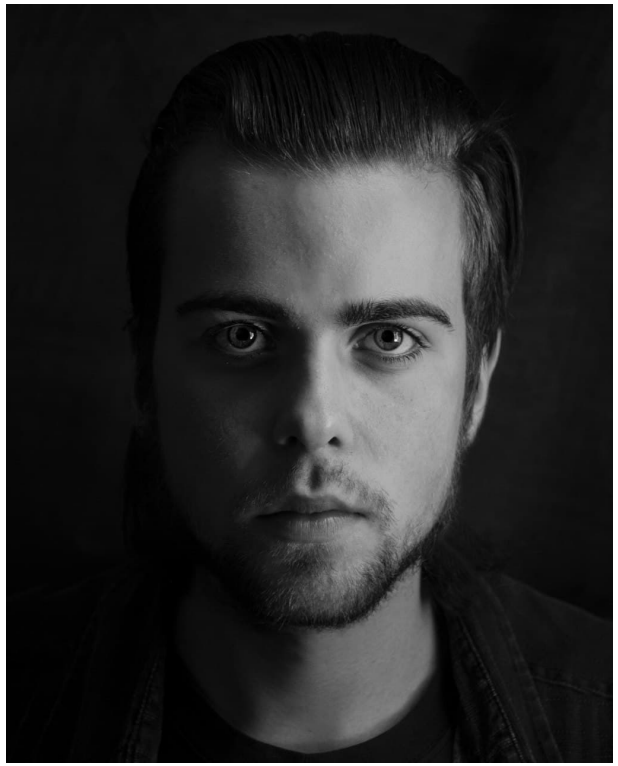
Веб разработка на PHP Symfony

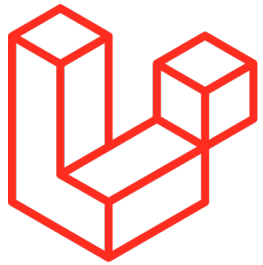
Валидатор и формы

Introduction



Демьян Костельный
Middle PHP Developer

demian-kostelny-613b90151



Тема урока

Валидатор и формы

План урока

1. Описание компонента Validation
2. Валидация скалярных данных
3. Валидация сложных типов данных
4. Компонент Forms
5. Создание форм
6. Процессинг форм
7. Валидация форм

Описание компонента Validation

Описание компонента Validation

Если нам нужно будет перед тем как внести какие-то данные в БД их профильтровать, то нужно будет воспользоваться компонентом Validation. Установить компонент в приложение можно через Composer, если его ещё нету:

```
composer require symfony/validator doctrine/annotations
```

После того как компонент установлен, можно начинать внедрять валидацию в объекты и компоненты приложения.

Описание компонента Validation

Настроить Validation компонент можно в файле config/packages/framework.yaml:

```yaml
# see https://symfony.com/doc/current/reference/configuration/framework.html
framework:
    validation: { enabled: true }
```

Описание компонента Validation

Рассмотрим небольшой пример на Entity под название Product.php и полем в нем name.
Для начала нужно подключить класс валидатора.

```php
namespace App\Entity;

use App\Repository\ProductRepository;
use Doctrine\ORM\Mapping as ORM;

use Symfony\Component\Validator\Constraints as Assert;


/**
 * @ORM\Entity(repositoryClass=ProductRepository::class)
 */
class Product
{
```

Описание компонента Validation

После подключения класса, можем прописать в комментарии параметра $name валидацию на то, что поле не должно быть пустым.

```
/**
 * @ORM\Column(type="string", length=175)
 * @Assert\NotBlank
 */
private $name;
```

Как мы видим, сделали мы это добавив строчку @Assert\NotBlank

Описание компонента Validation

Дальше, мы сможем воспользоваться сервисом валидатора, чтобы использовать валидацию на новый объект прямо в контроллере.

```php
use Symfony\Component\Validator\Validator\ValidatorInterface;
use App\Entity\Product;
```

```php
public function index(ValidatorInterface $validator): Response
{


    $product = new Product();

    $errors = $validator->validate($product);

    if ( count( $errors ) > 0 ) {

        $errorsString = (string) $errors;

        return new Response($errorsString);

    }
```

