Everything is a Resource

Introducing the Sling Paradigm: Everything is a Resource

Status: IMPLEMENTED
Created: 22. December 2007
Author: fmeschbe

1 Current State

Currently Sling uses resources, servlets and scripts as follows:

- The `Resource` interface is mainly used to abstract JCR `Node` instances
- The `ServletResolver` uses an internal registration of servlets registered as OSGi services with the interface `javax.servlet.Servlet` and selects the servlet based on the resource type of the `Resource` of the request only.
- The `ScriptResolver` uses the `ResourceResolver` to find request handling scripts based on the resource type of the `Resource` of the request, the request selector string and the request method or request extension.
- Request processing filters are based on OSGi services registered with the interface name `javax.servlet.Filter`
- Error handling is implemented in the `ServletResolver` implementation using the same mechanism to find a servlet (or script) based on the response status code or the caught `Throwable` as the pseudo request method name and using a different default error handling servlet.

This mechanism works rather good, but there are currently enhancement requests, which may not easily be implemented with the current concepts:

- Allow scripting of request processing filters. Implementing this requires special filter wrappers, which may select filter scripts.
- Enhance servlet selection to include the same parameters as script resolution, namely the request selector string and the request method or request extension. Implementing this would require replicating much of the code of the current `ScriptResolver` implementation.

2 Enter the Sling Paradigm

To overcome the limitations we introduce the Sling paradigm

Everything is a Resource

The Sling paradigm brings the paradigm of Java Content Repository API (JCR) *Everything is Content* to Sling.

This means, that every script, servlet, filter, error handler, etc. is available from the `ResourceResolver` just like normal content providing data to be rendered upon requests. To enable this resource resolution and resources have to provide certain functionality:

- Allow registration of resources with the resource resolver. This is required to access servlets and filters registered as OSGi services through the resource resolver.
- Provide eventing mechanism to support caching and cache management
- Extend resource adapter mechanism, that is to provide extension to the `Resource.adaptTo(Class<?>)` method
- Extend resource enumeration to include resources from various sources

3 Implementing the Sling Paradigm

3.1 Resource Provisioning

To be able to access resources from different locations through a single resource resolver, a new `ResourceProvider` interface is added. A resource provider is able to provide resources below a certain location in the (virtual) resource tree. The resource resolver selects a resource provider to ask for a resource looking for a longest match amongst the root paths of the providers. If the longest match resource provider cannot find the requested resource, the provider with the second longest match is asked, and so forth.

Accessing the JCR repository is also implemented in the form of a resource provider. This JCR resource provider is registered at the root – `/` – of the (virtual) resource tree. Thus the JCR repository is always asked if, no more specific resource provider has the requested resource.

The `ResourceProvider` interface is defined as follows:

```
package org.apache.sling.api.resource;

public class ResourceProvider {

    /**
     * The name of the service registration property containing the root paths
     * of the resources provided by this provider (value is "provider.roots").
     */
    static final String ROOTS = "provider.roots";

    /**
     * Returns a resource from this resource provider or <code>null</code> if
     * the resource provider cannot find it. The path should have one of the
     * {@link #getRoots()} strings as its prefix.
     * <p>
     * This method is called to resolve a resource for the given request. The
     * properties of the request, such as request parameters, may be use to
     * parametrize the resource resolution. An example of such parametrization
     * is support for a JSR-311 style resource provider to support the
     * parametrized URL patterns.
     *
     * @throws SlingException may be thrown in case of any problem creating the
     *             <code>Resource</code> instance.
     */
    Resource getResource(/* ResourceResolver resourceResolver, */
          HttpServletRequest request, String path) throws SlingException;

    /**
     * Returns a resource from this resource provider or <code>null</code> if
     * the resource provider cannot find it. The path should have one of the
     * {@link #getRoots()} strings as its prefix.
     *
     * @throws SlingException may be thrown in case of any problem creating the
     *             <code>Resource</code> instance.
     */
    Resource getResource(String path) throws SlingException;

    /**
     * Returns an <code>Iterator</code> of {@link Resource} objects loaded
     * from the children of the given <code>Resource</code>.
     * <p>
     * This method is only called for resource providers whose root path list
     * contains an entry which is a prefix for the path of the parent resource.
     *
     * @param parent The {@link Resource Resource} whose children are requested.
     * @return An <code>Iterator</code> of {@link Resource} objects or
     *         <code>null</code> if the resource provider has no children for
     *         the given resource.
     * @throws NullPointerException If <code>parent</code> is
     *             <code>null</code>.
     * @throws SlingException If any error occurs acquiring the child resource
     *             iterator.
     */
    Iterator<Resource> listChildren(Resource parent) throws SlingException;
}
```

Resource providers are registered as OSGi services under the name `org.apache.sling.api.resource.ResourceProvider` providing the list of resource path roots as a service registration property with the name `provider.roots`.

