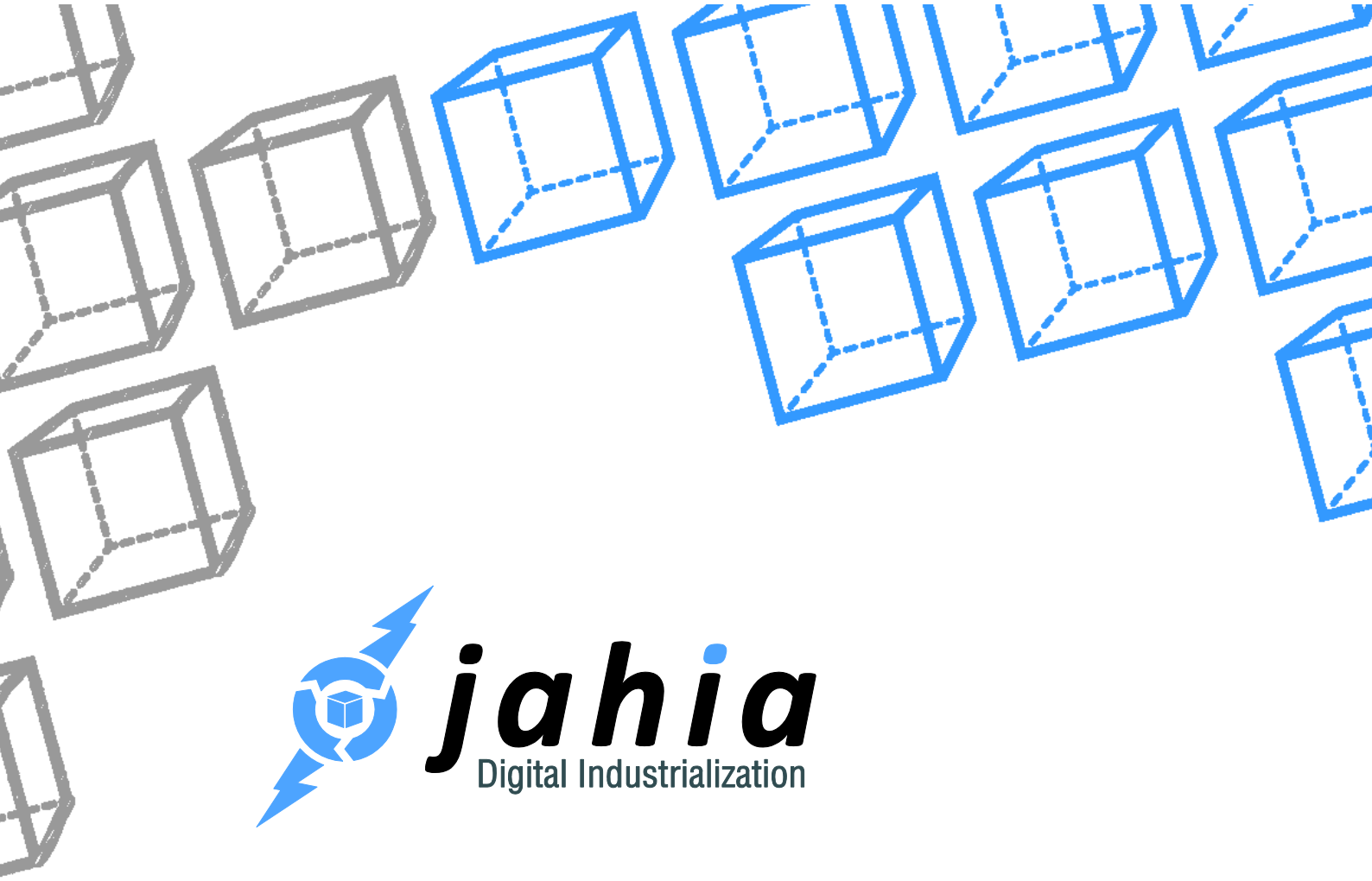


jahia
Digital Industrialization

Form Factory Module Architecture and Technical Design

Rooted in Open Source CMS, Jahia's Digital Industrialization paradigm is about streamlining Enterprise digital projects across channels to truly control time-to-market and TCO, project after project.

Jahia Solutions Group SA

9 route des Jeunes,
CH-1227 Les acacias
Geneva, Switzerland

http://www.jahia.com

Summary

1 Description 4
2 Form Builder 5
2.1 Definitions 5
2.1.1 Node Types 5
2.1.2 Import 9
2.1.3 Service 10
2.1.4 Popover System 11
2.1.5 Backbone Model/Collections/Views 13
2.2 Builder 15
2.2.1 Flow 15
2.2.2 Frameworks and Libraries 18
2.2.3 JCR/JSON Serialization 18
2.2.4 Translation 20
2.2.5 Structure Updates 22
3 Form Display/Submission 23
3.1 Display/Validation 23
3.1.1 Flow 23
3.1.2 Validation 27
3.2 Submission 31
3.2.1 Data Serialization 31

3.2.2 Metadata 31
3.2.3 Actions 32
4 Results Analysis 34
4.1 Action/Result Binding 36
4.1.1 ApiBackendType Interface 36
4.1.2 ResultsProviderService Interface 37
4.1.3 Implementations Retrieval 38
4.1.4 Views 38
4.1.5 API Mapping JSP file 39
4.2 Results API 41
4.3 Views 42
4.3.1 The Metadata View 43
4.3.2 The Datatable View 43
5 Appendices 44
5.1 Form XML Export from JCR 44
5.2 Form Serialized JSON 46

1 Description

This document describes how the Form Center is architectured, from a technical standpoint. It explains which frameworks are used and how they are used, how queries and data are handled, how forms are stored and displayed, how the structure/definitions of the builder are stored and finally how the forms are displayed/validated and the results stored.

2 Form Builder

2.1 Definitions

Definitions are created with two elements:

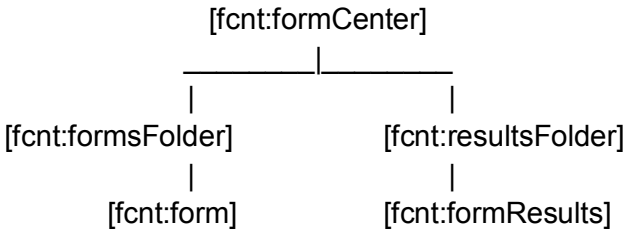
- data, registered in the JCR via the file repository.xml. We serialize the data in JSON and inject them in backbone.js
- templates, rendered from a JSP. We serialize the view in a string, then we use backbone.js to map with the equivalent data and underscore.js to render the template with the data.

On the client side, we use a library inspired from this. This library uses backbone.js and underscore.js to map templates and data under a snippet form.

Mapping between templates and data is done on the client side.