Описание компонента Validation

В дальнейшем ошибки можно вывести прямо в шаблон.

```
<h3>The author has the following errors</h3>
<ul>
{% for error in errors %}
    <li>{{ error.message }}</li>
{% endfor %}
</ul>
```

Описание компонента Validation

Все доступные параметры, которые можно использовать в Entity, можно посмотреть в документации, https://symfony.com/doc/current/validation.html#using-the-validator-service. Некоторых из них показаны ниже:

Basic Constraints ¶

These are the basic constraints: use them to assert very basic things about the value of properties or the return value of methods on your object.

- NotBlank
- Blank
- NotNull
- IsNull
- IsTrue
- IsFalse
- Type

String Constraints ¶

- Email
- ExpressionLanguageSyntax

Описание компонента Validation

Кроме того что можно задавать параметры валидации через комменты, можно сразу же и задавать более точные значения для валидации поля в объекте:

```
/**
 * @ORM\Column(type="string", length=255)
 * @Assert\Choice({"CD", "Vinyl"})
 */
private $format;
```

В данном примере поле format может принимать только два значения это CD/Vinyl.
Можно также добавить чтобы в случае ошибки и выводило сообщение :

```
/**
 * @ORM\Column(type="string", length=255)
 * @Assert\Choice(
 *      choices = {"CD", "Vinyl"},
 *      message = "Choose one valid format."
 * )
 */
private $format;
```

Валидация скалярных данных

Валидация скалярных данных

Валидация скалярных (или сырых) данных представляет из себя валидацию довольно-таки простых объектов, например, строку или e-mail.
Такая валидация делается прямо в контроллере:

```php
public function addUserMail($email, ValidatorInterface $validator) {

    $emailConstraint = new Assert\Email();  // Мы выбираем класс Email()
                                            // потому что мы как раз хотим проверить входящий e-mail точно ли он
                                            имеет тип e-mail

    $emailConstraint->message = "Enter valid email"; // Создаём сообщение ошибки

    if ( 0 === count( $errors ) ) {
        // Если нету ошибок то выполнить действие прописанное здесь
    } else {
        $errorMessage = $errors[0]->getMessage(); // Достаём сообщение ошибки
    }

}
```

Валидация скалярных данных

В предыдущем примере мы сделали проверку только на одно поле, но можно делать проверку прямо в массиве на все входные данные:

```php
use Symfony\Component\Validator\Validation; // Для начала в контролер нужно подключить класс Validation

$validator = Validation::createValidator(); // Дальше создать объект класса Validation для валидации

$inputData = [ // И для примера давайте создадим массив на котором протестируем валидацию
    'name' => 'Trent',
    'last_name' => 'Reznor',
    'email' => 'info@ninexample.com',
    'age' => 55,
    'tags' => [
        'music',
        'electronic',
        'art'
    ];
];
```

Валидация скалярных данных

После этого нужно прописать сами правила валидации для каждого поля в входных данных:

```
$constraint = new Assert\Collection([
    'name' => new Assert\Length(['min' => 3]),
    'last_name' => new Assert\Length(['min' => 4]),
    'email'     => new Assert\Email(),
    'age'       => new Assert\Positive(),
    'tags'      => new Assert\All()
]);

$errors = $validator->validate( $input, $constraint );
```

Валидация сложных типов данных

Валидация сложных типов данных

Мы рассмотрели то как можно делать валидацию скалярных типов данных, а что мы можем сказать о валидации сложных типов данных? К примеру файлов, изображений и т.п. Валидация таких, более сложных типов данных, не представляет из себя ничего такого сложного. Главное - правильно использовать параметры, которые указываются прямо в классе для валидации, например:

```
/**
 * @ORM\Column(type="string", length=255)
 * @Assert\Image(
 *  minWidth=200
 *  minHeight=200
 *  maxWidth=3000
 *  maxHeight=3000
 * )
 */
private $image;
```

Мы сделали валидацию для поля $image с классом Assert\Image и внутри прописали отдельные параметры на минимальные и максимальные размеры изображения.

