FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Evaluation of State-of-the-Art resource handling in JavaServer Faces based web applications and development of an enhanced resource handler for JSF 2.

BACHELORARBEIT

von

Jakob Korherr
Matrikelnummer 0925036

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Thomas Grechenig
Mitwirkung: Marcus Büttner, Bakk.techn.
Monika Suppersberger, BSc

Wien, 05.06.2014

Erklärung zur Verfassung der Arbeit

Jakob Korherr
Schwarzspanierstraße 18/5, 1090 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

Acknowledgements

First of all, I want to thank Marcus Büttner for his valuable input throughout the development of the relative-resource-handler, and for supporting me on writing this thesis. Marcus was one of the very first people, who actually made use of the relative-resource-handler, even though the work on the project had just started and it was not in a stable state. But this did not keep Marcus from using it, and after a lot of changes and improvements, he was able to adopt the relative-resource-handler in many of his projects.

I would also like to thank some special guys from the Apache MyFaces open source community, Gerhard Petracek, Mark Struberg, and Hanspeter Dünnenberger, for their help on the relative-resource-handler project. Each one of them contributed lots of brilliant ideas to this project.

Of course, I also want to thank my girlfriend, Stefanie, who very much supported me on writing this thesis, and who encouraged me many times to carry on writing in order to be able to complete it.

A big thanks also goes out to my family and friends. Thanks for being there for me!

Finally, I want to thank everyone who helped to complete this thesis and the respective open source project.

Abstract

This bachelor thesis describes different approaches for handling resources (i.e. stylesheets, JavaScript files, images) in JavaServer Faces (JSF) 2 based web applications. At first, general problems of resource management in web applications are explained. Afterwards, simple approaches for handling resources in JSF 1.1 and JSF 1.2 based web applications are described. After that, a non-standard framework for handling resources in Java based web applications is explored. Finally, the resource handling mechanism introduced in the JSF 2 specification is explained in detail, including all its flaws and shortcomings, as well as some configuration options which improve the behaviour of the standard resource handler. Using the findings of the previous sections, an enhanced resource handler for JSF 2, called the *relative-resource-handler*, is developed. This resource handler eliminates many of the previously described (design) flaws of the standard resource handler of JSF 2, while still satisfying the JSF 2 specification, and thus being standard compliant. In addition, this resource handler is able to deal with all mentioned problems of resource management, utilizing many ideas from previously described non-standard technologies. At the end, the findings of this thesis are summarized and an outlook on future developments in this area is given.

Contents

CHAPTER 1

Introduction

1.1 Resources

Resources in the context of this thesis are files, which are needed to correctly serve and display a web application, such as cascading stylesheets (CSS), JavaScript files (JS) or any kind of image files (PNG, JPEG, GIF, and others). This paper deals with the problems that emerge in the context of resource handling in web applications, particularly in JavaServer Faces (JSF) based web applications.

1.2 Problems and requirements in resource management

In the last decade there was a big shift in software development, away from fat clients and towards browser based solutions [34]. However, clients do not want to go without the comfort a fat client application brings along, such as easy-to-use components or fast answer times. Thus modern web applications tend to become very big and complex in order to fulfill the same kind of demands [13]. Besides the logic needed for correct communication between client and server, state of the art web applications also need dozens of resources in order to function properly. The HTTP Archive states that an average web application is currently composed of more than 90 resources [13] [14]. In [22], the authors explain this new kind of web applications, which is referred to as Web 2.0 applications. These applications provide a *rich, responsive user interface* [22], and make use of a technology stack called AJAX. While AJAX allows to improve the interactivity, speed, and usability of a web page, it also requires a lot of resources, mostly XHTML or HTML, cascading stylesheets (CSS), JavaScript, and XML. [23] explains how AJAX can be used to build rich web applications.

Different problems emerge when dealing with resources in this context, which can roughly be divided into the following categories: Organization, Delivery, Caching and update mechanism, and Dependency management. These problems directly lead to the respective requirements for a state of the art resource handling framework for Web 2.0 applications.

Organization

First of all, resources need to be organized in a file system (on a web server), in order to be available to the application. A lot of web applications store their resources directly in subfolders of the main application. Most commonly those are named `css` or `style` for cascading stylesheets, `js` for JavaScript files, or `img` for images. This approach works fine for small web applications, but bigger ones need other solutions such as separate servers for serving resources or even content delivery networks (CDN), because of the lack of scalability of this simple method [20] [30]. In addition, some applications might serve a lot of images or videos, which can cause masses of traffic and definitely needs special treatment. In [20] the authors discuss four approaches to content delivery of modern web applications, including different options for CDNs. Other than that, web applications often use different libraries, which themselves may use some resources[1], and those need to be handled too, of course. A resource handling framework should therefore provide a simple way to organize resources, and it should support placing resources directly into the web application, storing them anywhere in the classpath, or even having them placed somewhere in a CDN.

Delivery

As already mentioned in the previous section, resources can be stored on the same web server the application is served from, which is mostly the case for small applications. However, they can also be made available through a different server or even through an own content delivery network that consists of multiple servers. Apart from that, resources can also be compressed before being sent to the client, e.g. using GZIP. As outlined in [30], using GZIP to compress resources can reduce data sizes by 70%, resulting in an improvement of response times, especially for users with slow network connections. In addition, there are blocking and non-blocking web servers available, which will behave very differently when serving a lot of resources concurrently and under high server load. These various delivery options should all be supported by a state of the art resource handling framework.

Caching and update mechanism

Because of the fact that resources normally do not change very often after a web application has been rolled out (at least not from one request to the next one in the same session), it is a good idea to cache resources at certain levels. Those levels of caching include memory caches on the server, caches in proxy servers of clients, and browser caches. Using caches allows a browser to load a resource only once and then to use it for multiple requests or even for several sessions of the same web application. Caching of resources at the client is normally accomplished by setting the Expires HTTP response header, as presented in [30]. Even though this caching mechanism can drastically improve performance and lower server loads, it can become very hard to update a certain resource to a new version (e.g. when a part of the web application is changed), if the respective Expires header has been set to a future date. Hence, a resource handling framework

[1] In the case of Java these are often shipped within JAR files that are located on the web application's classpath.

must provide a mechansim to deal with this kind of scenario, i.e. it must be possible to invalidate outdated cache copies of resources efficiently.

Dependency management

Many modern web application frameworks use a component based approach for building the user interface [29]. This means that the developer does not directly use HTML tags, but rather higher level components that will render to (a collection of) HTML tags. Those components can require certain resources in order to be rendered properly. Thus, a mechanism is needed, which links components to their corresponding resources and which automatically includes the respective resources for all the components on the current page.

1.3 JavaServer Faces (JSF)

JavaServer Faces (JSF) is a web application framework standard in the Java Community Process (JCP). It follows the model-view-controller design pattern and uses a component based approach for building the user interface [29]. The JSF standard is available in the versions 1.1 [1], 1.2 [2], 2.0 and 2.1 [3], each of which is implemented by the Apache MyFaces project[2] and the Oracle Mojarra project[3]. In addition, the JSF 2.2 [4] standard is currently under development. In JSF 1.x there was no special support for handling resources, but as of JSF 2.0 the concept of the resource handler was introduced, which will be presented in detail in chapter 2.

This thesis specifically deals with resource handling in JSF 2 based web applications, because JSF 2 was heavily used when building the information system of the Vienna University of Technology (TISS)[4].

1.4 Aim of this work

The aim of this thesis is to analyze current solutions which are available for handling resources in JSF 2 based web applications in order to include the findings in the development of the information system of the Vienna University of Technology (TISS). More precisely, the resource handling mechanism of JSF 2 as well as third party solutions should be studied, the problems of the various approaches should be pointed out, and based on this outcomes, a solution should be worked out, which fulfills all requirements that are needed for the development of TISS.

Moreover, any findings acquired in this thesis will be made available to the JSR-344 [6] expert group[5], in order to improve the JavaServer Faces standard.