3.2 Adapters

The `Resource` and `ResourceResolver` interfaces are defined with a method `adaptTo`, which adapts the object to other classes. Using this mechanism the JCR session of the resource resolver calling the `adaptTo` method with the `javax.jcr.Session` class object. Likewise the node on which a resource is based can be retrieved by calling the `Resource.adaptTo` method with the `javax.jcr.Node` class object.

To use resources as scripts, the `Resource.adaptTo` method must support being called with the `org.apache.sling.api.script.SlingScript` class object. But of course, we do not want to integrate the script manager with the resource resolver. To enable adapting objects to classes which are not foreseen by the original implementation, a factory mechanism is used. This way, the script manager can provide an adapter factory to adapt `Resource` to `SlingScript` objects.

3.2.1 Adaptable

The `Adaptable` interface defines the API to be implemented by a class providing adaptability to another class. The single method defined by this interface is

```
/**
 * Adapts the adaptable to another type.
 *
 * @param <AdapterType> The generic type to which this resource is adapted
 *            to
 * @param type The Class object of the target type, such as
 *            <code>Node.class</code>
 * @return The adapter target or <code>null</code> if the resource cannot
 *         adapt to the requested type
 */
<AdapterType> AdapterType adaptTo(Class<AdapterType> type);
```

This method is called to get a view of the same object in terms of another class. Examples of implementations of this method is the Sling `ResourceResolver` implementation providing adapting to a JCR session and the Sling JCR based `Resource` implementation providing adapting to a JCR node.

3.2.1 SlingAdaptable

The `SlingAdaptable` class is an implementation of the `Adaptable` interface, calls the `AdapterManager` (see below) to provider an adapter to the `SlingAdaptable` object to the requested class. This class may be extended to have extensible adapters not foreseen at the time of the class development.

An example of extending the `SlingAdaptable` class will be the Sling JCR based `Resource` implementation. This way, such a resource may be adapted to a `SlingScript` by means of an appropriatley programmed `AdapterFactory` (see below).

3.2.1 AdapterFactory

The `AdapterFactory` interface defines the service interface and API for factories supporting extensible adapters for `SlingAdaptable` objects. The interface has a single method:

```
/**
 * Adapt the given adaptble object to the adaptable type. The adaptable
 * object is guaranteed to be an instance of one of the classes listed in
 * the {@link #ADAPTABLE_CLASSES} services registration property. The type
 * parameter is on of the classes listed in the {@link #ADAPTER_CLASSES}
 * service registration properties.
 *
 * @param <AdapterType>
 * @param adaptable
 * @param type
 * @return
 */
<AdapterType> AdapterType getAdapter(Object adaptable,
        Class<AdapterType> type);
```

This method is called by the `AdapterManager` on behalf of the `SlingAdaptable` object providing the `SlingAdaptable` as the `adaptable` parameter the requested class as the `type` parameter. Implementations of this interface are registered as OSGi services providing two lists: The list of classes wich may be adapted and the list of classes to which the adapted class may be adapted.

3.2.1 AdapterManager

The `AdapterManager` is an internal class used by the `SlingAdaptable` objects to find an `AdapterFactory` to delegate the `adaptTo` method call to. To make the `AdapterManager` available globally, it is actually defined as a service interface. Thus the adapter manager may be retrieved from the service registry to try to adapt whatever object that needs to be adapted - provided appropriate adapters exist.

The `AdapterManager` interface is defined as follows:

```
public interface AdapterManager {

    /**
     * Returns an adapter object of the requested <code>AdapterType</code> for
     * the given <code>adaptable</code> object.
     * <p>
     * The <code>adaptable</code> object may be any non-<code>null</code>
     * object and is not required to implement the <code>Adaptable</code>
     * interface.
     *
     * @param <AdapterType> The generic type of the adapter (target) type.
     * @param adaptable The object to adapt to the adapter type.
     * @param type The type to which the object is to be adapted.
     * @return The adapted object or <code>null</code> if no factory exists to
     *         adapt the <code>adaptable</code> to the
     *         <code>AdapterType</code> or if the <code>adaptable</code>
     *         cannot be adapted for any other reason.
     */
    <AdapterType> AdapterType getAdapter(Object adaptable,
            Class<AdapterType> type);

}
```

3.3 Change Events

The Sling `ResourceResolver` implementation defines events to be fired on changes in the (virtual) resource tree:

- All repository events are forwarded
- Resource provider addition and removal events are generated

Events are transmitted using the OSGi EventTracker specification. That is interested parties must register as OSGi event listener services.

