Flow Form Framework Documentation

Release 2.0

The Neos Team

May 17, 2019

Contents

This documentation covering version 2.0 has been rendered at: May 17, 2019

CHAPTER 1

Forms

The Form API is an extensible and flexible framework to **build web forms**.

It includes the following features:

- Ready-to-use standard form elements based on the Twitter Bootstrap CSS framework
- Creation of **Multi-Step Forms**
- Server- and Client-Side **Validation**
- highly **flexible rendering** of form elements, based on Fluid
- **Extensible** with new form elements
- robust, **object oriented API**
- great integration into **TYPO3 Flow**

Furthermore, we built a **Form Builder** which is a web application for graphically assembling and modifying forms, displayed below:

CHAPTER 2

About This Documentation

This documentation contains a number of tutorial-style guides which will explain various aspects of the Form API and the Form Framework. It is not intended as in-depth reference, although there will be links to the in-depth API reference at various points.

2.1 Quickstart

After working through this guide, you will have learned:

- the structure of a form
- creating a form using the API
- rendering a form
- adding validation rules
- invoking actions after the form is finished

2.1.1 API Overview

The image below shows the high-level API overview of the this package.

First, we will dive into the API part *defining a form*, and then go over to *rendering a form*. In later chapters, we will also show how loading / saving a form definition works.
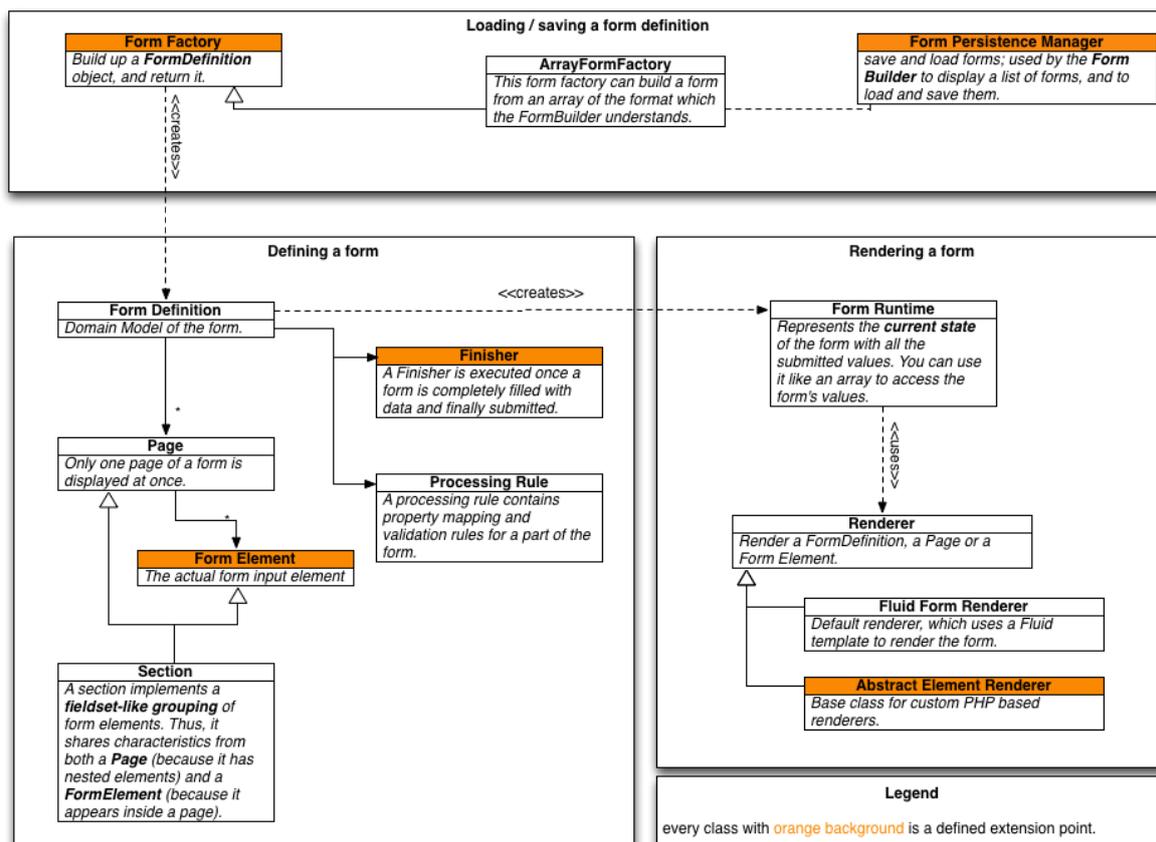
2.1.2 Anatomy of a Form

A form is described by the so-called FormDefinition, which is a container object for the form that consists of one or more `Pages` in which the actual `FormElements` are located.

As an example, let's take a basic contact form with the following structure:

- **Contact Form** *(Form)*
    - **Page 01** *(Page)*
        * Name *(Single-line Text)*
        * Email *(Single-line Text)*
        * Message *(Multi-line Text)*

Our form just has a single page that contains three input fields: Name, Email and Message.

Note: Every form needs to have at least one page.

Further Information

In-depth information about the form structure can be found in the FormDefinition API Documentation.

2.1.3 Create your first form

Now, let's try to create the basic contact form from above. For this we need to implement a so-called FormFactory, which is responsible for creating the form.

Note: The package comes with a ready-to-use factory for building forms based on YAML files describing the forms. See *Configuring form rendering with YAML* for details.

If you want to build a form with PHP, the skeleton for building a form looks as follows:

```php
namespace Your\Package;

use TYPO3\Flow\Annotations as Flow;
use TYPO3\Form\Core\Model\FormDefinition;

class QuickstartFactory extends \TYPO3\Form\Factory\AbstractFormFactory {

        /**
         * @param array $factorySpecificConfiguration
         * @param string $presetName
         * @return \TYPO3\Form\Core\Model\FormDefinition
         */
        public function build(array $factorySpecificConfiguration, $presetName) {
                $formConfiguration = $this->getPresetConfiguration($presetName);
                $form = new FormDefinition('yourFormIdentifier',
↪$formConfiguration);
                // Now, build your form here
                return $form;
        }
}
```

As you can see there is the `build()` method that you have to implement; and this method needs to return the FormDefinition.

