PrettyFaces Reference Guide

URL-rewriting, dynamic parameters, bookmarks, and navigation for Servlet, Java EE, and JSF

Version: 3.3.2-SNAPSHOT

by Lincoln Baxter III and Christian Kaltepoth

1. Introduction 1
1.1. Using this guide 1
1.2. What PrettyFaces provides 1
1.3. Business value of URL-rewriting 1
2. Get Started 3
2.1. Get PrettyFaces 3
2.2. Configure PrettyFaces in web.xml 4
2.3. Create /WEB-INF/pretty-config.xml 4
2.4. PrettyFaces Development Mode 6
3. Configure PrettyFaces 7
3.1. Create your first bookmarkable URL (mapping a URL) 7
3.2. Add Path-parameters to your mapped URL (REST) 7
3.2.1. Named path parameters 8
3.2.2. * EL-injected path parameters 9
3.2.3. Restrict what Path Parameters will accept (Custom regex patterns) 10
3.3. Manage Query-parameters in mapped URLs 12
3.4. Inherit from a parent URL-mapping 13
3.5. * Dynamic view-IDs (DynaView) 15
3.6. * Load data when accessing mapped URLs (Page-action methods) 16
3.6.1. Invoking navigation from a page-action 17
3.6.2. Invoking during specific JSF Phases 18
3.6.3. Toggling invocation on postback 18
3.7. * Parameter validation 19
3.7.1. Query parameters 19
3.7.2. Path parameters 20
3.8. Annotation based configuration 20
3.8.1. Basic setup 20
3.8.2. Simple URL mappings 21
3.8.3. Page actions 22
3.8.4. Query Parameters 24
3.8.5. Parameter validation 25
3.8.6. Bean name resolving 26
4. Order of Events 29
4.1. Request processing order 29
4.2. Configuration processing order 29
5. Simplify Navigation 31
5.1. * Integrating with JSF action methods 31
5.2. * Integrating with JSF commandLink and commandButton 33
5.3. Redirecting from non-JSF application Java code 33
5.4. * Preserving FacesMessages across redirects 34
6. Inbound URL Rewriting 35
6.1. Common rewriting features 35
6.2. Rewriting URLs that match a specific pattern 35
6.3. Rewrite options 36

7. Outbound URL-rewriting 39
7.1. Rewriting URLs in Java 39
7.2. * Rewriting URLs in JSF views 39
7.3. Disabling outbound-rewriting for a URL-mapping 41
8. * Rendering HTML links and URLs 43
8.1. The <pretty:link> component 43
8.2. The <pretty:urlbuffer> component 44
8.3. Using JSF standard components 44
9. Extend PrettyFaces 47
9.1. com.ocpsoft.pretty.faces.spi.ConfigurationProvider 47
9.2. com.ocpsoft.pretty.faces.spi.ConfigurationPostProcessor 47
9.3. com.ocpsoft.pretty.faces.spi.ELBeanNameResolver 48
9.4. com.ocpsoft.pretty.faces.spi.DevelopmentModeDetector 49
10. Feedback & Contribution 51
11. Updating Notes 53
11.1. Updating to 3.x.y from earlier versions 53
12. FAQ 55

Introduction

1.1. Using this guide

While compatible with any Servlet 2.5+ container, some features are only available in JavaServer Faces.

Sections describing these features are marked with a '*'. This means that the feature either requires JSF, or requires an environment that has configured JSF in order to function; otherwise, those marked features will not be available.

1.2. What PrettyFaces provides

PrettyFaces is an OpenSource extension for Servlet, Java EE, and JSF, which enables creation of bookmarkable, REST-ful, "pretty" URLs. PrettyFaces solves several problems elegantly, such as: custom URL-rewriting, page-load actions, seamless integration with JSF navigation and links, dynamic view-id assignment, managed parameter parsing, and configuration-free compatibility with other JSF frameworks.

1.3. Business value of URL-rewriting

Any business knows how important Search Engine Optimization can be for sales. PrettyFaces allows SEO-friendly URLs, and improved customer experience. Give your site a uniform, well understood feeling, from the address bar to the buy button.

What is sometimes neglected, even when building web-sites that aren't for external customers, is the consistency of the URL displayed to users in the browser address bar. Keeping the URL tidy can make a big difference in usability, providing that smooth web-browsing experience.

Example 1.1. A URL rewritten using PrettyFaces

Consider the following URL.

```
http://ocpsoft.com/index.jsf?post=docs&category=prettyfaces
```

When presenting information to users (frequently your clients,) it is generally bad practice to show more information than necessary; by hiding parameter names, we are able to clean up the URL. The following URL-mapping accomplishes our goal:

```
<url-mapping>
  <pattern value="/#{post}/#{category}" />
  <view-id>/index.jsf</view-id>
```

```
</url-mapping>
```

Our final result looks like this:

```
http://ocpsoft.com/docs/prettyfaces/
```

This is just a simple example of the many features PrettyFaces provides to standardize your URLs

Tip

Notice that outbound links encoded using *`HttpServletResponse.encodeRedirectURL(url)`* [`http://java.sun.com/webservices/docs/1.6/api/javax/servlet/http/HttpServletResponse.html#encodeRedirectURL%28java.lang.String%29`] will also be automatically rewritten to new URLs, unless disabled by using the `outbound="false"` attribute on a given URL-mapping configuration.

Get Started

It should not be difficult to install PrettyFaces into any new or existing Servlet-based application. Follow the next few steps and you should have a working configuration in only a few minutes.

2.1. Get PrettyFaces

This step is pretty straight-forward, right? Copy the PrettyFaces JAR file into your /WEB-INF/lib directory, or include a *Maven* [http://maven.apache.org/] dependency in your pom.xml (recommended):

Non-Maven users may download JAR files manually from one of the following repositories: (*Central* [http://repo1.maven.org/maven2/com/ocpsoft/], *OcpSoft* [http://ocpsoft.com/repository/com/ocpsoft/]). Be sure to select the correct package for your version of JSF.

Note

As of PrettyFaces 3.3.0, no additional JAR files for are required for Non-Maven users; installation is just a single download and install.

`List of Maven Dependencies:`

```
<!-- for JSF 2.x -->
<dependency>
  <groupId>com.ocpsoft</groupId>
  <artifactId>prettyfaces-jsf2</artifactId>
  <version>${latest.prettyfaces.version}</version>
</dependency>

<!-- for JSF 1.2.x -->
<dependency>
  <groupId>com.ocpsoft</groupId>
  <artifactId>prettyfaces-jsf12</artifactId>
  <version>${latest.prettyfaces.version}</version>
</dependency>

<!-- for JSF 1.1.x (UNSUPPORTED) -->
<dependency>
  <groupId>com.ocpsoft</groupId>
  <artifactId>prettyfaces-jsf11</artifactId>
  <version>${latest.prettyfaces.version}</version>
</dependency>
```

2.2. Configure PrettyFaces in web.xml

If you are using a Servlet 3.0 compliant container, you may skip this step because PrettyFaces will automatically register the required Servlet Filter; otherwise, make sure PrettyFilter is the first filter in your web.xml file. (The dispatcher elements are required to ensure PrettyFaces intercepts both internal and external requests.)

Example 2.1. /WEB-INF/web.xml (Ignore if using Servlet 3.0)

```
<filter>
  <filter-name>Pretty Filter</filter-name>
  <filter-class>com.ocpsoft.pretty.PrettyFilter</filter-class>
</filter>

<filter-mapping>
  <filter-name>Pretty Filter</filter-name>
  <url-pattern>/*</url-pattern>
  <dispatcher>FORWARD</dispatcher>
  <dispatcher>REQUEST</dispatcher>
  <dispatcher>ERROR</dispatcher>
</filter-mapping>
```

2.3. Create /WEB-INF/pretty-config.xml

This is where you'll tell PrettyFaces what to do, which URLs to rewrite. Each URL-mapping contained in the configuration must specify a pattern and a viewId. The pattern specifies which inbound URL to match, and the viewId specifies the location that URL should resolve -be redirected- to.

