GSoC 2010 mini-CMS project

Contents

Overview (from SLING-1438)

This is a Google Summer of Code 2010, Federico Paparoni has been accepted as a student to work on it, mentored by Bertrand Delacretaz

The goal is to create a mini-CMS with Sling, that demonstrates Sling best practices.

See http://tinyurl.com/asfgsoc for the full list of GSoC 2010 projects at the ASF, and http://community.apache.org/gsoc for general GSoC information.

Introduction

Apache Sling is an opensource project with a lot of technologies and features. The goal of this project is to create a mini-CMS, that developers can use to understand how to develop a simple application with Sling. So it is necessary to know a little about two main topics: OSGi and JCR. The following links are useful resources to read something about these technologies.

OSGi

- http://en.wikipedia.org/wiki/OSGi
- http://felix.apache.org/
- http://www.osgi.org/Links/BasicEducation

JCR

- http://jackrabbit.apache.org/
- http://www.ibm.com/developerworks/java/library/j-jcr/
- http://wiki.apache.org/jackrabbit/DavidsModel

The repository for this project can be found at http://code.google.com/p/davidgsoc2010/.

Some words about David Mini CMS

This project shows some features of Apache Sling and can be used for educational purpose to move your first steps with this framework.

David uses the following opensource library/technologies:

- jQuery 1.4.2
- jQueryUI 1.8
- LavaLamp - A menu plugin for jQuery with cool hover effects
- WP-Cumulus SWF - used to display tag cloud
- iText
- CKEditor

The available features are

- CRUD (Create, Read, Update and Delete) for the content
- Full HTML creation of article
- Creation of PDF rappresentation of a single post or the entire list of contents available

- Tagging system
- Search by title,text,tag
- Send email to someone with the link of article
- Background service that checks number of articles/tags in the repository

These aren't space age features, but are useful to understand how to create an application using Apache Sling

First steps into Sling

Firstly you must setup the environment for Sling, so you can follow the guide at http://sling.apache.org/site/getting-and-building-sling.html.

Now that you created your environment, you can setup some other tools that can be useful during the development:

- cURL : Command line tool to send HTTP request
- JCR Explorer : Extension that create a useful explorer for the JCR repository you are working in
- BitKinex : FTP/SFTP/HTTP/WebDAV Client that can be used to manage the files you uploaded in Sling

It's time to make our "Hello world" in Apache Sling.
Open a console and simply launch the next command:

```
curl -F"sling:resourceType=foo/bar" -F"title=Hello world" http://admin:admin@localhost:8080/content/myfirstnode
```

This is a simple HTTP request, where you pass some parameters and values. Using it you have created a first resource under Sling.
The resource is a JCR node, as every resource in Sling, put under folder /content. This node has two parameters, title and sling:resourceType.

Node creation is a simple task, but you must understand how you can render the information stored in the nodes using Sling.
The first document you can read is the next one: http://dev.day.com/content/ddc/blog/2008/07/cheatsheet/_jcr_content/par/download/file.res/cheatsheet.pdf . It simply describes how content resolution works in Sling.

Another important information to better understand Sling, is that a resource is rendered by a script.
Script files are stored under the folders /apps or /libs and there is a wide choice of possible scripting engine:

- ESP
- JSP
- Java Servlet
- Scala
- Python
- Groovy
- Ruby

To select a script, Sling uses the node's sling:resourceType property. So if we say that sling:resourceType is foo/bar, Sling will search under the /apps/foo /bar/ folder.
The following links describes how the scripts work and what type of variables we have during the script execution:

- https://cwiki.apache.org/SLING/scripting-variables.html
- https://cwiki.apache.org/SLING/url-to-script-resolution.html

Content loading

You can setup some initial contents that can be used in your application. It is a useful thing, because with a simple configuration you have some nodes already created when your application starts.

In David there are two different nodes created when you deploy your application: /content/david and /content/tags .

These nodes are defined in the application folders, using a JSON format. Every information stored in these JSON will be a property of the created nodes.

The most important property is the next one

```
"sling:resourceType": "david"
```

This property defines "david" as resource type, so Sling knows that it will search under the folder /apps/david to find the scripts that will be called on this node.

These JSON files are loaded using the Maven Bundle Plugin , as you can see in the David core/pom.xml file

```
<!-- initial content to be loaded on bundle installation -->
<Sling-Initial-Content>
        initial-content;overwrite:=true;uninstall:=true
</Sling-Initial-Content>
```

JSON isn't the only way to load initial content. Further informations about content loading can be found in the Content Loading Bundle Documentation.

