reef Documentation

Release 0.1.0

Marco

Nov 29, 2018

Contents

CHAPTER 1

Introduction

This is the documentation for Reef, the Responsive Embeddable Extensible Form generator for PHP.

Reef is an open source, framework agnostic form builder for PHP, providing functionality for building and showing forms and receiving form submissions from users. Building forms can be done using either a drag and drop user interface, a chaining PHP interface, or using YAML files or form definition arrays.

Reef requires PHP 7.2 or higher, and depends on the jQuery javascript library.

CHAPTER 2

User's guide

Reef provides a user interface for both managing and filling out forms. This user's guide is intended to as documentation on how to manage forms using the form builder UI. For technical documentation on the Reef API or on extending Reef, please refer to the integration guide or contribution guide, respectively.

2.1 Terminology

Before we start, let's agree on some terminology first:

- A *component* is a type of input, e.g. a text input, dropdown or checkbox
- A *field* is a component inserted in a form. You may for example add a text input component to ask for a user's name, and use another text input component to ask for his favourite type of food
- A *form* is a collection of fields, e.g. you may compose a form asking for a user's name, age and gender using a text field, number field and radio (bullet selection) field
- A *submission* is form data submitted by the user

Additionally, the following terminology is used in configurations:

- A *title* represents text that is presented to the end user, often the user viewing a form. For installations using multiple locales, titles can always be translated in these multiple locales.
- A *name* on the other hand represents the technical name of some object, for instance a field. A name may only consist of lowercase alphanumeric characters and underscores, that is `a-z`, `0-9` and `_`, under the condition that the name starts with a letter (`a-z`) and an underscore may not be followed by another underscore. These technical names are used internally to identify the object you are configuring: the storage in the database will be named after it, and the name can be used in conditions. In the builder, all objects always get assigned a valid default name, so you are not required to name objects yourself, but you can change the name if necessary.

2.2 Outline

The process of creating a new form typically consists of:

1. Adding fields to your form
2. Configuring each field
3. Optionally, configuring the form itself
4. Saving the form

These actions have their own tabs in the interface, available in the top left.

1. In the leftmost tab, all available components are displayed, which you can drag into your form, or on which you can click to append them to your form
2. When selecting a field you have added, the middle tab provides the possibility to configure the field in question
3. The rightmost tab may (or may not, depending on the integration) provide the possibility for some form configuration, after which you can also save the form here

2.3 Configuring fields

The fields are divided into a number of groups:

- Text input fields, in which users can type data
- Choice fields, where users can choose from a predefined set of choices
- Static fields, e.g. headings and paragraphs of text, that do not present any input possibility
- ‘Other’ fields, including any fields that do not fit in any of the above categories

Each component has its own specific set of configuration possibilities. A heading is quite simple: you can set the size and the text. A number field can be configured quite extensively, providing possibilities to set e.g. a minimum, maximum and default value, and the possibility to set custom error feedback texts.

The specific configuration settings of all components can be found in the components reference. Here, we will focus on two general settings applying to almost all components.

2.3.1 Visible setting

You may wish to optionally show or hide fields, depending on data entered previously. For example, you may want to ask for one’s telephone number after they have checked a checkbox asking whether they want to get in touch. In this case, the telephone number field would be shown if the checkbox is ticked, or hidden otherwise. You can achieve this by setting the ‘visible’ setting to ‘condition’, and configuring this property in the correct way. More on how to do this can be found on the condition component page.

2.3.2 Required setting

Analogous to the visible setting, you may want to optionally require some fields. For instance, in the above example, instead of hiding the telephone field if the box is not ticked, you may wish to show the field but not require it, and only require it if the user indicates he wants to get in touch. In this case you can use a condition for the required property. Of course you can also set it to ‘Yes’ or ‘No’ to either always or never require it. Again, more information on conditions can be found on the condition component page.

CHAPTER 3

Components reference

Click on a reference below to view its guide.

3.1 Text

Text components allow the user to enter text. The entered value may be anything within the set boundaries.

3.1.1 Text line

The text line component allows the user to fill in a freeform text value consisting of one single line.

Table 1: Declaration parameters

Name	Type	Default value	Description
Default value `default`	Text	‘’ (empty)	The default value
Max. input length `max_length`	Number	1000	The maximum input length, in number of characters

Table 2: Language items

Name	Description
Title `title`	The field title/label
Placeholder `placeholder`	The default gray background text

3.1.2 Textarea

The textarea component allows the user to fill in a (possibly large) freeform text value consisting of possibly multiple lines.

Table 3: Declaration parameters

Name	Type	Default value	Description
Default value `default`	Text	‘’ (empty)	The default value
Max. input length `max_length`	Number	15000	The maximum input length, in number of characters

Table 4: Language items

Name	Description
Title `title`	The field title/label
Placeholder `placeholder`	The default gray background text

3.1.3 Number

The number component allows the user to fill in a number. It adheres to the min/max/step construction that the HTML element `<input type="number">` uses.

Table 5: Declaration parameters

Name	Type	Default value	Description
Default value `default`	Number	‘’ (empty)	The default value
Minimum `min`	Number	‘’ for none	The minimum value
Maximum `max`	Number	‘’ for none	The maximum value
Step `step`	Number	‘’, implies ‘1’	The step between values. Valid field values are defined by the sequence `min`, `min + step`, `min + 2*step`, et cetera not exceeding `max`.

Table 6: Language items

Name	Description
Title `title`	The field title/label
Validation error `error_not_a_number`	Feedback text when the user has entered non-numeric input
Minimum error `error_number_min`	Feedback text when the user has entered a value that is too low
Maximum error `error_number_max`	Feedback text when the user has entered a value that is too high
Min/max error `error_number_min_max`	Feedback text when the user has entered a value that is not within the set bounds

3.2 Choice

Choice components allow the user to choose between a predefined set of values, i.e. no manual input is requested.

3.2.1 Checkbox

The checkbox component presents the user with a single checkbox.

