Supporting CRUD

The current Sling resource API only allows reading of resources - with the PersistableValueMap we have a simple mechanism to support updates, however this approach comes with some problems (see below).

This is a concept how to fully implement CRUD via the resource API. As a first step we angle the problem from the users via: the API to be used by Sling applications. (SLING-2530)

Client API

Read

The support for read is sufficient, we don't need to change that much.

Delete

There are basically two options for a delete:

- add a *delete* method to the **Resource** interface
- add a *delete(Resource)* method to the **ResourceResolver** interface

While the first option seems to be more logical, it has the disadvantage that we have to enhance the **Resource** interface. Therefore I opt for the second option.
See section about persisting changes belowe.

Create

We add a new method *addChild(Resource parent, String name, ValueMap properties)* to the resource resolver, where the value map is optional. This will create the resource at the given parent with the given name and add the provided properties.
In the case of a JCR backed repository, the properties might contain *jcr:primaryType* and *jcr:mixins* - which are used to set the node type and mixins. Otherwise the defaults apply.
See section about persisting changes belowe.

Update

We currently have the PersistableValueMap which is an easy way of modifying a resource.

Like with delete, we have different options:

- add an *update(ValueMap)* method on the **Resource** interface
- provide a new *ModifiableValueMap* interface

The first option would require a new method on the **Resource** interface and has usability options, like how to get a changeable value map, how to ensure that a value map get from resource A might not be used to update resource B etc.
A new *ModifiableValueMap* is the easier approach. As the operations on a map usually do not throw exceptions, the modifiable map will store the changes locally and get an *update* method which pushes the changes into the resource provider.
See section about persisting changes belowe.

We should deprecate the *PersistableValueMap* as a *save()* call saves the whole session and this might include changes made through any other means to the session.(see SLING-1391 for some discussion about this)

Persisting Changes

There are two possibilities to handle changes:

1. A call to one of the methods, modifying a resource as outlined above are persisted immediately and **JTA** transactions will be supported
2. Changes are not persisted immediately but stored transient. When all changes are done, a *save/commit* calls needs to be done trying to persist all changes to all resource providers.

The second approach is more like people are used to when they are familiar with JCR and it allows to do bulk changes to the persistence. The first approach without using a transaction is easier to implement inside the resource providers and for many use cases sufficient as usually just a single resource is affected by REST calls.
For now, we don't have any transaction support, so if a save call goes to several resource providers, the first provider saves but the second fails, then the changes can't be rolled back from the first provider.

Resource Providers

A resource provider is mounted into the resource tree. In general, the providers are processed ordered by service ranking, highest first. A service provider gets a service property specifying the sub tree it is claiming to use. For example, a service provider might be mounted at /a/some/path. Everything below this path is processed by this resource resolver. However another resource provider might claim /a/some/path/and/down. So the longest matching path wins.

We need to add a new interface which can be implemented by a *ResourceProvider*. It gets a create method. Update and delete are directly handled by the resource.

Access Control

It's the task of a resource resolver to check if the current user is able/allowed to access a resource. If this is not possible, the corresponding action is denied.
As resource providers are mounted at a path (see above), the resource resolver delegates to a resource provider for a given path. If the user is not allowed to perform the action, the processing is stopped. There is no fallback to another resource provider. This avoids the problem that different users might see different resources (provided by different providers) depending on their rights.