Lets add the one page and input fields for *name*, *email* and *message* of our contact form:

```php
public function build(array $factorySpecificConfiguration, $presetName) {
        $formConfiguration = $this->getPresetConfiguration($presetName);
        $form = new FormDefinition('contactForm', $formConfiguration);

        $page1 = $form->createPage('page1');

        $name = $page1->createElement('name', 'TYPO3.Form:SingleLineText');
        $name->setLabel('Name');

        $email = $page1->createElement('email', 'TYPO3.Form:SingleLineText');
        $email->setLabel('Email');

        $comments = $page1->createElement('message', 'TYPO3.Form:MultiLineText');
        $comments->setLabel('Message');

        return $form;
}
```

You see that we used the API method `createPage($identifier)`, which creates a new page inside the form object and returns it for further use. Then, we used `createElement($identifier, $type)` to create the form elements and set some options on them.

The `$identifier` is used to identify a form element, thus it needs to be unique across the whole form. `$type` references an **Element Type**.

Note: By default the `$identifier` is part of the *id* attribute of the rendered Form Element so it should be lowerCamelCased and must not contain special characters.

Tip: As you will learn in the next guide, you can define your own **Element Types** easily. The element types referenced above (`TYPO3.Form:SingleLineText` and `TYPO3.Form:MultiLineText`) are just element types which are delivered by default by the framework.

2.1.4 Render a form

Now that we have created the first FormDefinition how can we display the actual form? That is really easy with the provided form:render ViewHelper:

```
{namespace form=TYPO3\Form\ViewHelpers}
<form:render factoryClass="Your\Package\YourFactory" />
```

If you put that snippet in your Fluid template and replace *YourPackage* with your package namespace and *YourFactory* with the class name of the previously generated form factory, you should see a form consisting of the three text fields and a submit button.

But as you can see, none of the fields are required and the email address is not verified. So let's add some basic validation rules:

2.1.5 Validation

Every FormElement implements the FormElementInterface which provides a convenient way to work with TYPO3 Flow validators:

```
$name->addValidator(new \TYPO3\Flow\Validation\Validator\NotEmptyValidator());

$email->addValidator(new \TYPO3\Flow\Validation\Validator\NotEmptyValidator());
$email->addValidator(new \TYPO3\Flow\Validation\Validator\EmailAddressValidator());

$comments->addValidator(new \TYPO3\Flow\Validation\Validator\NotEmptyValidator());
$comments->addValidator(new
↪\TYPO3\Flow\Validation\Validator\StringLengthValidator(array('minimum' => 3)));
```

With the `addValidator($validator)` method you can attach one or more validators to a form element. If you save the changes and reload the page where you embedded the form, you can see that all text fields are required now, that the email address is syntactically verified and that you need to write a message of at least 3 characters. If you try to submit the form with invalid data, validation errors are displayed next to each erroneous field.

If you do enter name, a valid email address and a message you can submit the form - and see a blank page. That's where so called **Finishers** come into play.

2.1.6 Finishers

A Finisher is a piece of PHP code that is executed as soon as a form has been successfully submitted (if the last page has been sent and no validation errors occurred).

You can attach multiple finishers to a form.

For this example we might want to send the data to an email address, and we can use the EmailFinisher for that:

```php
$emailFinisher = new \TYPO3\Form\Finishers\EmailFinisher();
$emailFinisher->setOptions(array(
        'templatePathAndFilename' => 'resource://Your.Package/Private/Templates/
↪ContactForm/NotificationEmail.txt',
        'recipientAddress' => 'your@example.com',
        'senderAddress' => 'mailer@example.com',
        'replyToAddress' => '{email}',
        'carbonCopyAddress' => 'copy@example.com',
        'blindCarbonCopyAddress' => 'blindcopy@example.com',
        'subject' => 'Contact Request',
        'format' => \TYPO3\Form\Finishers\EmailFinisher::FORMAT_PLAINTEXT
));
$form->addFinisher($emailFinisher);
```

And afterwards we want to redirect the user to some confirmation action, thus we add the RedirectFinisher:

```php
$redirectFinisher = new \TYPO3\Form\Finishers\RedirectFinisher();
$redirectFinisher->setOptions(
        array('action' => 'confirmation')
);
$form->addFinisher($redirectFinisher);
```

2.1.7 Summary

That's it for the quickstart. The complete code of your form factory should look something like this now:

```php
namespace Your\Package;

use TYPO3\Flow\Annotations as Flow;
use TYPO3\Form\Core\Model\FormDefinition;

/**
 * Flow\Scope("singleton")
 */
class QuickstartFactory extends \TYPO3\Form\Factory\AbstractFormFactory {

        /**
         * @param array $factorySpecificConfiguration
         * @param string $presetName
         * @return \TYPO3\Form\Core\Model\FormDefinition
         */
        public function build(array $factorySpecificConfiguration, $presetName) {
                $formConfiguration = $this->getPresetConfiguration($presetName);
                $form = new FormDefinition('contactForm', $formConfiguration);

                $page1 = $form->createPage('page1');

                $name = $page1->createElement('name', 'TYPO3.Form:SingleLineText');
                $name->setLabel('Name');
                $name->addValidator(new␣
↪\TYPO3\Flow\Validation\Validator\NotEmptyValidator());

                $email = $page1->createElement('email', 'TYPO3.Form:SingleLineText
↪');
                $email->setLabel('Email');
                $email->addValidator(new␣
↪\TYPO3\Flow\Validation\Validator\NotEmptyValidator());
                $email->addValidator(new␣
↪\TYPO3\Flow\Validation\Validator\EmailAddressValidator());
```

(continues on next page)

```
                $comments = $page1->createElement('message', 'TYPO3.
↪Form:MultiLineText');
                $comments->setLabel('Message');
                $comments->addValidator(new␣
↪\TYPO3\Flow\Validation\Validator\NotEmptyValidator());
                $comments->addValidator(new␣
↪\TYPO3\Flow\Validation\Validator\StringLengthValidator(array('minimum' => 3)));

                $emailFinisher = new \TYPO3\Form\Finishers\EmailFinisher();
                $emailFinisher->setOptions(array(
                        'templatePathAndFilename' => 'resource://Your.Package/
↪Private/Templates/ContactForm/NotificationEmail.txt',
                        'recipientAddress' => 'your@example.com',
                        'senderAddress' => 'mailer@example.com',
                        'replyToAddress' => '{email}',
                        'carbonCopyAddress' => 'copy@example.com',
                        'blindCarbonCopyAddress' => 'blindcopy@example.com',
                        'subject' => 'Contact Request',
                        'format' => \TYPO3\Form\Finishers\EmailFinisher::FORMAT_
↪PLAINTEXT
                ));
                $form->addFinisher($emailFinisher);

                $redirectFinisher = new \TYPO3\Form\Finishers\RedirectFinisher();
                $redirectFinisher->setOptions(
                        array('action' => 'confirmation')
                );
                $form->addFinisher($redirectFinisher);

                return $form;
        }
}
```

2.1.8 Next Steps

Now, you know how to build forms using the API. In the next tutorial, you will learn how to adjust the form output and create new form elements – all without programming!

