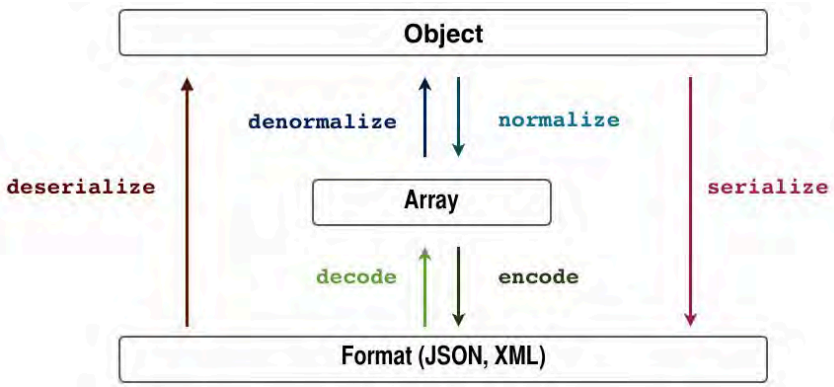
Introducción

El componente Serializer se utiliza para convertir objetos en cadenas de texto con formato específico (JSON, YAML, XML…) y viceversa.



Para instalar el componente con Flex:

> *composer require symfony/serializer*

Uso básico del Serializer

Para utilizar el Serializador, hay que configurarlo con los normalizadores y codificadores que queramos que soporte:

```
use Symfony\Component\Serializer\Serializer;
use Symfony\Component\Serializer\Encoder\XmlEncoder;
use Symfony\Component\Serializer\Encoder\JsonEncoder;
use Symfony\Component\Serializer\Normalizer\ObjectNormaliz
er;

$encoders = array(new XmlEncoder(), new JsonEncoder());
$normalizers = array(new ObjectNormalizer());

$serializer = new Serializer($normalizers, $encoders);
```

Serializar un objeto

```
$person = new App\Entity\Producto();
$person->setNombre('nombre del producto');
$person->setPrecio(99);

$jsonContent = $serializer->serialize($person, 'json');

// $jsonContent contiene {"nombre":"nombre del producto","
precio":99}
```

Desserializar un objeto

```
use App\Model\Person;

$data = <<<EOF
<person>
    <name>foo</name>
    <age>99</age>
    <sportsperson>false</sportsperson>
</person>
EOF;

$person = $serializer->deserialize($data, Person::class, '
xml');
```

Si hay atributos que no son de la clase, el comportamiento por defecto es ignorarlos. Pero se puede configurar el método deserialize para que lance un error en caso de encontrar atributos desconocidos:

```
$data = <<<EOF
<person>
    <name>foo</name>
    <age>99</age>
    <city>Paris</city>
</person>
EOF;
```

```
// this will throw a Symfony\Component\Serializer\Exceptio
n\ExtraAttributesException
// because "city" is not an attribute of the Person class
$person = $serializer->deserialize($data, 'Acme\Person', '
xml', array(
    'allow_extra_attributes' => false,
));
```

Se puede utilizar el serializar para actualizar un objeto existente:

```
$person = new Person();
$person->setName('bar');
$person->setAge(99);
$person->setSportsperson(true);

$data = <<<EOF
<person>
    <name>foo</name>
    <age>69</age>
</person>
EOF;

$serializer->deserialize($data, Person::class, 'xml', arra
y('object_to_populate' => $person));
// $person = App\Model\Person(name: 'foo', age: '69', spor
tsperson: true)
```

Si estamos des-serializando un array de objetos, entonces se le indica al método deserialize con []

```
$persons = $serializer->deserialize($data, 'Acme\Person[]'
, 'json');
```

Handling Circular References

Pongamos un ejemplo de una referencia circular entre 2 objetos:

```
class Organization
{
    private $name;
    private $members;

    public function setName($name)
    {
        $this->name = $name;
    }

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

    public function setMembers(array $members)
    {
        $this->members = $members;
    }

    public function getMembers()
    {
        return $this->members;
```

```
    }
}

class Member
{
    private $name;
    private $organization;

    public function setName($name)
    {
        $this->name = $name;
    }

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

    public function setOrganization(Organization $organization)
    {
        $this->organization = $organization;
    }

    public function getOrganization()
    {
        return $this->organization;
    }
```

```
}
```