3.4 Resource Enumeration

To be help in development and debugging and also to merely visualize the (virtual) resource tree, the resource tree must be explorable. That is, for every resource, the method `ResourceResolver.listChildren(Resource resource)` method must return all resources which may be considered children of the given resource.

Consider for example the following (partial) repository:

```
/
+-- filters
        +-- request
                +-- FilterA.esp
                +-- FilterB.jsp
```

Further consider the filter *FilterC* registered as an OSGi service. Thus the `listChildren` call for the resource at `/filters/request` must return three resources `/filters/request/FilterA.esp`, `/filters/request/FilterB.jsp` and `/filters/request/FilterC`. The first two will be JCR based resources, while the latter will be a servlet resource.

4 Employing the Sling Paradigm

4.1 Resources in Bundles

Resources may be located in OSGi bundles and mapped into the (virtual) resource tree by means of a `BundleResourceProvider`. Bundles containing resources indicate this fact by means of a special bundle manifest header: `Sling-Bundle-Resources`. Two notes regarding bundle resources:

1. Bundle entries are either files or directories. To have these files and directories be handled as if they would be file and folder nodes in a repository, bundle based files will have a resource type `nt:file` and bundle based directories will have a resource type `nt:folder`.

2. Bundle resource may be anything which may be represented by a file (or directory). That is the resources may be static content to be delivered to clients on request or resources may be scripts to be called to handle requests (or filter scripts even).

4.2 Servlets

Servlets to be used for request processing are registered as OSGi services with a series of required service registration properties:

1. `servlet.name` - The name of the servlet as returned from `ServletConfig.getServletName()`. If this property is not set, the `component.name`, `service.pid` and `service.id` properties are checked in order.
2. `servlet.path` - A list of absolute paths under which the servlet is provided in the (virtual) resource tree.
3. `sling.servlet.paths` - The name of the service registration property of a Servlet registered as a service providing the absolute paths under which the servlet is accessible as a Resource (value is "sling.servlet.paths"). The type of this property is a String or String[] (array of strings) denoting the resource types.
4. `sling.servlet.resourceTypes` - The name of the service registration property of a Servlet registered as a service containing the resource type(s) supported by the servlet (value is "sling.servlet.resourceTypes"). The type of this property is a String or String[] (array of strings) denoting the resource types. This property is ignored if the `SLING_SERVLET_PATHS` property is set. Otherwise this property must be set or the servlet is ignored.
5. `sling.servlet.selectors` - The name of the service registration property of a Servlet registered as a service containing the request URL selectors supported by the servlet (value is "sling.servlet.selectors"). The selectors must be configured as they would be specified in the URL that is as a list of dot-separated strings such as *print.a4*. The type of this property is a String or String[] (array of strings) denoting the resource types. This property is ignored if the `SLING_SERVLET_PATHS` property is set. Otherwise this property is optional and ignored if not set.
6. `sling.servlet.extensions` - The name of the service registration property of a Servlet registered as a service containing the request URL extensions supported by the servlet for GET requests (value is "sling.servlet.extensions"). The type of this property is a String or String[] (array of strings) denoting the resource types. This property is ignored if the `SLING_SERVLET_PATHS` property is set. Otherwise this property or `SLING_SERVLET_METHODS` must be set or the servlet is ignored.
7. `sling.servlet.methods` - The name of the service registration property of a Servlet registered as a service containing the request methods supported by the servlet (value is "sling.servlet.methods"). The type of this property is a String or String[] (array of strings) denoting the resource types. This property is ignored if the `SLING_SERVLET_PATHS` property is set. Otherwise this property or `SLING_SERVLET_EXTENSIONS` must be set or the servlet is ignored.

A `SlingServletResolver` will listen for `Servlet` services and - given the correct service registration properties - provide the servlets as resources in the (virtual) resource tree. Such servlets are provided as `ServletResource` instances which adapt to the `javax.servlet.Servlet` class.

