Docs

Form Project

Derafu Form

Anonymous · 09/11/2026 · #php

Table of Contents

Form Project 3
- ***Introduction*** 4
- ***Translations*** 9
- ***CSRF Protection*** 16
- ***Captcha*** 19
- ***Validation*** 22
- ***Required Fields*** 30
- ***Rendering Options*** 34
- ***Examples*** 37
  - From Data 38
  - Control 39
  - User Registration 40
  - User Login 41
  - User Profile with Cards 42
  - User Profile with Tabs 43
  - Setup Wizard 44
  - Invoice Issuance 45
  - Advanced Invoice Search 46
  - E-Invoicing Configuration 47
  - Contact Form 48
  - User Registration 49
  - Collection 50
  - Editors 51
  - Product Ecommerce 52
  - Rules 53
  - Cascade 54
- ***Service Configuration*** 55

Form Project

Derafu Form

Anonymous · 09/11/2026 · #php

Derafu Form

Last updated on 09/11/2026

Introduction

PHP form library with declarative JSON Schema and UI Schema definitions, rendering, validation and CSRF protection

Anonymous · 09/11/2026

Declarative Forms, Seamless Rendering

A modern PHP form library that leverages a declarative, schema-based approach to form definition and rendering, compatible with JSON Forms while providing a powerful backend-centric workflow.

Key Features

- **Schema-Based Form Definition**: Define forms using structured arrays that clearly separate data schemas from UI layouts.
- **Backend-First Approach**: Built for PHP developers who want control over form generation with minimum effort.
- **Symfony-Compatible Twig Extensions**: Familiar syntax for Symfony developers, minimizing learning curve.
- **Decoupled Rendering System**: Separate your form logic from its presentation with specialized renderers.
- **JSON Forms Compatibility**: Define once, use anywhere - same schema can be used with frontend JSON Forms library.
- **Automatic Form Generation**: Generate form UI schemas automatically from your data models.
- **Extensible Architecture**: Easily add custom types, layouts and renderers.
- **Translatable, Optionally**: Validation error messages and input action labels (show/hide password, copy, etc.) can be translated by wiring an optional translator — see Translations.

Why Choose This Library?

Compared to Traditional PHP Form Libraries

Traditional PHP form libraries often tightly couple form definition with HTML generation, making it difficult to separate concerns. This library takes a different approach:

1. **Declarative Rather Than Imperative**: Define *what* your form should be rather than *how* it

should be built, step by step.
2. **Clear Separation of Concerns**: Schema (data structure) is separate from UI schema (presentation) and data (values).
3. **Powerful Layouts Without HTML Knowledge**: Create complex, multi-column layouts without writing HTML.

Compared to Frontend-Only Solutions

Unlike frontend-only form builders, this library:

1. **Keeps Validation Logic Server-Side**: Where it belongs for security-critical applications.
2. **Provides PHP-Native Form Definition**: No need to write JavaScript to define your forms.
3. **Works With or Without JavaScript**: Generate the same forms for both traditional and SPA applications.

Installation

Install the library using Composer:

```
composer require derafu/form
```

Basic Usage

Creating a Simple Form

```
// Define the form structure.
$definition = [
    'schema' => [
        'type' => 'object',
        'properties' => [
            'name' => [
                'type' => 'string',
                'title' => 'Full Name',
            ],
            'email' => [
                'type' => 'string',
                'format' => 'email',
                'title' => 'Email Address',
            ],
        ],
        'required' => ['name', 'email'],
    ],
    'uischema' => [
        'type' => 'VerticalLayout',
        'elements' => [
```

```
            [
                'type' => 'Control',
                'scope' => '#/properties/name',
            ],
            [
                'type' => 'Control',
                'scope' => '#/properties/email',
            ],
        ],
    ],
    'data' => [
        'name' => '',
        'email' => '',
    ],
];

// Create the form.
$form = $formFactory->create($definition);
```

Rendering with Twig

```
{# Simple rendering of the entire form. #}
{{ form(form) }}

{# Or with more control over individual components. #}
{{ form_start(form) }}
    {{ form_row(form.fields.name) }}
    {{ form_row(form.fields.email) }}
    <button type="submit" class="btn btn-primary">Submit</button>
{{ form_end(form) }}
```

Automatic Schema Generation

The library can automatically generate a schema from your data:

```
$definition = [
    'data' => [
        'name' => 'John Doe',
        'email' => 'john.doe@example.com',
        'age' => 34,
        'birthdate' => "1990-01-01",
    ],
];

// Schema and UI schema will be automatically generated.
$form = $formFactory->create($definition);
```

Integration with JSON Forms

This library's form definitions are compatible with the JSON Forms JavaScript library, allowing you to use the same definitions for both server-side and client-side rendering:

```
<div id="json-form-container"></div>
```

```
//
https://raw.githubusercontent.com/derafu/form/refs/heads/main/assets/js/jsonforms-viewer.js
import './jsonforms-viewer.js';

// Render the form and save the reference.
const formDefinition = {{ jsonFormsDefinition | json_encode | raw }};
const formInstance = window.renderJsonForm('json-form-container', formDefinition);
// You can now use `formInstance` to get the form data or its errors.
```

Development Requirements

To render forms using JSON Forms you must install:

```
npm install @jsonforms/core @jsonforms/react @jsonforms/material-renderers \
    react react-dom @mui/material @emotion/react @emotion/styled
```

Form Layouts

The UI schema supports various layout types by default:

- **VerticalLayout**: Fields stacked vertically.
- **HorizontalLayout**: Fields arranged horizontally, in columns of the same width or of the size that each one asks for, see Rendering options.
- **Group**: Logical grouping of fields, often with a label.
- **Categorization**: Tab-based layouts for complex forms.

Example:

```
$uischema = [
    'type' => 'VerticalLayout',
    'elements' => [
        [
            'type' => 'Group',
            'label' => 'Personal Information',
            'elements' => [
                ['type' => 'Control', 'scope' => '#/properties/name'],
```

```
                ['type' => 'Control', 'scope' => '#/properties/email'],
            ],
        ],
        [
            'type' => 'Group',
            'label' => 'Address',
            'elements' => [
                ['type' => 'Control', 'scope' => '#/properties/street'],
                ['type' => 'Control', 'scope' => '#/properties/city'],
            ],
        ],
    ],
];
```

Guides

- Validation: how the rules that process the data are derived from the schema.
- Required fields: the marker of the label, the browser asking for the field, booleans that must be checked.
- Rendering options: where the options come from, which one decides, layouts and columns.
- Translations, CSRF protection and Captcha.