Para evitar un bucle infinito, los normalizadores, GetSetMethodNormalizer y ObjectNormalizer lanzan una Excepción CircularReferenceException cuando se encuentran con referencias circulares.

```
$member = new Member();
$member->setName('Kévin');

$organization = new Organization();
$organization->setName('Les-Tilleuls.coop');
$organization->setMembers(array($member));

$member->setOrganization($organization);

echo $serializer->serialize($organization, 'json'); // Lan
za una CircularReferenceException
```

El método **setCircularReferenceLimit()** de estos normalizadores establecen el máximo número de veces que se va a serializar un mismo objeto antes de considerarlo una referencia circular. Por defecto este valor es de 1.

En vez de lanzar una excepción, se pueden manejar con un *callable* para ejecutar el código que queramos.

En el siguiente objeto, se “cambia” el objeto que provoca la referencia circular por un string que lo representa, de forma que se corta el bucle infinito.

```
$encoder = new JsonEncoder();
$normalizer = new ObjectNormalizer();

$normalizer->setCircularReferenceHandler(function ($object
) {
    return $object->getName();
});

$serializer = new Serializer(array($normalizer), array($en
coder));
```

Handling Serialization Depth

El componente Serializer es capaz de detectar y limitar la profundidad de la serialización.

Vamos a verlo con un ejemplo:

```
namespace Acme;

class MyObj
{
    public $foo;

    /**
     * @var self
     */
    public $child;
}

$level1 = new MyObj();
$level1->foo = 'level1';

$level2 = new MyObj();
$level2->foo = 'level2';
$level1->child = $level2;

$level3 = new MyObj();
```

```
$level3->foo = 'level3';
$level2->child = $level3;
```

El serializador se puede configurar para establecer una profundidad máxima para una propiedad concreta.

```
use Symfony\Component\Serializer\Annotation\MaxDepth;

namespace Acme;

class MyObj
{
    /**
     * @MaxDepth(2)
     */
    public $child;

    // ...
}
```

Para que el serializador tenga en cuenta la limitación de profundidad, hay que indicárselo con la clave **enable_max_depth**.

```
$result = $serializer->normalize($level1, null, array('ena
ble_max_depth' => true));
/*
$result = array(
```

```
    'foo' => 'level1',
    'child' => array(
            'foo' => 'level2',
            'child' => array(
                    'child' => null,
                ),
        ),
);
*/
```

Normalizadores

En Symfony tenemos varios normalizadores disponibles:

- **ObjectNormalizer**

Este normalizador es capaz de acceder a las propiedades directamente y también a través de getters, setters, hassers, adders y removers.

Soporta la llamada al constructor durante la denormalización.

El renombrado de método a propiedad es como se ve en este ejemplo:

getFirstName() -> firstName

Es el normalizador más potente de los que vienen con Symfony.

- **GetSetMethodNormalizer**

Este normalizador lee el contenido de la clase llamando a los *getters* (métodos públicos que empiecen por *get*) y desnormaliza llamando al constructor de la clase y a los *setters* (métodos públicos que empiecen por *set*).

El renombrado de método a propiedad es como se ve en este ejemplo:

getFirstName() -> firstName

- **PropertyNormalizer**

Este normalizador lee y escribe todas las propiedades públicas, privadas y protected de la propia clase y de sus clases padre.

- **JsonSerializableNormalizer**

Este normalizador trabaja con clases que implementan JsonSerializable.

- **DateTimeNormalizer**

Convierte objetos DateTimeInterface (por ejemplo: DateTime y DateTimeImmutable) en strings. Por defecto utiliza el formato RFC3339.

- **DataUriNormalizer**

Convierte objetos SplFileInfo en *data URI string* (data:…) permitiendo embeber ficheros en datos serializados.

