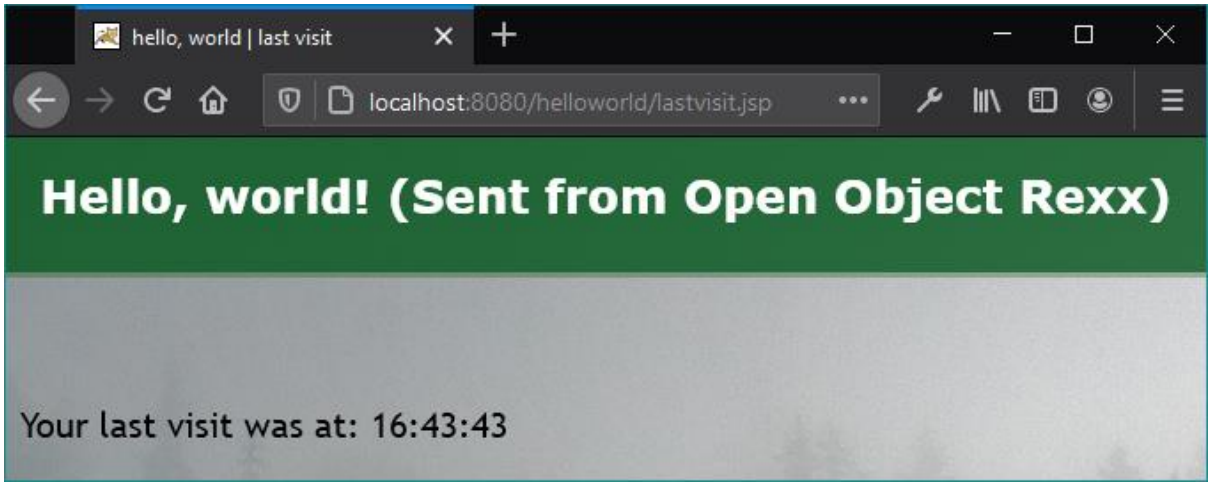

Figure 8: lastvisit.jsp in Web Browser on Consecutive Visit

Although simple, this third nutshell example might prove useful; it shows how cookies are created, transmitted, and afterwards accessed.

4.6. Combining User Input and Cookies `/helloworld/greeting.jsp`

The next example page, `greeting.jsp`, shows how information provided by the user can be stored and reused with a cookie. A visitor is asked for her name, allowing the web page to personally greet her on future visits. Again, the relevant code can be found in a script located in the document's body.

```
<s:script type="rexx">
USE ARG request, response, out

allCookies = request~getCookies

username = .nil    -- set as non-existent to begin
IF allCookies \= .nil THEN DO singleCookie OVER allCookies -- iterate over cookies
   IF singleCookie~name = "username" THEN username = singleCookie~value
END

IF username == .nil THEN DO
   SAY '<p>Hello, what is your name?</p>'
   SAY '<form>'
      SAY '<label for="username">Username:</label>'
      SAY '<input type="text" name="username" required>'
      SAY '<input type="submit" value="Ok">'
   SAY '</form>'
END
ELSE DO
   SAY '<p>Welcome back, 'username'!</p>'
END

IF request~getParameter("username") \= .nil THEN DO
   cookie = .bsf~new("jakarta.servlet.http.Cookie","username",-
            request~getParameter("username"))
   cookie~setMaxAge(60*60*24) --The cookie will expire after 1 day
   response~addCookie(cookie)
   response~sendRedirect(request~getRequestURI())  -- refresh page
END
</s:script>
```

Listing 13: greeting.jsp

After fetching the implicit objects and any cookies attached to the request, the script first checks whether a cookie called `username` is existent or not. Should none exist, a form is generated, asking the user to input her name. Since no submission method is declared, the default method `get` is used, appending the data to the URL. The action attribute is used to define a form's processing agent [102, Sec. 17.13.]. To clarify, here the web page is given, to which the submitted data is sent. Since this is a single page application, the form data is sent to the same page. For HTML5 it is sufficient to simply omit the `action` attribute to achieve this. HTML4 on the other hand, requires a value for the action attribute to function [103].

Lesson Learned: In HTML, elements like `<input type="text">` should always be accompanied by a label. This will help users who use screen readers or have trouble clicking on small fields [104].

To avoid unexpected behavior and potential malfunctions, it is important to design a web page in a way to make common mistakes impossible. One such scenario would be a user not entering a name, for example by prematurely submitting the form. This scenario would result in an awkward greeting message. Avoiding this can be done without any additionally code, in a simple but elegant way. The `required` attribute can be given for input types such as `text`, `url`, `email` or `password`, only allowing forms to be submitted if the field has been filled [105]. Therefore, when the page is visited for the first time, the following form is displayed.

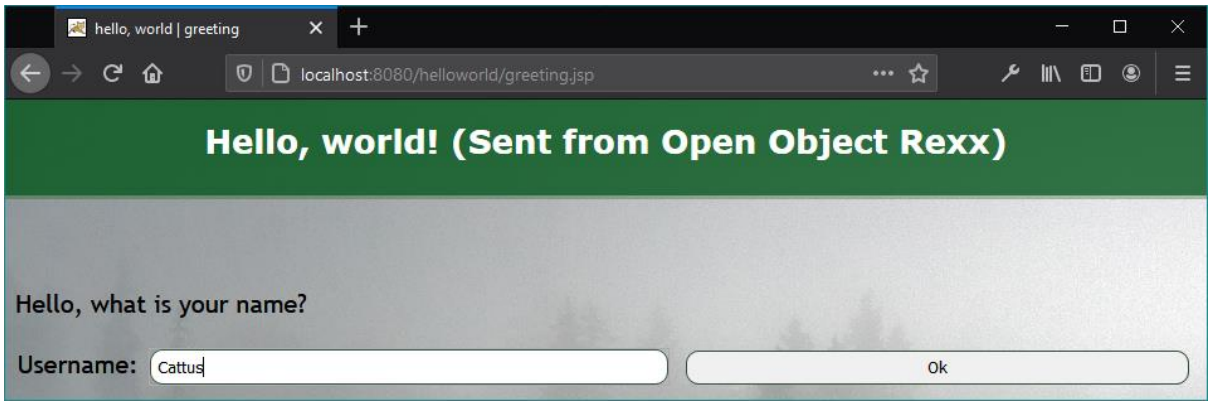

Figure 9: greeting.jsp in Web Browser on First Visit

On form submission, the page is reloaded, and the request will contain the parameter `username`. For this situation, the `IF` loop at the end of the program is activated and a cookie is created. The method `getParameter` is used to fetch the username previously provided. The corresponding name for the parameter is declared by the form's text input field, using the `name` attribute.

Finally, the `sendRedirect` method of the `response` object is used to refresh the page. This is accomplished by fetching the current page address with the `getRequestURI` method.

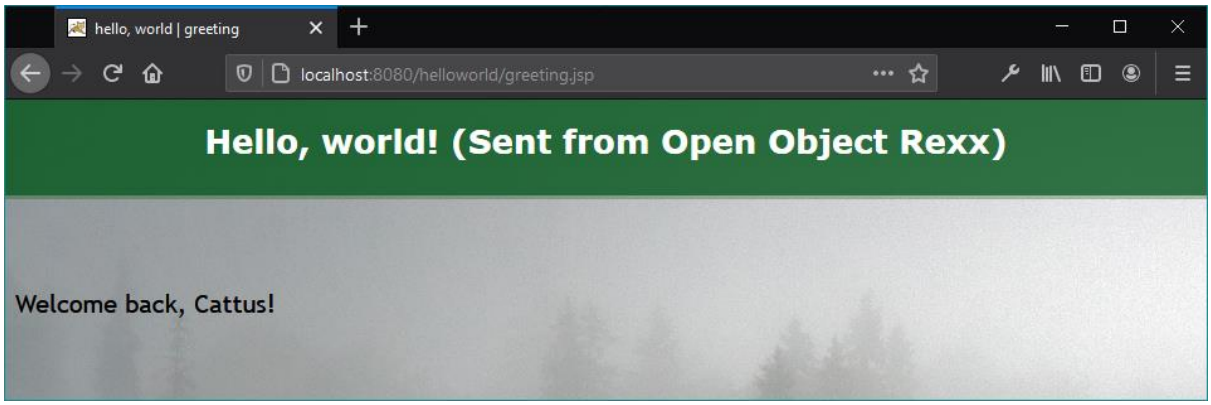

Figure 10: greeting.jsp in Web Browser Consecutive Visit

Since the request now contains a cookie called `username`, the form and cookie creation are skipped, and the personalized greeting is displayed instead.

Like any other program, code is executed top to bottom. Therefore, on each visit, the script uses three `IF` blocks to change the web page accordingly. If a username exists in form of a cookie, a personalized greeting is displayed. If not, a form is generated asking for a name. If the page is reloaded as the result of a form submission, the third `IF` block creates a cookie.

By directing form submissions to the same page, they originate from and adding `IF` loops, simple, self-contained web applications can be created on a single JSP.

4.7. Deleting Cookies, External Scripts `/helloworld/greeting_ext.jsp`

The next example page `greeting_ext.jsp` builds on the previous one, adding a logout button to demonstrate how the previously stored name can be removed. At first glance, the page looks almost identical, yet the structure has been improved. Depending on the existence of a cookie containing a username, either a login form or a logout button is displayed. The HTML code to generate those components is stored in a `::RESOURCE` directive, which can be found at the bottom of the script.

```
::RESOURCE logoutButton
<form>
   <input type="hidden" name="logoutButton" value="1">
   <input type="submit" value="Logout">
</form>
::END
```

Listing 14: greeting_ex.jsp Resource Logout Button

A `::RESOURCE` contains an unlimited number of strings up until the `::END` directive, which are stored in a `Stringtable`. The name given to the resource serves as an index, in this case, `logoutButton`, for which all the lines given are stored in an array. Afterwards the entries are fetched with the environment symbol `.RESOURCES` and the given name `~logoutButton` [106, p. 3]. This feature, which has been introduced in ooRexx 5.0 can prove particularly useful for reusing static HTML building blocks [35, p. 23]. Another benefit of this feature is the ability to effortlessly write HTML code without quotation marks.

Additionally, this example highlights how an input of the type `hidden` is used to attach data to a request. After the button is clicked, the request will include the parameter `logoutButton` with the value `1` attached. Essentially, information about the user's previous actions is transmitted, enabling state without utilizing cookies.

With this newest addition, the web page has four possible behaviors, depending on the information attached to the request. In case a cookie containing a username is present, a personalized greeting is displayed, otherwise the user will be asked to input a name. Should the request indicate that the user has just filled in the form or wishes to logout,

a cookie needs to be either created or deleted. While the first two behaviors are programmed directly in the JSP, the latter two are implemented by an external script.

The following line of code adds the external script to the body of the document:

```
<s:script type="rexx" src="code/logout.rex" cacheSrc="false" />
```

Listing 15: greeting_ext.jsp Attribute src

The linked script `logout.rex` is stored in the directory `helloworld\code`. Like the addition of a `.css` file, the path is relative. The script tag's `src` attribute allows the inclusion of an external file, containing code in the specified language. Additionally, the `cacheSrc` attribute is set to `false`. If a web page is still under development, it is highly recommended to set this attribute to `false`, preventing the file from being cached and instead rereading it each time it is used. If the attribute is omitted, it is set to `true` by default, necessitating a full server restart for changes to be reflected on the JSP [35, p. 11].

Additionally, multiple optional attributes for the `script` and `expr` tags are available, mainly for the purpose of debugging. For example, `throwException` will halt the JSP processing in case an error is encountered, while `debug` can be used to inject debug information. By default, these attributes come set to `false`. Furthermore, the `name` attribute allows the script to be given a name, while setting `arguments` to `false` disables the implicit objects `request`, `response` and `out` to be submitted as arguments to the script. Moreover, `slot` allows a developer to include a string, to be later fetched from inside the invoked script, facilitating information flow between components [35, p. 11]. Finally, `reflect` creates a HashMap object, holding information about the taglib, namespace and current attribute values [93].

Unique to the JSR-223 taglib is the `compile` attribute [35, p. 11]. By using a script in its compiled form, performance can be dramatically improved, especially if the code includes mathematic calculations. Compiling a script only makes sense if it is going to be reused multiple times, otherwise the conversion and compilation process needs to be considered [107].

The sample web application that comes with the tag libraries, `demoRexx` contains examples showcasing these features. For a download link, please refer to: `A.1. Software Required to Begin`

```
USE ARG request, response, out

IF request~getParameter("username") \= .nil THEN DO
   cookie = .bsf~new("jakarta.servlet.http.Cookie","username",-
            request~getParameter("username"))
   cookie~setMaxAge(60*60*24) -- the cookie will expire after 1 day
   response~addCookie(cookie)
   response~sendRedirect(request~getRequestURI()) -- refresh page
END

IF request~getParameter("logoutButton") \= .nil THEN DO
   removerCookie = .bsf~new("jakarta.servlet.http.Cookie","username","")  --
```

```
overwrite existing cookie
   removerCookie~setMaxAge(0) --The cookie will expire immediately
   response~addCookie(removerCookie)
   response~sendRedirect(request~getRequestURI()) -- refresh page
END

::REQUIRES "BSF.CLS"     -- make sure the Java bridge is there
```

Listing 16: logout.rex

The implicit objects `request`, `response` and `out` that are made available to a script, found inside a JSP can be accessed by an external script in the same fashion. The first `IF` loop of the script `logout.rex` contains the previously used code to store a username in a cookie. Should the request contain any value for the parameter `logoutButton`, the second `IF` loop is activated.

Since there is no specific method to delete cookies, instead a new cookie with the same name and an empty value is created, with its `maxAge` set to zero [108]. By adding this cookie to the response, the existing cookie is being replaced and the page gets refreshed afterwards. The `maxAge` attribute then causes this cookie to be deleted after zero seconds have passed. Since no cookie exists, the user is presented with the form to fill in a username once again.

While the page would still function, should the tag referring to an external script be at a different position, it has been placed at the beginning of the body on purpose. First, placing it inside the document's head might seem intuitive, but could easily result in it being overlooked. Afterall, the first lines of a web application might consist of copy and pasted building blocks, making modifications inconvenient. Since the page is executed like any other program, from top to bottom, placing it at the end of the body might result in unnecessary loading times. If any of the two `IF` loops found in the external file are activated, the page is refreshed, rendering the creation of other parts useless.

5. Database Connection

More sophisticated web applications require access to a persistent data source and the ability to freely add, delete and modify information. The separation of data and logic offers high flexibility and the chance to improve and update components separately. The ability to easily backup critical data is also a requirement for most operations. The following chapter describes the components necessary to connect a web application to a database.

5.1. Java Database Connectivity

In general, any database management system can be used, the only requirement being the availability of the Java Database Connectivity (JDBC) API.

JDBC is used to connect to a database, issue queries and commands, and to handle result sets. It can be implemented for both client-side and server-side connections. In a first layer, the Java application communicates with the JDBC manager through the JDBC API.

Afterwards, in a second layer, the JDBC manager communicates with the database driver [109].

The architecture for the system can be visualized as being three tiered. The user remotely accesses the web application from a web browser. The web application processes the user input and queries a database storing persistent data. The database sends back the result of the query to the application, which in turn uses it to create a web page for the user [110].

Each time a user connects to the database, resources are committed to creating, maintaining, and closing the connection. To allow a high number of users simultaneous and responsive access, the connections can be pooled and reused by means of connection pooling. Instead of closing and reopening connections for every request, the connections are cached and consecutively reused. For example, each PostgreSQL connection can take up to 1.3 megabytes in memory, multiplied by the number of connections, this number can easily skyrocket [111].

"It lets your database scale effectively as the data stored there and the number of clients accessing it grow. Traffic is never constant, so pooling can better manage traffic peaks without causing outages" [111].

Nonetheless, connection pooling can result in problems, if handled incorrectly. A so-called database connection pool leak can occur if a web application does not explicitly close objects related to the database connection, resulting in those resources being unavailable and a failure of the data connection [112].

5.2. Java Naming and Directory Interface

In many cases, applications utilize different services, provided by different components. For the given use case, a web application needs to find and cooperate with a database. The Java Naming and Directory Interface (JNDI) allows for different components to find each other.

Especially for distributed system, naming services are of great importance. Innovations like powerful microprocessors, high-speed computer networks and the miniaturization of computer systems have made distributed systems a possibility. Multiple autonomous computing elements are working together, while appearing as a single coherent system to the user [113, pp. 967-968]. For example, it might be plausible for the web application and the database to be running on different machines.

Names are used to refer to an entity, which can be practically anything, for example a host or a file. Those entities can then be used to perform operations on them. Each entity has one or multiple access points, which are another, special kind of entity. Their name is called an address. For example, a host, running a webserver is an entity whose access point is a combination of IP address and port. Since addresses are usually not readable in a human friendly way and might change over time, names are preferred [114].

Not only offers JNDI a single location for programs to find resources by name, but it also provides a common interface to existing naming services. In addition to naming, JNDI also offers directory services, which manage the storage and distribution of shared information [115].

6. E-Commerce Example `/treeshop`

From this point onward, all examples will be based on a fictional company, selling trees to be planted in the name of a buyer. A new web application has been created to showcase their products, including the ability for users to login and access a shopping cart. Furthermore, administrators can add new products and send promotional e-mails to customers. All content will be dynamically created, according to entries in a database.

The complementary database's data structure is kept minimalistic on purpose, only containing three tables with basic data. One to hold the products, another for the customers and a cart to connect them, realizing a many-to-many relationship. The following entity-relationship model is representative for the necessary database entries:

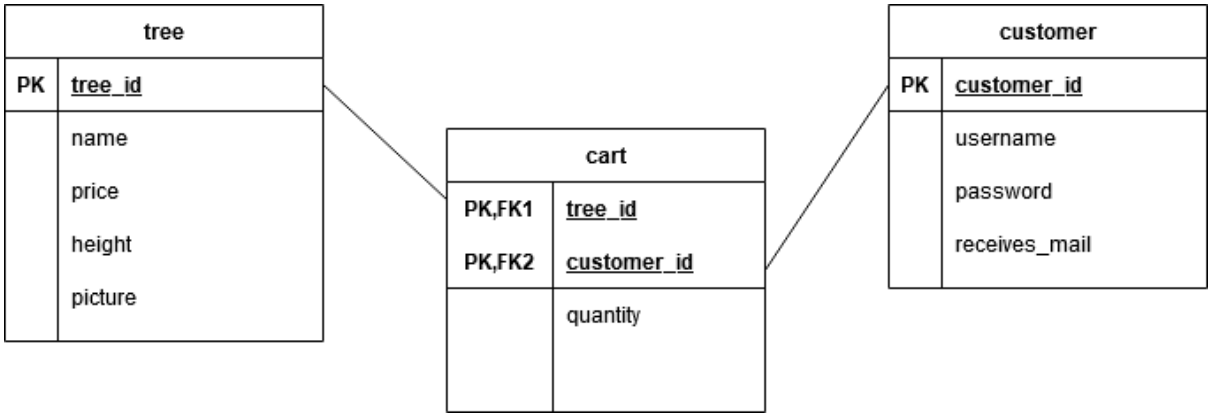

Figure 11: Entity-Relationship Model Database shop

6.1. Required Setup Steps

Highly Recommended: All setup steps are summarized to be viewed and copied from the URL: `http://localhost:8080/helloworld/support`

For the examples to function, the user is required to perform three configuration steps. First, Tomcat's configuration needs to be changed to enable the server to serve static files, like pictures. Then, a database management system needs to be installed and set up. Finally, two `.jar` files need to be copied to `TOMCAT_HOME\lib`.

6.1.1. Serving Static Content

Displaying pictures of available products to customers is an essential feature of any online shop. In general, static files can be served directly by a web application using the `DefaultServlet`. But, since web applications are often deployed from `.war` files, any

additions or changes would require redeployment [116]. Additionally, between redeployments, files might get lost.

Since the shopping website is intended to keep functioning, even if new products are added during deployment, this approach would not work. Later examples will introduce a way to add new products, including pictures. These pictures are to be stored in a directory outside of the web application, with the database only holding the path to access them.

To enable Tomcat to serve them and any other static content like stylesheets or HTML pages, from an external directory, some extra configuration steps are necessary.

The file `server.xml` can be found in `TOMCAT_HOME/conf`. The `<host>` element is found at the bottom of the document and needs to be extended with the following line: `<Context docBase="C:\Program Files\Apache Software Foundation\Tomcat 10.0\files\" path="/files" />`

This change allows Tomcat to independently serve files from a specified path, to be accessed directly, or to be used by web applications. The listing below shows the very bottom of the file `server.xml` and can be used as reference.

```
        <Context docBase="C:\Program Files\Apache Software Foundation\Tomcat
10.0\files\" path="/files" />
      </Host>
    </Engine>
  </Service>
</Server>
```

Listing 17: server.xml Context Tag

`Docbase` is used to indicate the directory where the static files are to be stored. Generally, the direct path to any folder on the machine running Tomcat can be given. `TOMCAT_HOME` has been chosen, since it is assumed that all readers have a directory with the exact or at least a similar path. For this purpose, the folder `files` needs to be created in `TOMCAT_HOME`.

Furthermore, the `path` attribute is used to define the URL, that files will be made accessible from; in this case: `http://localhost:8080/files`. For example, after the image `Maple.jpg` has been placed in `TOMCAT_HOME\files`, it can be accessed from the URL: `http://localhost:8080/files/Maple.jpg` [116].

The directory of the complementary archive: `ZIP_ARCHIVE\supportfiles` contains a folder named `files` that has already been set up with six sample product pictures. For the website to be properly displayed, it is necessary to copy the folder `files` to `TOMCAT_HOME` and to restart Tomcat after the configuration has been concluded.

6.1.2. Database Configuration

For the web application to function, the database needs to hold three tables and allow a specific user to access them. The appendix contains detailed instruction starting from

downloading the database management system to adding six example products: D. PostgreSQL

6.1.3. Tomcat's Handling of .jar Files

To ensure smooth development of new web applications, copying bsf4ooRexx-v641-20210205-bin.jar (or newer) and postgresql-42.2.18.jar (or newer) to TOMCAT_HOME\lib is essential. Both files can be found in: ZIP_ARCHIVE\supportfiles. Additionally, bsf4ooRexx-v641-20210205-bin.jar needs to be deleted from helloworld\WEB-INF\lib. The rest of this section discusses why these steps are necessary.

By default, Tomcat creates four classloaders, while ignoring the CLASSPATH environment that is used by standard Java environments. The loading of classes also slightly differs from what is standard practice for Java, where classes are in a parent-child relationship to each other [117].

In Tomcat's default configuration, the classloader on top of the hierarchy is called Bootstrap and loads classes provided by the Java Virtual Machine and the extensions directory of the Java Runtime Environment. Next, the WebappX classloader makes classes available to a specific web application. To accomplish this, it looks for classes located in the directories WEBAPP\WEB-INF\lib and WEBAPP\WEB-INF\classes. Next the System classloader loads classes required to initialize Tomcat as well as classes for logging and the Apache Commons Daemon project. Only then, the Common classloader loads classes from TOMCAT_HOME\lib [117]. To summarize, Tomcat first loads classes from WEBAPP\WEB-INF and only then from TOMCAT_HOME\lib. Should the same class be present in both locations it gets loaded from WEBAPP\WEB-INF.