Continue with: *Adjusting Form Output*

2.2 Configuring form rendering with YAML

2.2.1 Setup

To render a form based on a YAML configuration file, simply use the TYPO3.Form `render` ViewHelper. It uses the `TYPO3\Form\Factory\ArrayFormFactory` by default, which needs to know where the form configuration is stored. This is done in `Settings.yaml`:

```
TYPO3:
  Form:
    yamlPersistenceManager:
      savePath: 'resource://AcmeCom.SomePackage/Private/Form/'
```

From now on, every YAML file stored there can be loaded by using the filename as the persistence identifier given to the `render` ViewHelper. So if you have a file named `contact.yaml`, it can be rendered with:

```
<form:render persistenceIdentifier="contact"/>
```

2.2.2 Form configuration

Generally speaking, the configuration is a nested structure that contains the keys `type`, `identifier` and `renderables` and further options (e.g. `label`) depending on the type of the current level.

The element types referenced below (`TYPO3.Form:SingleLineText` and `TYPO3.Form:MultiLineText`) are just element types which are delivered by default by the framework. All available types can be found in the settings of the TYPO3.Form package under `TYPO3.Form.presets.default.formElementTypes`.

On the top level, the `finishers` can be configured as an array of `identifier` and `options` keys. The available options depend on the finisher being used.

Let us examine the configuration for a basic contact form with the following structure:

- **Contact Form** *(Form)*
    - **Page 01** *(Page)*
        * Name *(Single-line Text)*
        * Email *(Single-line Text)*
        * Message *(Multi-line Text)*

The following YAML is stored as `contact.yaml`:

```yaml
type: 'TYPO3.Form:Form'
identifier: 'contact'
label: 'Contact form'
renderables:
  -
    type: 'TYPO3.Form:Page'
    identifier: 'page-one'
    renderables:
      -
        type: 'TYPO3.Form:SingleLineText'
        identifier: name
        label: 'Name'
        validators:
          - identifier: 'TYPO3.Flow:NotEmpty'
      -
        type: 'TYPO3.Form:SingleLineText'
        identifier: email
        label: 'Email'
        validators:
          - identifier: 'TYPO3.Flow:NotEmpty'
          - identifier: 'TYPO3.Flow:EmailAddress'
      -
        type: 'TYPO3.Form:MultiLineText'
        identifier: message
        label: 'Message'
        validators:
          - identifier: 'TYPO3.Flow:NotEmpty'
finishers:
  -
    identifier: 'TYPO3.Form:Email'
    options:
      templatePathAndFilename: resource://AcmeCom.SomePackage/Private/Templates/
→Form/Contact.txt
      subject: '{subject}'
```

```
        recipientAddress: 'info@acme.com'
        recipientName: 'Acme Customer Care'
        senderAddress: '{email}'
        senderName: '{name}'
        format: plaintext
```

2.3 Adjusting Form Output

After working through this guide, you will have learned:

- how to adjust the form output

- how to create custom Form Presets

- how to create custom form elements

Generally, this guide answers the question: **How can form output be modified without programming?**

2.3.1 Presets Explained

In the *Quickstart* guide, you have seen how a basic form can be built. We will now dissect the form element creation a little more, and explain the lines which you might not have understood yet.

Let's have a look at the boilerplate code inside the form factory again:

```php
public function build(array $factorySpecificConfiguration, $presetName) {
        $formConfiguration = $this->getPresetConfiguration($presetName);
        $form = new FormDefinition('contactForm', $formConfiguration);
        // ...
}
```

You see that the second parameter is a `$presetName` which is passed to `getPresetConfiguration()`. So, let's introduce the concept of *presets* now.

A **Preset** is a container for pre-defined form configuration, and is the basic way to adjust the form's output. Presets are defined inside the `Settings.yaml` file, like in the following example:

```yaml
TYPO3:
  Form:
    presets:
      preset1:
        title: 'My First Preset'
        formElementTypes:
          'TYPO3.Form:SingleLineTextfield':
            # configuration for the single line textfield
      preset2:
        title: 'My Second Preset'
        parentPreset: 'preset1'
        # because preset2 *inherits* from preset1, only the changes between
        # preset1 and preset2 need to be defined here.
```

The above example defines two presets (`preset1` and `preset2`). Because `preset2` defines a `parentPreset`, it **inherits** all options from `preset1` if not specified otherwise.

Tip: The TYPO3.Form package already defines a preset with the name `default` which contains all standard form elements. Look into `TYPO3.Form/Configuration/Settings.yaml` for the details on the defined form elements.

In most cases, you will create a sub-preset of the `default` preset, modifying only the parts you need.

The method `getPresetConfiguration($presetName)` in AbstractFormFactory evaluates the preset inheritance hierarchy and returns a merged array of the preset configuration.

2.3.2 Form Element Types Explained

Now that we have seen that presets can inherit from each other, let's look *inside* the preset configuration. One particularily important part of each preset configuration is the *form element type* definition, which configures each form element correctly.

As an example, let's create a text field with the following snippet:

```
$name = $page1->createElement('name', 'TYPO3.Form:SingleLineText');
$name->setLabel('Name');
```

In the above example, the form element type is `TYPO3.Form:SingleLineText`, and when creating the form element, it *applies all default values* being set inside the form element type. As an example, take the following type definition:

```
'TYPO3.Form:SingleLineText':
  defaultValue: 'My Default Text'
  properties:
    placeholder: 'My Placeholder Text'
```

That's exactly the same as if one wrote the following PHP code:

```
$name->setDefaultValue('My Default Text');
$name->setProperty('placeholder', 'My Placeholder Text');
```

So `$page->createElement($identifier, $formElementType)` is essentially a very specialized *factory method*, which automatically applies the default values from the *form element definition* on the newly created form object before returning it.

Tip: The defaults are not only applied on single form elements, but also on the FormDefinition and Page objects. The FormDefinition object has, by convention, the *form element type* `TYPO3.Form:Form`, but you can also override it by passing the to-be-used type as third parameter to the constructor of FormDefinition.

A *page* has, by default, the *form element type* `TYPO3.Form:Page`, and you can override it by supplying a second parameter to the `createPage()` method of FormDefinition.

2.3.3 Supertypes

Now, there's one more secret ingredient which makes the form framework powerful: Every form element type can have one or multiple **supertypes**; and this allows to only specify the differences between the "parent" form element and the newly created one, effectively creating an inheritance hierarchy of form elements.

