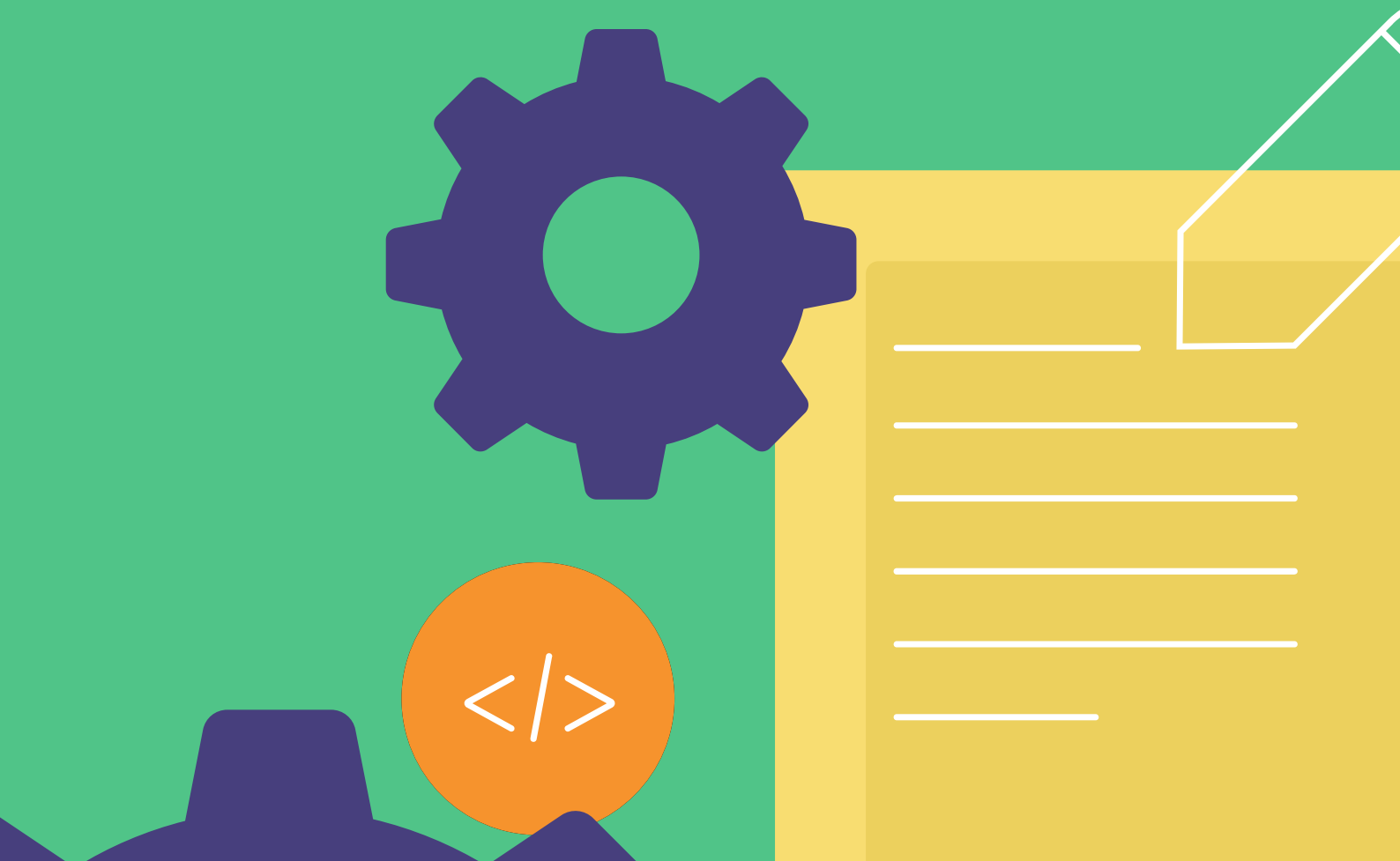
FORMCENTRIC

Developer Manual

Version 2406.0.2.1



Formcentric for CoreMedia: Developer Manual

Breite Str. 61, 22767 Hamburg
Germany

Disclaimer

We reserve the right to alter the software and the contents of the manual without notice. We accept no liability for the accuracy of the contents of the manual, nor for any losses that may result from the use of this software.

Trademarks

In the course of this manual, references may be made to trademarks that are not explicitly marked as such. Even if such a mark is not given, the reader should not conclude that the name is free of third-party rights.

1. Introduction

This manual describes how to install, configure and extend the Formcentric form manager extension. It is intended to be read by administrators and developers. To get the most out of this document, you will need knowledge of CoreMedia from both an administrator and user perspective, as well as experience in developing Java software.

Chapter 4, *Integration* : describes the steps that you need to complete in order to install Formcentric.

Chapter 5, *Configuration* : describes how you configure the various Formcentric components.

Chapter 6, *Programming and customisation* : shows how you can extend Formcentric to offer additional functionality.

1.1. Terminology

This manual makes use of the following terms:

Term	Description
Form author	The person that creates and edit forms.
User	The person that fills out a form.
Form	An HTML web form displayed in a web browser.
Form elements	All of the elements used when constructing a form (input fields, drop-down lists, check boxes, etc.).
Editor	CoreMedia Studio
Form Editor	An extension to CoreMedia Studio, with which forms can be created and edited.
Form data	The data entered into the form by the user.

2. Overview

On the editing side, Formcentric provides a CoreMedia Studio app with a graphical form editor, with which form authors can create and edit any number of web-based forms.

For form presentation and the processing of the submitted form data, Formcentric provides you with an integration into the CoreMedia CAE or, alternatively, a stand-alone server application.

When using the CAE integration, the HTML output is generated server-side by using content beans and FreeMarker templates. When using the Formcentric Headless Server, form rendering is completed browser-side by a React client supplied by Formcentric.

Both integrations include various Spring controllers for processing the data. A form controller validates the data it receives and forwards these to purpose-built actions, which then carry out the final processing. This approach permits the integration of various backend systems, such as mail servers, Formcentric Analytics or databases.

The Analytics component included with Formcentric provides storage and reporting functions for the form data submitted. Formcentric Analytics consists of two web applications. The Backend application is responsible for storing the data in a relational database. To do so, it provides a REST interface, which clients can use to communicate with the Backend. Alongside the actual form data, the Backend also stores form sessions, if this feature has been activated for the form in question.

The Reporting application is a single-page application with which the form data stored in the Backend can be displayed, edited, deleted and exported.

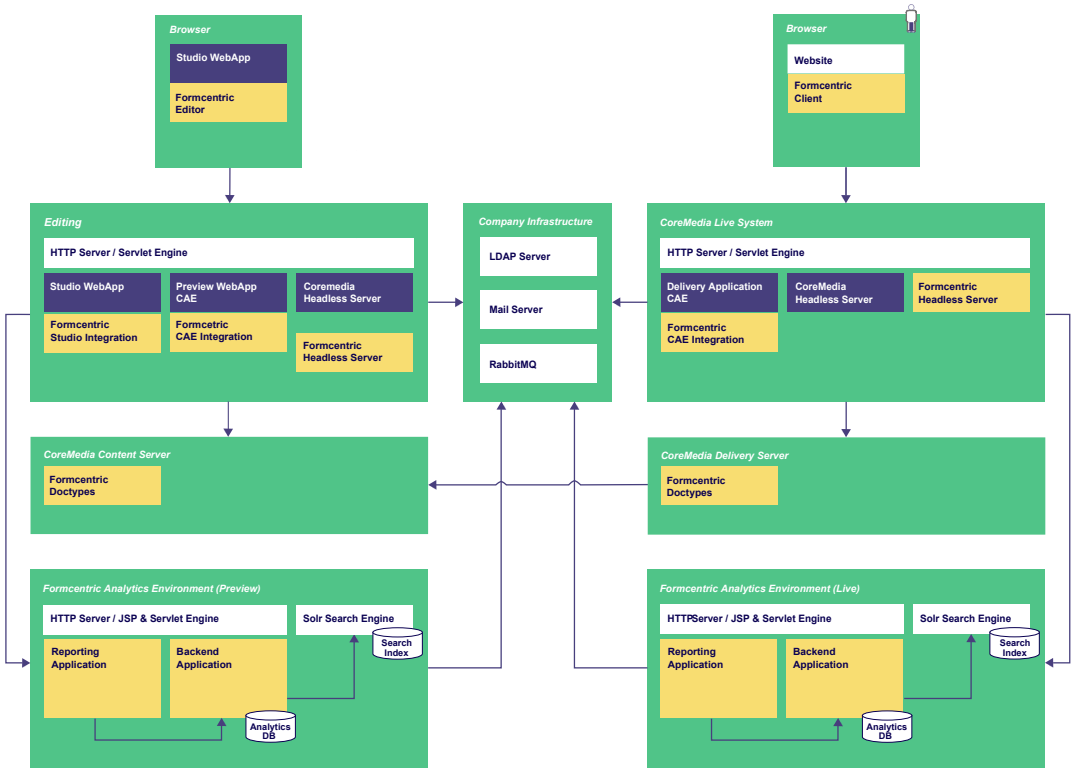


Figure 2.1. Architecture overview

3. System requirements

Formcentric 2406.0.2.1 is intended for use only with the matching version of the CoreMedia Content Cloud.

Formcentric requires the JavaScript framework “jQuery” from version 1.12.4 and Java 17.

Apart from this, the same system requirements apply as for the CoreMedia Content Cloud version deployed.

4. Integration

This section, written from a software developer's point of view, shows you how to integrate Formcentric into the CoreMedia Blueprint.

If your solution is not based on the CoreMedia Blueprint, you can integrate the Formcentric Extension exactly as described in the guide. All you need to do is ensure you remove the dependencies used out of the CoreMedia Blueprint.

4.1. Add Maven Repository and npm registry

Extend your Maven *settings.xml* to include the Formcentric Artifactory:

```
<servers>
  ...
  <server>
    <id>maven.monday-consulting.com</id>
    <username>my-username</username>
    <password>{my-encrypted-password}</password>
  </server>
</servers>
```

To obtain your personal login details, please contact our Helpdesk (helpdesk@formcentric.com).

Configure npm for access to the packages in the @formcentric scope in the Formcentric Artifactory:

```
pnpm config set @formcentric:registry \
    https://maven.monday-consulting.com/artifactory/api/npm/formcentric-npm/
```

Use npm to log into the npm registry:

```
npm login --scope=@formcentric --registry=\
    https://maven.monday-consulting.com/artifactory/api/npm/formcentric-npm/
```

You can use https://maven.monday-consulting.com to search for artifacts, download artifacts in your browser or stay informed about new releases.

4.2. Download Formcentric Extensions archive

The Formcentric Extensions archive contains all the files that you need to integrate Formcentric as a CoreMedia extension into the standard CoreMedia Blueprint, in version 2406.0.2. We provide you with the archive as a tar or zip file. Both archives include the same content: you can pick the format that is right for you.

```
mvn dependency:copy -Dartifact=com.formcentric.coremedia:
        formcentric-blueprint-extension:2406.0.2.1:tar
    -DoutputDirectory=.
```

The folder structure for the archive is adjusted to the Blueprint Workspace: the various *formcentric* extension folders for the various CoreMedia components, each under *modules/extensions*, must be moved accordingly into the corresponding directories (e.g. *apps/cae/modules/extensions*) in the Blueprint workspace.

