OCPSoft PrettyFaces Reference Guide

v2.0.4

--

Please submit questions or feedback to the PrettyFaces Users group:
http://groups.google.com/group/prettyfaces-users

--

Lincoln Baxter, III
Scott Carnett
Dominik Dorn
Bram Van Dam
Derek Hollis
Stefan Tomm
Aleksei Valikov

Table of Contents

1 User's Guide: OcpSoft PrettyFaces....3
1.1 Getting Started....3
1.1.1 Get PrettyFaces....3
1.1.2 Configure PrettyFaces in WEB-INF/web.xml....3
1.1.3 Create /WEB-INF/pretty-config.xml....3
1.1.3.1 <pretty-config> 4
1.1.3.2 URL Mapping: <url-mapping>....4
1.1.3.2.1 <pattern value="/store/#{cat:categoryBean.catId}/" />....5
1.1.3.2.2 <query-param name="locale">#{userPrefs.locale}</query-param>....5
1.1.3.2.3 <view-id> /faces/shop/item.jsf <view-id>....6
1.1.3.2.4 <action>#{storeBean.loadItem}</action>*....6
1.1.3.3 URL Rewriting: <rewrite/>....7
1.1.4 Order of processing....9
1.1.5 Configuration Loading....9
1.2 Advanced Features....10
1.2.1 Using Dynamic View Ids*....10
1.2.2 Using Managed Query Parameters....10
1.2.3 Validating URL Parameters*....11
1.2.4 Wiring navigation into JSF bean methods*....12
1.2.5 Parsing complex / dynamic-length URLs....13
1.2.6 Accessing PrettyContext through EL....14
1.2.7 Inbound URL rewriting....14
1.2.8 Outbound URL rewriting....14
1.2.9 Rendering HTML Links and URLs....15
1.2.9.1 <pretty:link>*....15
1.2.9.2 <pretty:urlbuffer>*....16
1.2.10 Configuring Logging (log4j)....16

* = this functionality is only provided when the request is handled by the JavaServer Faces Servlet.

1 User's Guide: OcpSoft PrettyFaces

Setting up PrettyFaces is simple.

1.1 Getting Started

This version of the documentation is for PrettyFaces 2.0.4, but you can view older versions of the PrettyFaces docs (1.1.0, 1.2.x, and 2.0.2_GA)

1.1.1 Get PrettyFaces

This step is pretty straight-forward, right? Copy necessary JAR files into your /WEB-INF/lib directory, or include a maven dependency in your pom.xml

```
<!-- For JSF 2.0 -->
<dependency>
     <groupId>com.ocpsoft</groupId>
     <artifactId>ocpsoft-pretty-faces</artifactId>
     <version>2.0.4-SNAPSHOT</version>
     <scope>compile</scope>
</dependency>
```

1.1.2 Configure PrettyFaces in WEB-INF/web.xml

PrettyFilter does most of the work; without it, not much would happen.

```
<!-- This is not required when using a Servlet 3.0 compliant container such as JBoss 6
or GlassFish V3 -->
<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>
```

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

This is where you'll tell PrettyFaces what to do, which URLs to rewrite. Read on for details.

```
<pretty-config xmlns="http://ocpsoft.com/prettyfaces/2.0.4"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://ocpsoft.com/prettyfaces/2.0.4
     http://ocpsoft.com/xml/ns/prettyfaces/ocpsoft-pretty-faces-2.0.4.xsd">

     <!-- Begin RewriteRules -->
     <rewrite trailingSlash="append" />
     <rewrite toCase="lowercase" />
     <rewrite match="^/good_news/(\w+)/$" url="http://ocpsoft.com/$1/"
```

```
                redirect="301" outbound="false" />

    <!-- Begin UrlMappings -->
    <url-mapping id="store">
          <pattern value="/store/" />
          <view-id>/faces/shop/store.jsf</view-id>
    </url-mapping>
    <url-mapping id="viewCategory">
          <pattern value="/store/#{cat:categoryBean.catId}/" />
          <view-id>/faces/shop/store.jsf</view-id>
    </url-mapping>
    <url-mapping id="viewItem">
          <pattern value="/store/#{cat:categoryBean.catId}/#{pid:itemBean.itemId}" />
          <view-id>/faces/shop/item.jsf</view-id>
          <action> #{storeBean.loadItem} </action>
    </url-mapping>

</pretty-config>
```

STOP! That's all you should have to do in order to use PrettyFaces – Read on for detailed configuration options.

