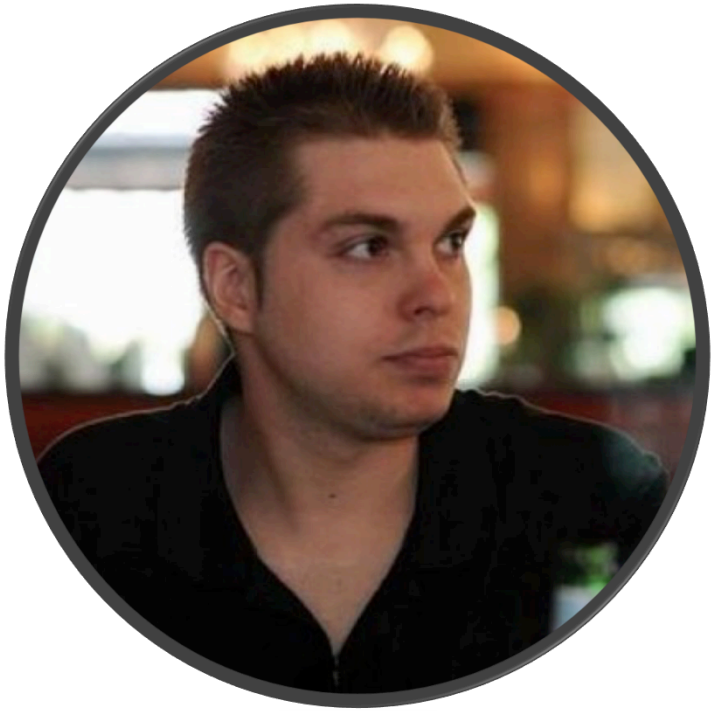
Data Serialization with Symfony & Drupal

Symfony

SensioLabs

Hugo Hamon



Head of training at SensioLabs

Book author
Speaker at Conferences
Symfony contributor
Travel lover
@hhamon

« Serialization is the process of **translating data structures or object state into a format that can be stored and reconstructed** later in the same or another computer environment. »

-- Wikipedia.

Examples:

HTTP Messages

XML

SOAP

JSON

YAML

CSV...

Most Common Usages:

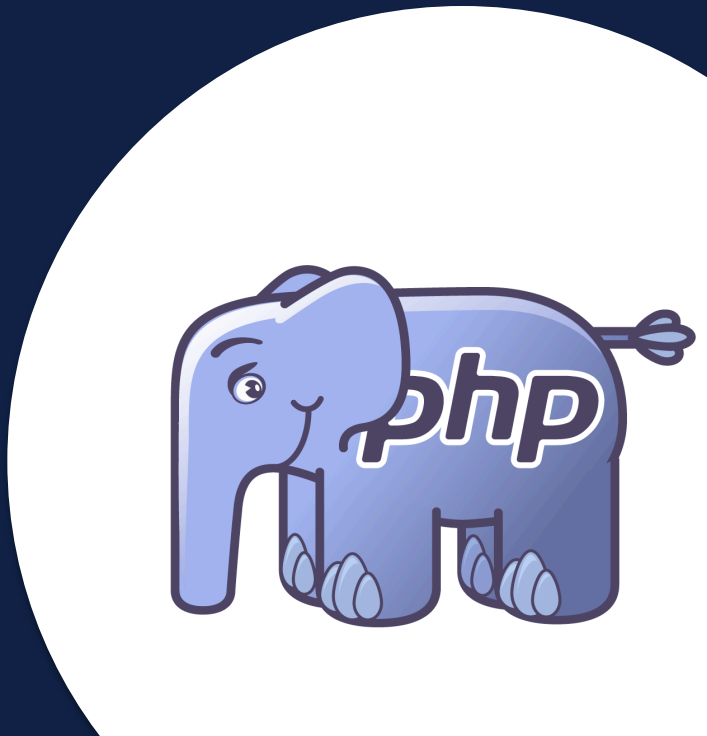
Storage in a file or a database

REST APIs

SOAP Web Services

Distributing objects (Java)