The matching folder structure means that unpacking and relocating the extension is possible by executing just one command in the Blueprint's root directory:

```
tar -xvf formcentric-blueprint-extension-*.tar
```

4.3. Integrate Formcentric Extensions

In Section 4.2, "Download Formcentric Extensions archive", you downloaded the Formcentric Extensions and unpacked these to the correct location in the Blueprint. As a next step, you modify the Maven parent in all Formcentric Extensions (such as *apps/cae/modules/extensions/formcentric/pom.xml*) and in all submodules to match the GroupId and version of your Blueprint:

```
<parent>
    <groupId>your.blueprint.groupId</groupId>
    <artifactId>component.extensions</artifactId>
    <version>your.blueprint.version</version>
</parent>
```

To do this, you can use the scripts already present in the Blueprint Workspace, *set-blueprint-version.sh* and *set-blueprint-groupId.sh*, because the Formcentric Extensions ship with the default Blueprint values for GroupId and version.

The Formcentric Extensions are now integrated into your Blueprint. As a final step, the CoreMedia extensions need to be synchronised and activated. To do this, follow the CoreMedia documentation for the *CoreMedia Maven Extension Plugin* and activate the extension with the name *formcentric*:

```
mvn -f workspace-configuration/extensions extensions:sync \
    -Denable=formcentric
```

4.4. Add Formcentric Studio app

To use the Formcentric Editor app in the Studio, you first need to add the app to the Studio. In *apps/studio-client/global/studio/jangaroo.config.js*, add the following lines to the *appPath*:

```
"@formcentric/studio-app.customizations": {
```

```
  buildDirectory: "dist",
},
```

In *apps/studio-client/global/studio/package.json*, add the following line to the *dependencies*:

```
"@formcentric/studio-app.customizations": "<WORKSPACE VERSION>"
```

Replace *WORKSPACE VERSION* with the version from *package.json*.

As a final step, add the Formcentric app to the *packages* in *apps/studio-client/pnpm-workspace.yaml*.

```
 - "apps/formcentric/app"
```

4.5. Download Formcentric Frontend archive

The Formcentric Frontend archive contains all the files that you need to integrate the Formcentric Brick into the standard CoreMedia Blueprint (or Frontend) Workspace, in version 2406.0.2. We provide you with the archive as a tar or zip file. Both archives include the same content: you can pick the format that is right for you.

```
mvn dependency:copy -Dartifact=com.formcentric.coremedia:
  formcentric-blueprint-frontend:2406.0.2.1:tar
  -DoutputDirectory=.
```

You use the same technique to integrate the Formcentric Brick for extending the theme. Move the *formcentric* folder in *frontend/bricks* into the *frontend/bricks* folder in the Blueprint Workspace or alternatively into the *bricks* folder in the Frontend Workspace.

The matching folder structure means that unpacking and relocating the extension is possible by executing just one command in the Blueprint's root directory:

```
tar -xvf formcentric-blueprint-frontend-*.tar
```

or for the Blueprint Frontend Workspace:

```
tar -xvf formcentric-blueprint-frontend-*.tar --strip=1
```

4.6. Integrate Formcentric Brick

To extend the theme, the final step is to integrate the Formcentric Brick that is now present in *frontend/bricks/formcentric* or in *bricks/formcentric*.

These instructions are based on the *chefcorp-theme*, an example theme from the Blueprint. Integration into project-specific themes follows the same principles.

1. All of the Bricks used are listed in the *frontend/themes/chefcorp-theme/package.json* directory. Add *@formcentric/formcentric-coremedia-frontend* here:

```
{
  "name": "@coremedia/corporate-theme",
  ...
  "dependencies": {
    ...,
    "@formcentric/formcentric-coremedia-frontend": "workspace:*"
  },
  ...
```

Avoid version conflicts by adjusting the jQuery version used in the package.json for the Formcentric Brick to match the jQuery version used in the theme (if one is present here).

To be able to use autocomplete with the Formcentric FreeMarker macros, the formcentric.ftl can be added to the implicit imports in the *frontend/src/main/resources/freemarker_implicit.ftl* file:

```
...
[#-- asset management download portal --]
[#import "/lib/coremedia.com/blueprint/am.ftl" as am]
[#-- Formcentric --]
[#import "/lib/formcentric.com/formcentric.ftl" as fc]
...
```

4.7. Building the Workspace

Formcentric is now fully integrated as a Blueprint extension and the Blueprint theme has been extended by the Formcentric Brick. You can now build the Workspace as usual with Maven.

5. Configuration

For form presentation and the processing of the submitted form data, two alternatives are available.

5.1. CoreMedia Headless Server

The GraphQL extension from Formcentric for the CoreMedia Headless Server extends the schema for the standard form document type *Form* by two properties: these are required for use with the Formcentric Client and the Formcentric Headless Server.

These are:

formDefinition (String): Encrypted form definition for use with the Formcentric Client.

formReferences (String): Encrypted references that reference other CoreMedia content in the form definition and which are only used by the Formcentric Headless Server.

formReferencedContent: All of the content documents linked from the form definition, in the form of an array of CMTeasable objects.

A minimal GraphQL example query might look as follows.

```
{
   content {
      content(id: "<replace with FORM ID>") {
         ... on Form {
            formDefinition
            formReferences
            formReferencedContent {
               ... on CMTeasable {
                  type
               }
            }
         }
      }
   }
}
```

5.2. CAE extension

Please take the Spring configuration classes from the Formcentric Blueprint Workspace mentioned above. We supply this to you as a ZIP archive for integration into the CoreMedia Workspace.

The files are located at *formcentric-blueprint-cae/src/main/...*

The folder structure is as follows:

File/directory	Description
java/com/formcentric/...	Spring configuration classes and blueprint integration code
resources/META-INF/coremedia	Configuration properties and Content-bean declaration
resources/META-INF/spring	Spring Boot autoconfiguration imports

5.2.1. Spring configuration classes

The configuration of the form extension within the web application is performed using Spring configuration classes, which are stored in the *formcentric-blueprint-cae/src/main/java/com/formcentric/coremedia/blueprint/cae/config* source directory. The following settings can be configured here:

FormcentricAutoConfiguration.java

This autoconfiguration aggregates all the required Spring configuration classes.

FormcentricAnalyticsConfig.java

Configures the Formcentric Analytics integration, if it is enabled.

FormcentricCaptchaConfig.java

The open source JCaptcha framework is used to generate captchas. The configuration here is a standard JCaptcha configuration, which can be used to influence the appearance and behaviour of individual captchas. For a detailed description of configuration options, please visit the project website.

https://jcaptcha.atlassian.net/wiki/display/general/Home

FormcentricControllersConfig.java

Configures the form controller, the REST controller and the *FormCommandBeanFactory* that is used to create the *FormCommandBean*. The *FormCommandBean* calls the configured initializer, validators and actions, and generates the form model.

You configure new actions, REST services, mail body renderers and validators by registering them in the Spring applicationContext.

formcentric-contentbeans.xml

Configures the form content bean. If your document model differs, then you will need to modify or delete this configuration.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
```

```
            http://www.springframework.org/schema/beans/spring-beans.xsd">

  <import resource="classpath:/framework/spring/blueprint-contentbeans.xml"/>

  <bean name="contentBeanFactory:Form"
        class="com.formcentric.coremedia.blueprint.contentbeans.FormImpl"
        scope="prototype"
        parent="abstractTeasable-blueprint-doctypes">
      <description>
        contentbean mapping for contents of type 'Form'
      </description>
      <property name="digester" ref="formDigester"/>
  </bean>
</beans>
```

For the form framework to be capable of handling your specific form content bean, it must implement the *com.formcentric.contentbeans.WebForm* interface.

An implementation can be found in the package *com.formcentric.coremedia.blueprint.contentbeans* .

5.2.2. Usage without Formcentric Analytics

Setting the property *analytics.enabled* to *false* fully disables the Formcentric Analytics integration on the backend side. You will need to manually adapt the editor configuration to fully remove the Analytics action from the available actions.

5.2.3. Formcentric license file

The property *formcentric.license* configures the LicenseLoader for Formcentric. With it the path towards the license file can be configured.

Example (Linux / Unix): */path/to/formcentric-license*

Example (Windows): *C:/path/to/formcentric-license*

Paths, that do not start with a / will be resolve relative from the Webapp root.

5.2.4. Web security

Formcentric contains a security servlet filter as a safeguard against cross-site scripting (XSS) attacks and cross-site request forgery (XSRF) attacks. This filter removes illegal HTML tags from the form data submitted. The filter also checks to confirm that the form data contains a valid XSRF token.

To safeguard against XSRF attacks, each form can be given an additional XSRF token as a hidden parameter: this is then submitted to the web application along with the normal form data. The security filter verifies that the token submitted matches the

token stored in the user's session. If this is the case, the request is forwarded to the web application. If not, a 401 error message is returned to the calling client and the failed access is logged in the web application log using the *warn* log level with the following information:

- URL accessed
- Form data submitted (POST parameter)
- IP address of the accessing client
- Fully qualified name of the accessing client or the last proxy used

The following example shows you how to insert the XSRF token into the form document's output template:

```
...
<#assign targetUrl=cm.getLink(self, "ajax")!""/>
<form method="post" class="mwf-form ${self.properties['style_class']!""}"
        ...
        data-mwf-form="${self.shortId}"
        data-mwf-settings='{
            "url":"${targetUrl}",
            ...
        }'>

    <#-- Include XSRF token&#8211; -->
    <@fc.xsrfToken />

    ...

</form>
```

Alongside the form template, you must also include XSRF token handling in the output template of the *InputField, ComboBox, RadioGroup, CheckBoxGroup* and *FileUpload* elements.

The following example shows you how you can insert the XSRF token in the *InputNode.fileUpload.ftl* template's form data.