2.1.1 Node Types

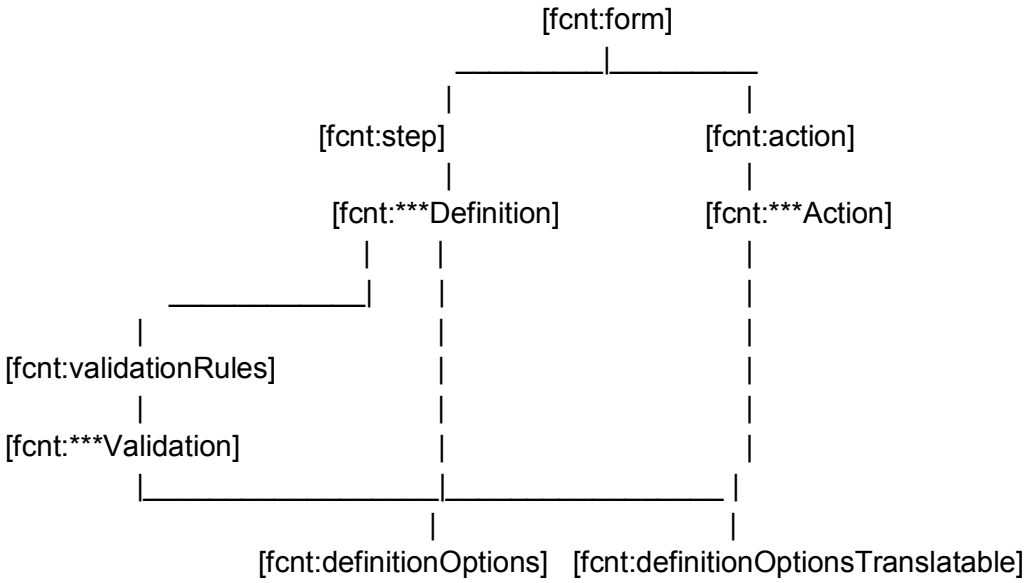
Below is the main structure of the node types tree:



- [fcnt:formCenter]: the main folder at the root of your site, under which you will find two other folders. Its name in the repository is ‘formCenter’.
- [fcnt:formsFolder]: this folder will contain your form. Its name in the repository is ‘forms’.
- [fcnt:resultsFolder]: this folder will contain your form results. Its name in the repository is ‘results’.

- [fcnt:form]: the node type that contains the structure of your form. Its name is based on the name of your form.
- [fcnt:formResults]: the node type that contains the results of your form. Even if you don't use the SaveToJCR Action, some data will be stored in this folder under the name of your form, such as metadata, translation etc. (! this node is present in the live workspace only)

Below is the structure of a form:



Basically, a form node has two subnodes:

- [fcnt:step]: this node type is the representation of a step in your form. It will handle the node type of each field such as [fcnt:***Definition], which is the representation of a field in your step. It contains two different node types:

    - [fcnt:definitionOptions]
    - [fcnt:definitionOptionsTranslatable]

    They have a very similar structure. The difference is that one of them has the jsonValue property i18n and the other does not.

- [fcnt:action]: this node type handles all the actions you have in your form. The structure is similar to [fcnt:step]. It's the same for [fcnt:***Action], which has the same structure as [fcnt:***Definition] and both use the same children: [fcnt:definitionOptions] or [fcnt:definitionOptionsTranslatable].

[fcnt:***Definition]: this node has a particular subnode of the type [fcnt:validationRules], named ‘validations’. The latter node contains node type [fcnt:***Validation] which has exactly the same structure as the node [fcnt:***Definition]. This node is designed to handle the validations you want to apply to your fields.

2.1.2 Import

In the file *repository.xml*, in the **<contents>** tag, which is located inside the **<templates>** tag—because we do not want the content to be duplicated on different sites—, we have added five tags, all with the primary node type *[jnt:contentFolder]*, each one being the representation (value) of a different element:

- **<form-center-definitions>**
  This tag contains the representation of snippet definitions like input, email, checkboxes etc. They will be used to generate snippets.
- **<form-center-validations>**
  This tag contains the representation of validations like range, length etc.
  They will be used in the popovers of a field to generate validation fields.
- **<form-center-actions>**
  This tag contains the representation of actions like *SaveToJCR*, *RedirectTo* etc.
  They will be used to generate action snippets that you can use in your form.
- **<form-center-popovers>**
  This tag contains the representation of popovers like input, checkboxes etc.
  They will be used to create popover fields.
- **<form-center-renderers>**
  This tag contains the representation of renderers like file-renderer, country-renderer etc.
  They will be used to render special fields in the Form Center Result.

To learn how to extend or create these tags, please refer to this document (only available in French at this time).

2.1.3 Service

The only OSGI service associated with the Builder is *FCDefinitionService*.
This service provides the JSON for the snippets (definitions, actions, validations) generated from the JCR repository. It also provides templates for the snippets, retrieved from the associated JSP (definitions, actions, validations, resultsRender, popovers).
We created this service to get data and templates, so that we don't need to regenerate them every time someone accesses the builder. This means that with this service, templates and data are generated only one time per site and language, or when they have changed in the *modules* repository on the platform. The generated result is stored in a cache.

This service is used in three different classes:

- *FCBuilderFlowHandler.java*
  The flow of the builder. When we initialize the flow, we need to get the snippets (definitions, actions, validation and popovers).
- *FCFormFlowHandler.java*
  The flow of the form in live mode. When we initialize the flow in live mode we need to get the snippet definitions only.
- *Functions.java*
  We created a *taglib* to get the renderers for some special fields in the results.

2.1.4 Popover System

All snippets have a popover containing the necessary information to personalize the snippet.
The popover is generated using a main template, which is a JSP file, and a template for each field generated from the snippet definition data. In your file *repository.xml*, when you create your snippet and add a *[fcnt:definitionOptions]* or *[fcnt:definitionOptionsTranslatable]*, you declare a type that can be one of the following:

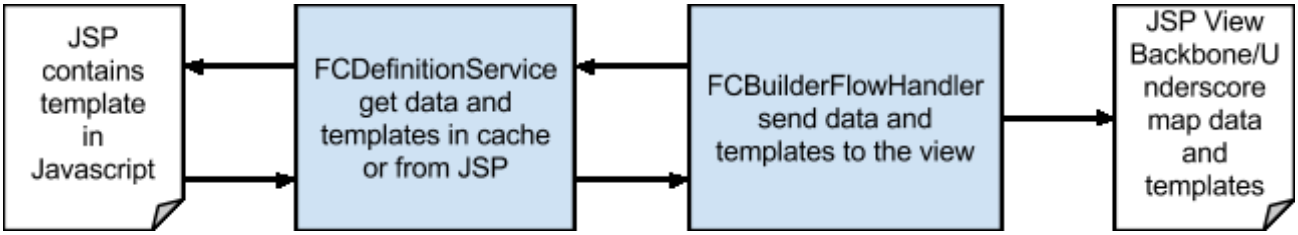
- *checkbox*
- *input*
- *hidden*
- *inputDisabled*
- *key-value* (used for *select*, *select multiple*, *radio*, *checkbox*; offers the possibility to translate the value)
- *pageFinder* (provides typeahead functionality and a tree selector that allow you to select a page of your current site; used in the action *Redirect to a page*)
- *select*
- *textarea*
- *textareaSplit* (used in the action *Send email*; it's a *textarea* in which each line is an email, with lines acting as separators)

This type will be present in the JSON, so *backbone.js* will be able to map the snippet data with the equivalent template. This is done in the file *javascript>views>snippet.js* .
It is possible to create new popover fields, but to do so you will have to add a case to the switch in the file ***my-form-snippet*.js***[1].

[1] This file exists in three different versions: one for the builder, one for the translator and one for live view.

In this file we also do the merge between default data and existing data. For instance, when you load an existing form to create the snippet that is already in the form we merge the original data and the data from the current form.

Templates are stored in a JSP file. The JSP contains HTML and JavaScript variables, which will be interpreted by *underscore.js* (variables are written as <@ ... @> or <@= … @>).



2.1.5 Backbone Model/Collections/Views

Backbone is a JavaScript framework. For further information, please click here.

The Form Center Builder has one Backbone model:

- *snippet.js*
  The model of our snippet (setter, getter etc.).

two collections:

- *snippets*.js*[2]
  The collection of snippets for each tab.
- *my-form-snippets*.js*[2]
  The collection of snippets in the current form.

and six views:

- *my-form*.js*[2]
  The view of the current form, with the actions related to drag and drop and the rendering of the form.
- *my-form-snippet*.js*[2]
  The view of the popover for the current snippet, with all the actions you can perform in the popover.
- *snippet*.js*[2]
  The view of the selected snippet. This is where we add the validations available for the snippet and merge values.
- *tab.js*
  The view of the current tab. This is where we add the snippets collection to the tab.
- *tab-snippet.js*
  The view of the snippet in the tab, render.
- *temp-snippet.js*
  The view of the snippet currently being moved (drag and drop).