Create new entry

There is a script that provide this basic function, /apps/david/new.esp. As you can see in David I choose the ESP scripting language, but as we already said, you can choose among a lot of scripting engines with Sling. This script loads two other script files, used in every script of David: /apps/david/header.esp and /apps/david/menu.esp. These scripts, as the name suggests, contain header informations and the menu for David.

In the header there are jQuery functions and CSS definitions, useful for the whole CMS. In the menu script we can find the definition of a classic menu.

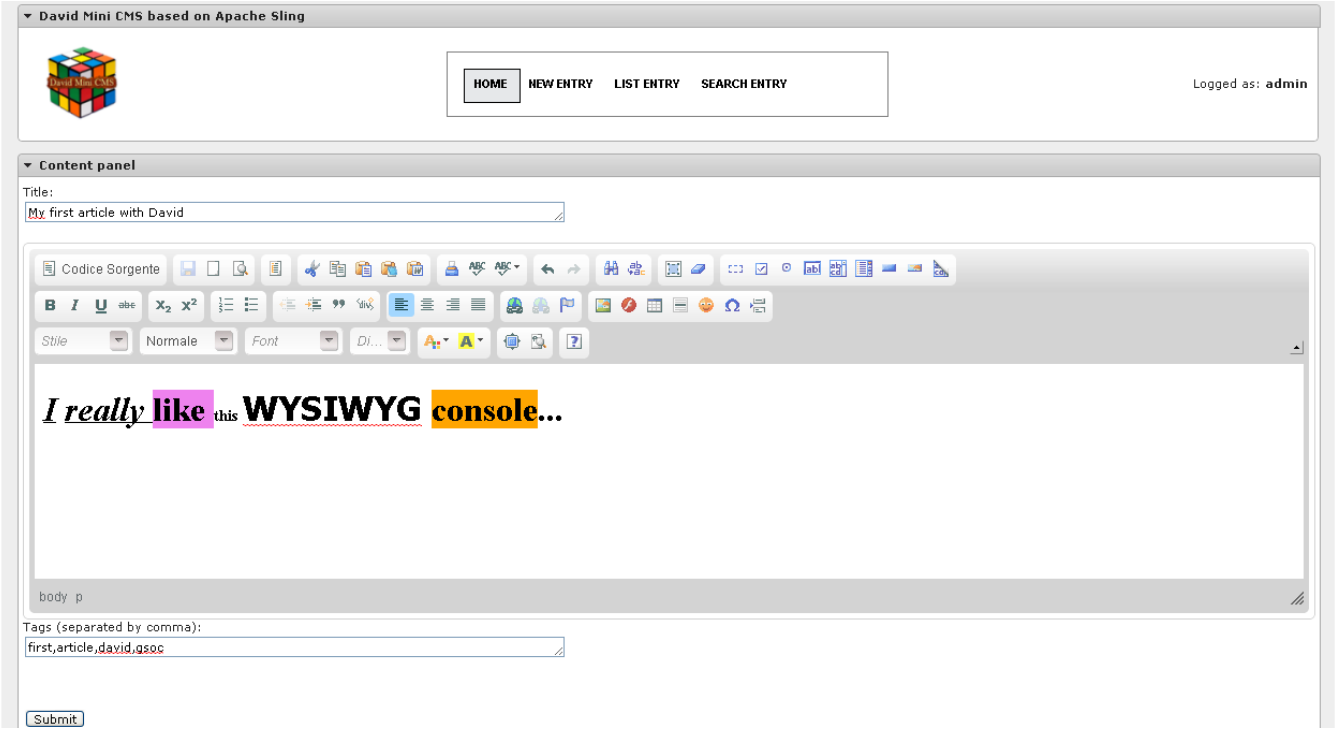
Turning back to the new.esp script, we can see in the following code as header and menu are loaded, using an ESP function.

```
...
...
<title>David Mini CMS</title>
<%
load("header.esp");
%>

</head>
<body onload="checkAuth()">
        <div id="lCenter">
        <div id="desktop">
        <%
        load("menu.esp");
        %>
        <div id="contentPanel" class="centralPanel">
...
...
```

So we loaded these two scripts in new.esp. In addition to this, in this script we defined a simple form, with some input texts and a CKEditor panel.