1.1.3.1 <pretty-config>

This is the required root element of any PrettyFaces configuration file, and is where you should specify the XSD:

```
<pretty-config xmlns="http://ocpsoft.com/prettyfaces/2.0.4"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://ocpsoft.com/prettyfaces/2.0.4
    http://ocpsoft.com/xml/ns/prettyfaces/ocpsoft-pretty-faces-2.0.4.xsd">
```

1.1.3.2 URL Mapping: <url-mapping>

The heart of PrettyFaces, this is where application integration occurs. Mappings define when/how requests will be intercepted and forwarded within the container, and also when/how outbound URLs will be rewritten.

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

Attributes:

- **id** – (required) This is the “pretty:mapping-ID” – the unique identifier by which this url-mapping will be referenced in the application (e.g.: link components, navigation cases, etc...)

  Note: “pretty:store”, or other mapping-IDs like it, can be used anywhere in place of JSF navigation strings.

- **outbound** – Enable or disable outbound URL rewriting for this mapping (default: 'true' /

enabled.) If enabled, any matching links encoded using HttpServletResponse.encodeURL() will be rewritten (if possible) using parameters mapping to named path parameters specified in the pattern.

Eg, given the following mapping:

```
<url-mapping id="viewItem">
     <pattern value="/store/#{cat:categoryBean.catId}/#{pid:itemBean.itemId}" />
     <view-id>/faces/shop/item.jsf</view-id>
</url-mapping>
```

The following outbound rewrite will occur when an <h:link> component with the necessary <f:param> values is rendered on the page: “/faces/shop/item.jsf?cat=foods&pid=1234” → “/store/foods/1234”

1.1.3.2.1 <pattern value="/store/#{cat:categoryBean.catId}/" />

(Required.) Specifies the pattern for which this URL will be matched.

Any value expressions #{someBean.paramName} found within the pattern will be processed as value injections. The URL will be parsed and the value found at the location of the expression will be injected into the location specified. Note: Expressions will not match over the ‘/’ character.

Additionally, each expression may specify a name, or only a name: #{myParam:someBean.paramName} or #{myParam} – this provides parameter access to non-JSF applications by adding the extracted path value to the HttpServletRequest property Map.

In order to take advantage of outbound-URL rewriting, the parameter name specified must match the parameter name used internally in the application.

For example, if no value injection is required:

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

The values found at **#{cat}** and **#{pid}** will be available through the ServletRequest object, or through JSF2 view metadata:

```
String cat = (String)request.getParameter(“cat”);
String pid = (String)request.getParameter(“pid”);
```

1.1.3.2.2 <query-param name=”locale”>#{userPrefs.locale}</query-param>

(Optional, may specify more than one.) Defines a managed query parameter of the form http://site.com/url?**key=somevalue**, where if the parameter exists, the value will be injected into the specified managed bean. *This also handles JSF commandLink and AJAX <f:param> values.*

Attributes:

- ***name – (required) The name of the request parameter.***
- ***decode*** *– (default: true) URLDecode this parameter (see: java.net.URLDecoder)*

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

1.1.3.2.3 <view-id> /faces/shop/item.jsf <view-id>

(Required.) Specify the URI displayed by this mapping, by either supplying a dynamic view EL Method (must return an object for which the toString() method will return the view Id) or by returning a literal String value.

The view-ID may be any resource located within the current Servlet Context: *E.g. PrettyFaces can also forward to a* ***non****-Faces servlet.*

```
<url-mapping id="store">
     <pattern value="/store/" />
     <view-id>/legacy/StoreServlet</view-id>
</url-mapping>
```

```
<url-mapping id="store">
     <pattern value="/store/" />
     <view-id> #{storeBean.getViewId} </view-id>
</url-mapping>
```

1.1.3.2.4 <action>#{storeBean.loadItem}</action>*

(Optional, more than one may be specified.) Specify a managed-bean action method to be called after URL parameters have been parsed and injected.

Attributes:

- **phaseId** – (Optional, default: RESTORE_VIEW) if set to a valid JSF PhaseId, the action will occur immediately before the specified Phase (or immediately after RESTORE_VIEW). (see javax.faces.event.PhaseId)

  Valid values for this attribute are: RESTORE_VIEW, APPLY_REQUEST_VALUES, PROCESS_VALIDATIONS, UPDATE_MODEL_VALUES, INVOKE_APPLICATION, RENDER_RESPONSE, ANY_PHASE.