Table 7: Declaration parameters

Name	Type	Default value	Description
Checked by default `default`	Checkbox		This option indicates the default value of the checkbox, either checked or not checked

Table 8: Language items

Name	Description
Title `title`	The field title/label
Box label `box_label`	A label near the checkbox

3.2.2 Checklist

The checklist component presents the user with a list of checkboxes, each of which can be independently checked.

Table 9: Declaration parameters

Name	Type	Default value	Description
Options `options`	Option list		Each option added here represents a single checkbox that can be checked. The checkbox indicates the default value of that option, either checked or not checked. At least two options should be added. See Option list for more details.

Table 10: Language items

Name	Description
Title `title`	The field title/label

3.2.3 Choice list (radio buttons)

The choice list component allows the user to choose one of multiple options. In contrast to the Dropdown (select) component, all options are always presented, with bullet points before the options that can be clicked to choose that option.

Table 11: Declaration parameters

Name	Type	Default value	Description
Options `options`	Option list		Each option added here represents a radio bullet to choose from. The checkbox indicates the default value (at most one). At least two options should be added. See Option list for more details.

Table 12: Language items

Name	Description
Title `title`	The field title/label

3.2.4 Dropdown (select list)

The dropdown component allows the user to choose one of multiple options. In contrast to the Choice list (radio buttons) component, only one options is presented, with the possibility to extend the field to show all options to choose from.

Table 13: Declaration parameters

Name	Type	Default value	Description
Options `options`	Option list		Each option added here represents an option to choose from. The checkbox indicates the default value (at most one). At least two options should be added. See Option list for more details.

Table 14: Language items

Name	Description
Title `title`	The field title/label

3.3 Static

Static components allow the creator of a form to add some text without an input field, to add some contextual information to the form.

3.3.1 Heading

The heading component allows a form creator to add a title or heading to a form.

Table 15: Declaration parameters

Name	Type	Default value	Description
Level `level`	Number	1	The heading level, may be a number from 1 to 6. A level 1 heading corresponds to the HTML `<h1>` tag. Generally the level 1 heading is the largest heading, while the a level 6 heading is the smallest.

Table 16: Language items

Name	Description
Title `title`	The title text

3.3.2 Paragraph

The paragraph component allows a form creator to add a paragraph of plain text.

Table 17: Declaration parameters

Name	Type	Default value	Description
			This component has no declaration parameters

Table 18: Language items

Name	Description
Content `content`	The paragraph text

3.4 Internal

Internal components are only meant to be used in the builder UI, hence they can not be used in forms. However, due to their more elaborate configuration options, the following pages are still useful to end users.

3.4.1 Option list

The option list component is an internal component, and hence cannot be used in the builder. It is used as configuration component in the builder however.

Option list in the builder

For components involving multiple predefined options, an option list is presented allowing you to fill in the options the user is allowed to choose from. For each option, there are three settings to configure:

- Name: The name of the options. This is a technical name that can be used for e.g. conditions and overviews, and hence should only consist of lowercase letters, numbers and underscores. If the names are not important, the default names `option_1`, `option_2`, et cetera are perfectly fine to use. For more information on names, please refer to the terminology.
- Title: The title of the option. This is the text that is presented to the user. When using multiple locales, you can enter a different text for each locale.
- Checkbox: The checkbox often indicates whether the option is a default option, i.e. whether it is choosen by default or not.

Both the name and checkbox may be absent from the option list, depending on the particular needs of the component.

Technical details

The option list allows the user to provide multiple options in multiple locales. Besides a name and localized title, a checkbox can be utilized to provide a toggle for default options.

Table 19: Declaration parameters

Name	Type	Default value	Description
Min. number of options `min_num_options`	Number	2	The minimum amount of options the user should provide. Often, this should be 2
Max. number of options `max_num_options`	Number	None	The maximum amount of options the user may provide. By default, no explicit limit is used
Min. number of checked defaults `max_checked_defaults`	Number	None	The maximum amount of options the user may mark as default
Default number of options `default_num_options`	Number	3	The default number of options that is added when in the builder a new field is added that uses an option list in its configuration
Ask field names `names`	Checkbox	true	Whether names should be asked. Toggles the display of the name input fields
Default options `default`	Option list	`default_num_options` default options	A list of default options

Table 20: Language items

Name	Description
Title `title`	The field title/label

3.4.2 Condition

The condition component is an internal component, and hence cannot be used in the builder. It is used as configuration component in the builder however.

Conditions in the builder

Conditions provide you the possibility to perform certain actions based on values that the user filled in previously. If you have two fields and only want to show field 2 if field 1 is filled in, this can be achieved using conditions. Reef itself provides functionality to show or hide fields dependent on certain conditions, and to mark fields as required based on certain conditions. These two settings are called 'visible' and 'required' respectively, for these settings, conditions should be read as follows:

- visible: *This field should be visible only if the following is true*
- required: *This field should be required only if the following is true*

Note: Invisible fields cannot be required. However, you do not need to check this yourself: Reef checks this for you

In the builder, a condition can be set by either of four methods:

- 'Yes' (or `true`), indicating a positive result (e.g. the field should be visible)
- 'No' (or `false`), indicating a negative result (e.g. the field should not be visible)
- 'Condition', this option provides you with an interface to build a custom condition. More information below.
- 'Manual condition', this option is only provided for those who have enough technical knowledge to type conditions themselves, and wish to create more complicated conditions

The condition interface

Any condition returns either yes/true or no/false (the first two of the four options above). Which of these two, depends on the input of the user. In the interface, you can select three fields. Always select them from left to right!

- The first dropdown allows you to select the field which has to be used in the condition. By default, Reef assigns names like `field_123456789abcdef` to fields, but you may change these names in order to ease the condition building
- Dependent on the choice in the first dropdown, the second dropdown present different operators you can use. These could for instance be checks for equality, checking a number is higher/lower than some number, et cetera
- Often, an operator in the second dropdown requires some operand, which can be chosen in the third and final input. The exact behaviour of this last input varies per field and operator. Refer to their documentation for more information.