Example 2.2. /WEB-INF/pretty-config.xml (You will need to customize this)

```
<pretty-config xmlns="http://ocpsoft.com/prettyfaces/3.3.0"
               xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xsi:schemaLocation="http://ocpsoft.com/prettyfaces/3.3.0
                                   http://ocpsoft.com/xml/ns/prettyfaces/ocpsoft-pretty-faces-3.3.0.xsd">

  <!-- Begin Example RewriteRules

    // These are custom rewrite-rules, and are probably not necessary for your application.

    <rewrite match="^/old-url/(\w+)/$" substitute="/new_url/$1/" redirect="301" />
```

```
  -->

  <!-- Begin UrlMappings
    // These are examples of URL mappings, and should be customized for your application.

    <url-mapping id="home">
        <pattern value="/" />
        <view-id value="/faces/index.jsf" />
    </url-mapping>

    <url-mapping id="store">
        <pattern value="/store/" />
        <view-id value="/faces/shop/store.jsf" />
    </url-mapping>

    <url-mapping id="viewCategory">
        <pattern value="/store/#{ cat : bean.category }/" />
        <view-id value="/faces/shop/store.jsf" />
    </url-mapping>

    <url-mapping id="viewItem">
        <pattern value="/store/#{ cat : bean.category }/#{ iid : bean.itemId }/" />
        <view-id value="/faces/shop/item.jsf" />
        <action>#{bean.loadItem}</action>
    </url-mapping>

  -->

</pretty-config>
```

Tip

In JAR files, pretty-config.xml may also be placed in /META-INF/pretty-config.xml - additionally, comma-separated configuration file locations may be specified in web.xml, using the servlet context-param configuration API:

```
<context-param>
 <param-name>com.ocpsoft.pretty.CONFIG_FILES</param-name>
                <param-value>/WEB-INF/custom-mappings.xml,/META-INF/another-
config.xml</param-value>
</context-param>
```

Congratulations! That's all you should have to do in order to use PrettyFaces. The rest of this reference guide covers detailed configuration options, framework integration with JavaServer Faces, and special use-cases.

2.4. PrettyFaces Development Mode

During the development of your application you will typically have to make changes to your `pretty-config.xml` very often. As server restarts take a lot of time and decrease productivity, PrettyFaces is capable of reloading its configuration on a regular basis. As the configuration reloading may be a time consuming process for large applications, PrettyFaces reloads its configuration only if the application is executed in development mode.

To enable the development mode, add the following context parameter to your `web.xml`:

```
<context-param>
  <param-name>com.ocpsoft.pretty.DEVELOPMENT</param-name>
  <param-value>true</param-value>
</context-param>
```

If your are using the JSF 2.0 version of PrettyFaces, you typically won't have to explicitly enable the development mode. In this case the development mode is automatically enabled for all project stages except for `Production`.

Configure PrettyFaces

PrettyFaces offers in-depth configuration options for completely customizable use. We'll start with the most commonly used features, then move on to the advanced goodies.

3.1. Create your first bookmarkable URL (mapping a URL)

The URL mapping element is the core configuration element for PrettyFaces most applications. Let's consider, for instance, that we are building a web-store and we want our users to access the store at the URL `/store/` - we only need the most basic features of the URL-mapping to achieve this.

```
<url-mapping id="store">
  <pattern value="/store/" />
  <view-id value="/faces/shop/store.jsf" />
</url-mapping>
```

With these two simple lines of configuration, the user sees: `pattern="/store/"` in the browser URL and in the output HTML, but the server is actually rendering the resource: `/faces/shop/store.jsf` (the actual location of the page on the server.)

Ignore the `id="store"` attribute for now, we'll cover that under *navigation*. Which is useful when redirecting a user between different pages in your application.

3.2. Add Path-parameters to your mapped URL (REST)

Suppose we want to create a URL mapping that allows users to access items in a specific category of a web-store, such as: `/store/category/`. In other words, we want to take a piece of the URL string itself - for example: `/store/[category]/` and use that value: `[category]` in our application's logic.

This is commonly known as creating a REST-ful URL, but here we'll use the term "path-parameter" to represent each individual data element created in our URL path.

Tip

Path-parameters are defined by using #{...} expressions in the pattern element. By default, these expressions match any character except for '/' in a URL. So the following pattern will match `/foo/`, but it will not match `/foo/bar/`.

```
<pattern value="/#{ name }/" />
```

More specifically, path-parameter expressions such as `#{ name }` are replaced with the regular expression `[^/]+` before being matched against the URL. *Custom regular expressions* can also be supplied if more control over parameter matching is required.

```
<url-mapping id="viewCategory">
  <pattern value="/store/#{ cat }/" />
  <view-id value="/faces/shop/store.jsf" />
</url-mapping>
```

Here we've used PrettyFaces path-parameter syntax to capture values stored in part of inbound URLs that match our pattern. Take, for instance, the following URL:

```
/store/shoes/
```

Part of this URL will be matched by the path-parameter expression: '#{ cat }', making its value, 'shoes', available in the application logic, the rest of the URL will be matched against the pattern, but ignored.

```
<pattern value="/store/#{ cat }/" />
```

The application can gain access to these path-parameters in a few ways, via *request parameter naming*, or *EL bean injection*; both techniques will be covered below.

3.2.1. Named path parameters

In order to gain access to a value matched by our path-parameter `#{...}`, we can use the named path-parameter syntax: `#{ cat }`, which means that the value in the URL will be available in the HTTP Servlet request parameter map: `HttpServletRequest.getParameter(String name); stored in the key 'cat'.`

In our application code, we would access the URL just like we access request query-parameters, e.g.: `url?cat=value` , where 'cat' is the key, and 'value' is the value.

```
String category = request.getParameter("cat");
```

You could also use JSF 2.0 meta-data *view-parameters* [https://javaserverfaces.dev.java.net/nonav/docs/2.0/pdldocs/facelets/f/viewParam.html] to propagate the named request parameter value to EL.

Tip

If you are adding PrettyFaces to an existing web-application, it is very likely that you will want to use named path-parameters in order to tie in to existing application features.

When starting a new application, it is good practice to always include a name for each path-parameter, even if you are using *EL bean injection* as well.

3.2.2. * EL-injected path parameters

Another method of accessing path-parameter values is via EL value injection, where PrettyFaces can inject the value of the URL parameter directly into a managed-bean. This requires a syntax similar to specifying a named parameter, but PrettyFaces looks for '.' characters, an easy way to distinguish a name from an EL value-injection expression:

```
<pattern value="/store/#{ bean.category }/" />
```

In this case, we must have a managed bean defined in our application; these beans can be registered either in JSF, Spring, Google Guice, or as shown here using CDI/Weld. Based on the configuration in this particular example, the bean must be named "bean", and must have an accessible field named "category".

Any value matched by the path-parameter will be injected directly into the bean.

```
@Named("bean")
@RequestScoped
public class CategoryBean {
 private String category;

 /* Getters & Setters */
}
```

Please note that PrettyFaces will automatically use the JSF converter registered for the type of the referenced bean property to convert the path parameter. This means that PrettyFaces supports all JSF standard converters and converters that have been manually registered to be used for a specific type using the `converter-for-class` element in the `faces-config.xml` (or the `forClass` attribute of the `@FacesConverter` annotation).

Tip

Notice, you can specify both a name and an EL value-injection for the same path-parameter.

```
<pattern value="/store/#{ cat : bean.category }/" />
```

3.2.3. Restrict what Path Parameters will accept (Custom regex patterns)

In PrettyFaces, each url-pattern compiles into a regular expression that is used to match incoming URLs. By default, any path-parameter expressions (`#{...}`) found in the URL pattern will be compiled into the regular expression: `[^/]+`, meaning that path-parameters do not match over the '/' character.

The time will come when you need to make a URL match more selectively, for instance, when you have two URLs that share the same parameter order. Or perhaps you need to take a very complex URL and parse it more selectively (matching across multiple '/' characters, for instance.) It is in scenarios like these when you should consider using a custom regular expression within your url-mapping pattern.

