Flow Form Framework Documentation

Release 5.0

The Neos Team

Mar 17, 2020
# Contents

1 Forms

2 About This Documentation
   2.1 Quickstart ................................................. 5
   2.2 Configuring form rendering with YAML .......................... 11
   2.3 Adjusting Form Output ..................................... 13
   2.4 Translating Forms .......................................... 20
   2.5 Extending Form API ......................................... 23

3 Credits ......................................................... 27
This documentation covering version 5.0 has been rendered at: Mar 17, 2020
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
- Creation of Multi-Step Forms
- Server- and Client-Side Validation
- Highly flexible rendering of form elements, based on Fluid or any custom renderer
- Extensible with new form elements
- Robust, object-oriented API
- Integration into Neos Flow and Neos CMS

Also, there are some useful Related Packages that make use of the extension points and provide useful additions. For example, there is the original Form YAML Builder that is a Flow-based web application for graphically assembling and modifying forms:
Since version 5 there’s also a Neos Form Builder that adds these features and more to the Neos CMS Backend:
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 Neos\Flow\Annotations as Flow;
use Neos\Form\Core\Model\FormDefinition;

class QuickstartFactory extends \Neos\Form\Factory\AbstractFormFactory {
    /**
     * @param array $factorySpecificConfiguration
     * @param string $presetName
     * @return \Neos\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.

Let’s 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', 'Neos.Form:SingleLineText');
    $name->setLabel('Name');
    $email = $page1->createElement('email', 'Neos.Form:SingleLineText');
    $email->setLabel('Email');
    $comments = $page1->createElement('message', 'Neos.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.
**Flow Form Framework Documentation, Release 5.0**

---

**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 (Neos.Form:SingleLineText and Neos.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:

```php
{namespace form=Neos\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 *Your-Factory* 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 Neos Flow validators:

```php
$name->addValidator(new \Neos\Flow\Validation\Validator\NotEmptyValidator());
$email->addValidator(new \Neos\Flow\Validation\Validator\NotEmptyValidator());
$email->addValidator(new \Neos\Flow\Validation\Validator\EmailAddressValidator());
$comments->addValidator(new \Neos\Flow\Validation\Validator\NotEmptyValidator());
$comments->addValidator(new \Neos\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:
$emailFinisher = new \Neos\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' => \Neos\Form\Finishers\EmailFinisher::FORMAT_PLAINTEXT
));
$form->addFinisher($emailFinisher);

The corresponding NotificationEmail.txt template could look like this:

```
New contact form mail:
From: {formValues.name} ({formValues.email})
Message:
{formValues.message}
```

**Note:** Form values can be accessed via `formValues.*`

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

```php
$redirectFinisher = new \Neos\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 Neos\Flow\Annotations as Flow;
use Neos\Form\Core\Model\FormDefinition;

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

    /**
     * @param array $factorySpecificConfiguration
     * @param string $presetName
     * @return \Neos\Form\Core\Model\FormDefinition
     */
    public function build(array $factorySpecificConfiguration, $presetName) {
        $formConfiguration = $this->getPresetConfiguration($presetName);
        $form = new FormDefinition('contactForm', $formConfiguration);
        $page1 = $form->createPage('page1');
    }
}
```

(continues on next page)
$name = $page1->createElement('name', 'Neos.Form:SingleLineText');
$name->setLabel('Name');
$name->addValidator(new \Neos\Flow\Validation\Validator\NotEmptyValidator());
$email = $page1->createElement('email', 'Neos.Form:SingleLineText');
$email->setLabel('Email');
$email->addValidator(new \Neos\Flow\Validation\Validator\NotEmptyValidator());
$email->addValidator(new \Neos\Flow\Validation\Validator\EmailAddressValidator());
$comments = $page1->createElement('message', 'Neos.Form:MultiLineText');
$comments->setLabel('Message');
$comments->addValidator(new \Neos\Flow\Validation\Validator\NotEmptyValidator());
$comments->addValidator(new \Neos\Flow\Validation\Validator\StringLengthValidator(array('minimum' => 3)));
$emailFinisher = new \Neos\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' => \Neos\Form\Finishers\EmailFinisher::FORMAT_PLAINTEXT
));
$form->addFinisher($emailFinisher);
$redirectFinisher = new \Neos\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 Neos.Form render ViewHelper. It uses the Neos\Form\Factory\ArrayFormFactory by default, which needs to know where the form configuration is stored. This is done in Settings.yaml:

```yaml
Neos:
  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:

```html
<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 (Neos.Form:SingleLineText and Neos.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 Neos.Form package under Neos.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: 'Neos.Form:Form'
| identifier: 'contact'
| label: 'Contact form'
| renderables:
  - type: 'Neos.Form:Page'
    identifier: 'page-one'
    renderables:
      - type: 'Neos.Form:SingleLineText'
        identifier: name
        label: 'Name'
        validators:
          - identifier: 'Neos.Flow:NotEmpty'
      - type: 'Neos.Form:SingleLineText'
        identifier: email
        label: 'Email'
```

(continues on next page)
2.2.3 File Uploads

The default preset comes with an FileUpload form element that allows the user of the form to upload arbitrary files. The EmailFinisher allows these files to be sent as attachments:

```json

type: 'Neos.Form:Form'
identifier: 'application'
label: 'Example application form'
renderables:
  -
    type: 'Neos.Form:Page'
    identifier: 'page-one'
    renderables:
      -
        type: 'Neos.Form:SingleLineText'
        identifier: email
        label: 'Email'
        validators:
          - identifier: 'Neos.Flow:NotEmpty'
          - identifier: 'Neos.Flow:EmailAddress'
    -
      type: 'Neos.Form:FileUpload'
      identifier: applicationform
      label: 'Application Form (PDF)'
      properties:
        allowedExtensions:
          - pdf
      validators:
        - identifier: 'Neos.Flow:NotEmpty'
finishers:
  -
    # Application email that is sent to "customer care" with all uploaded files
    attached
```

Note: Instead of setting the templatePathAndFilename option to specify the Fluid template file for the EmailFinisher, the template source can also be set directly via the templateSource option.
identifier: 'Neos.Form:Email'

options:
  templatePathAndFilename: 'resource://AcmeCom.SomePackage/Private/Form/EmailTemplates/Application.html'
  subject: 'New Application'
  recipientAddress: 'application@acme.com'
  senderAddress: '(email)'
  format: html
  attachAllPersistentResources: true

# Confirmation email that is sent to the user with a static file attachment
identifier: 'Neos.Form:Email'

options:
  templatePathAndFilename: 'resource://AcmeCom.SomePackage/Private/Form/EmailTemplates/Confirmation.html'
  subject: 'Your Application'
  recipientAddress: '{email}'
  senderAddress: 'application@acme.com'
  format: html
attachments:
  - resource: 'resource://AcmeCom.SomePackage/Private/Form/EmailTemplates/Attachments/TermsAndConditions.pdf'

Note: attachments can also referenced via formElement paths explicitly, for example: - formElement: 'image-field.resource'

### 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:
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 Neos.Form package already defines a preset with the name default which contains all standard form elements. Look into Neos.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 particularly 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:

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

In the above example, the form element type is Neos.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:

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

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

```php
$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 Neos.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 Neos.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:

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

'Neos.Form:SpecialText':
  superTypes:
    'Neos.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:

```yaml
'Neos.Form:SpecialText':
  superTypes:
    'Neos.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:

```yaml
Neos:
  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:

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

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

### 2.3. Adjusting Form Output
2.3.5 Adjusting a Form Element Template

The templates of the default Form Elements are located in `Neos.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:

```html
<f:layout name="Neos.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>
```

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:

```html
<f:layout name="Neos.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 achieved by overriding the rendering option `templatePathPattern` in the `form element type definition`.

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

```yaml
Neos:
    Form:
        presets:
            myCustom:
                title: 'Custom Elements'
                parentPreset: 'default'
                formElementTypes:
                    'Neos.Form:SingleLineText':
                        renderingOptions:
```

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 Neos.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):

```html
{namespace form=Neos\Form\ViewHelpers}
<f:validation.results 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)}"
    >
      <!-- 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="Neos.Form:Field/Required" />
        </f:if>
      </label>
    </f:if>
  </div>
</f:validation.results>

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

```html
{namespace form=Neos\Form\ViewHelpers}
<f:validation.results 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)}"
    >
      <!-- 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="Neos.Form:Field/Required" />
        </f:if>
      </label>
    </f:if>
  </div>
</f:validation.results>
```

(continues on next page)

2.3. Adjusting Form Output 17
Additionally you need to adjust the default form template located in Neos.Form/Resources/Private/Form/Form.html (remember that a FormDefinition also has a form element type, by default of Neos.Form:Form), which looks as follows by default:

```html
{namespace form=Neos\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="Neos.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:

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

