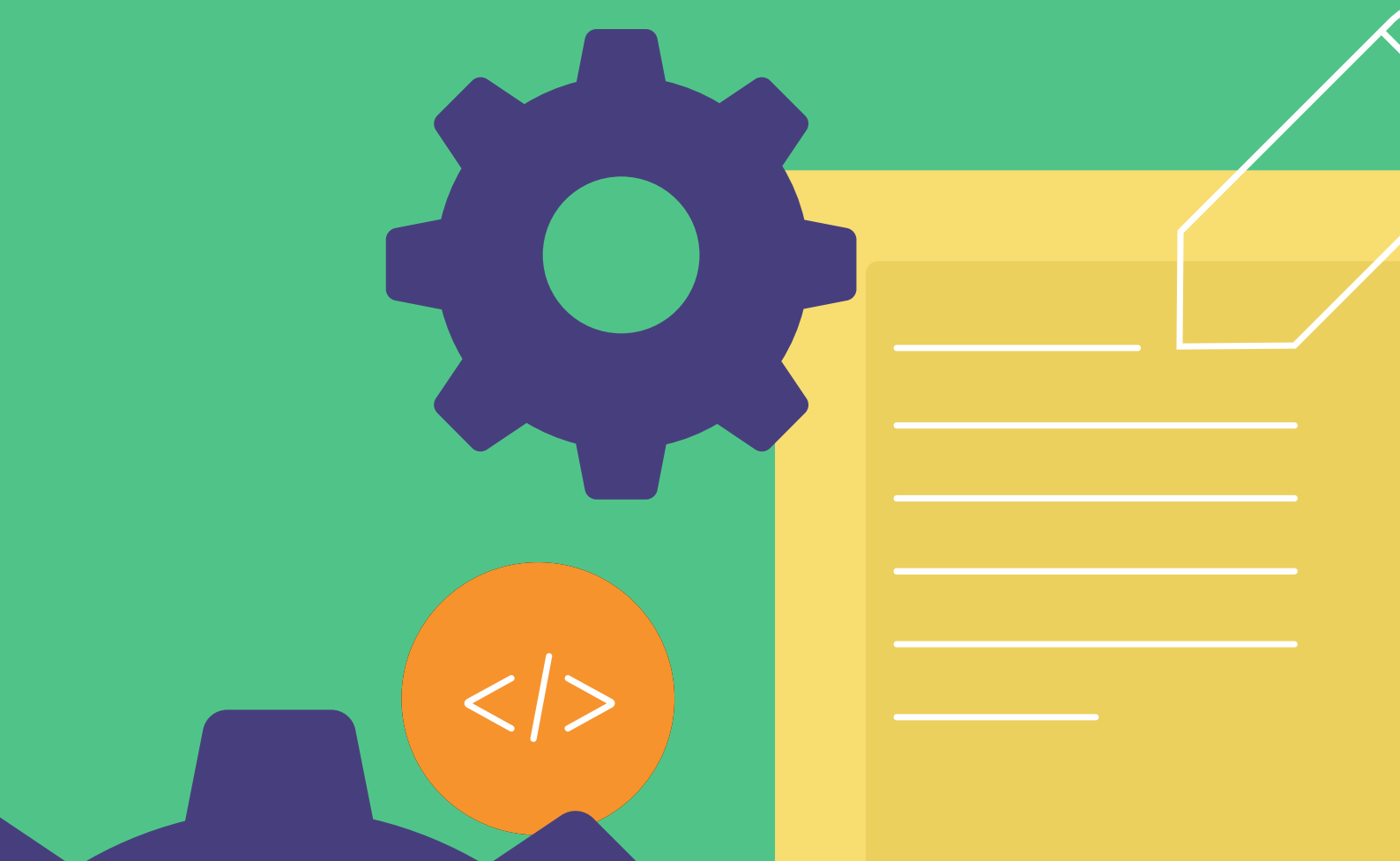
FORMCENTRIC

Migration Guide

Version 2401.6.0



Formcentric for CoreMedia: Migration Guide

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.