Tip

When using any escape sequences in a custom regular expression, be sure to "double-escape" the backslash character '\', otherwise the pattern will not be properly compiled, just as you would in Java itself. For example, in order to match digits: "`\d+`", you would actually need to add an additional backslash, like so: "`\\d+`"

Let's consider, for instance, that you are building a blog application, where the URLs: `/blog/2010/12/` and `/blog/lincoln/23/` have the same number of parameters, but mean different things. One is going to display a list of articles from December 2010, and the other is going to display Lincoln's 23rd post. There has to be a way for the system to tell the two apart, and the answer is custum regex patterns.

```
<url-mapping id="archives">
  <pattern value="/#{ /\\d{4}/ year }/#{ /\\d{2}/ month }/" />
  <view-id value="/faces/blog/archives.jsf" />
</url-mapping>
```

This pattern specifies custom regular expressions for each path-parameter, the former must match exactly four digits (numbers 0-9), while the latter must match exactly two digits. In order to understand what this means, it might make sense to think of what the pattern would look like once it is compiled into a regular expression:

```
/(\\d{4})/(\\d{2})/
```

Below we see how to map the second of our two example URLs.

```
<url-mapping id="viewAuthorPost">
  <pattern value="/#{ /[a-z]+/ blogger }/#{ /\\d+/ postId }/" />
  <view-id value="/faces/blog/viewPost.jsf"/>
</url-mapping>
```

Notice that this compiled pattern looks somewhat different from the first example URL:

```
/([a-z]+)/(\\d+)/
```

This is how you can completely customize and parse complex URLs (even capture values that include the '/' character.) Any path-parameter expression can accept a `/custom-regex/`, but the custom regex must be placed before any name or EL injections.

Warning

Using custom regular expressions can lead to very difficult trouble-shooting, and is only recommended in cases where basic path-parameter expressions are not sufficient.

Sometimes you might want PrettyFaces to inject path parameters only on GET requests. This could be the case if you inject path parameters into a view-scoped bean and want to change these values at a later time.

You can use the `onPostback` attribute of `url-mapping` to specifiy if values should be injected on postbacks. Please note that the default value of the attribute is `true`.

```
<url-mapping id="viewCategory" onPostback="false">
  <pattern value="/store/#{ cat }/" />
  <view-id value="/faces/shop/store.jsf"/>
</url-mapping>
```

Warning

Please note that the bean properties bound to path parameters must never be `null` as they are required to build the URL for form postbacks.

3.3. Manage Query-parameters in mapped URLs

Most people are already familiar with URL query-parameters. They come in `key=value` pairs, and begin where the '?' character is found in a URL. For example:

```
http://example.com/path?query=data
```

Here, 'query' is the name of the parameter, and 'data' is the value of the parameter; order of parameters does not matter, and if duplicates parameter names are encountered, an array of multiple values will be stored for that parameter.

While query-parameters are automatically stored in the `HttpServletRequest` parameter map, it may sometimes be convenient to also inject those values directly into JSF managed beans.

```
<query-param name="language"> #{bean.language} </query-param>
```

In this case, we must have a managed bean defined in our application; these beans can be registered either in JSF, Spring, Google Guice, or as shown here using CDI/Weld. Based on the configuration in this particular example, the bean must be named "bean", and must have an accessible field named "language".

```
<url-mapping id="store">
  <pattern value="/store/" />
  <view-id value="/faces/shop/store.jsf" />
  <query-param name="language"> #{bean.language} </query-param>
</url-mapping>
```

Any value matched by the query-parameter will be injected directly into the bean.

```
@Named("bean")
@RequestScoped
public class LanguageBean {
 private String language;
```

```
 /* Getters + Setters */
}
```

Please note that PrettyFaces will automatically use the JSF converter registered for the type of the referenced bean property to convert the query parameter. This means that PrettyFaces supports all JSF standard converters and converters that have been manually registered to be used for a specific type using the `converter-for-class` element in the `faces-config.xml` (or the `forClass` attribute of the `@FacesConverter` annotation).

Tip

By default, query-parameters are URL-decoded. If this is not desired, URL-decoding may be disabled using the optional `decode="false"` attribute:

```
<query-param name="language" decode="false"> #{bean.language} </query-param>
```

In some situations you might want that PrettyFaces doesn't inject the value of a query parameter on JSF postbacks. A typical usecase for this would be a query parameter that is used to initially populate a bean property holding the value bound to an input component. In this case you will want the query parameter value to be injected only on the initial GET request but not on postbacks, because the postback will contain the submitted value of the input component.

You can use the `onPostback` attribute to tell PrettyFaces whether you want the query parameter to be injected on postbacks. Please note that the default value of the attribute is `true`.

```
<query-param name="query" onPostback="false">#{searchBean.query}</query-param>
```

3.4. Inherit from a parent URL-mapping

Frequently, you may find that several - or all - of your mappings share a common base-URL. It is on these occasions when you should consider using a parent URL-mapping by using the `parentId` attribute.

```
<url-mapping parentId="store" id="category"> ... </url-mapping>
```

Notice that the `parentId` attribute must refer to a `mappingId` of another URL-mapping. See examples below:

```
<url-mapping id="store">
  <pattern value="/store/" />
  <~-- Result: /store/ -->
  <view-id value="/faces/shop/store.jsf" />
</url-mapping>

<url-mapping parentId="store" id="category">
  <pattern value="/#{category}" />
  <~-- Result: /store/#{category} -->
  <view-id value="/faces/shop/category.jsf" />
</url-mapping>

<url-mapping parentId="category" id="item">
  <pattern value="/#{item}" />
  <~-- Result: /store/#{category}/#{item} -->
  <view-id value="/faces/shop/item.jsf" />
</url-mapping>

<url-mapping parentId="category" id="sales">
  <pattern value="/sales" />
  <~-- Result: /store/#{category}/sales -->
  <view-id value="/faces/shop/sales.jsf" />
</url-mapping>
```

Child mappings will inherit all properties from the parent mapping. This includes the pattern, validators and query parameters. URL actions won't be inherited per default. If you want an action to be inherited by child mappings, set it's `inheritable` attribute to `true`.

```
<url-mapping id="store">
  <pattern value="/store/" />
  <view-id value="/faces/shop/store.jsf" />
  <action inheritable="true">#{storeBean.someAction}</action>
</url-mapping>

<url-mapping parentId="store" id="category">
  <pattern value="/#{category}" />
  <view-id value="/faces/shop/category.jsf" />
</url-mapping>
```

3.5. * Dynamic view-IDs (DynaView)

Dynamic view-IDs (referred to as DynaView) allow us to determine (at the time the page is requested) the page our users should see for a given URL-mapping pattern.

```
<url-mapping id="home">
 <pattern value="/home/" />
 <view-id value="#{bean.getViewPath}" />
</url-mapping>
```

Notice that we've used an EL method-expression where we would normally have specified a view-ID. PrettyFaces will invoke this method-expression, and use the result of the method as a final URL to be displayed by the server. In a sense, we're asking the system to figure out which page to display, and telling it that it can get the answer by calling our method:

```
@Named("bean")
@RequestScoped
public class HomeBean {

 @Inject CurrentUser user;

 public String getViewPath() {
  if ( user.isLoggedIn() )
  {
   return "/faces/home/home.jsf";
  }

  return "/faces/login.jsf";
 }
}
```

Here, our method `#{bean.getViewPath}` is going to check the current user's logged-in status, display the home page if he/she is logged in, or display the login page if they are not.

Caution

Automatic *out-bound URL-rewriting* will not function on pages with dynamic view-ID's.

3.6. * Load data when accessing mapped URLs (Page-action methods)

Caution

This feature is currently only available for URL mappings that target a JSF view-ID.

Most of the time, when creating bookmarkable URLs, it is not enough to simply display a page to the user; we also need to load data to be shown on that page - allowing for pages to appear completely stateless to the end-user. This would typically be difficult in JSF, but PrettyFaces has another option to satisfy this requirement that breaks the coupling typically associated with other solutions such as using @SessionScoped data beans to save data across pages, or passing values across views using: <f:setPropertyActionListener/>.

Consider, for a moment, that we have a web-store, and would like to map a URL to display one specific item in that store:

Example 3.1. Displaying a single item in a web-store.

Here we have defined a page-action to be executed on a bean, `#{bean.loadItem}`, when a URL matching our pattern is requested. For example: `/store/item/32`.

```
<url-mapping id="viewItem">
  <pattern value="/store/item/#{ iid : bean.itemId }/" />
  <view-id value="/faces/shop/item.jsf" />
  <action>#{bean.loadItem}</action>
</url-mapping>
```

And the corresponding managed-bean, here shown using CDI/Weld annotations:

```
@Named("bean")
@RequestScoped
public class CategoryBean {
 private String itemId;
 private String category;

 private Item item;

 @Inject StoreItems items;

 public String loadItem() {
```

```
  if ( itemId != null ) {
   this.item = items.findById(itemId);
   return null;
  }

  // Add a message here, "The item {..} could not be found."
  return "pretty:store";
 }

 /* Getters & Setters */
}
```

3.6.1. Invoking navigation from a page-action

Once we have defined our action method, it is very likely that situations will arise where we do not want to continue displaying the current page, say, when data could not be loaded, or if the user does not have sufficient permissions to perform the action; instead, we want to redirect the user to another page in our site.

