CRUD bundle Documentation

Release 0.1.0

evgenyvas

Jul 28, 2021

Contents

CHAPTER 1

Documentation

1.1 Install

install package via composer

```
composer require evgenyvas/crud-bundle
```

create file *config/routes/crud_routing.yaml* with content:

```
app_crud:
  resource: '@CRUDBundle/Resources/config/routes.yaml'
```

execute commands:

```
bin/console doctrine:migrations:diff
bin/console doctrine:migrations:migrate
```

add parameters for date format in file *config/services.yaml*

```
date_format: 'd.m.Y'
datetime_format: 'd.m.Y H:i:s'
flatpickr_date_format: 'd.m.Y'
flatpickr_datetime_format: 'd.m.Y H:i'
```

include in your base template inside *javascripts* block before other scripts

```
<script src="{{ asset('bundles/fosjsrouting/js/router.min.js') }}"></script>
<script src="{{ path('fos_js_routing_js', { callback: 'fos.Router.setData' }) }}"></
↪script>
```

insert template for modal windows. It must be inside Vue app template

```
{% if modal_size is not defined %}
  {% set modal_size = 'lg' %}
```

(continues on next page)

(continued from previous page)

```
{% endif %}
{% if modal_expand_size is not defined %}
  {% set modal_expand_size = 'xl' %}
{% endif %}
{% if modal_expanded is not defined %}
  {% set modal_expanded = false %}
{% endif %}

{% include '@CRUD/components/modal.html.twig' with {
  'modal_size': modal_size,
  'modal_expand_size': modal_expand_size,
  'modal_expanded': modal_expanded
} %}
```

attach mixins in your Vue instance

```
mixins: [
  gridValue(),
  AjaxModal,
],
```

1.2 List

1.2.1 Base config

create .xml config in directory *config/crud/*. For example, *user.xml*

```
<?xml version="1.0" encoding="utf-8"?>
<view>
  <layout id="users" entity="\App\Entity\User" path="/user" controller=
↪"App\Controller\UserController">
    <action type="list" header="Users">
      <row>
        <position>
          <elem type="content"/>
        </position>
      </row>
    </action>
    <action type="view" header="View user attributes">
      <block name="heading" class="popup-hide"/>
      <block name="heading_title">Viewing user attributes</block>
      <row>
        <position>
          <elem type="content"/>
        </position>
      </row>
    </action>
    <action type="add" header="Create user">
      <block name="heading" class="popup-hide"/>
      <block name="heading_title">Adding a new user</block>
      <row>
        <position>
          <elem type="content"/>
```

(continues on next page)

(continued from previous page)

```
        </position>
      </row>
    </action>
    <action type="edit" header="Editing a user">
      <block name="heading" class="popup-hide"/>
      <block name="heading_title">Editing a user</block>
      <row>
        <position>
          <elem type="content"/>
        </position>
      </row>
    </action>
    <action type="delete"/>
  </layout>
</view>
```

create controller, which will be used for crud operations. For example, *src/Controller/UserController.php*

```
<?php

namespace App\Controller;

use Ecode\CRUDBundle\Service\ObjectFormatter;
use Ecode\CRUDBundle\Traits\CRUDTrait;
use Ecode\CRUDBundle\Traits\FilterTrait;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\EventDispatcher\EventDispatcherInterface;
use Symfony\Contracts\Translation\TranslatorInterface;
use App\Fields\UserFields;

class UserController extends AbstractController
{
    use CRUDTrait, FilterTrait;

    private $dispatcher;
    private $fmt;
    private $translator;

    public function __construct(
        EventDispatcherInterface $dispatcher,
        ObjectFormatter $fmt,
        TranslatorInterface $translator,
        UserFields $fields
    ) {
        $this->dispatcher = $dispatcher;
        $this->fmt = $fmt;
        $this->translator = $translator;
        $this->fields = $fields;
    }
}
```

create fields config. For example in file *src/Fields/UserFields.php*

```
<?php
```

(continues on next page)