  Note, however: if the phase does not occur, neither will your action method.

  If ANY_PHASE is specified, the action method will fire on EVERY phase.

- **onPostback** – (Optional, default: true) If set to false, this action method will not occur on form

postback. (see ResponseStateManager.isPostback())

```
<url-mapping id="viewItem">
      <pattern value="/store/#{cat:categoryBean.catId}/#{pid:itemBean.itemId}" />
      <view-id>/faces/shop/item.jsf</view-id>
      <action> #{storeBean.loadItem} </action>
</url-mapping>
```

1.1.3.3 URL Rewriting: <rewrite/>

Rewrite engine rules are used to provide completely custom inbound and outbound URL rewriting. Rules are evaluated until a redirect is issued, or the end of the chain is reached.

Attributes:

- **inbound** – (Default: true) Enable or disable inbound URL rewriting for this rule . Inbound URL rewriting intercepts incoming requests. Setting this value to false means that this rule will be ignored on incoming requests.
- **match** – (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.

  For example, the pattern below will match: “/good_news/food/” but not “/good_news/food/charity/”

```
<rewrite match="^/good_news/(\w+)/$" ... />
```

- **outbound** – (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.

  For examples, <h:link> and <h:form> component URLs generated by a framework like JavaServer Faces are encoded using this method, and will be rewritten if an appropriate rewrite rule with outbound="true" is triggered.
- **processor** – (Optional.) Specify a custom processor class to perform custom URL rewriting. This class must implement the interface: 'com.ocpsoft.pretty.rewrite.Processor'

  For example:

```
import javax.annotation.processing.Processor;
import com.ocpsoft.pretty.config.rewrite.RewriteRule;

public class CustomClassProcessor implements Processor
{
    public static final String RESULT = "I PROCESSED!";

    public String process(final RewriteRule rewrite, final String url)
    {
        return RESULT;
    }
}
```

- **redirect** – (Default: 301) Specifies which type of redirect should be issued when this rule triggers.
  - '**301**' or '**permanent**' (default): will issue a redirect notifying of a permanent address change; search engines index through 301 redirects to maintain existing search rank, even when a page is moved. The new address is considered to be the new location of the resource.
  - '**302**' or '**temporary**': will issue a temporary redirect; search engines do not index through 302 redirects. The new URL is considered to be an alternate/duplicate resource - simply a different page the server wants the browser to see.
  - '**chain**': will redirect internally after all other chaining rules have triggered.

  301 and 302 redirects are issued immediately upon a rule triggering. Chaining is issued via an internal Servlet forward once all chaining rules have executed on the request.
- **substitute** – (Optional.) The regular expression substitution value of the "match" attribute. This effectively enables a "search and replace" functionality.

```
<rewrite match="^/foo/(.*)$" substitute="/bar/$1" />
```

  Will match the following URL: '/context-path/foo/subst', and will substitute: '/context-path/bar/subst' in its place.
- **toCase** – (Default: ignore) Change the entire URL (excluding context-path and query-parameters) to 'UPPERCASE' or 'lowercase'.
  - **ignore –** ../context-path/Ignore/Example → ../context-path/Ignore/Example
  - **uppercase –** ../context-path/Uppercase/Example → ../context-path/UPPERCASE/EXAMPLE
  - **lowercase –** ../context-path/Lowercase/Example → ../context-path/lowercase/example
- **trailingSlash** – (Default: ignore) Control whether trailing slashes on a URL should be appended if missing, or removed if present.
  - **ignore**:
    - ../context-path/ignore/slash/ --> ../context-path/remove/slash/
    - ../context-path/ignore/slash --> ../context-path/remove/slash
  - **append**:
    - ../context-path/append/slash --> ../context-path/append/slash/
  - **remove**:
    - ../context-path/remove/slash/ --> ../context-path/remove/slash/
- **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.)

```
<rewrite match="^/other-url-rewriting-tools" url="http://www.ocpsoft.com/prettyfaces/"
redirect="301" />
```

Note: You must provide a fully qualified URL, including scheme (such as 'http://", 'ftp://', 'mailto:')

Regular expression backreferences to the match="..." attribute are supported in the URL, so $ and / may change the value of the result.

```
<rewrite match="(?i)^.*to-external/(\w+)$" redirect="302"
          url="http://ocpsoft.com/$1/" />
```

1.1.4 Order of processing

1. Inbound URL rewriting
2. URL pattern parsing, path-parameter and query-parameter extraction (request map is populated with mapped values.)
3. Bean value injection* (After RESTORE_VIEW)
4. View-ID calculation* (if a view Id is dynamic, the el method will be called.)
5. Request forwarded to view-ID via RequestDispatcher.forward("/context/faces/viewId.jsf").
6. Action methods* are called after RESTORE_VIEW phase, unless the optional phaseId attribute is specified.
7. Outbound URL rewriting (processed whenever HttpServletResponse.encodeURL() is called.)

