FAQ

Frequently Asked Questions

This page lists a series of common questions and answers. It is of course work in progress ...

This page is *not* meant for asking questions

Use the Sling users mailing lists for that, see http://sling.apache.org/project-information.html#mailing-lists - this page is about *answers*. Thanks!

If you find anything wrong in the Sling site or in and on the Wiki, do not hesitate to also contact the user's mailing list. Thanks.

Administration

How do I change Jackrabbit's admin password?

Using the userManager:

```
curl \
 -F"oldPwd=admin" \
 -F"newPwd=Fritz" \
 -F"newPwdConfirm=Fritz" \
 http://admin:admin@localhost:8080/system/userManager/user/admin.changePassword.html
```

You will also have to set that password in the Felix Web Management Console (/system/console/configMgr) under "Apache Sling Embedded JCR Repository." This is used by Sling to create an admin JCR session (using SlingRepository.loginAdministrative()) for components that need to have full access to the repository.

Note: Only after restarting the framework the old password will become invalid (as of 09-11-10).

Note: depending on the login module used in Jackrabbit, the password might not be checked at all (SimpleLoginModule, standard in Jackrabbit <= 1.4). Since Jackrabbit 1.5, the DefaultLoginModule provides full user support.

RESTful API

How do I create a node by posting a json document to a URL?

At the moment, you cannot do this. (Soon to change as per SLING-1172!) Instead, each value must be a field in the request POST. For example, suppose you have the json document:

```
{
  "greetings":"Hello, World!",
  "multi" : ["first","second"],
  "translations" : { "en": "Hello", "zh", "" }
}
```

You would do a post such as:

```
curl -F"greetings=Hello, World!" -F"multi=first" -F"multi=second" -F"translations/en=Hello" -F"translations/zh=
" http://admin:admin@localhost:8080/content/../../..
```

What so special about the 'content','apps' and '*' urls?

'apps' is reserved for matching scripts evaluated by sling.

The "*" url is used for POSTing to a child node.

By default, if a resource cannot be found from the root url, sling will try appending "content". For example, if you request the following non-existent resource:

```
http://localhost:8080/blog/first_post
```

Sling will look in:

```
http://localhost:8080/content/blog/first_post
```

Before returning a 404.

I posted a resource, where did it go?

Let's start by creating a resource:

```
curl -F"greetings=Hello, World" -F"translations/en=Hello" -F"translations/es=hola" http://admin:admin@localhost:
8080/content/greet
```

We can now view the resource with:

```
curl http://admin:admin@localhost:8080/content/greet.json
{"greetings":"Hello, World","jcr:created":"Fri Nov 06 2009 16:26:23 GMT-0800","jcr:primaryType":"sling:Folder"}
```

Notice that the "greet" resource is a sling:Folder. Also notice that it's a little hard to read the result. Let's tidy it up:

```
curl http://admin:admin@localhost:8080/content/greet.tidy.json
{
  "greetings": "Hello, World",
  "jcr:created": "Fri Nov 06 2009 16:26:23 GMT-0800",
  "jcr:primaryType": "sling:Folder"
}
```

But where did our translations go? To get them, we have to request 2 nodes down into the tree:

```
curl http://admin:admin@localhost:8080/content/greet.tidy.2.json
{
  "greetings": "Hello, World",
  "jcr:created": "Fri Nov 06 2009 16:26:23 GMT-0800",
  "jcr:primaryType": "sling:Folder",
  "translations": {
    "en": "Hello",
    "jcr:created": "Fri Nov 06 2009 16:26:23 GMT-0800",
    "es": "hola",
    "jcr:primaryType": "sling:Folder"
  }
}
```

However, if we try and get the resource without an extension, we get nothing found. This is because we're requesting a folder, so sling tries to find either an index.html or return a directory list, just like a normal directory on a webserver.

To fix this, we can use a script and a sling resource type. Let's update our greeting document:

```
curl F"sling:resourceType=greeting" -F"greetings=Hello, World" -F"translations/en=Hello" -F"translations
/es=hola" http://admin:admin@localhost:8080/content/greet
```

Then we'll post a simple esp script to apps:

GET.esp

```
<html>
  <head><title></title></head>
  <body>
    <h1><%= currentNode.greetings %></h1>
  </body>
</html>
```

```
curl -X MKCOL http://admin:admin@gandalf.local:8080/apps/greeting
curl -T GET.esp http://admin:admin@localhost:8080/apps/greeting/GET.esp
```

Now you should be able to see an HTML version of the resource at http://localhost.local:8080/content/greet. This script matches the sling:resourceType we set and the HTTP method we used. Note that resourceType matches must be exact.