This can be done by returning a JSF navigation-string, just like we would do from a normal JSF action-method. PrettyFaces *integrated navigation* can also be used to perform dynamic redirection.

Example 3.2. Invoking JSF navigation from a page-action method

Here we have defined a page-action to be executed on a bean, `#{bean.loadItem}`, when a URL matching our pattern is requested. For example: `/store/item/32`.

```
<url-mapping id="viewItem">
  <pattern value="/store/item/#{ iid : bean.itemId }/" />
  <view-id value="/faces/shop/item.jsf" />
  <action>#{bean.loadItem}</action>
</url-mapping>
```

The corresponding managed-bean is shown here using CDI/Weld annotations. If the item cannot be found, we will invoke JSF navigation for the outcome: "failure". This outcome must be defined in `faces-config.xml`, or JSF will not know how to process it.

```
@Named("bean")
@RequestScoped
public class CategoryBean {

 public String loadItem() {
```

```
  if ( itemId != null ) {
   this.item = items.findById(itemId);
   return null;
  }

  return "failure";
 }
}
```

If a PrettyFaces URL-mapping ID is returned in this manner (e.g: "pretty:mappingId"), PrettyFaces *integrated navigation* will be invoked instead of JSF navigation.

3.6.2. Invoking during specific JSF Phases

By default, actions are executed after the Faces RESTORE_VIEW phase, but actions may also be triggered `before` other JSF phases, using the optional `phaseId="..."` attribute.

```
<action phaseId="RENDER_RESPONSE">#{bean.loadItem}</action>
```

Caution

If the JSF Phase does not occur, page-actions associated with that phase will not execute; in this case -for example- if no validation is required, this action will never be called:

```
<action phaseId="PROCESS_VALIDATIONS">#{bean.loadItem}</action>
```

3.6.3. Toggling invocation on postback

Sometimes we only need to trigger a page-action the first time a page is requested, and not when the user submits forms on that page. In order to accomplish this, we use the optional `onPostback="false"` attribute.

```
<action onPostback="false">#{bean.loadItem}</action>
```

This will ensure that the action is never executed if the user is submitting a JSF form via POST (otherwise referred to as a postback.) The action will only be executed if the page is accessed via a HTTP GET request.

3.7. * Parameter validation

The validation of path and query parameters is very important as they are directly modifiable by the user. Therefore PrettyFaces offers the possibility to attach validation code to each of your parameters.

3.7.1. Query parameters

Validation of query parameters can be achieved by attaching standard JSF validators to the parameter.

```
<query-param name="language" validatorIds="languageValidator"> #{bean.language} </query-param>
```

This example shows how to attach the JSF validator with the ID `languageValidator` to the query parameter. This validator will be executed directly after the URL has been parsed. If the validation fails, PrettyFaces will send a 404 HTTP response. You can change this behavior by using the `onError` attribute and specifying an alternative view to show if validation fails.

```
<query-param name="language" validatorIds="languageValidator" onError="pretty:error"> #{bean.language} </query-param>
```

Tip

You can attach multiple validators to the same query parameter by using a space-separated list of validator IDs.

If you don't want to write a custom JSF validator, you can also reference a managed bean method performing the validation. This method must have the exact same signature as the `validate` method of the `Validator` interface

```
<query-param name="language" validator="#{languageBean.validateLanguage}"> #{bean.language} </query-param>
```

```
@Named("languageBean")
@RequestScoped
public class LanguageBean {

 public void validateLanguage(FacesContext context, UIComponent component, Object value) {
```

```
    for( String lang : Locale.getISOLanguages() ) {
      if( lang.equals( value.toString() ) ) {
        return;
      }
    }
    throw new ValidatorException( new FacesMessage("invalid.language") )

  }

}
```

3.7.2. Path parameters

Validation of path parameters is very similar to the validation of query parameters. You just have to add a `validate` element to the `pattern` element of your mapping. It accepts the same validation attributes as the `query-param` element.

The only important difference to the declaration of query parameter validation is that you have to specify the `index` of the path parameter you want to validate. The `index` is the absolute position of the path parameter in the pattern. The first parameter is addressed with `0`.

```
<url-mapping id="viewCategory">
  <pattern value="/store/#{ cat }/">
    <validate index="0" validatorIds="categoryValidator" onError="pretty:error" />
  </pattern>
  <view-id value="/faces/shop/store.jsf" />
</url-mapping>
```

3.8. Annotation based configuration

Recently PrettyFaces added support to configure URL mappings via annotations. This feature is primarily intended for people who don't want to maintain a separate XML configuration file for the mappings and instead prefer to declare them directly on the affected classes.

3.8.1. Basic setup

PrettyFaces supports configuration via annotations out of the box. Nevertheless it is strongly recommended to manually specify the java packages that contain your annotated classes. In this case the annotation scanner isn't required to scan the complete classpath, which might be a performance problem when you have many dependencies.

You can specify the packages to scan for annotations by adding a comma-separated list of packages to your `web.xml`:

```
<context-param>
  <param-name>com.ocpsoft.pretty.BASE_PACKAGES</param-name>
  <param-value>com.example.myapp,com.ocpsoft</param-value>
</context-param>
```

PrettyFaces will scan these packages recursively. So typically you will only have to add the top-level package of your web application here.

If you don't want to use PrettyFaces annotations at all, you can completely disable the annotation scanning by setting the package configuration parameter to `none`.

```
<context-param>
  <param-name>com.ocpsoft.pretty.BASE_PACKAGES</param-name>
  <param-value>none</param-value>
</context-param>
```

In the default configuration PrettyFaces will only scan for annotations in the `/WEB-INF/classes` directory of your web application. If you want the JAR files in `/WEB-INF/lib` to be scanned as well, add the following entry to your `web.xml`:

```
<context-param>
  <param-name>com.ocpsoft.pretty.SCAN_LIB_DIRECTORY</param-name>
  <param-value>true</param-value>
</context-param>
```

3.8.2. Simple URL mappings

To create a simple URL mapping, you must annotate one of your beans with a `@URLMapping` annotation. You will typically want to place this annotation on a class that is primarily responsible for the page.

```
@Named("bean")
@RequestScoped
@URLMapping(id = "store", pattern = "/store/", viewId = "/faces/shop/store.jsf")
public class StoreBean {
  /* your code */
}
```

You can see that the annotation attributes are very similar to the attributes of the `url-mapping` element in the PrettyFaces XML configuration file. Refer to *Mapping a simple URL* for details on the configuration of URL mappings.

If you want to use path parameters in the URL pattern, you can add these the same way as you would in `pretty-config.xml`.

```
@Named("bean")
@RequestScoped
@URLMapping(id = "categoryBean", pattern = "/store/#{ bean.category }/", viewId = "/faces/shop/store.jsf")
public class CategoryBean {

  private String category;

  /* Getters & Setters */
}
```

Sometimes you may want to declare multiple URL mappings on a single class. Unfortunately Java does not allow to add the same annotation to a class more than once. PrettyFaces offers a simple container annotation called `@URLMapping` that can be used in this case.

```
@Named("bean")
@RequestScoped
@URLMappings(mappings={
  @URLMapping(id = "categoryBean", pattern = "/store/#{ bean.category }/", viewId = "/faces/shop/store.jsf"),
  @URLMapping(id = "categoryBean2", pattern = "/shop/#{ bean.category }/", viewId = "/faces/shop/store.jsf")
})
public class CategoryBean {

  private String category;

  /* Getters & Setters */
}
```

3.8.3. Page actions

PrettyFaces offers a very intuitive way to specify page actions with annotations. All you have to do is add a `@URLAction` annotation to the method you want to be executed.

```
@Named("bean")
@RequestScoped
@URLMapping(id = "viewItem", pattern = "/store/item/#{ bean.itemId }/", viewId = "/faces/shop/item.jsf")
public class CategoryBean {

  private String itemId;

  private Item item;

  @Inject
  StoreItems items;

  @URLAction
  public String loadItem() {
    if ( itemId != null ) {
      this.item = items.findById(itemId);
      return null;
    }

    // Add a message here, "The item {..} could not be found."
    return "pretty:store";
  }

  /* Getters & Setters */
}
```

Note

In some environments you may need to add an additional @URLBeanName annotation to the class containing the page action. Please refer to the chapter *Bean name resolving* for details.