Tomcat uses a separate classloader for each web application deployed. BSF4ooRexx caches Java classes to increase performance. If those cached classes are used by a classloader that did not originally load them, runtime errors might occur [35, p. 22]. Therefore, should multiple webapps be using BSF4ooRexx, the bsf4ooRexx-v641-20210205-bin.jar (or newer) needs to be placed in the TOMCAT_HOME\lib directory instead of WEBAPP\WEB-INF\lib. This approach allows a single instance of the BSF4ooRexx library to be used for all web applications deployed on the web server. Otherwise, only the first application using the library will function normally.

Since the focus of this thesis now switches from the helloworld web application to treeshop, this step is made necessary. To prevent errors, it is also required to delete bsf4ooRexx-v641-20210205-bin.jar from helloworld\WEB-INF\lib. It has originally been included to provide an easier introduction.

Additionally, it is recommended to place all JDBC drivers in the TOMCAT_HOME\lib directory. In case PostgreSQL is used, the driver is packaged in the file postgresql-42.2.18.jar (or newer). This file can be found in ZIP_ARCHIVE/supportfiles; alternatively, a download link can be found in: A.2. Software Required for Advanced

Examples. In contrast to other required .jar files, this file comes not included with the treeshop web application. Therefore, it is essential for the reader to manually copy it.

This is due to a broken service provider mechanism, which enables drivers to announce themselves without specific registration. Tomcat's JRE Memory Leak Prevention Listener fixes this issue by loading all drivers on server startup. If the .jar file is placed inside the web application though, the listener will not be able to find the driver. Instead, it will be loaded by the first web application requiring it. This approach can lead to various errors and unexpected behavior [112].

6.2. Reading Data /treeshop/productlist.jsp

For database access to function, a web application's WEBAPP\META-INF directory needs to contain a file called context.xml. This context is used to specify additional configuration information. While entering the data source solely in this file is sufficient, it is recommended to also define the resource in the previously mentioned web.xml file, mainly to document a web application's resource requirements [118]. Since these files are specific to the web application, they already come shipped with the .war file, requiring no further action from the reader.

```
<Context>
<Resource name="jdbc/postgres" auth="Container"
          type="javax.sql.DataSource" driverClassName="org.postgresql.Driver"
          url="jdbc:postgresql://127.0.0.1:5432/shop"
          username="cattus" password="tomtom12" maxTotal="100" maxIdle="10"
          maxWaitMillis="-1" removeAbandonedOnBorrow="true"
removeAbandonedTimeout="60" />
</Context>
```

Listing 18: context.xml

First, the name to be used by JNDI and attributes relating to the driver are specified. The url attribute is used to point to the database server's IP address and a database name. The database shop should be fully set up at this point. For complete instructions, please refer to: D. PostgreSQL. Should the reader prefer a different database management system the entries need to be adjusted accordingly.

It is also necessary to specify the username and password of a previously created user. The web application will use the given credentials to login and perform operations on the database. It is beneficial to create a unique user, since only the minimum rights needed can be assigned and actions taken by the application therefore quickly be identified. For this purpose, the user cattus has been previously created and assigned the password tomtom12.

Lesson Learned: When working with databases, a proper configuration is important. Should the given user not have the necessary permissions, nothing will work. Additionally, often a problem's source cannot be easily identified using Tomcat's logs. Should inexplicable problems occur, it is therefore recommended to apply the debug method shown in the appendix: F. Debug Code Snippet

To mitigate any possible database connection pool leaks the two attributes `removeAbandonedOnBorrow` and `removeAbandonedTimeout` are added. After a database connection has been left idle for the specified amount of time, it is terminated automatically [112].

To begin with, `productlist.jsp` gives a quick overview of the products currently listed in the database.

```
<s:script type="rexx">
cntxt = .bsf~new("javax.naming.InitialContext")
ds = cntxt~lookup("java:/comp/env/jdbc/postgres")
con = ds~getConnection  -- connect to database

stmt = con~createStatement
qry = "SELECT * FROM tree;"
rs = stmt~executeQuery(qry) -- retrieve all data from the table

SAY '<ul>'
DO WHILE rs~next -- iterate through all the rows stored in the table
    SAY '<li>'rs~getstring("name")':' rs~getstring("price")'€</li>' -- for each
row, fetch data
END
SAY '</ul>'

rs~close
stmt~close
con~close
</s:script>
```

Listing 19: productlist.jsp

First, the `InitialContext` class gets instantiated. Since this class is already included in the Java Standard Edition, no additional class files are needed. A context represents a set of bindings that all share the same naming convention. The created object gives access to the most basic methods, like naming or looking up objects [119]. The use of Tomcat further simplifies the configuration since it provides a JNDI `InitialContext` implementation instance, that gets configured for each web application during its initial deployment. Resources are placed in the JNDI namespace under `java:comp/env` [118]. Therefore, no further JNDI configuration is necessary and the database can be accessed effortlessly, using the `lookup` method.

After the `context` has been configured, the `getConnection` method is used to establish a connection. Once again, it is recommended to use the script's first three lines as standard building blocks for future web pages.

The `executeQuery` method of the `Statement` interface uses a SQL (Structured Query Language) statement as input parameter and returns a `ResultSet` object, containing the data returned from the query [120]. The `ResultSet` interface represent the data of the query in form of a table. The data is navigated by a cursor, which is initially at a position before the first row. The `next` method is used to advance the cursor along the table's rows. By default, the type is set as `TYPE_FORWARD_ONLY`, meaning that it is not possible to

go backwards and that the object is not sensitive to changes of the underlying data. The `ResultSet` offers a multitude of methods to access any desired data, for example the `getString` Method can be used to retrieve data from a column by name [121]. A `DO` loop iterates through entries of the result set, each representing a row in the output of the database query.

This example's query resulted in a row for each tree that is sold. Its name and price are fetched and displayed as part of a list. Afterwards, the cursor of the `ResultSet` is advanced to the next row, until none are left. Since the list's entries are dynamically created, should any new entries be added in the future, the page will automatically adapt.

For good practice and to avoid errors, the `ResultSet`, `Connection`, `Statement` and the later discussed `preparedStatement` should always be explicitly closed. Especially with connection pools, it is uncertain at what time `statements` and `preparedStatements` are otherwise closed [122].

Lesson Learned: If connections are not properly closed, the web page will break.

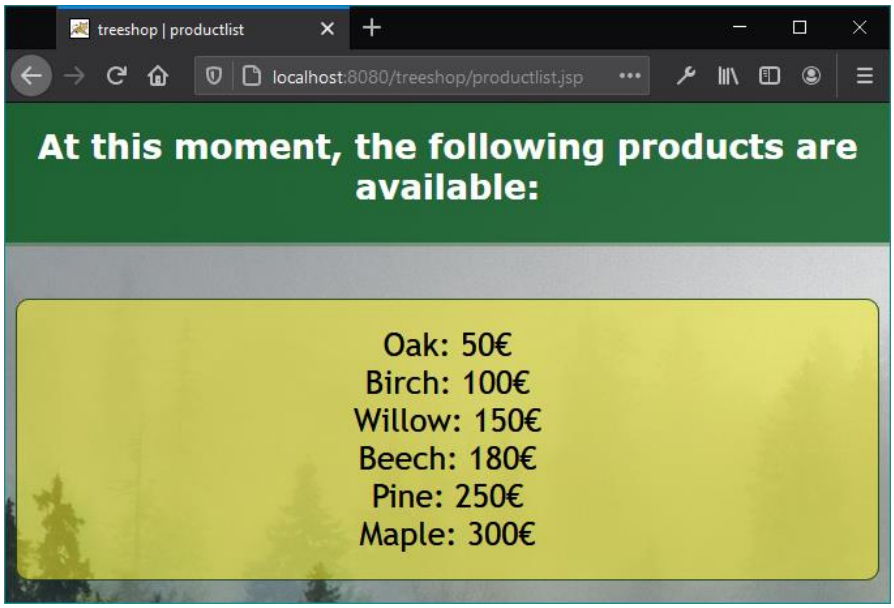

Figure 12: productlist.jsp in Web Browser

6.3. Writing Data, Security Aspects `/treeshop/signup.jsp`

Most dynamic web applications not only make data available, but also allow the user to interact and provide new data. As a minimum, almost all modern websites allow users to create a personal account.

While the writing of new data is relatively straightforward and quite like the previously shown solution, the storage of user provided data requires the consideration of additional security aspects. Not only does the user's data need to be stored safely, but the web application itself needs to be protected from unwanted manipulation by ill-intentioned actors.

The page `signup.jsp` starts like the previously shown pages and continues to display a form for a user to enter an e-mail address as username and a password, which needs to be repeated. Once again, all three fields have been set as `required`, not allowing the user to proceed without filling them first. This is particularly important since blank form fields might result in erroneous database entries.

Additionally, a checkbox can be ticket, for users who wish to receive promotional e-mails. The automatic generation of said e-mails will be implemented in a later example. At this point, the user's consent is requested to flag the newly created account accordingly. Even though, according to the European Union's General Data Protection Regulation, direct marketing e-mails about products or services can be sent to existing customers, any other promotional e-mails require prior consent [123]. Generally, it is good practice to only send e-mails to users who explicitly wish to receive them, not only to avoid complaints, but also to build a positive brand image. Consent should be given in the form of a clear, affirmative action; therefore, the checkbox needs to be explicitly clicked on and the corresponding label is not formulated ambiguously [124].

After the form data has been transmitted to the web server, the external script `createuser.rex` is activated and only progresses if both passwords entered match and the e-mail address has not previously been registered.

6.3.1. The Methods `GET` and `POST`

The first important difference can be observed in the form using the `post` method, instead of the default, `get`.

`post` signals the webserver that data is being sent and attaches it to the body of the message. In contrast, the data transmitted by a `get` request is appended to the URL and therefore easily visible. Not only might it be concerning for the user to see potentially sensitive data such as passwords in plain text, `get` requests are usually cached by web browsers and might additionally appear in their history. In conclusion, it is good practice to default to `post`, especially when dealing with forms of this nature [125].

6.3.2. Securely Storing Passwords

It is imperative not to store users' passwords as plain text. Vulnerabilities previously unknown or other security risks might result in a compromised database. Passwords are especially sensitive since users might use the same password for multiple websites [126].

The suggested solution to safe password storage is the application of a cryptographic hash function. It takes an input, the preimage, and generates a unique cryptographic fingerprint, called digest, for it. Each fingerprint is unique to an input and irreversible, making it impossible to backtrack to the original input [127].

"A hash function is a function that deterministically maps an arbitrarily large input space into a fixed output space" [127].

A good hash function needs to be deterministic and therefore always create the same digest for a given preimage. It should also create a fixed output size for any input. Thirdly, it should be uniform and therefore generate outputs that are evenly spread across all possible values. In case of a cryptographic hash function, it additionally should be one-way and therefore invertible. To conclude, the only way possible to get access to the preimage is by trying out all possible combinations in a so-called brute force attack [127].

Since some users might use similar or identical passwords, that might otherwise result in the same fingerprint, the concept of salting is introduced. Before the cryptographic hash function is performed, a unique, randomly created string is attached to the user's password. Otherwise, an attacker in possession of all the stored hash values, who managed to guess one of them correctly, might have access to all the other user's accounts that use the same password [128].

The Open Web Application Security Project (OWASP) suggests to strictly use third party libraries to implement the necessary algorithms. While Java itself offers cryptographic functionality and the creation of a message digest, there is too much room for error when creating a custom solution. For example, the widely used SHA-256 algorithm is simply too fast. OWASP recommends the Bcrypt hashing algorithm as the default choice [128]. A fast algorithm greatly reduces the time it takes to brute force a password.

The Blowfish encryption algorithm, developed by Provos and Mazières, allows users to increase the verification time, by modifying the cost value. This allows adjustment to increasing processor speeds and heightened security. It is based on their Eksblowfish Algorithm and offers a possible salt space so large, that it makes the precomputation of hash values based on common passwords incredibly difficult, since the required storage would be enormous [129, pp. 6-11]. The goal should be to find a balance between performance impact and security, tailored to the CPU speeds of the current day and age. It is also worthy to note, that should the algorithm be too taxing, an attacker might be able to perform a denial-of-service attack on the webserver [128].

Conveniently, the implementation of Bcrypt not only makes the process of password storage much more secure, but also extremely simple. Damien Miller offers the functionality of Bcrypt in form of a Java library called `jBcrypt`, which comes included with the `treeshop` web application.

```
bcrypt = .bsf~new("org.mindrot.jbcrypt.BCrypt")
fingerprint = bcrypt~hashpw(pw1,bcrypt~gensalt(12)) -- create a hash value that is
safe to store
```

Listing 20: createuser.rex jBcrypt hashpw

The method `hashpw` takes the user's input and a salt value to output the hash value in string format. Since the library also offers a secure method to create the salt value, the method `gensalt` is used. Most curiously, this method takes the previously discussed

work factor as input. Even though `jBcrypt-0.4` uses a work factor of 10 as default, OWASP recommends raising it to 12 [128].

6.3.3. SQL Injection

Before showcasing how the username and hashed password are stored, injection flaws need to be discussed. OWASP identifies injection as the number one web application security risk. A hostile individual might send untrusted data as part of a command or a query, to trick the interpreter to execute unintended commands or accessing data without authorization [130].

Su and Wasserman find web applications being susceptible to a large class of malicious attacks know as command injection attacks: *"This is because queries are constructed dynamically in an ad hoc manner through low-level string manipulations. This is ad hoc because databases interpret query strings as structured, meaningful commands, while web applications often view query strings simply as unstructured sequences of characters"* [110, p. 1].

To give an easy example, on an unprotected database anybody could enter `user; DROP TABLE customer;` in a field requesting a username. This could result in the following, or similar SQL query, should the web application forward the user input directly to the database: `SELECT * FROM customer WHERE username = user; DROP TABLE customer;` This could lead to destroyed databases and data exposure [131].

There is a quite simple solution to this problem, instead of the previously used `statement` interface, the extended version `PreparedStatement` can be used. When a SQL query is executed, it first gets parsed and compiled. Afterwards, the data acquisition path is planned and optimized. In the final step the query is executed, and the result gets returned. In comparison to the normal `Statement`, which goes through the four steps when the query is executed, `PreparedStatement` performs the first three steps when the statement is created and only performs the last step during execution. Not only does this increase the speed of database access and allows other features like batch processing, but all special characters are automatically escaped [132]. Escaping special characters results in them being treated as regular parts of a string, removing any ability to influence the essence of a query [133]. The above-mentioned exploit is therefore not possible since the user input is strictly treated as a normal string with no power to change the database query. In conclusion, when dealing with database queries based on user inputs, the minimum-security measure suggested is using the `PreparedStatement` interface.

```
prepstmt = con~prepareStatement("INSERT INTO customer (username, password) VALUES
(?,?)")
prepstmt~setString(1,username)
prepstmt~setString(2,fingerprint)
prepstmt~executeUpdate -- add new user to database
prepstmt~close
```

Listing 21: createuser.rex prepareStatement

The parts of the query where user inputs are used are omitted and instead filled with question marks. In the next lines the `setString` method is used to replace the question marks with the values the user has provided.

6.3.4. Hypertext Transfer Protocol Secure

Above all, the most important security measure necessary to facilitate a secure web application is the use of the Hypertext Transfer Protocol Secure (HTTPS) in place of regular HTTP. The Transport Layer Security (TLS) protocol, which was formerly known as Secure Sockets Layer (SSL) protocol, is used to encrypt data traffic by means of an asymmetric public key infrastructure. Otherwise, both protocols work the same. In contrast, all information sent by the regular HTTP protocol is sent as plaintext and therefore extremely vulnerable [134].

Tomcat supports the use of SSL/TLS but requires additional configuration steps [135]. To prove one's identity, it is necessary to obtain a certificate for a domain from a certificate authority. Since this work focusses on the development of web applications on a Tomcat server, running on a local network, the HTTPS protocol is not further discussed. Should the reader decide to make a web application accessible over the internet, the use of HTTPS as an absolute necessity.

All the described security measures are suggested as a bare minimum, with encouragement to invest additional time in research.

6.4. Creating an Online Shop, Sessions `/treeshop/index.jsp`

After establishing how database access works, now the focus shifts to the main page of the `treeshop` web application.

The main difference compared to previous examples is the usage of sessions. While sessions utilize cookie technology, they are more advanced and require the server to store data for each user. In contrast to cookies, the user only stores and transmits a session id, which the server uses to access data corresponding to it. For web development, using sessions is quite like using cookies, the only difference being slightly different methods used.

Since the data is stored by the server and not transmitted, the usage of sessions is more secure. Compared to cookies, which have a maximum size of 4 kilobytes, sessions can hold up to 128 megabytes each. To summarize, sessions and cookies both store user related data, the first on the web server, the latter on the user's web browser [136].

Tomcat makes the implementation of sessions very easy; they are automatically created by setting the `session` attribute of the `page` directive to `true`. The `HttpSession` interface is used to create a session id for each user and to store it in a cookie called `JSESSIONID`, which gets sent with each request. If cookies are disabled, the URL is rewritten instead [137]. Consecutively, objects related to a session id can be stored and accessed.

Additionally, general information about the session is retrievable. For example, the time of session creation can be requested with the method `getCreationTime` [138]. The figure below shows the `JSESSIONID` cookie, which is created upon first visiting a web page of the `treeshop` web application.

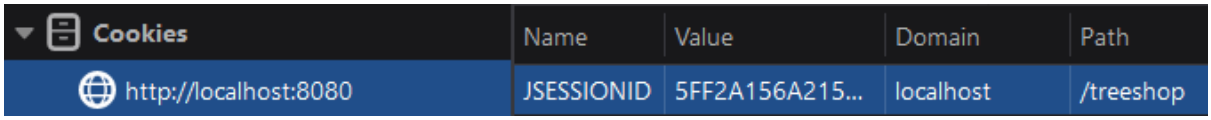

Figure 13: JSESSIONID Cookie in Web Browser

Since the default timeout value for Tomcat sessions is only thirty minutes, the `web.xml` file needs to be adjusted to extend this duration. By changing the `session-timeout` attribute of the `session-config` keyword, the lifetime can be easily extended [139]. Once again, the `web.xml` specific to the `treeshop` web application comes already edited, therefore the reader is not required to change it. The value has been set to 24 times 60 minutes, as a result a guest's data will be deleted after one day has passed.

Throughout this example, a user's login status is verified by means of a session. For this purpose, the use of a session instead of a cookie is highly beneficial for security. Were the user information directly transmitted with a cookie, a third party could easily replicate a cookie with a particular user's id to gain access. If a session token is transmitted instead, a unique value is generated each time the user logs in. Furthermore, sessions expire after a shorter time span has passed [140].

The main page of `treeshop` is created by the file `index.jsp`, two ooRexx scripts are used to build its components:

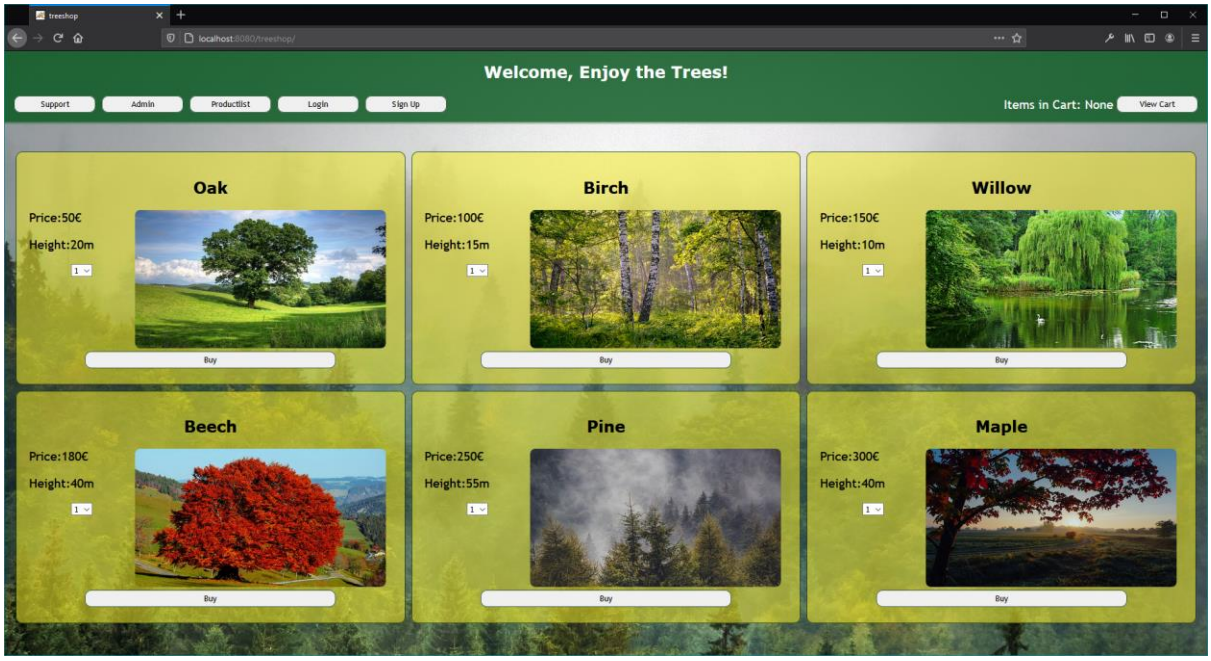

Figure 14: treeshop Main Page in Web Browser

6.4.1. `mainpage.rex`

The body of the shopping website's main page is created by the external script `mainpage.rex`. After the implicit objects, `request`, `response` and `out` are fetched, the method `request~getSession` is used to get access to data related to the session.

To begin with, the contents of the main page are built by querying all entries for the table tree, which contains all available products and information related to them. The routine `createProduct` uses this data to create a box for each product, displaying related information and enabling the user to put a specified quantity into the shopping cart. To enable the web page to adapt to any given number of products, these boxes are organized in a grid layout. The necessary styling parameters have been defined in the linked stylesheet.

```
::ROUTINE createProduct
PARSE ARG name, picture, price, height, tree_id

SAY '<div class="grid-item">'
    SAY '<h2>'name'</h2>'
    SAY '<img src="'picture'" height="100" width="100" />'
    SAY '<p>Price:'price'€</p>'
    SAY '<p>Height:'height'm</p>'

    SAY '<form name="choice" method="post">'
        SAY '<input type="hidden" name="choice" value="'tree_id'">'
        SAY '<select name="quantity">'
            SAY '<option value="1">1</option>'
            SAY '<option value="2">2</option>'
            SAY '<option value="3">3</option>'
            SAY '<option value="4">4</option>'
            SAY '<option value="5">5</option>'
        SAY '</select>'
        SAY '<input type="submit" value="Buy">'
    SAY '</form>'
SAY '</div>'
```

Listing 22: mainpage.rex Routine createProduct

The attribute `src` of the `<img>` tag specifies the URL of an image. The database entries for the images all look the same way: `/files/IMAGENAME.jpg`. The slash at the beginning of the path indicates a relative URL, referring to the current page. For example, the web page loads `/files/Maple.jpg` from: `http://localhost:8080/files/Maple.jpg`. The main advantage of this approach is, should the domain change, the web application will still work as intended [141]. Another benefit is the opportunity to easily modify the page. In case the pictures need to be loaded from another source, the only thing that needs to change is the URL stored in the database.

The script contains the necessary code for two approaches to handling the page's shopping cart. Depending on the session containing the attributed `logged`, the quantity chosen for a given product is processed differently.

