URL rewriting

Tapestry URL Rewriting Support

Starting with Tapestry 5.2, the URLRewriterRule service has been replaced with the new LinkTransformer service. This page needs to be revised to reflect the new API. Meanwhile, please see Igor Drobiazko's excellent blog post on this topic.

Since 5.1.0.1, Tapestry has basic support for URL rewriting. Incoming requests and links generated by Tapestry can be rewritten using exactly the same API, based on a chain of URLRewriterRule interfaces. These rules are executed before all other Tapestry request handling, so your application does not otherwise know that the received request is not the original one.

Each URL rewriter rule, in its `Request process`, can choose between returning another `Request`, effectively rewriting it, or returning the received request unchanged, meaning that this rule does not apply to that request.

To facilitate `Request` creation, Tapestry provides the `SimpleRequestWrapper` class. It wraps a `Request`, delegating all methods except `getPath()` and `getServerName()`. More request wrappers may be added in the future on demand.

Configuration

Tapestry's URL rewriting support is configured by Tapestry-IoC through contribution of a `URLRewriterRule` to the `URLRewriter` service. The following example is part of the Tapestry's tests.

Simple example of rule chaining

This example just rewrites all incoming requests to `/struts` to `/tapestry`. In your `AppModule` or any other Tapestry-IoC module class:

```
public static void contributeURLRewriter(OrderedConfiguration<URLRewriterRule> configuration)
{

    URLRewriterRule rule = new URLRewriterRule()
    {

        public Request process(Request request, URLRewriteContext context)
        {
            final String path = request.getPath();
            if (path.equals("/struts"))
            {
                request = new SimpleRequestWrapper(request, "/tapestry");
            }

            return request;

        }

        public RewriteRuleApplicability applicability()
        {
            return RewriteRuleApplicability.INBOUND;
        }

    };

    configuration.add("myrule", rule);
}
```

Example of rule chaining

In your `AppModule` or any other Tapestry-IoC module class.

```
public static void contributeURLRewriter(OrderedConfiguration<URLRewriterRule> configuration)
{

    URLRewriterRule rule1 = new URLRewriterRule()
    {

        public Request process(Request request, URLRewriteContext context)
        {
            final String path = request.getPath();
```

```
        if (path.equals("/struts"))
        {
            request = new SimpleRequestWrapper(request, "/jsf");
        }

        return request;

    }

    public RewriteRuleApplicability applicability()
    {
        return RewriteRuleApplicability.INBOUND;
    }

};

URLRewriterRule rule2 = new URLRewriterRule()
{

    public Request process(Request request, URLRewriteContext context)
    {
        final String path = request.getPath();
        if (path.equals("/jsf"))
        {
            request = new SimpleRequestWrapper(request, "/tapestry");
        }
        return request;

    }

    public RewriteRuleApplicability applicability()
    {
        return RewriteRuleApplicability.INBOUND;
    }

};

URLRewriterRule rule3 = new URLRewriterRule()
{

    public Request process(Request request, URLRewriteContext context)
    {
        String path = request.getPath();
        if (path.equals("/tapestry"))
        {
            path = "/urlrewritesuccess";
            request = new SimpleRequestWrapper(request, path);
        }
        return request;

    }

    public RewriteRuleApplicability applicability()
    {
        return RewriteRuleApplicability.INBOUND;
    }

};

URLRewriterRule rule4 = new URLRewriterRule()
{

    public Request process(Request request, URLRewriteContext context)
    {
        String serverName = request.getServerName();
        String path = request.getPath();
        final String pathToRewrite = "/urlrewritesuccess/login";
        if (serverName.equals("localhost") && path.equalsIgnoreCase(pathToRewrite))
        {
            request = new SimpleRequestWrapper(request, "http://login.domain.com", "/");
        }
```

```
            return request;

        }

        public RewriteRuleApplicability applicability()
        {
            return RewriteRuleApplicability.OUTBOUND;
        }


    };

    configuration.add("rule1", rule1);
    configuration.add("rule2", rule2, "after:rule1");
    configuration.add("rule3", rule3, "after:rule2");
    configuration.add("rule4", rule4);

}
```

This examples shows the URL rewriting chaining: the first rule rewrites requests to `/struts` and rewrites them to `/jsf` and leaves requests to other URLs unchanged. The second rewrites `/jsf` to `/tapestry` and the third rewrites `/tapestry` to `/urlrewritesuccess`.

The result is that any request to `/struts` end up being handled by the same class that handles `/urlrewritesuccess`, while the browser, the user and Tapestry still sees `/struts`.

Note that this applies to rewriting links generated by Tapestry too: a `PageLink` to the `urlrewritesuccess` page with an activation context of `login` (path `/urlrewritesuccess/login`) will generate an a tag pointing to `http://login.domain.com`.

The URLRewriteContext (added in 5.1.0.4) provides additional information for rewriting, particularly in the context of rewriting generated link urls. In the following example, we'll reconfigure the url used to render pages. Whereas the previous examples used separate rules for handling inbound and outbound rewriting, this demonstration will utilize a single rule for both scenarios. To simplify the example, we will assume that every page is named "XXXPage" (UserPage, TransactionPage, IndexPage, etc.). This naming convention also means that we don't have to worry about tapestry's auto-stripping of "index" from URLs, because our page would be IndexPage, rather than Index.

```
public static void contributeURLRewriter(OrderedConfiguration<URLRewriterRule> configuration)
{
    URLRewriterRule rule = new URLRewriterRule()
    {
        public Request process(Request request, URLRewriteContext context)
        {
            if (context.isIncoming())
            {
                //these look like component event requests, which we didn't rewrite, so ignore.
                if (request.getPath().contains(".") || request.getPath().contains(":"))
                {
                    return request;
                }
                String pageName = request.getPath().substring(1,request.getPath().indexOf('/',1));
                return new SimpleRequestWrapper(request, request.getPath().replaceAll(pageName,pageName +
"page"));
            }
            else
            {
               //if this is a component event, getPageParameters() will return null.
               if (context.getPageParameters() != null)
               {
                    String path = request.getPath();
                    String pageName = context.getPageParameters().getLogicalPageName().toLowerCase();
                    String newPageName = pageName.replaceAll("page$","");
                    return new SimpleRequestWrapper(request,path.replaceAll(pageName,newPageName));
               }
            }
            return request;
        }

        public RewriteRuleApplicability applicability()
        {
            return RewriteRuleApplicability.BOTH;
        }
    };

    configuration.add("rule1",rule);

}
```

In the first part of `process`, `context.isIncoming()` determines if the call to `process` occurred due to an inbound request. If so, the rule reverses the mapping done in the second portion of the method, so tapestry sees the original request.

The second half of `process` rewrites only page links by retrieving the logical page name and replacing its occurrence in the url with the shortened form of the link. This code segment demonstrates how the additional information provided by `URLRewriteContext` can be used to rewrite urls in a generalized manner.

Note that `getPageParameters()` will only return non-null when `process` is called due to page link creation, and `getComponentEventParameters()` will only return non-null when `process` is called as a result of creating component event links.