Once the user fills the input, the page is like in the following image



The submit button of this page is bounded to a jQuery function defined in the header.esp file. To create a new entry we have only to create a HTTP POST request, including the informations the user put in the input texts. The function that create the new entry is the following:

```
......
...
//Function called when there is a click of the
//submit button in new.esp page
$('#button').click(function() {

    //Retrieve the input texts from the HTML
    var title = $("#title").val();
    var text = $("#editor1").val();
    var editor_data = CKEDITOR.instances.editor1.getData();
    text=editor_data;
    var tagValues = $("#tags").val();

    //Now create a list with tags
    tagValues=tagValues.split(",");


    for(var i=0; i<tagValues.length; i++) {
        tagValues[i] = tagValues[i].replace(/ /g,'');
    }

    var token = new Array();
    for(var i=0; i<tagValues.length; i++)
                if(tagValues[i] != "")
                        token.push(tagValues[i]);

    //Every information is stored in the "data" variable
    //sling:resourceType tells the repository that this entry is a David one
    //jcr:mixinTypes=mix:referenceable defines this entry as referenceable (see JCR)
    var data="title="+title+"&text="+text+"&sling:resourceType=david&jcr:mixinTypes=mix:referenceable";

    for (var j = 0 ; j < token.length ; j++){
                data=data.concat("&tag="+token[j]);
    }

    //The current date will be used to create the folders
    //in the Sling repository where we will put the content
    var currentDate = new Date();
    var year = currentDate.getYear()+1900;
    var month = currentDate.getMonth()+1;
    var day = currentDate.getDate()+1;

    if (month<10)
        month = "0" + month;
    if (day<10)
        day = "0" + day;


    //The url is under the David root node /content/david
    //created during the initial content loading
    var url="/content/david/"+year+"/"+month+"/"+day+"/";

    //Simple AJAX call to create a HTTP POST request
    $.ajax({
        type: "POST",
        url: url,
        data: data,
        success: function(msg){
                alert("Entry saved");
                window.location = "/content/david.html";
        },
        error: function(msg){
                alert("Error during save");
        }
    });
});
...
...
...
```

If we click on the button and everything goes well, we will see a popup with the text "Entry saved" and our article will be in the repository at the path */content/david/YEAR/MONTH/DAY/something/liketitle*.

As you can see, there isn't a definition for the name of the entry but anyway we will have this entry saved...how is it possible?

The request URL we created is where we would like to insert our entry. If this URL already exist, we will only update properties of this node (and so you can already understand that the edit page will be equal to this one).

If the resource doesn't exist, a new item is created. If the resource name ends with /* or /, the name of the item will be created using an algorith that also uses the name of new node. The creation of the new node goes through the SlingPostServlet , a frontend for the content manipulation. This servlet provides also other content operations, as described here.

In the HTTP POST request there are also some others fields:

- ***sling:resourceType=david*** : This information is stored on the new node, so when we will retrieve this node from the browser, using any extension (for example .article provided by /apps/david/article.esp) Sling will search under the folder /apps/david.
- ***jcr:mixinTypes=mix:referenceable*** : Another information that will be stored. In that way this node can be referenceable. This feature will be useful for the tags management.

Authentication

You can submit new entry only if you have already authenticated with Sling. So you can see that on the /apps/david/menu.esp script there is a check for the credentials

```
......
...
<td width="33%" align="right">
        Logged as: <b id="username">????</b>
</td>
</tr>
</table>
<script language="javascript">
        var info = Sling.getSessionInfo();
        if (info.userID=="anonymous")
                document.getElementById("username").innerHTML =
                        info.userID+"<a href='/system/sling/form/login?resource=/content/david.html'>Login</a>";
        else
                document.getElementById("username").innerHTML = info.userID;
</script>
...
...
```

Let's now explain this code. Using the sling.js, system utility defined in Sling, we can get the session information object. If the userid of this object is ***anonymous*** we put a link to the authentication form.

Otherwise we simply print the userid.

The authentication form receives a ***resource*** parameter, that is the resource where the user will go after the authentication. This sort of authentication is a basic one, if you want to know more about authentication in Apache Sling you can read the authentication documentation on Sling website.

Edit existing entry

As said before, the update operation is really similar to the creation of new content. Anyway in David there a separated script to handle this operation, /apps/david/edit.esp.

It's an unuseful script, because also this operation can call new.esp. There is a need of a bit refactory about it.

List existing entries

Once you saved some entries in David, you can see a complete list of articles using the script /apps/david/list.esp.