The following example demonstrates this:

```
'TYPO3.Form:SingleLineText':
  defaultValue: 'My Default Text'
  properties:
    placeholder: 'My Placeholder Text'
'TYPO3.Form:SpecialText':
  superTypes:
    'TYPO3.Form:SingleLineText' : TRUE
  defaultValue: 'My special text'
```

Here, the `SpecialText` inherits the `placeholder` property from the `SingleLineText` and only overrides the default value.

Together, presets (with parent presets) and form element types (with supertypes) form a very flexible foundation to customize the rendering in any imaginable way, as we will explore in the remainder of this guide.

Note: If multiple super types are specified, they are evaluated from *left to right*, i.e. later super types override previous definitions.

Previously the superTypes configuration was just a simple list of strings:

```
'TYPO3.Form:SpecialText':
  superTypes:
    'TYPO3.Form:SingleLineText': TRUE
  defaultValue: 'My special text'
```

But this made it impossible to *unset* a super type from a 3rd party package. The old syntax is still supported but is deprecated and might be removed in future versions.

2.3.4 Creating a Custom Preset

First, we create a sub-preset inheriting from the `default` preset. For that, open up `Your.Package/Configuration/Settings.yaml` and insert the following contents:

```
TYPO3:
  Form:
    presets:
      myCustom:
        title: 'Custom Elements'
        parentPreset: 'default'
```

You now created a sub preset named `myCustom` which behaves exactly the same as the default preset. If you now specify the preset name inside the `<form:render>` ViewHelper you will not see any differences yet:

```
<form:render factoryClass="..." presetName="myCustom" />
```

Now we are set up to modify the custom preset, and can adjust the form output.

2.3.5 Adjusting a Form Element Template

The templates of the default Form Elements are located in `TYPO3.Form/Resources/Private/Form/`. They are standard Fluid templates and most of them are really simple. Open up the `Single-Line Text` template for example:

```
<f:layout name="TYPO3.Form:Field" />
<f:section name="field">
  <f:form.textfield property="{element.identifier}" id="{element.uniqueIdentifier}
↪"
                    placeholder="{element.properties.placeholder}" errorClass=
↪"error" />
</f:section>
```

As you can see, the Form Element templates use layouts in order to reduce duplicated markup.

Tip: The Fluid Form Renderer expects layout and partial names in the format `<PackageKey>:<Name>`. That makes it possible to reference layouts and partials from other packages!

We'll see how to change the layout in the next section. For now let's try to simply change the `class` attribute of the *SingleLineText* element.

For that, copy the default template to `Your.Package/Private/Resources/CustomElements/SingleLineText.html` and adjust it as follows:

```
<f:layout name="TYPO3.Form:Field" />
<f:section name="field">
    <f:form.textfield property="{element.identifier}" id="{element.uniqueIdentifier}
→"
                   placeholder="{element.properties.placeholder}" errorClass=
→"error"
                   class="customClass" />
</f:section>
```

Now, you only need to tell the framework to use your newly created template instead of the default one. This can be archieved by overriding the rendering option `templatePathPattern` in the *form element type definition*.

Adjust `Your.Package/Configuration/Settings.yaml` accordingly:

```
TYPO3:
  Form:
    presets:
      myCustom:
        title: 'Custom Elements'
        parentPreset: 'default'
        formElementTypes:
          'TYPO3.Form:SingleLineText':
            renderingOptions:
              templatePathPattern: 'resource://Your.Package/Private/CustomElements/
→SingleLineText.html'
```

Now, all `Single-Line Text` elements will have a class attribute of `customClass` when using the `myCustom` preset.

A more realistic use-case would be to change the arrangement of form elements. Read on to see how you can easily change the layout of a form.

2.3.6 Changing The Form Layout

By default, validation errors are rendered next to each form element. Imagine you want to render validation errors of the current page *above* the form instead. For this you need to adjust the previously mentioned **field layout**.

The provided default field layout located in `TYPO3.Form/Resources/Private/Form/Layouts/Field.html` is a bit more verbose as it renders the label, validation errors and an asterisk if the element is required (we slightly reformatted the template here to improve readability):

```
{namespace form=TYPO3\Form\ViewHelpers}
<f:form.validationResults for="{element.identifier}">
  <!-- wrapping div for the form element; contains an identifier for the form
→element if we are
    in preview mode -->
  <div class="clearfix{f:if(condition: validationResults.flattenedErrors, then: '
→error')}"
      <f:if condition="{element.rootForm.renderingOptions.previewMode}">
        data-element="{form:form.formElementRootlinePath(renderable:element)}"
      </f:if>
  >
    <!-- Label for the form element, and required indicator -->
    <label for="{element.uniqueIdentifier}">{element.label -> f:format.nl2br()}
      <f:if condition="{element.required}">
        <f:render partial="TYPO3.Form:Field/Required" />
```

(continues on next page)

```
        </f:if>
     </label>

     <!-- the actual form element -->
     <div class="input">
        <f:render section="field" />

        <!-- validation errors -->
        <f:if condition="{validationResults.flattenedErrors}">
           <span class="help-inline">
              <f:for each="{validationResults.errors}" as="error">
                 {error -> f:translate(id: error.code, arguments: error.arguments,
                                       package: 'TYPO3.Form', source:
↪'ValidationErrors')}
                 <br />
              </f:for>
           </span>
        </f:if>
     </div>
   </div>
</f:form.validationResults>
```

Copy the layout file to `Your.Package/Private/Resources/CustomElements/Layouts/Field.html` and remove the validation related lines:

```
{namespace form=TYPO3\Form\ViewHelpers}
<f:form.validationResults for="{element.identifier}">
   <!-- wrapping div for the form element; contains an identifier for the form
↪element if we are
      in preview mode -->
   <div class="clearfix{f:if(condition: validationResults.flattenedErrors, then: '
↪error')}"
        <f:if condition="{element.rootForm.renderingOptions.previewMode}">
           data-element="{form:form.formElementRootlinePath(renderable:element)}"
        </f:if>
   >
     <!-- Label for the form element, and required indicator -->
     <label for="{element.uniqueIdentifier}">{element.label -> f:format.nl2br()}
        <f:if condition="{element.required}">
           <f:render partial="TYPO3.Form:Field/Required" />
        </f:if>
     </label>

     <!-- the actual form element -->
     <div class="input">
        <f:render section="field" />
     </div>
   </div>
</f:form.validationResults>
```

Additionally you need to adjust the default form template located in `TYPO3.Form/Resources/Private/Form/Form.html` (remember that a FormDefinition also has a form element type, by default of `TYPO3.Form:Form`), which looks as follows by default:

```
{namespace form=TYPO3\Form\ViewHelpers}
<form:form object="{form}" action="index" method="post" id="{form.identifier}"
        enctype="multipart/form-data">
   <form:renderRenderable renderable="{form.currentPage}" />
   <div class="actions">
      <f:render partial="TYPO3.Form:Form/Navigation" arguments="{form: form}" />
   </div>
```

```
</form:form>
```