4.3 Filters

Filters may be provided in two different ways: As `javax.servlet.Filter` instances registered as OSGi services and as scripts located in a predefined place. When requests are processed the filters are looked up in the (virtual) resource tree below the `/filters` node. The list of filters is comprised of all the filters directly below the respective scope – *request* or *resource* – and the those below the respective scope and the type of the resource of the request.

The filters are sorted by their names. Hence a convention for the names of the filters in the (virtual) resource tree is defined such that the names is composed of an ordering number and the actual filter name, e.g. *0_sample*.

4.3.1 Filter Services

Filters registered as OSGi services have three required service registration properties:

1. `filter.scope` - (String) Scope of the filter, which must be either *request* or *resource*
2. `filter.order` - (Integer) Call order of the filter used to define the filter call sequence
3. `filter.name` - (String) The name of the filter as returned `FilterConfig.getFilterName()`. If this property is not set, the `component.name`, `service.pid` and `service.id` properties are checked in order.
4. `filter.resource.type` - (String[]) The list of resource types to which this filter applies. This property is optional. If missing, the filter applies to all resource types. If this property is an empty list, the filter is not used as it applies to an empty list of resource types.

Such Filter services are added to the (virtual) resource tree at a path defined as follows for each resource type `resource_type` listed in the `filter.resource.type`.

```
/filters/${filter.scope}/${resource_type}/${filter.order}_${filter.name}
```

If the `filter.resource.type` property is missing, the filter is added at

```
/filters/${filter.scope}/${filter.order}_${filter.name}
```

4.3.2 Filter Scripts

Filter scripts may just be added as resources in the JCR repository at the appropriate location. For example for a request level filter applicable to `nt:file` nodes only, the filter would be placed in the `/filters/request/nt/file` folder.

4.4 Scripts from Resource

A `Resource` returned from the resource resolver may be a script. The script manager registers an `AdapterFactory` to adapt `Resource` to `SlingScript`. This factory will resolve a script engine for the resource file extension and return a `SlingScript` instance based on the `Resource`. If no script engine exists, the `Resource` may not be adapted.

The `AdapterFactory` adapting to a `SlingScript` is also able to adapt to `Servlet` by wrapping the adapted `SlingScript` in a `ScriptServlet`.

h3 4.5 Object Content Mapping

To cope with the new extensible functionality based on the `SlingAdaptable` class and adapter factories, object content mapping cannot be hard coded to just respond to any class. Instead, the Object Content Mapping functionality is in fact provided in terms of adapter factories, which are registered to be able to adapt instances the `Resource` interface to predefined types.

This way, Object Content Mapping takes part in adapter resolution just like any extensible adaption.

As a consequence, Object Content Mapping may probable be taken out of the current `jcr/resource` project into its own project.

5 Changes to the Code

5.1 Sling API

1. Add `org.apache.sling.api.adapter.Adaptable` interface
2. `Resource` and `RespourceResolver` interfaces extend the `Adaptable` interface
3. Add `org.apache.sling.api.resource.ResourceProvider` interface
4. Merge `SlingScriptResolver` and `ServletResolver`
5. Add `AdapterFactory` and `AdapterManager` service interfaces

5.2 OSGi Commons

The `org.apache.sling.osgi.commons` bundle is a new project providing the following functionality:

- `ServiceLocator` implementation (moved from `sling/core` project

5.3 Merge `scripting/resolver` into `sling/servlet-resolver`

The `SlingScriptResolver` and `ServletResolver` interfaces are merged into a single `ServletResolver` interface, which has a `resolve(SlingHttpServletRequest)` and a `find(ResourceResolver, String relPath)` method. The implementation of this method will apply the alogirthm of the current `scripting/resolver` implementation of the `SlingScriptResolver`.

Any script (or servlet or actually code) may call any script or servlet by just resolving the script or servlet to a `Resource` and adapting the resource found to a `SlingScript` or `Servlet`.

5.4 Separate Object Content Mapping from Resource Resolution

By applying the mechanisms of adapter factories, Object Content Mapping can be broken out of the `jcr/resource` project into its own project `jcr/ocm`.

5.5 Enhance Sling Console

Provide a Sling Console enhancement to explore the (virtual) resource tree

5.6 Create New Adapter Project

A new Adapter project `sling/adapter` takes the following classes:

- `SlingAdaptable` class implementing `Adaptable` and leveraging adapter factories
- Implementation of the `AdapterManager` service also used by `SlingAdaptable` class