Firstly there is a JCR query to find all the items

```
var queryManager = currentNode.getSession().getWorkspace().getQueryManager();
var query = queryManager.createQuery("/jcr:root/content/david/*/*/*/element(*, nt:unstructured) order by
@created descending", "xpath");
var result = query.execute().getNodes();
```

In this way we have a variable *result* with a NodeIterator object. So we simply have to make a loop on this iterator and print its values

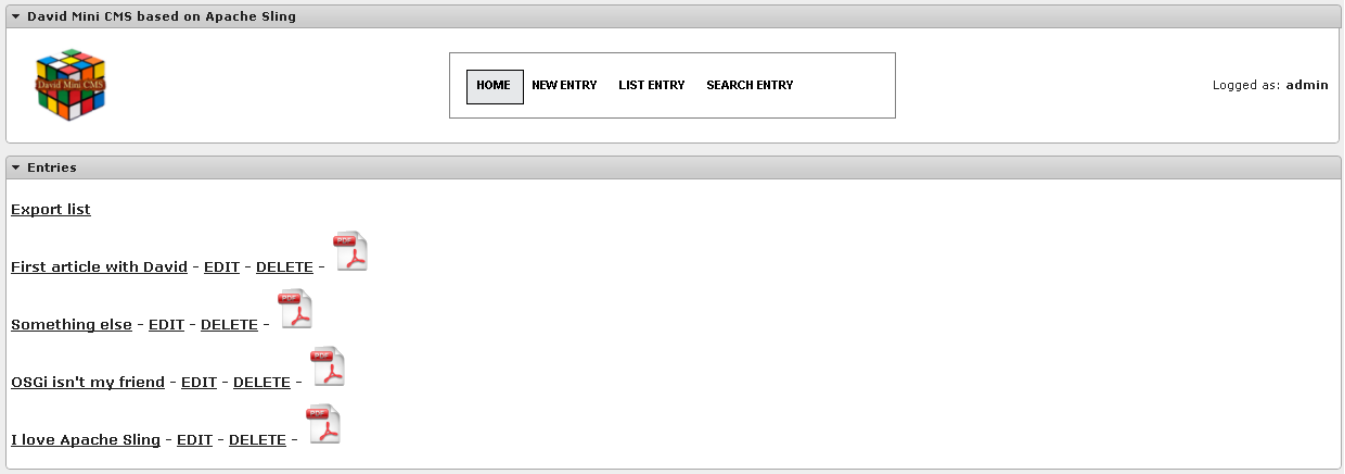
```
<h3>Entries</h3>
<div>
        <h3><a href='/content/david.pdf'>Export list</a></h3>
        <%
        while(result.hasNext()) {
                post = result.nextNode()
        %>

        <h3> <a href="<%= post.getPath() %>.article"><%=post.title%></a> -
        <a href="<%= post.getPath() %>.edit">EDIT</a> -
        <a id="<%= post.getPath() %>" class="delete" href="#">DELETE</a> -
        <a href="<%= post.getPath() %>.pdf"><img alt="Get pdf" src="/libs/images/pdf-icon.png"></a>
        </h3>

        <%
        }
        %>

</div>
```

The result is something like the following image



When you click on the article, you call the /apps/david/article.esp script, that is a simple script where the contents of article are rendered. For the others links, read the following paragraphs.

Delete existing entry

The delete operation is an easy task, because we only have to make a HTTP POST request to the article URL, with a special parameter ***:operation=delete***

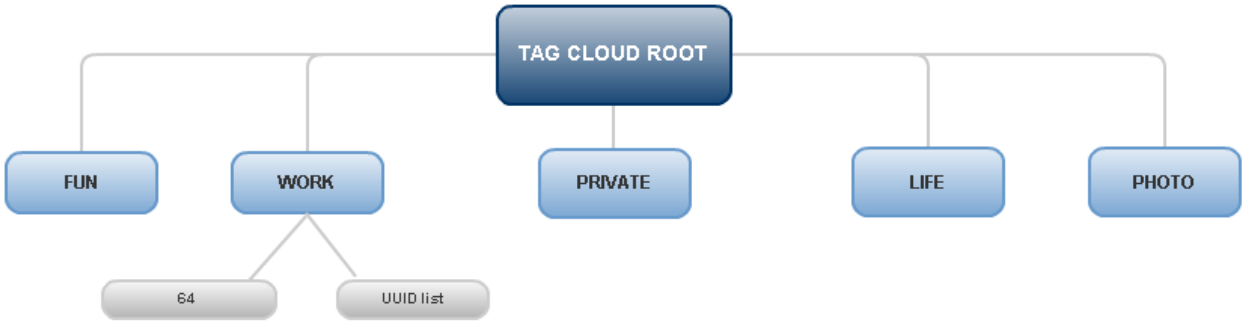
This request is made with a jQuery function bounded to the DELETE link, as you can see in the /apps/david/list.esp script

```
$.ajax({$.ajax({
        type: "POST",
           url: id,
           data: ":operation=delete",
           success: function(msg){
                $(this).dialog('close');
                location.reload();
           },
        error: function(msg){
                alert(msg);
                $(this).dialog('close');
        }
});
```

Generate the tags content structure

David manages a simple tagging system for the articles. The tags related to the article, that user insert in the new.esp page, are saved as a multi-value property on the article node.