Last updated on 09/11/2026

Translations

Translations

Anonymous · 09/11/2026

Translations

`derafu/form` has four independent, optional translation surfaces. All of them are powered by `derafu/translation`, and all of them do nothing unless you explicitly wire a translator — without one, everything renders in English exactly as before.

1. Validation Error Messages

`FormDataProcessor` runs field values through `derafu/data-processor` rules. When a rule fails, it throws a `Derafu\DataProcessor\Exception\ValidationException` (or one of the other translatable exceptions from that package — see its Translations guide). `FormDataProcessor` accepts an optional translator and locale; when given, it translates each error before adding it to the result:

```
final class FormDataProcessor implements FormDataProcessorInterface
{
    public function __construct(
        FormRulesResolverInterface $resolver,
        ProcessorInterface $processor,
        UiSchemaRuleEvaluatorInterface $evaluator = new UiSchemaRuleEvaluator(),
        ?TranslatorInterface $translator = null,
        ?string $locale = null,
    ) {
    }
}
```

Without a translator, `getErrors()` returns the untranslated (English) messages — the exact same behavior as before this parameter existed.

2. Input Action Labels

Controls rendered with the `Control` UI schema type can declare `options.actions` — input-group buttons like “show/hide password”, “generate a random password”, or “copy the field’s value”. Their labels (and the default copy confirmation message) are resolved by `InputActionResolver`, which also accepts an optional translator:

```
final class InputActionResolver
```

```
{
    public function __construct(
        ?TranslatorInterface $translator = null,
        ?string $locale = null,
    ) {
    }
}
```

An explicit `label` or `message` given directly in `options.actions` is always used as-is and never translated — only the three built-in defaults are:

```
[
    'type' => 'Control',
    'scope' => '#/properties/password',
    'options' => [
        'actions' => ['toggle-password'],
    ],
]
```

`ElementRendererProvider` forwards `InputActionResolver` to `ControlRenderer`. The service configuration injects the application's translator into that resolver. For explicit PHP construction, inject a translator-aware resolver into the provider (see Activating It below).

3. The Texts of a Form

The titles, descriptions, labels, help texts, placeholders and options that the author of a form writes are translated when the form is created. The author writes them in English, as normal text, and each form says in which translation domain they are, with the option `translation_domain` (in the `options` of the definition, where the other options of the form are):

```
# resources/forms/contact-form.yaml
options:
    translation_domain: contact-form

schema:
    type: object
    properties:
        name:
            type: string
            title: Your Name
uischema:
    type: VerticalLayout
    elements:
        - type: Control
          scope: '#/properties/name'
          options:
```

```
            placeholder: Type your name
```

`Derafu\Form\Factory\TranslatingFormFactory` decorates the form factory, which is where every form ends: the loaders (PHP, YAML and JSON) and whoever builds a definition by hand call `create()`. The text in English is the id of the translation, the domain is the `options.translation_domain` of the definition, and the language is the one of the translator when the form is created. A text without an entry in the catalogue stays as it was written, and a definition without `options.translation_domain` is not touched.

The catalogue is `resources/translations/<domain>+intl-icu.<locale>.php` in the directory of the resource provider of the package or the site. A site can have as many domains as it wants: every file of the directory is loaded.

Which texts

Only these keys are texts. Everything else is data or configuration and is never translated.

Where	Keys
The schema: the form and each property (also inside `items`, `$defs` and `definitions`)	`title`, `description`
The options of a property: each element of `oneOf`, `anyOf` and `allOf`	`title`
Each element of the UI schema (also inside `elements`, and inside `options.detail`)	`label`, and `text` for a `Label`
The `options` of an element	`placeholder`, `help`, `footer`, `error_message`, `label`, `unit`, `input_group_prepend_text`, `input_group_append_text`
The `choices` of an element (and `cascade.choices`)	The title of each choice: the values of the dict, never its keys (they are the values of the field)

The list lives in `Derafu\Form\Translation\FormTexts`, and the factory and the audit use the same one, so what is translated and what is audited are always the same.

What is not in the list: PHP

A form that needs to translate something that is not in the list (a text in `options.attr`, for example), or a text with parameters, is written in PHP. `PhpFormLoader` receives an optional translator, and when it has one the closure of the file gets in its context `translator` and `_t`, a function `(string $id, array $parameters = [], string $domain = 'messages')`. What the caller already put in the context under those names is not replaced:

```
// resources/forms/profile.form.php
return function (array $context = []): array {
    return [
```

```
        'options' => ['translation_domain' => 'profile'],
        'schema' => [
            'type' => 'object',
            'properties' => [
                'name' => [
                    'type' => 'string',
                    'title' => $context['_t']('Hello {name}', ['name' =>
$context['user']], 'profile'),
                ],
            ],
        ],
    ];
};
```

The result of the closure also goes through the factory, so the texts of the list are translated too if the definition has `options.translation_domain`. A text that was already translated has no entry in the catalogue and stays as it is.

Auditing the texts

`Derafu\Form\Lint\FormTranslationAudit` reads the definitions of a directory (YAML, JSON and PHP) without running them and checks their texts against the catalogues of the package. It needs `symfony/yaml` and `nikic/php-parser`, as development dependencies. Like the audit of `derafu/translation`, it only gives facts, and the test says which ones are a problem:

```
$report = (new FormTranslationAudit())->audit(
    dirname(__DIR__, 2) . '/resources/forms',
    new MyPackageTranslationResourceProvider(),
);

$this->assertFalse($report->nothingFound);
$this->assertSame([], $report->describe($report->dynamicTexts));
$this->assertSame([], $report->describe($report->withoutDomain));
$this->assertSame([], $report->describe($report->missingTranslations));
$this->assertSame([], $report->describe($report->notUsedBySources));
```

Property	The fact
`texts`	The texts that can be read and have a domain.
`dynamicTexts`	Texts that are not a literal, so they can not be checked: a call of a local function (`$t('Name')`), a concatenation, a call of `_t` with a variable.
`withoutDomain`	Texts of definitions without `options.translation_domain`: they are not translated.
`missingTranslations`	Texts without an entry in the catalogue of their domain.

Property	The fact
`notUsedBySources`	Entries of the catalogues, in the domains that the definitions use, that no text uses. If the domain is shared with templates or code, use it as a list to subtract from (the other audits flag these entries as unused).

The calls `$context['_t']('Text', [], 'domain')` are found wherever they are, because they are written for what is not in the list. To have a PHP form audited, write them with that form; a local function (`$t = fn ...`) works, but its texts are reported as dynamic and are not audited. A PHP definition is read when it is an array written in the file, and a part built in a variable is not read.