Data Serialization with PHP



Serializing Data Structures with PHP

```
serialize(18);
serialize(12.50);
serialize(null);
serialize(true);
serialize(false);
serialize('John Smith');
serialize([ 'a', 'b' ]);
serialize(new stdClass());
```

```
i:18;
d:12.5;
N;
b:1;
b:0;
s:10:"John Smith";
a:3:{i:0;s:1:"a";i:1;s:1:"b";}
O:8:"stdClass":0:{}
```

```
VALUE  T:VALUE;

   18  i:18;
 12.5  d:12.5;
 true  b:1;
false  b:0;
 null  N:;
```

```
VALUE  s:1:"VALUE";

    a  s:1:"a";
   it  s:2:"it";
peach  s:5:"peach";
John Smith  s:10:"John Smith";
```

```
$data = [ 'a', 'b' ]
```

```
a:2:{i:0;s:1:"a";i:1;s:1:"b";}
```

```
a = Array
2 = Array length (# of elements)
i = Index
n = Index name
s = String
1 = String length (# of chars)
"x" = Value
```

```
$data = new stdClass();

O:8:"stdClass":0:{}


O = Object
8 = Class name length
"stdClass" = Class name
0 = Object size (# of properties)
```

```php
$a = unserialize('i:18;');
$b = unserialize('d:12.5;');
$c = unserialize('b:1;');
$d = unserialize('b:0;');
$e = unserialize('N:;');
$f = unserialize('s:10:"John Smith";');
$g = unserialize('a:2:{i:0;s:1:"a";i:1;s:1:"b";}');
$h = unserialize('O:8:"stdClass":0:{}');
```

Object (de)Serialization Handling

__sleep()

__wakeup()

```php
namespace Database;

class Connection
{
    private $link;
    private $dsn;
    private $user;
    private $pwd;

    public function __construct($dsn, $username, $password)
    {
        $this->dsn = $dsn;
        $this->user = $username;
        $this->pwd = $password;
    }

    private function connect()
    {
        if (!$this->link instanceof \PDO) {
            $this->link = new \PDO($this->dsn, $this->user, $this->pwd);
        }
    }
}
```

```php
class Connection
{
    // …

    public function __sleep()
    {
        return [ 'dsn', 'user', 'pwd' ];
    }

    public function __wakeup()
    {
        $this->connect();
    }
}
```

```php
use Database\Connection;

$dsn = 'mysql:host=localhost;dbname=test';
$usr = 'root';
$pwd = '';

$db = new Connection($dsn, $usr, $pwd);
$db->query('SELECT ...');

$serialized = serialize($db);

$db = unserialize($serialized);
$db->query('SELECT ...');
```

Serializable Interface

```php
class Connection implements \Serializable
{
    public function serialize()
    {
        return serialize([
            'dsn' => $this->dsn,
            'user' => $this->user,
            'password' => $this->pwd,
        ]);
    }
}
```

```php
class Connection implements \Serializable
{
    public function unserialize($data)
    {
        $data = unserialize($data);

        $this->dsn = $data['dsn'];
        $this->user = $data['user'];
        $this->pwd = $data['password'];

        $this->connect();
    }
}
```

What about JSON as serialization format?

json_encode()
json_decode()