The above holds for a single condition. However, you may wish to combine multiple conditions. E.g., ask *A* if *B* or *C*. You can add multiple conditions using an and/or construction: `B and C` is true if both `B` and `C` are true, `B or C` is true if at least one of `B` and `C` is true.

Technical details

The option list allows the user to provide multiple options in multiple locales. Besides a name and localized title, a checkbox can be utilized to provide a toggle for default options.

Table 21: Declaration parameters

Name	Type	Default value	Description
Default `default`	Condition	None	A default condition

Table 22: Language items

Name	Description
Title `title`	The field title/label

CHAPTER 4

Integration guide

Before continuing, make sure you've read up on the terminology first.

Reef requires a reasonable amount of set-up and integration before it operates within your website. Depending on your requirements, you will have to write code in these languages:

- In PHP, code for preparing the builder and forms, receiving submissions and passing them to Reef, and providing feedback to the browser where required. In addition, an entry point should be created for internal requests. Most likely, you will also have to check authorization here: determining who may create forms and who may fill them in.
- In HTML, some code in which the resulting Reef HTML will be injected. Probably a wrapper `<div>` or the like
- In JavaScript, creating an instance of the `Reef` or `ReefBuilder` class, in order for the interactive elements to work

The integration is divided into four parts: the general Reef setup, implementing internal requests, builder integration and form integration.

4.1 Initializing Reef

Reef is initialized in two steps: first you create a `ReefSetup` object, which is used to build to configuration. Then, by passing this setup to a new `Reef` instance, the setup is frozen and checked. One of the main checks that is performed is compatibility checks, checking that the components, extensions and layouts are compatible with each other. If not, an exception is thrown.

An example of creating a Reef setup:

```
$Setup = new \Reef\ReefSetup(
        new \Reef\Storage\PDO_MySQL_StorageFactory($YourPDOObject),
        new \Reef\Layout\bootstrap4\bootstrap4(),
        new \Reef\Session\PhpSession()
);
```

The first argument defines the specific storage class that is to be used, you can choose from `PDO_MySQL_StorageFactory`, `PDO_SQLite_StorageFactory` and `NoStorageFactory`, of which the latter can only be used if you do not want to store forms and submissions.

Note: When using one of the `PDO` factories, Reef requires the `PDO` attribute `PDO::ATTR_ERRMODE` to be set to `PDO::ERRMODE_EXCEPTION`. Additionally, if you use MySQL, the connection charset should be set to `utf8mb4`.

The second argument defines the layout you want to use. Each layout has its own set of configurable options that you can set at creation. Note you can also pass an array of layout objects.

The third argument specifies the session implementation to use. For most basic components, a session object is not required, in which case you can use `NoSession`, but for example the `reef:upload` component requires a session object.

After creating the `ReefSetup` object, one can start adding additional components and extensions. The ReefSetup is initialized by default with some basic native components that need not be added manually:

- `reef:text_line`
- `reef:textarea`
- `reef:checkbox`
- `reef:radio`
- `reef:check_list`
- `reef:text_number`
- `reef:heading`
- `reef:paragraph`
- `reef:hidden`
- `reef:option_list`
- `reef:select`
- `reef:condition`
- `reef:submit`

Other components, from `reef-extra`, a third-party developer, or from yourself, can be added to this list using `ReefSetup::addComponent()`. For example, to add the upload component, one would do:

```
$Setup->addComponent(new \Reef\Components\Upload\UploadComponent);
```

Similarly, one can add extensions using `ReefSetup::addExtension()`. By default, no extensions are loaded into Reef. To load the `reef-extra::required-stars` extension, you could for instance do:

```
$Setup->addExtension(new \ReefExtra\RequiredStars\RequiredStars);
```

Once the setup is finished, one can initialize Reef itself:

```
$Reef = new \Reef\Reef(
        $Setup,
        [
                'cache_dir' => './cache/',
                'locales' => ['en_US'],
                'internal_request_url' => './reefrequest.php?hash=[[request_hash]]',
```

(continues on next page)

(continued from previous page)

```
        ]
);
```

The second parameter here is an array of configuration settings. The cache dir defines where Reef may put cache files. It is not optional, as currently the assets functionality relies on caching being present. The locales setting defines which locales to use. You can pass any number of locales here; passing multiple locales allows you to switch between locales and creating forms in these multiple locales. The `internal_request_url` setting defines the entry point of internal requests. Internal requests are used for assets like JS and CSS files and images, while it also provides the possibility for callacks to PHP for e.g. uploading files. This setting is also non-optional. More options are available, these can all be found TODO.

This concludes this section on creating the Reef object. Next we will use it to integrate the builder into your website.

4.2 Implementing internal requests

Before we start integrating the builder or form, we should start with one prerequisite: the internal requests entry point.

With the internal request URL configured as above, the request hash will be available to the internal request controller in `$_GET['hash']`. This hash should be passed to Reef in the following way, using the `$Reef` object as created above:

```
$Reef->internalRequest($_GET['hash']??null, [
        'form_check' => function($Form) {
                // Check that user is allowed to view $Form
        },
        'submission_check' => function($Submission) {
                // Check that user is allowed to view $Submission
        },
]);
```

The result of the internal request can be anything: JS code, CSS code, an image or, more generally, any (binary) file. Make sure your code can handle this.

At this stage, you probably do not yet know how to check whether a user is allowed to view something. For the time being, we can leave them empty. When finished, you can implement them. If a user is not allowed to view view the given form or submission, you should make sure to stop execution by either throwing an exception or exiting the script.

Important: The internal requests URL can be called with both `GET` and `POST` methods, so be sure the code answers to both of these.