1.1.5 Configuration Loading

At application startup time – before any requests are processed – PrettyFaces processes all provided configuration resources, located according to the following algorithm:

1. Search for classpath resources named **META-INF/pretty-config.xml** in the ServletContext resource paths for this web application (including JAR files,) and load each as a configuration resource.

2. Check for the existence of a context initialization parameter named **com.ocpsoft.pretty.CONFIG_FILES** (usually specified in web.xml). 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.

```
     <context-param>
          <param-name>com.ocpsoft.pretty.CONFIG_FILES</param-name>
          <param-value>/WEB-INF/store-pretty-config.xml, /WEB-INF/shipping-pretty-
config.xml</param-value>
     </context-param>
```

3. Check for the existence of a web application configuration resource named **/WEB-INF/pretty-config.xml**, and load it if the resource exists.

1.2 Advanced Features

1.2.1 Using Dynamic View Ids*

Dynamic view IDs allow a mapped URL to display content from any resource within the ServletContext. This is prevents doing redirects which would otherwise destroy information stored in the URL, and also provides some extra flexibility in application design.

Note: This functionality is only available if Faces Servlet is registered and mapped in the current context. (Faces Servlet needs to be configured, even if it is not used. This should already be done if using a Servlet 3 compliant container such as JBoss 6 or GlassFish v3)

This pretty-config mapping uses a dynamic view-id:

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

The corresponding backing-bean method must return either of:

1. A valid resource path – in this case, PrettyFaces will forward the request to the desired resource in the container.
2. A pretty:mapping-ID – in this case, PrettyFaces will “switch mappings” and instead treat the request as if it had been sent to the new mapping-ID, without issuing a browser redirect.

```
@Named
public class ManagedBean
{
    public String getViewId()
    {
        // This method returns the path of the JSF view to display
        // when the URL /home is accessed.
        if(user.isLoggedIn())
        {
            // Note: the extension '.jsf' is the mapped faces extension
            return "/faces/home.jsf";
        }

        // The home page can instead display a different view; return
        // the pretty:mapping-ID of the view you wish to display.
        // Note that this will not cause a redirect, and will not
        // change the client browser URL.
        // If you wish to issue a redirect, you should use a page
        // load action instead of a dynamic view Id function.
        return "pretty:login";
    }
}
```

1.2.2 Using Managed Query Parameters

Managed query parameters allow automatic assignment of values into JSF managed bean fields, instead of parsing and URL Decoding the value manually out of the request object. Examining this sample mapping, we see that the developer has specified two managed query-parameters. The

'**sortBy**' and '**itemId**' parameters.

```
<url-mapping id="store">
     <pattern value="/store/" />
     <query-param name="sortBy">#{itemBean.sortByField}</query-param>
     <query-param name="itemId">#{itemBean.currentItemId}</query-param>
     <view-id> /faces/shop/store.jsf </view-id>
     <action>#{itemBean.loadItems}</action>
</url-mapping>
```

The managed bean that accompanies this mapping:

```
@Named
public class ItemBean
{
    private List<Item> items;
    private Integer currentItemId;
    private String sortByField;

    public String deleteItem()
    {
        // currentItemId will be automatically populated by
        // PrettyFaces if the parameter was passed in the request
        // (see example JSF page below)
        ItemManager.deleteById(currentItemId);

        // Redisplay the current page via redirect.
        return "pretty:"
    }

    public void loadItems()
    {
        // The sortByField member will be null if the sortBy
        // query-parameter is not found in the request
        this.items = ItemManager.getSortedItems(sortByField);
    }

    //... getters and setters...
}
```

Example JSF page: (*Notice the <f:param> tag.) This will generate a link that provides the 'itemId' parameter to the request for PrettyFaces to parse.*

```
<c:forEach var="item" items="#{itemBean.items}">
     <h:commandLink action="#{itemBean.deleteItem}">Delete this item.
          <f:param name="itemId" value="${item.id}" />
     </h:commandLink>
</c:forEach>
```