JsonSerializable

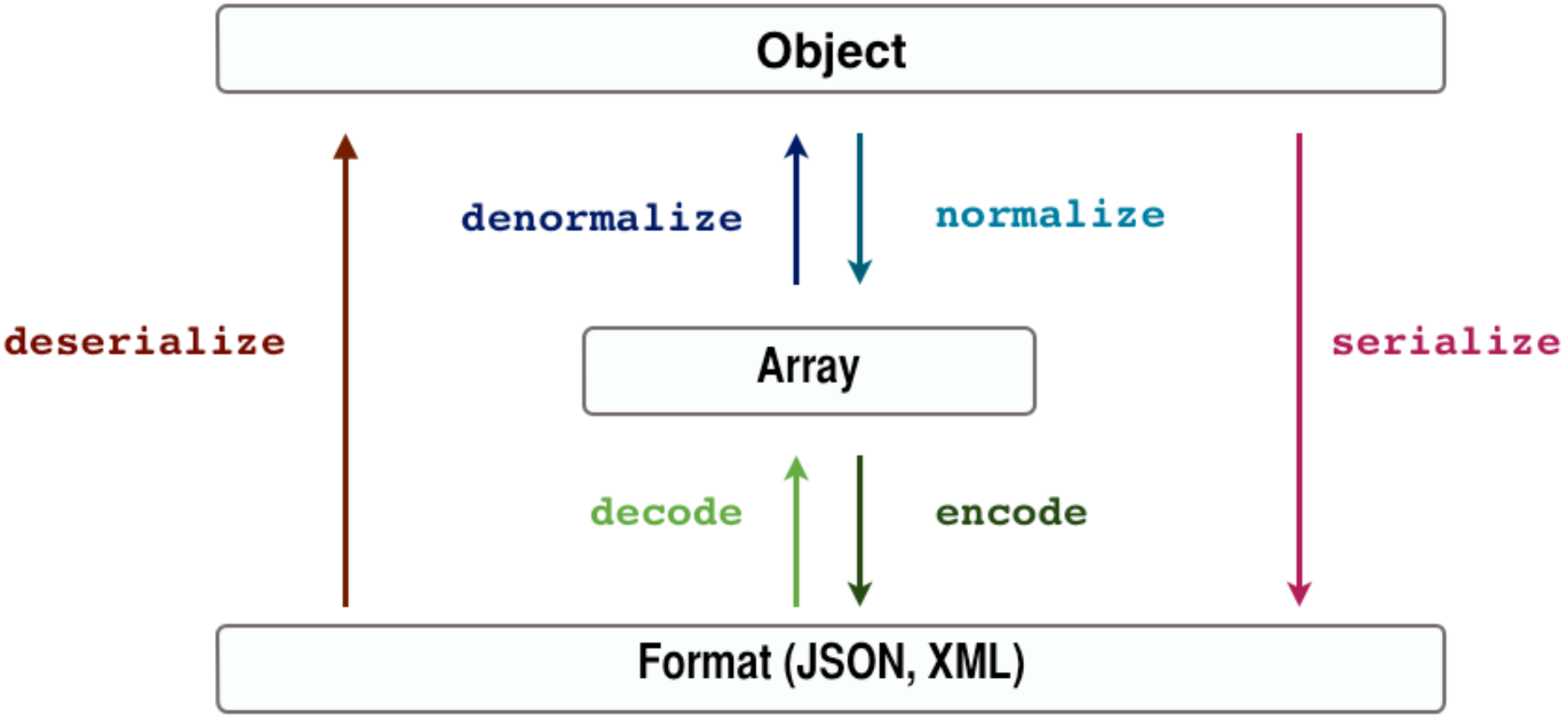
```php
class ArrayValue implements JsonSerializable
{
    public function __construct(array $array)
    {
        $this->array = $array;
    }

    public function jsonSerialize()
    {
        return $this->array;
    }
}

json_encode(new ArrayValue([1, 2, 3]));
```

Serialization is a very complex task…

The Symfony Serializer

«The Serializer component is meant to be used to turn objects into a specific format (XML, JSON, YAML, ...) and the other way around. »

-- Symfony.com



http://symfony.com/doc/current/components/serializer.html

The Serializer Public API

```
class Serializer
{
    final function serialize($data, $format, array $context = [])
    final function deserialize($data, $type, $format, array $context = []);

    function normalize($data, $format = null, array $context = [])
    function denormalize($data, $type, $format = null, array $context = []);

    function supportsNormalization($data, $format = null);
    function supportsDenormalization($data, $type, $format = null)

    final function encode($data, $format, array $context = []);
    final function decode($data, $format, array $context = []);

    function supportsEncoding($format);
    function supportsDecoding($format);
}
```

```php
use Symfony\Component\Serializer\Serializer;
use Symfony\Component\Serializer\Normalizer;
use Symfony\Component\Serializer\Encoder;

// Setup the normalizers
$normalizers[] = new Normalizer\PropertyNormalizer();

// Setup the encoders
$encoders[] = new Encoder\JsonEncoder();
$encoders[] = new Encoder\XmlEncoder();

// Setup the serializer
$serializer = new Serializer($normalizers, $encoders);

// Use the serializer
$serializer->serialize($object, 'json');
$serializer->deserialize($data, 'Acme\User','json');
```

Normalizers / Denormalizers

Name	Goal
`Property`	Normalizes public / private properties to an associative array.
`GetSetMethod`	Normalizes properties by calling getter, isser & setter methods.
`Object`	Normalizes objects with the PropertyAccess component.
`Custom`	Normalizes an object by delegating serialization to it.

Encoders / Decoders

Name	Goal
`JsonEncoder`	Encodes & decodes an array from/to JSON.
`XmlEncoder`	Encodes & decodes an array from/to XML.
`ChainEncoder`	Chains multiple encoders.
`ChainDecoder`	Chain multiple decoders.