4.3 Integrating the builder

Note: If you do not want to use the builder interface, you can skip this section

To integrate the builder, we expect a `$Reef` object is available, as generated in the previous section. Getting a builder object is not hard. Actually, it is as easy as:

```
$Builder = $Reef->getBuilder();
```

Next you may want to change some settings:

```
$Builder->setSettings([
        'submit_action' => 'path/to/builder/submit',
        'definition_form_creator' => function($Creator) {
                // ...
        },
        'components' => [
                // ...
        ],
]);
```

The available settings are:

- The `submit_action` is the entry point to which the builder data (the form configuration) may be submitted (POSTed).
- The `definition_form_creator` is a callback function with which you can modify the form configuration form (read: the form containing the form configuration). By default, this form contains a `storage_name` field which you may or may not want (you may delete it, but then you should provide your own value programmatically when submitting, more on that further below), and you may add your own configuration values.
- The `components` array defines which components you want your users to be able to use in the builder. It should (of course) be a subset of the components added to your Reef setup. You can also add and remove components using `Builder::addComponents()` and `Builder::removeComponents()`.

Next, we want to display the builder. Before creating the builder, we have to create the form we want to edit. An existing stored form can be fetched using `$Form = $Reef->getForm($i_formId);`, a new form can be created using `$Form = $Reef->newTempStorableForm(['storage_name' => 'some_unique_table_name']);`. Note that the storage name should match a certain format, please refer to the information on names on in the terminology.

Having obtained the form, we can obtain the HTML as follows:

```
$s_html = $Builder->generateBuilderHtml($Form, [
        'definition_submission' => [
                // ...
        ],
]);
```

The `definition_submission` setting is optional, and can provide values for fields you added to the form configuration form you added earlier to the `definition_form_creator`.

The JS code can be obtained using:

```
$s_js = $Reef->getReefAssets()->getJSHTML('builder', [
        'exclude' => [
                'jquery',
                'popper',
                'bootstrap4',
        ],
]);
```

Here, `exclude` defines which libraries your website has already loaded and hence should not be included again. The CSS code can be obtained similarly using `getCSSHTML()`.

Include the JS and CSS code in your `<head>`, and the `$s_html` in somewhere in your `<body>`, like:

```
<div class="builderWrapper">
        <?php echo($s_html); ?>
</div>
```

At this stage you should be able to see the builder interface on your builder page. However, it will not be interactive yet, as we have not done anything with JavaScript yet. To attach javascript, you use something like:

```
var builder;
$(function() {
        builder = new ReefBuilder('.builderWrapper', {
                submit_before : function(ajaxParams) {
                        ajaxParams.data.some_custom_variable = "some_custom_value";
                }
        });
});
```

The first argument defines the element where the builder HTML resides in. In this case we use the `div.builderWrapper` as used above. The second argument may contain some settings. The `submit_before` setting may be used to alter the AJAX request to the server before performing it; it can be used to add additional parameters. Of course, you can also ommit the second argument.

Having added the javascript, reloading the builder interface you should now have achieved an interactive builder. The last step remaining is implementing the submit entry point.

Once a user saves a form, a request is POSTed to the `submit_action` URL provided in the `$Builder` settings, with the form data in the `$_POST['builder_data']` variable. In most cases, you will want to do something along these lines:

```
$Builder->processBuilderData_write($Form, $_POST['builder_data'], function(&$a_return,
↪ $DefinitionSubmission) use(&$Form) {
        if($a_return['result']) {
                // Here you may want to perform some database actions of your own,␣
↪probably saving $Form->getFormId() somewhere

                $a_return['redirect'] = 'some/url/to/redirect/to/after/success';
        }
});
```

Important: Note that in order for the updater to be able to correctly migrate the old form state to the new form state, you will need to use the same `$Builder` and `$Form` objects in both `generateBuilderHtml()` and `processBuilderData_write()`.

The `processBuilderData_write()` method processes the builder data, which might be a two-step process:

- In the first request, the JS builder indicates no data loss is permitted. If the updater recognizes that dataloss *might* or *will* occur, the form is not updated and information about the data loss is returned instead, asking the user whether he/she really wants to do this.
- If the user accepts data loss, another request is done, with the indication that data loss is permitted.

When the builder may proceed, the form is migrated from the old state to the new state. After the form has been migrated successfully to the new state, the callback you may pass to the third argument of `processBuilderData_write()` is performed. Note that whenever you want to use `$Form` in the callback, you have to pass it by reference, as the form in the new state will be a new `Form` instance. The first parameter of the callback (`$a_return`) contains the data returned to the JS builder. If `$a_return['result']` is truthy, the form has updated successfully. You may provide an URL for `redirect` to redirect the user to on success. The second parameter of the callback (`$DefinitionSubmission`) contains the submission of the form configuration form. If you used `definition_form_creator` earlier on to augment this form, here is your chance to read out the values the user submitted.

This concludes the integration of the builder into your website. While you cannot yet view and fill out the forms themselves, you should now be able to create and edit forms. You can also implement a delete function using `$Form->delete();`.

Note: If there are no submissions to a form, the builder will never complain about data loss as there is no data to be lost. Keep this in mind if you are testing the builder integration at this stage.

4.4 Integrating the form viewer

To integrate the form viewer, we expect a `$Form` object is available, created in the builder and loaded through `$Reef->getForm()` or `$Reef->getFormByUUID()`.

To view the form, we need a `Submission` object. For a new submission, this can be created using:

```
$Submission = $Form->newSubmission();
$Submission->emptySubmission();
```

To edit an existing submission stored by Reef, you can use:

```
$Submission = $Form->getSubmission($i_yourSubmissionId);
```

Now you can obtain the form HTML using:

```
$s_html = $Form->generateFormHtml($Submission, ['main_var' => 'form_data']);
```

You may also pass `null` for `$Submission`, in this case automatically a new submission will be used. The `main_var` defines which `$_POST` entry should be used for the form data.

In a similar way as with the builder, you can fetch the JS code using:

```
$Form->getFormAssets()->getJSHTML('form', [
	'exclude' => [
		'jquery',
		'popper',
		'bootstrap4',
	],
]);
```

The CSS code can be obtained using `getCSSHTML()`.