The loaders in the container

`form-services.yaml` registers `PhpFormLoader` and `YamlFormLoader` for the forms of the application, with the directory `resources/forms` of the project (`<name>.form.php` and `<name>.form.yaml`). They are private services: an application that does not use them does not have them in its container. The `PhpFormLoader` gets the translator of the container, so its closures receive `_t`, and both create the form with the factory that translates. `FormLoaderInterface` is not set: which loader it is, or a chain of them (`ChainFormLoader`), is a decision of each application:

```
services:
    Derafu\Form\Contract\Loader\FormLoaderInterface:
        alias: Derafu\Form\Loader\PhpFormLoader
```

What Ships

`derafu/form` includes:

- `resources/translations/form+intl-icu.es.php` — a Spanish translation for the four strings `InputActionResolver` can produce (validation error messages come from `derafu/data-processor`'s own translations instead — `derafu/form` doesn't duplicate them).
- `Derafu\Form\Translation\FormTranslationResourceProvider` — a `TranslationResourceProviderInterface` implementation pointing at that directory.
- `Derafu\Form\Factory\TranslatingFormFactory` and `Derafu\Form\Translation\FormTexts` (the texts of a form, section 3), and `Derafu\Form\Lint\FormTranslationAudit` to audit them.

Neither does anything on its own — `derafu/form` is a library, it doesn't build or own a `Translator`.

Activating It (Plain PHP)

```
use Derafu\DataProcessor\Processor;
use Derafu\DataProcessor\Registrar\DefaultRuleRegistrar;
use Derafu\DataProcessor\RuleParser;
use Derafu\DataProcessor\RuleRegistry;
```

```
use Derafu\DataProcessor\RuleResolver;
use Derafu\DataProcessor\Translation\DataProcessorTranslationResourceProvider;
use Derafu\Form\Processor\FormDataProcessor;
use Derafu\Form\Processor\FormRulesResolver;
use Derafu\Form\Translation\FormTranslationResourceProvider;
use Derafu\Translation\TranslatorFactory;

$translator = TranslatorFactory::create(
    defaultLocale: 'es',
    fallbackLocales: ['es', 'en'],
    resourceProviders: [
        new DataProcessorTranslationResourceProvider(), // Validation error messages.
        new FormTranslationResourceProvider(),           // Input action labels.
    ],
);

$registry = new RuleRegistry();
(new DefaultRuleRegistrar())->register($registry);

$processor = new FormDataProcessor(
    new FormRulesResolver(),
    new Processor(new RuleResolver($registry), new RuleParser()),
    translator: $translator,
    locale: 'es',
);
```

For the action labels, build the renderer with a translator-aware `ElementRendererProvider`:

```
use Derafu\Form\Renderer\ElementRendererProvider;
use Derafu\Form\Renderer\ElementRendererRegistry;
use Derafu\Form\Renderer\FormRenderer;
use Derafu\Form\Renderer\Support\InputActionResolver;

$elementRegistry = new ElementRendererRegistry(new ElementRendererProvider(
    actionResolver: new InputActionResolver($translator, 'es'),
));
$formRenderer = new FormRenderer($templateRenderer, $elementRegistry,
$widgetRegistry);
```

Activating It (Dependency Injection)

If your application uses `symfony/dependency-injection`, import both package configuration files. `form-services.yaml` injects `InputActionResolver` into its default element provider. `ElementRendererProviderInterface` combines all tagged providers; its registry is injected into `FormRendererInterface`. Register your own `Translator` to translate the action labels. The configuration also decorates the form factory with `TranslatingFormFactory`, which needs the application's `TranslatorInterface`. Register the translator under that interface:

```
imports:
    - { resource: '../vendor/derafu/data-processor/resources/config/data-processor-
services.yaml' }
    - { resource: '../vendor/derafu/form/resources/config/form-services.yaml' }

services:
    Symfony\Contracts\Translation\TranslatorInterface:
        class: Symfony\Component\Translation\Translator
        factory: ['Derafu\Translation\TranslatorFactory', 'create']
        arguments:
            $defaultLocale: 'es'
            $fallbackLocales: ['es', 'en']
            $resourceProviders: !tagged_iterator derafu_translation.resource_provider
```

Both `DataProcessorTranslationResourceProvider` and `FormTranslationResourceProvider` are already tagged `derafu_translation.resource_provider` in their respective recipe files, so they're picked up automatically. `FormDataProcessor` and `InputActionResolver` both autowire `?TranslatorInterface`, so they resolve to the interface registration above; neither needs explicit argument wiring.

4. Array Reading Errors

Form definitions and typed data readers use Derafu Array and Derafu Value for array/type validation. Failures from the readers, including `ProcessResult::getReader()` for a non-array result, follow the Array exception contract. Register Array's translation resource provider to translate those messages. Direct Value operations follow Value's exception contract.

These read failures are exceptions; they are not field validation errors added to `ProcessResult::getErrors()`. Form processing and the Data Processor translation setup described above retain their own contracts.

Last updated on 09/11/2026

CSRF Protection

The CSRF token of the forms, and the errors of a form as a whole

Anonymous · 09/11/2026

CSRF Protection

A form is **protected with a CSRF token** unless it says otherwise. The form renders a hidden field `_token` with a token, and when its data is processed the token that comes back is checked. `derafu/form` does not know where tokens are kept, nor what a session or a request is: that is what a **CSRF token manager** knows, and the application gives one.

The token manager

```
namespace Derafu\Form\Contract\Csrf;

interface CsrfTokenManagerInterface
{
    // The token for an id (the form it is for).
    public function getToken(string $id): string;

    // Whether a token that came with a form is the one of the id for this user.
    public function isValid(string $id, string $token): bool;
}
```

The form asks for a token every time it is shown, and checks the one that comes with the data. How long a token lasts, and whether checking it uses it up, is up to each implementation.

The id is the **name of the schema** of the form, or `form` if it has none, so a token of one form is not good for another.

An implementation is specific to each environment, because it is the one that knows which session the tokens belong to (for example, one that wraps the token manager of Symfony, or one that works over the session of a Mezzio request). The package defines the contract and does not bring any.

Wiring

With the services of the package (`form-services.yaml`) the manager is **optional**: if the application has a service that implements `CsrfTokenManagerInterface`, the renderer and the processor use it. When assembling services directly, inject it into the renderer and the processor:

```
use Derafu\Form\Renderer\FormRenderer;

$renderer = new FormRenderer(
    $templateRenderer,
    $elementRendererRegistry,
    $widgetRendererRegistry,
    csrfTokenManager: $manager,
);

$processor = new FormDataProcessor($resolver, $dataProcessor, csrfTokenManager:
$manager);
```

A form that is protected needs a manager

Rendering or processing a form that is protected, **without a manager, is an error** (an exception with the name of the form), not a form that is silently left open. Either register a manager or turn the protection off for the form.