Two other files are in use:

- *main*.js*

---

[2] This file exists in three different versions: one for the builder, one for the translator and one for live view.

This file is the initializer of Require, in which we define paths and variables.

- *app*.js*
  This file is the initializer of Backbone, in which tabs and form views are initialized as well as snippets views.

2.2 Builder

2.2.1 Flow

The flow uses:

- one FlowHandler and one (very simple) model used only in the library:

```
<var name="flowHandler"
class="org.jahia.modules.formcenter.webflow.FCBuilderFlowHandler"/>
<var name="formLibrary"
class="org.jahia.modules.formcenter.webflow.model.FCFormLibrary"/>
```

- four views:

```
<view-state id="fcLibrary" model="formLibrary">
</view-state>
```

This view is the main view on which the flow starts. It has its own model *FCFormLibrary.java*, which sets and gets the list of existing forms.

```
<view-state id="fcBuilder" model="currentForm">
</view-state>
```

This view is where we can build a new form or modify an existing form.

```
<view-state id="fcMetaData" model="currentForm">
</view-state>
```

This view offers you the possibility to copy you form in other languages, to add a localized description or title, and/or to add a CSS class to you form if needed.

```
<view-state id="fcTranslate" model="currentForm">
</view-state>
```

This view is very similar to the builder, but here you cannot modify the structure of your form. This view will only display the translatable fields of your form, thus giving you the possibility to translate your form.

- and a group of actions:

```
<action-state id="deleteForm">
    <evaluate
expression="flowHandler.markToDeleteForm(requestParameters.selectedForm,
externalContext.requestMap.renderContext)"/>
    <transition to="fcLibrary"/>
</action-state>

<action-state id="undeleteForm">
    <evaluate expression="flowHandler.undeleteForm(requestParameters.selectedForm,
externalContext.requestMap.renderContext)"/>

      ….
```

When we start the flow, we initialize a number of variables:

```
<on-start>
    <evaluate expression="flowHandler.initFormCenterBuilder()"
result="flowScope.formCenterModel"/>
    <evaluate
expression="flowHandler.getSnippets(externalContext.requestMap.renderContext)"
result="flowScope.snippets"/>
    <evaluate
expression="flowHandler.getValidationMap(externalContext.requestMap.renderContext)"
result="flowScope.validationsData"/>
    <evaluate
expression="flowHandler.getSnippetsAction(externalContext.requestMap.renderContext)"
result="flowScope.actions"/>
    <evaluate
expression="flowHandler.getPopovers(externalContext.requestMap.renderContext)"
result="flowScope.popovers"/>
</on-start>
```

These variables handle data (JSON Object) and templates (String) so that we can use them in the view with Backbone.

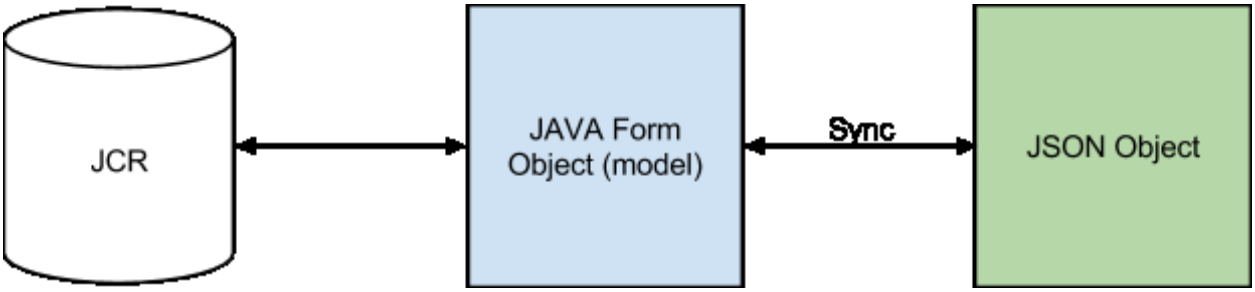
The FlowHandler handles all the methods that are called by the flow, but some methods used by the FlowHandler are externalized in the class *FCFlowUtils.java* .
Basically, all the methods to convert a Form object to a JCR node and vice versa are in this class.

2.2.2 Frameworks and Libraries

The Form Center uses various frameworks, most of which are already in use in Jahia except for Backbone.js. It is used everywhere (results, builder, live) on the client side, but it is new in Jahia. Backbone is present in the Form Center in version 1.1.2. In the builder and in live mode we use Backbone with Require.js version 2.1.1 (Require is a JavaScript file and a module loader), but in the results we use Backbone independently of Require.
We also use Underscore.js with Backbone. It is a JavaScript library that provides a set of useful methods to manipulate data.
Finally, in live mode we use a Backbone module: Backbone.Validation. This module allows us to validate the form and display error messages.

2.2.3 JCR/JSON Serialization

To create the JSON used by Backbone/Underscore, we use Jackson. Jackson is a High-performance JSON processor Java library.
The major part of the form can be serialized/deserialized using standard Jackson methods, but for specific fields we created a serializer (*java.org.jahia.modules.formenter.webflow.model.CustomFieldSerializer*) and a deserializer (*java.org.jahia.modules.formenter.webflow.model.CustomFieldDeserializer*).
The JSON is created from a Java Form Object (model). We keep the model and the JSON synchronized as much as possible. Every time the user performs a webflow action (save, add/delete step etc.), we update the form model (*java.org.jahia.modules.formenter.webflow.model.Form*) with the information provided by the JSON.
The form model is filled with the data from the JCR if it's an existing form, and when the user decides to save the form we update the JCR with the data in the form model.



! The original JSON generated by the service *FCDefinitionService.java* does not use a Java Object; the JSON is created directly from the JCR.

2.2.4 Translation

When you create a new form, a property is set with your current language in order to have a fallback language. So when you create a form, if it is not intended for your current language, you will have to copy it in the desired language.
To translate your form, you first need to copy it to your target language. To do so, go in the Form Center Library and in the actions menu of the form that you want to translate, click on 'Edit metadata'. Then, select one of the languages available on your site and 'Copy' the form.
Now switch to edit mode in the relevant language: the main action is now 'Translate' instead of 'Modify' in the Form Center Library.
The Form Center Translator interface is similar to the Form Center Builder except that you cannot change the structure of your form, but for all the fields in your form you can open the popover and make modifications.

As explained in the Popover System chapter:
The popover is generated with a main template and a template for each field generated from the snippet definition data.