Copy this template file to `Your.Package/Private/Resources/CustomElements/Form.html` and add the validation result rendering:

```
{namespace form=TYPO3\Form\ViewHelpers}
<form:form object="{form}" action="index" method="post" id="{form.identifier}"
           enctype="multipart/form-data">
   <f:form.validationResults>
      <f:if condition="{validationResults.flattenedErrors}">
         <ul class="error">
            <f:for each="{validationResults.flattenedErrors}" as="elementErrors"
                   key="elementIdentifier" reverse="true">
               <li>
                  {elementIdentifier}:
                  <ul>
                     <f:for each="{elementErrors}" as="error">
                        <li>{error}</li>
                     </f:for>
                  </ul>
               </li>
            </f:for>
         </ul>
      </f:if>
   </f:form.validationResults>
   <form:renderRenderable renderable="{form.currentPage}" />
   <div class="actions">
      <f:render partial="TYPO3.Form:Form/Navigation" arguments="{form: form}" />
   </div>
</form:form>
```

Now, you only need to adjust the form definition in order to use the new templates:

```
TYPO3:
  Form:
  presets:
    ########## CUSTOM PRESETS ##########

    myCustom:
      title: 'Custom Elements'
      parentPreset: 'default'
      formElementTypes:

        # ...

        ### override template path of TYPO3.Form:Form ###
        'TYPO3.Form:Form':
          renderingOptions:
            templatePathPattern: 'resource://TYPO3.FormExample/Private/
→CustomElements/Form.html'

        ### override default layout path ###
        'TYPO3.Form:Base':
          renderingOptions:
            layoutPathPattern: 'resource://TYPO3.FormExample/Private/
→CustomElements/Layouts/{@type}.html'
```

Tip: You can use **placeholders** in `templatePathPattern`, `partialPathPattern` and `layoutPathPattern`: `{@package}` will be replaced by the package key and `{@type}` by the current form element type without namespace. A small example shall illustrate this:

If the form element type is `Your.Package:FooBar`, then `{@package}` is replaced by `Your.Package`, and `{@type}` is replaced by `FooBar`. As partials and layouts inside form elements are also specified using the `Package:Type` notation, this replacement also works for partials and layouts.

2.3.7 Creating a New Form Element

With the Form Framework it is really easy to create additional Form Element types. Lets say you want to create a specialized version of the `TYPO3.Form:SingleSelectRadiobuttons` that already provides two radio buttons for `Female` and `Male`. That's just a matter of a few lines of yaml:

```
TYPO3:
  Form:
    presets:
        ########## CUSTOM PRESETS ##########

      myCustom:
        title: 'Custom Elements'
        parentPreset: 'default'
        formElementTypes:

          # ...

          'Your.Package:GenderSelect':
            superTypes:
              'TYPO3.Form:SingleSelectRadiobuttons': TRUE
            renderingOptions:
              templatePathPattern: 'resource://TYPO3.Form/Private/Form/
→SingleSelectRadiobuttons.html'
            properties:
              options:
                f: 'Female'
                m: 'Male'
```

As you can see, you can easily extend existing Form Element Definitions by specifying the `superTypes`.

Tip: We have to specify the `templatePathPattern` because according to the default path pattern the template would be expected at `Your.Package/Private/Resources/Form/GenderSelect.html` otherwise.

Note: Form Elements will only be available in the preset they're defined (and in it's sub-presets). Therefore you should consider adding Form Elements in the `default` preset to make them available for all Form Definitions extending the default preset.

2.4 Extending Form API

After working through this guide, you will have learned:

- how to create custom PHP based Form Element implementations
- how to create a custom Form Element renderer

Generally, this guide answers the question: **How can the form output be modified with programming?**

2.4.1 Custom PHP-based Form Elements

In the previous guides you have learned how to create custom Form Elements without writing a single line of PHP. While this is sufficient for most cases where you mainly want to change the visual representation or create a *specialized* version of an already existing element, there are situations where you want to adjust the *Server-side* behavior of an element. This is where you want to get your hands dirty and create custom Form Element implementations. Examples for such custom Form Elements are:

- A *DatePicker* that converts the input to a `DateTime` object
- A *File upload* that validates and converts an uploaded file to a `Resource`
- A *Captcha* image

A Form Element must implement the `FormElementInterface` interface located in `TYPO3.Form/Classes/Core/Model/FormElementInterface.php`.

Tip: Usually you want to extend the provided `AbstractFormElement` which already implements most of the methods of the interface.

Most commonly you create custom Form elements in order to preconfigure the so called `Processing Rule` which defines validation and property mapping instructions for an element. Lets have a look at the `DatePicker` Form Element located in `TYPO3.Form/Classes/FormElements/DatePicker.php`:

```
class DatePicker extends \TYPO3\Form\Core\Model\AbstractFormElement {
    public function initializeFormElement() {
        $this->setDataType('DateTime');
    }
}
```

The method `initializeFormElement()` is called whenever a Form Element is **added to a form**. In this example, we only set the target data type to a DateTime object. This way, property mapping and type conversion using the registered TypeConverters is automatically triggered.

Besides being able to modify the Form Element configuration during *initialization* you can also implement the callbacks `beforeRendering()` or/and `onSubmit()` in order to adjust the behavior or representation of the element at *runtime*. Lets create a new Form Element that is required only if another form field has been specified (for example a "subscribe to newsletter" checkbox that requires you to provide an email address if checked). For this create a new PHP class at `Your.Package/Classes/FormElements/ConditionalRequired.php`:

```
namespace Your\Package\FormElements;

class ConditionalRequired extends \TYPO3\Form\Core\Model\AbstractFormElement {

    /**
     * Executed before the current element is outputted to the client
     *
     * @param \TYPO3\Form\Core\Runtime\FormRuntime $formRuntime
     * @return void
     */
    public function beforeRendering(\TYPO3\Form\Core\Runtime\FormRuntime
→$formRuntime) {
        $this->requireIfTriggerIsSet($formRuntime->getFormState());
    }

    /**
     * Executed after the page containing the current element has been submitted
     *
     * @param \TYPO3\Form\Core\Runtime\FormRuntime $formRuntime
     * @param mixed $elementValue raw value of the submitted element
```

```
     * @return void
     */
    public function onSubmit(\TYPO3\Form\Core\Runtime\FormRuntime $formRuntime, &
→$elementValue) {
        $this->requireIfTriggerIsSet($formRuntime->getFormState());
    }


    /**
     * Adds a NotEmptyValidator to the current element if the "trigger" value is
→not empty.
     * The trigger can be configured with $this->properties['triggerPropertyPath']
     *
     * @param \TYPO3\Form\Core\Runtime\FormState $formState
     * @return void
     */
    protected function requireIfTriggerIsSet(\TYPO3\Form\Core\Runtime\FormState
→$formState) {
        if (!isset($this->properties['triggerPropertyPath'])) {
            return;
        }
        $triggerValue = $formState->getFormValue($this->properties[
→'triggerPropertyPath']);
        if ($triggerValue === NULL || $triggerValue === '') {
            return;
        }
        $this->addValidator(new
→\TYPO3\Flow\Validation\Validator\NotEmptyValidator());
    }
}
```

beforeRendering() is invoked just before a Form Element is actually outputted to the client. It receives a reference to the current FormRuntime making it possible to access previously submitted values.