Turning it off

It is an option of the form, which both the renderer and the processor read, so they always agree:

```
# resources/forms/search.form.yaml
options:
    csrf_protection: false
```

A form that is not protected renders no token, and its data is processed without one. Use it for forms that do not need it, for example a search form with the method GET, or a form that is posted by clients that are not browsers.

When the token is not valid

The token is **not part of the data**: it is taken out before the fields are processed. If it is missing, or it is not valid, the result is not valid and it has a message in the **errors of the form**:

```
$result = $processor->process($form, $_POST);

$result->isValid();        // false
$result->getFormErrors();  // ['The form is not valid or has expired. ...']
```

The fields are processed all the same, so their own errors come along. The message is translated if the processor has a translator (the text is in the domain `errors`).

The errors of a form as a whole

Some errors do not belong to any field, like this one. They are in `ProcessResult::getFormErrors()`, they count in `hasErrors()` and they come first in `getAllErrors()`. The form that the result gives (`getForm()`) has them (`$form->getErrors()`), and the template of the form shows them **before its fields** with `form_global_errors(form)`:

```
{{ form_global_errors(form) }}
```

The errors of the fields are still shown in their rows, as before.

Last updated on 09/11/2026

Captcha

The captcha of the forms of derafu/form, with the provider of the application

Anonymous · 09/11/2026

Captcha

A form can ask for a **captcha**: the form renders a widget, and when its data is processed what the visitor solved is checked. `derafu/form` does not know which captcha service it is, nor its keys: that is what a **captcha provider** knows, and the application gives one (for example the ones of `derafu/captcha`).

The option of the form

It is an option of the form, in the `options` of its definition, next to `csrf_protection` and `translation_domain`:

```
# resources/forms/contact.form.yaml
options:
    captcha_protection: true
```

Value	What happens
absent	The form is not protected with the captcha. Nothing is rendered or checked, even if the application has a provider.
`false`	The same, written down.
`true`	The form is **protected with the captcha**, always. With a provider available the widget is rendered and the data is checked, and the form is not valid if the visitor did not solve it, or if the service could not be asked (it fails closed). **If the application did not configure a captcha, it is an error**: the form can not be rendered nor processed, and the message says what to configure.

A form is never left open because nobody thought of the captcha. A package that ships forms (a contact form, a login) declares `captcha_protection: true`, and the application configures a captcha once; no variable per form is needed. The simplest one needs no account: ALTCHA, with `CAPTCHA_PROVIDER=altcha` and a secret key (see `derafu/captcha`).

The application can also decide, **on purpose**, not to have a captcha (`CAPTCHA_PROVIDER=none` in `derafu/captcha`): then the forms that are protected have none and there is no error. It is not the same as not configuring anything, which fails.

The provider

```
namespace Derafu\Form\Contract\Captcha;

interface CaptchaProviderInterface
{
    // Whether the application has configured a captcha.
    public function isAvailable(): bool;

    // Whether the application decided, on purpose, not to have one.
    public function isDisabled(): bool;

    // The name of the field that the widget writes in the form.
    public function getResponseField(): string;

    // The HTML of the widget, with its scripts.
    public function getWidget(string $formId): string;

    // Whether what the visitor solved is the proof of a person for that form.
    // It throws CaptchaUnavailableException if the service could not be asked.
    public function verify(string $token, string $formId): bool;
}
```

The id of the form is the name of its schema, or `form` if it has none (`$form->getId()`): the services that have an action (reCAPTCHA v3, Turnstile) use it, so what was solved for a form is not good for another.

Rendering

`form_captcha(form)` renders the widget. `form.html.twig` and `renderBody` call it, after the fields and before the CSRF token, so a form that is rendered whole has it. A template that builds the form by hand writes it where it wants, before the button:

```
{{ form_start(form) }}
    {{ form_element(form.uiSchema, form) }}
    {{ form_captcha(form) }}
    {{ form_csrf(form) }}
    <button type="submit">Send</button>
{{ form_end(form) }}
```

It renders nothing when the form is not protected with the captcha or the application disabled it on purpose.

Processing

`FormDataProcessor` checks the captcha according to the option of the form, **never according to what arrives in the POST**: a request that is made by hand without the field is not valid.

- What the visitor solved is read from the raw data, in the field of the provider, and it is **not part of the processed data**.
- The service is asked last: after the CSRF token and after the fields, and only if they are valid, so it is not asked about a form that is not going to be accepted.
- When it is not valid, the result is not valid and has a message in the **errors of the form** (`getFormErrors()`). There are two messages: “The captcha is not valid. Try again.” and “The captcha could not be verified. Try again in a moment.” (the service did not answer). Both are translatable, in the domain `errors`.

Wiring

With the services of the package (`form-services.yaml`) the provider is **optional**: if the application has a service that implements `CaptchaProviderInterface`, the renderer and the processor use it. Without one, the forms that are protected with the captcha fail. When assembling services directly, inject it into the renderer and the processor:

```
use Derafu\Form\Renderer\FormRenderer;

$renderer = new FormRenderer(
    $templateRenderer,
    $elementRendererRegistry,
    $widgetRendererRegistry,
    captchaProvider: $provider,
);

$processor = new FormDataProcessor($resolver, $dataProcessor, captchaProvider:
$provider);
```

Last updated on 09/11/2026

Validation

How the rules that process the data of a form are derived from its schema, and what each keyword of the schema validates

Anonymous · 09/11/2026

Validation

`FormDataProcessor` processes the data that comes with a form: it casts, sanitizes, transforms and validates the value of each field, and gives back the processed data and the errors. The rules are the ones of `derafu/data-processor`. Nobody writes them for the common cases: they are **derived from the schema of the form**, and the `rules` section of the definition adds the rest.

```
$result = $processor->process($form, $_POST);

if ($result->hasErrors()) {
    $errors = $result->getErrors();     // ['field' => ['message']]
    $form = $result->getForm();         // The form with its errors, to render it
again.
} else {
    $data = $result->getProcessedData();
}
```

Typed Data Readers

`ProcessResultInterface::getReader()` returns a `Derafu\Array\ArrayReader` over the processed data. `getProcessedData()` continues to return the original value as `mixed`; requesting a reader requires that value to be an array, without converting its contents.

```
$result = $processor->process($form, ['contact' => ['address' => 'Street 1']]);
$reader = $result->getReader();

$address = $reader->requireString('contact.address');
$zone = $reader->getString('contact.zone', 'North');
```

The reader is also available on `FormDataInterface`. It is a snapshot: obtain another reader after changing FormData. Objects stored inside the array retain their identity.

```
use Derafu\Form\Data\FormData;

$data = new FormData(['contact' => ['address' => 'Street 1']]);
```

```
$before = $data->getReader();
$data->set('contact.address', 'Street 2');

$before->requireString('contact.address');           // 'Street 1'
$data->getReader()->requireString('contact.address'); // 'Street 2'
```

Reader keys use literal-key precedence and dot notation, including numeric positions. JSON Forms scopes such as `#/properties/contact/properties/address` belong to FormData's existing `get()`, `has()` and `set()` methods; use `contact.address` with the reader.