Note

If the class the annotated method belongs to declares multiple URL mappings using the `@URLMappings` annotation, the action will be used for each of the mappings.

The annotation supports all attributes that are available in the XML configuration file.

```
@URLAction(phaseId=PhaseId.RENDER_RESPONSE, onPostback=false)
```

```
public String loadItem() {
  // do something
}
```

Sometimes you might want to call methods on other beans than the bean annotated with the @URLMapping. In this case just refer to the foreign mapping using the `mappingId` attribute.

```
@Named("bean")
@RequestScoped
@URLMapping(id = "viewItem", pattern = "/store/item/#{ bean.itemId }/", viewId = "/faces/shop/item.jsf")
public class CategoryBean {
  /* some code */
}

@Named("otherBean")
@RequestScoped
public class OtherBean {

  @URLAction(mappingId = "viewItem")
  public void myAction() {
    /* your code */
  }

}
```

3.8.4. Query Parameters

If you configure PrettyFaces using annotations, you can declare query parameters by adding a `@URLQueryParameter` annotation to the target field. PrettyFaces will then inject the value of the query parameter into that field.

```
@Named("bean")
@RequestScoped
public class LanguageBean {

  @URLQueryParameter("language")
  private String language;

  /* Getters + Setters */
}
```

Note

Please not that the annotated field must be accessible by the expression language. This means that you'll have to create getters and setters for the field, so that PrettyFaces can inject the value into the object.

Note

In some environments you may need to add an additional @URLBeanName annotation to the class containing the query parameter. Please refer to the chapter *Bean name resolving* for details.

Note

If the class the annotated field belongs to declares multiple URL mappings using the `@URLMappings` annotation, the query parameter will be used for each of the mappings.

3.8.5. Parameter validation

Validation is of major importance when processing any kind of user input. This also applies to path and query parameters as they are directly modifiable by the user.

The declaration of validation rules is very simple when using PrettyFaces annotations. To validate a query parameter with a standard JSF validator, you'll just have to add a `@URLValidator` annotation to the field.

```
@Named("bean")
@RequestScoped
public class LanguageBean {

  @URLQueryParameter("language")
  @URLValidator(validatorIds="com.example.LanguageValidator")
  private String language;

  /* Getters + Setters */
}
```

This example shows how to attach the `com.example.LanguageValidator` validator to the query parameter `language`. You can also specify a mapping to redirect to if the validation fails or attach multiple validators to the same query parameter.

```
@Named("bean")
@RequestScoped
public class LanguageBean {

  @URLQueryParameter("language")
  @URLValidator(onError="pretty:error",
     validatorIds= { "com.example.LanguageValidator", "com.example.OtherValidator" })
  private String language;

  /* Getters + Setters */
}
```

To validate path parameters, you have to add the `@URLValidator` to the `@URLMapping` and specify the index of the path parameter you are referring to.

```
@Named("bean")
@RequestScoped
@URLMapping(id = "viewItem", pattern = "/store/item/#{ bean.itemId }/", viewId = "/faces/shop/
item.jsf",
  validation=@URLValidator(index=0, validatorIds="com.example.ItemIdValidator"))
public class CategoryBean {
  /* some code */
}
```

This will tell PrettyFaces to validate the first path parameter `#{bean.itemId}` with the validator `com.example.ItemIdValidator`.

3.8.6. Bean name resolving

PrettyFaces is required to access your managed beans in multiple ways. If you declare a page action to be executed for a specific URL, the framework will create a method expression and execute it. If you want to inject a query parameter into your bean, a value expression is created to write the value into the field.

All these actions require PrettyFaces to know the name of your beans in the EL context. In most cases this can be done by an auto-detection mechanism that supports the most common environments for defining managed beans. Currently PrettyFaces supports:

- JSF standard mechanism (`faces-config.xml` and `@ManagedBean`)
- CDI (Weld/OpenWebBean)
- JBoss Seam 2.0

- Spring

If you are using a non-standard way of defining managed beans within your application, the auto-detection will not work. In this case you'll have two options.

The first option is to use a `@URLBeanName` annotation on the class to explicitly tell PrettyFaces the name of the bean. The framework will then use this name to build EL expressions for this bean.

```
@URLBeanName("bean")
public class LanguageBean {

  @URLQueryParameter("language")
  private String language;

  /* Getters + Setters */
}
```

In this example PrettyFaces will create the EL expression `#{bean.language}` to access the language field.

The other option to tell PrettyFaces about your beans names is to implement a custom `BeanNameResolver`. PrettyFaces already ships with resolver implementations for the most common environments. If your environment is not supported, you can easily *create your own resolver*.

Order of Events

PrettyFaces follows a set order of events when processing each request. If a request is not mapped, or does not match a rewrite-rule, then the request will not be processed by PrettyFaces, and will continue normally.

4.1. Request processing order

1. URL-matching, *path-parameter* parsing, *query-parameter* handling, and *value injection* into managed beans.
2. *DynaView* calculation (if a view Id is dynamic, the EL method will be called.)
3. PrettyFaces relinquishes control of the current request via RequestDispatcher.forward("/context/faces/viewId.jsf").
4. If the view-ID is a JSF view, *page-action* methods are called after RESTORE_VIEW phase, unless the optional phaseId attribute is specified.
5. The server continues to process the request as normal.

4.2. Configuration processing order

1. Search for classpath resources named META-INF/pretty-config.xml in the ServletContext resource paths for this web application, and load each as a configuration resource.
2. Check for the existence of a context initialization parameter named com.ocpsoft.pretty.CONFIG_FILES. If it exists, treat it as a comma-delimited list of context relative resource paths (starting with a /), and load each of the specfied resources.
3. Check for the existence of a web application configuration resource named /WEB-INF/pretty-config.xml, and load it if the resource exists.

Simplify Navigation

Navigation is a critical part of any web-application, and PrettyFaces provides simple integrated navigation - both with JSF, and in non-JSF environments (such as Servlets) that only have access to the `HttpRequest` object. PrettyFaces navigation is non-intrusive, and may be used optionally at your discretion.

5.1. * Integrating with JSF action methods

Typically when navigating in JSF, one must define navigation cases (in JSF 1.x) or return the path to a view (in JSF 2.x). PrettyFaces, however, lets us simply reference the URL-mapping ID in these scenarios. Since the URL-mapping pattern already contains the locations of all required values (path-parameter EL expressions), these values are extracted from managed beans and used in URL generation.

Simply return a URL-mapping ID from any JSF action-method just as you would when using an `<h:commandLink action="..." />`. This outcome string should be prefixed by "pretty:" followed e.g.: "pretty:store", where "store" is our URL-mapping ID.

```
<url-mapping id="viewItem">
  <pattern value="/store/item/#{ iid : bean.itemId }/" />
  <view-id value="/faces/shop/item.jsf" />
  <action>#{bean.loadItem}</action>
</url-mapping>
```

Now look at our action-method; notice that if the item is found, we return null, signaling that no navigation should occur, but if the item cannot be loaded, we return `"pretty:store"` which is a PrettyFaces navigation outcome; PrettyFaces will intercept this outcome string, and redirect the user to the URL-mapping identified by "store" in `pretty-config.xml`.

```
public String loadItem() {
 if(itemId != null) {
  this.item = items.findById(itemId);
  return null;
 }

 // Add a message here, "The item {..} could not be found."
 return "pretty:store";
}
```

Tip

Returning "pretty:", without a URL-mapping ID will cause the current page to be refreshed.

If any action method (JSF action-method or PrettyFaces page-actions) returns a mapping-ID that specifies *path-parameters* in the pattern, PrettyFaces will automatically extract values from bean locations specified in the pattern. The client browser will be redirected to the URL built using these values injected back into the pattern. For example:

```
<pattern value="/store/#{ cat : bean.category }/" />
```

Let's assume for a moment that `#{ bean.category }` contains the value, "shoes". if we return `"pretty:viewCategory"` from our action method, where "viewCategory" is the `id` of a URL-mapping in pretty-config.xml, PrettyFaces will extract the current value from the bean `#{ bean.category }` and use that value to generate a new URL:

```
/store/shoes/
```

This means that we can control the generated URLs directly from Java code. Simply set the value of the bean field (the field used in the URL pattern) to control the outcome:

```
public String loadItem() {
 ...
 // Add a message here, "The item {..} could not be found."
 this.setCategory("shoes");
 return "pretty:viewCategory";
}
```

Tip

It is generally good practice to dedicate a separate bean to store page-parameters, this way, the same bean can be used throughout the application when setting values for PrettyFaces navigation.