In addition to this, there is also a redundant structure to achieve the tags information, as shown in the next image



So David must manage tags in the creation of the article and in its removal. To accomplish these tasks I used a two different approach.

For the creation of the article there is a component deployed in Apache Sling. This component listens for the creation of a new Node and manage the add of tag values.

This service is implemented in the class sling.gsoc.david.jcr.TagGenerator. This class is a OSGi component, with annotations that define it as a component (see Maven SCR Plugin for more details)

```
@Component(metatype = false, immediate = true)
```

Using the annotations we have also the reference to SlingRepository, that will be used to gain access to Sling repository

```
<at:var at:name="Reference" />Reference
private SlingRepository repository;
private SlingRepository repository;private SlingRepository repository;
```

Being an OSGi Component there is the implementation of ***activate*** and ***deactivate*** methods, which is called when the component is activated/deactivated.

TagGenerator class implements EventListener interface, so in the ***onEvent*** method we add the tags of the created node to the tagging structure.

When we delete a node, we can see that there is another AJAX call in the /apps/david/list.esp script before the real removal. There is a call which uses the ***:operation=deletetag*** parameter.

This operation is implemented by David, using the sling.gsoc.david.operation.DeleteTagOperation class. In Sling you can define new operations extending SlingPostOperation. The new operation is defined with the constant ***sling.post.operation*** as we can see in the next code

```
@Property(value = "deletetag")
static final String OPERATION = "sling.post.operation";
```

When we deploy David, the new operation is registered and every request with this new operation will be managed by our class. When this operation is called, DeleteTagOperation select the nodes related to the article tags and remove the UUID of the article from this list. Every operation, the add and the removal, based on the JCR repository must end with a call to the save method of the root node where we added/removed nodes.

Display tags using the Flash-based animation

The homepage of this CMS is related to the /apps/david/html.esp script. From the script side, there is only the inclusion of WP-Cumulus, a flash based tag cloud. This engine requires an XML with tags informations and, as you can see in the script, it points to the next url:

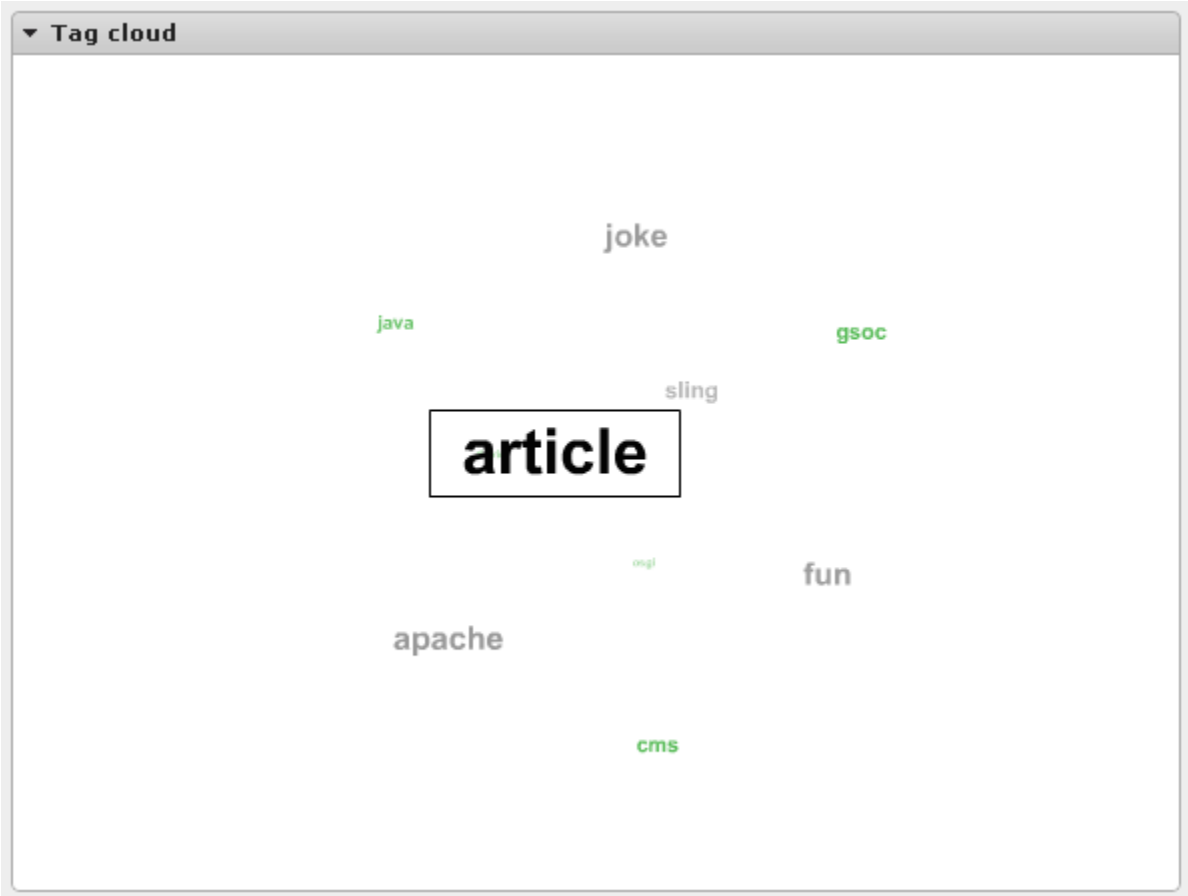
```
so.addVariable("xmlpath", "/content/tag.cloud");
```