Serializer
Basic Usages

```
class Movie
{
    private $id;
    private $title;
    private $slug;
    private $description;
    private $duration;
    private $releaseDate;
    private $storageKey;
}
```

Serializing an Object

```
$movie = new Movie();
$movie->setTitle('Seven');
$movie->setSlug('seven');
$movie->setDescription('A brilliant…');
$movie->setDuration(130);
$movie->setReleaseDate('1996-01-31');
```

```
$data = $serializer->serialize(
    $movie,
    'json'
);

$movie = $serializer->deserialize(
    $data,
    'Movie',
    'json'
);
```

Properties Serialization

JSON Serialization


```
{
  "id":null,
  "title":"Seven",
  "slug":"seven",
  "description":"A … thriller!",
  "duration":130,
  "releaseDate":"1996-01-31",
  "storageKey":null
}
```

XML Serialization

```
<?xml version="1.0"?>
<response>
  <id/>
  <title>Seven</title>
  <slug>seven</slug>
  <description>A … thriller!</description>
  <duration>130</duration>
  <releaseDate>1996-01-31</releaseDate>
  <storageKey/>
</response>
```

String Deserialization

JSON Deserialization

```
$data = <<<DATA
{
  "id":null,
  "title":"Seven",
  "slug":"seven",
  "description":"A brilliant thriller!",
  "duration":130,
  "releaseDate":"1996-01-31",
  "storageKey":null
}
DATA;

$movie = $serializer->deserialize($data, 'Movie', 'json');

print_r($movie);
```

XML Deserialization

```
$data = <<<DATA
<?xml version="1.0"?>
<response>
  <id/>
  <title>Seven</title>
  <slug>seven</slug>
  <description>A … thriller!</description>
  <duration>130</duration>
  <releaseDate>1996-01-31</releaseDate>
  <storageKey/>
</response>
DATA;

$movie = $serializer->deserialize($data, 'Movie', 'xml');

print_r($movie);
```

String Deserialization

```
Movie Object
(
    [id:Movie:private] =>
    [title:Movie:private] => Seven
    [slug:Movie:private] => seven
    [description:Movie:private] => A … thriller!
    [duration:Movie:private] => 130
    [releaseDate:Movie:private] => 1996-01-31
    [storageKey:Movie:private] =>
)
```

Constructor Initialization

```
class Movie
{
    // ...

    function __construct($id = null, $title = null, $slug = null)
    {
        $this->id = $id;
        $this->title = $title;
        $this->slug = $slug;
    }
}
```

Constructor arguments must match properties names.

Going Further with the Serializer

Getter, Hasser & Isser Methods Normalizer

```php
// Setup the normalizers
$normalizers[] = new Normalizer\ObjectNormalizer();
$normalizers[] = new Normalizer\GetSetMethodNormalizer();
$normalizers[] = new Normalizer\PropertyNormalizer();

// Setup the encoders
$encoders[] = new Encoder\JsonEncoder();
$encoders[] = new Encoder\XmlEncoder();

// Setup the serializer
$serializer = new Serializer($normalizers, $encoders);

// Use the serializer
$serializer->serialize($object, 'json');
$serializer->deserialize($data, 'Acme\User','json');
```

The object normalizer can invoke « hasser » methods.

```php
class Movie
{
    public function getId()
    {
        return $this->id;
    }

    public function getTitle()
    {
        return $this->title;
    }

    public function hasGenre()
    {
        return false;
    }

    // ...

    public function isReleased()
    {
        return new \DateTime($this->releaseDate) <= new \DateTime();
    }
}
```

The normalizer invokes getter & isser methods.

JSON Serialization

```
{
  "id":null,
  "title":"Seven",
  "slug":"seven",
  "description":"A … thriller!",
  "duration":130,
  "releaseDate":"1996-01-31",
  "storageKey":null,
  "genre":false,
  "released":true,
}
```

XML Serialization

```
<?xml version="1.0"?>
<response>
  <id/>
  <title>Seven</title>
  <slug>seven</slug>
  <description>A … thriller!</description>
  <duration>130</duration>
  <releaseDate>1996-01-31</releaseDate>
  <storageKey/>
  <genre>0</genre>
  <released>1</released>
</response>
```

Ignoring Attributes

```php
$normalizer = new GetSetMethodNormalizer();
$normalizer->setIgnoredAttributes([ 'storageKey' ]);
```

```xml
<?xml version="1.0"?>
<response>
  <id/>
  <title>Seven</title>
  <slug>seven</slug>
  <description>A … thriller!</description>
  <duration>130</duration>
  <releaseDate>1996-01-31</releaseDate>
  <released>1</released>
</response>
```