(continued from previous page)

```
namespace App\Fields;

class UserFields
{
    public function getAttSettings($params=[]) {
        $att = [
            'id' => [
                'label' => 'Identificator',
                'type' => 'number',
                'widget' => 'number',
                'sort' => false,
                'show_list' => false,
                'show_view' => false,
                'load_list' => true,
                'filter' => false,
                'render_add' => false,
                'show_edit' => false,
                'show_print' => false,
            ],
            'name' => [
                'label' => 'Name',
                'type' => 'string',
                'widget' => 'text',
                'required' => 'required',
                'sort' => true,
                'show_list' => true,
                'show_add' => true,
                'show_edit' => true,
            ],
            'login' => [
                'label' => 'Login',
                'type' => 'string',
                'widget' => 'text',
                'required' => 'required',
                'sort' => true,
                'show_list' => true,
                'show_add' => true,
                'show_edit' => true,
            ],
            'password' => [
                'label' => 'Password',
                'repeat_label' => 'Confirm password',
                'type' => 'string',
                'widget' => 'password',
                'required' => 'required',
                'sort' => false,
                'ignore_format' => true,
                'show_list' => false,
                'render_list' => false,
                'load_list' => false,
                'show_view' => false,
                'show_add' => true,
                'show_edit' => true,
                'show_single' => false,
                'show_print' => false,
                'filter' => false,
                'search' => false,
```

(continues on next page)

(continued from previous page)

```
            ],
            'roles' => [
                'label' => 'Roles',
                'type' => 'json',
                'widget' => 'multichoice',
                'widget_params' => [
                    'expanded' => true, // checkboxes
                    'choices' => [
                        'Administrator' => 'ROLE_ADMIN',
                        'User' => 'ROLE_USER',
                    ],
                ],
                'required' => 'required',
                'sort' => false,
                'filter' => true,
                'show_list' => true,
                'show_add' => true,
                'show_edit' => true,
                'render_add' => true,
                'render_edit' => true,
                'show_single' => false,
                'show_print' => false,
            ],
        ];
        return $att;
    }
}
```

1.2.2 Changing rows ordering

To enable changing rows order:

1. add new column in table

```
<field name="ordering" type="integer" column="ordering" nullable="false"/>
```

2. add field config for this column

```
'ordering' => [
    'label' => 'Ordering',
    'type' => 'number',
    'widget' => 'number',
    'sort' => true,
    'show_list' => false,
    'show_view' => false,
    'load_list' => true,
    'change' => false,
    'filter' => false,
    'render_add' => false,
    'show_edit' => false,
    'show_print' => false,
],
```

Table data must return column named *ordering*. If your column has different name, add config like this:

```
'ordering' => [
    'label' => 'Ordering',
    'type' => 'number',
    'widget' => 'number',
    'sort' => true,
    'show_list' => false,
    'show_view' => false,
    'load_list' => true,
    'change' => false,
    'filter' => false,
    'render_add' => false,
    'show_edit' => false,
    'show_print' => false,
    'format_func' => function($col_val) {
        return $col_val['num'];
    },
],
```

3. add route for save ordering

config/routes.yaml

```
user_table_save_ordering:
  path: /user/table_save_ordering
  methods: [POST]
  options:
    expose: true
  controller: App\Controller\UserController::saveOrdering
```

src/Controller/UserController.php

```
public function saveOrdering(Request $request, EntityManagerInterface $em) {
    $toUpd = json_decode($request->get('toUpd'), true);

    $foundEntity = $this->userRepo->createQueryBuilder('e', 'e.id')
        ->where('e.id IN (:ids)')->setParameter('ids', array_column($toUpd, 'id'))
        ->getQuery()->getResult();
    // set null value to escape unique constraint
    foreach ($foundEntity as $obj) {
        $obj->setOrdering(null);
        $em->persist($obj);
    }
    $em->flush();
    foreach ($toUpd as $upd) {
        $obj = $foundEntity[$upd['id']];
        $obj->setOrdering($upd['val']);
        $em->persist($obj);
    }
    $em->flush();
    return $this->fmt->jsonResponse([
        'status'=>'success',
        'message'=>'Successfully saved',
    ]);
}
```

also add in *beforeSave* method

```
public function beforeSave($obj, $request, $formData) {
    if (!$obj->getId()) {
        // get max ordering
        $res = $this->userRepo->createQueryBuilder('u')
            ->select('MAX(u.ordering)')
            ->getQuery()
            ->getSingleScalarResult();
        $maxOrdering = $res ? $res + 1 : 1;
        $obj->setOrdering($maxOrdering);
    }
}
```

templates/user/list.html.twig

```
{% set save_ordering_route = 'user_table_save_ordering' %}
```

1.3 Form

Create *Data* class for your entity, for example *src/Data/UserData.php*

This class intended for use as class for form instead of using entity class. It usually not content setter methods and data must be handed over through controller.

