Flexible Resource Resolution

Flexible Resource Resolution

Status: IMPLEMENTED
Created: 25. November 2008
Author: fmeschbe
JIRA: SLING-249
References: http://markmail.org/message/cibaddpdfk2jwm7p, Mappings for Resource Resolution
Update: 28. November 2008, fmeschbe, Addition to node types and add section on backwards compatibility

Introduction

Currently the resource resolver has a limited set of options to resolve incoming requests to resources and to map resource paths to paths to be used in links in the response:

- **Vanity URLs** – replace requested path with different path
- **URL Mappings** – replace path prefixes with different prefixes
- **Regular Expressions** – regular expressions to resolve and map paths
- **VanityPath** – property set to an absolute path matched against URL

To implement SLING-249 I initially proposed to add another resolution and mapping option:

- **VirtualHost** – add path prefix based on Host: header (my proposal)

After an internal discussion I proposed a modified implementation, which drops the existing configuration in favor of full flexibiliy on all resources, by allowing a `sling:vanityPath` to specify a relative or absolute path or even an URL. Absolute paths and URLs would be used to resolve incoming requests to different locations in the resource tree.

Roy Fielding noted, that allowing any content node to define an absolute entry point for resource resolution would be an open door to bypass delegated security and would also become unmanegeable over time. In his message he proposed a slightly different approach, by specifying a location in the repository, where such root entries would be configured. On the content level, only local aliases are allowed to be defined.

This page is about trying to write up the fine print in this proposal and also document the actual implementation.

Goals

- Separate concerns for root level mappings and local aliases. This allows administrators to define root level mappings preventing regular authors from tampering and allows page authors to define aliases for their pages.
- Allow providing different content trees for different virtual hosts while at the same time allowing to share recources amongst all virtual hosts.
- Provide funtionality to externally and internally redirect. External redirects are implemented by sending a 302/FOUND response with a different `Location` to the client. Internal redirects are handled by just resolving a different actual resource path.
- Allow authors to define alias names for their resources which may be used in URLs.

JCR Environment

Properties

When dealing with the new resource resolution we have a number of properties influencing the process:

- `sling:match` – This property when set on a node in the `/etc/map` tree (see below) defines a partial regular expression which is used instead of the node's name to match the incoming request. This property is only needed if the regular expression includes characters which are not valid JCR name characters. The list of invalid characters for JCR names is: /, :, ,, *, ', ", | and any whitespace except blank space. In addition a name without a name space may not be `.` or `..` and a blank space is only allowed inside the name.
- `sling:redirect` – This property when set on a node in the `/etc/map` tree (see below) causes a redirect response to be sent to the client, which causes the client to send in a new request with the modified location. The value of this property is applied to the actual request and sent back as the value of `Location` response header.

- `sling:status` – This property defines the HTTP status code sent to the client with the `sling:redirect` response. If this property is not set, it defaults to 302 (Found). Other status codes supported are 300 (Multiple Choices), 301 (Moved Permanently), 303 (See Other), and 307 (Temporary Redirect).
- `sling:internalRedirect` – This property when set on a node in the `/etc/map` tree (see below) causes the current path to be modified internally to continue with resource resoltion.
- `sling:alias` – The property may be set on any resource to indicate an alias name for the resource. For example the resource `/content/visitors` may have the `sling:alias` property set to `besucher` allowing the resource to be addressed in an URL as `/content/besucher`.

Node Types

To ease with the definition of redirects and aliases, the following node types are defined:

- `sling:ResourceAlias` – This mixin node type defines the `sling:alias` property and may be attached to any node, which does not otherwise allow setting a property named `sling:alias`
- `sling:MappingSpec` – This mixin node type defines the `sling:match`, `sling:redirect`, `sling:status`, and `sling:internaleRedirect` properties to define a matching and redirection inside the `/etc/map` hierarchy.
- `sling:Mapping` – Primary node type which may be used to easily construct entries in the `/etc/map` tree. The node type extends the `sling:MappingSpec` mixin node type to allow setting the required matching and redirection. In addition the `sling:Resource` mixin node type is extended to allow setting a resource type and the `nt:hierarchyNode` node type is extended to allow locating nodes of this node type below `nt:folder` nodes.

Note, that these node types only help setting the properties. The implementation itself only cares for the properties and their values and not for any of these node types.

Namespace Mangling

There are systems accessing Sling, which have a hard time handling URLs containing colons – `:` – in the path part correctly. Since URLs produced and supported by Sling may colons because JCR Item based resources may be namespaced (e.g. `jcr:content`), a special namespace mangling feature is built into the `ResourceResolver.resolve` and `ResourceResolver(map)` methods.

