SerializerBundle Documentation

Release 1.0.0

Tales Santos

Apr 30, 2018

Contents

This bundle integrates the library TSantos Serializer into a Symfony application.

CHAPTER 1

Installation

1.1 Applications that use Symfony Flex

Open a command console, enter your project directory and execute:

```
$ composer require tsantos/serializer-bundle
```

1.2 Applications that don't use Symfony Flex

1.2.1 Step 1: Download the Bundle

Open a command console, enter your project directory and execute the following command to download the latest stable version of this bundle:

```
$ composer require tsantos/serializer-bundle
```

This command requires you to have Composer installed globally, as explained in the **'installation chapter'_** of the Composer documentation.

1.2.2 Step 2: Enable the Bundle

Then, enable the bundle by adding it to the list of registered bundles in the `app/AppKernel.php` file of your project:

```
<?php
// app/AppKernel.php

// ...
class AppKernel extends Kernel
{
```

```
    public function registerBundles()
    {
        $bundles = array(
            // ...

            new TSantos\SerializerBundle\TSantosSerializerBundle(),
        );

        // ...
    }

    // ...
}
```

CHAPTER 2

Usage

This bundle exposes the service `tsantos_serializer` which contains a reference to :class:`TSantos\Serializer\SerializerInterface`.

```
$serializer = $container->get('tsantos_serializer');
```

Note: The format used by the serializer is the one configured in your configuration file. Please go to the page *Configuration Reference* to see information about this configuration.

Note: You can create and register new formats. You can read the dedicated documentation about encoders at *Encoder / Decoder*

2.1 Auto-wiring the Serializer

Instead of fetching the serializer directly from the container, you can define the serializer as a dependency.

```
use TSantos\Serializer\SerializerInterface;
use Symfony\Component\HttpFoundation\JsonResponse;

class DefaultController
{
    private $serializer;

    public function __construct(SerializerInterface $serializer)
    {
        $this->serializer = $serializer;
    }

    public function indexAction(): JsonResponse
```

```
    {
        $post = ...;
        return JsonResponse::fromJsonString($this->serializer->serialize($post));
    }
}
```

CHAPTER 3

Configuration Reference

```yaml
tsantos_serializer:
    debug: "%kernel.debug%"

    # Format of serialization operations (e.g: encoder)
    format: "json"

    # Directory that will store the generated classes
    class_path: "%kernel.cache_dir%/tsantos_serializer/classes"

    # Class generation strategy. Can be one of:
    #
    # `never`: means that the serializer classes will never be generated (best for
↪production environment,
    # but will require to generate the classes in your continous deployment system).
    #
    # `always`: means that every time a new class will be generated (best for
↪debugging).
    #
    # `file_not_found`: means that the serializer classes will be generated only if
↪the class not exists
    # (best for development environment)
    generate_strategy: "file_not_exists"

    mapping:
        # The bundle will try to read the mappings automatically from the directories
↪following the order:
        #
        # 1. `config/serializer` - YAML or XML configuration files
        # 2. `src/Entity` - Annotations
        # 3. `src/Document` - Annotations
        # 4. `src/Model` - Annotations
        auto_configure: true

        # A list of paths and its namespaces like so:
        # - { namespace: "My\\Document", path: "%kernel.project_dir%/config/serializer
↪" }
```

```
        paths: {  }

        cache:

            # Type of cache implementation. Should be one of "file", "psr" or
↪"doctrine".
            type: file

            # Prefix of cache keys. Required only for "doctrine" and "psr" cache␣
↪types.
            prefix: TSantosSerializer

            # Directory that will stored the metadata cache files. Required only for
↪"file" cache type.
            path: "%kernel.cache_dir%/tsantos_serializer/metadata"
```

CHAPTER 4

Event Dispatcher

Sometimes you need to change the state of the data before and/or after some serialization operation. In TSantos Serializer you can accomplish this through *Event Listeners* and *Event Subscribers*.

4.1 Event Listeners

Event listeners are single PHP methods or *callable* that will be called when some event is triggered by the serializer. To register listener in this bundle, first you need to create a class:

```
<?php

    namespace App\EventListener;

    use TSantos\Serializer\EventDispatcher\Event\PreSerializationEvent;
    use TSantos\Serializer\EventDispatcher\EventSubscriberInterface;
    use TSantos\Serializer\EventDispatcher\SerializerEvents;

    class PostSerializerListener
    {
        public function onPreSerialization(PreSerializationEvent $event): void
        {
            $post = $event->getData();
            $post->setTitle('serialized_title');
        }
    }
```

And then register and tag it with *tsantos_serializer.event_listener*:

```
services:
    App\EventListener\PostSerializerListener:
        tags:
            - { name: "tsantos_serializer.event_listener", event:"serializer.pre_
↪serialization", method:"onPreSerialization" }
```

You can even omit the attribute *method* from the tag if your class have the *__invoke* method:

```
<?php

    namespace App\EventListener;

    use TSantos\Serializer\EventDispatcher\Event\PreSerializationEvent;
    use TSantos\Serializer\EventDispatcher\EventSubscriberInterface;
    use TSantos\Serializer\EventDispatcher\SerializerEvents;

    class PostSerializerListener
    {
        public function __invoke(PreSerializationEvent $event): void
        {
            $post = $event->getData();
            $post->setTitle('serialized_title');
        }
    }
```

```
services:
     App\EventListener\PostSerializerListener:
         tags:
             - { name: "tsantos_serializer.event_listener", event:"serializer.pre_
↪serialization" }
```