The HTML and JavaScript required to create an interactively checked form will look like this:

```
<form action="path/to/submission/submit" method="post" onsubmit="return reef.
↪validate();">
	<div class="form-wrapper">
		<?php echo($s_html); ?>
	</div>
</form>

<script>
var reef;
```

(continues on next page)

(continued from previous page)

```
$(function() {
        reef = new Reef($('.form-wrapper'));
});
</script>
```

In the controller behind the action URL, you can save the submission using:

```
if($Submission->processUserInput($_POST['form_data']??[])) {
        // Successfully saved. Here you can redirect or take some other action
}
```

Here, `processUserInput()` is a utility function performing `fromUserInput()`, `validate()` and `save()` inside a SQL transaction.

Tip: If you use the same controller for the view and submit actions, using e.g. `if($_SERVER['REQUEST_METHOD'] == 'POST')` to distinguish them, you should put this call to `processUserInput()` before `generateFormHtml()`. In this way, `generateFormHtml()` will include any validation errors if these occurred and JavaScript did (or could) not catch them!

You may probably want to also include functionality to delete a submission, this can be done using `$Submission->delete();`. With this last note, this concludes the integration of the form viewer.

Attention: Don't forget to implement the form and submission permission checks in your internal request implementation!

4.5 Submitting the form using AJAX

You may wish to submit your form submissions through AJAX rather than through a standard POST request. In this case, the following changes should be applied compared to the instructions in the previous section.

You should give the `<form>` element an id (e.g. `<form id="submission_form" ...>`), and then use:

```
reef = new Reef($('.form-wrapper'), {
        submit_url : 'path/to/submission/submit',
        submit_form : $('#submission_form')
});
```

You can of course also implement your own submit call, using `reef.validate()` for validation.

In your submit PHP, you can then do something along the lines of:

```
$Submission->fromUserInput($_POST['form_data']??[]);
if($Submission->validate()) {
        $Submission->save();
}
else {
        $a_return = [
                'errors' => $Submission->getErrors(),
        ];

        echo(json_encode($a_return));
```

(continues on next page)

(continued from previous page)

```
        die();
}
```

When errors occur, the automatic reef JS submitter will recognize the `errors` entry and add the errors to the respective fields. When implementing your own submit call, you can use `reef.addErrors(response.errors);` to achieve this.

4.6 Presenting hardcoded forms

You may want to present forms that are not saved by Reef, and store the submissions yourself. In this case, the following changes should be applied compared to the instructions in the integration section.

First, you will not be loading the `$Form` from Reef but creating it yourself. You could do this using the Creator object, like this:

```
$Form = $Reef->newTempForm();
$Form->newCreator()
        ->addField('reef:text_line')
                ->setName('name')
                ->setLocale(['title' => 'Your name'])
        ->apply();
```

Next, we need to replace the `processUserInput()` call with the following:

```
$Submission->fromUserInput($_POST['form_data']??[]);
if($Submission->validate()) {
        $a_yourStructuredData = $Submission->toStructured();
        // Save the data
}
```

Now, `$a_yourStructuredData` contains a structured representation of the submission data. You may for example `json_encode()` it and save it in a file or database.

The last line that needs to be replaced is the `getSubmission()` line, to import the data from the structured data representation:

```
$Submission = $Form->newSubmission();
$Submission->fromStructured($a_yourStructuredData);
```

Important: Note that not all components may be compatible with this way of using Reef. Forms that are not saved within Reef for example lack the ability to (reliably) use the filesystem functionality, and hence cannot use the `reef:upload` component.

CHAPTER 5

Contribution guide

There are multiple ways you can customize, extend or contribute to Reef. Most of these can be done by using mechanisms provided by Reef, such as for layouts or components. A guide for each of these types is available below:

5.1 Creating a component

The best way to create a component is probably by looking how other components are built up, and copying from them. In addition, this guide will help you to better understand what is going on.

5.1.1 Terminology

Please remember the following terminology, in addition to the basic terminology:

- A component is a complete set of PHP, HTML, JS & CSS code that defines the behaviour of a specific type of user input, being for instance a text field, checkbox or heading.
- A field is an instance of a component added to a form, complete with filled-in configuration values such as field name (=technical name!), field title (as shown to users), and things like placeholders, default values, required status, visibility conditions et cetera.
- A field value is an object entirely devoted to reading, parsing, validating and saving the value a user submitted to a field.

Additionally:

- The structured storage methods `toStructured()` and `fromStructured()` define how the value of the field should be registered in PHP. The structured format is a mixed format: it may be e.g. an int, string, (multi-dimensional) array, et cetera.
- The flat storage methods `toFlat()` and `fromFlat()` define how the value of the field should be stored in the database. It has a more restricted format: `toFlat()` should return a one-dimensional array of data, where each entry in the array corresponds to one column in the database table created for the form. A component may use any number of entries in the flat representation; either 0, 1 or more, as desired.

5.1.2 Directory structure

A component has a specific directory structure:

```
src
├── js (or css)
│   ├── {view_name}.js (e.g. form.js)
│   └── {layout_name}-{view_name}.js (e.g. bootstrap4-form.js)
├── locale
│   └── {locale_name}.yml (e.g. en_US.yml)
├── view
│   └── {layout_name} (e.g. bootstrap4)
│       ├── form.mustache
│       └── submission.mustache
├── config.yml
├── {view_name}.css (e.g. form.css)
├── {view_name}.js (e.g. form.js)
├── {component_name}Component.php
├── {component_name}Field.php
└── {component_name}Value.php
```

Here, `view_name` can be either `form`, `submission`, `builder` or `all`. When using a specific view, the js/css file is only included when required (i.e., when that view is loaded). When using all, the js/css file is always included whenever the component is used.

You can put a general JS or CSS file in the root `src` dir, or put it in the js/css subdirectory. Additionally, you can place JS and CSS files in the js/css subdirectory with a filename `{layout_name}-{view_name}.js/css` to specify code for a specific layout.

In the locale folder, you can add translations in multiple languages. Note that the keys used in this file also need to be registered in the `config.yml` file.

In the view folder, all HTML code is situated, using Mustache as templating engine, in PHP as well as in Javascript. For each supported layout, you should add two view files, `form.mustache` and `submission.mustache`.

The file `config.yml` defines the general properties of your component.