```
...
<#-- Construct upload URL, add XSRF token parameters and
store in variable -->
<#assign uploadUrl=cm.getLink(self, "upload", {"form": form,
    "tokenName": fc.xsrfTokenName(),
    "tokenValue": fc.xsrfTokenValue()})!"" />

<div class="mwf-upload"
     data-mwf-fileupload='{
        "url":"${uploadUrl}",
        "id":"${self.id}",
        "name":"${self.name!""}",
        "autoUpload": ${self.properties['auto_upload']},
        "labels": ${rowLabels},
```

```
        "previewMaxWidth": "120",
        "previewMaxHeight": "120"
    }'>

...
```

You configure the security filter with the help of the available properties with the *security.* prefix. You can use the following configuration parameters:

Parameter	Defaultvalue	Description
security.xsrfPrevention	true	You use this parameter to specify that the user session should be secured with an additional XSRF token (true|false).
security.xsrfMethods	POST	Here, specify the HTTP methods (GET, POST) for which an XSRF token will be required.
security.xsrfSessionBased	true	You can use this parameter to specify whether the token should be valid for the entire session (true) or should be renewed on each page reload (false).
security.xsrfTokenName	com.formcentric.XSRFToken	Here, enter the name of the request parameter that will be used to pass the XSRF token. The current form ID is automatically added to the end of the token name. Example: myTokenName:1234=5E29A7...
security.xssPrevention	true	You use this parameter to specify that illegal HTML tags will be removed from the form data submitted. The default behaviour is to strip all HTML tags.

5.2.5. Saving the form state

By default, all the data entered by the user is saved in the user session on the server. Where forms are complex, however, the session may expire before the user has

finished completing and sending the form. In this case, the data items stored in the session are lost.

Formcentric therefore offers you the option of saving the entered data for a longer period of time. This means users can take a break from form entry and continue filling out the form at a later point in time.

Formcentric offers you two separate store implementations as Spring beans:

FileFormStateStore

This implementation stores the form data in an encrypted file on the server. The associated file name is stored in a cookie. The cookie's directory, encryption password, lifetime, domain and path can all be specified in the Spring bean declaration. By default, it is only enabled if Formcentric Analytics is disabled.

BackendFormStateStore

This implementation stores the form data in the Formcentric Analytics backend database. It is enabled by default if Formcentric Analytics is enabled.

5.2.6. Password encryption

In the default configuration, login credentials for databases, mail servers, etc. are stored in various configuration files in plaintext. In the event of a security breach affecting the server, attackers would gain access to valid login credentials. For this reason, you are given the option of storing passwords in an encrypted format. In this case, passwords are decrypted only when the application starts, using the stored encryption password. The password to be used for encryption must be stored in an environment variable before the Formcentric web applications are started. The default environment variable used by Formcentric for this is *fc_ENCRYPTION_PASSWORD*.

```
export MWF_ENCRYPTION_PASSWORD=my-encryption-password
```

You are provided with a command line program for encrypting the passwords. Once you have encrypted the passwords, they must then be entered manually into the corresponding configuration file.

Download the program from the Monday Maven repository by executing the following command at the command line. You can obtain the necessary login credentials by contacting our Helpdesk (helpdesk@formcentric.com).

```
mvn org.apache.maven.plugins:maven-dependency-plugin:3.0.2:copy \
-Dartifact=com.monday.webforms:encryption-cli:1.0:jar \
-DoutputDirectory=.
```

To encrypt a password, enter the following command at the command line:

```
java -jar encryption-cli-1.0.jar \
  -p '<encryption-password>' -e '<password>'
```

Please note that the parameters must be entered in single quotation marks. You can enter the following command line parameters when starting:

Parameter	Description
-p encryption-password	The password to be used for encryption or decryption.
-d	Decrypt password
-e	Encrypt password
-?	Show help

5.3. Formcentric Analytics Servers

The Formcentric Analytics Servers, i.e. the Formcentric Analytics backend application as well as the Formcentric Analytics reporting application, integrate themselves as an additional app workspace into the CMCC Workspace. After unpacking the Formcentric extension (see Section 4.2, “Download Formcentric Extensions archive”), the Formcentric Analytics Servers are provided as a new Maven module in the *apps/formcentric-analytics* directory. The structure and the layout are inspired by the CMCC app workspaces, so you will already be familiar with most of the principles used here.

To fully integrate the Formcentric Analytics Server workspace into the CMCC build, include the Maven module in the list of app workspace modules in the root POM of the CMCC Workspace:

```
<module>apps/formcentric-analytics</module>
```

The Formcentric extension provides the required SQL schema for the default MySql DB (see. *global/deployment/docker/mysql/init.db/createFormcentricAnalyticsDB.sql* and might be adapted for every other DB supported by Formcentric Analytics. It also provides the config for the Solr core/configset required by the Formcentric Analytics backend server (see the *fcanalytics* configset and *0-config-formcentric-analytics.sh* under *apps/solr/...*).

Full integration of the Formcentric Analytics Servers into the CMCC Workspace is an optional step. The Workspace can also be used individually, e.g. to enable a deployment or operational scenario outside the CoreMedia stack.

Both Formcentric Analytics Servers are Spring Boot applications. As such, all the standard configuration types are available, such as properties files, environment variables, runtime parameters, etc. The available parameters are described in Formcentric Analytics documentation, bundled as part of the Formcentric extension (see *apps/formcentric-analytics/doc-4.3.6*).

An example integration into the Docker Compose Setup is provided as part of the Formcentric extension and will be extracted into *global/deployment/docker/compose/formcentric.yml*.

5.4. Formcentric Headless Server

The Formcentric Headless Server integrates itself as an additional app workspace into the CMCC Workspace. After unpacking the Formcentric extension (see Section 4.2, “Download Formcentric Extensions archive”), the Formcentric Headless Server is provided as a new Maven module in the *apps/formcentric-server* directory. The structure and the layout are inspired by the CMCC app workspaces, so you will already be familiar with most of the principles used here.

To fully integrate the Formcentric Headless Server workspace into the CMCC build, include the Maven module in the list of app workspace modules in the root POM of the CMCC Workspace:

```
<module>apps/formcentric-server</module>
```

Full integration of the Formcentric Headless Server into the CMCC Workspace is an optional step. The Workspace can also be used individually, e.g. to enable a deployment or operational scenario outside the CoreMedia stack.

The Formcentric Headless Server is a Spring Boot application. As such, all the standard configuration types are available, such as properties files, environment variables, runtime parameters, etc. The available parameters are described in the following section.

The location used to store the Formcentric licence is configured for the Formcentric Headless Server in the *license.properties* file.

An example integration into the Docker Compose Setup is provided as part of the Formcentric extension and will be extracted into *global/deployment/docker/compose/formcentric.yml*.

6. Programming and customisation

6.1. Extending the Formcentric Form Editor

The Formcentric Studio integration is a single-page application that is based on the JavaScript *React* framework. The Form Editor's user interface is generated client-side on the browser using the JSON data sent by the server. The interface layout is specified declaratively using a number of JavaScript configuration files. This approach makes it easy for you to make changes and create extensions to the form editing interface.

The main starting-point for making changes to the form editing interface consists of the JavaScript configuration files that are stored in the development workspace in the *apps/studio-client* module in the *apps/formcentric/app/config* directory. All of the changes described below are made to the files in this directory.

The available form elements and their properties are described as JSON objects. The React application uses these to generate the form editing interface. The following example shows a configuration snippet for the *textArea* form element.

```
{
    icon: 'textarea',
    type: 'textArea',
    properties: {
        general: [
            {
                title: 'name',
                type: 'text',
                properties: {
                  required: true
                }
            },
            {
                title: 'label',
                type: 'text'
            },
            {
                title: 'hint',
                type: 'text'
            },
            {
                title: 'value',
                type: 'wysiwyg'
            },

            ...
        ]
    }
}
```

6.1.1. Adding a new form element

Extend the Form Editor to include a new form element by extending the configuration *fields_custom.js*. If you want to extend the Editor to include the form element *termsCheckbox* with the properties *name, text* and *link*, for example, then add the following object definition to the JavaScript array in the configuration *fields_custom.js*.

```
[
    {
        icon: 'termscheckbox',
        type: 'termsCheckbox',
        properties: {
            general: [
                {
                    title: 'name',
                    type: 'text',
                    properties: {
                        required: true
                    }
                },
                {
                    title: 'text',
                    type: 'wysiwyg',
                    properties: {
                        required: true
                    }
                },
                {
                    title: 'link',
                    type: 'reference',
                    properties: {
                        refType: 'pageref',
                        FS_refType: 'pageref'
                    }
                },
            ]
        },
        specialProperties: {
            condition: {
                conditionable: false,
                operators: {}
            }
        }
    }
]
```

Please note: The external JavaScript array already exists and simply needs to be extended by the configuration object.

The table below describes the possible attributes that a field definition can have at the first level.

Attribute	**Description**
icon	Type: *String*

Attribute	Description
	Name of the icon to load. The name specified must match the filename of the icon without the file extension.
type	Type: *String* Form element name
properties	Type: *Object* Defines the properties of a field that can be edited in the Form Editor on the right-hand side, under *Field properties*. The object properties of *properties* each correspond to individual Editor tabs. The following JSON snippet configures two tabs with the name *general* and *special*, with a total of three properties: *name, label, hint*. `properties: { general: [ { title: 'name', type: 'text' }, { title: 'label', type: 'text' } ], special: [ { title: 'hint', type: 'text' } ] }` To ensure that the field can be uniquely identified during later processing, the property *titel* is required with the value *name* in the *general* array. For a list of all available property types, please see Section 6.1.5, “Input elements for element properties”.
specialProperties	Type: *Object* You use the *specialProperties* attribute to configure properties that are evaluated by the Editor for internal functions. The following JSON snippet defines the usage of the field within a condition. `specialProperties: { condition: { conditionable: true, operators: {`

Attribute	Description
	<pre> startswith: { values: [], freeField: true, useChildren: false }, endswith: { values: [], freeField: true, useChildren: false }, contains: { values: [], freeField: true, useChildren: false } } } }</pre> Set *conditionable: true* to specify that the field can be selected in a condition. You specify the operators that are selectable in the condition for this form element type in the *operators* object. An operator definition always utilises the schema `<operator-name>: {values: [], freeField: true, useChildren: false}`. The name of the operator is also used as the translation ID for user interface internationalisation (see Section 6.1.7, “User interface internationalisation”). In the *values* attribute, you can specify a string array containing values that can be selected by the form author when defining a condition. If you specify the attribute `freeField: true`, this lets form authors enter user-defined values. This option is required for comparison operators, for example, where form authors need to enter their own comparison values. If the new field type is a list type with predefined options, you can specify the attribute *useChildren* if you want to make the list options selectable as a value for the condition.

6.1.2. Adding a new validator

To add a new validator to an input field, extend the *format* property of the corresponding input element.

The example below shows the configuration of the email validator for the single-line text field (*inputField*).

```
{
    title: 'format',
    type: 'dropdown_format',
    properties: {
        options: {
            email: {
                enabled: true,
                fields: {
                    errormessage: {
                        title: 'errormessage',
                        type: 'text'
                    }
                }
            }
        }
    }
}
```

The specified attribute name (*email* in the example) must match the external name of the validator. The name is also used for user interface internationalisation. In the translation file, the translation ID `<validator-name>Validator` is used to search for a label for the validator.

You can use *fields* to define the required fields for the validator. The available field types are listed in the table under Section 6.1.5, “Input elements for element properties”.

6.1.3. Adding a new action

Extend the Form Editor to include a new action by extending the configuration *actions_custom.js*.

If you want to extend the Editor to include the action *simpleMailAction* with the properties *to, subject, body* and *note*, for example, then add the following object definition to the JSON array in the configuration *actions_custom.js*.