[2] Available at http://myfaces.apache.org

[3] Available at http://javaserverfaces.java.net

[4] Available at https://tiss.tuwien.ac.at

[5] The Java Specification Request (JSR) 344 was opened to create the JavaServer Faces 2.2 standard within the Java Community Process (JCP).

1.5 Structure of this thesis

This thesis is structured as follows: In chapter 2 existing solutions for handling resources in JSF will be presented. Those include mechanisms of the JSF standard as well as approaches from third parties. Based on the acquired information, chapter 3 will then develop a solution which deals with the problems of the existing solutions and which adds new functionality that is required for TISS. Finally, chapter 4 will summarize the findings of this thesis and give an outlook on future work in this area.

CHAPTER 2

Resource handling in JSF

2.1 JSF 1.1 and 1.2

The JavaServer Faces specification in the versions 1.1 and 1.2 does not include any mechanism for handling resources and does not even give hints on how to do it [1] [2]. However, because of the fact that those JSF versions are using JavaServer Pages (JSPs) as the view declaration language[1], developers can fall back to established methods that are used when developing JSP based web applications. Unfortunately, JavaServer Pages also does not provide an own mechanism for handling resources, but it allows you to simply use the standard HTML tags used for resource inclusion, namely the `link` and `script` tags. A typical head section of a JSP file is shown in listing 2.1.

Listing 2.1: Head section of a JSP file including a stylesheet and a JavaScript file.

```
<html>
  <head>
    <title>Example JSP file</title>
    <link rel="stylesheet" type="text/css"
          href="css/style.css" />
    <script language="javascript" type="text/javascript"
            src="js/somescript.js" />
  </head>
  <body>...</body>
</html>
```

The code in listing 2.1 includes the cascading stylesheet file `style.css` located in the subfolder `css`, and the JavaScript file `somescript.js` located in the subfolder `js`. This solution works, because every servlet container running the JSP (e.g. Apache Tomcat or Codehaus Jetty)

[1] Facelets is available as a separate, non-standard library for JSF 1.1 and 1.2. As of JSF 2.0, however, Facelets became part of the standard.

knows how to serve static resources, like HTML, CSS or JavaScript files, which are located in the public section of the web application archive. However, the servlet container will not add any special HTTP cache headers (e.g. the Expires HTTP response header as discussed in [30]) per default and it can happen that the resources will never be cached and have to be served again and again for every request. Also, the paths to the resource files are static in the JSP, which can be a problem when using any kind of templating mechanism that accesses the main template (containing the resource references) from a different folder of the web application. This solution also does not offer any kind of resource dependency management, as mentioned in the introduction. Furthermore, because of the fact that resources must reside in the public area of a web application archive in order to be served by the servlet container, they cannot be put into JAR files that are located on the web application's classpath, which might be the case when using component libraries.

This simple approach for handling resources hardly fulfils any of the requirements stated in section 1.2, and is therefore very limited. As already pointed out, this approach does not allow the resource files to be stored anywhere else than the public area of the web application archive, which is a severe limitation. Furthermore, this approach completely relies on the standard HTTP file transfer implementation of the respective servlet container, and thus the delivery of resources can only be changed marginally. In addition, there is no support for any of the requirements of the categories *caching and update management* and *dependency management* from section 1.2.

A common solution to some of these problems is to render the respective resource paths in the head section of the JSP file in some way[2] and to provide a Servlet that serves the respective resources from the correct folder or even from a JAR file on the classpath. This solution, however, can quickly become very complex and also unhandy for a normal web application developer [19]. Fortunately, there exists a non-standard framework which implements this solution for handling resources. This framework, called Weblets, will be explained in detail in section 2.2.

2.2 Weblets

Weblets is a non-standard resource handling framework for Java based web applications [18] [32]. It was hosted on the java.net platform[3] before Sun Microsystems was bought by Oracle in 2009 [25] and was then moved to Werner Punz's GitHub account [33]. The latest version of Weblets is 1.3. The released artifacts can be found in the Maven central repository[4].

In order to install Weblets on any Java based web application, one has to add the required libraries (`weblets-api` and `weblets-impl`) to the web application classpath. In addition, the configuration shown in listing 2.2 has to be added to the `web.xml` file of the web application.

[2] For example by writing an own JSF component that renders the appropriate HTML tags.

[3] It used to be available via http://weblets.dev.java.net

[4] http://repo2.maven.org/maven2/com/github/weblets/

Listing 2.2: The configuration of Weblets in `web.xml`

```
<listener>
    <listener-class>
        net.java.dev.weblets.WebletsContextListener
    </listener-class>
</listener>
<servlet>
    <servlet-name>Weblets Servlet</servlet-name>
    <servlet-class>
        net.java.dev.weblets.WebletsServlet
    </servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>Weblets Servlet</servlet-name>
    <url-pattern>/weblets/*</url-pattern>
</servlet-mapping>
```

The architecture of Weblets is very much alike the Servlet API of Java EE [5]. In order to serve resources from a specific source, one has to create a Weblet for that source. However, the web application developer does not have to implement the Weblet himself (like e.g. a Servlet), but he can rather use an implementation provided by Weblets, which just needs some configuration in order to function properly[5]. The most important Weblet classes include `PackagedWeblet`, which can be used to load resources from a specific package in the classpath[6], `WebappWeblet`, which can be used to load resources from the web application context, and `URLWeblet`, which allows loading resources from an external location (e.g. a content delivery network). Listing 2.3 shows an example configuration of a Weblet.

Listing 2.3: Configuration of a `PackagedWeblet`

```
<?xml version="1.0" encoding="UTF-8" ?>
<weblets-config xmlns="http://weblets.dev.java.net/config">
    <weblet>
        <weblet-name>weblets.resources</weblet-name>
        <weblet-class>
            net.java.dev.weblets.packaged.PackagedWeblet
        </weblet-class>
        <init-param>
            <param-name>package</param-name>
            <param-value>at.jakobk.resources</param-value>
        </init-param>
    </weblet>
```

[5]Of course, a developer can also build his own implementation of a Weblet, but for most cases, the standard implementations provided by Weblets will be sufficient.

[6]This package can also reside in any JAR file on the classpath. It does not have to be directly in the classes folder of the web application.

```
    <weblet-mapping>
        <weblet-name>weblets.resources</weblet-name>
        <url-pattern>/resources/*</url-pattern>
    </weblet-mapping>
</weblets-config>
```

This configuration creates a `PackagedWeblet`, which will be serving resources from the package `at.jakobk.resources`. It must be stored in a file called `weblets-config.xml` in the `WEB-INF` folder of the web application or in any `META-INF` folder in the classpath in order to be discovered by Weblets. The configuration schema is very similar to the schema of the `web.xml` file of every Java based web application: the `weblet` entry corresponds to the `servlet` entry, and the `weblet-mapping` entry corresponds to the `servlet-mapping` entry. Every Weblet has a unique name, an implementation class, some init parameters, various other configuration options (like e.g. MIME type [12] mapping or weblet version), and at least one mapping[7]. Depending on the chosen implementation of a Weblet, the required init parameters may be different, for example `PackagedWeblet` requires the `package` parameter, whereas `WebappWeblet` requires the `resourceRoot` parameter.

In order to use the above Weblet in a JSP page, Weblets provides the class `WebletUtils`, which can be added as a page bean to the JSP. Listing 2.4 shows the usage of the Weblet configured above in a JSP page. It will include the resource file `style.css` from the subfolder `css` of the package `at.jakobk.resources` (which can reside anywhere on the web application classpath, even in a JAR file) as a cascading stylesheet on the resulting HTML page.

Listing 2.4: Usage of a Weblet in a JSP page

```
<%@ page contentType="text/html;charset=UTF-8"
         language="java" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<jsp:useBean class="net.java.dev.weblets.WebletUtils"
             scope="application" id="wbl"/>
<html>
  <head>
    <title>Weblets JSP test</title>
    <link rel="stylesheet" type="text/css"
href="<%=wbl.getURL("weblets.resources","/css/style.css")%>"/>
  </head>
  <body>
    ...
  </body>
</html>
```

The code in listing 2.4 produces the output shown in listing 2.5 (cut for clarity).