To create this XML there is another registered component, which manages the ***cloud*** resource type. This component is sling.gsoc.david.servlet.CloudExtension , which extends SlingAllMethodsServlet to manage this new resource type. In the following code of this component you can see how it is used the @Property annotation to configure it

```
...
...
@Component(metatype = false, immediate = true)
@Service(value = javax.servlet.Servlet.class)
public class CloudExtension extends SlingAllMethodsServlet {

    @Property(value = "PDF Extension Servlet")
    static final String DESCRIPTION = "service.description";
    @Property(value = "David Mini CMS")
    static final String VENDOR = "service.vendor";
    @Property(value = "sling/servlet/default")
    static final String RESOURCE_TYPES = "sling.servlet.resourceTypes";
    @Property(value = "cloud")
    static final String EXTENSIONS = "sling.servlet.extensions";
...
...
```

This component scans the tag structure and creates the XML. The final result is showed in the next image



List entries by tags

When you click on one tag of the previous flash tag cloud, you will land on the page created by /apps/david/taglist.esp script. There using the relationship that there is between a tag and the UUID list of the articles, we have only to execute this simple search to print all the articles with the tag passed as parameter

```
<h3>Entries</h3><h3>Entries</h3>
        <div>
        <%
        var uuids=tagNode.getProperty("UUIDs").getValues();
        for(var i =0; i<uuids.length; i++) {
                var uuid=uuids[i];
                var nodeR=session.getNodeByUUID(uuid.getString());
        %>

                <a href='<%=nodeR.getPath() %>.article'><%=nodeR.title%></a><br>

        <%
        }

        %>
```

You can see in this script as ESP language can be used to manage object related to JCR. Using the method getNodeByUUID of Session we have the node related to a particular UUID, so we can print it in the list.

Search for entries

Also the search is based on ESP script, /apps/david/search.esp . The user can make three different type of search: by title, by text and by tag. The search with title or text are executed using a XPATH query like this

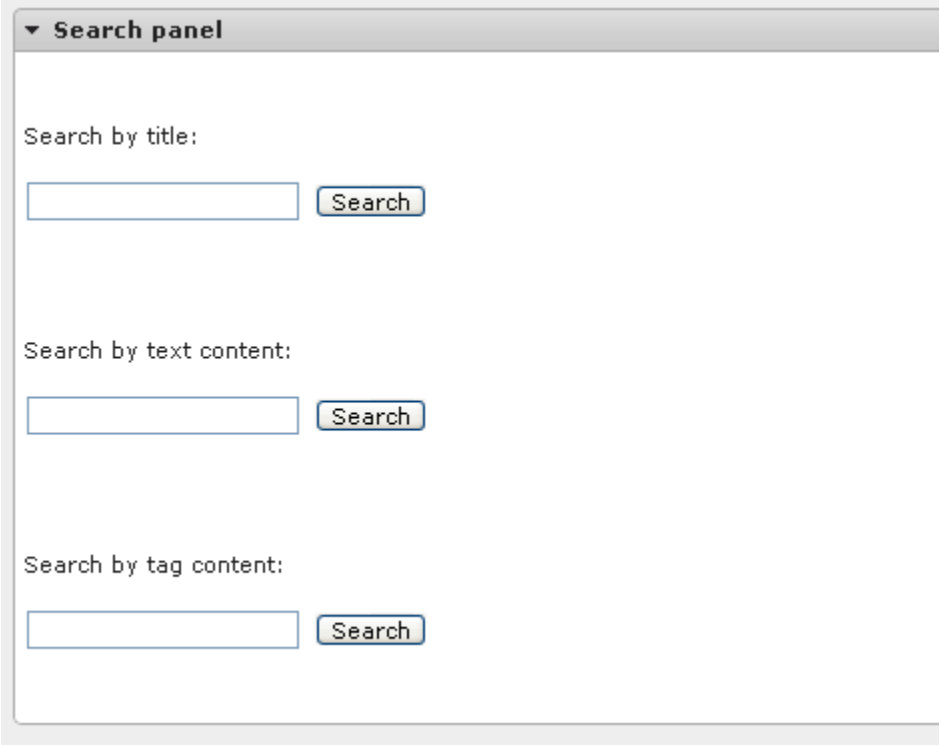
```
var query = queryManager.createQuery(
        "/jcr:root/content/david/*/*/*/element(*, nt:unstructured)[jcr:like(@title, '%"+request.getParameter
("qtitle")+"%')]
        order by @created descending", "xpath");
```