1.2.3 Validating URL Parameters*

One of the important factors to consider when dealing with any type of user input, is validation. PrettyFaces offers hooks into JSF validation, allowing validators to be attached to individual parameters in each dynamic URL. If you are not using the JSF2 <f:metadata> and <f:viewParam>

tags to manage parameters, then you may want to consider using PrettyFaces validation. (See example…)

Note: This functionality is only available if the <view-id> specified is that of a JSF resource.

```
<url-mapping id="validate">
     <pattern value="/validate/#{validationBean.pathInput}">
          <validate index="0" validatorIds="validator1 validator2"
onError="#{validationBean.handle}" />
     </pattern>
     <query-param name="param1" validatorIds="validator2" onError="pretty:demo">
          #{validationBean.queryInput}
     </query-param>
     <view-id>/faces/validation/test.jsf</view-id>
</url-mapping>
```

Attributes of <validate> and <query-param>:

- **index** – (Begins at 0) The index of the targeted #{parameter} in the pattern.
- **validatorIds** – A space-separated list of all JSF validators you wish to attach to a given parameter. These validator IDs must have been defined in faces-config.xml or via the JSF2 @FacesValidator annotation.
- **onError** – (Default: show 404 page.) The pretty:mapping-ID to be displayed, or the #{bean.method} to evaluate, should validation fail on this parameter.

1.2.4 Wiring navigation into JSF bean methods*

```
@Named
public class PageBean
{
    public String goHome()
    {
        // this will tell pretty to redirect the client to the home-page
        // no parameters are mapped, so this is pretty simple
        return "pretty:home";
    }

    public String goHomeAndWelcome()
    {
        // this will tell pretty to redirect the client to the home-page
        // since there is a managed query-parameter defined in the mapping,
        // PrettyFaces will generate the URL, and append the mapped param
        // eg: /home?displayWelcome=true
        homeBean.displayWelcomeMessage(true);
        return "pretty:home";
    }

    public String goViewStory()
    {
        // this will tell pretty to redirect the client to the viewStory page
        // PrettyFaces will generate the URL by extracting any values from
        // the mapping beans and using them to inject back into the pattern
        // therefore, navigation can be controlled by placing a value into
        // the mapped field before PrettyFaces extracts it and generates the URL
        // so... /story/#{myBean.currentStoryId}/ ...becomes... /story/12/
```

```
        viewStoryBean.setCurrentStoryId(12);
        return "pretty:viewStory";
    }

    public String doRefreshByRedirect()
    {
        // using the "pretty:" prefix without a mapping-id will cause a
        // redirect to the current page
        return "pretty:";
    }

    public String doNormalJSFRender()
    {
        // returning an value without the "pretty:" prefix will fall back to
        // the default JSF navigation handlers
        return "someNavigationCase";
    }
}
```

1.2.5 Parsing complex / dynamic-length URLs

Consider the following example configuration and bean: The URL contains many dynamic layers of mysite.com/value/value2/value3/…/valueN, which cannot be specified using a static URL pattern.

Since patterns are regexes, we can use a catch-all pattern, and process the values ourselves. Note, however, that <pretty:link>, outbound URL rewriting, and other URL PrettyFaces URL generation tools, will NOT be able to generate dynamic links to URLs mapped in this fashion; you will need to do that yourself as well. (Complex patterns can lead to poor URL generation, and it is recommended to disable outbound URL rewriting for these mappings.)

```
<url-mapping id="dynamicUrl">
     <pattern> /blog/categories/.* </pattern>
     <view-id> #{urlParsingBean.parseComplexUrl} </view-id>
</url-mapping>
```

Example of a complex URL parsers. Remember to URLDecode before using any values from the URL:

```
public class UrlParsingBean
{
    public String parseComplexUrl() throws UnsupportedEncodingException
    {
        String uri = PrettyContext.getCurrentInstance().getOriginalUri();
        List<String> categoryChain = new ArrayList<String>();

        while(uri.length() > 0)
        {
            int index = uri.indexOf('/');
            String value = uri.substring(0, index);
            categoryChain.add(URLDecoder.decode(value, "UTF-8"));
            uri = uri.substring(index);
        }

        //now load the data...
```

```
        return "/blag/viewArticle.jsf";
    }
}
```

1.2.6 Accessing PrettyContext through EL

Since a new PrettyContext is generated on each request, and stored into the requestMap, it is possible to access the context object through EL in a JSP, or Facelet, or other configurations.

```
#{prettyContext} <!-- returns the current PrettyContext -->
#{prettyContext.currentMapping.id} <!-- returns the current UrlMapping Id -->
<!-- And so on... -->
```

1.2.7 Inbound URL rewriting

PrettyFaces inbound URL rewriting provides seamless URL rewriting to all Servlets within the Context. (See URL Rewrite: <rewrite> section above for more detailed information in rewriting capabilities.) 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.