[7] In contrast to Servlets, Weblets only supports path based mappings, because it needs the additional path info only available in this mapping variant in order to identify the correct resource.

Listing 2.5: Output of code in listing 2.4.

```
...
<link rel="stylesheet" type="text/css"
      href="/weblets-demo/weblets/resources/css/style.css" />
...
```

The generated output URL contains the context-path of the web application (`weblets-demo`), the mapping of the Weblets Servlet (`weblets`), the mapping of the Weblet itself (`resources`) and the path to the resource (`css/style.css`). Using this information, the Servlet container can determine the correct Servlet to handle the request (the Weblets Servlet) and Weblets can find the respective Weblet to serve the resource.

Although Weblets was designed to work in any Java based web application, it also provides special support for JSF based web applications. In order to get an URL of a resource from Weblets in a JSF view, one can simply use the preconfigured managed bean `weblet`, like shown in listing 2.6.

Listing 2.6: Using Weblets in a JSF view via the managed bean `weblet`.

```
...
<link rel="stylesheet" type="text/css"
  href="#{weblet.url['weblets.resources']['/css/style.css']}" />
...
```

Furthermore, Weblets also provides a Facelets function when using JSF in combination with Facelets[8]. Listing 2.7 shows the usage of the Facelets function of Weblets.

Listing 2.7: Using Weblets in a JSF view via the provided Facelets function.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:wbl="http://weblets.dev.java.net/facelet-tags">
  <h:head>
    <title>Weblets JSF test</title>
    <link rel="stylesheet" type="text/css"
    href="#{wbl:url('weblets.resources','/css/style.css')}" />
  </h:head>
  <h:body>
    ...
  </h:body>
```

[8]Using Facelets was optional in JSF 1.x, because it was not part of the standard at that time. As of JSF 2.0 Facelets became part of the specification, and is now the per se standard for creating JSF views.

```
</html>
```

Resource versioning support

As mentioned in the introduction, it is a good idea to cache resources on various levels for performance reasons. Those levels include web server caches, proxy servers and also the browser cache on the client system. However, if a new version of the web application is rolled out, some resources will have been changed, and those need to be updated in all caches. Of course, this will happen eventually[9], but that is not an acceptable solution. If anything, it should happen immediatly, or else some clients will encounter display problems after the rollout.

In order to tackle this problem, Weblets provides the `weblet-version` configuration parameter as a child element of the `weblet` element in `weblets-config.xml`. Listing 2.8 shows how to configure this property.

Listing 2.8: A Weblet configuration entry including `weblet-version`.

```
<?xml version="1.0" encoding="UTF-8" ?>
<weblets-config xmlns="http://weblets.dev.java.net/config">
  <weblet>
    <weblet-name>weblets.resources</weblet-name>
    <weblet-class>
        net.java.dev.weblets.packaged.PackagedWeblet
    </weblet-class>
    <weblet-version>1.0</weblet-version>
    ...
  </weblet>
</weblets-config>
```

Setting this property for a specific Weblet in `weblets-config.xml` has the effect that the configured version (including a prefixed dollar sign) will be added to the rendered Weblet context path automatically. Listing 2.9 shows the rendered output of a resource reference from the Weblet used earlier in this section, which was configured with `weblet-version` equal to `1.0`.

Listing 2.9: Output of a Weblet with `weblet-version` set to `1.0`.

```
...
<link rel="stylesheet" type="text/css"
      href="/weblets-demo/weblets/resources$1.0/css/style.css"/>
...
```

As shown in the listing above, the URL of every resource served by the respective Weblet now contains the configured `weblet-version`. Due to the fact that every cache implementation uses exactly this URL to store and retrieve the cached version of a resource, a change in

[9] This can be hours, days, weeks or even months, depending on the different caching policies.

the URL of a resource directly effects all caches, because they will not be able to find a cached version of the resource anymore, and therefore contact the server to retrieve a new copy of the (changed) resource. [16] explains how web caches work in detail.

When using a build tool, like e.g. Apache Maven[10], it is very easy to automatically set the current version of the project as `weblet-version`, simply by enabling resource filtering for the Weblets configuration file. Using this approach, it is guaranteed that for every new version of a web project, all resources will be reloaded when a client visits the newly deployed application, and all outdated resources, which are stored in the various caches between client and server, are not referenced by the application anymore and will eventually be removed from the caches. In addition, if the `weblet-version` ends with `-SNAPSHOT` (which normally means that the project is currently under development), Weblets will behave as if the version parameter would have been omitted. This will disable all internal resource caches of Weblets, which is required in order to make hot deployments[11] of resources work correctly [32].

Resource references

As discussed so far, Weblets provides various ways to include different resources in web pages. However, it is often the case that those resources need to reference or include other resources themselves, like e.g. a CSS file that references an image for a `background-image` declaration as shown in listing 2.10.

Listing 2.10: CSS file declaring a `background-image`.

```
background-image: url(images/bg.jpg);
```

For references to resources, which reside in the same Weblet as the current resource, it is possible to simply use the relative path to that resource inside the Weblet directory as reference, because Weblets includes the path to every resource in the request URL (e.g. `css/style.css` in listing 2.9), and thus the browser can correctly resolve the relative path to the resource. For resources that come from a different Weblet, however, this is not possible. For that case, Weblets provides the function `weblet:resource()`, which can be used in any JavaScript or CSS file. This is possible, because Weblets parses every JavaScript and CSS file when processing it the first time, and at that time, replacing every occurrence of `weblet:resource()` with the respective Weblet path. Listing 2.11 shows the usage of `weblet:resource()` in a `background-image` declaration inside a CSS file, and listing 2.12 shows the resolved version of the same resource.

Listing 2.11: CSS definition using `weblet:resource()`.

```
background-image:
  url('weblet:resource("weblets.resources","/images/bg.jpg")');
```

[10] Available at http://maven.apache.org

[11] A hot deployment updates already deployed resources on a running web application server, without having to restart the server.

Listing 2.12: Resolved CSS definition from listing 2.11.

```
background-image:
  url('/weblets/resources$1.0/images/bg.jpg');
```

Conclusion

Weblets provides a lot of features, which meet many of the requirements of section 1.2. At first, Weblets supports three important places to store and organize resource files via `PackagedWeblet`, `WebappWeblet`, and `URLWeblet` respectively. Second, Weblets provides many configuration options to change the way resources are delivered to the client, e.g. MIME type mapping. In addition, it provides a sophisticated way to deal with cache problems in form of the `weblet-version` configuration parameter. Furthermore, the requirements related to dependency management of resources are addressed by the resource reference support, however, Weblets merely allows resources to reference each other, but it does not directly support web components. In a nutshell, Weblets is a very good choice in order to tackle many of the problems stated in section 1.2.

2.3 JSF 2.0

The specification of JavaServer Faces 2.0 [3] introduced a new, standardized API for handling resources in JSF 2 based web applications. Using this API it is possible to reference resources in JSF views as well as in Java classes. In addition, it is also possible that resources itself reference other resources. The actual resource files can thereby reside on the web application root or on the classpath. Judging by these points one could argue that the resource handling concept of JSF 2 is just the standardized version of Weblets. This is in fact true to some extend, because Weblets is referenced as an influencing technology in the specification. However, there are still certain major differences between the two technologies, which will be demonstrated in this section. Important to note here is that the resource handling concept of JSF 2 is actually a side product of two new features of the JavaServer Faces 2.0 specification, namely Composite Components[12] and the AJAX support[13], because both features themselves rely on accessing and serving resource files. Thus, the expert group was forced to add a resource handling mechanism to the specification, and the resource handling API is the result of that. Fortunately, the API was not only used internally of JSF 2 for the implementation of those two features, but was rather extended in order to also tackle the general problem of handling resources in JSF based web applications.

Resources in JSF views

There are generally two different ways to include resources in a JSF view: Either by using a specific JSF component that is designed to include a certain resource type, or by using the `#{resource['...']}` value expression. Both methods are presented below.

[12] See JSF 2.0 specification, section 3.6 *Composite User Interface Components*.
[13] See JSF 2.0 specification, section 2.5.8 *Ajax*.

For JavaScript files, cascading stylesheets and images, JSF 2 provides own components: `<h:outputScript>`, `<h:outputStylesheet>` and `<h:graphicImage>` respectively. Listing 2.13 shows how to use those components in order to include a JavaScript file, a CSS file and an image on a standard HTML page. The generated HTML output is then shown in listing 2.14.

Listing 2.13: Example of using `<h:outputScript>`, `<h:outputStylesheet>` and `<h:graphicImage>`.