onSubmit() is called whenever the page containing the current Form Element is submitted. to the server. In addition to the FormRuntime this callback also gets passed a reference to the raw value of the submitted element value before property mapping and validation rules were applied.

In order to use the new Form Element type you first have to extend the Form Definition and specify the implementationClassName option:

```
TYPO3:
  Form:
    presets:
      somePreset:
        # ...
        formElementTypes:
          'TYPO3.FormExample:ConditionalRequired':
            superTypes:
              'TYPO3.Form:FormElement': TRUE
            implementationClassName:
→'TYPO3\FormExample\FormElements\ConditionalRequired'
            renderingOptions:
              templatePathPattern: 'resource://TYPO3.Form/Private/Form/
→SingleLineText.html'
```

This makes the new Form Element TYPO3.FormExample:ConditionalRequired available in the preset somePreset and you can use it as follows:

```
$form = new FormDefinition('myForm', $formDefaults);

$page1 = $form->createPage('page1');
```

```php
$newsletter = $page1->createElement('newsletter', 'TYPO3.Form:Checkbox');
$newsletter->setLabel('Subscribe for Newsletter');

$email = $page1->createElement('email', 'TYPO3.FormExample:ConditionalRequired');
$email->setLabel('E-Mail');
$email->setProperty('triggerPropertyPath', 'newsletter');
```

The line `$email->setProperty('triggerPropertyPath', 'newsletter');` makes the `email` Form Element required depending on the value of the `newsletter` element.

This example is really simple but it demonstrates how you can profoundly interact with the Form handling at every level.

2.4.2 Custom Form Element Renderers

By default a form and all its elements are rendered with the `FluidFormRenderer` which is a specialized version of the `Fluid TemplateView`. For each renderable Form Element there exists an corresponding Fluid template. The template path can be changed for all or specific Form Elements as well as layout and partial paths, so the default renderer is flexible enough to cover most scenarios. However if you want to use your own templating engine or don't want to render HTML forms at all (think of Flash or CLI based forms) you can implement your own Renderer and use it either for the complete form or for certain Form Elements.

As a basic example we want to implement a `ListRenderer` that simply outputs specified items as unordered list. A Form Element Renderer must implement the `RendererInterface` interface located in `TYPO3.Form/Classes/Core/Renderer/RendererInterface.php` and usually you want to extend the provided `AbstractRenderer` which already implements most of the methods of the interface:

```php
namespace Your\Package\Renderers;

class ListRenderer extends \TYPO3\Form\Core\Renderer\AbstractElementRenderer {

    /**
     * @param \TYPO3\Form\Core\Model\Renderable\RootRenderableInterface $renderable
     * @return string
     */
    public function
        renderRenderable(\TYPO3\Form\Core\Model\Renderable\RootRenderableInterface
        $renderable) {
        $items = array();
        if ($renderable instanceof \TYPO3\Form\Core\Model\FormElementInterface) {
            $elementProperties = $renderable->getProperties();
            if (isset($elementProperties['items'])) {
                $items = $elementProperties['items'];
            }
        }
        $content = sprintf('<h3>%s</h3>', htmlspecialchars($renderable->getLabel()));
        $content .= '<ul>';
        foreach ($items as $item) {
            $content .= sprintf('<li>%s</li>', htmlspecialchars($item));
        }
        $content .= '</ul>';
        return $content;
    }
}
```

Tip: If you write your own Renderer make sure to sanitize values with `htmlspecialchars()` before outputting them to prevent invalid HTML and XSS vulnerabilities.

2.5 Configuring Form Builder

After this guide, you will have learned how to **Configure the Form Builder through settings**

2.5.1 Adding a New Form Element Inside "Create Elements"

Let's say you have created your form element, and want to make it available inside the Form Builder. For that, you need some YAML configuration which looks as follows:

```
# we are now inside TYPO3:Form:presets:[presetName]
formElementTypes:
  'Your.Package:YourFormElement':
    # the definitions for your form element
    formBuilder:
      label: 'Your New Form Element'
      group: custom
      sorting: 200
```

To determine whether a form element is visible in the Form Builder, you must set `formBuilder:group` to a valid group. A *form element group* is used to visually group the available form elements together. In the default profile, the following groups are configured:

- input

- select

- custom

- container

The `label` is – as you might expect – the human-readable label, while the `sorting` determines the ordering of form elements inside their form element group.

2.5.2 Creating a New Form Element Group

All form element groups are defined inside `formElementGroups` inside the preset, so that's how you can add a new group:

```
# we are now inside TYPO3:Form:presets:[presetName]
formElementGroups:
  specialCustom:
    sorting: 500
    label: 'My special custom group'
```

For each group, you need to specify a human-readable `label`, and the `sorting` (which determines the ordering of the groups).

2.5.3 Setting Default Values for Form Elements

When a form element is created, you can define some default values which are directly set on the form element. As an example, let's imagine you want to build a `ProgrammingLanguageSelect` where the user can choose his favorite programming language.

In this case, we want to define some default programming languages, but the integrator who builds the form should be able to add custom options as well. These default options can be set in `Settings.yaml` using the `formBuilder:predefinedDefaults` key.

Here follows the full configuration for the `ProgrammingLanguageSelect` (which is an example taken from the `TYPO3.FormExample` package):

```
# we are now inside TYPO3:Form:presets:[presetName]
formElementTypes:
  'TYPO3.FormExample:ProgrammingLanguageSelect':
    superTypes:
      'TYPO3.Form:SingleSelectRadiobuttons': TRUE
    renderingOptions:
      templatePathPattern: 'resource://TYPO3.Form/Private/Form/
↪SingleSelectRadiobuttons.html'

      # here follow the form builder specific options
      formBuilder:
        group: custom
        label: 'Programming Language Select'

        # we now set some defaults which are applied once the form element is␣
↪inserted to the form
        predefinedDefaults:
          properties:
            options:
              0:
                _key: 'php'
                _value: 'PHP'
              1:
                _key: 'java'
                _value: 'Java etc'
              2:
                _key: 'js'
                _value: 'JavaScript'
```

Contrasting Use Case: Gender Selection

Inside *Creating a new form element*, we have implemented a special *Gender Select*. Let's think a second about the differences between the *Gender Select* and the *Programming Language Select* examples:

For a *Gender* select field, the integrator using the form builder does not need to set any options for this form element, as the available choices (`Female` and `Male`) are predefined inside the *form element template*.