A guest user's shopping cart is stored in a simple Java array, consisting of integers for both the index and the corresponding element. The index refers to a product id, referring to an item in the database, while the element specifies the quantity chosen. The array is stored in the session, allowing it to potentially scale in size. Special attention is given to products already present in the cart, instead of overwriting the quantity, it needs to be updated instead.

```
IF session~getAttribute("cart") == .nil THEN DO
    cartArray = .bsf~bsf.createJavaArray("java.lang.String",100)   -- create a new
cart if it doesn't exist
    session~setAttribute("cart",cartArray)
END

cart = session~getAttribute("cart")
IF cart[choice] == .nil THEN
    cart[choice] = quantity -- add a new product to the cart
ELSE
    cart[choice] = cart[choice] + quantity  -- update the quantity of an existing
product
```

Listing 23: mainpage.rex cartArray

Should the user be logged in, her shopping cart is stored in the database instead, using a `preparedStatement` to add entries.

```
qry = "INSERT INTO cart (customer_id, tree_id, quantity) VALUES (?,?,?) ON
CONFLICT (tree_id,customer_id) DO UPDATE SET quantity = ?;"
prepstmt = con~prepareStatement(qry)
prepstmt~setInt(1,session~getAttribute("logged"))
prepstmt~setInt(2,choice)
prepstmt~setInt(3,quantity)
prepstmt~setInt(4,quantity + cartquantity)
prepstmt~executeUpdate    -- update shopping cart
prepstmt~close
```

Listing 24: mainpage.rex Edit Table cart

When looking at the query, the user's choice of products gets inserted in the table `cart` as a combination of `customer_id` and `tree_id`, realizing the many-to-many relationship, in addition to the quantity chosen. Since each combination of values is defined as unique in the database, should a query attempt to duplicate it, an exception will occur. By using `ON CONFLICT`, this situation is resolved by updating the values, combining the new quantity chosen with the one previously stored.

6.4.2. `userheader.rex`

This script, creating the header for multiple web pages, demonstrates how external scripts can be conveniently reused. It enhances the header with the current number of products in the shopping cart, as well as multiple buttons and a personalized greeting. These elements adapt dynamically, according to the user's login status.

6.5. Creating a Shopping Cart `/treeshop/shoppingcart.jsp`

A shopping cart button, found in the header, redirects to the page `shoppingcart.jsp`. It allows to review all items stored in the cart, as well as adding, removing, or fully deleting products. The main functionality is held in the script `shoppingcart.rex`.

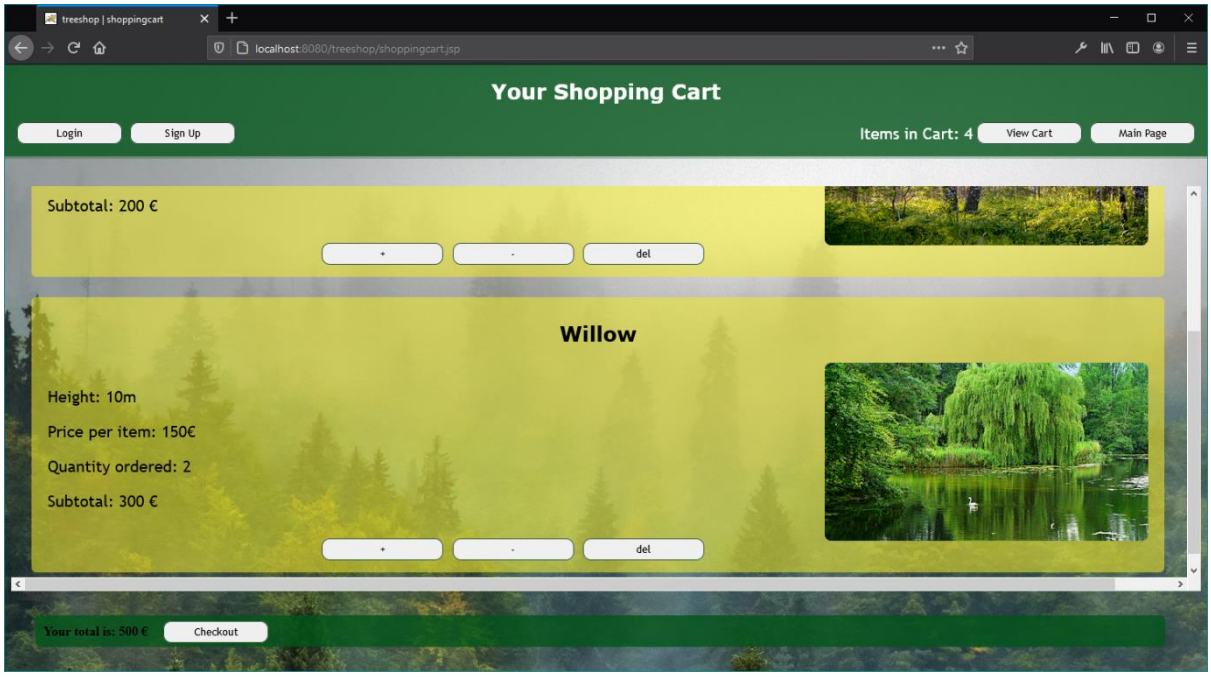

Figure 15: shoppingcart.jsp in Web Browser

Just like the main page, the script is split into two parts, which get executed based on the user's logins status.

```
cart = session~getAttribute("cart")
totalprice = 0
cartsupp = cart~supplier
SAY '<div id="cartcontainer">'
DO WHILE cartsupp~available   -- iterate over cart array
    qry = "SELECT * FROM tree WHERE tree_id="cartsupp~index";"
    stmt = con~createStatement
    rs = stmt~executeQuery(qry) -- get data for product in cart

    DO WHILE rs~next
        totalprice += rs~getString("price") * cartsupp~item
        CALL createProduct rs~getString("name"), rs~getString("picture"),
rs~getString("height"), rs~getString("price"), cartsupp~item,
rs~getString("price"), cartsupp~index
    END
    rs~close
    stmt~close
    cartsupp~next
END
SAY '</div>'
con~close

CALL printtotal totalprice
```

Listing 25: shoppingcart.rex Create Guest Cart

A supplier is created if an array, representing a guest's shopping cart, is stored in the session. Each iteration yields `cartsupp~index`, an index referring to a product id and `cartsupp~item`, its corresponding quantity. The index is used to retrieve product information from the database. Additionally, the total price of all items gets updated during each iteration, to be printed together with a checkout button at the bottom of the page. This is accomplished by the routine `printtotal`.

```
SAY '<form method="post">'
    SAY '<input type="hidden" name="tree_id" value="'treeid'">'
    SAY '<input type="hidden" name="qty" value="'quantity'">'
    SAY '<input type="submit" name="actn" value="+">'
    SAY '<input type="submit" name="actn" value="-">'
    SAY '<input type="submit" name="actn" value="del">'
SAY '</form>'
```

Listing 26: shoppingcart.rex Routine createProduct Buttons

The routine `createProduct` uses this information to create a box for each product in the shopping cart, similar to the approach chosen for the main page. Additionally, the routine creates three buttons, `+`, `-`, and `delete`, to manipulate the cart's contents. The listing above shows the code used to create them. When one of the buttons is clicked, the request not only contains the desired action, but also the corresponding product id and its current quantity. This information is added by a `hidden` attribute, which needs to be dynamically adjusted for each product.

```
IF request~getParameter("actn") == "-" THEN DO
    id = request~getParameter("tree_id")
    qty = request~getParameter("qty") - 1
    cart[id] = qty  -- reduce quantity by 1
    IF qty <= 0 THEN cart[id] = .nil    -- delete product from cart if quantity
goes below 1
    response~sendRedirect(request~getRequestURI)  -- refresh page
END
```

Listing 27: shoppingcart.rex Minus Button

Three `IF` blocks correspond to the generated buttons and get activated once they are clicked. The minus button is shown, since it necessitates to consider a situation, where the quantity reaches zero. In this case, the index value is set to `.nil`, indicating the product to be nonexistent. The same logic applies to the delete button where the value is set to `.nil` straight away. The plus button works in the same fashion, replacing subtraction with addition when it comes to modifying the quantity.

Should the user be logged in, the operations are similar in concept, except for the Java array being displaced by the database.

6.6. Logging In `/treeshop/login.jsp`

Previously shown pages adapt according to a user's login status. To make this possible, the page `login.jsp` takes a user's credentials and checks their validity. On success, the login status is stored in the session. Once again, the form input is processed by an external script called `login.rex`.

The script first uses a database query to determine the existence of the given username. If this is not the case, a label is used to jump to the same block of code that is used to display a message for a wrongly entered password. Thus, it is not made obvious whether the entered password is incorrect, or the e-mail address is nonexistent in the database. Otherwise, a third party would easily be able to determine if the owner of an e-mail address is a customer or not.

If the user exists, `jBcrypt's checkpw` method uses the entered password and the hash value stored in the user's database entry for authentication. On success, `TRUE` is returned and the attribute `logged` is added to the session, using the method `setAttribute`. By setting the attribute value to the corresponding user id, a link to the database entry can be established from this point onwards.

```
bcrypt = .bsf~new("org.mindrot.jbcrypt.BCrypt")
IF bcrypt~checkpw(pw, ha) THEN DO   -- only proceeds if password is correct
    session~setAttribute("logged",id)   -- store login status in session
```

Listing 28: login.rex jBcrypt checkpw

It is a common scenario for a user to browse and add products to the cart as a guest. Only on checkout, the user logs in, making it important that the shopping cart is not lost during the process. Therefore, on login, all the products stored in the guest cart need to be moved over to the database. Additionally, should any product be already present in the database cart, instead of overwriting the quantity, it needs to be updated. Like the way the shopping cart is displayed, a supplier is used to accomplish this. It is used to iterate over the Java array, storing each item in the database. After copying all its contents, the cart is deleted.

6.7. Logging Out, Invalidating a Session `/treeshop/logout.jsp`

Giving users the ability to log out is essential. To achieve this, the page `logout.jsp` is being made accessible by logout buttons throughout the website. Once the page is accessed, the method `invalidate` is used to clear the session and all its associated parameters. Given its brief nature, this functionality is directly implemented on the JSP.

```
session~invalidate    -- clear session
```

Listing 29: logout.jsp Invalidate Session

6.8. Concluding the Purchase Process `/treeshop/checkout.jsp`

The final page `checkout.jsp` simply removes all currently stored entries for a specified user from the `cart` table. The aim is to simulate a concluded purchase process. Should a guest attempt to checkout, a prompt to login will be displayed. In a real-world use case, the user would be asked for payment and shipping details instead.

7. Advanced Examples `/treeshop/admin`

This is a good moment to think about design decisions. For most examples until now, all the code is stored either directly in the JSP or an external ooRexx script. While this

approach has advantages, like all the code being in a single place and the ability to conveniently update it, the countless `IF` blocks complicate programs unnecessarily and result in redundantly executed lines of code. It might be beneficial to separate request specific actions from the generation of content. This would also result in increased efficiency, since less unnecessary elements need to be processed and loaded. Nonetheless, for a beginner, the shown approach is a fantastic way to quickly develop functional web pages. The conclusory web examples found in the subfolder `admin` of the `treeshop` web application will be used to show a different approach.

While the contents of `WEBAPP\WEB-INF` are shared across all directories, each subdirectory can be assigned its own resources and a unique welcome page: `http://localhost:8080/treeshop/admin`. The folder structure directly influences the path to access a web page. Also, special attention needs to be given to shared resources like stylesheets.

```
<link rel="stylesheet" href="../css/treeshop.css">
```

Listing 30: Link Resource in Subdirectory

The two leading dots indicate for the resource to be accessed from its parent directory. Therefore, the linked folder `css` is not found in the subdirectory `treeshop\admin`, but one level up. This enables pages found in subfolders to use the same stylesheet as pages directly placed in `WEBAPP\`.

7.1. Uploading Files `/treeshop/admin/addproducts.html`

Since entering data directly into the database can be time consuming and complicated, the page `addproducts.html` offers the functionality to add new products to be sold on the main shop page. Meanwhile, images are an essential part of modern web pages. This example facilitates their upload, to be seamlessly named, stored, and integrated. Since all of `treeshop's` pages are dynamically created, newly added products will appear momentarily.

The page `addproducts.html` displays a set of fields corresponding to the database's columns. To enable uploading files, a form needs to be given the attribute `enctype` with the value `multipart/form-data`.

Lesson Learned: Should a program perform mathematical operations on user inputs, it is essential to only allow numbers to be filled in the corresponding form field. This can be achieved with `input type="number"`.

```
<form action="uploader" method="post" enctype="multipart/form-data" id="mailform">
```

Listing 31: addproducts.html Upload Form

An `enctype` defines the document's encoding type, with `multipart/form-data` allowing file uploads. Usually, it is not necessary to specify a form's encoding, with file uploads being the exception [142].

In contrast to previous examples, no external `.rex` script is used to process the data, instead the form points to the servlet `uploader`. For a JSP to be used this way, the `web.xml` file specific to the web application needs to be edited, as can be seen in the listing below. Should the reader have copied the nutshell examples, no further modifications are necessary.

```
<servlet>
  <servlet-name>uploader</servlet-name>
  <jsp-file>/admin/code/uploader.jsp</jsp-file>
  <multipart-config>
    <location>C:\Program Files\Apache Software Foundation\Tomcat
10.0\files\</location>
    <max-file-size>10000000</max-file-size>
    <max-request-size>10000000</max-request-size>
  </multipart-config>
</servlet>

<servlet-mapping>
  <servlet-name>uploader</servlet-name>
  <url-pattern>/admin/uploader</url-pattern>
</servlet-mapping>
```

Listing 32: web.xml Uploader Servlet Configuration

The file `uploader.jsp` is configured as a servlet named `uploader`. This enables further configuration, otherwise only available to Java Servlets. For example, the load order, initialization attributes and security roles can be configured [143]. Since the JSPs content is written in the ooRexx language, a fully functional Rexx Servlet has been created. Additionally, the servlet's `MultipartConfig`, which controls file uploads, can be easily modified.

To begin with the location for temporary files is specified. The proper location and filename for the file will be chosen by the script at processing time. A temporary location is necessary since the file is first written as a temporary file and only afterwards processed to be stored permanently [144]. The `maxRequestSize` and `maxFileSize` Elements are used to set a limit for the size of both the file and the request, in bytes [145]. Here the maximum size has been set to the equal of 10 megabytes.

Additionally, the servlet is registered in the `servlet-mapping`. This map is used by the container to resolve requests [143]. Henceforth, the servlet is accessible from the path: `http://localhost:8080/treeshop/admin/uploader`. This configuration allows the JSP to directly process the request generated by the form found on the page `addproducts.html`. Since the request only gets sent to the servlet if needed, no more `IF` loops monitoring request parameters are required.

7.1.1. Upload Servlet `/treeshop/admin/upload`

The servlet functions like any other JSP, the only difference being the omission of any HTML start tags; after the JSP declarations, the script content immediately starts. Since all form fields have been set as `required`, the requests will always contain all necessary

form fields, as well as an uploaded file. The field values can be easily accessed like in any other form, using the `getParameter` method. The script first checks the database for any entries with an identical name. Should an entry with the same name exist, a warning is displayed to the user. Whenever content needs to be displayed, the resources `leadin` and `leadout` are used to create a proper HTML page.

Before a new product can be added to the database, the uploaded file needs to be processed.

```
filename = name || ".jpg"
location = "/files/" || filename
request~getPart("file")~write(filename)  -- permanently write file to temporary
location
```

Listing 33: uploader.jsp File Processing

The product's name has been given in the form and will be used together with the `.jpg` file extension to name the file. The file is placed in the `files` folder, which has been defined as a source for static content in: `6.1.1. Serving Static Content`. The string `location` gets stored in the database and is later used by the `img` tag to access the picture from its relative path.

Lesson Learned: Before new files are uploaded, the designated folder to hold them should already have been created. Otherwise, exceptions might occur.

The `getPart` method is used to access a specific part from the request. The file has been given the name `file` in the form, which is used to fetch it. At this point, the file is stored in the temporary location, defined in the `MultiPartConfig` found in the `web.xml` file. The method `write` is used to write the file to the disk, using the previously given filename. Since no specific path is given, it gets permanently stored in the temporary location [146].

After writing the file to the disc, a confirmation page is generated for the user. The resources `leadin` and `leadout` are used again, to create a proper HTML page. The HTML conventions should always be followed, by using resources, this can easily be accomplished.

Should multiple files be contained in the request, the method `getParts` can be used to get a collection of all Parts, to be iterated over [147].

After all fields are entered and an image is uploaded, the newly created product will immediately be visible on all the application's web pages.

7.2. Sending E-Mails `/treeshop/admin/sendnewsletter.jsp`

The final example page demonstrates how a web application can be used to send e-mails. Since the database already includes product details and the customers' e-mail addresses, all information necessary to create a newsletter, promoting currently available products, exists.

To begin with, the script found in the body of `sendnewsletter.jsp` creates a list of all products in the database and allows them to be selected by a checkbox. The JSP `mailer.jsp` is then used to create and send e-mail messages. Therefore, just like in the previous example, a servlet called `mailer` needs to be registered in the `web.xml` file.

For this web application to function the `.jar` files containing Jakarta Mail and Jakarta Activation are required. The demo web application already includes both. Jakarta Mail was previously known as JavaMail and empowers Java applications to implement e-mail functionality, such as sending and reading e-mail messages [150]. Jakarta Mail depends on Jakarta Activation to function. According to the Eclipse Foundation, it is used to: *"determine the type of an arbitrary piece of data; encapsulate access to it; discover the operations available on it; and instantiate the appropriate bean to perform the operation(s)"* [151].

Lesson Learned: The correct file `jakarta.activation-2.0.0.jar` should not be mixed up with `jakarta.activation-api-2.0.0.jar`. Furthermore, version 2.0.0 of Jakarta Mail requires Jakarta Activation to be using the Jakarta namespace, therefore Version 2.0.0 (or newer) should be used for both.

```
SAY '<form action="mailer" method="post" id="emailchoice">'
    DO WHILE rs~next
        SAY '<br>'
        SAY '<li><label for="choice">'rs~getstring("name")'</label>'
        SAY '<input type="checkbox" name="choice"
value="'rs~getstring("name")'"></li>'
    END
    rs~close
    stmt~close
    con~close
    SAY '<input type="submit" value="Send Newsletter to 'count' Receivers">'
SAY '</form>'
```

Listing 34: sendnewsletter.jsp Create Checkbox

The page `sendnewsletter.jsp` generates a form with a checkbox for each product in the database. The listing above demonstrates how this checkbox is created. All the form's checkboxes have the name `choice` and the corresponding product id as value. The page also uses a database query to count the total number of recipients, to dynamically display them inside the submit button.

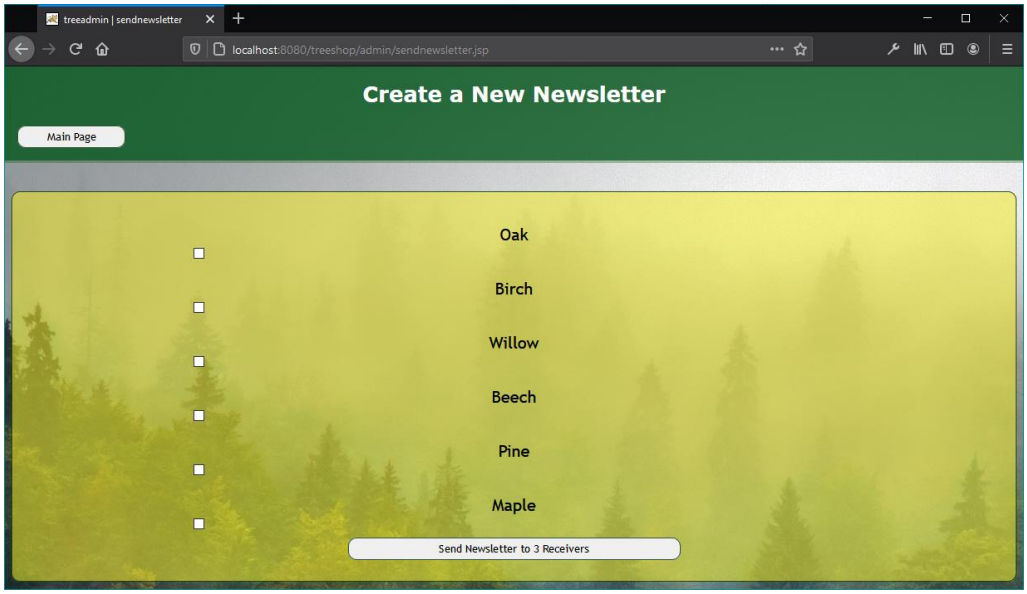

Figure 16: sendnewsletter.jsp in Web Browser

7.2.1. E-Mail Servlet `/treeshop/admin/mailer`

After the form has been submitted, the servlet `mailer.jsp` first makes sure, that at least one product has been selected, preventing an empty e-mail from being sent. Since `choice` most likely has multiple values, instead of `getParameter` the method `getParameterValues` is needed. It will fetch values related to all checkboxes ticked and stores them in an iterable string array [152].

```
choices = request~getParameterValues("choice")
choice = ""
DO productname OVER choices -- append all product names to a string
    choice = choice || "'" || productname || "'" || ","
END
choice = choice~delStr(choice~length)   -- remove the string's last comma
```

Listing 35: mailer.jsp Choicearray

To determine the products to be mentioned in the e-mail, first an empty string is defined as the variable `choice`. The previously fetched string array is iterated over, to create a list of product names, which is usable in a database query. For each iteration, a new product is added until none are left. The string `choice`, which comes empty for the first iteration, is gradually extended with product names, enclosed in single quotation marks and a comma. On conclusion, the last comma is removed to ensure a functioning database query. This is achieved by the `delStr` Method.

Delete String removes the character at the given position. By giving the length of the whole string, the last character is deleted [101].

Concerning the receivers, when a user signs up for the web page, an e-mail address is given, and newsletter preferences are stated. Therefore, a list of receivers can be easily generated. It might be a mistake to simply set all the shop's customers as receivers for a single e-mail. Afterall, each of them would see a whole list of other customers in the

recipient field and personalization would be rendered impossible. Therefore, the script sends a separate e-mail to each customer.

```
stmt1 = con~createStatement
qry1 = "SELECT * FROM customer WHERE receives_mail = 't';"
rs1 = stmt1~executeQuery(qry1)  -- select all customers who wish to receive the
newsletter

emailcount = 0
DO WHILE rs1~next -- create an e-mail for each customer and send it
```

Listing 36: mailer.jsp Select Receivers

In consequence, the database is queried for all customer entries that are signed up for the mailing list. These entries have the Boolean column `receives_mail` set to `TRUE`. The variable `mailcount` meanwhile keeps track of the number of e-mails sent. The created `resultSet` then yields an e-mail address during each iteration, which is used to create and send a personal e-mail. Since this first query will contain another, the variables `stmt`, `qry` and `rs` have been numbered accordingly. Should the same variables be used for both, they might overwrite each other and cause problems.

```
props = .bsf~new("java.util.Properties")
session = bsf.loadclass("jakarta.mail.Session")~getInstance(props)
msg = .bsf~new("jakarta.mail.internet.MimeMessage",session)

sender = .bsf~new("jakarta.mail.internet.InternetAddress",
"newsletter@treeshop.com")
msg~setFrom(sender)

receiveraddress =  rs1~getString("username")
receiver = .bsf~new("jakarta.mail.internet.InternetAddress",receiveraddress)
type = bsf.loadclass("jakarta.mail.Message$RecipientType")
msg~addRecipient(type~to,receiver)

msg~setSubject("Here Are the Latest Products from treeshop!")
```

Listing 37: mailer.jsp Create Message

To begin with, a Jakarta Mail session needs to be created. To instantiate the corresponding class, a set of Java properties is necessary. The class `java.util.Properties` creates a persistent set of properties where a key corresponds to a property, both of which are strings [153]. For the approach taken, no special properties are necessary, they need to be defined either way, since a set of properties is needed to create a mail session instance.