This query searches the entry with title LIKE (as in SQL) the parameter submitted.

The tag search is a bit different, because the structure of tags in David make it possible to search under the root node /content/tags, where we save the tags informations. So the query is like the following one

```
var searchString =
"/jcr:root/content/tags//element(*,nt:unstructured)[fn:name()= '"+request.getParameter("qtag")+"'] order by
@created descending";
```

The initial form to perform the search is a simple HTML one, as you can see in the following image



Create PDF renditions

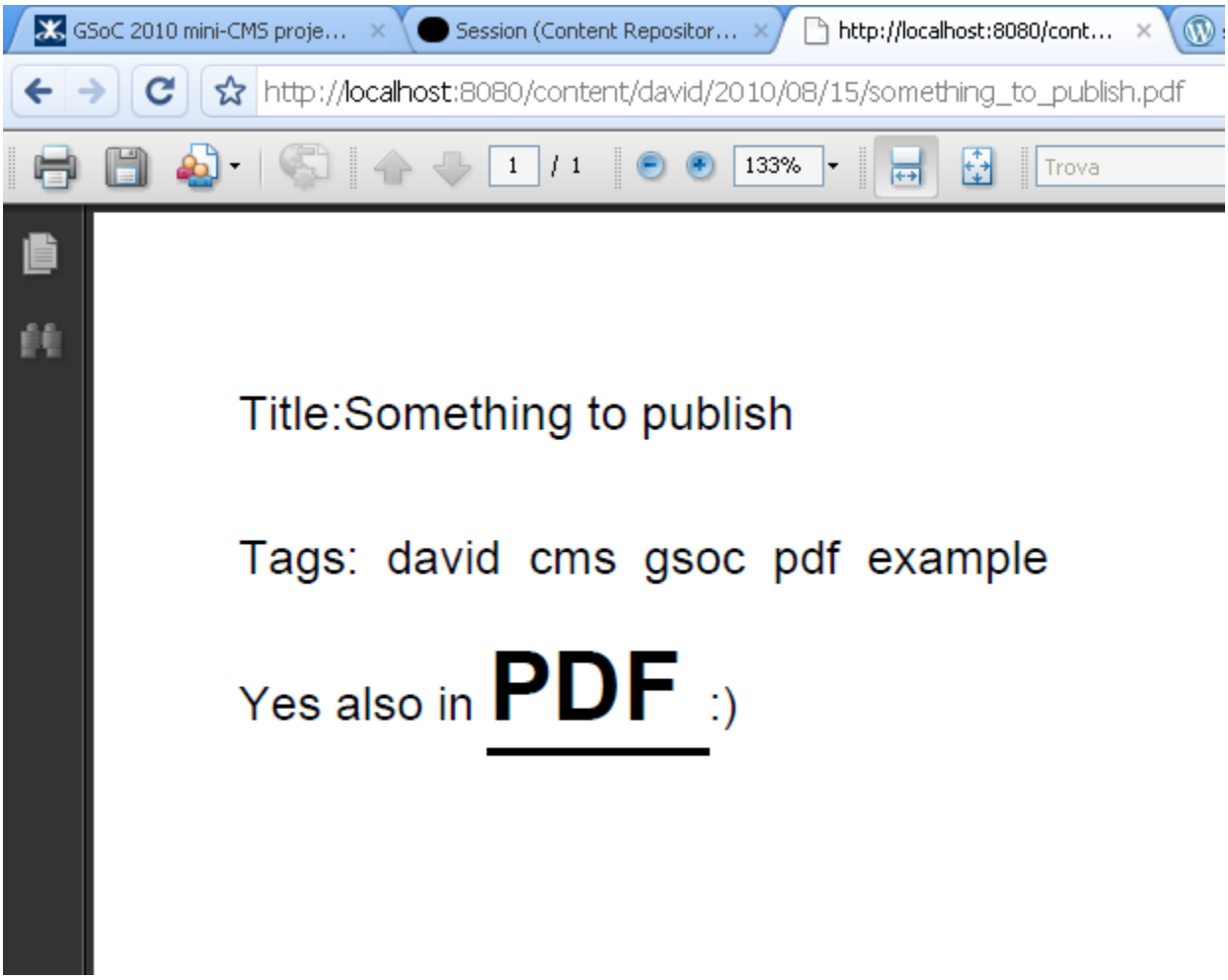
Every node in David is an article and it's an easy task to render it as PDF file.