```
[
    {
        icon: 'simplemailaction',
        type: 'simpleMailAction',
        properties: {
            general: [
                {
                    title: 'to',
                    type: 'text',
                    properties: {
                        required: true
                    }
                },
```

```
                {
                    title: 'subject',
                    type: 'text',
                    properties: {
                        required: true
                    }
                },
                {
                    title: 'body',
                    type: 'wysiwyg'
                },
                {
                    title: 'note',
                    type: 'wysiwyg'
                },
            ]
        },
        specialProperties: {
            condition: {
                conditionable: false,
                operators: {}
            }
        }
    }
]
```

The table below describes the possible attributes that an action definition can have at the first level.

Attribute	**Description**
icon	Type: *String* Name of the icon to load. The name specified must match the filename of the icon without the file extension.
type	Type: *String* Name of the field type.
properties	Type: Object Describes the properties of an action that can be edited in the Form Editor on the right-hand side, under *Properties*. The object properties of *properties* each correspond to individual tabs. The following JSON snippet creates two tabs with the name *general* and *special*, with a total of three properties: *to, subject* and *hint*. `properties: { general: [ { title: 'to', type: 'text', properties: { required: true`

Attribute	Description
	<pre> } }, { title: 'subject', type: 'text', properties: { required: true } }], special: [{ title: 'hint', type: 'text' }] }</pre> For a list of all available property types, please see Section 6.1.5, “Input elements for element properties”.
specialProperties	Type: *Object* You use the *specialProperties* attribute to configure properties that are evaluated by the Editor for internal functions. `specialProperties: { maxCount: 1 }` You use `maxCount: <count>` to specify how many times the action can be used within a form.

6.1.4. Adding new element properties

Element properties are defined under the *properties* attribute of the parent form element definition (see Section 6.1.1, “Adding a new form element”). You add a new property to the form element (form field, action or validator) by specifying a JSON object with the following structure.

```
{
    title: '<attribute-name>',
    type: '<field-type>',
    value: 'DefaultValue',
    properties: {
      required: true
    }
}
```

The following table describes the attributes of the configuration object.

Attribute	Description
title	Name used to store the field property in the form definition.

Attribute	Description
type	Property type. The available types are explained in the following list.
value	Optional specification of a default value.
properties	Other type-specific configuration options
properties.required	Specifies whether the property is a required field.

6.1.5. Input elements for element properties

The following table describes the configuration objects for the input elements of the available element properties. You can use these when defining the various form element properties. Please note that some types cannot be used with all form elements.

Type	Description
text	Text field Usage: all form elements `{ title: 'label', type: 'text' }`
number	Number field that only allows numeric input. Usage: all form elements `{ title: 'maxlength', type: 'number', properties: { min: 0, max: null } }` Also supports scientific number notation (e.g. *10e6*). Set *properties.min* and *properties.max* so define limits. Use *properties.min: null* to reset an existing limit.
date	Data selection element. Usage: all form elements `{ title: 'from', type: 'date' }`

Type	Description
	You can define a default value with `value`. Uses the standard JavaScript date format.
checkbox	Checkbox Usage: all form elements `{ title: 'requiredField', type: 'checkbox' }`
dropdown	List with fixed options from which the form author can select a single entry. Usage: all form elements `{ title: 'pattern', type: 'dropdown', properties: { options: ['dd.MM.yyyy', 'yyyy-MM-dd'] } }` You can specify the selection options as a string array with `properties.options`. `properties: { options: [ {text: 'Value 1', value: 'one'}, {text: 'Example Two', value: two} ] }` Options can also be defined as objects with *value* (value) and *text*.
dropdown_format	Drop-down list for field validators Usage: Input element (*inputField, passwordField,* etc.) During selection, the properties of the selected validator are shown underneath the drop-down list. The following example illustrates the definition of the email validator. `{ title: 'format', type: 'dropdown_format', properties: { options: {`

Type	Description
	<pre>email: { enabled: true, fields: { errormessage: { title: 'errormessage', type: 'text' } } } } }</pre> You can specify the selectable validators with `properties.options`. You can manage the validator properties with `properties.options["<validator-name>"].fields`. You use `properties.options["<validator-name>"].enabled=true` or `properties.options["<validator-name>"].enabled=false` to activate the validator or to deactivate it so that it is no longer selectable.
syntax	Multi-line JavaScript input field with syntax highlighting. Usage: all form elements <pre>{ title: 'script', type: 'syntax', value: 'function calculate() {};' }</pre>
condition	Input element for processing conditions. Usage: *condition* <pre>{ title: 'conditionContent', type: 'condition', properties: { conditional_fields: [], condition_conjunction: 'true', condition: [] } }</pre>
wysiwyg	Multi-line text input field that allows formatting syntax to be used. Usage: all form elements <pre>{</pre>

Type	Description
	`` title: 'value', type: 'wysiwyg' } ```
element	Allows the selection of other elements in the same form. Usage: all form elements ``` { title: 'elements', type: 'element' } ```
dataSource	Input element for a data source's variable parameter list. Usage: inputField, comboBox, radioGroup, checkboxGroup, hiddenField ``` { title: 'datasource', type: 'dataSource', properties: { datasource_params: [] } } ```
reference	Element for creating a dropzone for Content from the CoreMedia Studio. Usage: all form elements ``` { title: 'content', type: 'reference' } ```
multi_dropdown	List with fixed options from which the form author can select multiple entries. Usage: all form elements ``` { title: 'numberType', type: 'multi_dropdown', properties: { configuration_name: 'phoneNumberTypes' } } ``` You can define the selection options (as *strings*) with the *properties.options* attribute.

Type	Description
	Alternatively, the values can also be taken from the paragraph style sheet. In this case, specify the name of the corresponding GOM element in the *configuration_name* attribute. The *configuration_name* parameter specifies the name that is used to store the value in the form definition.
regEx_dropdown	Drop-down list for regular expressions. Usage: regex `{ title: 'mailPattern', type: 'regEx_dropdown', properties: { options: [ { text: '^[+]{0,1}[0-9\\s-/]*$', label: 'phone' }, { text: '^[a-zA-ZÁ-ÿöäüÖÄÜß\\s-]*$', label: 'characters' } ] } }`
field_mapping	Element for selecting a PDF template. Usage: *pdfAction* A PDF field from the template can then be assigned to the form fields. If the fields are drop-down lists, their options can be selected and assigned to one another. Usage: pdfAction `{ title: 'field_mapping', type: 'field_mapping', value: '[]' }`
custom_mapping	You can use this input element to design your own mapping tables. `{ title: 'custom_table', type: 'custom_mapping', properties: { mapping: [ { type: 'dropdown',`

Type	Description
```
            name: 'field',
            placeholder: 'custom_table.field',
            selectableFieldTypes: [
               'inputField', 'radioGroup'
            ]
         },
         {
            type: 'dropdown',
            name: 'option',
            placeholder: 'custom_table.option',
            connectedField: 'field'
         },
         {
            type: 'text',
            name: 'otherValue',
            placeholder: 'custom_table.otherValue'
         },
         {
            type: 'dropdown',
            name: 'attribut',
            loadRemoteData: 'FS_ServiceField',
            loadRemoteDataOptions: [
               {
                  name: 'connectedField',
                  key: 'task',
                  value: 'someDropdown'
               },
               {
                  name: 'connectedMapField_type',
                  key: 'type',
                  value: 'field'
               },
               {
                  name: 'connectedMapField',
                  key: 'value',
                  value: 'field'
               },
            ],
            placeholder: 'custom_table.attribut',
         },
         {
            type: 'dropdown',
            name: 'attributoption',
            placeholder: 'custom_table.attributoption',
            loadRemoteData: 'FS_ServiceField',
            connectedField: 'attribut',
         },
      ],
   },
},
```
 |

Type	Description
	You define the columns in the *properties.mapping* attribute. You use the *type* key to decide whether this is a selection field (drop-down list) or an input field (text). The *name* key sets the key for the export of the respective fields in a row. You use the *placeholder* key to define the placeholder for the field. You use the *loadRemoteData* key to decide, as you can with a drop-down field, if the options should be provided by a FirstSpirit service. If the options are provided by a FirstSpirit service, you can use the *loadRemoteDataOptions* key to pass additional attributes, such as values of other fields on the element, for example, or from the mapping itself. The following example creates this object, so as to pass it with the loadRemoteData call. `{task: <ValueOfFieldsomeDropdown>, type: <Typeof FormElementSelectedinField>, value: '<ValueofMappingDropdownField>'}` If you want to select existing form fields in a drop-down, you can pass these by using the *selectableFieldTypes* key. If you want to access nested options, from a form field or from the options from loadRemoteData, then you can use *connectedField* and specify the name of a mapping field to access these and make them available for selection.

6.1.6. Editing existing form elements

To modify an existing form element, you copy its full element definition from the corresponding default configuration (*fields_default.js* or *actions_default.js*) into the corresponding configuration file (*fields_custom.js* or *actions_custom.js*).

You can then change or add to the element properties according to your requirements as has been described above.

Please note: Changes made to the default configurations in the development workspace have no effect on the Form Editor.

6.1.7. User interface internationalisation

For the internationalisation of the user interface, the language-dependent labels are read from master language files. Out of the box, Formcentric supports the languages English and German.

To modify or add labels for existing or new form elements, you need to extend or modify the *formeditor_de_custom.json* and *formeditor_en_custom.json* language files, which you will find in the development workspace.

Each label is stored in the language files with a unique translation ID. Typically, the translation IDs of the element properties are each made up of the internal element name and the name of the respective property. For the *placeholder* property of the password field, the entry is as follows:

```
"passwordField.placeholder": "Placeholder"
```

You can add a label for a new element property by adding the corresponding entry to each language file.

6.2. Extending the CAE integration

6.2.1. FreeMarker templates

As with all other document types, the output of the forms and form elements is handled by FreeMarker templates within the CAE. As is the case with CoreMedia document types, this involves each form element type being assigned its own template.

The example below shows the template *InputNode.inputField.ftl* for the single-line text input field.

```
<@spring.bind fc.bind(self) />
<#assign hasErrors=spring.status.error />
    <#assign restUrl=cm.getLink(self, "rest", {"form": form,
        "tokenName": fc.xsrfTokenName(),
        "tokenValue": fc.xsrfTokenValue()})!"" />

    <#assign params=self.properties['datasource_params']!"{}"/>

    <input id="${self.id}"
           type="text"
           name="${spring.status.expression!""}"
           value="${spring.status.value!""}"
           class="mwf-text ${self.properties['style_class']!""}"
           ${self.properties['readonly']?boolean?then("readonly", "")}
           maxlength="${self.properties['maxlength']!""}"
           data-mwf-id="${self.id}"
           placeholder="${self.properties['placeholder']!""}"
           data-mwf-datasource='{
               "type" : "suggestion",
               "url" : "${restUrl}",
               "data" : {},
               "params" : ${params}
           }'/>
<@spring.showErrors separator="<p>" classOrStyle="mwf-error"/>
```

At the data level (model), all form elements are represented by an object of the *com.formcentric.model.xml.InputNode* type. To access the properties *name, label,*

value and *children*, you can use the corresponding getter methods on the *InputNode* bean. Access to all other properties is performed using the properties map from the InputNode bean.

The following table shows you all the form element types and their properties. The properties shown in square brackets must be read from the properties map.

Element	**Properties**
form	name, [style_class, next_label, submit_label, cancel_label, script, save_state]
inputField	name, label, value, [hint, placeholder, style_class, readonly, maxlength, datasource, datasource_params]
textArea	name, label, value, [hint, placeholder, style_class, readonly, maxlength, rows, cols]
passwordField	name, label, [hint, placeholder, style_class]
button	name, label, [hint, style_class, onclick]
checkBoxGroup	name, label, children, [hint, style_class, datasource, dynamic, datasource_params]
comboBox	name, label, value, children, [hint, style_class, datasource, dynamic, datasource_params]
pageBreak	name, label, [style_class, condition, style_class, next_label, back_label, script]
paragraph	name, value, [bold, italic, style_class]
captcha	name, label, [hint]
radioGroup	name, label, children, [hint, style_class, datasource, dynamic, datasource_params]
summary	label, [style_class, elements]
hiddenField	name, value
fileUpload	name, [multiple, hint, style_class, auto_upload]
condition	[condition, condition_conjunction, conditional_fields]
pageCondition	[condition, condition_conjunction, next_page, script]
layout	label, [layout]
calculatedValue	name, label, [script, visible, clientside, style_class]
mailAction	[subject, to, cc, bcc, from, body, format, note, replyto, send_hidden_fields, condition, condition_conjunction]
datastoreAction	[note, condition, condition_conjunction]
redirectAction	[note, condition, condition_execute, condition_conjunction, url, content, delay]

Element	Properties
webhookAction	[note, condition, condition_execute, condition_conjunction, url, fields, url_parameters, custom_headers, content_type]
sequenceAction	–

In addition to the InputNode beans described above, the system passes other objects in the request to the form templates. The following table gives you an overview of all objects passed.

Parameter name	Type	Description
self	com.formcentric.contentbeans.WebForm com.formcentric.model.xml.InputNode	Bean for the current form document or form element
pageElements	java.util.List	List with the elements of the current form page
pageCount	java.lang.Integer	Number of form pages
form	com.formcentric.contentbeans.WebForm	Form definition
currentPage	java.lang.Integer	Page number of the current form page
currentPageNode	com.formcentric.model.xml.InputNode	Current form page bean
formdata	java.util.Map	Map containing the form data entered by the user

FreeMarker functions and macros

Formcentric provides you with a FreeMarker library that contains specialised functions for displaying the forms.

To utilise these functions, insert the following instruction into the FreeMarker templates:

```
<#import "/lib/formcentric.com/formcentric.ftl" as fc>
```

The following section gives you a description of the functions contained in this library.

fc.forEachPageElement

List function that contains the elements on the current page.

```
forEachPageElement(boolean layoutFacets, boolean removeEmptyFacets,
```

```
    final String exclude, final String include)
```

Parameter	Description
layoutFacets	If this value is set to “true”, the elements will be split across layoutFacets (optional).
removeEmpty-Facets	Specifies whether empty layouts should be ignored when creating the list (optional). **Default value:** *false*
exclude	Comma-separated list of the element types that should be ignored when creating the list. If nothing is specified here, then all element types – with the exception of excluded types – are included (optional).
include	Comma-separated list of element types that should be included when creating the list (optional).

```
<#list fc.forEachPageElement(true, false, "condition", "") as layout>
    <ul class="${layout.properties['layout']!""}">

        <#if layout_index == 0 && currentPage == 0 && self.label?has_content>
            <li class="mwf-field"><h3>${self.label!""}</h3></li>
        </#if>

        <#list layout.items as input>
            <@cm.include self=input view=input.type />
        </#list>
    </ul>
</#list>
```

fc.forEachPage

List function that returns a list of the collected pages.

```
forEachPage(final boolean compact)
```

Parameter	Description
compact	Specifies whether form pages with the same title should be consolidated together (optional). **Default value:** *false*

fc.summary

Function that returns a list of all elements as a *com.formcentric.model.InputBean* for the form. This can also be used to query the data entered by the user.

This can be used to query the following properties:

name	Form element name
label	Form element label.
type	Form element type.
object	Form element value bean.
value	String representation of the value bean.
valueLabels	String array containing the labels of the options chosen in the selection field (*comboBox, radioGroup, checkboxGroup*). If the associated input element is not a selection, then the value of the element is returned in the array.
page	Number of the page on which the element is located.
pageLabel	Label of the page on which the element is located.
layout	Name of the layout in which the element is located.
input	*InputNode* of the element.

```
summary(InputNode self, String elements,
    final String include, final String exclude,
    final boolean hideEmptyFields, final String excludeIfEmpty)
```

Parameter	Description
self	*InputNode* of a form element. If this value is set, then the iteration is interrupted at the specified element.
elements	Comma-separated list containing the names of the form elements that should be shown in the summary. If this attribute contains a value, then the attribute *self* is ignored (optional).
include	Comma-separated list of element types that should be considered during iteration. If nothing is specified here, then all element types – with the exception of excluded types – are included (optional).
exclude	Comma-separated list of element types that should be ignored during iteration (optional). **Default value:** *button, hiddenField, condition, pageCondition, pagebreak, captcha, passwordField*
hideEmptyFields	Specifies that all empty fields should be ignored. **Default value:** *false*
excludeEmpty-Fields	Specifies that empty fields should be ignored (optional). **Default value:** *false*

```
<#list fc.summary(self, self.getPropertyAsString('elements'),
        self.getPropertyAsBoolean('hide_empty_fields', false)) as item>
    <tr>
        <#if item.input.type == "paragraph">
            <td colspan="2"><@markdown>${item.input.value}</@markdown></td>
        <#else>
            <td>${item.label?has_content?then(item.label, item.name!"")}</td>
            <td>${(item.valueLabels![])?join(", ")}</td>
        </#if>
    </tr>
</#list>
```

fc.captcha

Template that you can use to generate a captcha image.

Attribute	Description
url	URL of the captcha servlet.
id	ID of the captcha InputNode.
linkClass	CSS class(es) that is/are applied to the link to the captcha image (optional). **Default value:** *""*
imgClass	CSS class(es) that is/are applied to the captcha image (optional). **Default value:** *""*
title	The title attribute for the captcha image (optional). **Default value:** *""*
alt	The alt attribute for the captcha image (optional). **Default value:** *Captcha*

```
<#assign captchaUrl=cm.getLink(self, "captcha")!"" />
<@fc.captcha url=captchaUrl id=self.id linkClass="css-class__link"
    imgClass="css-class__img" title="A title" alt="Captcha" />
```

fc.ifCaptcha

Boolean function that evaluates whether the captcha *name* has **not** been entered correctly.

Parameter	Description
name	Name of the captcha element.

```
<#if fc.ifCaptcha(self.name!"")>
    <#assign captchaUrl=cm.getLink(self, "captcha")!"" />
    <@fc.captcha url=captchaUrl id=self.id />
</#if>
```

fc.getStandardButton

Function that supplies the standard form button determined by the “buttonType” attribute.

Parameter	Description
buttonType	Standard button type. Can have the following values: _next Button that takes the user to the next page in the form. _back Button that takes the user to the previous page in the form. _cancel Button that cancels form data entry. _finish Button that submits the form. _exit Button that can be used to exit from the form.

```
<#assign finishButton=fc.getStandardButton("_finish") />
<li data-mwf-container="${finishButton.id}" class="mwf-button mwf-next">
    <input type="button" value="${submitLabel}"
        data-mwf-submit='{"type":"finish",
            "query": "navigationId=${cmpage.navigation.contentId}"}'/>
</li>
```

fc.valueOut

Function that can be used to output the current value of a form field.

```
valueOut(String name, boolean preferLabel)
```

Parameter	Description
name	Form field name.
preferLabel	Specifies that the value’s label should be output instead of the value itself (optional).

```
${fc.valueOut(self.name!"", true)!""}
```

fc.conditions

Function with which the JavaScript definitions can be generated for conditional elements. Place this function at the end of the form template.

```
<#assign conditions=fc.conditions() />
```

fc.calculatedValues

Function that generates the JSON definitions for the *calculated values*.

```
<#assign calculatedValues=fc.calculatedValues() />
```

fc.markdown

Macro that can be used to output a value interpreted using markdown. This macro can handle both a passed value (see “value” parameter) or a body.

Parameter	Description
value	Value to be interpreted using markdown (optional).
inline	Specifies whether the output HTML should be restricted to inline elements (optional). **Default value:** *false*

```
<@fc.markdown value=self.value!"" inline=false />
<#-- or -->
<@fc.markdown inline=false>${self.value!""}</@fc.markdown>
```

fc.vars

Macro that can be used to replace variables from the form context in the output.

Parameter	Description
map	Map with the variable values (key, value). **Please note:** A map with the form variables (*formVariables*) and a map with the form values (*formdata*) is passed by default in the page scope.

```
<@fc.vars map=formdata>${action.properties['note']!""}</@fc.vars>
```

fc.bind

Function that returns the path to which the node is bound.

Parameter	Description
node	InputNode whose path is being queried.

```
<@spring.bind fc.bind(self) />
```

fc.encodeUrl

Function that encodes the URL passed in UTF-8.

Parameter	Description
url	URL that is to be encoded.

fc.hasValidator

Function that checks whether the validator specified by the *name* parameter is present in the InputNode *node*.

Parameter	Description
node	InputNode that is to be evaluated.
name	Name of the validator.

fc.validatorByName

Function that returns the validator specified in the *name* parameter of the InputNode *node*.

Parameter	Description
node	InputNode that is to be evaluated.
name	Name of the validator.

fc.elementByName

Function that returns the InputNode specified in the *name* parameter for the form *form*.

Parameter	Description
form	Form that contains the InputNode.
name	Name of the element.

Security library

Formcentric includes a security library for the generation and output of XSRF tokens (see Section 5.2.4, “Web security”). This security library is also utilised by the integration of the FreeMarker functions.

The FreeMarker functions that are included are described below.

fc.xsrfToken

Macro that generates a hidden form field with an XSRF token.

```
<@fc.xsrfToken />
```

fc.xsrfTokenName

Function that generates an xsrfTokenName from the form ID.

Parameter	Description
formId	ID of the form for which the token should be generated. If this parameter is empty, the form ID passed in the request is used (optional).

```
<#assign restUrl=cm.getLink(self, "rest", {"form": form,
  "tokenName": fc.xsrfTokenName(), "tokenValue": fc.xsrfTokenValue()})!""/>
```

fc.xsrfTokenName

Function that generates an xsrfTokenValue from the form ID.

Parameter	Description
formId	ID of the form for which the token should be generated. If this parameter is empty, the form ID passed in the request is used (optional).


```
<#assign restUrl=cm.getLink(self, "rest", {"form": form,
  "tokenName": fc.xsrfTokenName(), "tokenValue": fc.xsrfTokenValue()})!""/>
```


6.2.2. Implementing an action

As already described, the business logic for form data processing is encapsulated by actions in the web application. From a technical perspective, these are classes that implement the *com.formcentric.actions.Action* interface. Additional business beans can be injected into an action via Spring. In this way, data access objects (DAOs) can be made available in order to access external databases, for example. The actions are injected into the form controller via Spring when the application starts. For this to work, the action implementation must be registered as a Spring bean. This might be achieved by either annotating it directly as a Spring component and enabling component scanning, or by declaring it as a bean in a configuration class.

Variable action parameters, which must be entered by the form author when creating the form (such as the target address for the mail action), are passed to the action implementation via the properties map of the ActionNode bean.

The following example shows you how you can implement and configure the *CustomAction* described in section Section 6.1.3, “Adding a new action”.