5.2. * Integrating with JSF commandLink and commandButton

It is also possible to navigate without needing a bean action method at all; this is done by referencing the URL-mapping ID in the command component directly. Similar to *generating HTML links and URLs*, for example:

Example 5.1. Navigating directly from `<h:commandLink>` or `<h:commandButton>`

```
<url-mapping id="viewItem">
 <pattern value="/store/#{ cat : bean.category }/#{ iid : bean.itemId }/" />
 <query-param name="language"> #{ bean.language } </query-param>
 <view-id value="/faces/shop/item.jsf" />
 <action>#{bean.loadItem}</action>
</url-mapping>
```

Note that if the specified URL-mapping ID requires any parameters, the current values found in the required managed bean locations will be used when navigating.

```
<h:commandLink action="pretty:home"> Go home. </h:commandLink>
```

Navigating directly from a command component is most commonly used for refreshing a page:

```
<h:commandLink action="pretty:"> Refresh this page. </h:commandLink>
```

5.3. Redirecting from non-JSF application Java code

Often there you might find yourself somewhere in a custom Servlet, or in a situation where you do not have access to the Faces `NavigationHandler`. For these situations, PrettyFaces provides the `PrettyURLBuilder`, which can be used to generate the String representation of any URL-mapping; one need only have access to an `HttpServletRequest` object in order to get the current configuration, and the current `HttpServletResponse` in order to issue a redirect.

Example 5.2. Redirecting from a custom Servlet

```
public class CustomRedirector
{
 public void redirect(HttpServletRequest request, HttpServletResponse response,
      String mappingId, Map<String, String[]>... params)
```

```
 {
  PrettyContext context = PrettyContext.getCurrentInstance(request);
  PrettyURLBuilder builder = new PrettyURLBuilder();

  URLMapping mapping = context.getConfig().getMappingById(mappingId);
  String targetURL = builder.build(mapping, params);

  targetURL = response.encodeRedirectURL(targetURL);
  response.sendRedirect(targetURL);
 }
}
```

5.4. * Preserving FacesMessages across redirects

Many users add `FacesMessages` to the `FacesContext` in their action methods to display them on the resulting page. The messages stored this way are saved in the the current request so that they are available during page rendering.

Adding messages to the page this way won't work anymore if your are returning an action outcome with the `pretty:` prefix. In this case the page won't be rendered in the same request but the user will be redirected to a completely new URL. As the messages added to the FacesContext are stored on request level they won't be available in the new request and so will get lost.

Fortunately there is a way to work around this issue. PrettyFaces contains a phase listener called `MultiPageMessagesSupport` that will save the FacesMessages before the redirect occurs and restore them on the next request.

To include the `MultiPageMessagesSupport` phase listener in your application, just add the following lines to your `faces-config.xml`.

```
<lifecycle>
  <phase-listener>com.ocpsoft.pretty.faces.event.MultiPageMessagesSupport</phase-listener>
</lifecycle>
```

Note

Please refer to *this article* [http://ocpsoft.com/java/persist-and-pass-facesmessages-over-page-redirects/] for a detailed description of the way the `MultiPageMessagesSupport` uses to solve the redirect problem.

Inbound URL Rewriting

PrettyFaces inbound URL rewriting provides seamless URL rewriting to all Servlets within the Context. This is the capability of intercepting and changing the location of the client's browser URL, modifying or replacing that URL entirely, and displaying a resource.

6.1. Common rewriting features

There are several commonly used rewriting features that PrettyFaces provides for you: manipulating trailing slashes, upper and lower-casing, custom regex search and replace, and redirect types such as `'301 - Permanent'` and `'302 - Temporary'`.

```
<rewrite trailingSlash="append" toCase="lowercase" redirect="301"/>
<rewrite match="/foo" substitute="/bar" redirect="301"/>
```

The two rules above will cause ALL inbound and outbound URLs be appended with a trailing slash (if necessary,) but because of the "match" attribute, only URLs containing `'/foo'` will be replaced with `'/bar/`. Inbound URL-rewriting changes the browser URL unless the redirect attribute is set to “chain”:

```
<rewrite toCase="lowercase" redirect="chain" />
```

6.2. Rewriting URLs that match a specific pattern

Each `<rewrite />` rule may specify a `match="..."` attribute. This attribute defines which URLs will and will not be transformed by a given rewrite rule.

```
<rewrite match="/foo" trailingSlash="append" toCase="lowercase" />
```

Once a URL has been matched, regex groups may be used to perform value-substitution on the URL; this effectively takes parts of the original URL, and makes them availible in its replacement.

```
<rewrite match="/foo/(\w+)/" substitute="/bar/$1/" />
```

This is equivalent to calling the String method `url.replaceAll("/foo/(\w+)/", "/bar/$1/");` Note, also, that the same type of substitution is available when issuing an external redirect from a rewrite-rule:

```
<rewrite match="/foo/(\w+)/" url="http://example.com/$1/" />
```

Click *here* [http://ocpsoft.com/opensource/guide-to-regular-expressions-in-java-part-1/] for more detailed information on regular expressions in Java.

6.3. Rewrite options

The table below outlines each of the individual rewrite-rule options:

Option	Allowed values	Usage
inbound	true/false	(Default: true) Enable or disable inbound URL rewriting for this rule. Setting this value to false means that this rule will be ignored on incoming requests.
match	a regex	(Optional) Describes, via a regular expression pattern, when this 'rewrite' rule should trigger on an inbound or outbound URL. If empty, this rule will match all URLs.
outbound	true/false	(Default: true) Enable or disable outbound URL rewriting for this rule. If enabled, any matching links encoded using HttpServletResponse.encodeURL() will be rewritten according to the rules specified.
processor	qualified class name	(Optional.) Specify a custom processor class to perform more complex, custom URL-rewriting. This class must implement the interface: 'com.ocpsoft.pretty.faces.rewrite.Processor'
redirect	301, 302, chain	(Default: 301) Specifies which type of redirect should be issued when this rule triggers. If 'chain' is specified, a Servlet forward will be issued to the new URL instead of a redirect.
substitute	lifecycle	(Optional.) The regular expression substitution value of the "match" attribute. This effectively enables a "search and replace" functionality. Regular expression back-

```
public interface Processor {
   String processInbound(HttpServletRequest request,
  HttpServletResponse response, RewriteRule rule,
String url);
  String processOutbound(HttpServletRequest request,
  HttpServletResponse response, RewriteRule rule,
String url);
}
```

Option	Allowed values	Usage
		references to the match="..." attribute are supported in the URL, so using '$' and '/' characters may change the value of the result. See *Rewriting URLs that match a specific pattern*, for more details.
toCase	uppercase, lowercase, ignore	(Default: ignore) Change the entire URL (excluding context-path and query- parameters) to 'UPPERCASE' or 'lowercase'.
trailingSlash	append, remove, ignore	(Default: ignore) Control whether trailing slashes on a URL should be appended if missing, or removed if present.
url	a well-formed URL	(Optional.) Specify an well-formed URL to replace the current URL. This will overwrite the context-path and query-parameters. This attribute should usually be combined with redirect="301" (default), which is recommended to prevent adverse SEO effects, loss of page- rank.) Note: You must provide a fully qualified URL, including scheme (such as 'http://', 'ftp://', 'mailto:'). Regular expression back-references to the match="..." attribute are supported in the URL, so using '$' and '/' characters may change the value of the result. See *Rewriting URLs that match a specific pattern*, for more details.

Tip

Rewrite rules must be defined in `pretty-config.xml`.

Outbound URL-rewriting

Outbound URL-rewriting in provides natural integration with most existing URL components (including all of those from the JavaServer Faces framework.) When PrettyFaces is installed, any URL passed into `HttpServletRequest.encodeRedirectURL(String url)` will be processed by PrettyFaces outbound URL-rewriting.