```yaml
Neos:
  Form:
    presets:
      ############### CUSTOM PRESETS ###############
      myCustom:
        title: 'Custom Elements'
        parentPreset: 'default'
        formElementTypes:
```

(continues on next page)
### override template path of Neos.Form:Form ###

```
'Neos.Form:Form':
  renderingOptions:
    templatePathPattern: 'resource://Neos.FormExample/Private/CustomElements/Form.html'
```

### override default layout path ###

```
'Neos.Form:Base':
  renderingOptions:
    layoutPathPattern: 'resource://Neos.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:GenderSelect`, then `{@package}` is replaced by `Your.Package`, and `{@type}` is replaced by `GenderSelect`. Aspartials 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. Let's say you want to create a specialized version of the `Neos.Form:SingleSelectRadiobuttons` that already provides two radio buttons for Female and Male. That's just a matter of a few lines of yaml:

```
Neos:
  Form:
    presets:
      myCustom:
        title: 'Custom Elements'
        parentPreset: 'default'
        formElementTypes:
          # ...
          'Your.Package:GenderSelect':
            superTypes:
              'Neos.Form:SingleSelectRadiobuttons': TRUE
            renderingOptions:
              templatePathPattern: 'resource://Neos.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 Translating Forms

If a form has been set up, all elements will use the labels, placeholders and so forth as configured. To have the form translated depending on the current locale, you need to configure a package to load the translations from and add the translations as XLIFF files.

2.4.1 Configuration

The package to load the translations from is configured in the form preset being used. The simplest way to configure it is this:

```
Neos:
  Form:
    presets:
      default:
        formElementTypes: 'Neos.Form:Base':
          renderingOptions:
            translationPackage: 'AcmeCom.SomePackage'
```

Of course it can be set in a custom preset in the same way.

The translation of validation error messages uses the Neos.Flow package by default, to avoid having to copy the validation errors message catalog to all packages used for form translation. If you want to adjust those error messages as well, copy ValidationErrors.xlf to your package and set the option validationErrorTranslationPackage to your package key.

2.4.2 XLIFF files

The XLIFF files follow the usual rules, the Main catalog is used. The Form package comes with the following catalog (Main.xlf):

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xliff version="1.2" xmlns="urn:oasis:names:tc:xliff:document:1.2">
  <file original="" product-name="Neos.Form" source-language="en" datatype="plaintext">
    <body>
      <trans-unit id="forms.navigation.previousPage" xml:space="preserve">
        <source>Previous page</source>
      </trans-unit>
      <trans-unit id="forms.navigation.nextPage" xml:space="preserve">
        <source>Next page</source>
      </trans-unit>
      <trans-unit id="forms.navigation.submit" xml:space="preserve">
        <source>Submit</source>
      </trans-unit>
    </body>
  </file>
</xliff>
```

It should be copied to make sure the three expected units are available and can then be amended by your own units.
For most reliable translations, the units should be given id properties based on the form configuration. The schema is as follows:

- `forms.navigation.nextPage` In multi-page forms this is used for the navigation.
- `forms.navigation.previousPage` In multi-page forms this is used for the navigation.
- `forms.navigation.submitButton` In forms this is used for the submit button.

Forms and sections can have their labels translated using this, where where `{identifier}` is the identifier of the page or section itself:

- `forms.pages.{identifier}.label` The label used for a form page.
- `forms.sections.{identifier}.label` The label used for a form section.

The actual elements of a form have their id constructed by appending one of the following to `forms.elements.{identifier}.`, where `{identifier}` is the identifier of the form element itself:

- `label` The label for an element.
- `placeholder` The placeholder for an element, if applicable.
- `description` The text of a StaticText element.
- `confirmationLabel` Used in the PasswordWithConfirmation element.
- `passwordDescription` Used in the PasswordWithConfirmation element.

The labels of radio buttons and select field options can be translated using the following schema, where `{identifier}` is the identifier of the form element itself and `value` is the value assigned to the option:

- `forms.elements.{identifier}.options.{value}` Used to translate labels of radio buttons and select field entries.

### 2.4.3 Complete example

This is the example form used elsewhere in this documentation:

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

Assume it is configured like this using YAML:

```yaml
type: 'Neos.Form:Form'
identifier: 'contact'
label: 'Contact form'
renderables:
  - type: 'Neos.Form:Page'
    identifier: 'page-one'
    renderables:
      - type: 'Neos.Form:SingleLineText'
        identifier: name
        label: 'Name'
        validators:
          - identifier: 'Neos.Flow:NotEmpty'
        properties:
          placeholder: 'Please enter your full name'
```

(continues on next page)
Note: You may leave out label and placeholder if you use id-based matching for the translation. Be aware though, that you will get empty labels and placeholders in case the translation fails or is not available.