- **DateIntervalNormalizer**

Convierte objetos DateInterval en strings. Por defecto utiliza el formato P%yY%mM%dDT%hH%iM%sS.

Encoders

Los codificadores convierten arrays en XML, JSON… y viceversa.

Todos implementan el interfaz **EncoderInterface** para codificar (array a formato) y **DecoderInterface** para decodificar (formato a array).

Built-in Encoders

Symfony viene con unos cuantos Encoders:

- JsonEncoder
- XmlEncoder
- YamlEncoder
- CsvEncoder

```
use Symfony\Component\Serializer\Serializer;
use Symfony\Component\Serializer\Encoder\XmlEncoder;
use Symfony\Component\Serializer\Encoder\JsonEncoder;

$encoders = array(new XmlEncoder(), new JsonEncoder());
$serializer = new Serializer(array(), $encoders);
```

Grupos de atributos

Muchas veces queremos serializar diferentes juegos de atributos de nuestras entidades. Por ejemplo, en una API REST, queremos que los datos que se devuelvan de un producto sean distintos según la pantalla en la que te encuentras o según el rol del usuario, de forma que a un admin la API le devolverá más información sobre los objetos que a un usuario normal.

Esto se puede gestionar creando grupos de atributos:

Veamos un ejemplo en el que tenemos el siguiente objeto:

```
use Symfony\Component\Serializer\Annotation\Groups;

class Product
{

    public $name;

    public $price;

    public $itemsSold;

    public $commission;
}
```

Podemos definir los grupos con anotaciones con XML o con YAML.

```
use Symfony\Component\Serializer\Annotation\Groups;

class Product
{
    /**
     * @Groups({"admins", "affiliates", "users"})
     */
    public $name;

    /**
     * @Groups({"admins", "affiliates", "users"})
     */
    public $price;

    /**
     * @Groups({"admins"})
     */
    public $itemsSold;

    /**
     * @Groups({"admins", "affiliates"})
     */
    public $commission;
}
```

Una vez definidos los grupos, podemos elegir uno o más a la hora de serializar:

```
use Symfony\Component\Serializer\Serializer;
use Symfony\Component\Serializer\Normalizer\GetSetMethodNo
rmalizer;
use Symfony\Component\Serializer\Mapping\Loader\Annotation
Loader;
use Symfony\Component\Serializer\Mapping\Factory\ClassMeta
dataFactory;
use Doctrine\Common\Annotations\AnnotationReader;

$product = new Product();
$product->itemsSold = 20;
$product->commission = 7.5;
$product->price = 19.99;

$classMetadataFactory = new ClassMetadataFactory(new Annot
ationLoader(new AnnotationReader()));
$normalizer = new PropertyNormalizer($classMetadataFactory
);
$serializer = new Serializer([$normalizer]);

$data = $serializer->normalize($product, null, ['groups' =
> ['admins']]);

$data = $serializer->normalize($product, null, ['groups' =
```

```
> ['affiliates']]);

$data = $serializer->normalize($product, null, ['groups' =
> ['users']]);

$data = $serializer->normalize($product, null, ['groups' =
> ['affiliates', 'users']]);
```

Si se elige más de un grupo, el resultado será la fusión de las propiedades de ambos grupos.

Lo único extraño en el código anterior es que primero debemos inicializar la clase ClassMetadataFactory de alguna de las siguientes maneras:

```
use Symfony\Component\Serializer\Mapping\Factory\ClassMeta
dataFactory;
// For annotations
use Doctrine\Common\Annotations\AnnotationReader;
use Symfony\Component\Serializer\Mapping\Loader\Annotation
Loader;
// For XML
// use Symfony\Component\Serializer\Mapping\Loader\XmlFile
Loader;
// For YAML
// use Symfony\Component\Serializer\Mapping\Loader\YamlFil
eLoader;
```

```
$classMetadataFactory = new ClassMetadataFactory(new Annot
ationLoader(new AnnotationReader()));
// For XML
// $classMetadataFactory = new ClassMetadataFactory(new Xm
lFileLoader('/path/to/your/definition.xml'));
// For YAML
// $classMetadataFactory = new ClassMetadataFactory(new Ya
mlFileLoader('/path/to/your/definition.yaml'));
```

Por lo demás, todo muy sencillo e intuitivo.

