Sling API Redesign

Redesign of the Sling API

Status: IMPLEMENTED

There have been a number of threads on the Sling Dev Mailing List on simplifying the current *Component API* and turn it into a new Sling API. This page starts at the current state of the discussion as of Oct. 10, 2007, and tries to summarize what has been discussed and to resolve this into a modified proposal.

References

- SLING-28, Simplify the Sling (aka Component) API
- SLING-47, microsling, "Sling reduced to the max"
- Simplifying our component api - The original thread launched by Carsten
- Move ContentManager to Sling API - My own proposal to make the ContentManager part of the Sling API
- Breaking Sling into smaller pieces? - Bertrand's proposal to further modularize parts of Sling such as the current *sling-core* bundle

Current State

Currently, request processing is controlled by the *sling-core* bundle using two sets of filters: one set called at the time the client request enters Sling - so called request level filters - and the other set called for each Content object processed during request processing - so called content level filters.

Amongst the request level filters is the `ContentResolverFilter` which takes the request URL and finds a `Content` object for the URL. This filter implements the `ContentResolver` interface and is also registered as this service. So other parts of the system may use the same mechanism to resolve paths to `Content` objects. The `ContentResolver` also implements the default content loading described here.

Amongst the content level filters is the `ComponentResolverFilter` which asks the `Content` object for its component ID and resolves this ID using the registered {{Component}}s. This filter also implements the default component resolution described here.

To manage content Sling provides two interfaces:

- **`ContentManager`** - Basic interface allowing CRUD operations using `Content` objects. This interface is completely agnostic of the actual persistence used.
- **`JcrContentManager`** - Extends the `ContentManager` interface integrating with the Jackrabbit OCM `ObjectContentManager` interface. This provides the API actually used by the `ContentResolverFilter` to load `Content` objects from the JCR repository according to the request URL.

If components would want to create, update and delete content, they would access the `ContentManager` by retrieving the `org.apache.sling.jcr.content_manager` request attribute. If JCR tasks would have to be executed, that retrieved object would be cast to `JcrContentManager` and the session retrieved.

Examples:

Update Modified `Content`

After having modified the content, a component might do the following to persisted the modified content:

```
Content content = componentRequest.getContent();

// modify content

ContentManager contentManager = (ContentManager) componentRequest.getAttribute("org.apache.sling.jcr.
content_manager");
contentManager.store(content);
contentManager.save();
```

JCR based Operations

To operate on a JCR level or to directly access the JCR `Node` underlying the request `Content` the following might be done:

```
// get the JCR content manager
JcrContentManager jcrContentManager = (JcrContentManager) componentRequest.getAttribute("org.apache.sling.jcr.
content_manager");

// get the session
Session session = jcrContentManager.getSession();

// access the node addressed by the request URL
String contentPath = componentRequest.getContent().getPath();
Node contentNode = (Node) session.getItem(contentPath);
```

Arguably, this is tedious. So a first simplification proposal suggested to move the JCR agnostic `ContentManager` interface to the Sling API and to provide a getter method on the `ComponentRequest` interface. The returned object might also be cast to a `JcrContentManager` to then access the repository.

This proposal sparked a series of reactions (see references above) and so based on Bertrands thoughts, we propose the following change.

Replace `Content` by `Resource`

The "problem" of the current Component API is that is centered around a `Content` interface which presumably is data provided to the component loaded from the persistence (the JCR repository of course) actually hiding the repository. This also predefines how data is acquired and used, namely by using Object Content Mapping.