```
public class CustomAction extends BaseAction {

  public static final String PROP_CUSTOM = "anyCustomActionPropertyName";

  @Override
  public ModelAndView execute(ExecutionContext<WebForm> context, Map<String,
    Object> formData) throws Exception {

    WebForm formDefinition = context.getFormDefinition();
    ActionNode action = context.getAction();

    String customParam = action.getPropertyAsString(PROP_CUSTOM);

      // Business-Logic
```

```
      ...

    return HandlerHelper.createModelWithView(formDefinition, "success");
  }

  @Override
  public boolean isExecutable(ExecutionContext<WebForm> context,
     Map<String, Object> formData) throws Exception {
    return true;
  }

  @Override
  public String name() {
    return "customAction";
  }
}
```

When called via the *execute* method, the action is given all the available data. In addition to the actual form data, *ExecutionContext* passes the form definition, the action definition, the form variables (see Section 6.2.3, “Adding variables for pre-filling form fields”) and the request object. The *parameters* map contains only the values of the visible form elements. Access to all form data is provided by calling the method *getRawFormData()* on the *ExecutionContext* bean.

The *execute* method must return an object of the *ModelAndView* type. This is used in order to present the results page, as shown to the user after data has been submitted.

Typically, the *ModelAndView* object is generated with the form bean and a specialised view (such as *success*).

In addition, however, there is also the option of redirecting the request to another page. In this case, the *ModelAndView* object can be generated as follows:

```
ModelAndView mv = HandlerHelper.redirectTo(renderBean, viewName);
```

The *renderBean* object and the view name can be generated by the specialised business logic of the action implementation.

In some application scenarios, errors in the input data are discovered only during processing by the associated Backend. In this case, the user should not be presented with the results page but should be shown the form again, along with an error message. To achieve this, the action – in the same way as with validators – should create an error on the *Errors* bean passed in the *ExecutionContext*.

```
public ModelAndView execute(ExecutionContext<WebForm> context, Map<String,
    Object> formData) {
    ...
    context.getErrors().rejectValue("username", DUPLICATE_USER_ERROR,
        "That username is already being used.");

    return null;
}
```

6.2.3. Adding variables for pre-filling form fields

To pre-fill form fields, the form author can make use of a range of predefined variables. Standard variables made available to the form author include *date, time, serverDate, serverTime, clientDate, clientTime, timezone, url, language, ip, remoteUser, principal, userAgent* and *referer*.

To provide custom variables, you need to override the method *getVariables()* on the FormCommandBean.

```
public class CustomFormCommandBean extends DefaultFormCommandBean {

  @Override
  protected Map<String, Object> getVariables(HttpServletRequest request,
      WebForm formDefinition) {

    Map<String, Object> variables =
        super.getVariables(request, formDefinition);

    // Add your variables to the result map
    ...

    return variables;
  }
}
```

To instantiate the new *CustomFormCommandBean*, you will also need to adapt or replace the declaration of the the *DefaultFormCommandBeanFactory* found in the *FormcentricControllersConfig.java*.

```
public class CustomCommandBeanFactory extends DefaultFormCommandBeanFactory
{
  @Override
  public CustomFormCommandBean createBeanFor(WebForm formDefinition) {

     CustomFormCommandBean commandBean = new CustomFormCommandBean();
     initCommandBean(commandBean, formDefinition);

     return commandBean;
  }
}
```

Replace the *DefaultFormCommandBeanFactory* in the Spring configuration *FormcentricControllersConfig.java* with the *CustomCommandBeanFactory*:

```
@Bean
public CustomCommandBeanFactory commandBeanFactory(
        FormComponentRegistry<Action> actionRegistry,
        FormComponentRegistry<FormValidator> validatorRegistry,
        BaseFormStateStore formStateStore,
        Mailer mailer) {
    CustomCommandBeanFactory formCommandBeanFactory =
```

```
        new CustomCommandBeanFactory();
    ...
```

The form fields are initialised with the predefined pre-filled values once only, when the form is called for the first time. This also replaces the variables with their values. Accordingly, subsequent changes to variable values are not applied to an already-initialised form.

6.2.4. Implementing a REST service

Formcentric includes a REST interface, which you can use to fill drop-down lists or input fields at runtime with data from external systems. The data concerned can be static, dynamic or specific to the user. All the interface's specialised functions are encapsulated in classes of the *com.formcentric.rest.RestService* type. By implementing your own REST service, you can extend the interface to include additional functionality. The following example shows you a REST service that generates a map with static key/value pairs.

```
public class ExampleRestService extends BaseRestService {

  @Override
  public Object invoke(ServiceContext<WebForm> context, Map<String,
    Object> formData, Map<String, Object> data) {

    String myCustomParam = context.getConfigParameterMap()
        .get("myCustomParam");
    ...

    HashMap<String, String> data = new HashMap<String, String>();

    // fill the map
    data.put("key1", "value1");
    data.put("key2", "value2");
    data.put("key3", "value3");

    return data;
  }
}
```

By calling the *invoke* method, the REST service is passed both the ServiceContext as well as the user input already sent (*formData*parameter) and the user input not yet sent (*data* parameter). This enables you to react directly to user input, regardless of whether or not this input has already been sent.

The *ServiceContext* gives you access to the form definition, the input element, the configuration parameters for the REST service and the request object.

Register the REST service as Spring bean. This might be achieved by either annotating it directly as a Spring component and enabling component scanning, or by declaring it as a bean in a configuration class.

The service is accessed via the URL:

```
<context-path>/servlet/formcentric-rest?
     _service=Example&
     _id=<Dokument-ID>&
     _input=<Input-Name>
```

The following JSON string is returned as the response to this call:

```
[
  {
    "k":"key1",
    "v":"value1",
    "i":"mwf6aab0bb24033",
    "h":"8d0c3e13950d86c1a7383f066105f78c"
  },
  {
    "k":"key2",
    "v":"value2",
    "i":"mwf06a7a0930d37",
    "h":"d22d445101243a5f616cfd64c765e399"
  },
  {
    "k":"key3",
    "v":"value3",
    "i":"mwf1674ffb0a121",
    "h":"c0ad1fa77bb1b79ca757ee1ffce9f416"
  }
]
```

To prevent manipulation of the JSON data so transmitted, the individual key/value pairs are secured using an additional hash value that is validated on the server during form submission.

This security mechanism means that no calls may be made to external REST services, since their data does not contain the required hash values. If you need to access external services, however, you can implement your own proxy REST service, which in turn accesses the external REST service.

The REST service calls within FreeMarker templates are made using the HTML attribute *data-mwf-datasource*. In the attribute value, you must specify a JSON object that contains the URL of the REST service, the usage type (*checkbox, radio, selection* or *suggestion*) and any other parameters.

By default, you can specify a REST service for the following input elements:

inputField:

```
<#assign restUrl=cm.getLink(self, "rest", {"form": form,
        "tokenName": fc.xsrfTokenName(),
        "tokenValue": fc.xsrfTokenValue()})!"" />

<#assign params=self.properties['datasource_params']!"{}"/>
```

```
<input id="${self.id}"
       ...
       data-mwf-id="${self.id}"
       data-mwf-datasource='{
           "type" : "suggestion",
           "url" : "${restUrl}",
           "data" : {},
           "params" : ${params}
       }'/>
```

hiddenField:

```
<#assign restUrl=cm.getLink(self, "rest", {"form": form,
        "tokenName": fc.xsrfTokenName(),
        "tokenValue": fc.xsrfTokenValue()})!"" />

<#assign params=self.properties['datasource_params']!"{}"/>

<input type="hidden"
      ...
      data-mwf-id="${self.id}"
      data-mwf-datasource='{
            "type" : "hidden",
            "name" : "${self.name!""}",
            "url" : "${restUrl}",
            "data" : {},
            "params" : ${params}
         }' />
```

comboBox:

```
<#assign restUrl=cm.getLink(self, "rest", {"form": form,
        "tokenName": fc.xsrfTokenName(),
        "tokenValue": fc.xsrfTokenValue()})!"" />
<#assign userValue=fc.valueOut(self.name!"")!""/>
<#assign params=self.properties['datasource_params']!"{}"/>

<select data-mwf-id="${self.id}"
        data-mwf-datasource='{
            "type" : "selection",
            "url" : "${restUrl}",
            "preselected" : "${userValue}",
            "data" : {},
            "params" : ${params}
        }'>
    ...
</select>
```

checkBoxGroup:

```
<#assign restUrl=cm.getLink(self, "rest", {"form": form,
        "tokenName": fc.xsrfTokenName(),
        "tokenValue": fc.xsrfTokenValue()})!"" />
```

```
<#assign userValue=fc.valueOut(self.name!"")!""/>
<#assign params=self.properties['datasource_params']!"{}"/>

<fieldset data-mwf-id="${self.id}"
          data-mwf-datasource='{
             "type" : "checkbox",
             "name" : "${self.name!""}",
             "url" : "${restUrl}",
             "preselected" : "${userValue}",
             "data" : {},
             "params" : ${params}
          }'>
    ...
</fieldset>
```

radioGroup:

```
<#assign restUrl=cm.getLink(self, "rest", {"form": form,
        "tokenName": fc.xsrfTokenName(),
        "tokenValue": fc.xsrfTokenValue()})!"" />
<#assign userValue=fc.valueOut(self.name)!""/>
<#assign params=self.properties['datasource_params']!"{}"/>

<fieldset data-mwf-id="${self.id}"
          data-mwf-datasource='{
             "type" : "radio",
             "name" : "${self.name!""}",
             "url" : "${restUrl}",
             "preselected" : "${userValue}",
             "data" : {},
             "params" : ${params}
          }'>
    ...
</fieldset>
```

Since the double quotation mark must be used within the JSON string, you must use the single quotation mark for the HTML attribute.

As described previously, both the form input that has been sent and the form input not yet sent is available to you within the REST service. As one example of how to use this function, you could implement a REST service that takes a postcode entered by the user and returns a drop-down list of locations matching the postcode.

In this example, it would be advisable to update the drop-down list automatically if the user changes the postcode, since other locations may be referenced by the changed postcode. This can be achieved by using the parameter *dependsOn*. In the form editing interface, this can be entered into the parameter list of a REST service (see also section 3.5. in the Studio User Manual). The value to be entered here must specify the name of the input element on which the result of the selected REST service depends. Every change made to one of the input elements specified results in another call to the REST service.

6.2.5. JavaScript

Out of the box, Formcentric includes and requires the JavaScript dependencies as described below.

jQuery (npm package)

The Formcentric integration relies on the jQuery library and includes it as a dependency.

@formcentric/jquery-file-upload (npm package)

For file uploading, Formcentric uses a forked version of the blueimp-file-upload plugin. Depending on the browser used, files are either transferred using AJAX or within a hidden iframe.

jquery-autocomplete.js (bundled)

This JavaScript contains a jQuery plugin that can be used to add autocomplete functionality to input fields. Values for the autocomplete function are loaded asynchronously from the specified REST service.

jquery-format.js (bundled)

This JavaScript contains a jQuery plugin that enables the formatting or analysis of dates and numbers. Note that this is a JavaScript alternative to the Java classes *SimpleDateFormat* and *NumberFormat*.

JSON (npm package)

Formcentric uses the native *JSON* object supplied by modern browsers to parse and construct JSON objects. For older browsers that do not support the *JSON* object, the object is provided by this JavaScript.

jquery-webforms.js

This JavaScript contains a jQuery plugin that provides the JavaScript functions required by Formcentric. You can specify the plugin's variable configuration parameters as shown below on the *form* tag in the FreeMarker template *WebForm.ajax.ftl*.

