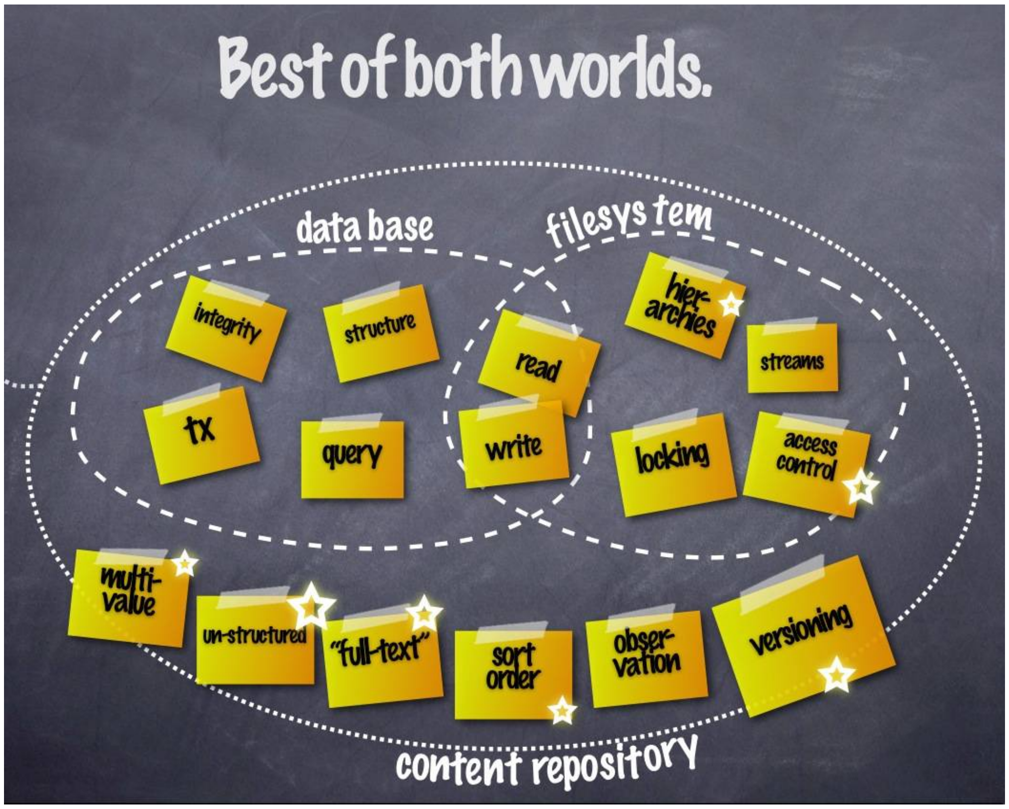
Java Content Repository: The Best Of Both Worlds

01.04.2010

| 32205 views |

Here's a short introduction to JCR, the Java Content Repository. Instead of going into details about the API, we'll just explain the bare minimum to allow you to get a feel for how JCR works and how it is used.

As shown in the picture below, JCR combines attributes of file systems and databases, and arguably provides the *best of both worlds*: using paths to locate atoms of contents, creating micro-trees to manage units of content, and still having database features like structured and full-text queries, transactions and optional schema constraints, creates a very powerful combination that's quite unique in the world of content management.



What is JCR?

According to JSR 283, the *Java Content Repository API* defines *an abstract model and a Java API for data storage and related services commonly used by content-oriented applications*.

JSR 283, released in 2009, is "V2.0" of the spec, and follows on JSR 170 which was released back in

2005. We'll talk about JSR 283 here, but if your repository "only" supports JSR 170 that won't make a difference for our simple examples.

The JCR storage model is a tree of nodes and properties: nodes (addressable by path like in a filesystem) are used to organize the content, and named properties store the actual data, either as simple types (string, boolean, number, etc.) or as binary streams for storing files of arbitrary size.

As an example, here's how an image file can be stored in JCR:

view source

print?

```
{
 "jcr:primaryType": "nt:file",
 "jcr:created": "Fri Dec 18 2009 13:45:03 GMT+0100",
 "jcr:content": {
  "jcr:primaryType": "nt:resource"
  ":jcr:data": 6269,
  "jcr:mimeType": "image/png",
  "jcr:lastModified": "Fri Dec 18 2009 13:44:51 GMT+0100",
 }
}
```

That JSON representation comes from *Apache Sling*, an application framework that provides a simple HTTP API on top of a JCR repository. We'll use Sling for our examples, as its HTTP requests and responses are easy to visualize here, but you could of course get the same results using the bare JCR API. We won't explain the Sling HTTP API in detail, for more details have a look at the *Sling in 15 minutes* tutorial.