Validation rules also defined there.

Example:

```
<?php

namespace App\Data;

use Doctrine\Common\Collections\Collection;
use Symfony\Component\Validator\Mapping\ClassMetadata;
use Symfony\Component\Validator\Constraints as Assert;

class UserData
{
    private $name;
    private $login;
    private $password;
    private $curPassword; // for changing password
    private $roles;

    public function __construct(
        $name,
        $login,
        $password,
        $roles
    ) {
        $this->name = $name;
        $this->login = $login;
        $this->password = $password;
        $this->roles = $roles;
    }

    // validation rules
```

(continues on next page)

(continued from previous page)

```
    public static function loadValidatorMetadata(ClassMetadata $metadata) {
        $metadata->addPropertyConstraint('name', new Assert\NotBlank());
        $metadata->addPropertyConstraint('login', new Assert\NotBlank());
        $metadata->addPropertyConstraint('password', new Assert\NotBlank(['groups' =>␣
↪['add']]));
        $metadata->addPropertyConstraint('roles', new Assert\NotBlank());
    }

    public function getName(): ?string {
        return $this->name;
    }

    public function getLogin(): ?string {
        return $this->login;
    }

    public function getPassword(): string {
        return (string)$this->password;
    }

    public function getCurPassword(): string {
        return (string) $this->curPassword;
    }

    public function setCurPassword(?string $curPassword): self {
        $this->curPassword = $curPassword;
        return $this;
    }

    public function getRoles(): array {
        return ($this->roles and is_array($this->roles)) ? $this->roles : [];
    }
}
```

It's recommended to define separate methods for manage form, because it allows to write specific logic. For example, for user before saving in database password must be hashed. To do this define in UserController method *beforeSave*:

```
use Symfony\Component\PasswordHasher\Hasher\UserPasswordHasherInterface;
.
.
.


    public function beforeSave($obj, $request, $formData) {
        if ($formData->getPassword()) {
            // password encode
            $obj->setPassword($this->passwordHasher->hashPassword($obj, $formData->
↪getPassword()));
        }
    }
```

1.4 Fields

1.4.1 text

```
'name' => [
    'label' => 'Name',
    'type' => 'string',
    'widget' => 'text',
    'required' => 'required',
    'sort' => true,
    'show_list' => true,
    'show_add' => true,
    'show_edit' => true,
],
```

1.4.2 textarea

only in form

```
'type' => 'string',
'widget' => 'textarea',
```

1.4.3 password

```
'label' => 'Password',
'repeat_label' => 'Confirm password',
'type' => 'string',
'widget' => 'password',
```

also for encode password add to controller:

```
public function beforeSave($obj, $request, $formData) {
    if ($formData->getPassword()) {
        // password encode
        $obj->setPassword($this->passwordHasher->hashPassword($obj, $formData->
↪getPassword()));
    }
}
```

1.4.4 radio

```
'type' => 'string',
'widget' => 'radio',
'choices' => [
    'Test Value'=>'val1',
    'Another text'=>'text',
],
```

set default value

```
'widget_params' => [
    'data' => 'val1'
],
```

1.4.5 boolean

```
'type' => 'boolean',
'widget' => 'radio',
```

you can set your own values

```
'type' => 'boolean',
'widget' => 'radio',
'yes_val' => 'ok',
'no_val' => 'not',
```

or show only checkbox

```
'type' => 'boolean',
'widget' => 'checkbox',
```

1.4.6 email

```
'type' => 'string',
'widget' => 'email',
```

1.4.7 file

```
'type' => 'string',
'widget' => 'file',
```

1.4.8 select

```
'type' => 'string',
'widget' => 'select',
'choices' => [
    'Test Value'=>'val1',
    'Another text'=>'text',
],
'widget_params' => [
    'placeholder' => '',
    'empty_data'  => null,
],
```