The three PHP files contain the PHP code of your extension. The component file contains general functionality, the field file contains functionality for fields (component instances in the form), and the value file contains functionality for parsing, validating and processing submitted values.

5.1.3 Specifics

Components come with a relatively large number of options you can use, hence we split this guide into four sections:

Configuration file

The configuration file defines most of the settings of your component. If contains the name, used locale entries, declaration form and condition operators.

In general, your `config.yml` will look like this:

```
vendor: vendor_name
name: component_name
category: text
documentation: https://link-to-your-docs
assets:
```

(continues on next page)

(continued from previous page)

```
        component_image: img/component-image.svg
basicLocale:
        title: rf_field_title
advancedLocale:
        error_too_long: error_too_long_title
internalLocale:
        error_empty:
basicDefinition:
        fields: []
advancedDefinition:
        fields: []
props: []
builder_operators:
        empty: is empty
        nempty: is not empty
```

Required values

The vendor name and component name should be names according to the conventional definition within Reef. The category can be one of the builder categories, e.g. text, choice, static or other. These three values (vendor, name, category) are always required. The `component_image` defines the path to the image file of this component, used in the builder. When the component is to be used in the builder, this value is also required.

Optional values

Locale and definition

The basic/advanced variants define whether to place the specific item in the basic tab or advanced tab in the builder. The `basicLocale` and `advancedLocale` contain `key:  value` pairs where:

- `value` is the locale name of the title of the text input in the builder, i.e. the text presented to the form maker. These are typically also present in a locale .yml file, either the component locale or the general reef locale.
- `key` is the locale name of the locale value, i.e. the text filled in by the form maker and presented to the form submitter

Additionally, `internalLocale` is an array of locale keys in the form `key:  (empty)`, where each key is a value present in the the component or reef locale files.

The definition arrays contain a (possibly empty) list of field declarations. These declarations define the custom configuration values for the component, e.g. a default value, placeholder, max input length, et cetera. The `name`, `required` and `visible` fields are automatically added by Reef, depending on the PHP implementations.

You may also have a `props` entry, which may hold a value that should not be presented in the builder. It is for example used in the `text_line` component to define a regexp value. Each entry in props is a hash with both a name and a default value.

Operators and props

There is also the `builder_operators` entry, defining which condition operators are supported. It holds an array of `key:  value` pairs where:

- `key` is the name of the operator, which is used to translate the operator name in the locale file (where `operator_{operator name}` is used)
- `value` is the operator as used in conditions. This should be an english name, just as in most programming languages

Creating the component & component class

Components define specific types of input: a text field, checkbox, dropdown, et cetera. Reef provides some basic components, but you can write components yourself to introduce custom input types.

Component member functions

The main component class should extend the abstract class `\Reef\Components\Component`. Each component should have a unique name set by the `const COMPONENT_NAME` constant, and the directory that the component code resides in should be given by the `getDir()` method. The `getDir()` implementation should be something along the lines of:

```
public static function getDir() : string {
        return __DIR__.'/';
}
```

In addition, components have the possibility to use a number of different methods:

`__construct()`

The `__construct()` function is not used by the `Component` base class, and hence can without limitations be used by the component itself for initialization purposes.

`init()`

The `init()` function can be used to initialize the component. It is called as soon as the component is added to the Reef setup class. The difference with the `__construct()` method is that `init()` receives the Reef setup object as first parameter, allowing you to use the entire Reef setup here.

`getCSS()` and `getJS()`

You can use the `getCSS()` and `getJS()` functions to add any (remote) CSS/JS assets the component requires. Each of these methods should return an array of arrays, where each contained array is an associative array defining an asset using the following entries:

```
type => 'local' (in component directory) or 'remote' (over http(s))
path => path or url to the file
view => for which view(s) to load, one of 'form', 'submission', 'builder' or 'all'.␣
↪Optional, defaults to 'all'
name => canonical name (required for remote files). Allows Reef to recognize when two␣
↪plugins want to include the same library, and to make sure it does not include them␣
↪twice.
integrity => Optionally, an integrity value for remote files
```

`checkSetup()`

The `checkSetup()` function can be used to check the component configuration. It is called by the `checkSetup()` function in the Reef setup class, which is called as soon as a `Reef` object is created. In this method you can check whether any dependencies are present that should be present, and whether there are any incompatibilities.

`validateDeclaration()`

The `validateDeclaration()` function is performed whenever a field declaration (possibly belonging to a form definition) is validated. You can override it to perform any custom checks. For example, the `reef:text_number` components checks whether the range `max` is not lower than the range `min`. By default this function returns `true` indicating a valid declaration, but this may change in the future, so be sure to call this method and use its result in your child class implementation. If you require the form context of the field in your check, you may better use `Field::validateDeclaration()`.

`requiredComponents()`

The `requiredComponents()` function returns an array of component names that are required to be present for the component to work. The main component class determines this list automatically based on the `basicDefinition` and `advancedDefinition` of the component configuration. However, you may need to augment this array a bit in specific cases.

`nativelySupportedLayouts()`

The `nativelySupportedLayouts()` function can be used to override the layouts supported by the component, if it uses templating. If you do not, you may override this function to return `null`.

`addLayout()`

Probably you cannot support each layout that's around. Hence, if a layout wishes to build in support for this component, the `addLayout()` function can be used by the layout to add a layout directory for that layout for this component.

`getConfiguration()`

This function returns the configuration array of the component. The result is cached for performance. The configuration is determined based on the component and its parent components; however do note that component inheritance is not fully developed yet.

`internalRequest()`

Internal requests routed to this component arrive here. Internal requests serve as communication method between the browser and PHP, to aid interactivity. Depending on the request hash (the first argument), you may return anything to the browser, like an image, JSON object, et cetera. The request paths of components are formed like `component:{component_vendor_name}:{component_component_name}:{custom_request_hash}`, where `custom_request_hash` is the hash passed to `Component::internalRequest()`.

The field class

Field are instances of components, in the sense that a field is a component assigned to a form. The form and component objects can be fetched using `getForm()` and `getComponent()`.