Afterwards, a `jakarta.mail.Session` instance is created. It is used as a bridge to the Jakarta Mail API, handling configuration and authentication. Using this session, the message to be sent is created; more specifically, the subclass `jakarta.mail.internet.MimeMessage`, which allows the use of different mime-types and headers [154].

Now, sender and receiver are added to the newly created message; these addresses need to be defined using the `jakarta.mail.internet.InternetAddress` class. While defining the sender is straightforward, the receiver additionally requires the recipient type to be

set by the class `jakarta.mail.Message$RecipientType`. Afterall, the receiver can take the form of `TO`, `CC` or `BCC` [155]. For this example, a simple `TO` receiver is used, the address being made available by the database. In case the exact same message needs to be sent to multiple receivers, `BCC` can be used to declare them without disclosing a full list of recipients. Besides, the subject is added.

The last piece missing is the message's body. To create it, a second database query is nested into the first.

```
stmt2 = con~createStatement
qry2 = "SELECT * FROM tree WHERE name in ("choice");"
rs2 = stmt2~executeQuery(qry2)  -- get data for all products that have been
selected

i = 0
DO WHILE rs2~next   -- create html code for each product
    line1 = '<div style="float: left; margin-right: 10px;">'
    line2 =  '<h2>'rs2~getstring("name")'</h2>'
    line3 = '<img src="http://localhost:8080' || rs2~getString("picture")'"
height="120" width="150" />'
    line4 = '<p>Price: 'rs2~getstring("price")' Euro</p>'
    line5 = '<p>Height:' rs2~getstring("height")' Meters</p>'
    line6 = '</div>'

    i += 1
    product.i = line1 || line2 || line3 || line4 || line5 || line6  -- append all
lines of html code
END
rs2~close
stmt2~close

text = '<html><head><meta charset="UTF-8" /></head><header>'  -- create proper
html leadin
text =  text || '<h1><a href="http://localhost:8080/treeshop">Vist
treeshop</a></h1>'
text =  text || '<h4><a
href="http://localhost:8080/treeshop/admin/unsubscribe.jsp?unsub=' ||
receiveraddress'">Click Here to Unsubscribe</a></h4>'

DO count = 1 TO i   -- append all products
    text = text || product.count
END

text = text || '</body></html>' -- append proper html leadout

msg~setContent(text,"text/html")
```

Listing 38: mailer.jsp Create Message Content

The main problem of this approach is that the whole message body needs to be contained in a single string. For this reason, and to easy formatting and the insertion of pictures, the message is created by HTML text.

The database queries data for all the products that are contained in the string `choice`, representing the checkboxes ticked on the previous page. A HTML segment for each product is created, for their sum to be amalgamated to form the main message. For each

product, six lines of HTML code create a `<div>` section. Additionally, each iteration increases the index value `i` by one. The six lines are then appended together to form a single line and stored under the variable `product.i`, where `i` is used to index them accordingly. This gives the script the flexibility needed to adapt to a varying number of products.

The e-mail's body commences with the necessary tags to properly define a HTML document. Afterwards, a headline, linking to the shop's main page is added, followed by a receiver-specific link to unsubscribe. The feature to unsubscribe from newsletters will be discussed at a later stage. Next, all the previously generated products are added to the string, followed by HTML closing tags. The `setContent` method is then used to set this string as the message's body, finalizing it.

7.2.2. Sending and Receiving E-Mails with MailHog

The Jakarta Mail API is platform and protocol independent and can therefore be used on any operation system and with most e-mail service providers allowing IMAP, POP3 or Simple Mail Transfer Protocol (SMTP) access [150]. Therefore, once this application has been properly tested, it can be linked to an e-mail server, to send messages into the real world.

During the first phases of testing, usually many e-mails need to be sent. To simplify this process, the author suggests the use of an open-source tool called MailHog. The appendix includes instructions on how to use it to set up a local SMTP test server: `E. MailHog Installation Guide`. The process is incredibly easy and can be done within minutes.

All incoming and outgoing e-mails will then be processed by MailHog, allowing to view e-mails from the receiver's perspective. The program creates an inbox, which can be accessed from a web browser using the URL: `http://localhost:8025`. This approach creates an environment that allows to refine the e-mails to be sent, as well as identifying any possible flaws in the program.

```
transport = session~getTransport("smtp")
transport~connect("localhost",1025,"username","pw")
transport~sendMessage(msg,msg~getRecipients(type~to))    -- send message using a
test server

emailcount += 1
```

Listing 39: mailer.jsp Send Message

The session's `transport` object is used to send e-mails. Its `connect` method establishes a connection to the server, using the attributes host, port, username, and password. Finally, the `sendMessage` method takes the previously created message and its recipients as inputs to send the e-mail.

By default, MailHog uses the port 1025 on the local host to process e-mails. Since the testing tool accepts any combination of username and password, placeholder values for

them will be used. Therefore, these configuration parameters will be used to demonstrate the capabilities of the nutshell example.

Instead of using Java properties, this easy approach to sending e-mails with Jakarta Mail sets the properties necessary at the final stage. Most noteworthy, the method `sendMessage` is used instead of the more commonly used `send` [156, p. 52].

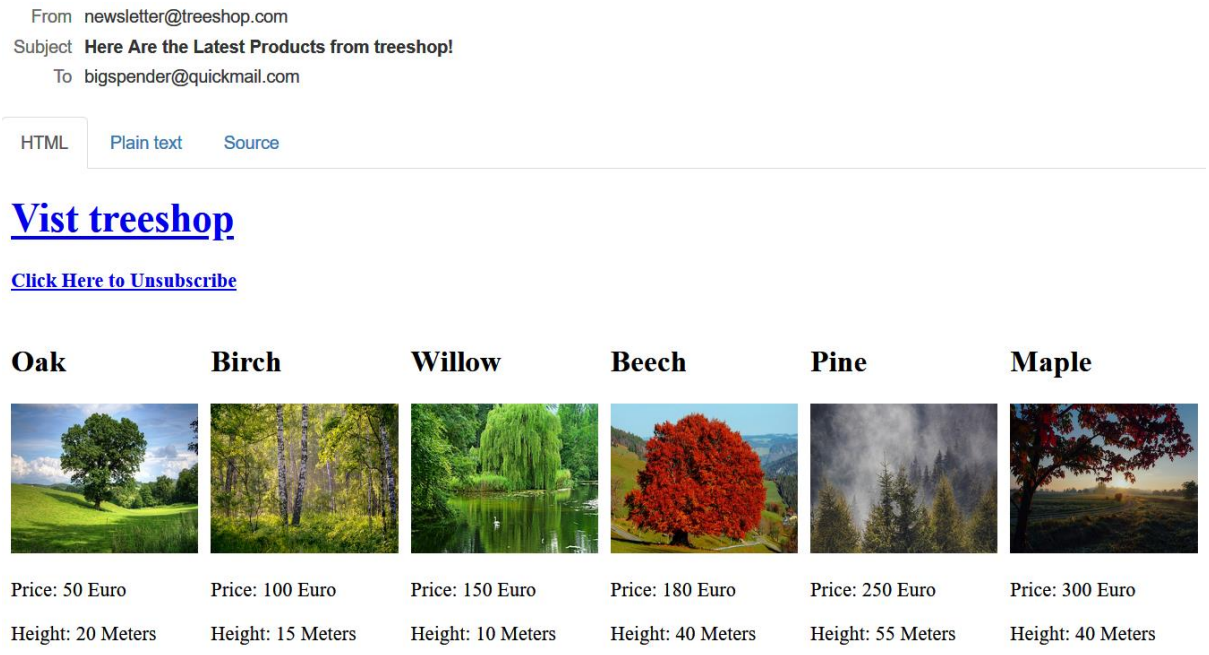

Figure 17: Newsletter in Web Browser

Lesson Learned: The pictures in the e-mail are only visible if the Tomcat server is up and running.

After the message has been refined to prove satisfactory, the next step would be to change the credentials to those of a real-world e-mail service, fully enabling the program's functionality. Although, the regular SMTP protocol is used for testing purposes, it is not secure and thus not recommended for everyday use. Therefore, like HTTPS, the Simple Mail Transfer Protocol Secure (SMTPS), which uses the SSL/TLS protocol needs to be used instead.

In addition to encrypting messages between the sender's e-mail client and e-mail server, SSL/TLS enables the use of digital certificates for identification purposes [157]. The appendix includes the code necessary to facilitate secure transmission of e-mails when using a real e-mail service provider: `G. SSL/TLS E-Mail Utility`

7.3. Unsubscribing from E-Mails `/treeshop/admin/unsubscribe.jsp`

The European Union's General Data Protection Regulation requires that users can object to receiving direct marketing e-mails at any time and that companies then must stop using their data immediately [123]. To implement this regulation, each e-mail includes a link to unsubscribe, containing a get request that gets dynamically created for each receiver.

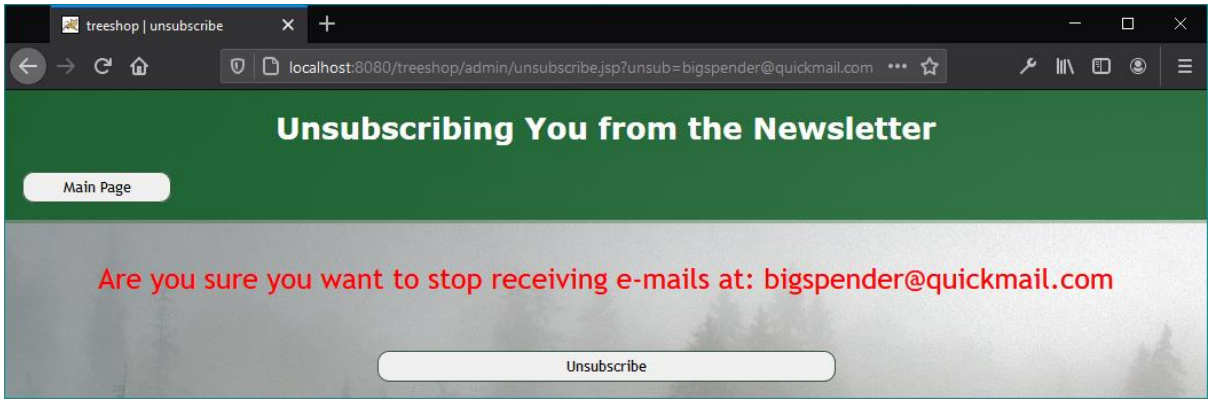

Figure 18: unsubscribe.jsp in Web Browser

By rewriting the URL, the script in the body of `unsubscribe.jsp` can fetch the e-mail address included in the request. It is appended to the URL, using the parameter `unsub`: `http://localhost:8080/treeshop/admin/unsubscribe.jsp?unsub=bigspender@quickmail.com`. Instead of immediately unsubscribing, the user is asked if she is certain and presented with a button to confirm. Upon clicking it, the user's e-mail address is forwarded to the servlet `unsubscriber.jsp`. Please note, that to function, the servlet needs to be registered in the `web.xml` file, just like in the previous examples. The servlet simply changes the column `receives_mail`, found in the table `customer`, for the given user to `FALSE`. Upon success, a confirmation message is displayed.

7.4. Common Gateway Interface

Not to forget, the Common Gateway Interface (CGI) offers the possibility to directly execute a script on the web server, generating a response for each request. Just like with JSPs, output methods can be used to create HTML pages. Tomcat allows the usage of CGI scripts by registering them like any other servlet. Since every request leads to the creation of a new process on the server, this approach can lead to significant performance problems in high traffic situations. Compared to Jakarta Servlets, which use Java, CGI scripts are dependent of the server's operating system, interpreters, and compilers. The request of a CGI script leads to its direct execution from the command line [148, pp. 13-17]. This results in the programs being run outside of the Java Virtual machine, bypassing the Java Security Manager.

Given all these limitations, CGI scripts are most commonly used during development [149]. While this work focuses on the use of JSPs, this method is still briefly mentioned, since it offers an alternative way to directly execute scripts.

8. Conclusion

After working with web applications extensively, one will never look at web pages the same way. It is astonishing how technologies, that now exist for over twenty years, are still used to create the modern world wide web we take for granted today. Furthermore, the author hopes to inspire readers to create their own web applications. By using the

building blocks introduced in this thesis, in combination with countless available external Java libraries, a beginner will be able to turn ideas into reality.

Three different approaches have been introduced: Adding scripting content directly to a JSP, linking external scripts containing the logic, and configuring a JSP containing script code as a servlet. While each of the approaches has its advantages and disadvantages, they offer great insights into web development and enable adaptation to a given situation.

For more sophisticated web applications, involving a team of developers, the intermingling of programming logic and design components will prove problematic. Such projects will use the model-view-controller design pattern, implemented by a framework like Apache Struts. Nonetheless, the approaches shown allow the creation of dynamic web applications in record time.

A. Prerequisites

This section not only contains a collection of hyperlinks to all the required software but can also be used as a checklist. This work was finished in the beginning of the year 2021 and reflects the current development stage. For future use, the download locations might change, and the software will be most likely be updated. The author recommends downloading the latest versions currently available.

A.1. Software Required to Begin

- Nutshell examples: `http://wi.wu.ac.at/rgf/diplomarbeiten/`
  - An archive containing the demo applications should come included with this work. In case it is missing, please search for this thesis in the collection provided
- OpenJDK: `https://bell-sw.com/pages/downloads/`
  - `Liberica Full JDK` should be chosen for maximum compatibility
  - Any other Java implementation will also work, this distribution is merely a suggestion
  - Needs to match the ooRexx version used, a 64-bit ooRexx installation requires a 64-bit version of Java, whereas a 32-bit version requires a matching 32-bit installation
- ooRexx: `https://sourceforge.net/projects/oorexx/`
  - As a minimum, Version 5.0.0 needs to be installed
- BSF4ooRexx: `https://sourceforge.net/projects/bsf4oorexx/`

- The file `bsf4ooRexx-v641-20210205-bin.jar` (or newer) can be found in the downloaded archive, or once installed, in the installation directory of BSF4ooRexx
- Also contains an IntelliJ IDEA plugin, enabling text highlighting for ooRexx
- Additionally, the latest version of the Tag Libraries (they already come included with the web applications) can be downloaded from: `https://sourceforge.net/projects/bsf4oorexx/files/Sandbox/rgf/taglibs/`
    - Also contains the `demoRexx` web application with additional examples
    - Also contains a language injections file for IntelliJ IDEA, enabling ooRexx to be highlighted in HTML, XML and JSP documents

❖ Apache Tomcat 10 (Beta status in January 2021): `https://tomcat.apache.org/download-10.cgi`
- As an Alternative, Apache Tomcat 9 (Stable status in January 2021): `https://tomcat.apache.org/download-90.cgi`
- Before beginning the installation, it is recommended to inquire about the current development status: `https://tomcat.apache.org/whichversion.html`

A.2. Software Required for Advanced Examples

❖ PostgreSQL: `https://www.postgresql.org/download/`

❖ PostgreSQL JDBC Driver: `https://jdbc.postgresql.org/`
- Already comes included

❖ jBcrypt: `https://www.mindrot.org/projects/jBCrypt/`
- Already comes included

❖ Jakarta Mail: `https://eclipse-ee4j.github.io/mail/`
- Already comes included

❖ Jakarta Activation: `https://eclipse-ee4j.github.io/jaf/`
- Already comes included

❖ MailHog: `https://github.com/mailhog/MailHog`

B. Tomcat Installation Guide

The following section will give a step-by-step installation guide for the Apache Tomcat Software version 10.0.0 on the Microsoft Windows 10 Operating system. Before beginning, as a minimum Java needs to be installed.

In case the reader prefers the stable Tomcat Version 9, the installation process is identical.

Before beginning the installation, it is recommended to inquire about the current development status from: `https://tomcat.apache.org/whichversion.html`

The apache-tomcat-10.0.0.exe can be downloaded from the webpage: `https://tomcat.apache.org/download-10.cgi`, by clicking on `32-bit/64-bit Windows Service Installer.`

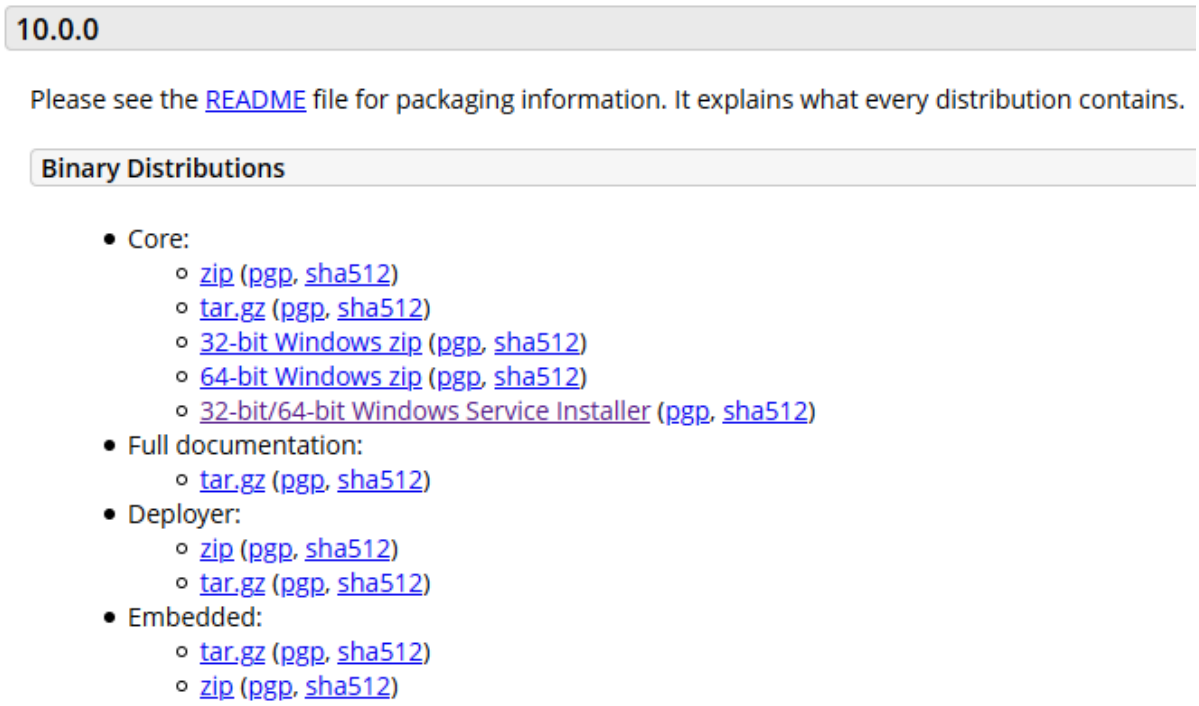

Figure 19: Tomcat 10 Download Page

After downloading and executing the file `apache-tomcat-10.0.0.exe`, one is greeted with the following window. It is worth to note, that under most Windows 10 configurations, upon running the exe file, one is greeted with a popup from Windows User Account Control, where it is necessary to grant the program permission to make changes on the device. The installer being still labeled as Apache Tomcat 9 is most likely a result of the software still being in Beta status.

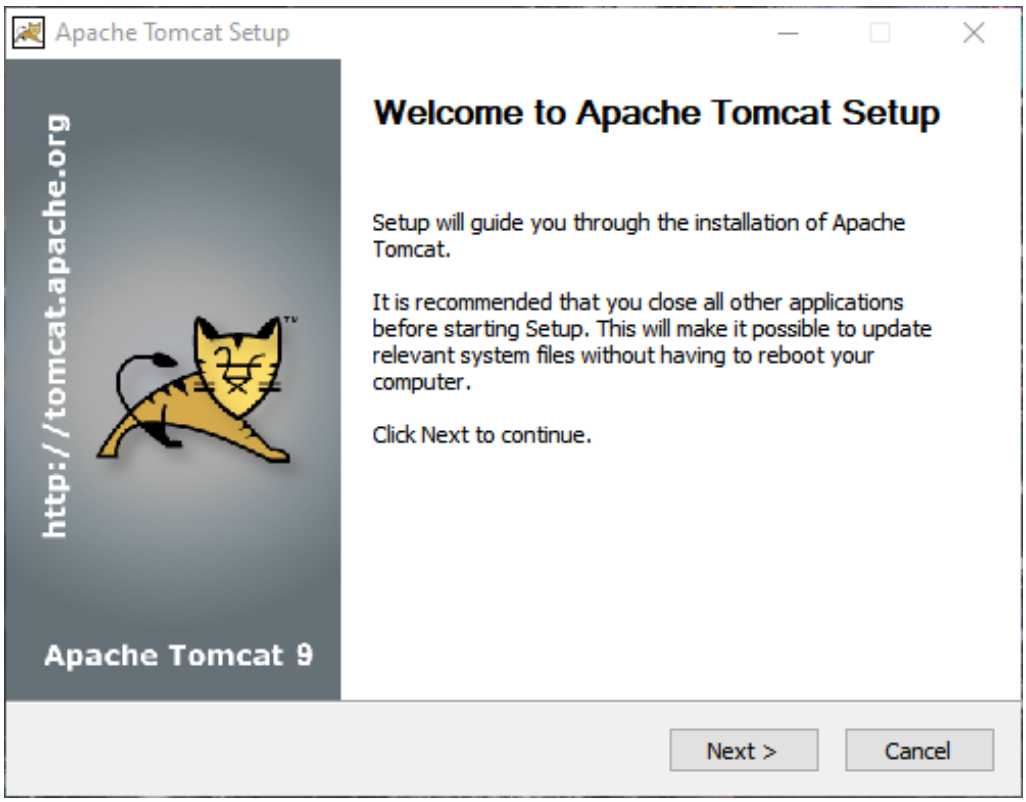

Figure 20: Tomcat 10 Setup Welcome

After clicking the `Next` button, the License Agreement can be reviewed and needs to be accepted by clicking on "I Agree".

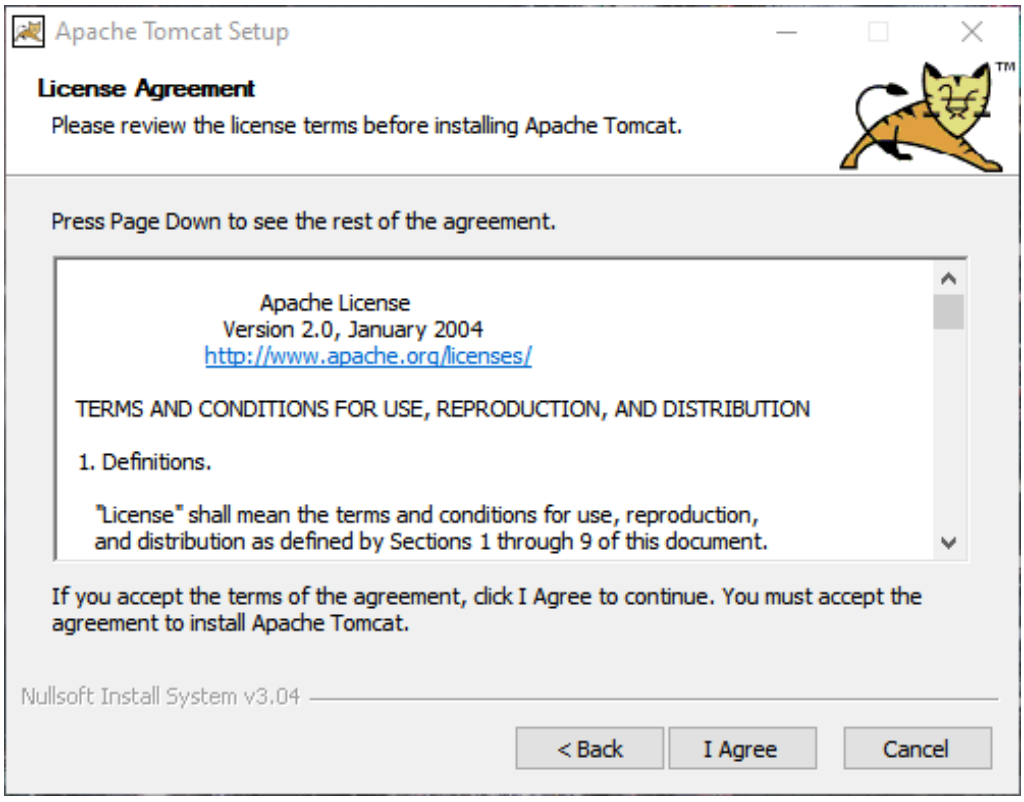

Figure 21: Tomcat 10 Setup License Agreement

The following window allows the customization of components to be installed. It is recommended to select a `Full` installation.