The following XLIFF would allow to translate the form:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xliff version="1.2" xmlns="urn:oasis:names:tc:xliff:document:1.2">
  <file original="" product-name="Neos.Form" source-language="en" datatype="plaintext">
    <body>
      <trans-unit id="forms.navigation.previousPage" xml:space="preserve">
        <source>Previous page</source>
      </trans-unit>
      <trans-unit id="forms.navigation.nextPage" xml:space="preserve">
        <source>Next page</source>
      </trans-unit>
      <trans-unit id="forms.navigation.submit" xml:space="preserve">
        <source>Submit</source>
      </trans-unit>
      <trans-unit id="forms.pages.page-one" xml:space="preserve">
        <source>Submit</source>
      </trans-unit>
      <trans-unit id="forms.elements.name.label" xml:space="preserve">
        <source>Name</source>
      </trans-unit>
      <trans-unit id="forms.elements.name.placeholder" xml:space="preserve">
        <source>Please enter your full name</source>
      </trans-unit>
      <trans-unit id="forms.elements.email.label" xml:space="preserve">
        <source>Email</source>
      </trans-unit>
      <trans-unit id="forms.elements.email.placeholder" xml:space="preserve">
        <source>Enter a valid email address</source>
      </trans-unit>
      <trans-unit id="forms.elements.message.label" xml:space="preserve">
        <source>Message</source>
      </trans-unit>
    </body>
  </file>
</xliff>
"
Copy it to your target language and add the target-language attribute as well as the needed <target>...</target> entries.

2.5 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?**

2.5.1 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 **PersistentResource**
- A **Captcha** image

A Form Element must implement the `FormElementInterface` interface located in `Neos/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 `Neos/Form/Classes/FormElements/DatePicker.php`:

```php
class DatePicker extends \Neos\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`:

```php
namespace Your\Package\FormElements;

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

    /**
     * Executed after the page containing the current element has been submitted
     *
     * @param \Neos\Form\Core\Runtime\FormRuntime $formRuntime
     * @param mixed $elementValue raw value of the submitted element
     * @return void
     */
    public function onSubmit(\Neos\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 \Neos\Form\Core\Runtime\FormState $formState
     * @return void
     */
    protected function requireIfTriggerIsSet(\Neos\Form\Core\Runtime\FormState $formState) {
        if (!isset($this->properties['triggerPropertyPath'])) {
            return;
        } elseif ($this->properties['triggerPropertyPath']) {
            $triggerValue = $formState->getFormValue($this->properties['triggerPropertyPath']);
            if ($triggerValue === NULL || $triggerValue === '') {
                return;
            }
            $this->addValidator(new \Neos\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:
This makes the new Form Element `Neos.FormExample:ConditionalRequired` available in the preset `somePreset` and you can use it as follows:

```php
$form = new FormDefinition('myForm', $formDefaults);
$page1 = $form->createPage('page1');
$newsletter = $page1->createElement('newsletter', 'Neos.Form:Checkbox');
$newsletter->setLabel('Subscribe for Newsletter');
$email = $page1->createElement('email', 'Neos.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.

### 2.5.2 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 a 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 an unordered list. A Form Element Renderer must implement the `RendererInterface` interface located in `Neos.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 \Neos\Form\Core\Renderer\AbstractElementRenderer {

    /**
     * @param \Neos\Form\Core\Model\Renderable\RootRenderableInterface $renderable
     * @return string
     */
    public function __renderRenderable(\Neos\Form\Core\Model\Renderable\RootRenderableInterface $renderable) {
    }
}
```

(continues on next page)
$renderable->beforeRendering($this->formRuntime);
$items = array();
if ($renderable instanceof \Neos\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';
$content = $this->formRuntime->invokeRenderCallbacks($content, $renderable);
return $content;
}

**Note:** Don’t forget to invoke `RootRenderableInterface::beforeRendering()` and `FormRuntime::invokeRenderCallbacks()` as shown above.

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

Make sure to have a look at the ‘FusionRenderer Package[:https://packagist.org/packages/neos/form-fusionrenderer]’ that provides Fusion based rendering for arbitrary Form Elements!
The initial implementation has been generously sponsored by AKOM360 - Multi Channel Marketing. Further work has been supported by internezzo ag - agentur für online-kommunikation and Format D GmbH.

It has been implemented by:

- Sebastian Kurfürst, sandstormmedia
- Bastian Waidelich, wwwision