1.4.9 entity

generate select with automatically loaded choices

```
'type' => 'entity',
'label_field' => 'description',
'data_full' => [
    'id', 'name', 'description',
],
'class' => 'App\Entity\UserType',
'widget' => 'select',
'widget_params' => [
    'placeholder' => '',
    'empty_data'  => null,
    'choice_label' => 'description',
],
```

1.4.10 multichoice entity

list of checkboxes

```
'type' => 'string',
'widget' => 'multichoice',
'widget_params' => [
    'expanded' => true,
    'choices' => [
        'Test Value'=>'val1',
        'Another text'=>'text',
    ],
],
```

or multiline select

```
'expanded' => false,
```

1.4.11 selectautocomplete

load options dynamically while typing

you must specify route for loading options

for example, list of user logins:

```
'type' => 'entity',
'class' => 'App\Entity\User',
'widget' => 'selectautocomplete',
'label_field' => 'login',
'route' => 'get_user_autocomplete',
```

and add route:

```
get_user_autocomplete:
    path: /get_user_autocomplete
    controller:␣
↪Ecode\CRUDBundle\Controller\AutocompleteController::getAutocompleteData
    options:
      expose: true
    defaults:
      res: true
```

(continues on next page)

(continued from previous page)

```
      entity: App\Entity\User
      search: [login]
      title: login
      value: id
```

1.4.12 multiselectautocomplete

same as selectautocomplete, but can change many values

```
'widget' => 'multiselectautocomplete',
```

instead of route you can set optional query builder

```
'opt_query_builder' => function (EntityRepository $er) {
    return $er->createQueryBuilder('e')
        ->select('e.id, e.description')
        ->where("e.login IN('userlogin','test')");
},
```

1.4.13 colour

colour selector

```
'type' => 'string',
'widget' => 'colour',
'colours' => json_encode([['#0000ff','#ff0000']]),
```

1.4.14 date

```
'type' => 'date',
'widget' => 'date',
```

1.4.15 daterange

only for filter

```
'type' => 'date',
'widget' => 'daterange',
```

1.4.16 datetime

```
'type' => 'datetime',
'widget' => 'datetime',
```

1.4.17 datetimesec

```
'type' => 'datetimesec',
'widget' => 'datetimesec',
```

1.4.18 time

```
'type' => 'time',
'widget' => 'time',
```

1.4.19 subform entity

subform with dynamically add

data class is required

```
'subform_labels' => [
    'id' => '',
    'humanSurname' => 'Surname',
    'humanName' => 'Name',
    'humanPatronymic' => 'Patronymic',
],
'prototype_data' => new \App\Data\PassHumanData(null, '', '', '', null, null, null,␣
↪false),
'type' => 'entity',
'data_full' => [
    'id', 'humanSurname', 'humanName', 'humanPatronymic',
],
'class' => 'App\Entity\PassHuman',
'widget' => 'subformadd',
'route' => 'get_human_autocomplete_select',
'filter_widget' => 'selectautocomplete',
'filter_field' => 'passCarHuman',
'entry_type' => 'App\Form\CarPassengerTypeSingleAdd',
'att_settings' => [
    'id' => [
        'label' => '',
        'type' => 'number',
        'search' => false,
        'widget' => 'hidden',
        'widget_params' => [
            'attr' => [
                'class' => 'd-none',
            ],
        ],
    ],
    'humanSurname' => [
        'label' => 'Surname',
        'type' => 'string',
        'widget' => 'text',
    ],
    'humanName' => [
        'label' => 'Name',
        'type' => 'string',
```

(continues on next page)

(continued from previous page)

```
        'widget' => 'text',
    ],
    'humanPatronymic' => [
        'label' => 'Patronymic',
        'type' => 'string',
        'widget' => 'text',
    ],
],
'valueformat' => 'list',
'hide_empty_table' => true,
'add_button_title' => 'Add human',
'format_fields' => [
    'humanSurname',
    'humanName',
    'humanPatronymic',
],
'format_params' => [
    'delimiter_field' => ' ',
    'get_label' => false,
],
'show_list'=>true,
```

1.4.20 joinfield

load field from join entity

join_field - key of join entity

in array *join_field_data* set which fields load from join entity

```
'type' => 'joinfield',
'widget' => 'text',
'join_field' => 'user',
'join_field_data' => [
    ['field'=>'name'],
],
```

Index