How do I create a multi-value property with a single value, in HTTP?

Use this:

```
curl -u admin:admin -F'foo=bar' -F'foo@TypeHint=String[]' http://localhost:8080/some/path
```

The *TypeHint* tells the Sling POST servlet to create a multi-value property for *foo*.

I cannot add a node under /content/config.author using a POST, the new node goes under /content /config

That happens if both the /content/config.author and /content/config nodes exist, and you do something like:

```
$ curl -F try=first -u admin:admin http://localhost:8080/content/config.author/underauthor
```

The underauthor node goes under /content/config in that case, as Sling finds a resource at that path and considers .author as an extension or selector.

This is inherent to the way Sling matches URL paths to resources - to work around that, use

```
$ curl -F underauthor/try=second -F "underauthor/jcr:primaryType=sling:Folder" -u admin:admin http://localhost:
8080/content/config.author
```

Which correctly creates the underauthor node under /content/config.author. You can of course add more properties to the request, like -F `underauthor/jcr:primaryType` if needed.

Here's the resulting content of our example (including specifying the jcr:primaryType):

```
$ curl http://localhost:8080/content.tidy.5.json
{
  "jcr:primaryType": "nt:unstructured",
  "config.author": {
    "foo": "bar",
    "jcr:primaryType": "nt:unstructured",
    "underauthor": {
      "try": "second",
      "jcr:createdBy": "admin",
      "jcr:created": "Fri Jan 25 2013 17:35:18 GMT+0100",
      "jcr:primaryType": "sling:Folder"
    }
  },
  "config": {
    "foo": "bar",
    "jcr:primaryType": "nt:unstructured",
    "underauthor": {
      "try": "first",
      "jcr:primaryType": "nt:unstructured"
    }
  }
}
```

Scripts and Servlets

How do I generate links to previous versions of a node?

Assuming a versionable node at /content/versioned, with sling:resourceType=foo, here's the /apps/foo/html.esp script that handles the /content/versioned.html request:

```
<html>

<%
// assume we have a versionable node
var iter = currentNode.getVersionHistory().getAllVersions();
%>

  <body>
    <h1>Versions of node <%= currentNode.getPath() %></h1>
    <%
            while(iter.hasNext()) {
              var v = iter.nextVersion();

              // use UUID of version node to build a link, and add a .version
              // selector to have another esp script process that request
              var uuid = v["jcr:uuid"];
              var vPath = currentNode.getPath() + ".version." + uuid + ".html";

              // Use Version creation date as the link text
              var vDate = v.getCreated().getTime();

              %>
              <a href="<%= vPath %>"><%= vDate %></a><br/>
              <%
            }
    %>
  </body>
</html>
```

The links to the individual versions look like: /content/versioned.version.313016e1.html where the first .version. selector causes a different esp script to be called to display the version data, and the 313016e1 selector is the UUID of the versioned node (real UUIDs are longer).

That request is handled by this second script, /apps/foo/version/html.esp (name will change soon, SLING-387):

```
<html>

<%
        // Get version node UUID, which is the second selector
        var uuid = null;
        var sel = request.getRequestPathInfo().getSelectors();
        if(sel.length >= 2) {
                uuid = sel[1];
        } else {
                response.sendError(400, "Version node UUID must be given as second selector");
        }

        // Get version node
        var v = currentNode.getSession().getNodeByUUID(uuid);
        var frozen = v.getNode("jcr:frozenNode");
        var title = frozen.title;
%>

  <body>
    <h1>Version of node <%= currentNode.getPath() %></h1>
    Name: <b><%= v.getName() %></b><br/>
    UUID: <b><%= uuid %></b><br/>
    Path: <b><%= v.getPath() %></b><br/>
    Frozen node path: <b><%= frozen.getPath() %></b><br/>

    <% if(title) { %>
            Frozen node title: <b><%= frozen.getProperty("title") %></b><br/>
    <% } else { %>
            Frozen node does not have a title property
    <% } %>
  </body>
</html>
```

Which uses the UUID selector to retrieve the versioned node.