Strict getters preserve existing types; explicit conversions use `getAsXXX()` / `requireAsXXX()`. Missing or null values return an optional getter's default and fail a required read. The reader does not run form processing rules or change the processing result's validity or errors. See Array Readers and Value Types.

Where the rules come from

For each field the processor joins three sources, in this order:

1. **The property of the schema**: its `type`, `format`, `pattern`, limits, `enum`, etc. (see below).
2. **The control of the UI schema**: a control of type `file` or `image` validates a file or an image (see Files), a control of type `editor` removes the tags of the HTML, and a control of type `ipv4`, `ipv6`, `hostname`, `uuid`, `url`, `time`, `color`, `month` or `week` validates that format.
3. **The `rules` section of the definition**: explicit rules by field.

```
'rules' => [
    'name' => [
        'transform' => ['uppercase'],
        'validate' => ['min_length:3'],
    ],
],
```

The rules of a field are written in four categories, and are applied in this order: `cast`, `transform`, `sanitize`, `validate`. The explicit ones are joined to the derived ones like this:

- `cast`: the explicit one **replaces** the derived one. When the type of the cast changes, the derived sanitizers (for example `trim`, that is for strings) are dropped, and the explicit ones are kept.
- `sanitize`, `transform` and `validate`: the explicit ones are **added after** the derived ones.
- `required`: the fields that are required (see Required fields) have it as the first rule of `validate`, once.

A field whose rule of the UI schema is not active (it is hidden, for example) is not validated and is not in the processed data, even if it came with the data.

Objects

The properties of an object of the schema are fields of the form with the name of their path, `address.city`, and the data has them inside the object, as PHP reads a form (`address[city]` in the name of the control):

```
'schema' => [
    'type' => 'object',
    'required' => ['contact'],
    'properties' => [
        'contact' => [
            'type' => 'object',
            'required' => ['address', 'zone'],
            'properties' => [
                'address' => ['type' => 'string'],
                'zone' => ['type' => 'string'],
                'city' => ['type' => 'string'],
            ],
        ],
    ],
],
'uischema' => [
    'type' => 'VerticalLayout',
    'elements' => [
        ['type' => 'Control', 'scope' => '#/properties/contact/properties/address'],
        ['type' => 'Control', 'scope' => '#/properties/contact/properties/zone'],
        ['type' => 'Control', 'scope' => '#/properties/contact/properties/city'],
    ],
],
```

```
$result = $processor->process($form, ['contact' => ['address' => 'Street 1', 'zone' =>
'North', 'city' => '']]);

$result->getProcessedData();  // ['contact' => ['address' => 'Street 1', 'zone' =>
'North', 'city' => '']]
$result->getErrors();         // ['contact.zone' => [...]] when the zone is missing.
```

- The processed data and the data that comes back to the form have the same shape as the data that came.
- The errors have the name of the path of the field (`contact.zone`).
- A field is required by the `required` of **its own object**: `address` and `zone` are, `city` is not. The `required` of the root only asks for the key `contact`: it does not make any of its fields required.
- A key with the whole name (`'contact.address'`) is the value of that field too.
- The conditions of the rules of the UI schema can be on a field of an object (`#/properties/contact/properties/city`).

Fields that are not in the schema

The form is closed: what comes with the data and is not a field of the form is **not** in the processed data. It is not an error, because a browser can send fields that are not of the form (the button that submitted it, for example). The `additionalProperties` of an object opens it:

`additionalProperties` of the object	The fields of the data that are not in the form
absent, or `false`	are dropped
`true`, or a schema	are kept in the object as they came, without validating them

Each object decides for itself, the root included:

```
'schema' => [
    'type' => 'object',
    'properties' => [
        'name' => ['type' => 'string'],
        'contact' => [
            'type' => 'object',
            'additionalProperties' => true,
            'properties' => ['address' => ['type' => 'string']],
        ],
    ],
],
```

```
$result = $processor->process($form, ['name' => 'Ana', 'extra' => 'x', 'contact' =>
['address' => 'Street 1', 'zip' => '123']]);

$result->getProcessedData();  // ['name' => 'Ana', 'contact' => ['address' => 'Street
1', 'zip' => '123']]
```

`extra` is dropped because the root is closed, and `zip` is kept because `contact` is open. The CSRF token and the response of the captcha are never in the processed data, and neither is a field that is not active.

Fields that depend on another

`dependentRequired` makes some fields of an object required when another one has a value:

```
'contact' => [
    'type' => 'object',
    'dependentRequired' => ['card' => ['address', 'zip']],
    'properties' => [
        'card' => ['type' => 'string'],
        'address' => ['type' => 'string'],
        'zip' => ['type' => 'string'],
```

```
    ],
],
```

```
$result = $processor->process($form, ['contact' => ['card' => '4111', 'address' => '',
'zip' => '']]);

$result->getErrors();  // ['contact.address' => ['Field is required.'], 'contact.zip'
=> ['Field is required.']]
```

- A field has a value when it is not `null`, `''` or `[]`, as `required` reads it. The exception is the field that triggers: a checkbox that is not checked (`false`) is not a value, so "if it asks for an invoice, the tax id is required" does not ask for it when the box is empty.
- The error is the one of `required`, in the path of each field that is missing.
- The names are the ones of the object: in `contact`, `card` is `contact.card`. A name that is not a field of the form, or a field that is not active, is not required and does not trigger.

`minProperties` and `maxProperties` are not derived: the properties of an object are the fields of the schema, so their number does not depend on what the visitor sends.