```
<html>
  <head>
    <h:outputScript name="script.js" library="js" />
    <h:outputStylesheet name="style.css" library="css"/>
  </head>
  <body>
    <h:graphicImage name="logo.jpg" library="images" />
  </body>
</html>
```

Listing 2.14: HTML output of listing 2.13.

```
<html>
  <head>
    <script type="text/javascript"
      src="/javax.faces.resource/script.js.xhtml?ln=js">
    </script>
    <link rel="stylesheet" media="screen" type="text/css"
      href="/javax.faces.resource/style.css.xhtml?ln=css" />
  </head>
  <body>
    <img src="/javax.faces.resource/logo.jpg.xhtml?ln=images" />
  </body>
</html>
```

The second approach to include resources in a JSF view is to use the `#{resource['...']}` value expression. This expression will evaluate to the request path of the referenced resource, however, it will not render any HTML markup like the JSF components presented above. Thus, the necessary markup has to be provided additionally. The syntax of the value expression is as follows (in reference to the attributes of the respective JSF components presented above): `#{resource['library:name']}`. Listing 2.15 illustrates the usage of this value expression, while generating the exact same output as listing 2.13 does.

Listing 2.15: Example of using `#{resource['...']}`.

```
<html>
  <head>
```

```
    <script type="text/javascript"
      src="#{resource['js:script.js']}">
    </script>
    <link rel="stylesheet" media="screen" type="text/css"
      href="#{resource['css:style.css']}" />
  </head>
  <body>
    <img src="#{resource['images:logo.jpg']}" />
  </body>
</html>
```

The Java API

The key classes of this API are `javax.faces.application.ResourceHandler` and `javax.faces.application.Resource`, which are shown in listing 2.16 and listing 2.17 respectively. The standard implementations of those classes are delivered by the concrete JSF 2 implementation[14], but can easily be exchanged by custom implementations.

Listing 2.16: The abstract class `javax.faces.application.ResourceHandler` of the JSF 2.0 specification.

```
public abstract class ResourceHandler {

  public static final String RESOURCE_EXCLUDES_PARAM_NAME
    = "javax.faces.RESOURCE_EXCLUDES";

  public static final String RESOURCE_IDENTIFIER
    = "/javax.faces.resource";

  public abstract Resource createResource(String resourceName);

  public abstract Resource createResource(String resourceName,
      String libraryName);

  public abstract Resource createResource(String resourceName,
      String libraryName, String contentType);

  public abstract  boolean libraryExists(String libraryName);

  public abstract String getRendererTypeForResourceName(
      String resourceName);
```

[14] Apache MyFaces or Oracle Mojarra.

```
  public abstract void handleResourceRequest(
      FacesContext context) throws IOException;

  public abstract boolean isResourceRequest(
      FacesContext context);

}
```

Listing 2.17: The abstract class `javax.faces.application.Resource` of the JSF 2.0 specification (some parts cut for clarity).

```
public abstract class Resource {

  public static final String COMPONENT_RESOURCE_KEY
      = "javax.faces.application.Resource.ComponentResource";

  private String _contentType;
  private String _libraryName;
  private String _resourceName;

  public String getContentType() {...}
  public void setContentType(String contentType) {...}

  public String getLibraryName() {...}
  public void setLibraryName(String libraryName) {...}

  public String getResourceName() {...}
  public void setResourceName(String resourceName) {...}

  public abstract InputStream getInputStream()
      throws IOException;

  public abstract URL getURL();

  public abstract String getRequestPath();

  public abstract Map<String, String> getResponseHeaders();

  public abstract boolean userAgentNeedsUpdate(
      FacesContext context);
}
```

The `ResourceHandler` API allows to create a specific `Resource` using three different methods, depending on the available information about the resource (resource name, library

name, and content type). The resource name is the file name of a resource within the context of a resource library. The optional library name is the name of the respective library a resource is located in. Finally, the content type is the MIME type of the resource, as defined in RFC 2046 [12]. If the content type is not given, it will be determined using JSF's `ExternalContext`. Those three properties - resource name, library name und content type - uniquely identify a resource. The `ResourceHandler` API also provides a method to identify the correct renderer type for a specific resource name, e.g. using a different renderer for CSS files and for JavaScript files. Then there is a method that checks whether or not a given library exists, and finally, there are two methods for handling resource requests. The first one is used by the JSF framework to determine if the current request is in fact a resource request, and thus should be treated as such. The latter one is then used for actually handling a resource request, mostly by serving the desired resource.

The `Resource` API provides getters and setters for the aforementioned properties, and provides the `InputStream` or the `URL` of the resource. Moreover, it offers a way to calculate the request path of the resource, in order to be able to compute the complete URL, which will later be used by the client's browser to create a request for this resource. Finally, there are two methods which are used by the framework, if the current request is a resource request. The first one is used to determine the necessary response headers for the response that is sent back to the client, and the second one is used to find out whether the client's user agent needs an update of the resource or not[15].

Handling of a resource request

Just like any normal faces request, a request for a resource also uses the `FacesServlet`. This fact can easily be verified by checking the resource URLs generated by JSF, as e.g. already shown in listing 2.14. In the example, the `FacesServlet` is mapped using a suffix mapping of `*.xhtml`, thus, the resource URLs all end with `.xhtml`. However, if the mapping is changed to a path mapping of `/faces/*`, the resource URLs will change too, now including the path prefix of `/faces/` instead of the suffix `.xhtml`, as shown in listing 2.18.

Listing 2.18: HTML output of listing 2.13 when using a path mapping for the `FacesServlet` instead of a suffix mapping.

```
<html>
  <head>
    <script type="text/javascript"
      src="/faces/javax.faces.resource/script.js?ln=js">
    </script>
    <link rel="stylesheet" media="screen" type="text/css"
      href="/faces/javax.faces.resource/style.css?ln=css" />
  </head>
  <body>
    <img
```

[15] Most likely by looking at the cache headers from the current HTTP request.

```
      src="/faces/javax.faces.resource/logo.jpg?ln=images" />
  </body>
</html>
```

Before initializing the JSF lifecycle, the `FacesServlet` first calls `isResourceRequest()` on the `ResourceHandler` implementation, in order to determine whether the current request is a resource request, or not. This is done by checking for the presence of `javax.faces.resource` in the request path, because every resource request will include this expression in the URL, as e.g. shown in listing 2.14 or listing 2.18. If the current request is in fact a resource request, it will be directly forwarded to the `ResourceHandler` implementation by invocation of `handleResourceRequest()`. This method is then responsible for parsing all request parameters, locating the correct resource, setting all necessary response headers and serving the resource file content back to the client.

For finding out which resource should be served by the current request, the `ResourceHandler` implementation needs to parse the request path and parameters. The request path will include the resource name, either prefixed by the path mapping of the `FacesServlet` (e.g. `/faces/`), or suffixed by the suffix mapping of the `FacesServlet` (e.g. `.xhtml`). In addition, the resource request will include the library name as request parameter `ln`. Using the resource name and library name, the `ResourceHandler` can then create the respective instance of `javax.faces.application.Resource` as described above.

In order for a resource file to be discovered by the standard `ResourceHandler` implementation, it has to be located in one of the following places, while using the respective resource name as file name[16]:

- The `resources` folder in the web application root.
- Any `META-INF/resources` folder in the classpath of the web application.

Optionally, resources can also be organized in libraries, which are the top level folders inside the locations stated above. These libraries are referenced via the library name of a resource. Because of the fact that only the top level folders are used as libraries, a library name can never include path information[17]. Furthermore, there can also be different versions of resource files and libraries, all stored in a special folder structure. However, this file versioning support is poorly backed by the JSF 2 implementations, due to huge performance issues resulting from it [17]. Thus, it will not be described in this paper.

Locale support

The new resource handler API also allows providing resources in localized versions, i.e. serving a different stylesheet for German speaking users as for English speaking users. Those different versions of resources must therefore be stored in locale specific subfolders inside the

[16] Additionally, the resource name can also contain path information, which corresponds to subfolders inside the `resources` directory.

[17] This observation will become very important in chapter 3.

resources folder(s). The name of the subfolder for a specific locale is determined by looking for the value of the key javax.faces.resource.localePrefix in the localized message-bundle of the JSF application. Assuming that one uses eng for English, and ger for German as values of the aforementioned key in the respective message-bundle files, the directory tree for having a localized CSS file called style.css in the library css must look like the following:

```
resources
├── eng
│   └── css
│       └── style.css
└── ger
    └── css
        └── style.css
```

Using this configuration, a client with a locale setting in the browser of de will get the German message-bundle of the JSF application, and thus the value ger for the key javax.faces.resource.localePrefix. Hence the resource handler will use the file ger/css/style.css when serving the resource with resource name style.css and library name css to this client.

Relocation of resource components

In contrast to Weblets or any other method of handling resources presented above, JSF's resource handling mechanism allows to relocate certain resource components[18] in the JSF view before rendering the response. This means that the respective resource component is not rendered at the same place it has been defined in the JSF view, but rather in one of the following places, depending on the value of the attribute target:

- head: The <head> of the HTML response.
- body: The <body> of the HTML resposne.
- form: The first <form> element of the HTML response.

This allows JSF components to define the resources, which are needed to properly display them in the browser, lazily and without any interception of the developer, while still being able to render them at the correct place as defined in the HTML specification[19].

Annotation based resource inclusion

A totally new feature of the resource handling mechanism in JSF 2 is the annotation based resource inclusion support. For that, a new annotation was added to the JSF 2 API: javax.faces.application.ResourceDependency. Listing 2.19 shows the API definition of this annotation.

[18] By default <h:outputScript> and <h:outputStylesheet>.

[19] The HTML specification e.g. states that all stylesheet information must be placed in the head section of the HTML page.

Listing 2.19: The annotation `javax.faces.application.ResourceDependency`.

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE})
@Inherited
public @interface ResourceDependency {
    java.lang.String library() default "";

    java.lang.String name();

    java.lang.String target() default "";
}
```

As the API definition already indicates, using the `@ResourceDependency` annotation on a class allows to define a `Resource`, which is needed to properly display the respective class. Moreover, using `@ResourceDependencies` allows to define more than one `@ResourceDependency` annotations on the same class. However, the said class must be one of the following JSF artifacts:

- JSF component
- Renderer of a JSF component
- Converter
- Validator

Whenever one of the above artifacts is used in a JSF view, its `@ResourceDependency` annotations are scanned, and the corresponding resources are automatically added to the component tree.

Value expression support and resource references

JSF's resource handling mechanism allows the usage of value expressions in certain types of resource files, by default all of type `text/css`. This means that certain resource file content can be dynamically generated using value expressions. More importantly, this enables dynamic resource references by using the `#{resource['...']}` value expression, just as using `weblet:resource()` in a resource managed by Weblets. Listing 2.20 shows the usage of value expressions in a `background-image` declaration inside a CSS file, and listing 2.21 shows the resolved version of the same resource.

Listing 2.20: CSS definition using `#{resource['...']}`.

```
background-image:
  url("#{resource['images:bg.jpg']}");
```

Listing 2.21: Resolved CSS definition from listing 2.20.

```
background-image:
  url("/javax.faces.resource/bg.jpg.xhtml?ln=images");
```

However, this feature must be used very carefully, because it may indicate that the evaluated values of the value expressions are allowed to change from one request to another, but unfortunately, this is not the case. Technically it would be possible, but due to the various resource caching mechanisms implemented in browsers and proxy servers, it just will not work properly in practice.

Conclusion

As already reasoned above, the standard resource handling mechanism of JSF 2 is very much alike Weblets, thus we could argue that it also addresses the problems of section 1.2 like Weblets does, but unfortunately, this is not the case. While Weblets allows to serve resources also from an external location (besides supporting the web application archive and the classpath), the standard resource handler of JSF 2 lacks support of this feature, which is very important when it comes to supporting content delivery networks. Also, JSF's resource handler does not allow to configure the delivery of a resource in any way, and there is also no feature for handling cache issues on version updates, i.e. there is nothing equivalent to the `weblet-version` configuration parameter. However, the standard resource handler is integrated very tightly into JSF, and therefore makes massive use of the component based approach of JSF. Thus, the resource handler provides many features related to the category *dependency management* of section 1.2, e.g. the annotation based resource inclusion. Unfortunately, in contrast to any approach presented before, the standard resource handler of JSF 2 does not support relative URLs between resources that are equivalent to the relative paths on the file system. Although this sounds trivial, the lack of this feature has major implications, which are explained in the following chapter. Thus, compared to Weblets, the standard resource handler of JSF 2 handles resource dependency management very well, however, in any other category of section 1.2, Weblets is superior to the standard resource handler of JSF 2. This fact served as motivation for the development of the relative-resource-handler, which will also be presented in the following chapter.

CHAPTER 3

An enhanced resource handler for JSF 2

3.1 Problems with existing approaches

The technologies presented in chapter 2 have some flaws, which will be discussed in detail in this section. Many of these flaws originate in the misuse of established web technologies as specified in various RFCs, most importantly RFC 3986 [7], RFC 1808 [8], and RFC 2616 [9].

Resource URL format

The biggest problem of many existing solutions results from the way of how the URL for retrieving a resource is constructed. Commonly used solutions on the world wide web use the HTTP protocol just as it has been designed in RFC 2616 [9]: to retrieve a resource at some particular path on a specific server. For example, listing 3.1 shows the URL for the logo of Google as of October 21st, 2013.

Listing 3.1: The URL of the logo of Google.

```
http://www.google.at/images/srpr/logo11w.png
```

This URL means that the browser needs to retrieve the file `logo11w.png` at the path `images/srpr` on the host `www.google.at`. Thus, the URL somehow reflects the file structure on the server, because it includes the relative path to the image from the web root directory, and nothing more. Now, assume there is a CSS file stored at `css/style.css` on the same server, and thus it is accessible via the URL shown in listing 3.2.

Listing 3.2: The URL of the file `css/style.css`.

```
http://www.google.at/css/style.css
```

If this CSS file uses the aforementioned image in a `background-image` declaration, it just needs to include the relative path to the image on the file system, which will be equal to the resulting relative path in the URL. Listing 3.3 shows the file `css/style.css`.

Listing 3.3: CSS file referencing the image from listing 3.1 with a relative path.

```
body {
    background-image: url('../images/srpr/logo11w.png');
}
```

This example shows how easy it is to let resources reference each other in a static way and without framework interception, if the underlying technologies are used as designed. However, if JSF's resource handling mechanism is used, the URLs for retrieving the two resources are very different, as shown in listing 3.4 and 3.5 respectively.