When you create your snippet and add fields, you have two possibilities:

- *[fcnt:definitionOptions]*
- *[fcnt:definitionOptionsTranslatable]*

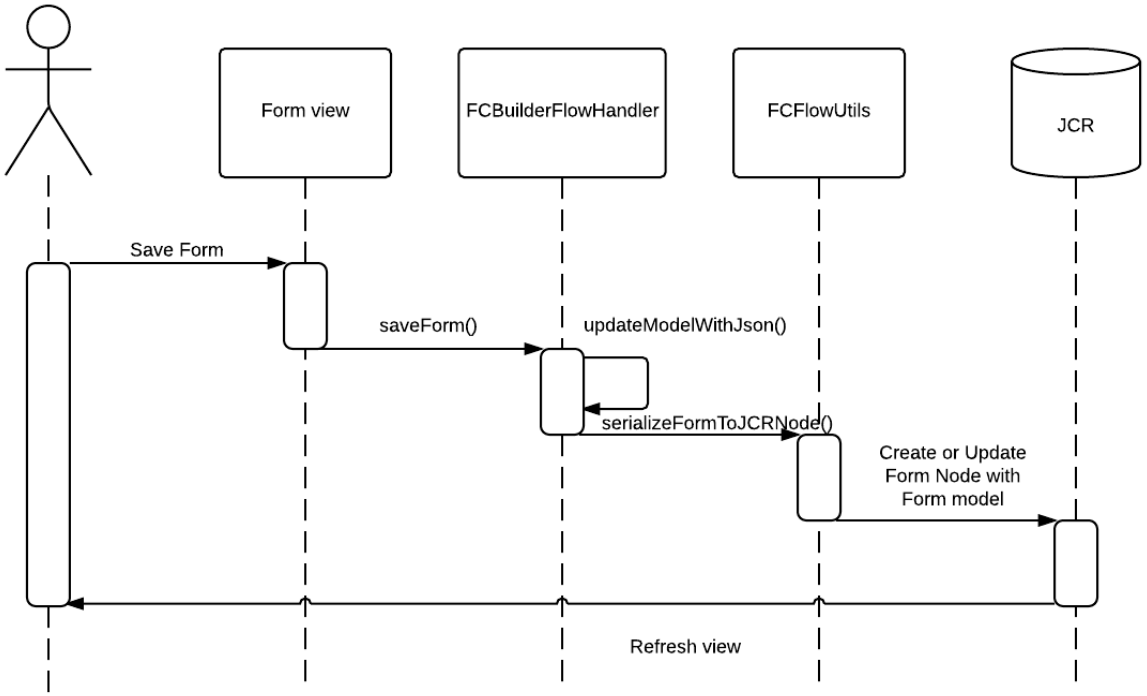
In the Form Center Translator, only the [fcnt:definitionOptionsTranslatable] will be displayed in the popover and therefore be translatable, because in the JSON (as shown below) this field will have the key "translatable" equal to *true*, so that Backbone can know if this field must be displayed or not.
This action is performed in the file *javascript>views>snippet-translate.js*

```
"label": {
      "type": "input",
```

```
    "label": "Label Text",
    "value": "Text Area",
    "translatable": true
  }
```

2.2.5 Structure Updates

To update an existing form, we will use its JSON model. This operation is performed when the user chooses to save. This means that if you create a new form but do not perform a saving action, your form will not be saved in the JCR.



As you can see, the user starts the Action ‘Save Form’. The view will send the form in JSON generated from the Backbone collection.
Then the JSON will be deserialized and the form model will be updated with the data.
At the end, if the form doesn’t exist, a new node will be created based on the form model. But if a node already exists, then the form model and the node will be merged, which means only the differences will be saved.

3 Form Display/Submission

3.1 Display/Validation

3.1.1 Flow

At the beginning of the flow, we declare the flow handler and initialize two variables:

```
<var name="flowHandler"
class="org.jahia.modules.formcenter.webflow.FCFormFlowHandler"/>

<on-start>
    <evaluate
expression="flowHandler.initForm(externalContext.requestMap.currentResource)
" result="flowScope.currentForm"/>
    <evaluate
expression="flowHandler.getSnippets(externalContext.requestMap.renderContext
)" result="flowScope.snippets"/>
</on-start>
```

currentForm is the form model we want to display. It contains the JSON of the form that will be used by Backbone.
snippets contains the data and templates of the snippets. We don't need the data here—we will use the data contained in the form model—but we do need the templates to be used by Backbone.

The flow uses two views, but only one of them does the entire job; the other one is used to display the results of the actions (error, success etc.).

```
<view-state id="viewFormStep">
    ....
</view-state>

<view-state id="formSubmitted"/>
```

viewFormStep is the main view. Everything is done here; this view handles the actions and will display all the steps of the form (see details below).

formSubmitted will, as mentioned above, display the results of the actions, but only if they have no redirection action at the end of the form.

Details of viewFormStep:

```
<view-state id="viewFormStep">
    <transition on="next" to="viewFormStep">
        <evaluate expression="flowHandler.saveValues(currentForm, requestParameters.formData)"/>
        <evaluate expression="flowHandler.goToNextStep(currentForm)"/>
    </transition>

    <transition on="previous" to="viewFormStep">
        <evaluate expression="flowHandler.saveValues(currentForm, requestParameters.formData)"/>
        <evaluate expression="flowHandler.backToPreviousStep(currentForm)"/>
    </transition>

    <transition on="submit" to="validTokenAndCaptcha">
        <evaluate expression="flowHandler.saveValues(currentForm, requestParameters.formData)"/>
    </transition>

    <transition on="cancel" to="cancel"/>
</view-state>

<action-state id="validTokenAndCaptcha">
    <evaluate expression="flowHandler.validTokenAndCaptcha(currentForm, externalContext.sessionMap['form-
tokens'], requestParameters['form-token'], requestParameters.captcha,
externalContext.requestMap.renderContext, messageContext)"/>
    <transition on="yes" to="finish"/>
    <transition on="no" to="viewFormStep"/>
</action-state>

<action-state id="finish">
    <evaluate expression="flowHandler.executeActions(externalContext.nativeRequest,
        externalContext.requestMap.renderContext, externalContext.requestMap.currentResource, currentForm)"/>
    <transition on="ok" to="formSubmitted" />
    <transition on="ko" to="cancel" />
</action-state>
```

As you can see the view has three main actions:

- next
- previous
- submit

All of them call the same method, *saveValues*. This method will get the data of the current step—which were serialized by Backbone and put in a hidden input named *formData*—and put them in a map that handles the results of the entire step.

next and **previous** are calling a method that increments/decrements the number of the current step, after the view is reloaded by webflow with the current step.

submit calls another action to check the token and, if the user is a guest, check the captcha. Once the token and captcha are checked, if they are valid the transition goes to finish, where the method *executeActions* is called.

executeActions will get the actions of the forms and execute all of them. If one of them is a redirection action, the last view won't be used.

3.1.2 Validation

Validation of the form is done on the client side by a Backbone module. More information is available on the project's site (Backbone.Validation) and also in this document (only available in French at this time).

However, our integration has its own specificities.

In order to work, the validation framework needs a JSON like this one:

```
validation: {
   name: {
     required: true
   },
   'address.street': {
     required: true
   },
   'address.zip': {
     length: 4
   },
   age: {
     range: [1, 80]
   },
   email: {
     pattern: 'email'
   },
   someAttribute: function(value) {
     if(value !== 'somevalue') {
       return 'Error message';
     }
   }
 }
```

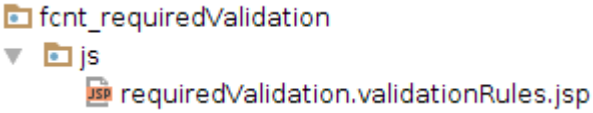
To generate this JSON, we use the data from the current form—each field handles its own validation data—and when the *viewFormStep* is initialized, we initialize this variable in the **<script>** tag:

```
var formValidation = <template:include view="validationRules" templateType="js" />;
```

This will call the view *fcnt_form>js>form.validationRules.jsp*, which contains JavaScript:

```
[
<c:forEach items="${jcr:getChildrenOfType(currentNode, 'fcnt:step')}" var="formStep"
varStatus="status">
    {
        <template:module node="${formStep}" view="validationRules" templateType="js"
/>
        <c:if test="${not renderContext.loggedIn and status.last}">
            <fmt:message key="fcmix_validation.error.captchaRequired"
var="errorCaptchaRequired"/>
            <c:set var="errorCaptchaRequired"
value="${functions:escapeJavaScript(errorCaptchaRequired)}"/>
            captcha:[{
                        required : true,
                        msg : 'errorCaptchaRequired'
                    }]
        </c:if>
    }<c:if test="${not status.last}">,</c:if>
</c:forEach>
]
```

And this view will call the other view—similar to this one—that is needed to generate the JSON, like for the required validation:




```
<jcr:node path="${currentNode.path}/message" var="nodeRequiredMessage"/>

<c:set var="requiredMessage"
value="${nodeRequiredMessage.properties['jsonValue'].string}"/>

{
    required: true
    <c:if test="${not empty requiredMessage}">
        ,msg: '${functions:escapeJavaScript(requiredMessage)}'
    </c:if>
}
```

The result will be a JSON like the one shown at the beginning of this chapter.