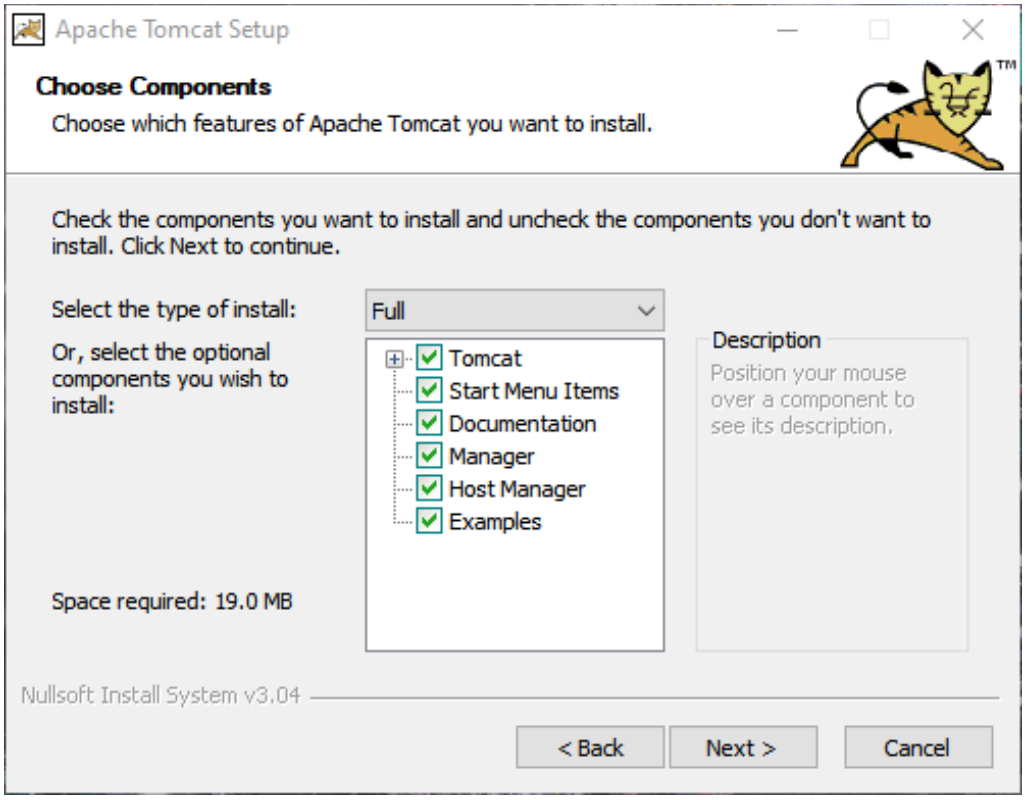

Figure 22: Tomcat 10 Setup Choose Components

The entries `Start Menu Items`, `Documentation` and `Examples` are self-explanatory and might prove useful. The `Manager` entry is used to create a web application, accessible from: `http://localhost:8080/manager`, with various functions like listing all users and currently installed web applications, as well as the option to deploy and undeploy them [80]. Meanwhile, the `Host manager` is used for creating multiple websites on a single server [158].

After choosing the desired components, the next window allows further configuration.

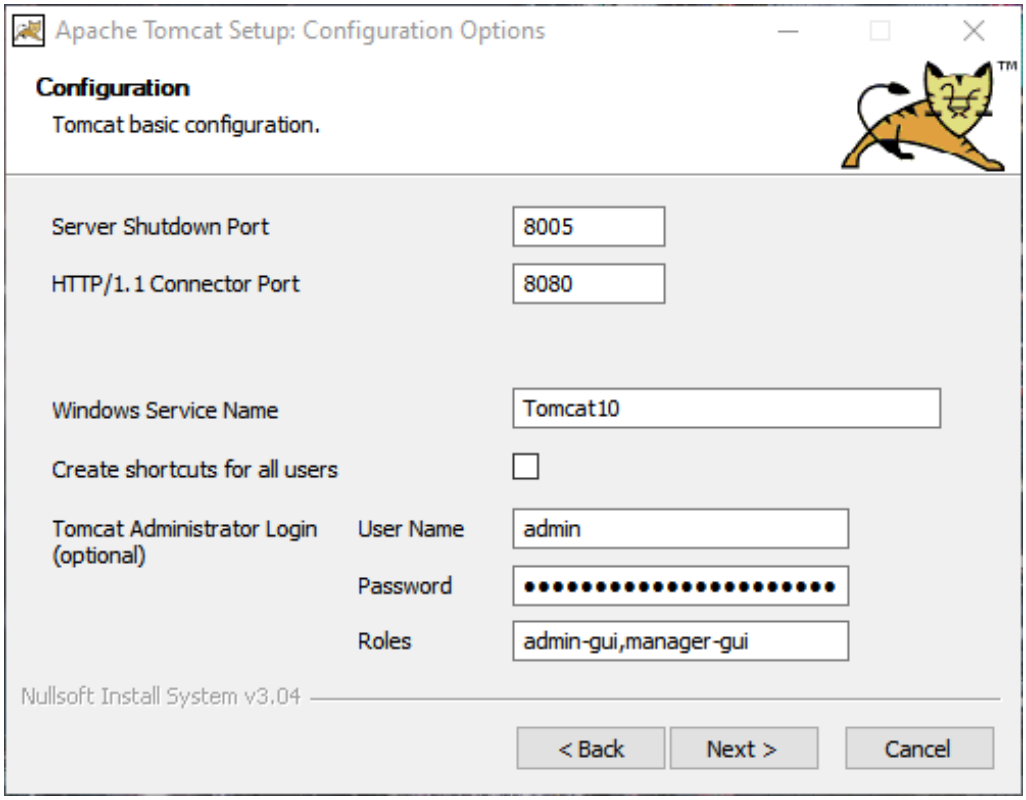

Figure 23: Tomcat 10 Setup Configuration

Most importantly, by default, the Server Shutdown Port is set to -1, or disabled. Here, it is highly recommended to choose another available port like 8005, the default shutdown port. This port is used for the server to wait for a shutdown command. It is not recommended to disable the port while running and stopping the server with the standard shell scripts. When using the Apache Commons Daemon from the taskbar, disabling it is an option [159]. Setting the Server Shutdown port allows the user to use both the Apache Commons Daemon, as well as the standard shell scripts to start and stop the server. The main part of this work describes in detail how Tomcat is used.

Additionally, it is recommended to pick a username and password, since this is the most convenient way to set them. These credentials are used to access the `Manager` web application.

The next part of the installation asks for the path of the Installed Java Runtime Environment, which should be automatically detected.

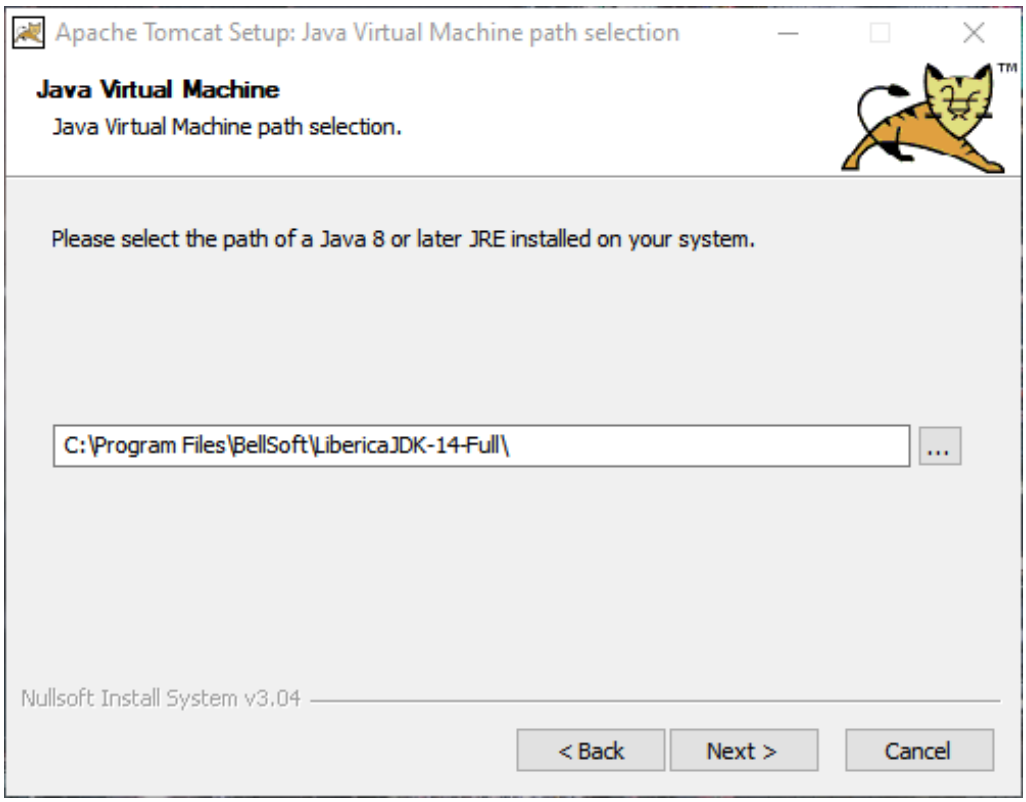

Figure 24: Tomcat 10 Setup Java Virtual Machine

Afterwards, the installation path is chosen. In the figure below, the default path is left unchanged.

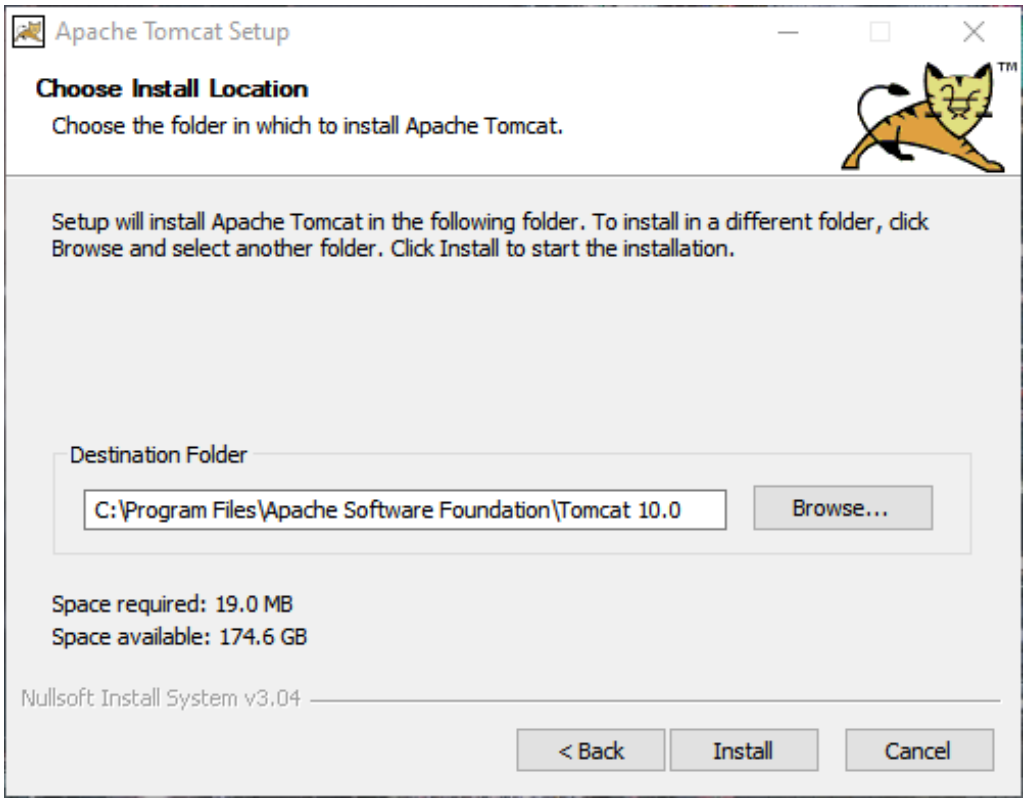

Figure 25: Tomcat 10 Setup Choose Install Location

After confirming the installation path, the software gets installed and the user is greeted with the following window. If desired, the server can be immediately started. Additionally, the Readme file can be viewed.

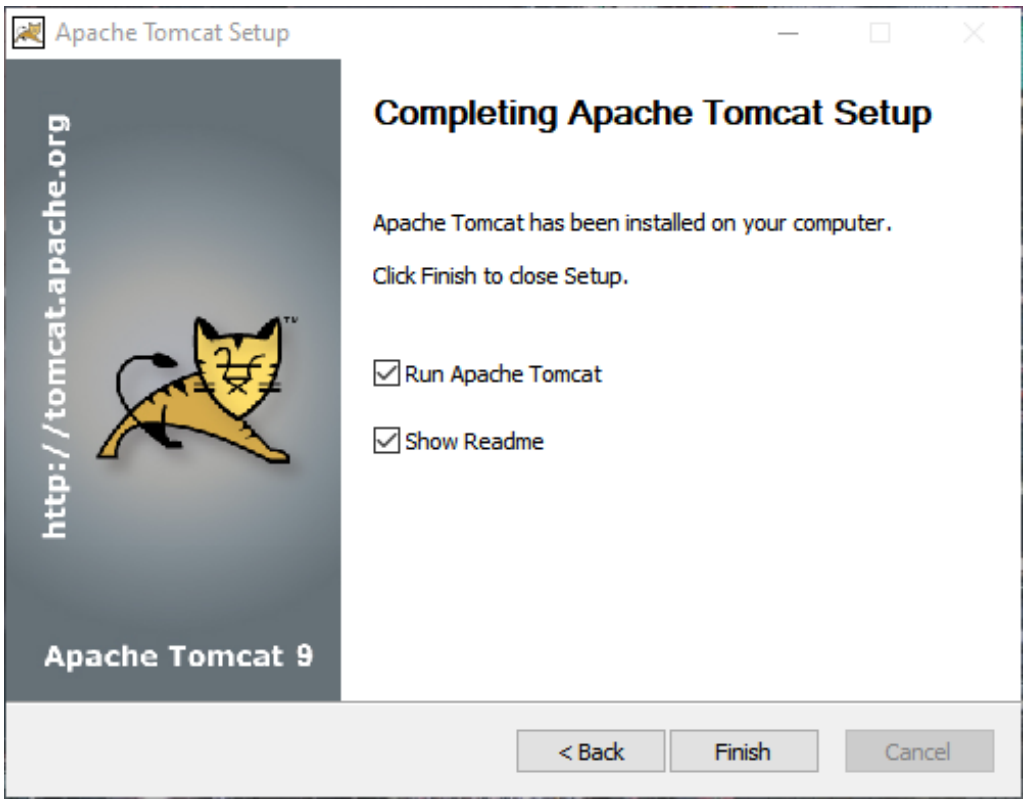

Figure 26: Tomcat 10 Setup Finish

C. Using Tomcat 9

At the time of writing, in January 2021, the Apache Tomcat 10 software was still in a Beta status. Should the reader prefer the stable Version 9, small changes are necessary. The reason being, that Tomcat 10 uses the Jakarta namespace, while Tomcat 9 uses the JavaX namespace.

Instead of using the web applications `helloworld.war` and `treeshop.war`, modified versions for Tomcat 9 can be found in the zip archive included with this work, more precisely in the directory `ZIP_ARCHIVE\javax_for_tomcat9`.

The main difference is the tag library used, since it is already part of the web applications, no further actions are required for the reader to perform. Therefore, should the reader wish to create web applications and prefer using Tomcat 9, it is instrumental to use the file `javax.ScriptTagLibs.jar` instead of `jakarta.ScriptTagLibs.jar`.

Other than the taglib used, the main difference can be observed in the naming of certain classes. For the examples shown in this thesis, the only difference is related to the creation of cookies.

For example, while in the Jakarta version a cookie is created using the class `jakarta.servlet.http.Cookie`, Tomcat 9 uses `javax.servlet.http.Cookie` instead. This name change is not universal; for example, both versions still use `javax.naming.InitialContext` to refer to the `InitialContext` class. Furthermore, since the latest version of Jakarta Mail is added as an external library, both Tomcat 9 and Tomcat 10 use the Jakarta namespace to send e-mails.

As a result, it is a good idea to keep this name change in mind, especially when encountering inexplicable errors messages referring to classes not being found.

D. PostgreSQL

This section will be used to show all necessary steps to install and setup a PostgreSQL database management system, enabling the full functionality of the `treeshop` web application.

D.1. Installation

To begin with, the latest version of the PostgreSQL installer can be downloaded from: `https://www.postgresql.org/download/`. At the time of writing the current version was 13.1. After downloading and executing the file `postgresql-13.1-1-windows-x64.exe`, one is greeted with the following screen on the Microsoft Windows 10 operating system. Usually, it is necessary to allow the application to make changes in a User Account Control popup warning.

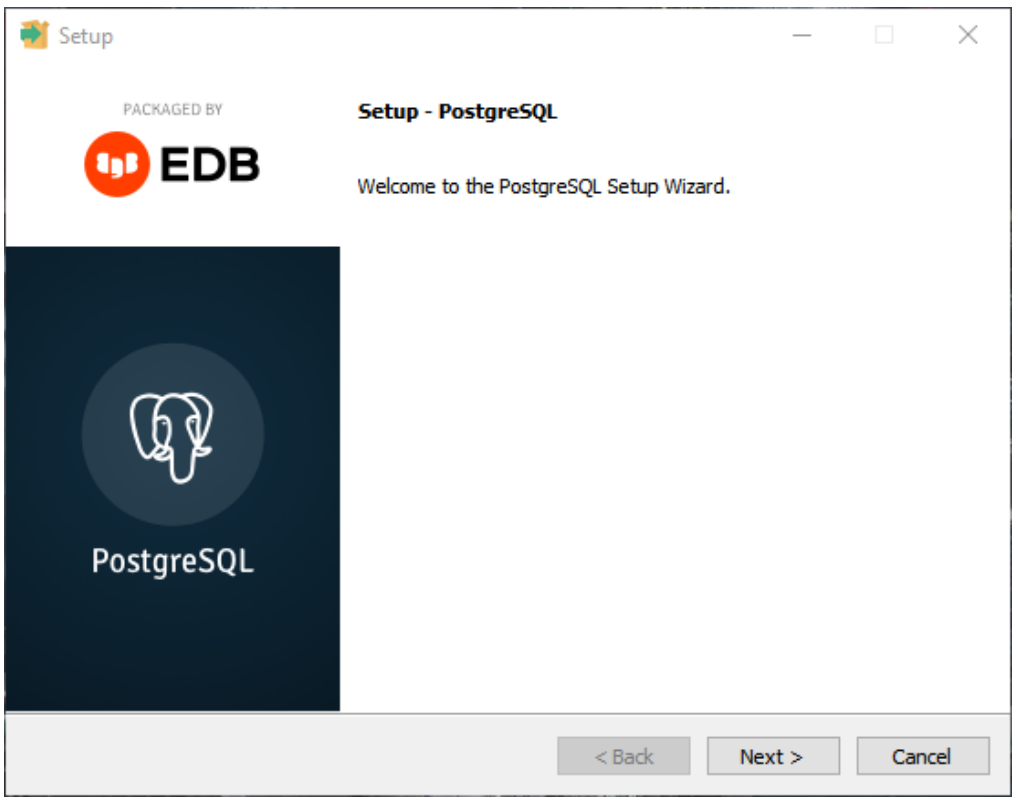

Figure 27: PostgreSQL Setup Welcome

After clicking on the `Next` button, the installation directory is selected. For most machines, the default directory should work just fine.

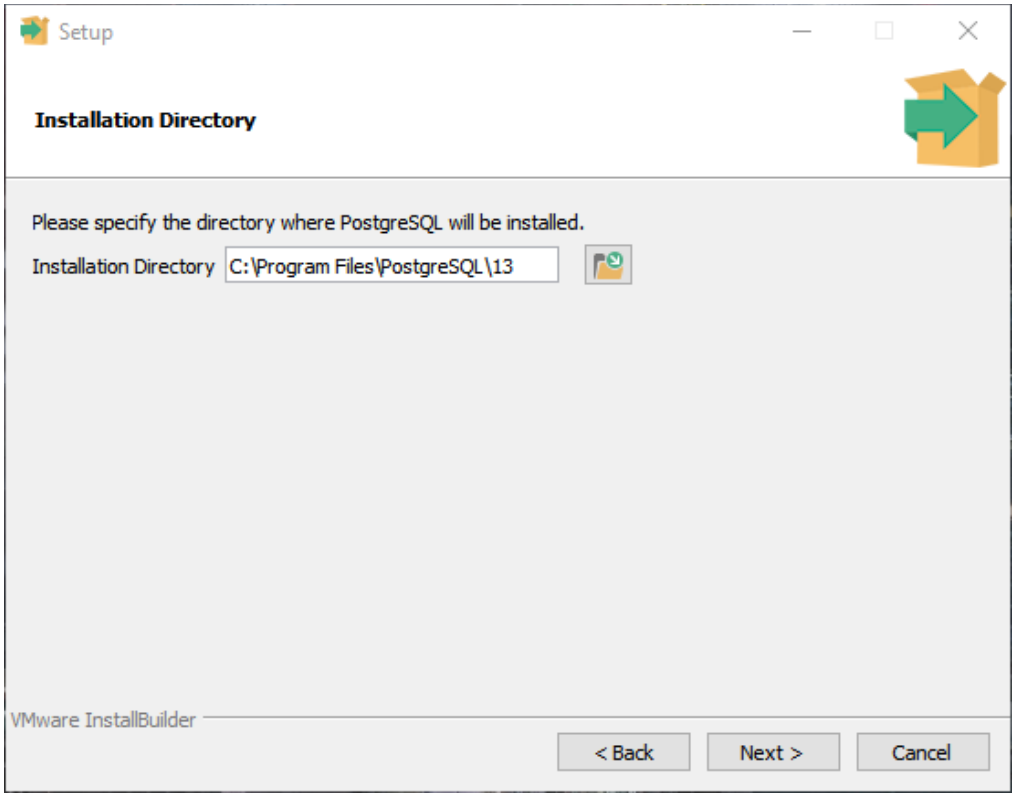

Figure 28: PostgreSQL Setup Installation Directory

Afterwards, the components to be installed are chosen. For the use case described, the only mandatory option is `PostgreSQL Server`. Users preferring a graphical interface might wish to install `pgAdmin 4`. This tool allows to administer the database server from within a web browser. This thesis uses the command line to perform the configuration necessary though.

