Add ResourceResolverFactory Service Interface

API Enhancement: Add ResourceResolverFactory

Status: IMPLEMENTED
Created: 9. November 2008
Author: fmeschbe
JIRA: SLING-1262 and SLING-2396
Prototype: http://svn.apache.org/repos/asf/sling/whiteboard/fmeschbe/resource
Updated: 17. November 2008, fmeschbe, Added Section 5.2
Updated: 05. July 2012, cziegele, Implemented with some additions as outlined in SLING-2396

1 Current State

The Sling API currently only defines the `ResourceResolver` interface but not how such a resolver is to be created/retrieved.

In addition the Sling implementation has:

- [Sling] Repository providers, which just register one or more repository instances as services
- `JcrResourceResolverFactory` service which creates `ResourceResolver` instances based on JCR Session objects
- The authenticator selects one of the Repository services to authenticate against and creates the Session to get the `ResourceResolver` from the `JcrResourceResolverFactory`

This implementation has a series of drawbacks:

- The authenticator decides which repository to access: Problem is, that the authenticator is most probably not the right instance to decide on this issue.
- Getting a Resource resolver is tedious in that a Session is first to be created and then a resolver can be retrieved from the factory
- Only a single workspace of a single repository may be accessed at a any time and the authenticator actually defines which of those.
- All in all, this decouples the ResourceResolverProvider too much from the repository and gives the authenticator too much "power"

Therefore I propose the extension of the Sling API as presented herein.

2 Extensions at a Glance

This is the list of additions, I propose for the Sling API. Please refer to then following sections for more details, description and motivation.

- Add `ResourceResolverFactory` service interface which provides a `ResourceResolver` instances
- Add `ResourceProviderFactory` service interface to help `ResourceResolverFactory` in creating `ResourceResolver` instances
- Add lifecyle support to the `ResourceResolver` and `ResourceProvider` interfaces
- Add `LoginException` to support failure reporting in the `ResourceResolverFactory`

3 ResourceResolverFactory

The `ResourceResolverFactory` is provided by the Sling framework as a service. The factory provides a single method

```
ResourceResolver getResourceResolver(Map<String, Object> credentials) throws LoginException;
```

This method uses the credentials object to authenticate the user and to create `ResourceResolver` instances, which may then be used to access resources. If authentication fails, a `LoginException` (see below) is thrown.

Authentication fails if at least one `ResourceProviderFactory` service which is registered with the `provider.required` property set to `true` throws a `LoginExcdeption`. This for example allows multiple `ResourceProviderFactory` services to be registered to access the JCR repository, each for a different location/workspace and one factory service to be the one to decide, whether authentication succeeds or not. `ResourceResovlerFactory` services not having the `provider.required` property set to true will just be ignored if they cannot return a `ResourceProvider`.

4 ResourceResolver

The `ResourceResolver` interface is extended with a new method

```
void close();
```

which must be called when the resource resolver is no longer used. This allows for any cleanup required in the `ResourceProvider` instances attached to the resource resolver.

5 ResourceProviderFactory

The `ResourceResolverFactory` will return a `ResourceResolver` which behind the scenes makes use of `ResourceProvider` instances. This interface already exists today and is implemented to access bundle or filesystem resources. Actually repository access is internally also implemented as a `ResourceProvider`.

To allow for authenticated `ResourceProvider` instances in addition to unauthenticated ones, a new factory interface `ResourceProviderFactory` is introduced which provides a single method:

```
// Creates a normal resource provider using credentials
// to indicate the user to login as.
ResourceProvider getResourceProvider(Map<String, Object> credentials) throws LoginException;

// Creates an administrative resource provider. Any user
// credentials are ignored. The ResourceProvider is assumed
// to have administrative (aka root aka superuser) rights
// on the resources provided.
// This method is targeted as background services and not
// intended for normal request processing.
ResourceProvider getAdministrativeResourceProvider(Map<String, Object> credentials) throws LoginException;

// Creates an anonymous resource provider. Any user
// credentials are ignored. The ResourceProvider is assumed
// to have restricted (read-only mostly) rights on the
// resources provided. This factory method may be used to
// implemented guest access to the system.
// This method is targeted as background services. It may
// also be used by authenticators to support anonymous access
// to the system in a public site where authentication is
// the exception rather than the rule.
ResourceProvider getAnonymousResourceProvider(Map<String, Object> credentials) throws LoginException;
```

This method returns a resource provider with the given credentials or throws a `LoginException` if the credentials do not allow access to the resources provided by the `ResourceProviderFactory`.

The `ResourceProviderFactory` is a service interface and as such the registered `ResourceProviderFactory` services have the following defined service registration properties:

- `provider.roots` – Defines the root paths under which the ResourceProvider is attached into Resource tree. This is the same property as for the `ResourceProvider` service and applies to all `ResourceProvider` instances created by the factory.
- `provider.required` – Indicates whether the `ResourceProvider` instances provided by the `ResourceProviderFactory` have to be assumed as required. Failing to create as `ResourceProvider` with this factory, will cause the `ResourceResolverFactory.getResourceResolver()` method to fail if this property is set to the boolean value `true`.