What the schema validates

In the schema	Rule
`type: string`	cast `string`; sanitize `trim`
`type: integer`	cast `integer`; validate `int`
`type: number`	cast `float`; validate `numeric`
`type: boolean`	cast `boolean`
`minLength`, `maxLength`	`min_length`, `max_length`
`minimum`, `maximum` (integer and number)	`gte`, `lte`
`exclusiveMinimum`, `exclusiveMaximum`, `multipleOf` (integer and number)	`gt`, `lt`, `multiple_of`
`minItems`, `maxItems`, `uniqueItems`	`min_items`, `max_items`, `unique`
`contains`, `minContains`, `maxContains`	`contains`, `min_contains`, `max_contains` (see Arrays)
`pattern`	`regex` (see below)
`enum`, `const`, or `oneOf` with `const`	`in` with the values
`items` with `enum` or `oneOf` (a multiple selection)	`in` with the values: each element must be one of them
`contentMediaType: application/json`	`json`
`format`	see the table below

A string is trimmed, except the one of a control of type `password` or with `format: password`: the spaces at its ends are part of it. A string with `format: email` is also transformed to lowercase.

The values of `in` are compared as text, and a comma that is in a value is escaped (`\,`) so it does not split it.

Formats

`format`	Rule
`email`	`email`
`uri`, `url`	`url`
`date`	`date_format:Y-m-d`
`date-time`	`datetime_local`: `YYYY-MM-DDTHH:MM[:SS]`, as `<input type="datetime-local">` sends it
`time`	`time`: `HH:MM[:SS]`
`color`	`color`: `#rrggbb`
`month`	`month`: `YYYY-MM`
`week`	`week`: `YYYY-Www`
`tel`	`regex` of telephone numbers
`hostname`	`hostname`
`ipv4`, `ipv6`	`ip:v4`, `ip:v6`
`uuid`	`uuid`
`base64`	`base64`
`json`	`json`
`regex`	`regex_pattern`: the value must be a regular expression
another	no rule: it only changes the control that is drawn

Patterns

The `pattern` of JSON Schema is written without delimiters, like the browsers read the `pattern` attribute of HTML. PHP needs them, so the form gives the pattern the slash as its delimiter and escapes the slashes that it has, once: `^a/b$` and `^a\/b$` are the same pattern. The same conversion is used by the patterns of the types and by the conditions of the rules of the UI schema.

The regular expression is a PCRE one, the dialect of PHP. A pattern that can not be compiled is an error of the definition, not of the value: it is reported when the field is processed. A field with `format: regex` is the opposite: the **value** is the pattern, and it is invalid if it can not be compiled.

Booleans

For JSON Schema, `required` says that the key must be in the data, not that it has a given value. A boolean that is only `required` is valid when it is checked and when it is not. To ask for a value, fix it in the schema:

```
'terms' => [
    'type' => 'boolean',
    'title' => 'I accept the terms',
    'const' => true,          // or 'enum' => [true]
],
```

- `const: true` or `enum: [true]` validate with `accepted`: the value must be `true`.
- `const: false` or `enum: [false]` validate with `declined`: the value must be `false`.

A browser does not send a checkbox that is not checked, so a missing key is read as `false`, and the form that asks for `true` fails. The control has the native `required` too, see Required fields.

Arrays

`contains` is derived only when its schema is a `const` or an `enum`, because the rules compare the elements with a list of values:

In the schema	Rule
`contains: {const: 'terms'}`	`contains:terms`
`contains: {enum: ['a', 'b']}`	`contains:a,b`
`contains` and `minContains: 2`	`min_contains:2,a,b`
`contains` and `minContains: 0`	no rule for the minimum
`contains` and `maxContains: 3`	`max_contains:3,a,b`

Without `minContains`, at least one element must be one of the values. The elements are compared as text, as `in` does. `minContains` and `maxContains` without `contains` are ignored, and so is a `contains` with another schema, or with values that the rules never match (a boolean, `null`, an array, an empty text).

Files

A control of type `file` or `image` has only the rules of the file: its value is an upload, so it is not cast or sanitized. The options of the control are:

- `max_size` and `min_size`: a number of bytes, or a size with a unit (`2M`, `1.5Mi`).
- `accept`: the extensions (`.pdf`) and the types (`application/pdf`, `image/*`) that are accepted. The file is checked by its **content**, not by its name or by what the browser says. The types of an

explicit `mimetype` rule are joined with the ones of `accept`.

Read-only fields

A property with `readOnly: true` is not taken from what came with the data. Its value is the one that the form has, the same that was drawn: the data of the form or, if it has none, the `default` of the schema, and `null` if there is neither. No rule is applied to it, so a `required` read-only field without a value is not an error.

```
// The form has the data ['code' => 'A-1'], and `code` is read-only.
$result = $processor->process($form, ['code' => 'HACK']);

$result->getProcessedData();  // ['code' => 'A-1']
```

The rules of the UI schema that depend on the field read the value of the form too, so what the visitor sends can not change them.

Translated errors

The messages of the errors are translated when the processor has a translator, see Translations.

Last updated on 09/11/2026

Required Fields

How a field is required, how its label and its control say it, and how the browser asks for it

Anonymous · 09/11/2026

Required Fields

A field is **required** when its property is in the `required` list of the object that has it, as in JSON Schema:

```
'schema' => [
    'type' => 'object',
    'required' => ['name', 'email'],
    'properties' => [
        'name' => ['type' => 'string'],
        'email' => ['type' => 'string', 'format' => 'email'],
        'nick' => ['type' => 'string'],
    ],
],
```

The form marks each of its fields (`$field->isRequired()`), and three things use it:

- The **processor** validates the value with the rule `required` before any other (see Validation).
- The **label** shows the marker of a required field.
- The **control** says that it is required to the browser and to the assistive technologies.

The marker of the label

The label of a required field has the class `required` and an asterisk (`<span class="text-danger">*</span>`). It is shown unless the option `hide_required_asterisk` is `true`:

```
// In the options of the form: all of its fields.
'options' => ['hide_required_asterisk' => true],

// In a control: only that field.
['type' => 'Control', 'scope' => '#/properties/name', 'options' =>
['hide_required_asterisk' => true]],
```

It can also be given when rendering (`{{ form_row(field, {hide_required_asterisk: true}) }}`). The option of the control decides first, then the ones that are given when rendering and last the ones of the form (see Rendering options). A value that is not a boolean is an error.