Listing 3.4: The URL of the logo of Google, if the JSF resource handling mechanism would be used (assuming a suffix mapping of `*.xhtml`).

```
http://www.google.at/
    javax.faces.resource/srpr/logo11w.png.xhtml?ln=images
```

Listing 3.5: The URL of the file `css/style.css`, if the JSF resource handling mechanism would be used (assuming a suffix mapping of `*.xhtml`).

```
http://www.google.at/javax.faces.resource/style.css.xhtml?ln=css
```

These URLs do not reflect the file system paths on the server, which makes static relative references between them very hard. In fact, the relative path in this example for the CSS file to include the image would be `srpr/logo11w.png.xhtml?ln=images`, which is totally different from the real relative path on the file system as indicated by listing 3.3. However, it is also possible to use a value expression to resolve the path to the image at runtime, as described in the previous chapter. Unfortunately, using a value expression means that the static resource becomes a dynamic resource which needs processing from the JSF framework at runtime, resulting in poor response times when serving the resource. In any case, all files produced by a web designer while creating the design of a web application have to be post-processed by the JSF developer in order for them to work correctly in the JSF based web application. Of course, this also applies to any third-party libraries used in the design, e.g. commonly used solutions like YUI[1] or Twitter Bootstrap[2]. Moreover, standard developer tools will not recognize the relative paths required by JSF, and thus produce many warnings while development.
To circumvent this problem, the ICEfaces development team created a tool called `cssurlmapper` [21] for an automatic conversion from normal CSS files, which use relative paths equal to the paths on the file system, to post-processed CSS files, which include value expressions as resource references for the JSF 2 resource handler. The presence of such a tool indicates that many people face this problem when creating JSF based web applications, and that the resource URL format used by JSF has a serious design flaw.

[1] http://yuilibrary.com/
[2] http://getbootstrap.com/

Another negative aspect of JSF's standard resource URI design is that it contains many implementation specific properties, which are subject to change if the underlying framework is updated. In addition, the URIs may contain wrong suffixes (e.g. ending in `.xhtml` whereas the requested resource is actually a cascading stylesheet), and they use query string information to transfer resource identifiers. In 1999, Tim Berners-Lee posted an article called "Cool URIs don't change" [31], in which he reasons how good URIs should be designed, and what makes URIs bad. Following this article, JSF's resource URI design is certainly a negative example of how URIs should be designed. Further articles also discuss good URI design, and they even introduce the term *Clean URIs* describing URIs, which are purely structural, which do not contain a query string, and which instead contain only the path of the resource [15] [24] [26].

Dynamic resource content

As already mentioned in the previous section, resources containing dynamic content (i.e. value expressions, or Weblets functions) require processing from the JSF framework prior to be served to a client, and this results in increased response times when serving these resources. Static resources, however, can be directly served to the client, which can improve response times dramatically. Additionally, static resources can be cached at various levels, including a server side cache, proxy servers, and even the browser cache at the client [16]. Moreover, this caching behaviour is very well established in the world wide web, which actually makes it very hard to really be able to serve dynamic content in resource files. Here, JSF actually fools the developers into believing that true dynamic resource content can be achieved by using the resource handler of JSF 2, however, this is not the case in practice. Thus, eliminating the need for dynamic resource content is very important.

Resource update management

As stated in the introduction of this paper, a common problem related to resources of web applications is the update management. Due to the fact that there are various levels of caches between a web application server and the client's browser, updating the content of an established resource can be very hard, without changing the URL of the respective resource. As described in section 2.2, Weblets provides a way to deal with this problem: the `weblet-version`, which is automatically added to every resource URL.
Another web application framework, the Google Web Toolkit (GWT), also implements a subtle way to deal with this kind of problem: the GWT compiler creates unique filenames for all resources of the web application (by using a hashed value of the respective file content) [27]. This way the generated resources of a GWT based web application can be cached indefinitely, because updated resources will be named differently, resulting in a new resource URL.
Unfortunately, JSF's standard resource handling mechanism lacks support of this feature, which means that the developers have to come up with own ways on how to deal with this problem. One possible approach is to manually rename the used resource libraries in order to get a different resource URL for all updated resources. However, this can be a lot of work and can lead to unwanted side effects.

Locale support cache problems

Section 2.3 describes the locale support of JSF's standard resource handler. With this feature it is possible to serve different versions of resources to users in different locales. The feature works well, if the same user always stays at the same locale, however, if the locale is ever changed, the feature won't work as expected. The problem here, again, origins in the cache hierarchy between the server and the client. As described in the previous subsection, the resource URL needs to change in order to ensure that the client really gets the new version of a resource. Otherwise, the chance is very high that the client will get a cached (old) version of the resource. This means that the resource URL must include the current locale in order to get rid of these cache problems, which is not the case for resource URLs created by JSF's standard resource handler.

3.2 Tweaking the standard resource handler

There are some obvious approaches to improve the resource handling mechanism while still utilizing JSF's standard resource handler, like e.g. avoiding dynamic resource content, or not using the provided locale support. Apart from that, the JSF 2 specification provides some ways to control the functionality of the standard resource handler, which can be used in a clever way in order to circumvent some of the major problems.

Path mapping for the `FacesServlet`

The most important approach to influence the behaviour of the resource handler is to use a path mapping for the `FacesServlet` (instead of suffix mapping). This way the resource URLs do not have to end with the suffix mapping expression, which has been configured for the `FacesServlet` (most likely `*.jsf` or `*.xhtml`). The resulting URLs will include the path mapping (e.g. `/faces/`) in the very beginning of the resource path, even before the marker for a resource request (`javax.faces.resource`). Listing 3.6 shows the resource URL from listing 3.5, if a path mapping is used.

Listing 3.6: The URL of the file `css/style.css`, if the JSF resource handling mechanism would be used (assuming a path mapping of `/faces/*`).

```
http://www.google.at/faces/javax.faces.resource/style.css?ln=css
```

Using this approach it is guaranteed that the resource URL will end with the correct extension of the resource, e.g. `*.css`, `*.jpg`, or `*.js`.

Avoid library names

Using a path mapping as described above is an important step towards a reasonable resource handling mechanism, nevertheless, the resource URL in listing 3.6 still does not reflect the relative path on the server, and furthermore it still contains URL parameters. To get rid of this problem, the usage of JSF's resource library concept must be abandoned. This means that instead of using a library name of `css` and a resource name of `style.css` to reference the resource `css/style.css`, a resource name of `css/style.css` and no library name must

be used. Note that this approach only works if no library name specific features of the JSF resource handler (like e.g. the locale support) are used. To illustrate this idea, listing 3.7 shows the two ways of including the resource `css/style.css`, and listing 3.8 show the resource URL generated by JSF when using no library name.

Listing 3.7: Two ways of including the resource `css/style.css`.

```
<h:outputStylesheet name="style.css" library="css" />
<h:outputStylesheet name="css/style.css" />
```

Listing 3.8: The URL of the file `css/style.css` using a path mapping of `/faces/*` and no library name.

```
http://www.google.at/faces/javax.faces.resource/css/style.css
```

Finally, the resource URL now reflects the relative path on the server, which means that a static relative reference to another resource (as e.g. shown in listing 3.3) will now work properly. Moreover, the resource URL can now be considered to be a *Clean URI* as specified in section 3.1.

3.3 An enhanced resource handler

Applying the tweaks described in section 3.2 is sufficient to cope with the problems of the standard JSF resource handler that arise in many web applications. However, those tweaks come along with a lot of consequences:

- A path mapping has to be used for the `FacesServlet`.
- No library names can be used.
- The locale support cannot be used.

In addition, many shortcomings of the standard resource handler are not covered by the tweaks:

- No resource update management.
- No support of content delivery networks.
- No compression support (e.g. GZIP).
- Insufficient support of value expressions.

In order to deal with all those problems, an enhanced resource handler for JSF, namely the relative-resource-handler, was created. This custom resource handler will be described in the following.

The relative-resource-handler project

The class `RelativeResourceHandler` serves as an extension to the standard resource handler of the respective JSF implementation. It extends `javax.faces.application.ResourceHandlerWrapper` and therefore makes use of the wrapper support of JSF. The project is packaged as JAR file, which includes a `faces-config.xml` file that registers the relative-resource-handler as resource handler for JSF. This configuration file will be automatically discovered by any JSF 2 implementation, if the JAR file is in the classpath of the web application. Hence, it is sufficient to add the JAR file to the web application in order to make use of this enhanced resource handler.
The open source project is available at `http://code.google.com/a/apache-extras.org/p/relative-resource-handler/`. The code is licensed under the Apache 2.0 license[3].