NOTA
Para utilizar el loader *annotation*, debemos tener instalados los packages *doctrine/annotations* y *doctrine/cache*.

Seleccionar atributos específicos

También es posible serializar atributos concretos sin haberlos configurado previamente.

```
use Symfony\Component\Serializer\Serializer;
use Symfony\Component\Serializer\Normalizer\ObjectNormaliz
er;

class User
{
```

```
    public $familyName;
    public $givenName;
    public $company;
}

class Company
{
    public $name;
    public $address;
}

$company = new Company();
$company->name = 'Les-Tilleuls.coop';
$company->address = 'Lille, France';

$user = new User();
$user->familyName = 'Dunglas';
$user->givenName = 'Kévin';
$user->company = $company;

$serializer = new Serializer(array(new ObjectNormalizer())
);

$data = $serializer->normalize($user, null, array('attribu
tes' => array('familyName', 'company' => ['name'])));
// $data = array('familyName' => 'Dunglas', 'company' => a
rray('name' => 'Les-Tilleuls.coop'));
```

Si además se indica algún grupo, solamente se pueden indicar atributos que pertenezcan a dicho grupo.

El mismo mecanismo aplica en la desserialización.

Convertir nombres de propiedades al serializar y deserializar

Si necesitamos renombrar propiedades, symfony nos proporciona una manera para hacerlo: El sistema Name Converter.

Pongamos que tenemos el siguiente objeto:

```
class Company
{
    public $name;
    public $address;
}
```

Y queremos prefijar todos los atributos con el prefijo *org_*:

{“org_name”: “Acme Inc.”, “org_address”: “123 Main Street, Big City”}

Para este propósito se debe utilizar un **Name Converter** personalizado.

Un Name Converter es una clase que implementa el interfaz **NameConverterInterface**.

```
use Symfony\Component\Serializer\NameConverter\NameConvert
```

```
erInterface;

class OrgPrefixNameConverter implements NameConverterInter
face
{
    public function normalize($propertyName)
    {
        return 'org_'.$propertyName;
    }

    public function denormalize($propertyName)
    {
        return 'org_' === substr($propertyName, 0, 4) ? su
bstr($propertyName, 4) : $propertyName;
    }
}
```

Los *name converter* se pasan como segundo parámetro de los normalizers:

```
use Symfony\Component\Serializer\Encoder\JsonEncoder;
use Symfony\Component\Serializer\Normalizer\ObjectNormaliz
er;
use Symfony\Component\Serializer\Serializer;

$nameConverter = new OrgPrefixNameConverter();
$normalizer = new ObjectNormalizer(null, $nameConverter);
```

```
$serializer = new Serializer(array($normalizer), array(new
 JsonEncoder()));

$company = new Company();
$company->name = 'Acme Inc.';
$company->address = '123 Main Street, Big City';

$json = $serializer->serialize($company, 'json');
// {"org_name": "Acme Inc.", "org_address": "123 Main Stre
et, Big City"}
$companyCopy = $serializer->deserialize($json, Company::cl
ass, 'json');
```

El name converter CamelCaseToSnakeCaseNameConv

Symfony viene con un name converter denominado **CamelCaseToSnakeCaseNameConverter** que convierte los nombres de tipo CamelCase a SnakeCase.

```
use Symfony\Component\Serializer\NameConverter\CamelCaseTo
SnakeCaseNameConverter;
use Symfony\Component\Serializer\Normalizer\ObjectNormaliz
er;

$normalizer = new ObjectNormalizer(null, new CamelCaseToSn
```

```
akeCaseNameConverter());

class Person
{
    private $firstName;

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

    public function getFirstName()
    {
        return $this->firstName;
    }
}

$kevin = new Person('Kévin');
$normalizer->normalize($kevin);
// ['first_name' => 'Kévin'];

$anne = $normalizer->denormalize(array('first_name' => 'An
ne'), 'Person');
// Person object with firstName: 'Anne'
```