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

The two rules above will cause ALL inbound and outbound URLs be appended with a trailing slash (if necessary,) but only URLs containing '/foo' will be transformed to lower-case. Inbound URL rewriting **changes the browser URL** unless the **redirect** attribute is set to “**chain**.”

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

1.2.8 Outbound URL rewriting

PrettyFaces outbound URL rewriting provides natural integration with most existing URL components (including all of those from the JavaServer Faces framework.)

Given the following mapping in **pretty-config.xml**:

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

We can render a Pretty URL by using a JSF2 <h:link> tag with the pretty:mapping-ID

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

or by using the normal JSF view-ID

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

The result of either:

```
<a id="j_012" href="/store/shoes">View category: Shoes</a>
```

The same result could have been achieved in our Java code by calling:

```
String url =
FacesContext.getCurrentInstance().getExternalContext().encodeResourceURL("/faces/shop/st
ore.jsf");
```

However, it looks like we forgot our trailing '/' character. We can add a <rewrite> rule to fix this for us:

```
<rewrite trailingSlash="append" />
```

Now the rendered URL will include the trailing '/' character:

```
<a id="j_012" href="/store/shoes/">View category: Shoes</a>
```

1.2.9 Rendering HTML Links and URLs

PrettyFaces provides JSF components to output an HTML links/URLs to the page. These components require a pretty:mapping-ID (specified in the pretty-config.xml,) identifying which URL to render.

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 pattern 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.

*1.2.9.1 <pretty:link>**

PrettyFaces provides a JSF component to output an HTML link to the page.

Given the following mapping in **pretty-config.xml**:

```
<url-mapping id="viewItem">
     <pattern value="/store/#{cat:categoryBean.catId}/#{pid:itemBean.itemId}" />
     <view-id>/faces/shop/item.jsf</view-id>
</url-mapping>
```

```
<html xml:lang="en" lang="en" xmlns="http://www.w3.org/1999/xhtml"
      xmlns:pretty="http://ocpsoft.com/prettyfaces"
      xmlns:f="http://java.sun.com/jsf/core">

      <pretty:link mappingId="viewCategory">
            <f:param value="shoes" />
            <f:param value="Z23G23" />
            Go to Comment. (This is Link Text)
      </pretty:link>

      <!-- will render the URL: /store/shoes/Z23G23 -->

</html>
```

*1.2.9.2 <pretty:urlbuffer>**

PrettyFaces provides a JSF component to generate a URL for use as a page scoped variable through EI. This design is intended to reduce complexity and prevent manual manipulation/generation of URLs.

Given the following mapping in **pretty-config.xml**:

```
<url-mapping id="store">
      <pattern value="/store/" />
      <query-param name="sortBy">#{itemBean.sortByField}</query-param>
      <query-param name="itemId">#{itemBean.currentItemId}</query-param>
      <view-id> /faces/shop/store.jsf </view-id>
      <action>#{itemBean.loadItems}</action>
</url-mapping>
```

```
<html xml:lang="en" lang="en" xmlns="http://www.w3.org/1999/xhtml"
      xmlns:pretty="http://ocpsoft.com/prettyfaces"
      xmlns:f="http://java.sun.com/jsf/core">

      <pretty:urlbuffer var="itemUrl" mappingId="store">
            <f:param name="itemId" value="22" />
            <f:param name="sortBy" value="price" />
      </pretty:urlbuffer>

      <h:outputText value="Generated Url Is: #{requestScope.itemUrl}" />
      <!-- /items/list?itemId=22&sortBy=price -->

</html>
```

1.2.10 Configuring Logging (log4j)

Example 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

### set log levels - for more verbose logging change 'info' to 'debug' ###

log4j.rootLogger=warn, stdout

#this will set PrettyFaces logging level to 'info'
log4j.logger.com.ocpsoft.pretty=info
```

Finished! Run your application!

You should now have a fully functional PrettyFaces configuration. Please submit questions or feedback to the PrettyFaces Users group: http://groups.google.com/group/prettyfaces-users

Thank you!

-The PrettyFaces Team