What the browser asks for

By default the control of a required field has the **native** `required`, so the browser does not send the form until it has a value:

Control	Attribute
Input (except `hidden`), textarea, select	`required`
Radio buttons	`required` in the first one (it is enough for the group), and `aria-required="true"` in the group (`role="radiogroup"`)
Group of checkboxes	`aria-required="true"` in the group (`role="group"`): a browser can not ask for one of several
Slider	none: it always has a value

The form can say that the browser does not ask for it, with the option `native_required`:

```
'options' => ['native_required' => false],
```

Then the controls only have `aria-required="true"` (the assistive technologies say that the field is required, and the browser lets the form go). `native_required` is read like `hide_required_asterisk`: from the control, from what is given when rendering, from the form.

When the form does not say it, `native_required` is the opposite of `novalidate`: a form with the attribute `novalidate` (`'options' => ['attributes' => ['novalidate' => true]]`) is not validated by the browser, so it does not ask for the required fields either. An explicit `native_required: true` wins over `novalidate`.

The widgets and the labels that are rendered alone (`form_widget(field)`, `form_label(field)`) have the same behavior as the ones inside a form: it is the field that carries what its form decided.

```
$field->isRequired();                // It is in the `required` list.
$field->isNativeRequired();          // Its control has the native `required`.
$field->isRequiredAsteriskHidden();  // Its label does not show the asterisk.
```

Booleans

`required` says that the key must be in the data, not that it has a value, so a boolean that is only required has no marker and no `required` in its control. To ask for `true` (to accept the terms of a service) the schema says it:

```
'terms' => [
    'type' => 'boolean',
    'title' => 'I accept the terms',
```

```
    'const' => true,        // or 'enum' => [true]
],
```

Its checkbox always has the native `required` and its label the asterisk, and the processor validates it with `accepted` (see Validation).

Fields that are not always there

A field with a rule of the UI schema that is not active when the form is rendered (it is hidden, or it is not enabled) has its controls `disabled`: the browser does not validate them nor sends them. When the data of the form changes, `ui-schema-rules.js` disables and enables them again as the rule says.

The processor does not validate the field that is not active either, even if its value came.

Tabs and sections that are closed

A browser validates the controls that are in a tab that is not active or in a section that is collapsed, but it can not focus them: it blocks the form and shows nothing. `invalid-reveal.js` solves it. When the form is invalid, it opens the tabs and the sections that have the first invalid control, focuses it and shows the message of the browser. It uses Bootstrap 5, the same that the templates of the categorizations and of the groups use.

```
import { InvalidReveal } from 'derafu-form/invalid-reveal.js';

document.addEventListener('DOMContentLoaded', () => InvalidReveal.init());
```

(With the Symfony bundle the module is in the import map under `derafu-form/`; without it, import it from `vendor/derafu/form/resources/js/invalid-reveal.js`.) If the control is already visible, the script does nothing.

Editors

The JSON editor (`json-editor.js`) and the HTML editor (`html-editor.js`) hide the control that they replace. A hidden control can not be focused, so both remove its `required`; the processor still asks for the value.

Setting the content of the JSON editor from code

`json-editor.js` exports two functions to work with the editor that replaces a control, given the control itself:

```
import { getJsonEditor, setJsonEditorValue } from 'derafu-form/json-editor.js';
```

```
const field = document.querySelector('[name="documents"]');

// Replace the content of the editor.
setJsonEditorValue(field, [{ name: 'Example' }]);

// The JSONEditor instance, to use any of its methods (undefined without editor).
const editor = getJsonEditor(field);
```

`setJsonEditorValue()` also updates the control, which is what the form submits: the editor does not notify the changes that are made by code. It throws an `Error` if the control has no editor.

Last updated on 09/11/2026

Rendering Options

Where the options of an element come from, which one decides, and the options of the layouts, the groups and the columns

Anonymous · 09/11/2026

Rendering Options

The way an element is rendered depends on its **options**, and they can be given in several places. The one that is closest to the element decides.

Which option decides

From the one that decides first to the one that decides last:

1. **The options of the element**: the `options` of its definition in the UI schema.
2. **The options that are given when rendering**: the second argument of the Twig functions (`form(form, options)`, `form_row(field, options)`, `form_element(element, form, options)`...) or the `$options` of the renderer. They go down to all the elements, so they are the defaults of the whole tree.
3. **The options of the form**: the `options` section of its definition.
4. **The defaults of the renderer.**

```
'options' => ['floating_labels' => false],           // 3. All the fields.
```

```
{{ form(form, {floating_labels: true}) }}             {# 2. Overrides the form. #}
```

```
['type' => 'Control', 'scope' => '#/properties/name', 'options' => ['floating_labels'
=> false]],  // 1. Wins.
```

The options of a layout, of a group or of a categorization are for itself: its children do not get them.

Options of the layouts

Element	Option	What it does
`VerticalLayout`	`layout_attr`	Attributes of its container (`class`, `id`, `data-*`...)
`HorizontalLayout`	`layout_attr`	Attributes of its row
`Group`	`group_attr`	Attributes of its card

Element	Option	What it does
`Group`	`collapsible`	The group can be collapsed
`Group`	`collapsed`	It starts collapsed
`Group`	`footer`	HTML of the footer of the card
`Categorization`	`categorization_attr`	Attributes of its container
`Categorization`	`tab_style`	`horizontal` (by default) or `vertical`
`Categorization`	`vertical_col_width`	Columns of the tabs when they are vertical (3 by default)
`Category`	`category_attr`	Attributes of the content of the tab

```
[
    'type' => 'Group',
    'label' => 'Address',
    'options' => ['collapsible' => true, 'collapsed' => true, 'footer' => 'We never
share it.'],
    'elements' => [/* ... */],
],
```

Columns of a horizontal layout

The elements of a `HorizontalLayout` are columns of the row. The ones that do not say their size share what is left, **in equal parts**, so with any number of elements the row is full. An element asks for a size with `col_size`, the number of columns of Bootstrap (from 1 to 12, or `auto`):

```
[
    'type' => 'HorizontalLayout',
    'elements' => [
        ['type' => 'Control', 'scope' => '#/properties/street', 'options' =>
['col_size' => 8]],
        ['type' => 'Control', 'scope' => '#/properties/number'],   // The rest: 4.
    ],
],
```

The class of the column is `col-md-8` for the first one and `col-md` for the second.