Валидация сложных типов данных

То же самое мы можем сделать и в контроллере:

```
$imageConstraint = new Assert\Image([
    'minWidth' => 200,
    'minHeight' => 200,
    'maxWidth'  => 3000,
    'maxHeight' => 3000
]);
```

Валидация сложных типов данных

Также пример для валидации файлов с помощью класса Assert\File:

```
/**
 * @ORM\Column(type="string", length=255)
 * @Assert\Image(
 *  minWidth=200
 *  minHeight=200
 *  maxWidth=3000
 *  maxHeight=3000
 * )
 */
private $image;
```

Валидация сложных типов данных

Из всего этого мы в общем получаем вывод, что сложные типы данных представляют валидацию именно для более сложных объектов, где нужно прописывать немало точных параметров для валидации.

Сложные типы данных есть в списке всех типов данных для валидации в Symfony https://symfony.com/doc/current/validation.html#constraints.
Как говорилось, они представляют из себя более сложные классы, которые имеют много параметров для того, чтобы сделать валидацию как нужно.

Компонент Forms

Компонент Forms

Вместо того чтобы писать форму в ручную, в Symfony можно воспользоваться компонентом, который позволяет генерировать и создавать формы с меньшим количеством усилий. Перед тем как начать использовать данный компонент, нужно его установить через Composer:

```
composer require symfony/form
```

Создание форм

Создание форм

Форму можно создавать прямо в контроллере через класс компонента создания форм. Перед этим только надо сразу же подключить типы для полей, которые будут использоваться в форме:

```
use Symfony\Component\Form\Extension\Core\Type\DateType;
use Symfony\Component\Form\Extension\Core\Type\SubmitType;
use Symfony\Component\Form\Extension\Core\Type\TextType;
```

После этого на готовый новый объект класса какого-то Entity создаётся форма на основе полей в этом же Entity:

```
$post = new Post();
$post->setTitle('Enter title here');
$post->setContent('Enter content of post here');

$form = $this->createFormBuilder($post)
    ->add('title', TextType::class)
    ->add('content', TextareaType::class)
    ->add('save', SubmitType::class, ['label' => 'Create new post'])
    ->getForm();
```

Создание форм

Кроме того что формы можно прописывать прямо в контроллере, можно также и создавать для них отдельные классы. Сделать это можно через maker bundle и следующую команду:

```
php bin/console make:form
```

И дальше уже прямо в классе прописывать поля для формы внутри функции buildForm():

```php
use Symfony\Component\Form\Extension\Core\Type\SubmitType;
use Symfony\Component\Form\Extension\Core\Type\TextType;
use Symfony\Component\Form\Extension\Core\Type\TextareaType;

class PostType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('title', TextType::class)
            ->add('content', TextareaType::class)
            ->add('save', SubmitType::class)
        ;
    }
```

Создание форм

Весь список типов данных для полей формы можно посмотреть по сыллке:
https://symfony.com/doc/current/reference/forms/types.html#text-fields

Создание форм

После того как мы создали форму, её нужно вывести в шаблон. Это делается в контроллере через функцию createView().

```
return $this->render('post/new.html.twig', [
    'form' => $form->createView(),
]);
```

И вывести саму переменную формы в шаблоне:

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

Процессинг форм

Процессинг форм

Время поговорить о том, как можно обрабатывать данные, полученные из полей форм. Разработчики Symfony рекомендуют размещать код для обработки данных формы в той же самой функции, где и есть код на рендер формы. Пример:

```php
public function new(Request $request): Response
{
    $post = new Post();

    $form = $this->createForm(PostType::class, $post);
    // Заметьте что форму мы бёрем из класса формы который мы создали

    $form->handleRequest($request); // Даём форме обработку запроса

    if ( $form->isSubmitted() && $form->isValid() ) { // Проверяем была ли отправлена форма

        $post = $form->getData(); // Достаём данные из полей формы

        // Здесь в дальнейшем можно прописать код на сохранение в БД
        // через Entity Post

        return $this->redirectToRoute('post_created');
    }

    return $this->render('post/new.html.twig', [
        'form' => $form->createView()
    ]);
}
```