Starting off this situation, we propose replacing the (fully loaded) `Content` by a data representation we will call `Resource`:

```
public interface Resource {

  // the original request URL leading to the resource
  // this is not necessairily the same as ServletRequest.getRequestURL as
  // it may have been processed by some URL mapping and folding
  String getOriginalURI();

  // the path to the actual resource providing the data
  // from the point of view of Sling this is just a string
  String getURI();

  // the selectors of the request or empty array if none
  // the selectors are dot-separated strings after the part of
  // original URI addressing the resource upto the extension
  // Examples:
  //    - /a/b/c has no selectors for resource /a/b/c
  //    - /a/b/c.html has no selectors for resource /a/b/c
  //    - /a/b/c.s1.s2.html has selectors [ s1, s2 ] for resource /a/b/c
  //    - /a/b/c.s.html/suffix has selector [ s ] for resource /a/b/c
  String[] getSelectors();

  // the extension of the request or empty string if none
  // the extension is a string after the last dot after the
  // part of the original URI addressing the resource upto the
  // end of the original URI or a slash
  // Examples:
  //    - /a/b/c has no extension for resource /a/b/c
  //    - /a/b/c.html has extension html for resource /a/b/c
  //    - /a/b/c.s1.s2.html has extension html for resource /a/b/c
  //    - /a/b/c.s.html/suffix has extension html for resource /a/b/c
  String getExtension();

  // the suffix of the request or empty string if none
  // the suffix is the string after the next slash after the part
  // of the original URI addressing the resource
  // Examples:
  //    - /a/b/c has no suffix for resource /a/b/c
  //    - /a/b/c.html has no suffix for resource /a/b/c
  //    - /a/b/c.s1.s2.html has no suffix for resource /a/b/c
  //    - /a/b/c.s.html/suffix has suffix suffix for resource /a/b/c
  String getSuffix();

}
```

The `ComponentRequest` interface would be modified as follows:

- The `getExtension()`, `getSelector(int)`, `getSelectors()`, `getSelectorString()` and `getSuffix()` methods are removed as this information can now be obtained from the `Resource` directly.
- The `getContent()`, `getContent(String)`, `getChildren(Content)` and `getRequestDispatcher(Content)` methods are replaced as follows:

```
public interface ComponentRequest extends HttpServletRequest {

  ...

  // Returns the Resource to which the getRequestURL method maps
  Resource getResource();

  // Returns a Resource to which the given URI String maps
  // Implicit: getResource().equals(getResource(getRequestURL()))
  Resource getResource(String uri);

  // Returns an Enumeration child Resources of the given Resource
  // If resource parameter is null, getResource() is used as parent
  // (use Enumeration to stay in line with the HttpServletRequest)
  Enumeration<Resource> getChildren(Resource resource);

  // Gets a RequestDispatcher to include the given resource
  RequestDispatcher getRequestDispatcher(Resource resource);

  ...

}
```

Extensions to the `Resource` interface

The `Resource` interface may be extended depending on the way, the resource is acquired. For example, there might be a `MappedContentResource` which would return an object mapped from any persistence layer, a `JcrResource` may encapsulate a JCR based resource. A resolver loading content from a JCR repository using Jackrabbit OCM might return a resource which implements both the `MappedContentResource` and the `JcrResource` interfaces.

MappedContentResource

```
public interface MappedContentResource extends Resource {

  // Returns the mapped data object
  Object getObject();

}
```

JcrResource

```
public interface JcrResource extends Resource {

  // Returns the JCR session used to acquire the Node
  // (this is actually convenience as getNode().getSession()
  // must return the same session)
  Session getSession();

  // Returns the JCR Node addressed by the Resource URI
  // this is the same as getSession().getItem(getURI());
  Node getNode();

}
```

The existing ContentResolver will be retargeted to the `Resource` interface and return an object implementing the `MappedContentResource` and the `JcrResource` interfaces if a mapping exists. Otherwise an object just implementing the `JcrResource` interface is returned providing just the resolved node.

Open Issues

This above definition leaves a series of issues open.

Resolving the Servlet

The `Component` interface is removed and the `Servlet` interface is used.

Currently the `Content` interface defines a method `getComponentId()` which returns the identifier of a `Component` to which processing of the request is dispatched. With the new `Resource` interface, no such method exists any more.

The intent is, that `Servlet` resolver would know about the concrete implementations of the `Resource` interface and could handle the respective resources. For example the Sling standard servlet resolver could try the following:

1. If the `Resource` is a `JcrResource` check the `sling:servletId` property of the resource node. If such a property exists and denotes a registered `Servlet` service, that servlet is used.
2. Otherwise, if the `Resource` is a `MappedContentResource`, find a `Servlet` service willing to handle requests for the actual object class of the mapped object. The `Servlet` service could be registered with a service property listing the names of the mapped object classes supported.
3. Otherwise try to find a registered `Servlet` interface willing to handle the request using the resource path, selectors and/or extensions.

Alternatively, the `Resource` interface might have a `getServletId()` method providing the identifier of the servlet to use. It might well be that the first solution is the better one.