3.2 Submission

3.2.1 Data Serialization

When you submit a form, the associated data are serialized on the client side in a JSON. This serialization results in a JSON like this one:

```
{
  "textinput_0_1": "rewer",
  "multiplecheckboxesinline_0_3": [
    "2",
    "4"
  ],
  "emailinput_0_2": "rewrewrew"
}
```

Basically it is a *key : value* JSON, where the key is the name that is the id of your snippet. If it is a block, the name will be something like:

- idOfTheFieldInTheBlock_block_idOfTheBlock
  This will be used to identify a field from the block when we will rewrite the map.

Then we put the data in a hidden field in order to get this field on the server side.

On the server side, we deserialize the JSON and put the data in a map, but if the form contains a block we need to reformat the data. To do so, when the method *executeActions* is called by the flow—i.e., when the form is submitted—we call a method named *rewriteMapResultIfBlock*. This method will parse the form and if the form contains a block, the method will parse the data and group them by block, which will allow us to keep the form structure and data bound.

3.2.2 Metadata

Each time a form is submitted we save in the JCR the structure and the metadata related to this form, even if your form does not have a save in JCR action and stores results in an external system or even just implement an action like a redirect.
Those information are saved under the node [fcnt:formResults] (see Node Types) who have the name of your form in the JCR repository.

So What we save are :

- metadata (created date, modified date, …)
- translation of the form
- labels with their translation
- actions

3.2.3 Actions

The actions of the form are Java Action classes. We call them when we are in the *executeActions* method. We get the names of the actions in the current form and iterate over the associated actions, which we get using *TemplateManagerService* as follows:

```
Action action = templateManagerService.getActions().get(fieldOption.getValue().toString());
```

Then in *executeActions* we call this method with each action:

```
/**
 * This function calls the selected action
 * @param action            : current Action
 * @param currentForm       : current model
 * @return ActionResult
 */
 private ActionResult callAction(HttpServletRequest request, RenderContext renderContext,
                                 Resource resource, Action action, Form currentForm) {
     URLResolver mainResolver = (URLResolver) request.getAttribute("urlResolver");
     String urlPathInfo = StringUtils.substringBefore(mainResolver.getUrlPathInfo(),
mainResolver.getPath())
                                                 + resource.getNode().getPath();
     urlPathInfo += "/*";
     URLResolverFactory f = (URLResolverFactory) SpringContextSingleton.getBean("urlResolverFactory");
     URLResolver resolver = f.createURLResolver(urlPathInfo, request.getServerName(), request);
     final Action originalAction = action;
     try {
        action = new SystemAction() {
            @Override
             public ActionResult doExecuteAsSystem(HttpServletRequest request, RenderContext
renderContext,
                                                   JCRSessionWrapper systemSession, Resource resource,
                                                   Map<String, List<String>> parameters,
                                                   URLResolver urlResolver) throws Exception {
                return originalAction.doExecute(request, renderContext, resource, systemSession,
                                                 parameters, urlResolver);
            }
         };
```

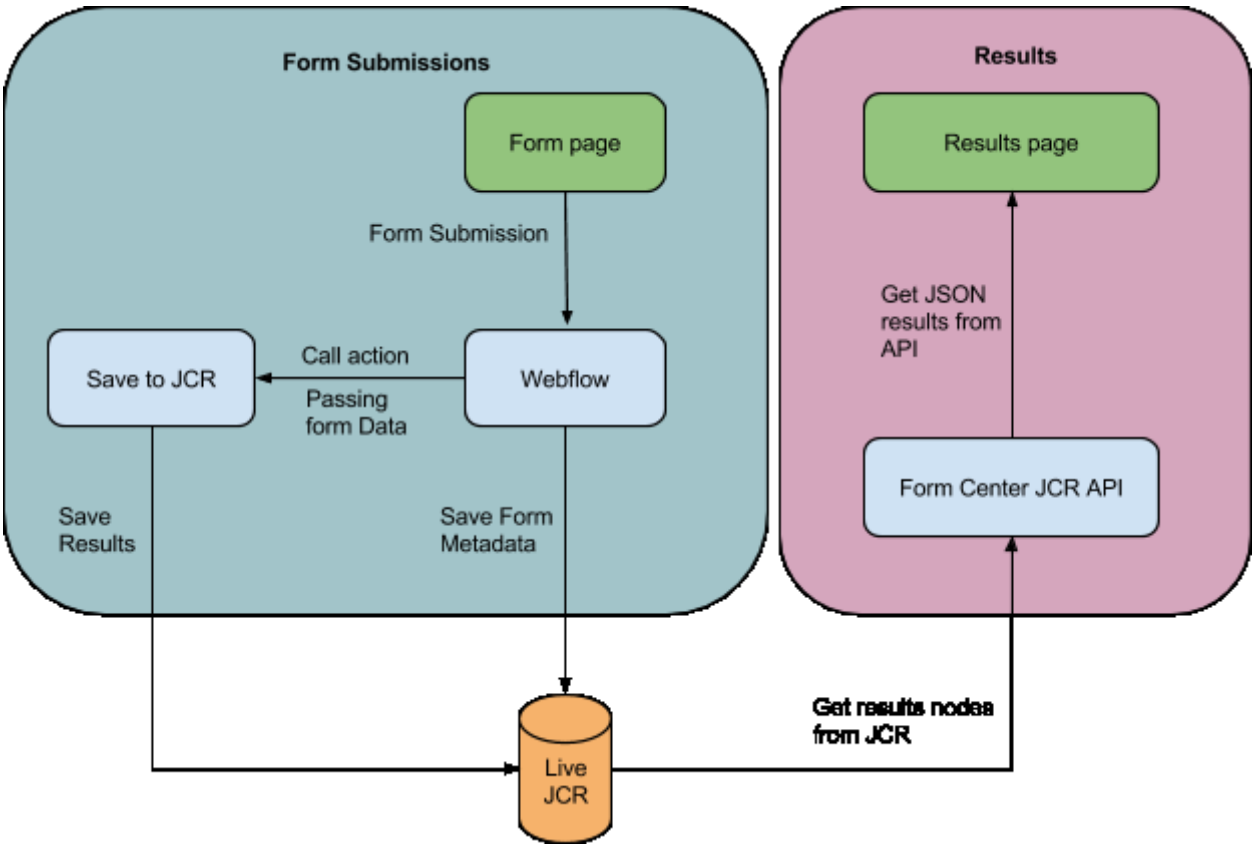
```
            return action.doExecute(request, renderContext, resource,
                                    renderContext.getMainResource().getNode().getSession(),
                                    currentForm.getMapResult(), resolver);
        } catch (Exception e) {
            logger.error("Error executing action", e);
        }
        return null;
    }
```

After that, we store the *ActionResult* of each action in the form model, and if there is no redirect action in the form, we display the result on the view *formSubmitted*.