Namespace mangling operates such, that any namespace prefix identified in resource path to be mapped as an URL in the `map` methods is modfied such that the prefix is enclosed in underscores and the colon removed.

Example: The path `/content/_a_sample/jcr:content/jcr:data.png` is modified by namespace mangling in the `map` method to get at `/content/_a_sample/_jcr_content/_jcr_data.png`.

Conversely the `resolve` methods must undo such namespace mangling to get back at the resource path. This is simple done by modifying any path such that segments starting with an underscore enclosed prefix are changed by removing the underscores and adding a colon after the prefix. There is one catch, tough: Due to the way the SlingPostServlets automatically generates names, there may be cases where the actual name would be matching this mechanism. Therefore only prefixes are modified which are actually namespace prefixes.

Example: The path `/content/_a_sample/_jcr_content/_jcr_data.png` is modified by namespace mangling in the `resolve` method to get `/content/_a_sample/jcr:content/jcr:data.png`. The prefix `_a` is not modified because there is no registered namespace with prefix a. On the other hand the prefix `_jcr` is modified because there is of course a registered namespace with prefix `jcr`.

Root Level Mappings

Root Level Mappings apply to the request at large including the scheme, host.port and uri path. To accomplish this a path is constructed from the request as `{scheme}/{host.port}/{uri_path}`. This string is then matched against mapping entries below `/etc/map` which are structured in the content analogously. The longest matching entry string is used and the replacement, that is the redirection property, is applied.

Mapping Entry Specification

Each entry in the mapping table is a regular expression, which is constructed from the resource path below `/etc/map`. If any resource along the path has a `sling:match` property, the respective value is used in the corresponding segment instead of the resource name. Only resources either having a `sling:redirect` or `sling:internalRedirect` property are used as table entries. Other resources in the tree are just used to build the mapping structure.

Example

Consider the following content

```
/etc/map
      +-- http
           +-- example.com.80
                +-- sling:redirect = "http://www.example.com/"
           +-- www.example.com.80
                +-- sling:internalRedirect = "/example"
           +-- any_example.com.80
                +-- sling:match = ".+\.example\.com\.80"
                +-- sling:redirect = "http://www.example.com/"
           +-- localhost_any
                +-- sling:match = "localhost\.\d*"
                +-- sling:internalRedirect = "/content"
```

```
                +-- cgi-bin
                     +-- sling:internalRedirect = "/scripts"
                +-- gateway
                     +-- sling:internalRedirect = "http://gbiv.com"
                +-- (stories)
                     +-- sling:internalRedirect = "/anecdotes/$1"
```

This would define the following mapping entries:

Regular Expression	Redirect	Internal	Description
http/example.com.80	http://www.example.com	no	Redirect all requests to the Second Level Domain to www
http/www.example.com.80	/example	yes	Prefix the URI paths of the requests sent to this domain with the string `/example`
http/.+\.example\.com\.80	http://www.example.com	no	Redirect all requests to sub domains to www. The actual regular expression for the host.port segment is taken from the `sling:match` property.
http/localhost\.\d*	/content	yes	Prefix the URI paths with `/content` for requests to localhost, regardless of actual port the request was received on. This entry only applies if the URI path does not start with `/cgi-bin`, `gateway` or `stories` because there are longer match entries. The actual regular expression for the host.port segment is taken from the `sling:match` property.
http/localhost\.\d*/cgi-bin	/scripts	yes	Replace the `/cgi-bin` prefix in the URI path with `/scripts` for requests to localhost, regardless of actual port the request was received on.
http/localhost\.\d*/gateway	http://gbiv.com	yes	Replace the `/gateway` prefix in the URI path with {{ http://gbiv.com }} for requests to localhost, regardless of actual port the request was received on.
http/localhost\.\d*/(stories)	/anecdotes/stories	yes	Prepend the URI paths starting with `/stories` with `/anecdotes` for requests to localhost, regardless of actual port the request was received on.

Regular Expression matching

As said above the mapping entries are regular expressions which are matched against path. As such these regular expressions may also contain capturing groups as shown in the example above: `http/localhost\.\d*/(stories)`. After matching the path against the regular expression, the replacement pattern is applied which allows references back to the capturing groups.

To illustrate the matching and replacement is applied according to the following pseudo code:

```
String path = request.getScheme + "/" + request.getServerName() + "." + request.getServerPort() + "/" + request.
getPathInfo();
String result = null;
for (MapEntry entry: mapEntries) {
    Matcher matcher = entry.pattern.matcher(path);
    if (matcher.find()) {
        StringBuffer buf = new StringBuffer();
        matcher.appendReplacement(buf, entry.getRedirect());
        matcher.appendTail(buf);
        result = buf.toString();
        break;
    }
}
```

At the end of the loop, `result` contains the mapped path or `null` if no entry matches the request `path`.