The second trick here is that the versioned data is saved as a "jcr:frozenNode" node under the Version node. This is explained for example at http://www.onjava.com/lpt/a/6784 .

How do I find out why a given script or servlet is preferred to another when processing a request?

See SLING-580, the SlingServletResolver class logs detailed information (at the DEBUG level) to indicate in which order the candidate scripts and servlets are considered for processing a request.

How do I render a script for a star "*" resource?

"*" resources do not have a sling:resourceType which can cause confusion when you're trying to render a specific script. Consider:

Suppose we have content such as:

```
content
--gradapp
----application
--------app1
--------app2
------------tabs
----------------tab1
----------------tab2

apps
--gradapp
----application
--------edit.esp
--------html.esp
--------list.esp
----tab
--------edit.esp
```

In this case, http://localhost:8888/gradapp/application/app1.edit.html will provide an edit page for the app1 resource using the script from apps/gradapp /application/edit.esp. However, http://localhost:8888/gradapp/application/*.edit.html will not use that edit.esp script.

By default the "star resource" does not have a resource type, so you get the default rendering. To give it a specific resource type based on its path, you can install and start the samples/path-based-rtp bundle.

Another suggestion is to register a generic node creation form script, e.g. at /apps/sling/servlet/default/create.esp. You should be able to invoke that script by browsing to /gradapp/application/*.create. If you want the create.esp script to be able to render different forms
(e.g. one for applications, one for tabs) you can use different selectors, like *.createTab, *.createApp, etc.

An older version of this answer suggests using a query parameter, this also works but we recommend using selectors in Sling, leading to cleaner and cachable URLs.

So this would work:

```
/gradapp/application/*.create?typeToCreate=application
```

for creating application nodes and

```
/gradapp/application/*.create?typeToCreate=tab
```

for showing the tab form, but this is the preferred way:

```
/gradapp/application/*.createApp
```

for creating application nodes and

```
/gradapp/application/*.createTab
```

for showing the tab form.

See: http://markmail.org/message/htl6r3uctuzb6l5q and http://mail-archives.apache.org/mod_mbox/sling-users/200911.mbox/%3c8A802DC6-7472-4040-807A-D55524F30D3E@gmail.com%3e

How to replace the default json renderer (for example) with my own?

The JSON rendering is done by the DefaultGetServlet, which is hardwired to use the JsonRendererServlet for .json extensions.

If a servlet or script is registered for the `sling/servlet/default` resource type, but with a specific `sling.servlet.extensions` property (set using the `@scr.property` annotation), it will take over and process GET requests which have a .json extension and no specific servlet or script.

As scripts and servlets are equivalent in Sling, the simplest way to do this to create a script at `apps/sling/servlet/default/json.esp`, for example.

The same logic applies to other extensions (html, txt, ...) handled by the DefaultGetServlet.

How to execute scripts directly?

The following servlet (inspired from the Sakai ScriptRunner) executes scripts directly when called with the script URL and a *.runscript* selector (for example */foo/myscript.esp.runscript.html*).

Note that this can be **insecure**: if users are allowed to upload scripts, they can execute any code supported by Sling, so use that only if you know what you're doing.

ScriptRunnerServlet

```
/* @scr.component
 *    immediate="true" label="ScriptRunner"
 *    description="Runs scripts using the .runscript selector"
 *
 * @scr.service
 *    interface="javax.servlet.Servlet"
 * @scr.property
 *    name="sling.servlet.resourceTypes"
 *    value="sling/servlet/default"
 * @scr.property
 *    name="sling.servlet.selectors"
 *    value="runscript"
 * @scr.property
 *    name="sling.servlet.methods"
 *    value="GET"
 */
public class ScriptRunnerServlet extends SlingAllMethodsServlet {

protected void doGet(
  SlingHttpServletRequest req,
  SlingHttpServletResponse resp)
  throws ServletException, IOException {
    Servlet s = req.getResource().adaptTo(Servlet.class);
    if(s == null) {
      throw new
        ServletException("Resource "
        + req.getResource()
        + " does not adapt to a Servlet");
    }
    s.service(req, resp);
  }
}
```

How do I create a new script engine?