7.1. Rewriting URLs in Java

Given the following URL-mapping, we can render a pretty URL simply by invoking `HttpServletResponse.encodeRedirectURL(String url)`:

```
<url-mapping id="viewCategory">
 <pattern value="/store/#{ cat : bean.category }/" />
 <view-id value="/faces/shop/store.jsf" />
</url-mapping>
```

For example:

```
HttpServletResponse response = getHttpServletResponse();
String rewrittenURL = response.encodeRedirectURL("/faces/shop/store.jsf?
cat=shoes&lang=en_US")';
```

Or if using JSF:

```
String rewrittenURL = FacesContext.getCurrentInstance().getExternalContext()
  .encodeResourceURL("/faces/shop/store.jsf?cat=shoes&lang=en_US");
```

Will produce the following output URL:

```
/store/shoes/?lang=en_US
```

Notice that even though we did not define a *managed query-parameter*, the resulting URL still contains the 'lang' parameter. This is because PrettyFaces only rewrites the *named path-parameters* defined in the URL-pattern; all other query-parameters are simply left unchanged.

7.2. * Rewriting URLs in JSF views

Given the following URL-mapping, some of our JSF URLs will automatically be rewritten:

```
<url-mapping id="viewCategory">
 <pattern value="/store/#{ cat : bean.category }/" />
 <view-id value="/faces/shop/store.jsf" />
</url-mapping>
```

For example:

```
<h:link outcome="/faces/shop/store.jsf" value="View category: Shoes>
 <f:param name="cat" value="shoes" />
 <f:param name="lang" value="en_US" />
</h:link>
```

And:

```
<h:link outcome="pretty:viewCategory" value="View category: Shoes>
 <f:param name="cat" value="shoes" />
 <f:param name="lang" value="en_US" />
</h:link>
```

Will both produce the same output URL:

```
/store/shoes/?lang=en_US
```

Notice that even though we did not define a *managed query-parameter*, the resulting URL still contains the 'lang' parameter. This is because PrettyFaces only rewrites the *named path-parameters* defined in the URL-pattern; all other query-parameters are simply left unchanged.

Tip

Notice that all `<f:param name="" value="">` elements contain both a name and a value attribute. These are required, since (unlike the `<pretty:link>` component,) `<h:link>` does not accept un-named parameters, even when passing a `"pretty:mappingId"` as the outcome.

This is due to the fact that PrettyFaces integration occurs **after** the original URL has already been rendered by the `<h:link>` component, intercepting the URL at the *externalContext.encodeRedirectURL* step, explained above (in the section "*Rewriting URLs in Java*".)

7.3. Disabling outbound-rewriting for a URL-mapping

Outbound URL-rewriting may be disabled on an individual mapping basis, by using the attribute `outbound="false"`.

```
<url-mapping id="viewCategory" outbound="false">
 ...
</url-mapping>
```

* Rendering HTML links and URLs

PrettyFaces provides several methods of generating HTML links via a set of components, and when operating in a JSF 2.0 environment, standard JSF 'h:link' components may be used instead. If the provided mappingId requires any url-pattern-parameters or managed-query-parameters, they can be passed in via the <f:param> tag.

URL pattern parameters can be passed individually, as a `java.util.List`, or as an Array. In the latter two cases, toString() will be called on each of the objects in the list/array. If an empty or null list/array is passed, it will be ignored.

URL path-parameters do NOT have a name attribute, and are parsed in the order they are passed into the tag. Managed query-parameters DO have a name attribute, and order is irrelevant.

8.1. The <pretty:link> component

PrettyFaces provides a JSF component to output an HTML link to the page. The link tag requires a mapping-id (specified in the pretty-config.xml,) identifying which link to render.

Example 8.1. Using the link component

```
<url-mapping id="viewItem">
 <pattern value="/store/#{ cat : bean.category }/#{ iid : bean.itemId }/" />
 <view-id value="/faces/shop/item.jsf"/>
 <action>#{bean.loadItem}</action>
</url-mapping>
```

```
<%@ taglib prefix="pretty" uri="http://ocpsoft.com/prettyfaces" %>

<pretty:link mappingId="viewItem">
 <f:param value="#{item.category}" />
 <f:param value="#{item.id}" />
 View Item. (This is Link Text)
</pretty:link>
```

Output, assuming that `#{item.category} == shoes`, and `#{item.id} == 24`

```
 /store/shoes/24
```

8.2. The <pretty:urlbuffer> component

PrettyFaces provides a JSF component to generate a URL for use as a page scoped variable through El. This tag requires a mapping-id (specified in the pretty-config.xml)

Example 8.2. Using the URL buffer component

```
<url-mapping id="viewItem">
 <pattern value="/store/#{ cat : bean.category }/#{ iid : bean.itemId }/" />
 <query-param name="language"> #{ bean.language } </query-param>
 <view-id value="/faces/shop/item.jsf"/>
 <action>#{bean.loadItem}</action>
</url-mapping>
```

```
<%@ taglib prefix="pretty" uri="http://ocpsoft.com/prettyfaces" %>

<pretty:urlbuffer var="itemListURL" mappingId="viewItem">
 <f:param value="shoes" />
 <f:param value="24" />
 <f:param name="language" value="en_US" />
</pretty:urlbuffer>
<h:outputText value="Generated URL Is: #{requestScope.itemListURL}" />
```

Output:

```
 /store/shoes/24?language=en_US
```

8.3. Using JSF standard components

Caution

Mappings using *DynaView* functionality will not function with JSF link components.

Because PrettyFaces provides *out-bound URL-rewriting*, one can actually use standard JSF components such as `<h:outputLink>` in JSF 1.x, or `<h:link>` in JSF 2.x.

Example 8.3. Using the `<h:outputLink>` in JSF 1.x

```
<url-mapping id="viewItem">
 <pattern value="/store/#{ cat : bean.category }/#{ iid : bean.itemId }/" />
 <query-param name="language"> #{ bean.language } </query-param>
 <view-id value="/faces/shop/item.jsf"/>
 <action>#{bean.loadItem}</action>
</url-mapping>
```

```
<%@ taglib prefix="h" uri="http://java.sun.com/jsf/html" %>

<h:outputLink value="/faces/shop/item.jsf">
 <f:param name="cat" value="shoes" />
 <f:param name="iid" value="24" />
 <f:param name="language" value="en_US" />
</h:outputLink>
```

Output:

```
 /store/shoes/24?language=en_US
```

In JSF 2.x, you can achieve an even greater level of abstraction by using the mapping-ID in combination with `<h:link>`

Example 8.4. Using the `<h:link>` in JSF 2.x

```
<url-mapping id="viewItem">
 <pattern value="/store/#{ cat : bean.category }/#{ iid : bean.itemId }/" />
 <query-param name="language"> #{ bean.language } </query-param>
 <view-id value="/faces/shop/item.jsf"/>
 <action>#{bean.loadItem}</action>
</url-mapping>
```

Both of the following components will generate the same output:

```
<h:link outcome="/faces/shop/item.jsf">
 <f:param name="cat" value="shoes" />
 <f:param name="iid" value="24" />
```

```
 <f:param name="language" value="en_US" />
</h:link>

<h:link outcome="pretty:viewItem">
 <f:param name="cat" value="shoes" />
 <f:param name="iid" value="24" />
 <f:param name="language" value="en_US" />
</h:link>
```

Output:

```
 /store/shoes/24?language=en_US
```

Extend PrettyFaces

As a modern web-framework, it is important to provide extension and integration points, enabling complex functionality that goes beyond the scope of the core framework itself. In order to do this, PrettyFaces provides several extension points, described below:

9.1. com.ocpsoft.pretty.faces.spi.ConfigurationProvider

It may sometimes be necessary to provide custom *configuration* options in PrettyFaces. Loading URL-mappings from a database, for example, or generating mappings based on the state of the file-system. Due to this requirement, PrettyFaces offers an extension point for configuration providers to supply their own additional configuration to be merged with the master configuration at the time when PrettyFaces boots up. This is when you would implement a custom `ConfigurationProvider`.

```
public class MyConfigurationProvider implements ConfigurationProvider {

  public PrettyConfig loadConfiguration(ServletContext servletContext)
  {
    // add new configuration elements here
    return new PrettyConfig();
  }
}
```

To let PrettyFaces know about your provider, you'll have to register your implementation by creating a file named `META-INF/services/com.ocpsoft.pretty.faces.spi.ConfigurationProvider` in your classpath and add the fully-qualified class name of your implementation class there.

```
META-INF/services/com.ocpsoft.pretty.faces.spi.ConfigurationProvider
com.example.MyConfigurationProvider
```