Please note: Printed copies are uncontrolled when printed.

1. Migrate Webforms to Formcentric

The Formcentric Editor for CoreMedia 11 is a standalone, single-page application based on the JavaScript React framework. This application is called from within CoreMedia Studio when a form needs to be edited. This offers a simpler way to configure and extend the Editor, without having to customise CoreMedia Studio itself directly. However, this means all of the enhancements made to the Webforms Editor need to be migrated to the new format. This guide focuses in particular on how these existing enhancements can be migrated successfully. For a full description of customisation options, please see the Formcentric Developer Manual.

1.1. Base configuration for the Formcentric Form Editor

Editor configuration can be completed in a settings document in the content. A global, site-wide configuration can be stored in the path */Settings/Options/Settings/FormcentricSettings*. A settings document with a full list of all parameters and their default values is part of the sample content. The global settings can be overwritten or extended locally on a site- or form-specific basis. The Editor starts with a default configuration if neither a global nor a local configuration is found.

1.2. Adding an additional form element

The customer-specific form elements for the Formcentric Editor are configured as a JSON string. For each customer-specific form element, add an entry into the configuration file *fields_custom.js*.

```
[
  {
    type: 'customerSpecificFormField',
    properties: {
      general: [
        {
          title: 'label',
          type: 'text',
          ...
```

The properties from the existing field can be simply transferred over to the new configuration. The previous implementation of the above configuration was as follows:

```
<TextField fieldLabel="Beschriftung">
  <plugins>
    <ui:BindPropertyPlugin
     bindTo="{ValueExpressionFactory.createFromValueHolder(
     new FormChildElementValueHolder(
     config.formElementValueExpression,'label')
     )}"
```

```
     bidirectional="true"/>
  </plugins>
</TextField>
```

From the use of the component *TextField*, we now obtain the setting *type: 'text'*. The *title* setting is used to configure the key under which the property will be stored in the form definition. This information can be found in the configuration for the ValueExpression – which is *label* in the above example. The FieldLabel is stored in the corresponding language file. In this case, the key would be *customerSpecificFormField.label*.

1.2.1. Icon form element

For the icon, the previous approach was to configure the icon class in the Editor's root element with the *icon* attribute.

```
iconCls="{resourceManager.getString(
'com.monday.webforms.blueprint.icons.WebformsBlueprintIcons',
'customerSpecificFormField')}">
```

However, this icon class can no longer be used for the new Editor. Instead, the icon must be provided as an SVG file. The icon property is set to the filename without extension.

```
{
  icon: 'customField',
  ...
```

1.2.2. Registering the new form element

The registration step for the form element is no longer needed in the Formcentric Editor, as the form element type is specified in the configuration. In the previous version, the type was used as a key in order to register the class of the form field editor.

```
FormFieldsRegistry.addFormFieldClasses(
    {customerSpecificFormField:
        'com.formcentric.blueprint.studio.CustomField'} );
```

The *customerSpecificFormField* type, which was previously used in the FormField-Registry, is now simply used as a type when the form element is configured.

```
[
    {
        ...
        type: 'customerSpecificFormField',
```

```
        properties: {
            general: [
            ...
```

1.3. Implementing an action

To configure the customer-specific actions, you extend the file *actions_custom.js*. When migrating the properties, proceed in the same way as for the form fields.

1.4. Changing format definitions

The format definitions are configured directly on the respective field. The *format* property of the *dropdown_format* type is provided for this purpose.

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. Please see the Formcentric Developer Manual for the available field types.

1.5. Extending form validation in Studio

As before, form definition validation is completed server-side in the Studio Server app. No modifications are needed here.

1.6. Displaying validation messages

Modifications are no longer needed to display validation messages in the Formcentric Editor.