The above JSON shows that our file is stored as a node of type *nt:file*, which is one of the standard JSR 283 node types. The name of that node (not shown in our example) is the filename, and the node contains a child node of type *nt:resource* (another standard node type), named *jcr:content*.

The file data itself is stored in the *jcr:data* binary property of the *jcr:content* node, and that node has a few other properties that provide the file size, last modification date and mime-type.

In JCR, node types can optionally be used to define constraints on the node content models. As an example, here's the definition of the *nt:resource*type, straight from the JSR 283 spec:

```
[nt:resource] > mix:mimeType, mix:lastModified
primaryitem jcr:data
- jcr:data (BINARY) mandatory
```

What this means is that the *nt:resource* type inherits the definitions of the *mix:mimeType* and *mix:lastModified* node types, and in addition requires a binary property named *jcr:data*. The *mix:** types use similar definitions to specify the names and types of the *jcr:mimeType* and *jcr:lastModified* properties.

This simple example shows how JCR splits things in nodes and properties, and as you can imagine using the right structures (or "micro-trees") to store your own content will help a lot in making things efficient and flexible. *David's model* will tell you more about this.

Unstructured content

Having precise definitions of content models, as in the file storage example, is very useful for content that has a well defined and stable structure. Binary files definitely fall into this category, but how about your own content?

Having to define such node types for all your content would be painful, especially when you're prototyping or exploring JCR. The *nt:unstructured* node type comes to the rescue by allowing a node to have any properties and child nodes of any type. Let's see an example, again using Sling's HTTP interface to create content.

view source

print?

```
$ curl -D - -F "title=hello, JCR" http://admin:admin@localhost:8888/foo
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
...

$ curl http://admin:admin@localhost:8888/foo.tidy.infinity.json
{
  "title": "hello, JCR",
  "jcr:primaryType": "nt:unstructured"
}
```

Using Sling's HTTP interface via *curl* (with *-F* for a POST request, without that for a GET), we create a node at */foo*, with a single *title* property.

To read the node, we make an HTTP GET request using the */foo.tidy.infinity.json* path, which tells Sling to return a formatted JSON representation of the */foo* node and all its child nodes and properties.

The JSON output of that request shows that the node has been created with the *nt:unstructured* type, so we should be able to add any content to it. Let's try that.

view source

print?

```
$ curl -D - \
  -F "text/content=This is some text" \
  http://admin:admin@localhost:8888/foo
HTTP/1.1 200 OK
...

$ curl http://admin:admin@localhost:8888/foo.tidy.infinity.json
{
  "title": "hello, JCR",
  "jcr:primaryType": "nt:unstructured",
  "text": {
    "content": "This is some text",
    "jcr:primaryType": "nt:unstructured"
  }
}
```

The first request is a POST that asks Sling to add some content at the *text/content* path relative to the */foo* node, and we see from the JSON output that this creates another *nt:unstructured* node under */foo*. Note that we didn't have to define any content model to do that, the beauty of *nt:unstructured* is that it lets you play with content without constraints.

Could we add a file under our */foo* node? You bet! Using Sling's WebDAV interface (provided by Jackrabbit), we just have to navigate to */foo/text*, which appears as a folder in the WebDAV explorer, and copy our file under that.

view source

print?

```
$ curl http://admin:admin@localhost:8888/foo.tidy.infinity.json
{
  "title": "hello, JCR",
  "jcr:mixinTypes": [
    "mix:lockable"
  ],
  "jcr:primaryType": "nt:unstructured",
  "text": {
    "jcr:mixinTypes": [
      "mix:lockable"
    ],
    "content": "This is some text",
    "jcr:primaryType": "nt:unstructured",
    "image1.jpg": {
      "jcr:mixinTypes": [
        "mix:lockable"
      ],
      "jcr:created": "Fri Dec 18 2009 14:31:32 GMT+0100",
      "jcr:primaryType": "nt:file",
      "jcr:content": {
        "jcr:uuid": "189d6fef-8c50-41c7-83c5-88026c78907d",
        ":jcr:data": 21784,
        "jcr:mimeType": "image/jpeg",
        "jcr:lastModified": "Fri Dec 18 2009 14:31:32 GMT+0100",
        "jcr:primaryType": "nt:resource"
      }
    }
  }
}
```

Once that's done, we see that the *image1.jpg* file has been added as an *nt:file* node alongside the *content* node that we created before, with a similar structure than in our first example above.