NOTE: Since the entries in the `/etc/map` are also used to reverse map any resource paths to URLs, using regular expressions in the Root Level Mappings prevent the respective entries from being used for reverse mappings. Therefor, it is strongly recommended to not use regular expression matching, unless you have a strong need.

Redirection Values

The result of matching the request path and getting the redirection is either a path into the resource tree or another URL. If the result is an URL, it is converted into a path again and matched against the mapping entries. This may be taking place repeatedly until an absolute or relative path into the resource tree results.

The following pseudo code summarizes this behaviour:

```
String path = ....;
String result = path;
do {
    result = applyMapEntries(result);
} while (isURL(result));
```

As soon as the result of applying the map entries is an absolute or relative path (or no more map entries match), Root Level Mapping terminates and the next step in resource resolution, resource tree access, takes place.

Resource Tree Access

The result of Root Level Mapping is an absolute or relative path to a resource. If the path is relative – e.g. `myproject/docroot/sample.gif` – the resource resolver search path (`ResourceResolver.getSearchPath()` is used to build absolute paths and resolve the resource. In this case the first resource found is used. If the result of Root Level Mapping is an absolute path, the path is used as is.

Accessing the resource tree after applying the Root Level Mappings has four options:

- Check whether the path addresses a so called Star Resource. A Star Resource is a resource whose path ends with or contains `/*`. Such resources are used by the `SlingPostServlet` to create new content below an existing resource. If the path after Root Level Mapping is absolute, it is made absolute by prepending the first search path entry.
- Check whether the path exists in the repository. if the path is absolute, it is tried directly. Otherwise the search path entries are prepended to the path until a resource is found or the search path is exhausted without finding a resource.
- Drill down the resource tree starting from the root, optionally using the search path until a resource is found.
- If no resource can be resolved, a Missing Resource is returned.

Drilling Down the Resource Tree

Drilling down the resource tree starts at the root and for each segement in the path checks whether a child resource of the given name exists or not. If not, a child resource is looked up, which has a `sling:alias` property whose value matches the given name. If neither exists, the search is terminated and the resource cannot be resolved.

The following pseudo code shows this algorithm assuming the path is absolute:

```
String path = ...; // the absolute path
Resource current = getResource("/");
String[] segements = path.split("/");
for (String segment: segments) {
    Resource child = getResource(current, segement);
    if (child == null) {
        Iterator<Resource> children = listChildren(current);
        current = null;
        while (children.hasNext()) {
            child = children.next();
            if (segment.equals(getSlingAlias(child))) {
                current = child;
                break;
            }
        }
        if (current == null) {
            // fail
            break;
        }
    } else {
        current = child;
    }
}
```

Current Status

In Revision 720647 of Sling trunk I have implemented a first shot at a new JCR resource resolver. This resource resolver currently lives side-by-side with the old one and the JcrResourceResolverFactoryImpl decides based on configuration (Configuration Admin configuration or framework property of the name `resource.resolver.new`) whether the new or the old resource resolver is to be used. The default is to use the new resource resolver.

The new resource resolver currently has the following setup:

- **`/etc/map`** – Mappings are read from `/etc/map` as described above

- **Vanity URLs** – Existing vanity URL configuration is added as map entries, where the regular expression{{**`/.`**}} as the prefix to indicate scheme and host.port are ignored.
- **URL Mappings** – Existing URL mapping configuration added as map entries where multiple internal path mappings for the same external path are collected into a single entry with multiple internal redirects. Again the scheme and host.port are ignored that is the entries are prefixed with `./.` .
- **Regular Expressions** – Regular expression mappings are not supported like this anymore. These must be migrated manually to respective entries in `/etc/map`. The main reason to have regular expression mapping was support for namespace mangling (see above), which has been implemented differently (see below).
- **VanityPath** – Existing `sling:vanityPath` settings are loaded as map entries where the `sling:vanityPath` property defines the regular expression (using the fixed string `./.` to indicate that the entry applies for any scheme and any host.port. The path of the node having the `sling:vanityPath` property is used as the redirect path. Finally the `sling:redirect` property of the node is used to decided whether the redirect is internal (property is `false` or missing) or external (property is set to `true`).
- **Namespace Mangling** – Namespace mangling as described in the *Namespace Mangling* section above is implemented.