Design principles

The most important design principle is that resource URLs must be built in a way, which reflects the path structure of the resource files of the web application, and all necessary information must be directly encoded in the resource URL path, i.e. instead of transporting it via request parameters. This way it is guaranteed that resources can reference each other using the relative URLs[4], which are equal to the relative paths on the file system as specified in RFC 3986 *Uniform Resource Identifier (URI): Generic Syntax* [7] (and originally specified in RFC 1808 *Relative Uniform Resource Locators* [8]). The resource URLs thus fulfill all requirements of *Clean URIs* as described in section 3.1.
Furthermore, the relative-resource-handler follows the REST[5] paradigm for the resource URLs it creates. The REST paradigm was introduced by Roy T. Fielding in his dissertation and accompanying papers [10] [11]. This means that all criteria used for the unique identification of a resource must be included in the resource URL. In addition, if the content of a resource is changed, it must get a new URL. Consequently, resources retrieved by such resource URLs can be cached indefinitely, because if the resources are changed, or if another version of the same resource should be served (e.g. because the user changed his locale), the resulting resource URL will be different, and thus the client will certainly receive the desired resource, instead of getting an outdated version of the resource from some cache.
As a result of these two important design principles, this kind of resource URLs will be called *Clean REST URLs*.
Another design principle of the relative-resource-handler is activation by configuration. By default, the relative-resource-handler will not handle any resources of the JSF implementation itself or any third party libraries. In order for a resource library to be handled by the relative-resource-handler, a configuration entry for that respective library must be created in the configuration file of the resource handler. This way, it is ensured that the relative-resource-handler does not interfere with any other libraries used by the web application.

[3] http://www.apache.org/licenses/LICENSE-2.0
[4] The relative-resource-handler also got his name from this important design principle.
[5] Representational State Transfer

Finally, the relative-resource-handler follows the design principle of extensibility through service provider interfaces (SPI). Thus, using the standard Java service loader pattern [28], essential classes of the relative-resource-handler project can be exchanged by enhanced versions of these classes.
The realization of these design principles will be described in detail in the following.

A resource URL design implementing Clean REST URLs

Resource URLs created by the relative-resource-handler consist of the following parts:

- Path mapping of the `FacesServlet` (i.e. `faces`).
- Marker for a resource request (`javax.faces.resource`).
- Version of the web application.
- Locale (and country code) of the user, who requests the resource.
- Library name (containing no path information).
- Resource name (possibly containing path information).
- Custom criteria (optional).

Listing 3.9 shows the resource URL generated by the relative-resource-handler of the file `css/style.css` from previous examples.

Listing 3.9: The URL of the file `css/style.css` using the relative-resource-handler.

```
http://www.google.at/
    faces/javax.faces.resource/1.0.0/de_AT/css/style.css
```

As can be seen in listing 3.9, the resource URL really reflects the path structure on the file system, and it also does not include any additional suffix or any additional request parameters, because all information (including the library name) are encoded in the resource path, rather than transporting them via request parameters. Thus, the resource URLs truly allow using relative path references between resources [7] [8].
In order for such resource URLs to work, the relative-resource-handler requires the web application to provide a path mapping for the `FacesServlet`. However, it is not required that the web application entirely uses this path mapping, as it is the case in section 3.2. Anything except resources can actually be served via a suffix mapping, if required. Therefore this requirement is not a limitation for any JSF based web application.
Due to the fact that the resource URL always contains the request locale (and country code) of the current user, the locale support will now work flawlessly, because a locale change of a user will result in a different resource URL, which means that the client will certainly get the correct version of the resource, regardless of any previously cached versions of the same resource. The respective resource, however, does not have to be available in every possible locale. If a resource is not available in the requested locale, the default locale will be used.

The functioning of the resource update management is ensured by the inclusion of the version of the web application in the resource URLs. If a new version of the web application is deployed, all resource URLs will change, and thus all clients will retrieve the most current version of every resource file, regardless of any previously cached content. This approach is very similar to the resource versioning support of Weblets (described in section 2.2).
The resource URLs may also contain custom criteria, which are e.g. needed for multi tenancy support of the web application. This feature can be achieved using the extensibility of the relative-resource-handler, which will be described later.
This resource URL design clearly implements *Clean REST URLs* as discussed above.

Configuration

As outlined above, the relative-resource-handler needs to be configured, in order to start handling resource libraries. The configuration must be stored in the file `/META-INF/relative-resources.xml` in the classpath of the web application. Listing 3.10 shows the configuration file from the example project of the relative-resource-handler.

Listing 3.10: The configuration file of the relative-resource-handler from the example project.

```
<relative-resources xmlns="http://code.google.com/
a/apache-extras.org/p/relative-resource-handler">
    <url-version>1.0.0</url-version>
    <gzip-enabled>false</gzip-enabled>
    <locale-support-enabled>true</locale-support-enabled>

    <libraries>
        <library name="css">
            <location type="webapp">/resources/css</location>
            <el-evaluation>
                <file-mask>*.css</file-mask>
            </el-evaluation>
        </library>

        <library name="images">
            <location type="webapp">/resources/images</location>
        </library>

        <library name="static">
            <location type="external">
                http://localhost:8080/some/static/path/static/
            </location>
        </library>

        <library name="nolocation" />
    </libraries>
```

```
</relative-resources>
```

Every resource library, which should be handled by the relative-resource-handler, must be defined using a `<library>` entry in the configuration file. Each `<library>` entry must specify the name of the respective resource library, and can optionally specify the location of the library folder. Possible location types are:

- `webapp`: A subfolder in the web application root.
- `classpath`: A package in the web application classpath.
- `external`: An (external) HTTP URL (as defined in RFC 3986 [7]).

The usage of `webapp` corresponds to using a `WebappWeblet`, the usage of `classpath` corresponds to using a `PackagedWeblet`, and the usage of `external` corresponds to using a `URLWeblet` in Weblets (as explained in section 2.2). The latter allows to specify an arbitrary HTTP URL, thus allowing support for content delivery networks (CDNs). If no location is specified for a specific library, the standard locations for resource libraries of JSF 2 will be used (see section 2.3).
Every `<library>` entry can also contain a `<el-evaluation>` section. This section allows to specify one or more `<file-mask>` elements, each one declaring a file mask for which value expression support should be enabled. These file masks can reference simple files, or can also refer to many files of the same type, e.g. `*.css` for all stylesheets. This way, the relative-resource-handler also supports value expressions in resource files, allowing it to be backwards compatible to existing resource libraries built for the standard resource handler. However, it provides a much more fine grained activation of this feature, which lowers performance impacts. Furthermore, if the value expression support is enabled for a specific resource, this resource will be pre-evaluated at the time it is first referenced, and the result will be cached on the server. This again lowers the performance impacts of this feature.
Besides the library specification, the configuration file also contains three important elements:

- `<url-version>`
- `<gzip-enabled>`
- `<locale-support-enabled>`

The `<url-version>` element is used to define the version of the web application, which will later be included in every resource URL in order to achieve a proper resource update management as described above. When using a build tool like Apache Maven, it is possible to automatically set this property to the current project version.
In order to activate the resource compression support of the relative-resource-handler, the element `<gzip-enabled>` has to be set to `true`. If so, the relative-resource-handler will create a GZIP compressed version of every resource at the time the resource is referenced first. In addition, every client who supports GZIP compressed files will be served with the compressed version of the respective resource.
Finally, the `<locale-support-enabled>` element specifies if localized resources should be supported. If not, the resource URLs will not include the current locale of the user.

Extensibility

The relative-resource-handler allows to exchange two of its components by using the standard Java service loader mechanism [28]. Those components are:

- `RelativeResourceHandlerConfigProvider`
- `RelativeResourceResolverProvider`

By changing the default `RelativeResourceHandlerConfigProvider` it is possible to use a different approach to configure the relative-resource-handler. The default implementation of this component simply parses the configuration file and provides the read information without modifications. An enhanced version of this component could, for example, read the configuration from a different source (e.g. a database record), or could provide different configurations depending on the current project stage.
The `RelativeResourceResolverProvider` component is used by the relative-resource-handler to create instances of `RelativeResource`[6] and to calculate the resource identifier of a resource. An enhanced version of this component can be used to add custom criteria to the resource URLs, or to add multi tenancy support to the relative-resource-handler[7].