Options for the children of a layout

A layout or a group can give options to each one of its children by its position, with `elements_options`: a list where the entry `n` is for the child `n`. They count as options that are given to that child when rendering, so the options of the child itself decide before them (see above).

```
[
    'type' => 'Group',
    'label' => 'Address',
    'options' => [
        'elements_options' => [
            ['attr' => ['autocomplete' => 'street-address']],   // For the first
child.
            [],                                                 // The second one has
none.
        ],
    ],
    'elements' => [/* ... */],
],
```

In a `HorizontalLayout` the entry of a child is also the one of its **column**: its `col_size` and its `class` are taken from it, and they decide before the `col_size` that the child asks for.

```
'options' => ['elements_options' => [['col_size' => 3, 'class' => 'text-end'], []]],
```

The end of the form

`form_end` renders the closing tag, and before it the fields that were not rendered yet (`form_rest`), with the same options as the rest of the form. The option `render_rest` set to `false` does not render them:

```
{{ form_start(form) }}
    {{ form_row(form.fields.name) }}
{{ form_end(form, {render_rest: false}) }}
```

Last updated on 09/11/2026

Examples

Examples of how to use the form component.

Anonymous · 09/11/2026

Form Examples

Last updated on 09/11/2026

From Data

The simplest form, created from data only.

Anonymous · 09/11/2026

From Data

Last updated on 09/11/2026

Control

The default controls provided by the form library.

Anonymous · 09/11/2026

Control

Last updated on 09/11/2026

User Registration

Form to create a new user account.

Anonymous · 09/11/2026

User Registration

Last updated on 09/11/2026

User Login

A user login form with username, password and 2FA options.

Anonymous · 09/11/2026

User Login

Last updated on 09/11/2026

User Profile with Cards

User profile form with grouped sections in cards.

Anonymous · 09/11/2026

User Profile with Cards

Last updated on 09/11/2026

User Profile with Tabs

User profile form with tabs and cards.

Anonymous · 09/11/2026

User Profile with Tabs

Last updated on 09/11/2026

Setup Wizard

Multi-step wizard for account setup.

Anonymous · 09/11/2026

Setup Wizard

Last updated on 09/11/2026

Invoice Issuance

Form for creating invoices with basic fields and line items.

Anonymous · 09/11/2026

Invoice Issuance

Last updated on 09/11/2026

Advanced Invoice Search

Complex search form for filtering invoices with conditional fields.

Anonymous · 09/11/2026

Advanced Invoice Search

Last updated on 09/11/2026

E-Invoicing Configuration

Comprehensive configuration form for electronic invoicing.

Anonymous · 09/11/2026

E-Invoicing Configuration

Last updated on 09/11/2026

Contact Form

Simple contact form with default values and validation.

Anonymous · 09/11/2026

Contact Form

Last updated on 09/11/2026

User Registration

Complex user registration form with nested fields and validation.

Anonymous · 09/11/2026

User Registration

Last updated on 09/11/2026

Collection

Collections and arrays in forms.

Anonymous · 09/11/2026

Collection

Last updated on 09/11/2026

Editors

Different editors using controls.

Anonymous · 09/11/2026

Editors

Last updated on 09/11/2026

Product Ecommerce

Product creation form with data processing rules.

Anonymous · 09/11/2026

Product Ecommerce

Last updated on 09/11/2026

Rules

JSON Forms rules in UI Schema.

Anonymous · 09/11/2026

Rules

Last updated on 09/11/2026

Cascade

Select options in a cascade.

Anonymous · 09/11/2026

Cascade

Last updated on 09/11/2026

Service Configuration

Form rendering services, isolated Twig and tagged contributions

Anonymous · 09/11/2026

Service Configuration

Import the production configurations and provide your application's translator. The translator receives the catalogues contributed by the imported packages:

```
imports:
    - { resource: '../vendor/derafu/data-processor/resources/config/data-processor-
services.yaml' }
    - { resource: '../vendor/derafu/form/resources/config/form-services.yaml' }

services:
    Symfony\Contracts\Translation\TranslatorInterface:
        class: Symfony\Component\Translation\Translator
        factory: ['Derafu\Translation\TranslatorFactory', 'create']
        arguments:
            $defaultLocale: en
            $fallbackLocales: [en]
            $resourceProviders: !tagged_iterator derafu_translation.resource_provider
```

Inject `FormFactoryInterface`, `FormRendererInterface` and `FormDataProcessorInterface` into your application services. The factory creates forms from definitions; the renderer and processor share the configured types, rule evaluator and optional CSRF/captcha integrations.

Isolated Rendering

Form rendering owns qualified services such as `Derafu\Twig\Contract\TwigServiceInterface $form` and `Derafu\Renderer\Contract\RendererInterface $form`. Importing Form does not replace the application's Twig or Renderer services. The internal `FormTwigExtension` is loaded only in the form environment. A host framework can register the same extension in its own page environment, with or without a prefix on the function names. The Symfony Form Bundle registers it without a prefix and chooses between the Derafu and the Symfony functions by the type of the first argument.

The package's template directory is contributed automatically. Scalar Twig options belong to `derafu.form.twig.options`, for example `{ extra: false }`. `derafu.form.twig.component_registrar.cache_key` identifies the form-specific component configuration when a PSR cache pool is provided.

Contributions

Each domain supplies only its own providers. The following tags belong to the form environment:

Tag	Provider contract	Contribution
`derafu_form.element_renderer_provider`	`ElementRendererProviderInterface`	Element type to renderer map
`derafu_form.widget_renderer_provider`	`WidgetRendererProviderInterface`	Widget type to renderer map
`derafu_form.twig.template_path_provider`	`Derafu\Twig\Contract\TemplatePathProviderInterface`	Template directories
`derafu_form.twig.extension_provider`	`Derafu\Twig\Contract\ExtensionProviderInterface`	Twig extensions
`derafu_form.twig.component_provider`	`Derafu\Twig\Contract\ComponentProviderInterface`	Twig component classes
`derafu_form.formatter_handler_provider`	`Derafu\Renderer\Contract\FormatterHandlerProviderInterface`	Named formatter handlers
`derafu_form.renderer.engine`	`Derafu\Renderer\Contract\EngineInterface`	Rendering engines

Individual component classes can also use `derafu_form.twig.component`. Providers are processed by descending tag priority. The first contribution for an element/widget type wins. Default element/widget providers and the package's template-path provider have priority `-1000`, so ordinary consumer providers can override their own entries without copying the defaults. Twig retains its first extension of each class; template directories are searched in priority order. Formatter providers use the first handler of each name.

```
services:
    App\Form\TemplatePathProvider:
        tags: ['derafu_form.twig.template_path_provider']

    App\Form\WidgetRendererProvider:
        tags:
            - { name: derafu_form.widget_renderer_provider, priority: 100 }
```

Registries remain injectable through `ElementRendererRegistryInterface` and `WidgetRendererRegistryInterface`. Rendering extensions are resolved when Twig creates its environment, allowing the internal form functions to call the same form renderer without eager construction cycles.

Last updated on 09/11/2026