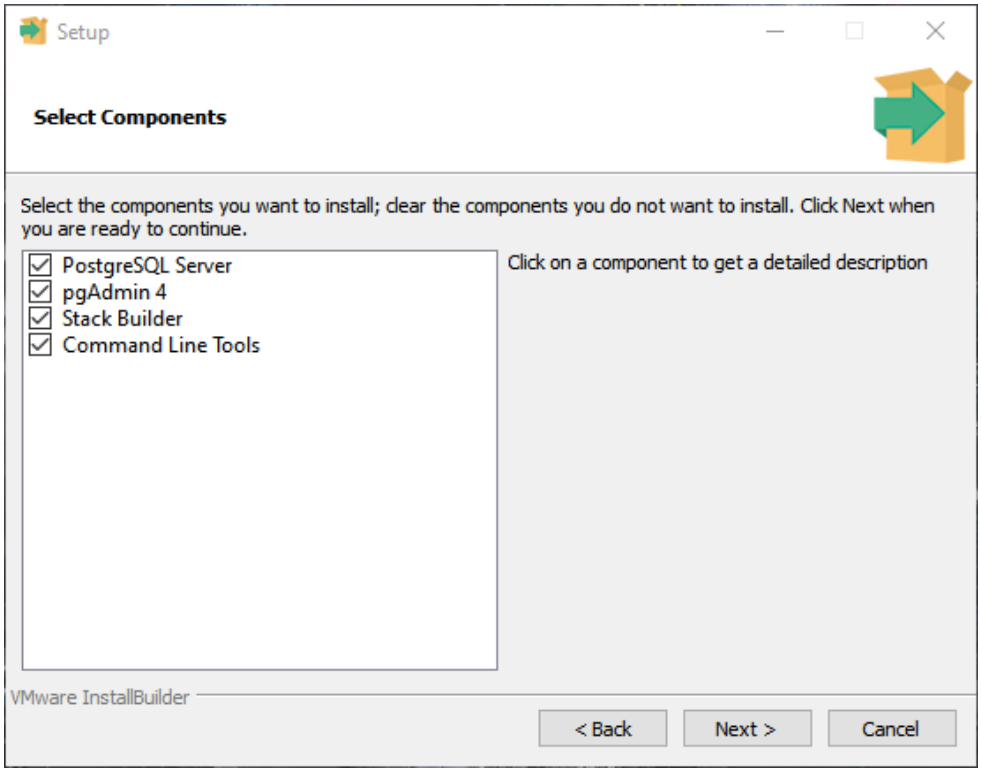

Figure 29: PostgreSQL Setup Select Components

The next window asks for a directory to store the actual data that gets managed in the database management system. Per default, a directory within the default installation directory is chosen. Again, for use on a private machine the default option is recommended.

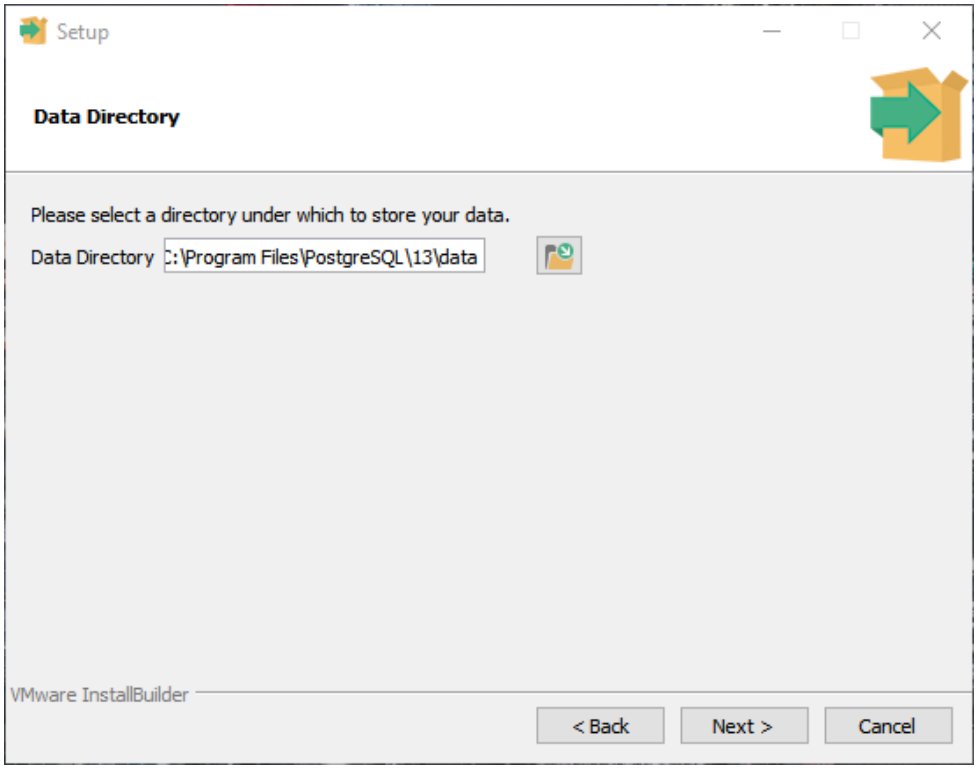

Figure 30: PostgreSQL Setup Data Directory

The next step requires choosing a password for the database superuser account `postgres`. Should the database include sensitive data, it is necessary to choose a strong password, since this account has all possible permissions. For Apache Tomcat, a separate account will be added later, therefore this superuser password is usually not repeated frequently after the configuration has been concluded.

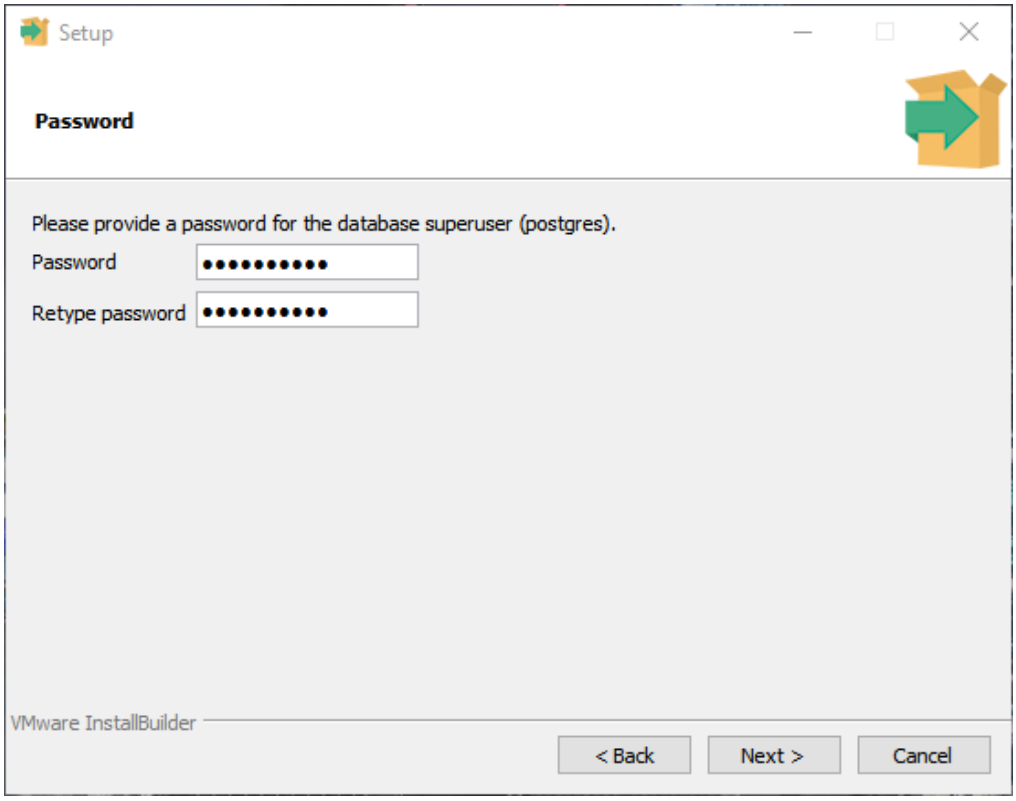

Figure 31: PostgreSQL Setup Password

In the next step, the port, under which the database server is made accessible is chosen. Once again, for the use on a private machine, the default port 5432 does not need to be changed and is also used for the demo web applications.

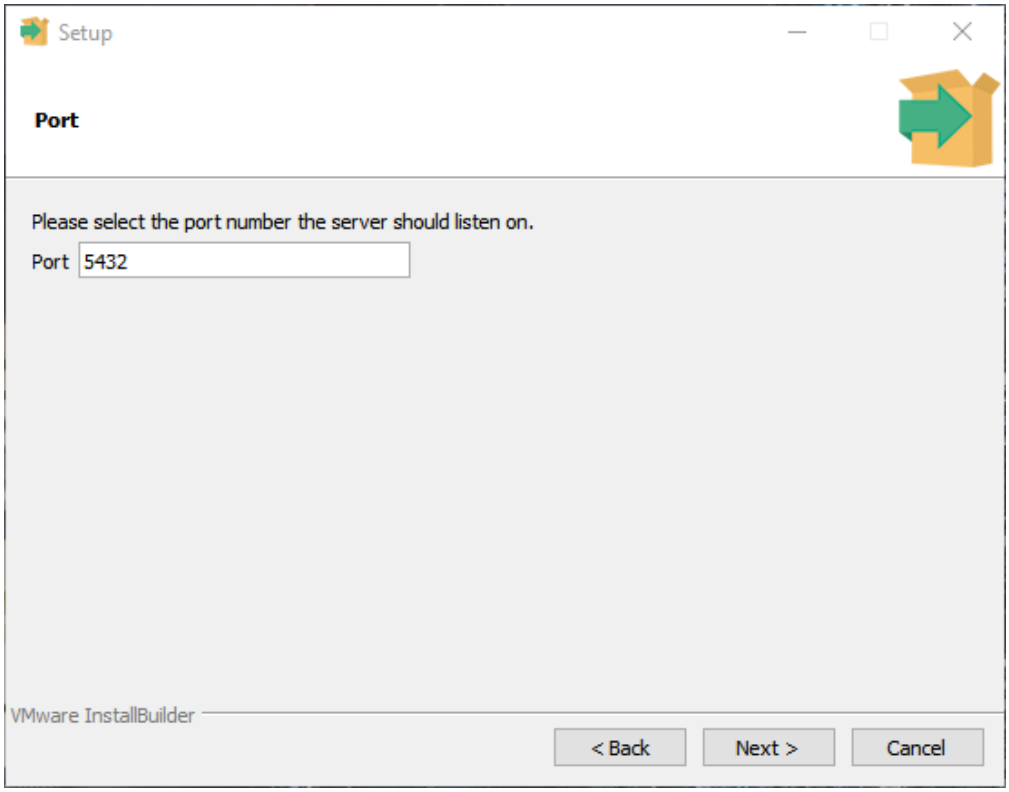

Figure 32: PostgreSQL Setup Port

In the next step, the locale to be used is chosen. This setting affects the language, alphabets, and number formatting used in the database cluster. By choosing the default locale, the locale of the operating system is used [160].

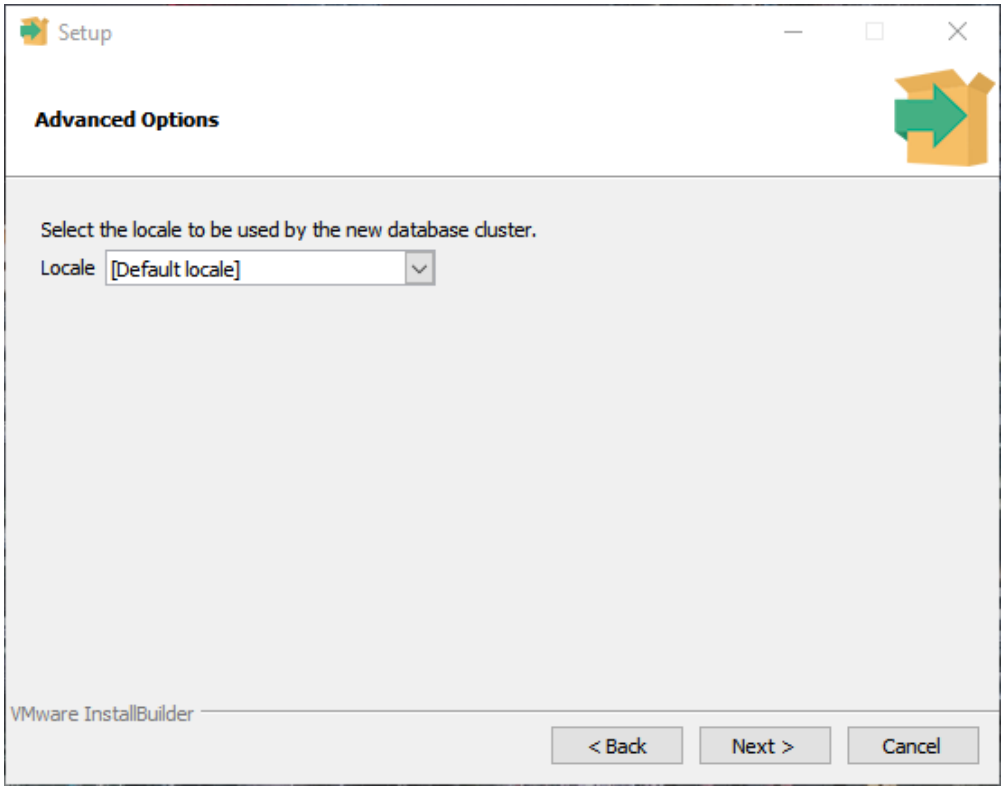

Figure 33: PostgreSQL Setup Advanced Options

The next window summarizes all previously chosen options. The image below shows an installation where all possible components are chosen, and the default parameters have been left unchanged.

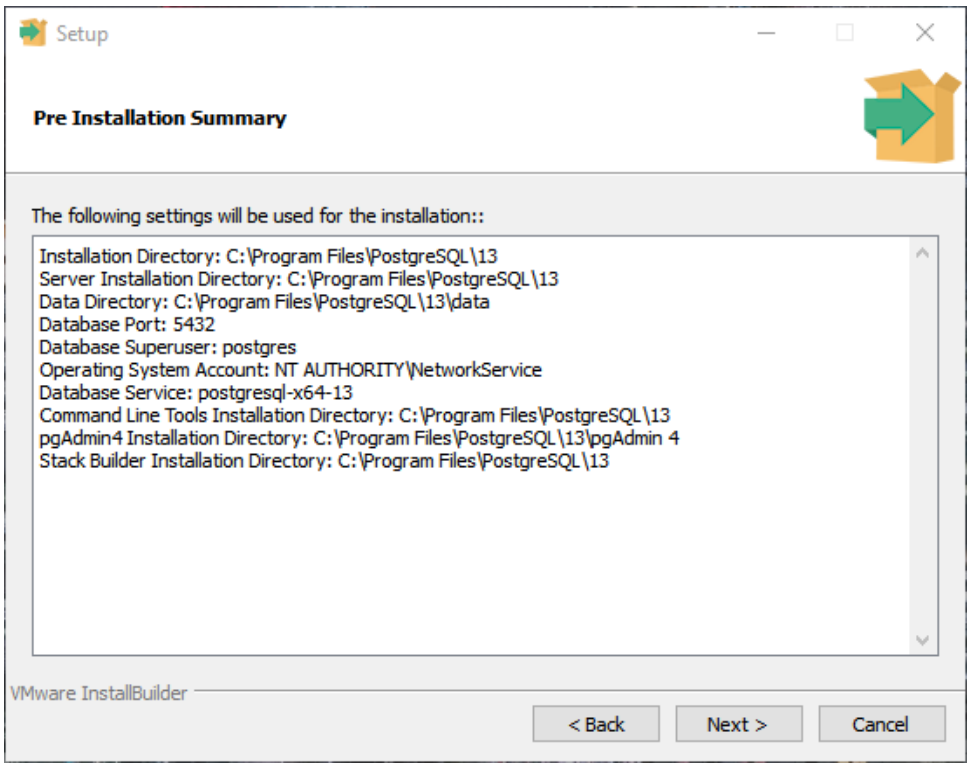

Figure 34: PostgreSQL Setup Pre Installation Summary

The next window informs the user that the setup is ready to start the installation.

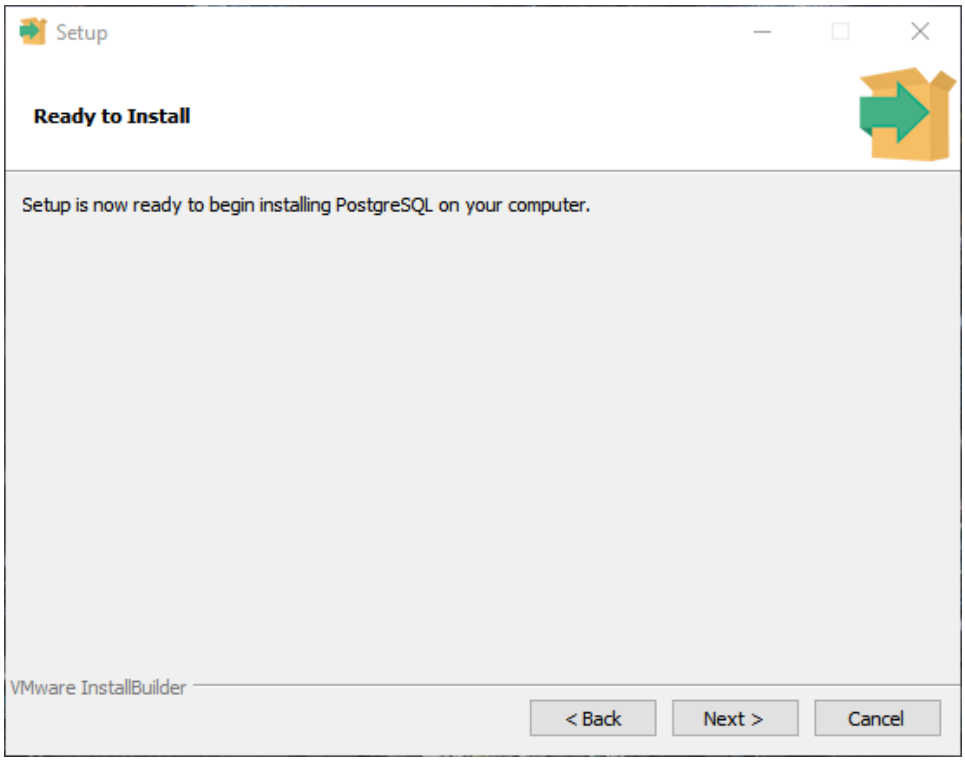

Figure 35: PostgreSQL Setup Ready to Install

After the installation is concluded, one is greeted by the window shown below. The optional Stack Builder feature is not required for the use case described.

Figure 36: PostgreSQL Setup Finish

D.2. Setting Up a PostgreSQL Environment Variable

Even though, the database comes with a graphic administration interface called `pgAdmin 4`, this tutorial uses the command line to work with the database. To use PostgreSQL from within the Windows PowerShell or the Command Prompt, it is necessary to add the database management system to the system's environment variables.

To accomplish this on Windows 10, first the `About your PC` window must be opened. This window can be easily found by typing `about` in the Windows Search. After clicking on `Advanced system settings` on the right, a smaller window with system properties is opened.

After clicking on `Environment Variables`, a new window is opened. By selecting the Variable `Path` on the lower half under `System Variables` and clicking on the `Edit` button, the environment variables can be accessed.

By clicking on `New` and then `Browse`, the `bin` folder from the PostgreSQL installation directory needs to be chosen: `C:\Program Files\PostgreSQL\13\bin`. Afterwards, the choice is confirmed by clicking on `OK`. After restarting all currently open Windows Powershell and Command Prompt windows, the PostgreSQL database management system can be accessed with the command `psql`.

D.3. Starting the Database Server

Like the Apache Tomcat software, the PostgreSQL server which allows access to the databases is started and stopped by a corresponding Windows Service. By typing `services.msc` in the Command Prompt or the Windows Powershell, all Windows Services are listed. By right-clicking on the entry `postresql-x64-13` the server can be started or stopped.

D.4. Setting Up a Database for `treeshop`

This section will list all commands necessary, to prepare a database for the `treeshop` web application. As an alternative to this guide, a support web page included with both web applications offers a convenient way to quickly copy and paste all shown commands. It can be accessed from: `http://localhost:8080/helloworld/support` and is highly recommended.

To begin with, the command: `psql postgres postgres` allows access to the database management system. `psql` is the environmental variable used to communicate to the command line, that PostgreSQL is to be addressed. The repetition of the variable `postgres` signals the database management system to access the default database `postgres` as the superuser `postgres`. Afterwards, the password for the superuser account, which has been defined during the installation process, needs to be entered. In this context, the command line will give no feedback for letters entered if they are related to passwords. At this point, one should be greeted by the following prompt:

```
PS C:\> psql postgres postgres
Password for user postgres:
psql (13.1)
Type "help" for help.

postgres=#
```

Listing 40: PostgreSQL Setup Start Database Management System

Now, a new database called `shop` is created. It is used exclusively by the `treeshop` web application.

```
postgres=# CREATE DATABASE shop;
CREATE DATABASE
postgres=#
```

Listing 41: PostgreSQL Setup Create Database shop

Afterwards, a connection to the newly created database is established.

```
postgres=# \connect shop;
You are now connected to database "shop" as user "postgres".
shop=#
```

Listing 42: PostgreSQL Setup Connect to Database shop

Next, the three tables required for `treeshop` to function are created. While `tree` holds all products, `customer` contains all registered customers. The table `cart` is used to connect them.

```
shop=# CREATE TABLE tree(
shop(# tree_id serial PRIMARY KEY,
shop(# name varchar (40) NOT NULL,
shop(# price varchar (20) NOT NULL,
shop(# height varchar (20),
shop(# picture varchar (100)
shop(# );
CREATE TABLE
shop=#
```

Listing 43: PostgreSQL Setup Create Table tree

```
shop=# CREATE TABLE customer(
shop(# customer_id serial PRIMARY KEY,
shop(# username varchar (40) NOT NULL,
shop(# password varchar (100) NOT NULL,
shop(# receives_mail boolean DEFAULT FALSE
shop(# );
CREATE TABLE
shop=#
```

Listing 44: PostgreSQL Setup Create Table customer

```
shop=# CREATE TABLE cart (
shop(# tree_id integer REFERENCES tree ON UPDATE CASCADE ON DELETE CASCADE,
shop(# customer_id integer REFERENCES customer ON UPDATE CASCADE ON DELETE CASCADE,
shop(# quantity integer,
shop(# PRIMARY KEY (tree_id, customer_id)
shop(# );
CREATE TABLE
shop=#
```

Listing 45: PostgreSQL Setup Create Table cart

All users and products have a unique id assigned to them using `serial`. This is done automatically by the `sequence` object, made available by PostgreSQL. It creates a unique identifier for each new row [161].