```
<#assign conditions=fc.conditions() />
<#assign calculatedValues=fc.calculatedValues() />

<form method="post" ...
    data-mwf-form="${self.shortId}"
    data-mwf-settings='{
        "url":"${targetUrl}",
        "statisticsUrl":"${statisticsUrl!""}",
        "query":"navigationId=${cmpage.navigation.contentId}",
        "calculatedValues" : ${calculatedValues},
        "conditions" : ${conditions}
    }'>
```

Configuration is completed by specifying a JSON string in the attribute *data-mwf-settings*, which can contain the following parameters.

Parameter	**Description**
conditions	Type: *JSON* JSON definition of the client-side conditions to be evaluated.
calculatedValues	Type: *JSON* JSON definition of the client-side *calculated values* to be calculated.
appendUrlVars	Type: *Boolean* You use this parameter to specify that the URL parameters in the host page will be appended to the AJAX request sent to the form controller.
createOption	Type: *Function($form, entry, selected)* Function that creates an option element in a dynamic drop-down list (*comboBox*).
createRadio	Type: *Function($form, name, entry, checked)* Function that creates a radio button in a dynamic Radio Button Select field (*radioGroup*).
createCheckBox	Type: *Function($form, name, entry, checked)* Function that creates a check box button in a dynamic Check Box Select field (*checkBoxGroup*).
createUploadFileRow	Type: *Function(Object $form, Object attr, file)* This function creates a new entry in the file list of the File-Upload element before the file is uploaded.
createDownload-FileRow	Type: *Function(Object $form, Object attr, file)* This function creates a new entry in the file list of the File-Upload element after the file has been uploaded.
updateCalculatedValue	Type: *Function($form, id, value)* Function that updates the display of a *calculatedValue* when this has been re-calculated.
updateFormValue	Type: *Function($form, $elem, name, l)* This function updates the value of a form field in the summary, if this value has been entered or changed by the user. If the corresponding form field is a selection (list) field, the labels of the options selected are passed in the *l* parameter. Otherwise, the text value entered is passed.

Parameter	Description
onFillDropdown	Type: *Function(Object $form, Object $elem))* Callback function that is called after a dynamic dropdown list has been filled.
onFillSelection	Type: *Function(Object $form, Object $elem))* Callback function that is called after a dynamic drop-down list has been filled.
onInit	Type: *Function(Object $form)* Callback function that is called after the jQuery plugin has been initialised. Use this function to perform your own initialisations.
onSubmit	Type: *Function(Object $form, String url, String query)* Callback function that is called when the form is submitted. The form is submitted only if the function returns the Boolean value *true*.
onSuccess	Type: *Function(Object $form, Object data, String status, Object jqXHR)* Callback function that is called after the form has been submitted successfully.
onAjaxError	Type: *Function(jqXHR jqXHR, String status, String error)* Function that is called when an error occurs during an AJAX request. As standard, this function creates an entry in the browser's error log.
operations.visible	Type: *Function(Object $form, Object field)* Function used to make input fields visible.
operations.hidden	Type: *Function(Object $form, Object field)* Function used to hide input fields.
operations.alterable	Type: *Function(Object $form, Object field)* Function used to change input fields from write-protected to editable.
operations.readonly	Type: *Function(Object $form, Object field)* Function used to change input fields from editable to write-protected.
operations.enabled	Type: *Function(Object $form, Object field)* Function used to change input fields from deactivated to activated.

Parameter	Description
operations.disabled	Type: *Function(Object $form, Object field)* Function used to change input fields from activated to deactivated.
operations.optional	Type: *Function(Object $form, Object field)* Function used to mark input fields as optional.
operations.mandatory	Type: *Function(Object $form, Object field)* Function used to mark input fields as mandatory fields.

The default implementations of the JavaScript functions listed can be found in the JavaScript *jquery-webforms.js*.

In the following example, the *operations.mandatory* function is replaced by a modified version.

```
<#assign conditions=fc.conditions() />
<#assign calculatedValues=fc.calculatedValues() />

<form method="post" ...
  data-mwf-form="${self.shortId}"
  data-mwf-settings='{
    "url":"${targetUrl}",
    "statisticsUrl":"${statisticsUrl!""}",
    "calculatedValues" : ${calculatedValues},
    "conditions" : ${conditions},
    "operations": {
      "mandatory": "function ($form, field) {\r\n
      var $label = $form.find('label[for=\"' + field.input + '\"]'),\r\n
      $span = $('<span>').attr('class', 'mwf-required').text('*');\r\n
      $label.children('span.mwf-required').remove();\r\n
      $label.append($span);};
      }"
    }
  }'>
```

You also have the option of specifying the configuration parameters listed above in a separate JavaScript file. This is the easier and preferred approach when specifying JavaScript functions in particular, since it avoids the error-prone masking of the reserved characters. An example file *jquery-webforms-custom.js* using this approach is provided.

```
(function($) {
  $.fn.webforms.defaults().operations.mandatory =
    function($form, field) {
      var $label = $form.find('label[for="' + field.input + '"]'),
          $span = $('<span>').attr('class', 'mwf-required').text('*');
```

```
            $label.children('span.mwf-required').remove();
            $label.append($span);
        };
})(jQuery);
```

Event reference

The Formcentric jQuery plugin makes a series of events available that enable you to respond to scenarios that match the various events. The corresponding event handler must be registered on the *document* object.

Detailed kinds of event-dependent information such as the associated form element, for example, are passed to the event handler in the *event.details* event object.

```
document.addEventListener("mwf-fill-selection",
    function(event) {
        console.log(event.detail.$form);
        console.log(event.detail.$elem);
    }
);
```

The following table describes the events that you can monitor and program specific responses to:

Event name	Detailed information	Is sent when
mwf-ajax-finished	*$dest*	the function *mwfAjaxReplace* has been executed successfully.
mwf-ajax-error	*$dest, jqXHR, textStatus, errorThrown*	the asynchronous call (AJAX call) has an error.
mwf-fill-dropdown	$form, $elem	a drop-down list has been filled by a data source.
mwf-fill-selection	$form, $elem	a radio button or check box select field has been filled by a data source.
mwf-fill-hidden	$form, $elem	a hidden field has been filled by a data source.
mwf-suggestion-selected	$form, $elem, id, selection, params	an autocomplete item has been selected for an input field.
mwf-value-changed	$form, $elem, name, value	the value of a form field has changed.
mwf-form-replaced	*$form, id*	the form has been submitted.

6.3. Extending the server application

The Formcentric Headless Server is a Spring Boot application that provides a REST interface with various end points for form processing. For browser-side connectivity to the Formcentric Headless Server, a ready-to-use React client is provided, which you can also configure to suit your requirements (see also Section 6.4, “Formcentric Client”).

The following section describes how to extend the functionality of the Formcentric Headless Server. Please note: some of the names of the framework classes are the same as those from the CAE integration but are located outside of the `com.formcentric.headless.rest` package.

6.3.1. Implementing an action

Similarly to the CAE integration, the Formcentric server also uses actions that encapsulate the business logic for the form data processing. These classes implement the interface `com.formcentric.headless.actions.Action`. You can then integrate any backend systems you need to by developing a custom action. These actions are Spring beans: as a result, configuration parameters can be passed to the action by using the standard Spring mechanisms. The following example shows you how to implement and configure a *CustomAction*.

```
import com.formcentric.headless.actions.*;

public class CustomAction extends BaseAction {

  public static final String PROP_CUSTOM = "anyCustomActionPropertyName";

  @Override
  public ActionResult execute(ExecutionContext context, Map<String,
     Object> formData) throws ActionException {

    WebForm formDefinition = context.getFormDefinition();
    ActionNode action = context.getAction();

    String customParam = action.getPropertyAsString(PROP_CUSTOM);

      // Business-Logic
      ...

    ActionResult actionResult = new ActionResult();
    actionResult.setView("success");
    return actionResult;
  }

  @Override
  public boolean isExecutable(ExecutionContext context,
     Map<String, Object> formValues) throws ActionException;
    return true;
  }
```

```
  public String name() {
    return "customAction";
  }
}
```

6.3.2. Adding variables for pre-filling form fields

Alongside the predefined variables, you also have the option of adding your own variables for pre-filling form fields. To do this, register a Spring bean of the *VariablesService* type in the application context of the Headless Server.

```
import com.formcentric.headless.services.VariablesService;
import com.formcentric.headless.model.WebForm;
import org.springframework.stereotype.Service;
import jakarta.servlet.http.HttpServletRequest;

@Service
public class CustomVariablesService implements VariablesService {
  @Override
  public final Map<String, Object> getVariables(HttpServletRequest request,
      WebForm formDefinition) {

    Map<String, Object> vars = new HashMap<>();

    // Add custom variables to the variables Map
    vars.put("custom_var", "custom_value");
    return vars;
  }
}
```

This *VariablesService* bean is a Spring bean, which means you can also access external systems or services when creating the variables.

In some application scenarios, you will need to pre-fill form fields with values from the web page or client application into which the form is embedded. For this use case, it is sufficient to specify the variables in the data attribute *data-fc-vars* from the *div* tag with which the form is associated.

```
<div
    data-fc-id="1249010"
    ...
    data-fc-vars='{"custom_var":"custom_value"}'
></div>
```

6.3.3. Implementing a REST service

All of the REST services described in section Section 6.2.4, “Implementing a REST service” are also available to you when deploying the Formcentric Headless Server. The following example shows you a REST service that generates a map with static key/value pairs.

```
import com.formcentric.headless.rest.*;
```

```
@Bean
public class ExampleRestService extends BaseRestService {

public Object invoke(ServiceContext context, Map<String, Object> formData)
  @Override
  public Object invoke(ServiceContext<WebForm> context, Map<String,
    Object> formData, Map<String, Object> data) {

    String myCustomParam = context.getConfigParameterMap()
        .get("myCustomParam");
    ...

    HashMap<String, String> data = new HashMap<String, String>();

    // fill the map
    data.put("key1", "value1");
    data.put("key2", "value2");
    data.put("key3", "value3");

    return data;
  }

  @Override
  public String name() {
    return "exampleRestService";
  }
}
```

By calling the *invoke* method, the RestService is passed both the ServiceContext and the current user input (*formData* parameter). This lets you respond directly to user input.

The *ServiceContext* gives you access to the form definition, the input element, the configuration parameters for the RestService and the request object.

All form elements, which also includes the REST services, must have their own unique name with which they can be referenced within the form definition. The name is determined when starting the application by calling the method `name()`.

Spring uses the *@Bean* annotation to instantiate your REST service automatically and register it using the specified name.

6.4. Formcentric Client

To present Formcentric forms in the browser, the NPM module `@formcentric/client` (https://www.npmjs.com/package/@formcentric/client) is required. This applies both for projects based on HTML only plus JavaScript as well as for projects that utilise frontend frameworks or frontend libraries.

The installed package includes various variants of modules for a wide range of applications. The files required are installed using NPM, which itself has no dependencies, however, and can also be used without any bundlers.

For installation, execute the following command:

```
npm install @formcentric/client
```

Or alternatively:

```
pnpm install @formcentric/client
```

The following items must be present in order for a form to be displayed correctly:

1. A *div* tag with an *fc-id* data attribute, into which the form will be rendered.
2. A loaded theme, consisting of CSS, templates and CSS custom properties, if these are being used in the CSS file.
3. To be able to be embedded as a script tag, *formapp.js* must also be accessible.