As I write this, we don't have documentation on how to create more script engines, but that's not too hard to do if you take one of the simple existing engines as an example.

The JRuby engine for example, implemented in the *scripting/ruby* module, is built out of two simple classes, one that inherits from AbstractSlingScriptEngine, and one that inherits from AbstractScriptEngineFactory. The code is very simple, it's basically only a wrapper around the JRuby engine, that adapts it for Sling.

If creating a script engine, don't forget the *META-INF/services/javax.script.ScriptEngineFactory* file, which lets scripting subsystem know about the factory class, so that the engine is activated when the bundle that contains it is loaded.

Once the script engine is created, loading its bundle into Sling should be enough to activate scripts having the extension defined by the engine. If several scripts are found with the same name but different script extensions, the priority in selecting them is currently unspecified.

The javascript and freemarker engines source code also shows how to add automated tests to a script engine, including making a JCR repository available to the tests.

To go further, the javascript and jsp script engines are the most interesting ones to study.

The javascript engine provides wrappers to make it easier to access JCR and Sling objects from server-side javascript, and also uses a clever EspReader (sorry it's not *that* ESP) to convert *.esp* scripts to plain javascript code.

The JSP engine is actually a compiler, so it can be an interesting example if your language needs or can benefit from compiling.

Note: If the script engine you add involves compiling the scripts to Java Class files which are consumed by a classloader, it may be worth it to consider to properly seriliaze access to scripts which are under compilation to prevent paralell compilation and consequential class loading issues. Sling's JSP Engine is based on Jasper from Apache Tomcat and employs such serialization.

Working with bundles

Is there an easy way to update bundles in a running installation during development?

The Sling Maven Plugin provides an install goal which is able to install or update a bundle in a running Sling application (if the Sling web console is deployed). If the plugin properties are configured accordingly you can just `mvn clean package org.apache.sling:maven-sling-plugin:install` and the bundle is uploaded.

You can use the `settings.xml` to set the url to your Sling application. See the Sling Maven Plugin for more information.

Classloading issues

Accessing Classes from the Environment

Mostly when using the Sling Web Application, that is running Sling inside a web application deployed into some servlet container, you might want to share classes between the servlet container and Sling. Some examples of such sharing are:

1. Accessing EJB from the Application Server
2. Sharing classe with another web application such as a Jackrabbit instance
3. Using other container features

For such cases the OSGi Core Specification provides a functionality to declare such class sharing. The functionality is defined in terms of two Framework properties `org.osgi.framework.system.packages` and `org.osgi.framework.bootdelegation`:

1. **`org.osgi.framework.bootdelegation`** - All classes matching any entry in this list are always loaded from the parent class loader and not through the OSGi framework infrastructure. This property is a comma separated list of package names. A package name may be terminated by a wildcard character such that any package starting with the list entry matches the entry and thus will be used from the parent class loader.
2. **`org.osgi.framework.system.packages`** - Additional package declarations for packages to be exported from the system bundle. This property is a simple package declaration list just like any `Export-Package` manifest header. In a sense the `org.osgi.framework.system.packages` property may be seen as the `Export-Package` manifest header of the system bundle. Namely these entries may not contain wildcards (as is allowed for the `bootdelegation` property) and may contain directives and attributes such as the `uses` directive and the `version` attribute. It is recommended to provide this additional information to help in resolving the bundles. The OSGi Core Specification even prescribes the use of the `uses` directive.

The problem with the `org.osgi.framework.bootdelegation` property is, that it completely bypasses any bundle import wirings and just asks the parent classloader. Such situations are not easily recognizable. Therefore the Sling Console will be enhanced to mark any package import which matchs an entry in the `org.osgi.framework.bootdelegation` appropriately (SLING-148).

Also note, that any package listed as an import in a bundle must be resolveable for the bundle resolve. The import resolution process does not take the `org.osgi.framework.bootdelegation` configuration into account. This means, that regardless of whether a package is listed in the `org.osgi.framework.bootdelegation` property or not, if the package is listed as a required import in the `Import-Package` header, it must be exported by some other bundle.

How are the `sling.bootdelegation` properties used ?

Sling uses the `sling.bootdelegation.class` property name prefix to define lists of classes that must be added to the `org.osgi.framework.bootdelegation` property. In case you want to have a closer look, this is implemented in the `org.apache.sling.launcher.app.Sling.resolve()` method.