In the next step, the user `cattus` is created and given the password `tomtom12`. These credentials are going to be used by Tomcat to access the database.

```
shop=# CREATE USER cattus WITH ENCRYPTED PASSWORD 'tomtom12';
CREATE ROLE
shop=#
```

Listing 46: PostgreSQL Setup Create User cattus

Since no schema has been defined, all previously created tables are assigned to the schema `public`. The following commands grants the newly created user the necessary rights to operate on the related tables.

```
shop=# GRANT ALL ON ALL TABLES IN SCHEMA public TO cattus;
GRANT
shop=#
```

Listing 47: PostgreSQL Setup Grant All Rights to cattus

Finally, special permissions need to be given, so that the new user may work on tables using sequences. Even though sequences look like fields, they are single-row tables that require explicit permission to perform functions on them. Each time a new row is added, a function is performed by the database management system, to auto increment the sequence number, which is stored in form of a `bigint` [162]. First, the rights are granted for the sequence of the `tree` table.

```
shop=# GRANT ALL ON SEQUENCE tree_tree_id_seq TO cattus;
GRANT
shop=#
```

Listing 48: PostgreSQL Setup Grant Sequence Rights tree_id to cattus

Afterwards, the operation is repeated for the table `customer`.

```
shop=# GRANT ALL ON SEQUENCE customer_customer_id_seq TO cattus;
GRANT
shop=#
```

Listing 49: PostgreSQL Setup Grant Sequence Rights customer_id to cattus

In the next step, six example products are added to the table `tree`.

```
shop=# INSERT INTO tree (name, price, height, picture) VALUES
shop-# ('Oak', 50, 20, '/files/Oak.jpg'),
shop-# ('Birch', 100, 15, '/files/Birch.jpg'),
shop-# ('Willow', 150, 10, '/files/Willow.jpg'),
shop-# ('Beech', 180, 40, '/files/Beech.jpg'),
shop-# ('Pine', 250, 55, '/files/Pine.jpg'),
shop-# ('Maple', 300, 40, '/files/Maple.jpg');
INSERT 0 6
shop=#
```

Listing 50: PostgreSQL Setup Insert Products in tree

Afterwards, three example customer accounts are created. All entries follow the same password convention: the password for `bigspender@quickmail.com` is `bigspender`.

```
shop=# INSERT INTO customer (username, password, receives_mail) VALUES
shop-# ('bigspender@quickmail.com', '$2a$12$21bWzhN.SG22WomYuO.w5uh0djvgsZPC5YZMasE.010.Zteeww8Eq', TRUE),
shop-# ('smallspender@slowmail.com', '$2a$12$gA49Tgpzh9KL3aKTonV8nuxvzyURj4xQ31eQWDNfZ5uM/0HmPGMpW', TRUE),
shop-# ('mediumspender@sendbypigeon.com', '$2a$12$xr4fQfB1VWqGfKChfUSVl.x2VrCc../FaXGPSAP173Tgm3cUsfy.i', TRUE);
INSERT 0 3
shop=#
```

Listing 51: PostgreSQL Setup Insert Users in customer

This concludes the setup process for the database. The `treeshop` should be fully functional now.

E. MailHog Installation Guide

The MailHog software can be downloaded from the following web page: `https://github.com/mailhog/MailHog`. By clicking on `Releases` on the right, both a 64-bit and a 32-bit version can be found.

On the Microsoft Windows operation system, once downloaded, the file `MailHog_windows_amd64.exe` / `MailHog_windows_amd32.exe` will open a Command Prompt window on execution. As long as this window remains open, the software is running.

After configuring Jakarta Mail to send e-mails with the SMTP server on the localhost and port 1025, all e-mails sent can be viewed from a web browser, using the URL: `http://localhost:8025`. The username and password choice does not matter, any values are accepted.

F. Debug Code Snippet

The following lines of code, written by Rony G. Flatscher, can be placed at the top and bottom of a script, creating a detailed output of any exceptions that occur when the program is run. This is particularly useful to determine problems related to database operations.

```
SIGNAL ON SYNTAX

/* CODE */

RETURN

SYNTAX:                    -- label to jump to, if syntax condition gets raised
above
   co=condition("object")  -- get condition object
       -- get Java exception chain as a Rexx string, insert "<br>" after LF
("0a"x)
   strChain=ppJavaExceptionChain(co)~changeStr("0a"x, "0a"x "<br>")
   .error~say(strChain)    -- write to error stream
   say strChain            -- write to output stream (generates as HTML text)
   raise propagate         -- propagate exception (recreate exception in caller)
::REQUIRES "BSF.CLS" --enable Java support
```

Listing 52: Debug Code Snippet

G. SSL/TLS E-Mail Utility

The following utility, written by Rony G. Flatscher, can be added to an ooRexx program to make the routine `sendMailSSL` available. It offers functionality to send e-mails using the SSL/TLS protocol. Afterwards, e-mails can be effortlessly sent by calling this routine and giving it a set of input parameters in the following order: `CALL sendMailSSL fromAddress, password, toAdress, subject, text`

```
#!/usr/bin/env rexx
/** Utility to send messages via SSL/TLS using either Java EE 8 or lower, or
Jakarta EE 9 or higher.
    (c) 2021 Rony G. Flatscher, Apache License 2.0
*/

   /* load/import and save the classes we use in this package (Rexx program)
      each entry in an environment directory can be fetched via its
      environment symbol (just prepend a dot to the name) */
namespace=determineNameSpace()   -- test which mail package we have, javax. or
jakarta.
if namespace~isNil then
   raise syntax 40.900 additional("Cannot find the mail package for Java EE 8 or
lower, nor for Jakarta EE 9 or higher")

pkgLocal=.context~package~local  -- get this package local object
pkgLocal~namespace=namespace     -- save namespace
   -- Java-Klassen laden und speichern
pkgLocal~Session              =bsf.loadClass(namespace".mail.Session")
pkgLocal~Message.RecipientType=bsf.loadClass(namespace".mail.Message$RecipientType
")
pkgLocal~Transport            =bsf.loadClass(namespace".mail.Transport")
pkgLocal~Authenticator        =bsf.loadClass(namespace".mail.Authenticator")

   -- imported classes can be used as if they were ooRexx classes, they understand
the "new" message
pkgLocal~PasswordAuthentication=bsf.importClass(namespace".mail.PasswordAuthentica
tion")
   -- the following classes are from the JRE (Java runtime environment)
pkgLocal~MimeMessage
=bsf.importClass(namespace".mail.internet.MimeMessage")
pkgLocal~InternetAddress
=bsf.importClass(namespace".mail.internet.InternetAddress")   -- from EE

   -- create a proxy class which will reroute Java invocations of
"getPasswordAuthentication" to the RexxProxy
pkgLocal~proxiedAuthenticator=bsf.createProxyClass(.Authenticator, ,
"getPasswordAuthentication")


::requires BSF.CLS   -- get Java bridge

/* ---------------------------------------------------------------------- */
::routine determineNameSpace public
  signal on syntax name no_javax
  clz=bsf.loadClass("javax.mail.Session")
  return "javax"     -- we were able to load that class!

no_javax:
```

```
  signal on syntax name no_jakarta
  clz=bsf.loadClass("jakarta.mail.Session")
  return "jakarta"    -- we were able to load that class!

no_jakarta:
  return .nil         -- indicate whe have no access to the mail package


/* ============================================================================ */

::routine sendMailSSL public
  use strict arg fromAddress, password, toAddress, subject, text

  props=.bsf~new("java.util.Properties")
  props~put("mail.smtp.host", "smtp.gmail.com")
  props~put("mail.smtp.socketFactory.port", "465")
  props~put("mail.smtp.socketFactory.class", "javax.net.ssl.SSLSocketFactory")
  props~put("mail.smtp.auth", "true")
  props~put("mail.smtp.port", "465")

   -- create the RexxProxy that handles the "getPasswordAuthentication" message
  jRxAuth=BsfCreateRexxProxy(.RexxAuthenticator~new(fromAddress, password))
  jAuth=.proxiedAuthenticator~new(jRxAuth)    -- create Java object and supply
RexxProxy

  session=.Session~getDefaultInstance(props, jAuth)
  msg=.MimeMessage~new(session)
  msg~addRecipient(.Message.RecipientType~to, .InternetAddress~new(toAddress))
  msg~setSubject(subject) -- alternative: msg~subject=subject
  msg~setText(text)       -- alternative: msg~text   =text
  .Transport~bsf.invoke("send",msg)

/* ============================================================================ */
-- {jakarta|javax}.mail.Authenticator maintains "from" address and "password"
::class RexxAuthenticator         -- implements "getPasswordAuthentication" from
abstract "javax.mail.Authenticator" class

::method init                     -- ooRexx constructor
  expose passwordAuthentication
  use strict arg from, password
   -- create the password authentication object
  passwordAuthentication=.bsf~new("javax.mail.PasswordAuthentication", from,
password)

::method getPasswordAuthentication  -- returns the password authentication object
 expose passwordAuthentication
 return passwordAuthentication



/* ============================================================================ */
::routine sendMailSSL.getPackageLocal  public   -- for debugging purposes, return
this package local object
  return .context~package~local
```

Listing 53: SSL/TLS E-Mail Utility

References

[1] World Wide Web Consortium, "Tim Berners-Lee," World Wide Web Consortium, 16 July 2020. [Online]. Available: https://www.w3.org/People/Berners-Lee/. [Accessed 10 September 2020].

[2] R. Fielding, J. Gettys, M. J., F. H., L. Masinter, P. Leach and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1," Internet Engineering Task Force, June 19999. [Online]. Available: https://tools.ietf.org/html/rfc2616. [Accessed 10 September 2020].

[3] Rexx Language Association, " About Open Object Rexx," Rexx Language Association, [Online]. Available: https://www.oorexx.org/about.html. [Accessed 26 December 2020].

[4] Jakarta Server Pages Team, "Jakarta Server Pages Specification, Version 3.0," Eclipse Foundation, 21 October 2020. [Online]. Available: https://jakarta.ee/specifications/pages/3.0/jakarta-server-pages-spec-3.0.html. [Accessed 10 January 2021].

[5] J. Ousterhout, "Scripting: Higher-Level Programming for the 21st Century," *IEEE Compute,* vol. 31, no. 03, pp. 23-30, 1998. [Online]. Available: https://web.stanford.edu/~ouster/cgi-bin/papers/scripting.pdf. [Accessed 29 January 2021].

[6] R. Sedgewick and K. Wayne, "8.2 Compilers, Interpreters, and Emulators," Princeton University, 24 October 2006. [Online]. Available: https://introcs.cs.princeton.edu/java/82compiler/. [Accessed 22 September 2020].

[7] D. Hemmendinger, " Machine language," Encyclopædia Britannica , 13 October 2016. [Online]. Available: https://www.britannica.com/technology/machine-language. [Accessed 22 September 2020].

[8] R. Toal, "Scripting Languages," Loyola Marymount University, [Online]. Available: https://cs.lmu.edu/~ray/notes/scriptinglangs/. [Accessed 22 September 2020].

[9] J. Gosling and M. Henry, "The Java Language Environment," Oracle, May 1996. [Online]. Available: https://www.oracle.com/java/technologies/language-environment.html. [Accessed 23 September 2020].

[10] Oracle, "About the Java Technology," Oracle, [Online]. Available: https://docs.oracle.com/javase/tutorial/getStarted/intro/definition.html. [Accessed 23 September 2020].

[11] C. Hermansen, "Using external libraries in Java," Opensource.com, 11 February 2020. [Online]. Available: https://opensource.com/article/20/2/external-libraries-java. [Accessed 26 December 2020].

[12] GeeksforGeeks, "Jar files in Java," GeeksforGeeks, 26 May 2017. [Online]. Available: https://www.geeksforgeeks.org/jar-files-java/. [Accessed 26 December 2020].

[13] Oracle, "JSR 223: Scripting for the JavaTM Platform," Oracle, [Online]. Available: https://jcp.org/en/jsr/detail?id=223. [Accessed 26 December 2020].

[14] Oracle, "Java Platform, Standard Edition Java Scripting Programmer's Guide," Oracle, [Online]. Available: https://docs.oracle.com/javase/10/scripting/toc.htm. [Accessed 24 September 2020].

[15] A. Fleck, "Prologue on Program Specification," University of Iowa, [Online]. Available: http://homepage.divms.uiowa.edu/~fleck/spec.html. [Accessed 06 September 2020].

[16] J. O'Conner, "Scripting for the Java Platform," Oracle, July 2006. [Online]. Available: https://www.oracle.com/technical-resources/articles/javase/scripting.html. [Accessed 04 January 2021].

[17] Oracle, "Interface ScriptEngine," Oracle, [Online]. Available: https://docs.oracle.com/javase/10/docs/api/javax/script/ScriptEngine.html. [Accessed 24 September 2020].

[18] GeeksforGeeks, "JavaBean class in Java," GeeksforGeeks, 14 September 2017. [Online]. Available: https://www.geeksforgeeks.org/javabean-class-java/. [Accessed 25 September 2020].

[19] Apache Software Foundation, "BSF FAQ," Apache Software Foundation, 17 October 2011. [Online]. Available: https://commons.apache.org/proper/commons-bsf/faq.html. [Accessed 25 September 2020].

[20] Apache Software Foundation, "BSF Manual," Apache Software Foundation, 17 October 2011. [Online]. Available: https://commons.apache.org/proper/commons-bsf/manual.html. [Accessed 25 September 2020].

[21] S. Weerawarana, M. J. Duftler, S. Ruby, O. Gruber, D. Schwarz and R. G. Flatscher, "Class BSFManager," 13 September 2008. [Online]. Available: http://wi.wu.ac.at:8002/rgf/rexx/bsf4rexx/current/docs/docs.apache.bsf/org/apache/bsf/BSFManager.html. [Accessed 25 September 2020].

[22] Apache Software Foundation, "BSF About," Apache Software Foundation, 2011 October 2011. [Online]. Available: https://commons.apache.org/proper/commons-bsf/index.html. [Accessed 25 September 2020].

[23] R. G. Flatscher, Introduction to REXX and ooRexx, Vienna, Austria: Facultas, 2013.

[24] R. G. Flatscher, "Java Bean Scripting With Rexx," in *Proceedings of the "12th International Rexx Symposium"*, Raleigh, North Carolina, USA, April 30th - May 2nd, 2001. [Online]. Available: http://wi.wu-wien.ac.at:8002/rgf/rexx/orx12/JavaBeanScriptingWithRexx_orx12.pdf. [Accessed 29 January 2021].

[25] R. G. Flatscher, "The Augsburg Version of BSF4Rexx," in *Proceedings of the "The 14th International Rexx Symposium"*, Raleigh, NorthCarolina, USA, May 2003. [Online]. Available:

http://wi.wu.ac.at:8002/rgf/rexx/orx14/2003_orx14_A_BSF_by2.pdf. [Accessed 29 January 2021].

[26] R. G. Flatscher, "The 2019 Edition of BSF4ooRexx," in *Proceedings of the "The 2019 International Rexx Symposium"*, Hursley, Great Britain, September 2019. [Online]. Available: https://www.rexxla.org/events/2019/presentations/201909-04_BSF4ooRexx.pdf. [Accessed 29 January 2021].

[27] R. G. Flatscher, "Camouflaging Java as Object REXX," in *Proceedings of the "2004 International Rexx Symposium"*, Sindelfingen/Böblingen, Germany, May 2004. [Online]. Available: https://www.rexxla.org/events/2004/ronyf2.pdf. [Accessed 29 January 2021].

[28] The Editors of Encyclopaedia Britannica, " Protocol," Encyclopaedia Britannica, 31 August 2018. [Online]. Available: https://www.britannica.com/technology/protocol-computer-science. [Accessed 08 September 2020].

[29] F5, "What Is a Web Server?," F5, [Online]. Available: https://www.nginx.com/resources/glossary/web-server/. [Accessed 21 January 2021].

[30] H.-C. Chua, "HTTP (HyperText Transfer Protocol)," Nanyang Technological University, 20 October 2009. [Online]. Available: https://personal.ntu.edu.sg/ehchua/programming/webprogramming/HTTP_Basics.html. [Accessed 07 January 2021].

[31] World Wide Web Consortium, "HTML 5.2," World Wide Web Consortium, 14 December 2017. [Online]. Available: https://www.w3.org/TR/html52/. [Accessed 27 December 2020].

[32] w3schools, "HTML Introduction," w3schools, [Online]. Available: https://www.w3schools.com/html/html_intro.asp. [Accessed 27 December 2020].

[33] H. W. Lie and B. Bos, "Cascading Style Sheets, level 1," World Wide Web Consortium, 17 December 1996. [Online]. Available: https://www.w3.org/TR/REC-CSS1-961217. [Accessed 15 December 2020].

[34] H.-C. Chua, "Java Server-Side Programming," Nanyang Technological University, October 2012. [Online]. Available: https://www3.ntu.edu.sg/home/ehchua/programming/java/JavaServlets.html. [Accessed 01 September 2020].

[35] R. G. Flatscher, "(BSF4)ooRexx and Java Web Server," in *Proceedings of the "The 2020 International Rexx Symposium"*, Online, September 29th - October 1st 2020. [Online]. Available: https://www.rexxla.org/events/2020/presentations/202011-ooRexxAndJavaWebServers-article.pdf. [Accessed 29 January 2021].

[36] Oracle, "Java Servlet Technology Overview," Oracle, [Online]. Available: https://www.oracle.com/java/technologies/servlet-technology.html. [Accessed 02 September 2020].

[37] A. Singh, "Introduction to Java Servlets," GeeksforGeeks, 23 October 2019. [Online]. Available: https://www.geeksforgeeks.org/introduction-java-servlets/. [Accessed 02 September 2020].

[38] Jakarta Servlet Team, "Jakarta Servlet Specification, Version 5.0," Eclipse Foundation, 07 September 2020. [Online]. Available: https://jakarta.ee/specifications/servlet/5.0/jakarta-servlet-spec-5.0.html. [Accessed 10 January 2021].

[39] N. Freed and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies," Internet Engineering Task Force, November 1996. [Online]. Available: https://tools.ietf.org/html/rfc2045. [Accessed 06 September 2020].

[40] Eclipse Foundation, "Class HttpServlet," Eclipse Foundation, [Online]. Available: https://jakarta.ee/specifications/platform/9/apidocs/jakarta/servlet/http/HttpServlet.html. [Accessed 02 September 2020].

[41] Eclipse Foundation, "Interface Servlet," Eclipse Foundation, [Online]. Available: https://jakarta.ee/specifications/platform/9/apidocs/jakarta/servlet/servlet. [Accessed 02 September 2020].

[42] M. Tyson, "What is JSP? Introduction to JavaServer Pages," InfoWorld, 29 January 2019. [Online]. Available: https://www.infoworld.com/article/3336161/what-is-jsp-introduction-to-javaserver-pages.html. [Accessed 02 September 2020].

[43] Oracle, "JSP Tag Libraries," Oracle, [Online]. Available: https://docs.oracle.com/cd/B14099_19/web.1012/b14014/taglibs.htm#i1012403. [Accessed 28 December 2020].

[44] Oracle, "JSP Scriptlets," Oracle, [Online]. Available: https://docs.oracle.com/javaee/5/tutorial/doc/bnaou.html. [Accessed 28 December 2020].

[45] S. Ryabenkiy, *Java Web Scripting and Apache Tomcat,* Vienna, Austria: Vienna University of Economics and Business, 2010. [Online]. Available: http://wi.wu.ac.at:8002/rgf/diplomarbeiten/BakkStuff/2010/201007_Ryabenkiy/201007_Ryabenkiy_WebScripting_ApacheTomCat_TagLib.pdf. [Accessed 29 January 2021].

[46] M. Tyson, "What is Tomcat? The original Java servlet container," InfoWorld, 19 December 2019. [Online]. Available: https://www.infoworld.com/article/3510460/what-is-apache-tomcat-the-original-java-servlet-container.html. [Accessed 28 December 2020].

[47] TEDBlog, "James Duncan Davidson," TEDBlog, [Online]. Available: https://blog.ted.com/author/duncandavidson/. [Accessed 01 September 2020].

[48] Apache Software Foundation, "The Tomcat Story," Apache Software Foundation, [Online]. Available: https://tomcat.apache.org/heritage.html. [Accessed 01 September 2020].

[49] MuleSoft, "Meet Tomcat Catalina," MuleSoft, [Online]. Available: https://www.mulesoft.com/tcat/tomcat-catalina. [Accessed 28 December 2020].

[50] Apache Software Foundation, "Introduction," Apache Software Foundation, 03 December 2020. [Online]. Available: https://tomcat.apache.org/tomcat-10.0-doc/introduction.html. [Accessed 28 December 2020].

[51] Wikipedians, "Apache Tomcat," Wikipedia, 13 December 2020. [Online]. Available: https://en.wikipedia.org/wiki/Apache_Tomcat. [Accessed 28 December 2020].

[52] Apache Software Foundation, "The Coyote HTTP/1.1 Connector," Apache Software Foundation, [Online]. Available: https://tomcat.apache.org/tomcat-4.1-doc/config/coyote.html. [Accessed 28 December 2020].

[53] The Apache Software Foundation, "The HTTP2 Upgrade Protocol," The Apache Software Foundation, 03 December 2020. [Online]. Available: https://tomcat.apache.org/tomcat-10.0-doc/config/http2.html. [Accessed 28 December 2020].

[54] Apache Software Foundation, "Apache HTTP Server HowTo," Apache Software Foundation, 09 March 2020. [Online]. Available: https://tomcat.apache.org/connectors-doc/webserver_howto/apache.html. [Accessed 28 December 2020].

[55] P. Manh, "The different between Web server, Web container and Application server," GitHub, 01 April 2020. [Online]. Available: https://ducmanhphan.github.io/2020-04-01-The-difference-between-web-server-web-container-application-server/. [Accessed 02 September 2020].

[56] Opensource.com, "What is open source?," Opensource.com, [Online]. Available: https://opensource.com/resources/what-open-source. [Accessed 28 December 2020].

[57] Opensource.org, "Frequently Answered Questions," Opensource.org, [Online]. Available: https://opensource.org/faq. [Accessed 28 December 2020].

[58] Apache Software Foundation, "What is the ASF?," Apache Software Foundation, [Online]. Available: https://www.apache.org/foundation/. [Accessed 01 September 2020].

[59] Apache Software Foundation, "Apache Tomcat," Apache Software Foundation, [Online]. Available: https://tomcat.apache.org/. [Accessed 01 September 2020].

[60] Apache Software Foundation, "Apache License, Version 2.0," Apache Software Foundation, [Online]. Available: https://www.apache.org/licenses/LICENSE-2.0. [Accessed 01 September 2020].

[61] Eclipse Foundation, "About the Eclipse Foundation," Eclipse Foundation, [Online]. Available: https://www.eclipse.org/org/. [Accessed 06 September 2020].

[62] Eclipse Foundation, "Explore Our Members," Eclipse Foundation, [Online]. Available: https://www.eclipse.org/membership/exploreMembership.php. [Accessed 06 September 2020].

[63] A. Tijms, "Transition from Java EE to Jakarta EE," Oracle, 27 February 2020. [Online]. Available: https://blogs.oracle.com/javamagazine/transition-from-java-ee-to-jakarta-ee. [Accessed 02 September 2020].

[64] Oracle, "Java Documentation," Oracle, [Online]. Available: https://docs.oracle.com/en/java/index.html. [Accessed 06 September 2020].

[65] R. Monson-Haefel, "TomEE vs. Tomcat," Tomitribe, 05 December 2019. [Online]. Available: https://www.tomitribe.com/blog/tomee-vs-tomcat/. [Accessed 28 December 2020].

[66] Apache Software Foundation, "Tomcat 10 Software Downloads," Apache Software Foundation, [Online]. Available: https://tomcat.apache.org/download-10.cgi. [Accessed 28 December 2020].

[67] R. G. Flatscher, "'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. [Online]. Available: http://www.rexxla.org/events/2017/presentations/201704-RexxScript-Article.pdf. [Accessed 29 January 2021].

[68] Cloudflare, "What do client side and server side mean? | Client side vs. server side," Cloudflare, [Online]. Available: https://www.cloudflare.com/learning/serverless/glossary/client-side-vs-server-side/. [Accessed 06 January 2021].

[69] MuleSoft, "Tomcat Configuration - A Step By Step Guide," MuleSoft, [Online]. Available: https://www.mulesoft.com/tcat/tomcat-configuration. [Accessed 28 December 2020].