Field member functions

The field class should extend the abstract class `\Reef\Components\Field`.

Fields have the possibility to use a number of different methods:

`__construct()`

The `__construct()` function is not used by the `Field` base class, and hence can without limitations be used by the component itself for initialization purposes.

`getDeclaration`

The function `getDeclaration` returns the declaration of the field, as an array.

`checkSetup()`

The `checkSetup()` function can be used to check the component configuration. It is called by the `checkSetup()` function in the Reef setup class, which is called as soon as a `Reef` object is created. In this method you can check whether any dependencies are present that should be present, and whether there are any incompatibilities.

`validateDeclaration()`

The `validateDeclaration()` function is performed whenever a field declaration (possibly belonging to a form definition) is validated. You can override it to perform any custom checks, but be sure you also include the result of this method in your child class implementation. This method is equivalent to `Component::validateDeclaration`, with the added possibility of using the form context of the field.

`getFlatStructure()`

The `getFlatStructure()` function should return an array of columns to be used in the database table for this field. We distinguish three cases:

1. If the field requires multiple columns in the database table, the result of this array should be an associative array with the column name as key and the column specification as value.
2. Otherwise, if the field requires only one column in the table, you may use either an associative array as above, or a numeric array with one entry having key 0 (zero), with as value again a column specification.
3. If the field requires no storage, you may return an empty array `[]`.

Note that this function is abstract in `Field`, hence it should always be implemented. A column specification is an array defining what type of table column to create for the field. It can consist of the following entries:

- `type`: Either `\Reef\Storage\Storage::TYPE_INTEGER`, `::TYPE_TEXT`, `::TYPE_BOOLEAN` or `::TYPE_FLOAT`.

- `min`: For `::TYPE_INTEGER`, the minimum possible value to be stored
- `max`: For `::TYPE_INTEGER`, the maximum possible value to be stored
- `limit`: For `::TYPE_TEXT`, the maximum number of characters to be stored (note: characters, not bytes!)

Schema update functions

The `needsSchemaUpdate()` function is called before performing a form update (when editing a stored form). By default it returns `false`, but you may perform custom checks to return `true` whenever you need the schema update functions (`beforeSchemaUpdate`, `afterSchemaUpdate`) to be called. Note that by default, these functions are also always called whenever the flat structure or field name changes with the update. Hence, you only need to implement this method if you have some custom logic going on within your columns.

The functions `beforeSchemaUpdate`, `beforeDelete` and `afterSchemaUpdate` are called within the form update process on the moments the names suggest. They are passed a data array with useful variables, for specifics refer to their docblocks.

`updateDataLoss()`

The `updateDataLoss()` function can be used to indicate whether a field update will lead to data loss. It is checked before updating a form; if there is (possible) dataloss the user is first asked whether he/she wants to proceed. This method is always called on the new field, and receives the old field instance as first argument. It should return either `\Reef\Updater::DATALOSS_NO` if there is a guarantee there is no dataloss, `::DATALOSS_DEFINITE` if there is a guarantee there will be dataloss, or `::DATALOSS__POTENTIAL` otherwise.

`getOverviewColumns()`

This function should return an array with the same keys as `getFlatStructure`, but with values containing the title of the fields that can be used in an overview table or CSV file.

View functions

The methods `view_form` and `view_submission` are called whenever the form HTML or submission HTML are generated, respectively. It receives the `FieldValue` to generate the view for as first parameter, and an option array as second. It should return an array of view variables to be used in mustache. You may override this method to modify the result, but then you should always call the parent function and use its result.

`internalRequest()`

Internal requests routed to this field arrive here. Internal requests serve as communication method between the browser and PHP, to aid interactivity. Depending on the request hash (the first argument), you may return anything to the browser, like an image, JSON object, et cetera. The request paths of fields are formed like `form:{form_uuid}:field:{field_name}:{custom_request_hash}`, where `custom_request_hash` is the hash passed to `Field::internalRequest()`.

The field value class

Warning: Work in progress

Javascript

Warning: Work in progress

5.2 Adding a locale

In contrast to other plugin types, for locales no PHP classes are required to be implemented. You only have to define the translations of the locale in question.

You can either write the required locale files and send a merge request to Reef including them, or add locales on the fly using `setLocale()`. The locale name is an *IETF language tag* with the dash (-) replaced with an underscore (_). All plugins should always include a `en_US` locale, others may be added as desired.

5.2.1 Adding a locale to Reef (merge request)

If you want to add locale files to Reef in a merge request, you will have to translate the following files into your desired locale:

- `locale/en_US.yml`
- `src/Components/{{component_name}}/locale/en_US.yml` (for each component)

Create a merge request containing translations of these files, and we may merge them into Reef.

5.2.2 Adding/setting locale on the fly

You may want to override some locale or add some custom locale. This can be done on multiple levels:

1. At the very base, Reef defines some language strings in `locale/en_US.yml`, which can be edited using `\\Reef\\Reef::setLocale()`.
2. On top of the Reef locale, language strings may also be set on component level by using `\\Reef\\Components\\Component::setLocale()`. These can also be set in the component locale file residing in `locale/en_US.yml` relative to the component directory.
3. On top of the Reef locale, language strings may also be set on form level by using `\\Reef\\Form\\Form::setLocale()`.
4. On top of the Form and Component locale, language strings may also be set on field level by using `\\Reef\\Components\\Field::setLocale()`. For a field, translations are fetched in the following order or prevalence: field, form, component, reef.

The `setLocale(string $s_locale, string[] $a_locale)` method is shared between these classes, and receives the locale name as its first parameter, and list of translations as second parameter.

5.3 Creating a layout

Layouts define how forms look like within Reef. By default, Reef ships with the Bootstrap 4 layout, but any other layout can also be added and used.

