# Sling IDE Tooling - API Refactoring

## Reasoning

The API being developed in the context of **SLING-2973** - Getting issue details... `STATUS` is very complex and also not consistently used all over the code yet ( **SLING-3089** - Getting issue details... `STATUS` and **SLING-4043** - Getting issue details... `STATUS` ). Also the API in the current form has some drawbacks: **SLING-3733** - Getting issue details... `STATUS` , **SLING-3684** - Getting issue details... `STATUS` , **SLING-3780** - Getting issue details... `STATUS` .

The refactored API being outlined in this page should address those concerns. Feel free to discuss this approach on the dev mailing list and directly add comments.

## ResourceProxy

Similar to `Resource` in Sling.

Instances are retrieved/created through the `ResourceManager`.

Represents a resource with properties and ordered child resources (tree structure).

No easy way to get the parent of an existing `ResourceProxy` (but shouldn't be necessary)

For reading it provides the methods

1. `String getName()`
2. `String getParentPath()`
3. `List<ResourceProxy> getChildren()` return values should be java.util.Collections.unmodifiableList(...)
4. `Map<String, Object> getProperties()` immutable, clarify which types are supported, only array types but not collection types for multivalue properties!
5. `Object getMetaData(String name)` custom meta data depending on the `ResourceManager` being used to create this instance. E.g. node type (for JCR) or file path (for ResourceProxies being created through a FS backed manager).

For writing it optionally provides the methods

1. `boolean setProperty(String key, Object value)` value could be array type, no collections are supported

## ResourceManager

Similar to `ResourceResolver` in Sling.

Retrieved via `ResourceManagerFactory.getResourceManager(Type, Credentials, URI)`.

1. `ResourceProxy getResource(String path)` may support lazy loading?
2. `putResource(ResourceProxy parent, String name, Map<String, Object> metaData)`
3. `reorderChildResources(ResourceProxy parent, String name, String afterName)` not supported by Sling API though so may not have an effect when being serialized/pushed to a server
4. `deleteResource(ResourceProxy parent, String name)`
5. `boolean validate(ResourceProxy parent, ResourceProxy child),` validates if the given ResourceProxy child is valid (e.g. might validate if there are node type restrictions being violated)
6. `close()` must implement `AutoClosable` and release all underlying resources (i.e. closing the remote repository and get rid off all http connections or closing file handles/input streams)
7. `revert()`
8. `commit()` for persisting the changes

## JCR DavEx Implementation

Uses remote JCR API calls through Sling's DavEx Support bundle.

## FileVault FS Implementation

Uses filesystem calls together with the FileVault Serialization Manager. Maybe even some more code from Jackrabbit FileVault can be reused.

## Package Manager Implementation

Probably not that useful, but may use package manager ReST API for mass-transfers. Would need to create the serialization format though internally.

# SerializationManager

Used by `ResourceManagers`

Problem is that one `ResourceProxy` is not necessarily bound to one serialization file. FileVault sometimes packages more than one resource in one XML file and sometimes a `ResourceProxy`is split up into multiple files (e.g. to separately store binary properties, see http://jackrabbit.apache.org/filevault/vaultfs.html).

Retrieved via `SerializationManagerFactory.getSerializationManager(Type).`

1. `SerializationData serialize(ResourceProxy, int depth, OutputStream)`
2. `ResourceProxy deserialize(InputStream)` may support lazy loading of Resource proxies!

SerializationData has `path, name,`

## FileVault Serialization Manager

1. Uses `DocViewSerializer`
2. Should use ContentParser

SerializationManager must have access to NodeTypeRegistry!