If a Sling property name starts with the `sling.bootdelegation.class.` prefix, the list of packages defined as the property value is appended to the `org.osgi.framework.bootdelegation` property, but only if the fully qualified class taken from the rest of the property name exists in the parent class loader.

Here's an example, from the jcr-client.properties file:

```
sling.bootdelegation.class.javax.jcr.Repository = \
 javax.jcr, \
 javax.jcr.lock, \
 javax.jcr.nodetype, \
 javax.jcr.observation, \
 javax.jcr.query, \
 javax.jcr.util, \
 javax.jcr.version
```

This means that, if the `javax.jcr.Repository` class is available in the parent class loader, all packages listed will be added to the `org.osgi.framework.bootdelegation`, making the corresponding classes available to OSGi bundles.

If the property name does not start with this `sling.bootdelegation.class.` property, the list of packages is just appended to the `org.osgi.framework.bootdelegation` property.

How does Sling support the `org.osgi.framework.system.packages` Property ?

Currently extending the `org.osgi.framework.system.packages` property in a Sling configuration file is only possibly by setting the `org.apache.sling.launcher.system.packages` property. The value of this property, which *must* start with a comma, is just appended to the `org.osgi.framewrok.system.packages` property.

A more elaborate support as is supported for the `org.osgi.framework.bootdelegation` Property has been prepared (SLING-147).

Should the `org.osgi.framework.bootdelegation` or the `org.osgi.framework.system.packages` Property be used ?

So, what mechanism should be used ? The answer is, that it depends.

Most of the time, you will want to use the `org.osgi.framework.system.packages` property. Because this property ensures that you will allways benefit from the normal class resolution mechanism through package imports and exports.

This allows creating the bundles normally by having the package import lists being built according to the packages used by the bundle classes. For example you may use the Apache Felix Maven Bundle Plugin to build your OSGi bundles and the imports are automatically calculated (by default).

The drawback of this method is, that there may be bundles in your system, which export packages also listed in the `org.osgi.framework.system.packages` property. Depending on the export version, the wrong package may be bound. So to prevent such collisions you should not install such bundles.

An example of such a declaration is the Servlet API packages (`javax.servlet`, `javax.servlet.http` and `javax.servlet.resources`). These packages are imported into the OSGi framework by the `SlingServlet` of the `launcher/webapp` project as part of the `org.osgi.framework.system.packages` property. To have this work correctly, no bundle should export the respective packages. In the case of Sling, this means, the `org.apache.felix.commons.sling-api` bundle must not be installed.

If on the other hand you cannot prevent the installation of such bundles and hence the export of the respective packages, you might want to set the `org.osgi.framework.bootdelegation` property conditionally as described above in the answer to how this property is supported in Sling. This ensures the property is only set, if the classes are actually available. This should be used as a fall back only, if the `org.osgi.framework.system.packages` method does not work.

How to share session between Sling and other web applications?

It is some times required to share HTTP session between Sling and other web applications. This is typically needed when existing Web MVC applicatios are getting migrated to Sling.
Most leading application servers (Oracle Weblogic, IBM Websphere) allow feature called 'shared session context'. It is allowed to share HTTP session between two web applications packaged in single EAR archive.
For sharing session with Sling, you need to package launchpad.war in to the EAR with other WARs. Sling is little tricky, because its not just a WAR. Its based on OSGI and on top of Web application classloading rules, it is also bounded by OSGI classloading rules.
I will explain configuration with weblogic here, because thats what I have tried out. Configuration in IBM Websphere should be similar.

The approach to share classes and session in Sling with other web
application in weblogic is as following.

1. Package CQ/Sling as web application in an existing EAR.
2. Deploy system.bundle extension fragment exporting all the packages
that you need to access in CQ.
3. In Weblogic, there are two ways to share classes
a) Put all the shared jars in APP-INF/lib directory in EAR. With
this approach you do not need Class-Path entries in MANIFEST.MF. And
classes are loaded by EAR classloader.
b) Put jars in some common lib in ear. Say EAR/lib. Then in all
the web applications have Class-path entry in MANIFEST.MF to list the
jars in EAR/lib. Make sure all the classes in shared jars are
serializable.
Update MANIFEST.MF in launchpad.war to have same Class-Path entries.
Now whenever you access a session object by
request.getSession().getAttribute(), the object will be serialized,
and then deserialized again. While deserialzing, it will resolve class
by Launchpad's classloader.