First of all there is a link on every articles and on the main list that links to

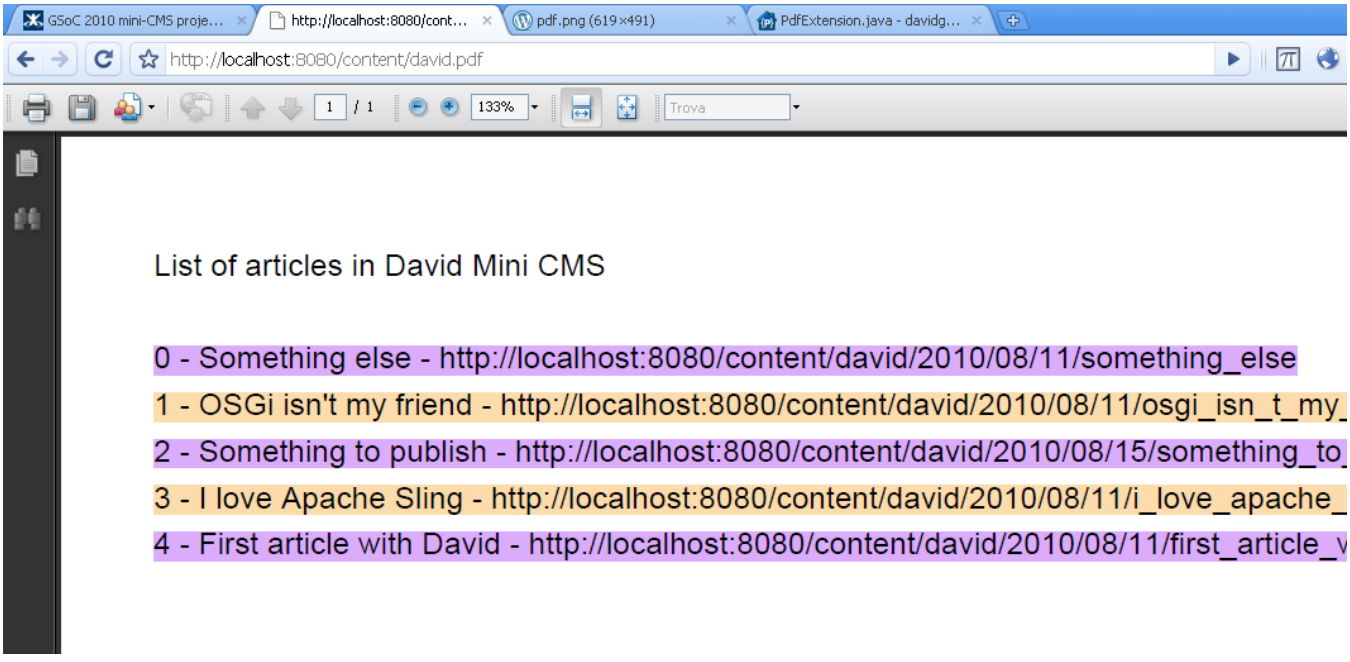
```
http://localhost:8080/content/david/2010/08/14/myarticle.pdf
```

So it's a new extension called for the node. This extension, as we already have seen, can be managed by a Java class defined in our bundle and deployed on Sling. In David the PDF Extension is managed by sling.gsoc.david.servlet.PdfExtension. In this component there is the rendering of the node using iText library and the node, for which this component can be activated, can be a standard article or the root node.

If this extension is used with a standard article, the following image is what we will give



In the other case, we will have a PDF with the list of articles in David.



Scheduler service

Every Sling application can use the opensource scheduler Quartz, provided by the Commons Scheduler bundle (this requires also another bundle, org.apache.sling.commons.threads).

Based on this scheduler there is a simple service, implemented by sling.gsoc.job.JobScheduler and sling.gsoc.job.SendTask. The first is the class that creates a SendTask and schedules it with Quartz. The second one is a simple class that send informations about David (the number of articles and tags available) with an email.