This simple example shows the power of JCR when it comes to combining different types of content. What we have created is a *micro-tree* of content that could represent a blog post, with attachments, comments etc. all stored under the same base path, with a combination of free-form and strictly structured data. That's quite a powerful way of creating agile content-based applications in a *data-first* way.

The API: Nodes and Properties

Let's give a quick overview of the main elements of the APIs, the ones that you use to create and retrieve content. In our examples, Sling uses those under the hood to process HTTP requests.

First, you need to acquire a *Repository* object. There are various ways of doing this, in Jackrabbit's case you can use a *TransientRepository* class to simplify setup for simple experiments.

The *Repository* class allows you to login and get a *Session*, which in turn allows you to create and retrieve content. The *Session* is tied to a *Workspace* - each workspace can store a distinct tree of nodes and properties, but for our simple examples we'll just use the default workspace.

view source

print?

```
Repository repository = new TransientRepository();
Session session = repository.login(...);
```

Now that we have a *Session*, we can access the root node of the tree, and create a simple subtree with a */hello/world* node that has a *message* property saying *Hello, JCR World!*.

view source

print?

```
Node root = session.getRootNode();
Node hello = root.addNode("hello");
Node world = hello.addNode("world");
world.setProperty("message", "Hello, JCR World!");
session.save();
```

Note the *session.save()* call at the end. JCR uses a *transient space* to prepare changes before saving them, providing a simplified transaction-like mechanism that's often sufficient when working with content. Real transactions are also possible with JCR, if needed.

Let's retrieve the node that we just created:

view source

print?

```
Node node = root.getNode("hello/world");
System.out.println(node.getPath());
System.out.println(node.getProperty("message").getString());
```

Quite straightforward: we just address the node by its full path relative to the root, and print out its path and the value of its message property. This would output */hello/world* and *Hello, JCR World!*.

The JCR API provides much more than the few simple interfaces shown here, but those are by far the most widely used parts of the API. I haven't checked but I'd guess 95% of the JCR API calls that we use in our content management products involve just the Node and Property interfaces.

Going further

Although we have just scratched the surface of JCR, the concepts exposed here are probably the most important ones:

- Combining unstructured and structured content provides a lot of flexibility, and content model constraints can be used selectively where they make sense. In terms of storage flexibility, think XML - but without the dreaded angle brackets.
- The storage model uses simple Node and Property interfaces, no need to study tons of documentation to use it.
- The API leads to fairly readable code, and although object-to-content mappers are available for JCR they are usually not necessary. Working directly with micro-trees is straightforward and close to the domain model of structured content.

To go further, the best might be to read the JSR 283 spec itself. API specifications are usually not one's favorite bedtime reading, but this one is surprisingly readable. I would not read it linearly however, and suggest the following reading order:

- The *Reading* chapter explains the various ways in which Nodes and properties can be accessed, using tree navigation.
- The *Query* chapter that follows explains how to query the JCR repository. Good content models allow you to go a long way without using queries, but they are obviously also needed at some point.
- The *Writing* chapter comes next in my opinion, as you'll obviously want to create some content!
- I would then suggest studying at least the *Observation* chapter, as getting selective callbacks when content is created or modified is very powerful and might be a bit unusual.

At this point, you would have most of the tools required to manage content, the rest of the spec's features are certainly useful but you might not need all of them.

It took me a while to really grasp JCR and use it effectively in my content management activities, but I'm not going back to the often messy world of hybrid *databases and files and message queues and a few others things* content management architecture. I like what my colleague Lars Trieloff sometimes calls the *Zero Bullshit Architecture* much better!

Author bio

Bertrand Delacretaz works as a senior developer in Day Software's R&D group, using open source tools to create world-class content management systems. Bertrand is a member of the Apache software foundation, served on its Board of Directors from 2008 to 2009, and is involved in a number of Apache projects related to content management.