APP-INF/lib is the prefered approach to MANIFEST.MF, as the later
is adding unnecessary overhead of serialization and deserialization
for session sharing within same JVM.

In J2EE 5, there is addition of library element in
application.xml in EAR, which allows you to define EAR level library.
That is much cleaner than above two approaches.

There is one more approach that was tried out in our case. This is buggy and will not work, but I am explaining it here so that anyone else trying it out knows that it does not work.

1. In this approach, instead of deploying a system.bundle extension fragment, you package all the shared jars in an OSGI bundle and export all the shared packages from that bundle.
2. This seems to resolve classes only in JSPs in sling but fails in Sling servlets with ClassCastException.
The reason for this is as following.
Weblogic uses context classloader of the thread to resolve classes while deserializing session objects. When JSP is processed in Sling, the context classloader is set to org.apache.sling.commons.classloader.impl.ClassLoaderFacade
This classloader can find all the classes ex ported from OSGI bundles loaded in felix.
So, when session attributes are accessed from migrated JSPs in Sling, the objects used to get serialized, and then deserialized. While deserializing, the classes were resolved by org.apache.sling.commons.classloader.impl.ClassLoaderFacade.

Even if it appeared to work fine, this is not the right solution for the problem at all. The worst part here is that, once the object is deserialized, weblogic replaces original object reference to the deserialized object. So if any other WAR needs the object again, it needs to be serialized/deserialized again. But that works only if original object is loaded with weblogic classloader. So once object is serialized/deserialized with Sling classloader, it will never be serialized/deserialized for other WARs and you will always get ClassCastException.

It did not work with Sling Servlets because, when servlet is executed, the context classloader is weblogic.utils.classloaders.ChangeAwareClassLoader. This classloader finds classes in EAR or classpath, so we get ClassCastException. Even here, some weblogic classloader magic is going on. The context classloader here, is the classloader for launchpad.war. This is the WAR file for Sling. The classes referred by servlet can not be loaded by this classloader, because classes exported from OSGI bundles are not visible to this classloader. So welogic, seeing the ClassNotFoundException from context classloader, uses the WAR classloader of the WAR which set the object in the session. Obviously, we get ClassCastException in the sling servlet.

So class/session sharing should never be done with classes packaged and exported in an OSGI bundle. Relying on weblogic to serialize and deserialize is always likely to fail.

System fragment extensions is the only safer approach in this case.

Miscellaneous

Why can't I connect to Sling's WebDAV using Windows NetworkDriveMapping ?

Since Windows XP SP 2 (thus also affects Windows Vista, Windows 7, etc.) support for HTTP Basic authentication is by default switched off unless using HTTPS. Support can be switched on again by setting a Windows registry value. See Microsoft Knowledge Base entry 841215 for full details and warnings regarding modifying registry entries.

Why is my WebDAV connection so slow on Windows ?

One reason might be automatic proxy detection being enabled in the Internet Options. See the Fix Slow WebDAV Performance in Windows 7 for the solution.

Why should I use the sling:Folder node type instead of nt:folder ?

As you can see in the folder.cnd file (in CND notation), `sling:Folder` inherits from `nt:folder` and in addition allows any single or multi-valued property, and any child node with `sling:Folder` as the default child node type. The `nt:folder` node type is much more restrictive.

In general, using `sling:Folder` is recommended, as it's more flexible.

How to change the service.ranking of a service through configuration?

This is possible even if that property is not exposed in the configuration admin (either if the service is not a meta type or if the property is private). You can use jcrinstall to configure services by placing the configuration inside a `config` folder, for example `/apps/myapp/config/<myservice-pid>.<extension>` (extension depends on the config format). Note that "service.ranking" must be an Integer property, which you need to explicitly specify. This is currently only possible if you use the properties file format (see SLING-2477 for more).

For example, to make a service named "com.foo.app.impl.MyServiceImpl" have a ranking of 1234:

- create a file `/apps/myapp/config/com.foo.app.impl.MyServiceImpl.config`
- contents must be one line: `service.ranking=I"1234"` (I stands for Integer)