In the case of the *programming language select*, we only want to set some sensible defaults for the integrator, but want him to be able to adjust the values.

Choosing which strategy to use depends mostly on the expected usage patterns:

- In the *gender select* example, if a **new option is added to the list afterwards**, this will directly be reflected in *all forms* which use this input field.

- If you use `predefinedDefaults`, changing these will be only applied to **new elements**, but not to already existing elements.

Note: In order to make the gender selection work nicely with the Form Builder, we should disable the `options` editor as follows (as the options should not be editable by the implementor):

```
# we are now inside TYPO3:Form:presets:[presetName]
formElementTypes:
  'TYPO3.FormExample:GenderSelect':
    formBuilder:
      editors:
        # Disable "options" editor
        options: null
```

> **Tip:** The same distinction between using `formBuilder:predefinedDefaults` and the form element type definition directly can also be used to add other elements like `Validators` or `Finishers`.

2.5.4 Marking Validators and Finishers As Required

Sometimes, you want to simplify the Form Builder User Interface and make certain options easier for your users. A frequent use-case is that you want that a certain validator, like the `StringLength` validator, is always shown in the user interface as it is very often used.

This can be configured as follows:

```
# we are now inside TYPO3:Form:presets:[presetName]
formElementTypes:
  'TYPO3.Form:TextMixin': # or any other type here
    formBuilder:
      editors:
        validation:
          availableValidators:
            'TYPO3.Flow:StringLength': # or any other validator
              # mark this validator required such that it is always shown.
              required: true
```

Finishers

The same works for Finishers, for example the following configuration makes the EmailFinisher mandatory:

```
# we are now inside TYPO3:Form:presets:[presetName]
formElementTypes:
  'TYPO3.Form:Form':
    formBuilder:
      editors:
        finishers:
          availableFinishers:
            'TYPO3.Form:Email': # or any other finisher
              # mark this finisher required such that it is always shown.
              required: true
```

2.5.5 Finishing Up

You should now have some receipes at hand on how to modify the Form Builder. Read the next chapter for some more advanced help.

2.6 Extending Form Builder

After working through this guide, you will have learned:

- How to include custom CSS into the form builder

- How to write a custom finisher editor

- **How can the form builder be adjusted**

An in-depth reference on how to extend the form builder using custom JavaScript can be found in the start page of the Form Builder API documentation.

2.6.1 Adjusting the Form Builder with Custom CSS

Let's say you want to adjust the form builder with a custom CSS file inside `Your.Package/Resources/Public/FormBuilderAdjustments.css`. Then, you need to tell the form builder to load this additional stylesheet as well. You can do that using an entry inside `Settings.yaml` of your package which looks as follows:

```yaml
TYPO3:
  FormBuilder:
    stylesheets:
      customAdjustments:
        files: ['resource://Your.Package/Public/FormBuilderAdjustments.css']
        sorting: 200
```

Most important is the `sorting` property, as it defines the *order in which the CSS files are included*. Every sorting up to 100 is reserved for internal use by the form builder, so you should use sorting numbers above 100 unless you have a good reason to do otherwise.

Tip: Loading additional JavaScript files into the form builder works in the same manner.

2.6.2 Overriding Form Builder Handlebars Template

Let's say we want to adjust the header of the form builder, such that it displays your company up there as well. For that, we need to modify the default *handlebars template* for the header area.

Warning: If you modify handlebars templates, you might need to adjust them after a new version of the form builder has been released! Modification of handlebars templates is useful for **unplanned extensibility**, but you should only do it as last resort!

The default template is located inside `TYPO3.FormBuilder/Resources/Private/FormBuilderTemplates/Header.html` and looks as follows:

```html
<h1>Form Builder – {{TYPO3.FormBuilder.Model.Form.formDefinition.label}}</h1>
{{#if TYPO3.FormBuilder.Model.Form.currentlyLoadingPreview}}
    <span id="typo3-formbuilder-loading">Loading..</span>
{{/if}}

<ul id="typo3-formbuilder-toolbar">
    <li class="typo3-formbuilder-preset">
        {{view TYPO3.FormBuilder.View.Header.PresetSelector}}
    </li>
    <li class="typo3-formbuilder-preview">
        {{#view TYPO3.FormBuilder.View.Header.PreviewButton class="typo3-formbuilder-
→button icon"}}Preview{{/view}}
    </li>
    <li class="typo3-formbuilder-save">
        {{#view TYPO3.FormBuilder.View.Header.SaveButton class="typo3-formbuilder-
→button icon"}}Save{{/view}}
    </li>
</ul>
```

We can just copy it to `Your.Package/Resources/Private/FormBuilderTemplates/Header.html` and adjust it as needed, modifying the part inside the `<h1>...</h1>` to:

```html
<h1>Your Company Form Builder – {{TYPO3.FormBuilder.Model.Form.formDefinition.
→label}}</h1>
```

Then, we need to tell the form builder that we want to use a different handlebars template for the header. For that, we need the following `Settings.yaml`:

```yaml
TYPO3:
  FormBuilder:
    handlebarsTemplates:
      Header: resource://Your.Package/Private/FormBuilderTemplates/Header.html
```

> **Warning:** Make sure that your package is loaded **after the FormBuilder package** if you want to override such settings.

2.6.3 Creating a Custom Editor

Every form element is edited on the right side of the Form Builder in the *element options panel*. In order to be flexible and extensible, the element options panel is a container for **editors** which, as a whole, edit the form element. There are a multitude of predefined editors, ranging from a simple text input field up to a grid widget for editing properties.

All editors for a given form element are defined inside the `formElementTypes` definition, looking as follows:

```yaml
# we are now inside TYPO3:Form:presets:[presetName]:formElementTypes
'TYPO3.Form:TextMixin':
  formBuilder:
    editors:
      placeholder: # an arbitrary key for identifying the editor instance
        sorting: 200 # the sorting determines the ordering of the different
→editors inside the element options panel
        viewName: 'JavaScript.View.Class.Name' # the JavaScript view class name
→which should be used here
        # additionally, you can define view-specific options here
    # here, you can define some more editors.
```

We will now create a custom editor for rendering a *select* box, and will add it to the *File Upload* form element such that a user can choose the file types he allows. The finished editor is part of the standard FormBuilder distribution inside `TYPO3.FormBuilder/Resources/Private/CoffeeScript/elementOptionsPanelEditors/basic.coffee`.