5.3.1 Layout member functions

Layouts should extend the abstract class `\Reef\Layout\Layout`. Each layout should have a unique name returned by the `getName()` method, and the directory that the layout code resides in should be given by the `getDir()` method. Just as with components, the `getDir()` implementation should be something along the lines of:

```
public static function getDir() : string {
        return __DIR__.'/';
}
```

In addition, layouts have the possibility to use a number of different methods:

`__construct()`

The `__construct()` function is not used by the `Layout` base class, and hence can without limitations be used by the layout itself for initialization purposes.

`init()`

The `init()` function can be used to initialize the layout. It is called just before Reef starts checking the entire setup. The difference with the `__construct()` method is that `init()` receives the Reef setup object as first parameter, allowing you to use the entire Reef setup here.

`getCSS()` and `getJS()`

Just as with components, you can use the `getCSS()` and `getJS()` functions to add any (remote) CSS/JS assets the layout requires.

`getConfig()`

The `getConfig()` function should return the current configuration of the layout.

`view()`

The `view()` function should return an array of template variables that should be passed to the template parser. These will be available in Mustache using `{{layout.the_var_name}}`.

5.3.2 Template files

Naturally, for layouts most work will probably not be in the PHP code, but rather in the Mustache templates. Templates should be added for both the form view (when a user fills in the form) and the submission view (when a user views the filled in values). These two templates should be present for the form itself, allowing you to wrap all fields in some wrapping HTML, and for each component you wish to support (at least all Reef base components).

For each component layout, you should consider adding the following hook template keys to your template files:

- `[[form_label_before]]` before the form label in `form.mustache`
- `[[form_label_append]]` within the form label, after the text, in `form.mustache`
- `[[form_label_after]]` after the form label in `form.mustache`
- `[[form_input_before]]` before the form input in `form.mustache`
- `[[form_input_after]]` before the form input in `form.mustache`
- `[[submission_label_before]]` before the form label in `submission.mustache`
- `[[submission_label_after]]` after the form label in `submission.mustache`
- `[[submission_input_before]]` before the form input value in `submission.mustache`
- `[[submission_input_after]]` before the form input value in `submission.mustache`

In addition, each component template file should have a base HTML tag `<div class="form-group">` at least containing the following declarations:

```
<!-- for form.mustache: -->
<div class="form-group {{CSSPRFX}}field {{#field.hasErrors}}{{CSSPRFX}}invalid{{/
↪field.hasErrors}} {{field.field_classes}}" data-{{CSSPRFX}}type="reef:the_component_
↪name" data-{{CSSPRFX}}name="{{field.name}}" {{{field.visible}}}>
<!-- for submission.mustache: -->
<div class="form-group {{CSSPRFX}}field {{field.field_classes}}" data-{{CSSPRFX}}type=
↪"reef:the_component_name" data-{{CSSPRFX}}name="{{field.name}}" {{{field.visible}}}>
```

Of source, the errors and name parts are only required if they are applicable to the component in question.

5.4 Creating an extension

Extensions can be used to extend Reef in ways that are not possible using any of the other methods. You can add extensions to for example add configuration values to components or add some HTML to fields.

5.4.1 Extension member functions

Extensions should extend the abstract class `\Reef\Extension\Extension`. Each extension should have a unique name returned by the `getName()` method, and the directory that the extension code resides in should be given by the `getDir()` method. Just as with components, the `getDir()` implementation should be something along the lines of:

```
public static function getDir() : string {
        return __DIR__.'/';
}
```

In addition, extensions have the possibility to use a number of different methods:

`__construct()`

The `__construct()` function is not used by the `Extension` base class, and hence can without limitations be used by the extension itself for initialization purposes.

init()

The init() function can be used to initialize the extension. It is called as soon as the extension is added to the Reef setup class. The difference with the __construct() method is that init() receives the Reef setup object as first parameter, allowing you to use the entire Reef setup here.

getCSS() and getJS()

Just as with components, you can use the getCSS() and getJS() functions to add any (remote) CSS/JS assets the extension requires.

checkSetup()

The checkSetup() function can be used to check the extension configuration. It is called by the checkSetup() function in the Reef setup class, which is called as soon as a Reef object is created. In this method you can check whether any dependencies are present that should be present, and whether there are any incompatibilities.

nativelySupportedLayouts()

The nativelySupportedLayouts() function can be used to override the layouts supported by the extension, if it uses templating. If you do not, you may override this function to return null.

addLayout()

Probably you cannot support each layout that's around. Hence, if a layout wishes to build in support for this extension, the addLayout() function can be used by the layout to add a layout directory for that layout for this extension.

getSubscribedEvents()

This function can be used to define listeners to events. Events are triggered by Reef (or e.g. an other extension) in order for extensions to be able to hook on. Each event has a unique name, in the form {vendor}.{event_name}. The getSubscribedEvents function returns an associative array defining a mapping between event names and function names. For example, if you wish to perform the (self-defined) extension member function component_configuration() when the reef.component_configuration event is triggered, you may use:

```
public static function getSubscribedEvents() {
        return [
                'reef.component_configuration' => 'component_configuration',
        ];
}
```

The component_configuration() function then receives an associative array of variables passed by the event. Make sure you fetch these by reference in order to be able to edit them:

```
public function component_configuration(&$a_vars) {
        // ...
}
```

5.4.2 Layout hooks

You can add HTML to layouts using layout hooks. A template hook is defined by a template hook tag `[[...]]` containing a template hook name, e.g. `[[form_input_after]]`, a hook after the input in a component form template. If you define a file `./view/{layout_name}/{hook_name}.mustache` in your extension, this template will be inserted at the position of the hook in the HTML. The hook tags are preprocessed and cached by Reef before being passed to Mustache, hence:

1. you should empty the Reef cache after editing a hook file, as Reef does not check modifications of hook files for performance reasons,
2. you have complete access to the entire scope of the mustache template you are hooking into, including any variables you may have added using the `reef.component_configuration` event.

5.5 Creating an icon set

Using icon sets, you can customize the icons used within Reef to use your own set of icons. In practice, an icon set is actually just an extension including some hook template files. For more information on how extensions work, please refer to the page on extensions. For an example of an icon set extension, you can take a look at the FA4IconSet.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search