```
<div
    data-fc-id="1249010"
    data-fc-formapp-url="/example-url/formapp.js"
    data-fc-theme-url="/example-url/formcentric.css"
    data-fc-template-url="/example-url/formcentric_templates.js"
    data-fc-theme-variable-url="/example-url/formcentric.json"
    data-fc-form-definition="K82AClxH1YpNGtKt ... ffUuAm4OyEQsC9"
    data-fc-refs="ffUuAm4OyEQsC9 ... 2AClxH1YpNGtKt"
    data-fc-vars='{}'
    data-fc-params='{"}'
    data-fc-data-url='https://example-url-to-formcentric-headless-server.com'
></div>
```

```
<script
    src="./formcentric.js"
    defer
></script>
```

6.4.1. Theme

The theme CSS must be loaded to ensure that the form can be displayed correctly. This can be achieved by using a link tag in the HTML head and the use of custom properties. If available, these must be set in the HTML code.

Each input field has its own template, which can be modified. These templates are defined on the *Window* object in a JS file called *formcentric_templates.js*. This ensures that they can be found later when rendering the form. The templates are required in order to present the form correctly (see Section 6.4.3, “Templates”).

6.4.2. Initialisation

To start the client, either the script *formcentric.js* can be loaded or, at a later point in time after this has been loaded, `window.formcentric.initFormcentric()` can be called. You use the data attribute `fc-data-url` to configure the URL for accessing the Formcentric Headless Server.

6.4.3. Templates

Templates always consist of a function whose return value is used by the Formcentric Client to render HTML code. To achieve this, the Formcentric Client passes two parameters (html and props) to a template function. The exact structure and the parameters used will depend on the specific usage of the template.

html: A template literal tag, which is used to render HTML code. This parameter enables the embedding of HTML into the template's JavaScript code.

props: An object that contains the properties of the form field. These consist of calculated values from the Formcentric Client as well as field data supplied by the Editor. The specific properties vary according to the form field.

The final HTML is created from a combination of static HTML code and the values from the *props* properties. This can be achieved by interpolation, by combining strings together or by using functions to render HTML. The resulting HTML created is then passed back as a template function return value to the Formcentric Client, which then renders it in the DOM.

All template functions can also be processed asynchronously by the Formcentric Client as a promise.

Template properties

The Formcentric Client passes template properties to the corresponding template function as parameters (*props*). These include and information required for the presentation and behaviour of the respective form fields.

The following properties as passed to the templates:

Property	Description
key	The element's unique ID.
oninput	(event: InputEvent) => void Updates the field value.
onfocus	(event: FocusEvent) => void If present, returns any validation errors for the field.
onclick	A function that evaluates the *onClick* functions defined by the editor.
fieldSuccess	A Boolean value that specifies whether the element was successfully validated and has no errors.
fieldError	An object that contains information about an error in the element.
properties	An object that contains the properties of the field element.
components	An object that contains components for certain field types. These components are used to render the element in the

Property	Description
	template. The object contains components for *captcha, fileUploader, comboBox, suggestions, hint, datePicker and markdown*.
fieldSetFields	An array that contains the fields from a *fieldset*.
layoutFields	An array that contains the fields from a *fieldset*.
summaryFields	An array that contains the information from fields as specified in a *SummaryField*.
fieldEmptyText	A piece of boilerplate displayed if the *SummaryField* contains no values.
contentMarkup	A content component that is returned by a function specified by the *form* div.
hasService	A Boolean value that specifies whether the element has a REST service.
setRESTParams	(params: Record<string,any>) => void: A function that is used to specify the parameters for the element's REST service.

Element: The properties also contain all of the information about the element to be shown.

```
interface fcElement {

id: string // ID des Elements

name: string // Technischer Name des Elements

type: fcFieldTypes // Elementtyp

fieldSetId?: string // ID des FieldSets, in dem das Element enthalten ist

layoutId?: string // ID des Layout Elements, in dem das Element enthalten ist

label?: string // Label des Elements

value?: string | string[] // Wert des Elements

validators?: fcElementValidator[] // Validatoren

children?: {

    id: string

    type: fcFieldTypes

    name: string

    label?: string
```

```
    value?: string | string[]
    checked?: boolean
    properties?: fcProperties
    validators?: fcElementValidator[]
}[]
properties?: fcProperties // Properties des Elements (siehe Properties)
}
```

Field types:

```
type fcFieldTypes =
| 'error'
| 'success'
| 'formHeader'
| 'formFooter'
| 'inputField'
| 'button'
| 'form'
| 'layout'
| 'condition'
| 'passwordField'
| 'textArea'
| 'radioGroup'
| 'comboBox'
| 'checkBoxGroup'
| 'fileUpload'
| 'calculatedValue'
| 'hiddenField'
| 'paragraph'
| 'summary'
```

```
| 'dateField'

| 'numberField'

| 'emailField'

| 'phoneField'

| 'shortText'

| 'captcha'

| 'content'

| 'option'

| 'fieldSet'
```

Validators:

```
interface fcElementValidator {

    id: string

    name: string

    properties?: {

    errormessage?: string

    from?: string

    to?: string

    days_from?: string

    days_to?: string

    pattern?: string

    max_files?: string

    max_size?: string

    file_types?: string
}
```

Properties: All HTML attributes and field properties from the form definition are contained in *props.properties*. These are calculated as a result of conditions by the Formcentric Client, for example, or configured by the form author for the corresponding form field. The following table gives an overview of possible properties:

Property	Description
hint	An optional note text that gives the user more information or provides instructions.
placeholder	An optional placeholder text that is displayed in an input field if no value has been entered.
selected	A Boolean value that specifies whether the element is selected by default.
errormessage	An optional error message that is displayed if the element is invalid.
multiple	A Boolean value that specifies whether multiple values can be selected for this element.
auto_upload	A Boolean value that specifies whether an automatic upload function is activated.
datasource	A character string that specifies a data source.
datasource_params	A character string that specifies the parameters for the data source.
dynamic	A Boolean value that specifies whether the element is dynamic and has properties that can be changed at runtime.
visible	A Boolean value that specifies whether the element should be visible.
hidden	A Boolean value that specifies whether the element should be hidden.
writable	A Boolean value that specifies whether the element should be writable.
readonly	A Boolean value that specifies whether the element should be read-only.
optional	A Boolean value that specifies whether the element is optional.
mandatory	A Boolean value that specifies whether the element is required.
disabled	A Boolean value that specifies whether the element should be deactivated.
enabled	A Boolean value that specifies whether the element should be activated.
type	A character string that specifies the element type.

Components

Internal components are provided for selection by *props.components*. This is intended to simplify work with individual form fields if there is no need to modify the functionality provided by these fields. The following components are available:

Property	Description
captcha	Loads captcha images from the Headless Server
combobox	Displays drop-down lists
datePicker	Displays a date picker
fileUploader	Displays an upload dialog
hint	Displays note text
markdown	Displays markdown as HTML elements
suggestions	Displays autocomplete items from REST services as a drop-down list under input fields

The following properties can be passed to the components named above:

captcha:

Property	Description
buttonText	An optional character string for the refresh button on the captcha component. If this is not specified, the button shows an icon instead.

combobox:

Property	Description
all	All properties from the template's parameter must be passed.

datepicker:

Property	Description
all	All properties from the template's parameter must be passed.

fileUploader:

Property	Description
trigger	The class or ID of the trigger element
inline	An optional Boolean value that specifies whether the element should be displayed inline.

hint:

Property	Description
all	All properties from the template's parameter must be passed.

markdown:

Property	Description
markdown	The *markdown* property accepts stringified Markdown as a value.

suggestions:

Property	Description
All	All properties from the template's parameter must be passed.

The components are executed within the HTML template literal tag in the templates:

```
${ props.components.captcha( {…} ) }
```

Modifying and extending templates

To extend the templates, you can modify the HTML elements and classes inside the templates. This gives you the option of modifying the appearance and behaviour of the components.

You can add or remove classes to modify the styling, or add additional HTML elements in order to provide additional functionality.

The templates can also be executed asynchronously: this means that you can access and display data that is not passed directly to the templates by the Formcentric Client. This gives you the option of integrating with APIs or other external data sources.

To use asynchronous data in the templates, you can use JavaScript functions like *fetch* to request data from a server. You can then display the data received in the templates by utilising the corresponding variables or placeholders.

Extension options:

1. Support for one or more user-defined CSS classes. Optional CSS classes can be added, to modify the styling of the input element. For this, a custom class can be used in the *className* definition:

   ```
   input className="customClass" />
   ```

2. Modifying the markup: New markup elements can easily be added and existing elements modified:

```
inputField: (html, props) => html`<div className="fc-field
        ${props?.properties?.hidden ? 'fc-hiddenField' : ''}
        ${props.properties?.hint ? 'fc-field--has-hint' : ''}
        ${props?.fieldError ? 'fc-field--has-error' : ''}
        ${props?.fieldSuccess ? 'fc-field--is-valid' : ''}">

    ${customFunction(props)}

    <div className="fc-textinput">
        <div className="fc-textinput__input">
            <input
                id=${props.id}
                name=${props.name}
                value=${props.value}
                oninput=${props.oninput}
                onfocus=${props.onfocus}
                onblur=${props.onblur}
                type=${props.properties.type || 'text'}
                autocomplete=${props.properties?.autocomplete}
                maxlength=${props.properties?.maxlength}
                disabled=${props.properties?.disabled}
                placeholder=${props.properties?.placeholder || ''}
                readonly=${props.properties?.readonly}

                ${...customProperties}
            />

            ${props.components.suggestions(props)}
        </div>

        ${label(html, props)} ${hint(html, props)} ${error(html, props)}
    </div>
</div>
```

6.4.4. Troubleshooting

Always check the browser log. If no client output can be found there, then the *formcentric.js* script was not loaded and/or executed.

There are two reasons for a message stating that the form *div* could not be found:

1. No *div* tag with the data attribute *fc-id* was found
2. The script *formcentric.js* was loaded without specifying the *defer* attribute

Several issues may cause a situation where no form is displayed although a form *div* was found:

1. No *div* tag with the data attribute *fc-id* was found
2. The script *formapp.js* was not loaded
3. A missing template has prevented the client from starting.

6.5. Special integration scenarios

For most use cases, the *@formcentric/client* script is simply loaded and then executed. However, there are some special scenarios, such as single-page applications (SPA), in which the script must not be executed until the DOM tree has been fully constructed. In such cases, it is useful to be able to import the script dynamically and execute it at the exact moment when the virtual DOM has been fully constructed. This point in time will depend on the SPA framework.

```
function App() {
    const ref = useRef(null);

    const formDef = "TGU5Kmxe4svPaahc2aSm-4PHzoKWWtvC ... D-ZwC6MPQRWA==";

    useEffect(() => {
        if (!ref) return;

        import("@formcentric/client/dist/formcentric");
    }, [ref]);

    return (
            <div
                ref={ref}
                data-fc-id="<<id>>"
                data-fc-form-definition={formDef}
            ></div>
    );
}
```

If no dynamic import is possible, the function *initFormcentric* from the *window.formcentric* object can be called after loading the script.

```
window.formcentric.initFormcentric()
```