> **Note:** If you want to create your completely own editor, you need to include the additional JavaScript file. How this is done is explained in detail inside *Adjusting the Form Builder with Custom CSS*

The Basic Setup

> **Note:** We'll develop the editor in CoffeeScript, but you are of course free to also use JavaScript.

We will extend our editor from `TYPO3.FormBuilder.View.ElementOptionsPanel.Editor.AbstractPropertyEditor`:

```coffee
TYPO3.FormBuilder.View.ElementOptionsPanel.Editor.SelectEditor =
→AbstractPropertyEditor.extend {
  templateName: 'ElementOptionsPanel-SelectEditor'
}
```

Then, we will create a basic handlebars template and register it underneath `ElementOptionsPanel-SelectEditor` (as described in *Overriding Form Builder Handlebars Template*). We'll just copy over an existing editor template and slightly adjust it:

```
<div class="typo3-formbuilder-controlGroup">
   <label>{{label}}:</label>
   <div class="typo3-formbuilder-controls">
      [select should come here]
   </div>
</div>
```

Note: Don't forget to register the handlebars template `ElementOptionsPanel-SelectEditor` inside your `Settings.yaml`.

Now that we have all the pieces ready, let's actually use the editor inside the `TYPO3.Form:FileUpload` form element:

```
# we are now inside TYPO3:Form:presets:[presetName]:formElementTypes
'TYPO3.Form:FileUpload':
 formBuilder:
   editors:
      allowedExtensions:
         sorting: 200
         viewName: 'TYPO3.FormBuilder.View.ElementOptionsPanel.Editor.SelectEditor'
```

After reloading the form builder, you will see that the file upload field has a field: `[select should come here]` displayed inside the element options panel.

Now that we have the basics set up, let's fill the editor with life by actually implementing it.

Implementing the Editor

Everything inside here is just JavaScript development with EmberJS, using bindings and computed properties. If that sound like chinese to you, head over to the EmberJS website and read it up.

We somehow need to configure the available options inside the editor, and come up with the following YAML on how we want to configure the file types:

```
allowedExtensions:
  sorting: 200
  label: 'Allowed File Types'
  propertyPath: 'properties.allowedExtensions'
  viewName: 'TYPO3.FormBuilder.View.ElementOptionsPanel.Editor.SelectEditor'
  availableElements:
    0:
      value: ['doc', 'docx', 'odt', 'pdf']
      label: 'Documents (doc, docx, odt, pdf)'
    1:
      value: ['xls']
      label: 'Spreadsheet documents (xls)'
```

Furthermore, the above example sets the `label` and `propertyPath` options of the element editor. The `label` is shown in front of the element, and the `propertyPath` points to the form element option which shall be modified using this editor.

All properties of such an editor definition are made available inside the editor object itself, i.e. the `SelectEditor` now magically has an `availableElements` property which we can use inside the Handlebars template to bind the select box options to. Thus, we remove the `[select should come here]` and replace it with `Ember.Select`:

```
{{view Ember.Select contentBinding="availableElements" optionLabelPath="content.
↪label"}}
```

Now, if we reload, we already see the list of choices being available as a dropdown.

Saving the Selection

Now, we only need to save the selection inside the model again. For that, we bind the current selection to a property in our view using the `selectionBinding` of the `Ember.Select` view:

```
{{view Ember.Select contentBinding="availableElements" optionLabelPath="content.
↪label" selectionBinding="selectedValue"}}
```

Then, let's create a *computed property* `selectedValue` inside the editor implementation, which updates the `value` property and triggers the change notification callback `@valueChanged()`:

```
SelectEditor = AbstractPropertyEditor.extend {
   templateName: 'ElementOptionsPanel-SelectEditor'
   # API: list of available elements to be shown in the select box; each element
↪should have a "label" and a "value".
   availableElements: null

   selectedValue: ((k, v) ->
      if arguments.length >= 2
         # we need to set the value
         @set('value', v.value)
         @valueChanged()

      # get the current value
      for element in @get('availableElements')
         return element if element.value == @get('value')

      # fallback if value not found
      return null
   ).property('availableElements', 'value').cacheable()
}
```

That's it :)

2.6.4 Creating a Finisher Editor

Let's say we have implemented an *DatabaseFinisher* which has some configuration options like the table name, and you want to make these configuration options editable inside the Form Builder. This can be done using a custom handlebars template, and some configuration. In many cases, you do not need to write any JavaScript for that.

You need to do three things:

1. Register the finisher as a *Finisher Preset*

2. Configure the finisher editor for the form to include the newly created finisher as available finisher

3. create and include the handlebars template

```
TYPO3:
  Form:
    presets:
      yourPresetName: # fill in your preset name here, or "default"
        # 1. Register your finisher as finisher preset
        finisherPresets:
          'Your.Package:DatabaseFinisher':
            implementationClassName: 'Your\Package\Finishers\DatabaseFinisher'
        formElementTypes:
          'TYPO3.Form:Form':
            formBuilder:
              editors:
                finishers:
```

```yaml
            availableFinishers:
              # Configure the finisher editor for the form to include
              # the newly created finisher as available finisher
              'Your.Package:DatabaseFinisher':
                label: 'Database Persistence Finisher'
                templateName: 'Finisher-YourPackage-DatabaseFinisher'
  FormBuilder:
    handlebarsTemplates:
      # include the handlebars template
      Finisher-YourPackage-DatabaseFinisher: resource://Your.Package/Private/
↪FormBuilderTemplates/DatabaseFinisher.html
```

Now, you only need to include the appropriate Handlebars template, which could look as follows:

```html
<h4>
    {{label}}
    {{#view Ember.Button target="parentView" action="remove"
                         isVisibleBinding="notRequired"
                         class="typo3-formbuilder-removeButton"}}Remove{{/view}}
</h4>

<div class="typo3-formbuilder-controlGroup">
    <label>Database Table</label>
    <div class="typo3-formbuilder-controls">
        {{view Ember.TextField valueBinding="currentCollectionElement.options.
↪databaseTable"}}
    </div>
</div>
```

Tip: Creating a custom *validator editor* works in the same way, just that they have to be registered underneath `validatorPresets` and the editor is called `validators` instead of `finishers`.

Credits

The initial implementation has been generously sponsored by AKOM360 - Multi Channel Marketing.

It has been implemented by:

- Sebastian Kurfürst, sandstorm|media
- Bastian Waidelich, wwwision