Валидация формы

Валидация формы

Для того чтобы было возможно делать валидацию в форме для начала нужно установить класс валидатора, который мы уже устанавливали через Composer. Следующим шагом является прописать типы валидации к полям в Entity, который мы будем использовать для формы. После этого можно будет реализовать валидацию внутри класса формы для Entity. Пример:

```
public function configureOptions(OptionsResolver $resolver)
{
    $resolver->setDefaults([
        'data_class' => Product::class,
        'require_title' => true // Указываем что title должен был обязательным
    ]);

    // Кроме этого можно и в самом классе формы указывать какие есть доступные типы данных
    $resolver->setAllowedTypes('require_title', 'string');
}
```

Валидация формы

```php
public function buildForm(FormBuilderInterface $builder, array $options)
{
    $builder
        ->add('title', TextType::class, [
            'required' => $options['require_title'] // Обращаемся к массиву options в котором опции были заданы
            через функцию configureOptions
        ])
        ->add('content', TextareaType::class)
        ->add('save', SubmitType::class)
    ;
}
```

Валидация формы

Параметры валидации для полей можно добавлять также и без массива $options:

```php
public function buildForm(FormBuilderInterface $builder, array $options)
{
    $builder
        ->add('title', TextType::class, [
            'required' => true,
            'label' => 'Название поста' // Добавит тег <label> перед полем
        ])
        ->add('content', TextareaType::class)
        ->add('save', SubmitType::class)
    ;
}
```

Информационный видеосервис для разработчиков программного обеспечения

TestProvider Pearson CyberBionic systematics Microsoft Partner Windows Azure EBA

Проверка знаний

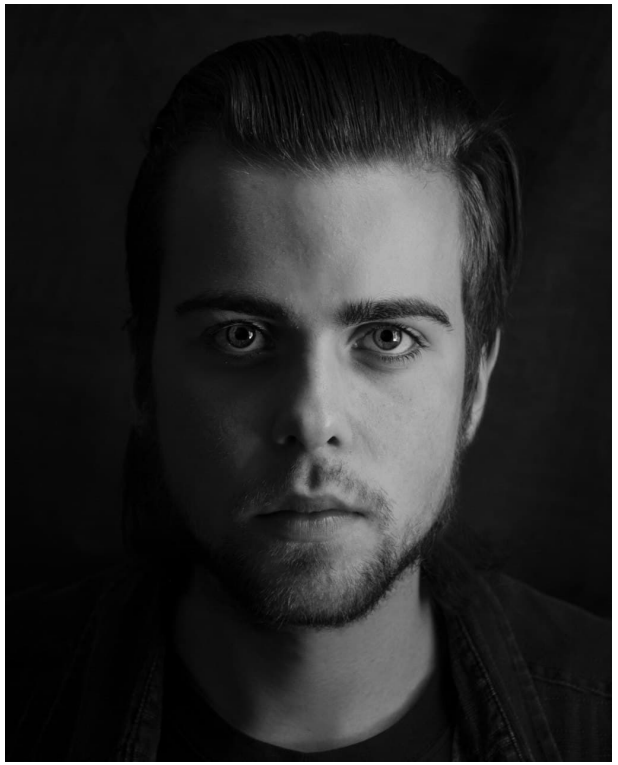
TestProvider.com

TestProvider – это online сервис проверки знаний по информационным технологиям. С его помощью Вы можете оценить Ваш уровень и выявить слабые места. Он будет полезен как в процессе изучения технологии, так и для общей оценки знаний IT специалиста.

Проверьте как Вы усвоили данный материал на TestProvider.com

Успешное прохождение финального тестирования позволит Вам получить соответствующий Сертификат.

Спасибо за внимание! До новых встреч!



Демьян Костельный
Middle PHP Developer