4.2 Event Subscribers

A better way to define event listener is through event subscribers. All the above examples can be achieved by creating a subscriber class:

```
<?php

namespace App\EventListener;

use TSantos\Serializer\EventDispatcher\Event\PreSerializationEvent;
use TSantos\Serializer\EventDispatcher\EventSubscriberInterface;
use TSantos\Serializer\EventDispatcher\SerializerEvents;

class PostSerializerListener implements EventDispatcherInterface
{
    public static function getListeners(): array
    {
        return [
            SerializerEvents::PRE_SERIALIZATION => 'onPreSerialization',
        ];
    }

    public function onPreSerialization(PreSerializationEvent $event): void
    {
        $post = $event->getData();
        $post->setTitle('serialized_title');
    }
}
```

Thanks to Symfony DIC‘s auto-configuration mechanism, all you need to do is to create your subscriber class and make sure that it implements the **:class:‘EventDispatcher\\EventDispatcherInterface‘** interface.

Note: If you are using a Symfony version prior to 3.3, you'll need to register and tag manually the service.

```
services:
    App\EventListener\PostSerializerListener:
        tags:
            - { name: "tsantos_serializer.event_subscriber"}
```

CHAPTER 5

Normalizer / Denormalizer

(De)normalizers are powerful services that transforms the data without encoding/decoding them. For example, supposing your entity has a *date/time* field. In this case, you don't need to create a custom mapping for **:php-class:'\DateTime'** class to define how to configure such data type. Instead, all you need is to create your custom (de)normalizers:

```
<?php

namespace App\Serializer;

use TSantos\Serializer\Normalizer\DenormalizerInterface;
use TSantos\Serializer\Normalizer\NormalizerInterface;
use TSantos\Serializer\DeserializationContext;
use TSantos\Serializer\SerializationContext;

class DateTimeNormalizer implements NormalizerInterface, DenormalizerInterface
{
    private $format;

    public function __construct(string $format = \DateTime::ATOM)
    {
        $this->format = $format;
    }

    public function normalize($data, SerializationContext $context)
    {
        if (!$data instanceof \DateTimeInterface) {
            throw new InvalidArgumentException('Data should be instance of ' .␣
↪\DateTimeInterface::class);
        }

        return $data->format($this->format);
    }

    public function supportsNormalization($data, SerializationContext $context): bool
    {
```

```
        return $data instanceof \DateTimeInterface;
    }

    public function denormalize($data, DeserializationContext $context)
    {
        return \DateTime::createFromFormat($this->format, $data);
    }

    public function supportsDenormalization(string $type, $data,␣
↪DeserializationContext $context): bool
    {
        return $type === \DateTime::class || $type === \DateTimeInterface::class;
    }
}
```

This bundle automatically recognize services that implements **:class:'Normalizer\\NormalizerInterface'** and **:class:'Normalizer\\DenormalizeInterface'** and tag them with the proper tag name.

See also:

Please, refer to TSantos Serializer Library repository for a detailed documentation about the normalization process and the built-in normalizers.

Note: If you are using a Symfony version prior to 3.3, you'll need to register and tag the normalizer services manually.

```
services:
    App\Serializer\UserNormalizer:
        tags:
            - { name: "tsantos_serializer.normalizer"}
            - { name: "tsantos_serializer.denormalizer"}
```

Tip: You can use normalizers to transform read-only entities and avoid unnecessary over-head due the serialization process:

```
<?php

namespace App\Serializer;

use App\Entity\User;
use App\Repository\UserRepository;
use TSantos\Serializer\Normalizer\DenormalizerInterface;
use TSantos\Serializer\Normalizer\NormalizerInterface;

class UserNormalizer implements NormalizerInterface, DenormalizerInterface
{
    private $userRepository;

    public function __construct(UserRepository $userRepository)
    {
        $this->userRepository = $userRepository;
    }

    public function normalize($data, SerializationContext $context)
    {
        return $data->getId();
```

```
    }

    public function supportsNormalization($data, SerializationContext $context): bool
    {
        return $data instanceof User;
    }

    public function denormalize($data, DeserializationContext $context)
    {
        return $this->userRepository->find($data);
    }

    public function supportsDenormalization(string $type, $data,␣
↪DeserializationContext $context): bool
    {
        return $type === User::class;
    }
}
```

CHAPTER 6

Encoder / Decoder

Encoders are services that transform data from string to array and vice-versa. Although the TSantos Serializer Library currently supports only JSON encoder, you can register new encoders to your application by implementing the **:class:'Encoder\\EncoderInterface'** interface, register the service in the container and tag it like following:

```php
// ./src/Serializer/JsonEncoder
namespace App\Serializer;

use TSantos\Serializer\Encoder\EncoderInterface;

class JsonEncoder implements EncoderInterface
{
    public function encode(array $data): string
    {
        return json_encode($data);
    }

    public function decode(string $content): array
    {
        return json_decode($content, true);
    }

    public function getFormat(): string
    {
        return 'json';
    }
}
```

Now you can tag your encoder and you are done!

```yaml
# ./config/services.yml
services:
    App\Serializer\JsonEncoder:
        tags:
            - { name: "tsantos_serializer.encoder", format: "format" }
```

Note: The attribute *format* is required! This value will be matched against the option *tsantos_serializer.format* to define what encoder will be used in your application.