5.1 Credentials

Since I cannot assume all real-world use cases right now and do not want to tie the Sling API into any specific use case, I propse the credentials to simply be a `Map<String, Object>` object. To simplify use, though, a few predfined entries are defined in the `ResourceResolverFactory` interface:

- `user.name` (String) – The name of the user to connect as to resource providers.
- `user.password` (String) – The password supplied by the user to connect to resource providers.
- `user.workspace` (String) – This convenience property may be used to hint at a primary workspace to connect to.
- `user.session` (javax.jcr.Session) – This property is primarily present to implement backwards compatibility for the `JcrResourceResolverFactory` interface. If this property is provided, `ResourceProviderFactory` services may wish to ignore any other properties and just use the JCR Session.

Other properties may of course be supplied and depend mainly on `ResourceProviderFactory` services, which may require or use them. Generally, though, providing the `user.name` and `user.password` properties should suffice it.

5.2 SlingRepository and AbstractSlingRepository

The `SlingRepository` interface and the `AbstractSlingRepository` abstract class currently provide support for getting administrative and anonymous sessions to the repository. Access details for such sessions is to be configured with the `AbstractSlingRepository` implementation.

With the implementation of a `ResourceResolverFactory` and its accessor methods for acquiring administrative and anonymous access, the respective `SlingRepository` and `AbstractSlingRepository` methods become superfluous.

In addition the `SlingRepository.getDefaultWorkspace()` method is probably also not required since, it is either the underlying actual JCR repository defining the default workspace or any JCR based `ResourceProvider` which defines which workspace to access.

As a consequence the `SlingRepository` interface and support for administrative and anonymous access to a repository in the `AbstractSlingRepository` may be phased out.

However, for some time, we might have to provide backwards compatibility:

- `SlingRepository.getDefaultWorkspace()` – Implemented by `AbstractSlingRepository`; always returns `null`
- `SlingRepository.loginAdministrative()` – Implemented by `AbstractSlingRepository`; calls `ResourceResolverFactory.getAdministrativeResourceResolver()` and returns a `Session` which on logout also closes the `ResourceResolver`
- `AbstractSlingRepository.login(null, workspace)` – Implemented by `AbstractSlingRepository`; calls `ResourceResolverFactory.getAnonymousResourceResolver()` and returns a `Session` which on logout also closes the `ResourceResolver`
- The `jcr/jackrabbit-server` and `jcr/jackrabbit-client` modules do not register `SlingRepository` instances anymore but the actual `Repository` instance as a `Repository` Service.
- The `SlingRepository` service is registered by a backwards compatibility class, probably related to the actual `ResourceResolverFactory`.

Alternatively, it might also be decided to drop `SlingRepository` altogether form Sling.

6 ResourceProvider

To allow for `ResourceProvider` instances returned by `ResourceProviderFactory` services to be cleaned up when no longer used, a new method is added to the `ResourceProvider` interface:

```
void close():
```

This method must close the `ResourceProvider` and cleanup any resources used. For example a JCR based `ResourceProvider` will logout the underlying session upon close.

This method will have no effect if called on a `ResourceProvider` which is registered as a service. For example the `BundleResourceProvider` will implement an empty `close()` method.

The `close()` method is not expected to throw any exception at all. If some error occurrs while processing cleanup, the method is expected to implement error processing, which might be logging a message or just to ignore the situation.

7 LoginException

The `LoginException` extends the existing `SlingException` and therefore is an unchecked `RuntimeException`. This exception may be thrown by the `ResourceProviderFactory.getResourceProvider()` and `ResourceResolverFactory.getResourceResolver()` methods to indicate failure to connect to any resources. The exception should provide a descriptive message as well as any cause which led to the `LoginException`.

8 Logging into the System

To log into the system, the authenticator will extract the credentials from the HTTP request and call the `ResourceResolverFactory.getResourceResolver` method with the credentials. If the `ResourceResolverFactory` throws a `LoginException`, authentication fails.

9 Changes in Sling

- The existing `JcrResourceResolverFactory` interface is deprecated
- The `JcrResourceResolverFactoryImpl` class is turned in to a `ResourceProviderFactory`, which accepts JCR Session instances in the credentials as the base for creating a `ResourceProvider` instance.
- A new implementation of the `JcrResourceResolverFactory` interface is created, which just relates to the `ResourceResolverFactory.getResourceResolver()` method providing the JCR Session as the only credentials entry.
- The jcr/resource module is split into two modules:
    - The `JcrResourceResolverFactoryImpl` and `JcrResourceResolver` classes are moved to a new module implementing the `ResourceResolverFactory` interface
    - The jcr/resource module keeps the exported classes and interfaces in the `o.a.s.jcr.resource` package as well as the internal implementations of the JCR support for the Sling Resource API

Existing code may still use the `JcrResourceResolverFactory` service to get a `ResourceResolver`. Still it is expected and proposed for this existing code to migrate to the new API over time. Existing code in the Sling project will be migrated to the new API as soon as it is available. New code should use the new `ResourceResolverFactory` interface.