4 Results Analysis

Form submissions are saved in the live workspace, under the folder named ‘formCenter/results’. In any case, whatever the destination of the form submitted data, the form metadata are saved in this folder. The actual data are only saved to the JCR when the form contains the action ‘Save to JCR’.

The following diagram describes how the results interact with other parts of the *fc-form-center* module.



We can see that the results are fetched from the JCR directly by the API that returns JSON objects to pages. The structure of the returned JSON object is:

```
{
  "status" : <"success" || "error">,
  "code" : <HTML CODE>,
```

```
  "message" : <String Message>,
  "data" : <Data Object>
}
```

In the object, the results are wrapped under the ‘data’ key.

4.1 Action/Result Binding

In the Form Center, actions must be mapped to result display methods in order to be able to fetch view data from the correct source (JCR, Database etc.).
Form Center action binding is based on four entities:

- Backend type: a constant to define where the results are saved.
- API: an API name (this represents how results can be fetched).
- Mapping view: a file that is defined for each API/Backend couple, returning the API entry points whenever the view is requested.
- Views: the views to display results fetched from a given API to the user.

To know which API should be mapped on each action, the form center monitors *ResultsProviderService* implementations, which map lists of APIs to Backend types. Since all actions declare a Backend type when implementing the *ApiBackendType* interface, *SaveToJCRResultAPI* gets the Backend type for each action and monitors the *ResultsProviderService* implementation with this *backendType*. Then, it gets the corresponding list of APIs and from there it can access all the views of the APIs.

4.1.1ApiBackendType Interface

The action class implements the *ApiBackendType* interface in order to declare the Action Backend type.

org.jahia.modules.formcenter.actions.SaveToJCRAction ApiBackendType implementation:

```
public class SaveToJcrAction extends Action implements ApiBackendType {

        …
```

```
        public String getBackendType(){
        return "JCR";
        }
       …
```

This Action Backend type is a String constant that must be reused in the class that will implement *ResultsProviderService* to map the backend to the right API.

4.1.2 ResultsProviderService Interface

The *ResultsProviderService* interface is implemented for each Action/API couple in order to map an API to a given Backend type. For the action *SaveToJCRAction*, the class *org.jahia.modules.formcenter.impl.SaveToJcrRawResultsImpl* is doing the mapping:

```
public class SaveToJcrRawResultsImpl implements ResultsProviderService {
        @Override
        public String getApiNames() {
            List<String> apiNames = new ArrayList<String>();
           apiNames.add("rawResults")
           return apiNames;
        }

        @Override
        public String getBackendType() {
        return "JCR";
        }
}
```

We can see here that the Backend type is the same as the one declared in the class *SaveToJCRAction*.

4.1.3 Implementations Retrieval

The function *getViewsByAction* of *SaveToJCRResultAPI.java* returns all the views and the APIs by action. To do so, it gets the list of actions in the form saved in the JCR, and from this list it gets all the APIs by action:

```
//Putting in the map all the views for each [backend,Api] couple
for (ResultsProviderService providerService : resultsProviderServices) {
    String providerBackEndType = providerService.getBackendType();
    if (providerBackEndType.equals(((ApiBackendType) action).getBackendType())) {
        List<Map<String, String>> prefixedViews = new ArrayList<Map<String, String>>();
        for(String apiName : providerService.getApiNames()){
            prefixedViews.addAll(prefixViews(providerBackEndType, viewMap.get(apiName)));
        }
        actionViewMap.put("views", prefixedViews);
    }
}
```

Then, it gets the list of views for each API.
Finally, it returns the list of views for each action.

4.1.4 Views

Views are linked to APIs because they have to submit to the format of the returned results. To declare this link, each view name must contain the name of the API. A results view is a view of the submissions node (of the type *fcnt:submissions*) and its name must be formed as follows:

submissions.<apiName>.<viewName>

A resource bundle property is used to properly display this view name to a user. This property key must be defined as follows:

fcnt_submissions.viewName.<ApiName>.<viewName>

For example, the the *datatable* view of the *rawResults* API will be named:

submissions.rawResults.dataTable

And its property key will be:

fcnt_submissions.viewName.rawResults.dashboard

4.1.5 API Mapping JSP file

The last link between the actions and the API is the JSP file that maps the view fetch keys to the API entry points. The different views that display results call the *getFetchURL* function:

```
//Generate a URL from parameters
 function getFetchURL(urlParameters) {
        return apiMethods[urlParameters.apiMethodName](urlParameters);
 }
```

This function takes the following object as parameter:

```
{
    formId:this.model.formId,
    apiMethodName:"results_submissiontime",
    fromDate: this.model.fromDate,
    toDate: this.model.toDate
}
```

The parameter *apiMethodName* is a constant used to map API entry points thanks to the object *apiMethods*:

```
{
        "results":function(parameters){
                if(parameters.fromDate!=null && parameters.toDate!=null){
                return apiURLBase + parameters.formId + "/from/" +
parameters.fromDate + "/to/" + parameters.toDate + "/results";
                }
                else{
                return apiURLBase + parameters.formId + "/results";
                }
        },
        "results_submissiontime":function(parameters){
                return apiURLBase + parameters.formId + "/results/submissiontime";
        },
        "results_submissionpage":function(parameters){
                return apiURLBase + parameters.formId + "/results/submissionpage";
        },
```

```
        "results_submissionempty":function(parameters){
                return apiURLBase + parameters.formId + "/results/submissionempty";
        },
        "results_choicelabel":function(parameters){
                return apiURLBase + parameters.formId + "/results/choicelabel";
        },
        "results_label":function(parameters){
                return apiURLBase + parameters.formId+"/results/labels";
        },
        "results_choice":function(parameters){
                return apiURLBase +
parameters.formId+"/results/choice/"+parameters.choiceId;
        },
        "total":function(parameters){
                return
"${url.context}/modules/formcenter/results/total/"+parameters.formId;
        },
        "lastDays":function(parameters){
                return
"${url.context}/modules/formcenter/results/${renderContext.UILocale.language}/totall
astdays/"+parameters.formId;
        },
        "groupTotal":function(parameters){
                return
"${url.context}/modules/formcenter/results/${renderContext.UILocale.language}/groupT
otal/"+parameters.formId;
        }
}
```

This object maps the different String keys to API get URLs.

This system makes it possible to:

- create new views for each API in separated modules (by naming the views the right way);
- map a Backend type to new APIs in any separated module (implementing *ResultsProviderService*);
- create new APIs for a given Backend type in any separated module (implementing the API and *ResultsProviderService*).

4.2 Results API

The results API is developed using JAX-RS. The source of different API functions (all HTTP GET methods) is located in *org.jahia.modules.formcenter.api.SaveToJCRResultsApi*.
This class uses various Spring injected variables:

- **renderService**: Jahia *RenderService* is used to get views of submissions node
- **repository**: *jcrSessionFactory* is used to get the current session
- **resultsProviderServices**: the list of implementations of the *ResultsProviderService* interface.
- **templateManagerService**: Jahia *TemplateManagerService* is used to get the actions available on the server.

These injections are made by the constructor called in *org.jahia.modules.formcenter.api.ResultApiApplication* with variables from *org.jahia.modules.formcenter.api.SpringBeanAccess* injected by the Spring file *df-form-center.xml*.

The API class contains a certain number of methods (all HTTP GET) that return JSON objects. Some of these methods will always be used even when the form results are not saved in the JCR:

- *getViewsByActions*: returns an object containing the views for each action based on the existing Form Center actions.
  Example:
  {<actionname>:{<view1Name>:<view1Label>, <view2Name>:<view2Label>}}
  [{"views":[{"viewName":<viewName>,"viewLabel":<viewLabel>}],"actionName":<actionName>}]

  Important note:
  The view name is written in the form *<apiName>.<viewName>*, which means that the name of the *dashboard* view of the *rawResults* API will be *rawResults.dashboard*.