9.2. com.ocpsoft.pretty.faces.spi.ConfigurationPostProcessor

After all configurations have been loaded and merged from any built-in or registered *ConfigurationProvider* sources, there is an additional opportunity to post-process the entire configuration, for instance, if you wanted to programmatically add security constraints to mappings based on various patterns. This is when you would implement a custom `ConfigurationPostProcessor`.

```
public class MyPostProcessor implements ConfigurationPostProcessor {

  public PrettyConfig processConfiguration(ServletContext servletContext, PrettyConfig config)
  {
    // make changes to the configuration here
    return config;
  }
}
```

To let PrettyFaces know about your post-processor, you'll have to register your implementation by creating a file named `META-INF/services/com.ocpsoft.pretty.faces.spi.ConfigurationPostProcessor` in your classpath and add the fully-qualified class name of your implementation class there.

```
META-INF/services/com.ocpsoft.pretty.faces.spi.ConfigurationPostProcessor
com.example.MyPostProcessor
```

9.3. com.ocpsoft.pretty.faces.spi.ELBeanNameResolver

As part of the *Annotations scanning* (one of PrettyFaces' configuration methods,) it may sometimes be necessary to integrate with a custom bean-container that is not one of the built in containers supported natively. It is in these cases when you should implement a custom ELBeanNameResolver. The following example shows a resolver that will resolve the bean name by searching for a `@Named` annotation.

```
public class MyBeanNameResolver implements ELBeanNameResolver {

    public boolean init(ServletContext servletContext, ClassLoader classLoader) {
      // tell PrettyFaces that initialization was successful
      return true;
    }

    public String getBeanName(Class<?> clazz) {

      // try to find @Named annotation
      Named annotation = clazz.getAnnotation(Named.class);

      // return name attribute if annotation has been found
      if(annotation != null) {
```

```
        return annotation.value();
      }

      // we don't know the name
      return null;

    }

  }
```

To let PrettyFaces know about your resolver, you'll have to register your implementation by creating a file named `META-INF/services/com.ocpsoft.pretty.faces.spi.ELBeanNameResolver` in your classpath and add the fully-qualified class name of your implementation class there.

```
META-INF/services/com.ocpsoft.pretty.faces.spi.ELBeanNameResolver
com.example.MyBeanNameResolver
```

9.4. com.ocpsoft.pretty.faces.spi.DevelopmentModeDetector

If PrettyFaces is executed in development mode, the configuration will be automatically reloaded every few seconds. This allows the developer to modify the `pretty-config.xml` and immediately test the changes without restarting the servlet container.

The DevelopmentModeDetector SPI allows the user to implement custom strategies to determine whether the application is in development mode. The following class shows a simple detector which enables the development mode if the system property `devmode` is set to `true`.

```
public class CustomDevelopmentModeDetector implements DevelopmentModeDetector {

  public int getPriority() {
    return 10;
  }

  public Boolean isDevelopmentMode(ServletContext servletContext) {
    return System.getProperty("devmode", "false").equals("true");
  }

}
```

Tip

You can give your own implementation a high precedence of the other implementations by giving it a very low priority.

To let PrettyFaces know about your custom detector, you'll have to register your implementation by creating a file named `META-INF/services/com.ocpsoft.pretty.faces.spi.DevelopmentModeDetector` in your classpath and add the fully-qualified class name of your implementation class there.

```
META-INF/services/com.ocpsoft.pretty.faces.spi.DevelopmentModeDetector
com.example.CustomDevelopmentModeDetector
```

Feedback & Contribution

Contribute ideas to *PrettyFaces* [http://code.google.com/p/prettyfaces/] by submitting a *feature request or bug report* [http://code.google.com/p/prettyfaces/issues/entry]. Join the team; check out the *source code* [http://code.google.com/p/prettyfaces/source/checkout] - submit a patch! Ask a question on the *forums* [http://ocpsoft.com/forums/] or the *mailing list* [http://groups.google.com/group/prettyfaces-users].

Updating Notes

11.1. Updating to 3.x.y from earlier versions

Maven Artifact

PrettyFaces uses new Maven artifact identifiers, changed from older versions. Please use the following artifacts depending on the JSF and Servlet versions you are using.

JSF 2.0 or Servlet 2.5/3.0

```
<dependency>
  <groupId>com.ocpsoft</groupId>
  <artifactId>prettyfaces-jsf2</artifactId>
  <version>3.1.1</version>
</dependency>
```

JSF 1.2 or Servlet 2.5/3.0

```
<dependency>
  <groupId>com.ocpsoft</groupId>
  <artifactId>prettyfaces-jsf12</artifactId>
  <version>3.1.1</version>
</dependency>
```

JSF 1.1

```
<dependency>
  <groupId>com.ocpsoft</groupId>
  <artifactId>prettyfaces-jsf11</artifactId>
  <version>3.1.1</version>
</dependency>
```

FAQ

1. **Q. Can I use PrettyFaces to handle UrlRewriting for other (non-JSF) resources on my server?**

   A. Yes. PrettyFaces still requires a configured JSF instance to function, but it can be used to map a URL to any resource in the Servlet Container – without invoking FacesServlet. Values will be injected into JSF beans as usual, but PrettyFaces Action methods will not trigger (since no JSF lifecycle executes for non-Faces requests.)

   ### Example 12.1. Mapping a non-JSF resource

```
<pretty-config>
 <url-mapping id="login">
  <pattern value="/login" />
  <view-id value="/legacy/user/login.jsp" /> <!-- Non JSF View Id -->
 </url-mapping>
 <url-mapping id="register">
  <pattern value="/register" />
  <view-id value="/faces/user/register.jsf" /> <!-- JSF View Id -->
 </url-mapping>
</pretty-config>
```

2. **Q. Why do my Tomahawk / MyFaces components, or other 3rd party add-ons, break when I use PrettyFaces?**

   A. Since PrettyFaces intercepts mapped HttpRequests then forwards those requests to JSF, it is necessary to enable any additional filters between PrettyFaces and JSF to listen to Servlet Forwards. This is done in the web.xml deployment descriptor by adding the following dispatcher elements to any needed Filters:

   ### Example 12.2. Enabling PrettyFaces for Tomahawk

```
<filter-mapping>
 <filter-name>Tomahawk Filter</filter-name>
 <url-pattern>/*</url-pattern>
 <dispatcher>FORWARD</dispatcher>
 <dispatcher>REQUEST</dispatcher>
 <dispatcher>ERROR</dispatcher>
</filter-mapping>
```

3. **Q. Why, when using MyFaces, am I getting a NullPointerException when I try to use normal faces-navigation?**

   A. Some MyFaces versions do not completely comply with the JSF specification, thus the ViewRoot is null when the request is processed. There is a *patch/workaround* [http://groups.google.com/group/prettyfaces-users/browse_thread/thread/134228ec728edd24], which can be added to fix this issue. You must add this ViewHandler to your faces-config.xml.

4. **Q. Can I configure PrettyFaces via Annotations?**

   A. Yes – please refer to *Annotation based configuration* for details.

5. **Q. How do I enable logging, so that I can tell what the heck is really going on?**

   A. Create or update your log4j.properties file with the following values:

Example 12.3. log4j.properties

```
### direct log messages to stdout ###
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.Target=System.out
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{ABSOLUTE} %5p %c{1}:%L - %m%n

log4j.rootLogger=warn, stdout

### Log for OcpSoft
log4j.logger.com.ocpsoft=debug
```

6. **Q. Can I map and process URLs that span a dynamic number of '/' characters?**

   A. Yes, please read about *custom path-parameter patterns*.

7. **Q. How do I save FacesMessage objects after performing a redirect or pretty:redirect?**

   A. You need to configure the optional MultiPageMessagesSupport PhaseListener (or something like it.) *JBoss Seam* [http://seamframework.org/Seam3/FacesModule] provides a Messaging API that goes above and beyond JSF, providing this feature automatically.

   See *Preserving FacesMessages across redirects* or *this article* [http://ocpsoft.com/java/persist-and-pass-facesmessages-over-page-redirects/] for a full explanation of how this works.

8. **Q. Does PrettyFaces work on IBM's WebSphere?**

   A. Yes, but websphere requires a custom setting in order to behave like a sane server.

Example 12.4. WebSphere Admin-console Configuration Screen

9. **Q. Why are non-ASCII characters distorted when using mod_jk?**

   A. This can happen because mod_jk partially reencodes the request URL after processing them inside Apache httpd and before forwarding it to Tomcat. You should set the following option to let mod_jk forward the unparsed URL.

Example 12.5. httpd.conf

```
JkOptions +ForwardURICompatUnparsed
```