[6]The class `RelativeResource` is a subclass of `javax.faces.application.Resource`, which provides additional features required by the relative-resource-handler.

[7]The open source project actually contains an example module, in which a custom implementation of `RelativeResourceResolverProvider` adds multi tenancy support to the relative-resource-handler.

CHAPTER 4

Summary and future work

There are two main approaches to deal with resources in JSF 2 based web applications:

- Weblets
- JSF's standard resource handler

Weblets provides many important features, and usually works very well in practice, because it addresses most of the problems and requirements of section 1.2. However, in contrast to the resource handler of JSF it is a non-standard library, which does not make use of the resource handling concepts of JSF. The standard resource handler of JSF on the other hand heavily uses these concepts. Unfortunately, this resource handler has many (design) flaws, and furthermore many requirements of section 1.2 are not met.
The relative-resource-handler presented in the previous chapter combines the advantages of weblets with the standard resource handling concept of JSF 2, while eliminating existing (design) flaws of the respective technologies. In addition, many new features have been introduced, which are not present in the JSF 2 specification. Thus, using the relative-resource-handler in a JSF 2 based web application allows to use all the features of weblets (and even more), while still relying on the standardized APIs of the JSF specification. Furthermore, the relative-resource-handler is backwards compatible to resource libraries that have been created for the standard resource handler of JSF 2 (i.e. because they contain value expressions). Hence, an easy adoption of this resource handler can be achieved.
In contrast to weblets or the standard JSF resource handler, the problems of resource management (as outlined in section 1.2) are all covered by the relative-resource-handler.
The *Organization* of resources is partly taken over from the standard JSF resource handling mechanism, because the resources are allowed to be grouped together in libraries, which reside either directly in the web application, or in any JAR file on the classpath of the web application. Additionally, the relative-resource-handler supports external resource locations, which allows supporting separate resource servers or even content delivery networks (CDNs).

The *Delivery* of resources is either carried out directly by the relative-resource-handler, or by an external server as already mentioned above. If the delivery is performed by the relative-resource-handler, it supports GZIP compression for all resource files.
The problems in the area of *Caching and update mechanism* are also covered by the relative-resource-handler by the fact that it applies the REST paradigm to its resource URLs. This means that one specific resource URL will always return the exact same copy of a resource. If this resource is ever changed, or if a different version of that resource is required by the client, a different resource URL will be used, eliminating all cache problems.
The programmatic *Dependency management* of resources, as specified in the JSF 2 specification, is not touched by the relative-resource-handler. Thus, this part works exactly as defined by JSF 2, allowing to use the rich support for resource dependency references in Java. Furthermore, the relative-resource-handler allows static relative references between resources, which correspond to the relative paths between the resources on the file system. Because of this fact, an easy integration of resource libraries from third parties can be achieved, and the need for dynamic resource content can be eliminated.
In the long term, the relative-resource-handler should not simply serve as an additional library for any JSF 2 based web applications, it should rather be an upgrade to the standard resource handling mechanism of JSF 2. Therefore, the main concepts of the relative-resource-handler should be applied to a future JSF specification, in order to improve the standardized resource handling mechanism. More specific features of the relative-resource-handler, however, need not find their way into any future standard. Those features can be provided by a future, more simplified version of the relative-resource-handler.
Furthermore, as the building blocks (i.e. protocols) of the world wide web evolve, future versions of JSF must cope with new underlying technologies, e.g. HTTP 2.0 [13]. The relative-resource-handler project can provide support for these new technologies much sooner than the JSF standard can adopt them, because changes must not undergo the formal specification process required by the JCP. It can therefore serve as an early adopter, and adviser for future JSF expert groups.

Bibliography

[1] *JSF 1.1 Specification*. Java Community Process, 2004.

[2] *JSF 1.2 Specification*. Java Community Process, 2008.

[3] *JSF 2.0 Specification*. Java Community Process, 2010.

[4] *JSF 2.2 Specification draft*. Java Community Process, 2011.

[5] *Servlet 3.0 Specification*. Java Community Process, 2011.

[6] Java Specification Request 344. http://www.jcp.org/en/jsr/detail?id=344, November 2011.

[7] T. Berners-Lee, R. T. Fielding, and L. Masinter. Uniform Resource Identifier (URI): Generic Syntax. *RFC 3986*, January 2005.

[8] R. T. Fielding. Relative Uniform Resource Locators. *RFC 1808*, June 1995.

[9] R. T. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. *RFC 2616*, June 1999.

[10] Roy T. Fielding. Architectural Styles and the Design of Network-based Software Architectures. *Doctoral dissertation, University of California, Irvine*, 2000.

[11] Roy T. Fielding and Richard N. Taylor. Principled design of the modern Web architecture. *ACM Transactions on Internet Technology (TOIT)*, 2(2):115–150, May 2002.

[12] N. Freed and N. Borenstein. Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types. *RFC 2046*, November 1996.

[13] Ilya Grigorik. Making the Web faster with HTTP 2.0. *Communications of the ACM*, 56(12):42–49, December 2013.

[14] HTTP Archive. http://www.httparchive.org, December 2013.

[15] Bill Humphries. URLS! URLS! URLS! *A List Apart*, 70, June 2000.

[16] Geoff Huston. Web caching. *The Internet Protocol Journal*, 2(3):2–20, September 1999.

[17] Issue 2538 in the MyFaces issue tracker. https://issues.apache.org/jira/browse/MYFACES-2538, February 2010.

[18] Jonas Jacobi and John R. Fallows. *Pro JSF and Ajax: Building Rich Internet Components*. Springer, 1. edition, 2006.

[19] Jonas Jacobi and John R. Fallows. AJAX and Mozilla XUL with JavaServer Faces. *Java Developer's Journal, SYS-CON Media*, May 2007.

[20] Tom Leighton. Improving Performance on the Internet. *Communications of the ACM - Inspiring Women in Computing*, 52(2):44–51, February 2009.

[21] Icefaces CSS URL Mapping. http://www.icesoft.org/wiki/display/ICE/ACE+CSS+URL+Mapping, January 2011.

[22] San Murugesan. Understanding Web 2.0. *IT Professional*, 9(4):34–41, July-August 2007.

[23] Linda Dailey Paulson. Building Rich Web Applications with Ajax. *Computer*, 38(10):14–17, October 2005.

[24] Thomas A. Powell and Joe Lima. Towards Next Generation URLs. *System Design Frontier*, 2(6):41–47, June 2005.

[25] Oracle Press Release. http://www.oracle.com/us/corporate/press/018363, April 2009.

[26] Richard Richter. Anomaly detection and analysis of web traffic. *Master's Thesis, Czech Technical University in Prague*, May 2011.

[27] Ryan Dewsbury. *Google Web Toolkit Applications*. Addison-Wesley Professional, 1. edition, December 2007.

[28] ServiceLoader API Documentation. http://docs.oracle.com/javase/6/docs/api/java/util/ServiceLoader.html, December 2011.

[29] Tony C Shan and Winnie W Hua. Taxonomy of Java Web Application Frameworks. *2006 IEEE International Conference on e-Business Engineering (ICEBE'06)*, pages 378–385, October 2006.

[30] Steve Souders. High-Performance Web Sites. *Communications of the ACM - Surviving the data deluge*, 51(12):36–41, December 2008.

[31] Tim Berners-Lee. 'Cool URIs Don't Change.'. http://www.w3.org/Provider/Style/URI.html, 1998.

[32] Weblets 1.2 Online Documentation. http://werpu.github.io/weblets/www/doc_12/longdoc/index.html, November 2012.

[33] Weblets Project on GitHub. https://github.com/werpu/weblets, March 2014.

[34] J. Sergio Zepeda and Sergio V. Chapa. From Desktop Applications Towards Ajax Web Applications. *2007 4th International Conference on Electrical and Electronics Engineering (ICEEE 2007)*, pages 193–196, September 2007.