The *viewLabel* is a resource bundle with a key written in the form *fcnt_submissions.viewName.<apiName>.<viewName>*, which means that the resource bundle key for the *dashboard* view of the *RawResult* API is *fcnt_submissions.viewName.rawResults.dashboard*.

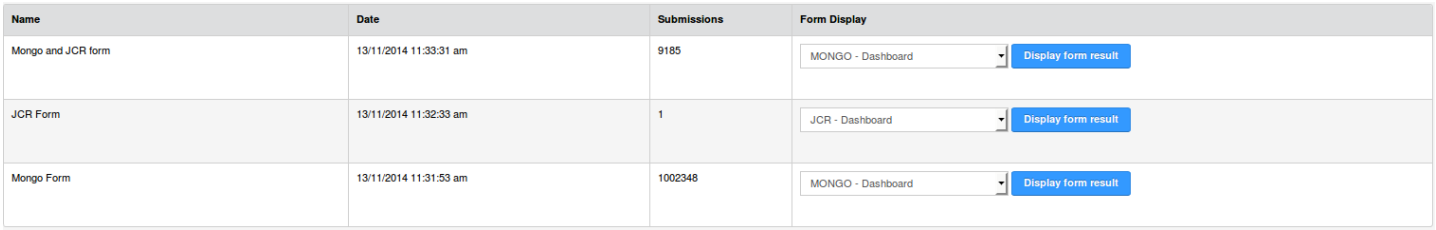
- *getActionsProviders*: returns the list of action ids in an array.
  Example:
  [{id:<actionId>, name:<actionName> }]

- *getFormsDetails*: returns the details (metadata) from a given form in an object.
  Example:

```
[{
        formId:<StructureFormId>,
        resultFormId:<ResultFormId>,
        created:<formCreationDate>,
        lastModified:<formLastModificationDate>,
        formName:<formName>,
        actions:[<actions names>],
        entryPoints:[{action:<actionName>, entryPoint:<apiEntryPoint>}]
}]
```

4.3 Views

The results section contains multiple pages. The first page (which we can call 'results home page') displays the list of forms that have at least one result already saved:



The JCR Results API (*org.jahia.modules.formcenter.api.SaveToJCRResultsApi*) is called to fill this table.

The function *getViewsByActions* is called with the API entry point '/views' to fill the selection view of the Form Display column.
The Name and Date columns are filled by the *getFormsDetails* function.

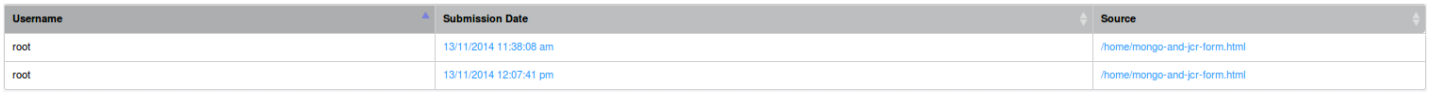
This table is displayed using a Backbone view (*FormsResultsView*) based on a collection (*FormsCollection*). All Backbone view definitions are located in the *main-results.js* file.

Each table line contains a form that is submitted to the same page. When the page loads, if a form and a view have been submitted, the form results are displayed in the selected view.

The form center comes with the *RawResult* API and its JCR implementation. The *rawResult* API contains 3 views:

4.3.1 The Metadata View

The *metadata* view is designed to display only metadata of the submitted results (source, user and date) and its code is located in the file *submissions.rawResults.metadata.jsp*.



It uses the Backbone view *MetadataResultsView,* which initializes a Datatable fetching a backbone model. In each line, the Submission Date value is clickable and displays the submitted result in a popover that is defined by the *displayData* function of the jsp file.

4.3.2 The Datatable View

The *datatable* view renders results in a jquery datatable.
The Backbone model used is a pageable collection called *PageableResults* from the *Backbone.Paginator* framework (https://github.com/backbone-paginator/backbone.paginator) defined in the *results.js* file under the 'collections' JavaScript folder.
The view implemented to display this collection is *PageableResultsView* (from *main-results.js).*
The view defines a datatable with unlimited scroll using the Scroller datatable extension (http://datatables.net/extensions/scroller/).
The column labels are requested from the server by the JavaScript function *getLabelsFromResults* located in the *fcResultsUtils.js* file.
This function calls the RawResultApi via the entry point.

5 Appendices

5.1 Form XML Export from JCR

```
<?xml version="1.0" encoding="UTF-8"?>
<formNode buildingLang="en" primaryType="fcnt:form" numberOfSteps="1">
        <translation_en language="en" title="test"/>
        <step1 primaryType="fcnt:step" stepNumber="0">
                <translation_en language="en" primaryType="jnt:translation"
title="Step 1"/>
                <textinput_0_1 choiceField="" primaryType="fcnt:inputDefinition"
supportedValidationTypes="equal length number regex required" tab="input">
                        <translation_en language="en" primaryType="jnt:translation"
title="Text Input" label="ID / Name"/>
                        <inputsize primaryType="fcnt:definitionOptions"
jsonValue="JSON OBJECT" type="select">
                                <translation_en language="en"
primaryType="jnt:translation" label="Input Size"/>
                        </inputsize>
                        <placeholder primaryType="fcnt:definitionOptionsTranslatable"
type="input">
                                <translation_en language="en"
primaryType="jnt:translation" jsonValue="placeholder" label="Placeholder"/>
                        </placeholder>
                        <label primaryType="fcnt:definitionOptionsTranslatable"
type="input">
                                <translation_en language="en"
primaryType="jnt:translation" jsonValue="Text" label="Label Text"/>
                        </label>
                        <helptext primaryType="fcnt:definitionOptionsTranslatable"
type="input">
                                <translation_en language="en"
primaryType="jnt:translation" jsonValue="help" label="Help Text"/>
                        </helptext>
                </textinput_0_1>
                <singlebutton choiceField="" primaryType="fcnt:buttonDefinition"
tab="input">
                        <translation_en language="en" primaryType="jnt:translation"
title="Single Button" label="ID / Name"/>
                        <buttoneventid primaryType="fcnt:definitionOptions"
jsonValue="_eventId_submit" type="hidden">
                                <translation_en language="en"
primaryType="jnt:translation" label="Button action type"/>
                        </buttoneventid>
                        <buttonsize primaryType="fcnt:definitionOptions"
jsonValue="JSON OBJECT" type="select">
                                <translation_en language="en"
primaryType="jnt:translation" label="Button Size"/>
                        </buttonsize>
                        <buttontype primaryType="fcnt:definitionOptions"
jsonValue="JSON OBJECT" type="select">
                                <translation_en language="en"
primaryType="jnt:translation" label="Button Type"/>
                        </buttontype>
                        <buttonlabel primaryType="fcnt:definitionOptionsTranslatable"
type="input">
```

```
                                <translation_en language="en"
primaryType="jnt:translation" jsonValue="Submit" label="Button Label"/>
                            </buttonlabel>
                            <buttonglyphiconswhite primaryType="fcnt:definitionOptions"
jsonValue="true" type="checkbox">
                                <translation_en language="en"
primaryType="jnt:translation" label="Button Glyphicons White"/>
                            </buttonglyphiconswhite>
                            <btnblock primaryType="fcnt:definitionOptions"
jsonValue="false" type="checkbox">
                                <translation_en language="en"
primaryType="jnt:translation" label="Button block"/>
                            </btnblock>
                    </singlebutton>
            </step1>
            <actions primaryType="fcnt:action">
                    <savetojcr_stepAction_0 primaryType="fcnt:saveToJcrAction"
tab="input">
                            <translation_en language="en" primaryType="jnt:translation"
title="Save to JCR" label="ID / Name"/>
                            <apiEntryPoint primaryType="fcnt:definitionOptions"
jsonValue="/modules/formcenter/results" type="hidden">
                                <translation_en language="en"
primaryType="jnt:translation" label="Api Entry Point"/>
                            </apiEntryPoint>
                            <label primaryType="fcnt:definitionOptions" jsonValue="Save
to JCR" type="inputdisabled">
                                <translation_en language="en"
primaryType="jnt:translation" label="Label Action"/>
                            </label>
                            <actionname primaryType="fcnt:definitionOptions"
jsonValue="savetojcr" type="inputdisabled">
                                <translation_en language="en"
primaryType="jnt:translation" label="Action Name"/>
                            </actionname>
                    </savetojcr_stepAction_0>
            </actions>
</formNode>
```