[70] Apache Software Foundation, "Application Developer's Guide," Apache Software Foundation, 03 December 2020. [Online]. Available: https://tomcat.apache.org/tomcat-10.0-doc/appdev/deployment.html. [Accessed 10 December 2020].

[71] G. Shachor, "Tomcat 3.3 User's Guide," Apache Software Foundation, [Online]. Available: https://tomcat.apache.org/tomcat-3.3-doc/tomcat-ug.html#directory_structure. [Accessed 28 December 2020].

[72] Microfocus, "Deploying and Running Your Application," Microfocus, [Online]. Available: https://supportline.microfocus.com/documentation/books/sx22sp1/pidepl.htm. [Accessed 29 December 2020].

[73] JavaTpoint, "War File," JavaTpoint, [Online]. Available: https://www.javatpoint.com/war-file. [Accessed 29 December 2020].

[74] Baeldung, "How to Deploy a WAR File to Tomcat," Baeldung, 12 February 2020. [Online]. Available: https://www.baeldung.com/tomcat-deploy-war. [Accessed 29 December 2020].

[75] Uniface, "Creating and Deploying a Web Application WAR File," Uniface, [Online]. Available: https://u.uniface.info/docs/1000/uniface/webApps/webDeployment/Prepare_your_Web_environment.htm. [Accessed 29 December 2020].

[76] FileInfo, ".EAR File Extension," FileInfo, 22 March 2019. [Online]. Available: https://fileinfo.com/extension/ear. [Accessed 29 December 2020].

[77] Microsoft, "Introduction to Windows Service Applications," Microsoft, 30 March 2017. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/framework/windows-services/introduction-to-windows-service-applications. [Accessed 18 September 2020].

[78] A. Sharma, "What is Local Host?," GeeksforGeeks, 09 August 2019. [Online]. Available: https://www.geeksforgeeks.org/what-is-local-host/. [Accessed 13 September 2020].

[79] K. Vijay Kulkarni, "14 common network ports you should know," Red Hat, 04 October 2018. [Online]. Available: https://opensource.com/article/18/10/common-network-ports. [Accessed 17 September 2020].

[80] Apache Software Foundation, "Manager App How-To," Apache Software Foundation, 03 December 2020. [Online]. Available: https://tomcat.apache.org/tomcat-10.0-doc/manager-howto.html. [Accessed 29 December 2020].

[81] MuleSoft, "The Tomcat Web app Quick Reference Guide," MuleSoft, [Online]. Available: https://www.mulesoft.com/tcat/tomcat-webapp. [Accessed 29 December 2020].

[82] R. Nazarov, "Tomcat web.xml Configuration Example," Java Code Geeks, 18 March 2015. [Online]. Available: https://examples.javacodegeeks.com/enterprise-java/tomcat/tomcat-web-xml-configuration-example/. [Accessed 10 December 2020].

[83] Eclipse Foundation, "Interface HttpSession," Eclipse Foundation, 2019. [Online]. Available: https://jakarta.ee/specifications/servlet/4.0/apidocs/javax/servlet/http/HttpSession.html. [Accessed 21 October 2020].

[84] R. Ishida, "Character encodings for beginners," W3C, 16 April 2015. [Online]. Available: https://www.w3.org/International/questions/qa-what-is-encoding. [Accessed 21 October 2020].

[85] O. Thereaux, "Don't forget to add a doctype," World Wide Web Consortium, 20 August 2002. [Online]. Available: https://www.w3.org/QA/Tips/Doctype. [Accessed 22 October 2020].

[86] webhint, "Use charset `utf-8`," webhint, [Online]. Available: https://webhint.io/docs/user-guide/hints/hint-meta-charset-utf-8/. [Accessed 14 December 2020].

[87] Maggie, "Why is <meta charset="utf-8"> important?," DEV, 19 October 2020. [Online]. Available: https://dev.to/maggiecodes_/why-is-lt-meta-charset-utf-8-gt-important-59hl. [Accessed 14 December 2020].

[88] N. Lengyel, *BSF4ooRexx: JSP with javax.script Languages,* Vienna, Austria: Vienna University of Economics and Business, 2020. [Online]. Available: wi.wu.ac.at:8002/rgf/diplomarbeiten/Seminararbeiten/2020/202001_Lengyel_BSF4ooRexx-JSP.pdf. [Accessed 29 January 2021].

[89] C. Singh, "Jsp Implicit Objects," BeginnersBook, [Online]. Available: https://beginnersbook.com/2013/11/jsp-implicit-objects/. [Accessed 22 October 2020].

[90] Apache Software Foundation, "Class JspWriter," Apache Software Foundation, [Online]. Available: https://tomcat.apache.org/tomcat-7.0-doc/jspapi/javax/servlet/jsp/JspWriter.html. [Accessed 22 October 2020].

[91] W3Schools, "HTML <link> Tag," W3Schools, [Online]. Available: https://www.w3schools.com/tags/tag_link.asp. [Accessed 14 December 2020].

[92] W3Schools, "HTML File Paths," W3Schools, [Online]. Available: https://www.w3schools.com/html/html_filepaths.asp. [Accessed 14 December 2020].

[93] R. G. Flatscher, "SourceForge BSF4ooRexx Taglibs Readme.md," 03 February 2021. [Online]. Available: https://sourceforge.net/projects/bsf4oorexx/files/Sandbox/rgf/taglibs/beta/. [Accessed 05 February 2021].

[94] B. Bos, T. Çelik, I. Hickson and H. W. Lie, "Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification," World Wide Web Consortium, 12 April 2016. [Online]. Available: https://www.w3.org/TR/CSS2/. [Accessed 15 December 2020].

[95] FileCloud, "Tech tip: How to do hard refresh in Chrome, Firefox and IE?," FileCloud, 06 March 2015. [Online]. Available: https://www.getfilecloud.com/blog/2015/03/tech-tip-how-to-do-hard-refresh-in-browsers/. [Accessed 04 January 2021].

[96] R. G. Flatscher, "External BSF4ooRexx Functions - Overview," 08 December 2010. [Online]. Available: http://wi.wu-wien.ac.at:8002/rgf/rexx/bsf4oorexx/current/additionalResources/refcardBSF4ooRexx.pdf. [Accessed 14 December 2020].

[97] Javatpoint, "welcome-file-list in web.xml," Javatpoint, [Online]. Available: https://www.javatpoint.com/welcome-file-list. [Accessed 17 December 2020].

[98] A. Barth, "HTTP State Management Mechanism," Internet Engineering Task Force, April 2011. [Online]. Available: https://tools.ietf.org/html/rfc6265. [Accessed 27 September 2020].

[99] Eclipse Foundation, "Interface HttpServletRequest," Eclipse Foundation, [Online]. Available: https://jakarta.ee/specifications/servlet/4.0/apidocs/javax/servlet/http/httpservletrequest. [Accessed 22 October 2020].

[100] Eclipse Foundation, "Class Cookie," Eclipse Foundation, [Online]. Available: https://jakarta.ee/specifications/servlet/4.0/apidocs/javax/servlet/http/Cookie.html. [Accessed 22 October 2020].

[101] W. D. Ashley, R. G. Flatscher, M. Hessling, R. McGuire, M. Miesfeld, L. Peedin, R. Tammer and J. Wolfers, "Built-in Functions," Rexx Language Association, 14 August 2009. [Online]. Available: https://www.oorexx.org/docs/rexxref/x23579.htm. [Accessed 22 October 2020].

[102] D. Ragget, A. Le Hors and I. Jacobs, "HTML 4.01 Specification," World Wide Web Consortium, 24 December 1999. [Online]. Available: https://www.w3.org/TR/html401/. [Accessed 14 December 2020].

[103] F. Bohórquez, "HTML Forms: The Action Attribute," Career Karma, 12 August 2020. [Online]. Available: https://careerkarma.com/blog/html-form-action/. [Accessed 16 December 2020].

[104] W3Schools, "HTML <label> Tag," W3Schools, [Online]. Available: https://www.w3schools.com/tags/tag_label.asp. [Accessed 04 January 2021].

[105] w3schools, "HTML <input> required Attribute," w3schools, [Online]. Available: https://www.w3schools.com/tags/att_input_required.asp. [Accessed 20 December 2020].

[106] R. G. Flatscher and G. Müller, "ooRexx 5 Yielding Swiss Army Knife Usability," in *The Proceedings of the Rexx Symposium for Developers and Users*, Hursley, Great Britain, 2019. [Online]. Available: https://epub.wu.ac.at/7412/1/201909-03_SwissArmyKnife_article.pdf. [Accessed 29 January 2021].

[107] V. Kaplan, "Compiling Scripts to Get Compiled Language Performance," EPS Software Corp/CODE Magazine, [Online]. Available: https://www.codemag.com/Article/2001071/Compiling-Scripts-to-Get-Compiled-Language-Performance. [Accessed 06 January 2021].

[108] baeldung, "Handling Cookies and a Session in a Java Servle," baeldung, 28 February 2020. [Online]. Available: https://www.baeldung.com/java-servlet-cookies-session. [Accessed 23 October 2020].

[109] M. Tyson, "What is JDBC? Introduction to Java Database Connectivity," InfoWorld, 11 April 2011. [Online]. Available: https://www.infoworld.com/article/3388036/what-is-jdbc-introduction-to-java-database-connectivity.html. [Accessed 12 November 2020].

[110] Z. Su and G. Wassermann, "The Essence of Command Injection Attacks in Web Applications," in *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Charleston, South Carolina, USA, January 11-13, 2006. [Online]. Available: https://web.cs.ucdavis.edu/~su/publications/popl06.pdf. [Accessed 29 January 2021].

[111] M. Aboagye, "Improve database performance with connection pooling," Stack Overflow, 14 October 2020. [Online]. Available: https://stackoverflow.blog/2020/10/14/improve-database-performance-with-connection-pooling/. [Accessed 15 November 2020].

[112] Apache Software Foundation, "JNDI Datasource How-To," Apache Software Foundation, 06 October 2020. [Online]. Available: https://tomcat.apache.org/tomcat-9.0-doc/jndi-datasource-examples-howto.html. [Accessed 12 November 2020].

[113] M. van Steen and A. S. Tanenbaum, "A brief introduction to distributed systems," *Computing,* vol. 98, no. 10, pp. 967-1009, 2016. [Online]. Available: https://link.springer.com/article/10.1007/s00607-016-0508-7. [Accessed 29 January 2021].

[114] F. T. Marchese, "Naming," Pace University Seidenberg School of CSIS, [Online]. Available: http://csis.pace.edu/~marchese/CS865/Lectures/Chap5/Chapter5.htm. [Accessed 11 November 2020].

[115] T. Sundsted, "JNDI overview, Part 2: An introduction to directory services," InfoWorld, 21 February 2000. [Online]. Available: https://www.infoworld.com/article/2076901/jndi-overview--part-2--an-introduction-to-directory-services.html. [Accessed 11 November 2020].

[116] S. Claridge, "Serving static content (including web pages) from outside of the WAR using Apache Tomcat," More Of Less, 04 April 2014. [Online]. Available: https://www.moreofless.co.uk/static-content-web-pages-images-tomcat-outside-war/. [Accessed 21 December 2020].

[117] Apache Software Foundation, "Class Loader How-To," Apache Software Foundation, 03 December 2020. [Online]. Available: https://tomcat.apache.org/tomcat-10.0-doc/class-loader-howto.html. [Accessed 10 December 2020].

[118] Apache Software Foundation, "JNDI Resources How-To," Apache Software Foundation, 06 October 2020. [Online]. Available: https://tomcat.apache.org/tomcat-9.0-doc/jndi-resources-howto.html. [Accessed 12 November 2020].

[119] T. Sundsted, "JNDI overview, Part 1: An introduction to naming services," InfoWorld, 01 January 2000. [Online]. Available: https://www.infoworld.com/article/2076888/jndi-overview--part-1--an-introduction-to-naming-services.html. [Accessed 11 November 2020].

[120] Oracle, "Interface Statement," Oracle, [Online]. Available: https://cr.openjdk.java.net/~iris/se/15/latestSpec/api/java.sql/java/sql/Statement.html. [Accessed 18 November 2020].

[121] Oracle, "Interface ResultSet," Oracle, [Online]. Available: https://cr.openjdk.java.net/~iris/se/15/latestSpec/api/java.sql/java/sql/ResultSet.html. [Accessed 18 November 2020].

[122] J. Holý and M. Mære, "JDBC: What resources you have to close and when?," DZone, 13 February 2013. [Online]. Available: https://dzone.com/articles/jdbc-what-resources-you-have. [Accessed 20 December 2020].

[123] European Union, "Data protection and online privacy," European Union, 09 March 2020. [Online]. Available: https://europa.eu/youreurope/citizens/consumers/internet-telecoms/data-protection-online-privacy/index_en.htm. [Accessed 25 December 2020].

[124] A. Beylkin, "Opt in checkboxes & consent for email marketing," Words on Marketing, [Online]. Available: https://www.amandabeylkin.com/marketing-blog/opt-in-checkboxes-consent-email-marketing/. [Accessed 25 December 2020].

[125] R. Degges, "Everything You Ever Wanted to Know About Secure HTML Forms," Twilio, 30 September 2017. [Online]. Available: https://www.twilio.com/blog/2017/09/everything-you-ever-wanted-to-know-about-secure.html-forms.html. [Accessed 18 November 2020].

[126] G. Barré, "How to store a password in a web application?," Meziantou's Blog, 17 June 2019. [Online]. Available: https://www.meziantou.net/how-to-store-a-password-in-a-web-application.htm. [Accessed 19 November 2020].

[127] H. Qureshi, "Hash Functions," Nakamoto, 29 December 2019. [Online]. Available: https://nakamoto.com/hash-functions/. [Accessed 19 November 2020].

[128] OWASP, "Password Storage Cheat Sheet," OWASP, [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html#password-hashing-algorithms. [Accessed 20 November 2020].

[129] N. Provos and D. Mazière, "A Future-Adaptable Password Scheme," in *Proceedings of the FREENIX Track: 1999 USENIX Annual Technical Conference*, Monterey, California, USA, June 6–11, 1999. [Online]. Available: https://www.usenix.org/legacy/events/usenix99/provos/provos.pdf. [Accessed 29 January 2021].

[130] OWASP, "OWASP Top Ten," OWASP, [Online]. Available: https://owasp.org/www-project-top-ten/. [Accessed 18 November 2020].

[131] W3Schools, "SQL Injection," W3Schools, [Online]. Available: https://www.w3schools.com/sql/sql_injection.asp. [Accessed 22 January 2021].

[132] P. Kumar, "JDBC Statement vs PreparedStatement – SQL Injection Example," JournalDev, [Online]. Available: https://www.journaldev.com/2489/jdbc-statement-vs-preparedstatement-sql-injection-example. [Accessed 18 November 2020].

[133] B. Brumm, "How to Escape Single Quotes in SQL," Database Star, 01 May 2017. [Online]. Available: https://www.databasestar.com/sql-escape-single-quote/. [Accessed 18 November 2020].

[134] Cloudflare, "What Is HTTPS?," Cloudflare, [Online]. Available: https://www.cloudflare.com/learning/ssl/what-is-https/. [Accessed 19 November 2020].

[135] Apache Software Foundation, "SSL/TLS Configuration How-To," Apache Software Foundation, 03 December 2020. [Online]. Available: https://tomcat.apache.org/tomcat-10.0-doc/ssl-howto.html. [Accessed 25 January 2021].

[136] Guru99, "Difference between Cookie and Session," Guru99, [Online]. Available: https://www.guru99.com/difference-between-cookie-session.html. [Accessed 02 January 2021].

[137] Pankaj, "Session Management in Java – HttpServlet, Cookies, URL Rewriting," JournalDev, [Online]. Available: https://www.journaldev.com/1907/java-session-management-servlet-httpsession-url-rewriting. [Accessed 20 December 2020].

[138] JavaTPoint, "https://www.javatpoint.com/http-session-in-session-tracking," JavaTPoint, [Online]. Available: https://www.javatpoint.com/http-session-in-session-tracking. [Accessed 20 December 2020].

[139] N. H. Minh, "How to configure session timeout in Tomcat," CodeJava, 06 August 2019. [Online]. Available: https://www.codejava.net/servers/tomcat/how-to-configure-session-timeout-in-tomcat. [Accessed 20 December 2020].

[140] S. Kamani, "Web security essentials - Sessions and cookies," { Soham Kamani }, 08 January 2017. [Online]. Available: https://www.sohamkamani.com/blog/2017/01/08/web-security-session-cookies/. [Accessed 03 January 2021].

[141] W3Schools, "HTML <img> src Attribute," W3Schools, [Online]. Available: https://www.w3schools.com/tags/att_img_src.asp. [Accessed 21 December 2020].

[142] C. Broadley, "Form Enctype HTML Code: Here's How It Specifies Form Encoding Type," HTML.com, [Online]. Available: https://html.com/attributes/form-enctype/. [Accessed 23 December 2020].

[143] Oracle, " Creating and Configuring JSPs," Oracle, [Online]. Available: https://docs.oracle.com/cd/E13222_01/wls/docs92/webapp/configurejsp.html. [Accessed 24 December 2020].

[144] Guru99, "JSP File Upload & File Download Program Examples," Guru99, [Online]. Available: https://www.guru99.com/jsp-file-upload-download.html. [Accessed 24 December 2020].

[145] Apache Software Foundation, "Annotation Type MultipartConfig," Apache Software Foundation, [Online]. Available: https://tomcat.apache.org/tomcat-10.0-doc/servletapi/jakarta/servlet/annotation/MultipartConfig.html. [Accessed 24 December 2020].

[146] Eclipse Foundation, "Uploading Files with Jakarta Servlet Technology," Eclipse Foundation, [Online]. Available: https://eclipse-ee4j.github.io/jakartaee-tutorial/servlets011.html. [Accessed 24 December 2020].

[147] N. H. Minh, "Java File Upload Example with Servlet 3.0 API," CodeJava, 27 June 2019. [Online]. Available: https://www.codejava.net/java-ee/servlet/java-file-upload-example-with-servlet-30-api. [Accessed 24 December 2020].

[148] L. Hubmaier, *Tomcat Web Server: CGI vs. Servlet,* Vienna, Austria: Vienna University of Economics and Business, 2017. [Online]. Available: http://wi.wu.ac.at:8002/rgf/diplomarbeiten/Seminararbeiten/2017/20171221_Hubmaier_TomcatWithRexx.pdf. [Accessed 29 January 2021].

[149] Apache Software Foundation, "CGI How To," Apache Software Foundation, 03 December 2020. [Online]. Available: https://tomcat.apache.org/tomcat-10.0-doc/cgi-howto.html. [Accessed 06 January 2021].

[150] Eclipse Foundation, "Jakarta Mail FAQ," Eclipse Foundation, [Online]. Available: https://eclipse-ee4j.github.io/mail/FAQ#1. [Accessed 24 December 2020].

[151] The Eclipse Foundation, "Jakarta Activation," The Eclipse Foundation, [Online]. Available: https://eclipse-ee4j.github.io/jaf/. [Accessed 24 December 2020].

[152] S. Kandula, "Example on getParameterValues() method of Servlet Request," Java4s, 28 January 2013. [Online]. Available: https://www.java4s.com/java-servlet-tutorials/example-on-getparametervalues-method-of-servlet-request/. [Accessed 24 December 2020].

[153] GeeksforGeeks, "Properties Class in Java," GeeksforGeeks, 24 November 2020. [Online]. Available: https://www.geeksforgeeks.org/java-util-properties-class-java/. [Accessed 24 December 2020].

[154] Tutorials Point, "JavaMail API - Core Classes," Tutorials Point, [Online]. Available: https://www.tutorialspoint.com/javamail_api/javamail_api_core_classes.htm. [Accessed 24 December 2020].

[155] Eclipse Foundation, "Uses of Class jakarta.mail.Message.RecipientType," Eclipse Foundation, [Online]. Available: https://jakarta.ee/specifications/mail/2.0/apidocs/jakarta/mail/class-use/message.recipienttype. [Accessed 24 December 2020].

[156] G. Mayer, *Scripting the ODF Toolkit (Proof of Concept),* Vienna, Austria: Vienna University of Economics and Business, 2012. [Online]. Available: http://wi.wu.ac.at:8002/rgf/diplomarbeiten/2012_Mayer/201211_Mayer_Scripting_ODF.pdf . [Accessed 29 January 2021].

[157] P. Malek, "Everything You Need to Know About SMTP Security," Railsware Products, 14 August 2019. [Online]. Available: https://blog.mailtrap.io/smtp-security/#Whats_SMTP_Is_it_secure. [Accessed 27 January 2021].

[158] R. Kumar, "How to Create VirtualHost in Tomcat 9/8/7," TecAdmin, [Online]. Available: https://tecadmin.net/create-virtualhost-in-tomcat/. [Accessed 12 September 2020].

[159] Apache Software Foundation, "The Server Component," Apache Software Foundation, [Online]. Available: https://tomcat.apache.org/tomcat-10.0-doc/config/server.html. [Accessed 10 December 2020].

[160] PostgreSQL Global Development Group, "23.1. Locale Support," PostgreSQL Global Development Group, [Online]. Available: https://www.postgresql.org/docs/current/locale.html. [Accessed 09 December 2020].

[161] PostgreSQL Global Development Group, "9.17. Sequence Manipulation Functions," PostgreSQL Global Development Group, [Online]. Available: https://www.postgresql.org/docs/current/functions-sequence.html. [Accessed 18 December 2020].

[162] S. Weiss, "error handling: permission denied for sequence _id_seq…," /* Code Comments */, 20 November 2018. [Online]. Available: https://stephencharlesweiss.com/20181120-error-handling-permission-denied-for-sequence-_id_seq/. [Accessed 18 December 2020].

Images Used

All images used, to create the web applications of this paper, originate from Pixabay. They are licensed under the Pixabay License, which allows them to be used for free for commercial and noncommercial use: `https://pixabay.com/service/license/`

Background: R. Balog, "Landscape Nature Forest Fog Misty Pine," Pixabay, 26 September 2015. [Online]. Available: https://pixabay.com/photos/landscape-nature-forest-fog-misty-975091/. [Accessed 20 January 2021].

Oak: K. Craft, "Tree Oak Landscape View Field Scenic Countryside," Pixabay, 14 July 2012. [Online]. Available: https://pixabay.com/photos/tree-oak-landscape-view-field-402953/. [Accessed 20 January 2021].

Birch: A. Стафичук, "Summer Landscape Background Dawn Fog Beautiful," Pixabay, 09 July 2017. [Online]. Available: https://pixabay.com/photos/summer-landscape-background-dawn-2913409/. [Accessed 20 January 2021].

Willow: M. Amber, "Weeping Willow Pond Water Swan Reflection Summer," Pixabay, 11 July 2019. [Online]. Available: https://pixabay.com/photos/weeping-willow-pond-water-swan-4334489/. [Accessed 20 January 2021].

Beech: Couleur, "Tree Beech Deciduous Tree Old Tree Gnarled Leaves," Pixabay, 19 October 2017. [Online]. Available: https://pixabay.com/photos/tree-beech-deciduous-tree-old-tree-3601155/. [Accessed 20 January 2021].

Pine: M. Szabolcs, "Pine Forest Pine Trees Forest Pine Trees Nature," Pixabay, 20 August 2020. [Online]. Available: https://pixabay.com/photos/pine-forest-pine-trees-forest-pine-5572944/. [Accessed 20 January 2021].

Maple: Free-Photos, "Maple Autumn Season Fall Foliage Sunset Scene," Pixabay, 09 November 2015. [Online]. Available: https://pixabay.com/photos/maple-autumn-season-fall-foliage-984420/. [Accessed 20 January 2021].