5.2 Form Serialized JSON


```
{
  "buildingLang": "en",
  "formName": "formNode",
  "formSteps": [
    {
      "stepNumber": 0,
      "stepContent": [
        {
          "title": "Text Input",
          "nodeType": "fcnt:inputDefinition",
          "fieldJcrId": "f3f37953-aa16-4961-9fad-4c2d39a6d57c",
          "choiceField": "",
          "blockInput": false,
          "supportedValidationTypes": [
            "required",
            "regex",
            "number",
            "length",
            "equal"
          ],
          "fields": {
            "id": {
              "optionJcrId": null,
              "type": "input",
              "label": "ID / Name",
              "value": "textinput_0_1",
              "translatable": false
            },
            "inputsize": {
              "optionJcrId": "10dc558b-0050-44a3-bcff-94fa96ed8a10",
              "type": "select",
              "label": "Input Size",
              "value": [
                {
                  "label": "Mini",
                  "value": "input-mini",
                  "selected": false
                },
                {
                  "label": "Small",
                  "value": "input-small",
                  "selected": false
                },
                {
                  "label": "Medium",
                  "value": "input-medium",
                  "selected": false
                },
                {
                  "label": "Large",
                  "value": "input-large",
                  "selected": true
                },
```

```
            {
              "label": "Xlarge",
              "value": "input-xlarge",
              "selected": false
            },
            {
              "label": "Xxlarge",
              "value": "input-xxlarge",
              "selected": false
            }
          ],
          "translatable": false
        },
        "placeholder": {
          "optionJcrId": "4c2efecf-ebf2-4e51-9693-80943c5c3b01",
          "type": "input",
          "label": "Placeholder",
          "value": "placeholder",
          "translatable": true
        },
        "label": {
          "optionJcrId": "1868ab7f-dd50-4632-8926-7992d5395153",
          "type": "input",
          "label": "Label Text",
          "value": "Text",
          "translatable": true
        },
        "helptext": {
          "optionJcrId": "4e938bf1-2088-491e-9ef5-cfe4bf665aa6",
          "type": "input",
          "label": "Help Text",
          "value": "help",
          "translatable": true
        }
      }
    },
    {
      "title": "Single Button",
      "nodeType": "fcnt:buttonDefinition",
      "fieldJcrId": "853a4fb0-d6aa-4ad6-84fa-c363345722c1",
      "choiceField": "",
      "blockInput": false,
      "fields": {
        "id": {
          "optionJcrId": null,
          "type": "input",
          "label": "ID / Name",
          "value": "singlebutton",
          "translatable": false
        },
        "buttoneventid": {
          "optionJcrId": "956f0f3f-ea6f-43a3-935f-76eed538ff80",
          "type": "hidden",
          "label": "Button action type",
          "value": "_eventId_submit",
          "translatable": false
        },
        "buttonsize": {
          "optionJcrId": "68fbe869-e0d1-4cdb-acd9-e0df2c158234",
          "type": "select",
          "label": "Button Size",
          "value": [
            {
```

```
            "label": "Mini",
            "value": "btn-mini",
            "selected": false
          },
          {
            "label": "Small",
            "value": "btn-small",
            "selected": false
          },
          {
            "label": "Default",
            "value": "",
            "selected": true
          },
          {
            "label": "Large",
            "value": "btn-large",
            "selected": false
          }
        ],
        "translatable": false
      },
      "buttontype": {
        "optionJcrId": "6ef7eec7-721e-4162-b1f7-59c23d79bc25",
        "type": "select",
        "label": "Button Type",
        "value": [
          {
            "label": "Default",
            "value": "btn-default",
            "selected": false
          },
          {
            "label": "Primary",
            "value": "btn-primary",
            "selected": true
          },
          {
            "label": "Info",
            "value": "btn-info",
            "selected": false
          },
          {
            "label": "Success",
            "value": "btn-success",
            "selected": false
          },
          {
            "label": "Warning",
            "value": "btn-warning",
            "selected": false
          },
          {
            "label": "Danger",
            "value": "btn-danger",
            "selected": false
          },
          {
            "label": "Inverse",
            "value": "btn-inverse",
            "selected": false
          }
        ],
```

```
            "translatable": false
          },
          "buttonlabel": {
            "optionJcrId": "108ad499-6b60-49b4-a532-a365e3980112",
            "type": "input",
            "label": "Button Label",
            "value": "Submit",
            "translatable": true
          },
          "buttonglyphiconswhite": {
            "optionJcrId": "1f266640-40b3-450b-9d86-fecf4bca8808",
            "type": "checkbox",
            "label": "Button Glyphicons White",
            "value": true,
            "translatable": false
          },
          "btnblock": {
            "optionJcrId": "4bafdc83-8e80-4d88-8e7c-a83a04c48e25",
            "type": "checkbox",
            "label": "Button block",
            "value": false,
            "translatable": false
          }
        }
      }
    ],
    "stepJcrId": "84d82584-10cc-42bf-b66f-624391245636"
  }
],
"formStepsNumber": 1,
"jcrId": "97b47e97-b922-4f2f-a020-7a3d686b6262",
"jcrPath": "/sites/ACMESPACE/formCenter/forms/actions",
"formStepAction": {
  "stepActionContent": [
    {
      "title": "Save to JCR",
      "nodeType": "fcnt:saveToJcrAction",
      "fieldJcrId": "5b7577a4-d41d-486c-ac52-16f29005c489",
      "choiceField": "",
      "blockInput": false,
      "fields": {
        "id": {
          "optionJcrId": null,
          "type": "input",
          "label": "ID / Name",
          "value": "savetojcr_stepAction_1",
          "translatable": false
        },
        "apiEntryPoint": {
          "optionJcrId": "b635e583-bc4c-4ea0-b508-ccda226457e0",
          "type": "hidden",
          "label": "Api Entry Point",
          "value": "/modules/formcenter/results",
          "translatable": false
        },
        "label": {
          "optionJcrId": "eaa0cc2c-d11a-4ee2-ac08-3080e2fc98d5",
          "type": "inputdisabled",
          "label": "Label Action",
          "value": "Save to JCR",
          "translatable": false
        },
        "actionname": {
```

```
            "optionJcrId": "b0e72ebc-6bca-4739-889b-794985de172a",
            "type": "inputdisabled",
            "label": "Action Name",
            "value": "savetojcr",
            "translatable": false
          }
        }
      }
    ],
    "stepActionJcrId": "32c3a2fb-0b8c-43b2-aefc-b781acb38f61"
  },
  "currentStep": 0,
  "cssClassName": null,
  "formDescription": null,
  "created": "2014-12-02T09:23:48.259-05:00",
  "hasNotBeenModified": true
}
```

Jahia Solutions Group SA

9 route des Jeunes,
CH-1227 Les acacias
Geneva, Switzerland

http://www.jahia.com