Converting properties names to underscore case.

```php
$converter = new CamelCaseToSnakeCaseNameConverter();
$normalizer = new GetSetMethodNormalizer(null, $converter);
```

```xml
<?xml version="1.0"?>
<response>
  <id/>
  <title>Seven</title>
  <slug>seven</slug>
  <description>A … thriller!</description>
  <duration>130</duration>
  <release_date>1996-01-31</release_date>
  <released>1</released>
</response>
```

Customizing all serialized properties names.

```php
class PrefixNameConverter implements NameConverterInterface
{
    private $prefix;

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

    public function normalize($propertyName)
    {
        return $this->prefix.'_'.$propertyName;
    }

    public function denormalize($propertyName)
    {
        if ($this->prefix.'_' === substr($propertyName, 0, count($this->prefix))) {
            return substr($propertyName, count($this->prefix));
        }

        return $propertyName;
    }
}
```

The NameConverterInterface has been introduced in 2.7.

```php
$converter = new PrefixNameConverter('movie');
$normalizer = new GetSetMethodNormalizer(null, $converter);
```

```xml
<?xml version="1.0"?>
<response>
  <movie_id/>
  <movie_title>Seven</movie_title>
  <movie_slug>seven</movie_slug>
  <movie_description>A … thriller!</movie_description>
  <movie_duration>130</movie_duration>
  <movie_release_date>1996-01-31</movie_release_date>
  <movie_released>1</movie_released>
</response>
```

Changing the XML root name.

```php
$serializer->serialize($movie, 'xml', [
    'xml_root_node_name' => 'movie',
]);
```

```xml
<?xml version="1.0"?>
<movie>
  <id/>
  <title>Seven</title>
  ...
</movie>
```

Deserializing into an existing object.

```
$data = <<<DATA
<?xml version="1.0"?>
<movie>
  <duration>130</duration>
  <releaseDate>1996-01-31</releaseDate>
</movie>
DATA;

$movie1 = new Movie(1234, 'Seven', 'seven');

$movie2 = $serializer->deserialize($data, 'Movie', 'xml', [
    'xml_root_node_name' => 'movie',
    'object_to_populate' => $movie1,
]);
```

```
Movie Object
(
    [id:Movie:private] => 1234
    [title:Movie:private] => Seven
    [slug:Movie:private] => seven
    [description:Movie:private] =>
    [duration:Movie:private] => 130
    [releaseDate:Movie:private] => 1996-01-31
    [storageKey:Movie:private] =>
    [genre:Movie:private] =>
)
```

The « description » property remains empty while « duration » and « releaseDate » properties are set.

Serializer Advanced Features

Serializing More Complex Object Graphs.

```php
class Movie
{
    /** @var Genre */
    private $genre;

    /** @var Directors[] */
    private $directors;

    /**
     * Each role keeps a reference to that Movie object
     * and a reference to an Actor object playing that
     * role in the movie.
     *
     * @var Role[]
     */
    private $roles;
}
```

One to One Unidirectional Relationship

```php
$genre = new Genre(42, 'Thriller', 'thriller');

$movie = new Movie(1234, 'Seven', 'seven');
$movie->setGenre($genre);
$movie->setStorageKey('movie-42-1234');
$movie->setDuration(130);
$movie->setDescription('A brilliant thriller!');
$movie->setReleaseDate('1996-01-31');

echo $serializer->serialize($movie, 'xml', [
    'xml_root_node_name' => 'movie',
]);
```

```
<?xml version="1.0"?>
<movie>
  <genre>
    <id>42</id>
    <slug>thriller</slug>
    <title>Thriller</title>
  </genre>
  <id>1234</id>
  <title>Seven</title>
  <duration>130</duration>
  <released>1</released>
  <slug>seven</slug>
  <description>A brilliant thriller!</description>
  <release_date>1996-01-31</release_date>
</movie>
```

```
{
  "genre":{
    "id":42,
    "slug":"thriller",
    "title":"Thriller"
  },
  "id":1234,
  "title":"Seven",
  "duration":130,
  "released":true,
  "slug":"seven",
  "description":"A brilliant thriller!",
  "release_date":"1996-01-31"
}
```

One to Many Unidirectional Relationship

```php
$fincher = new Director();
$fincher->setId(973463);
$fincher->setName('David Fincher');
$fincher->setBirthday('1962-05-10');

$kopelson = new Director();
$kopelson->setId(783237);
$kopelson->setName('Arnold Kopelson');
$kopelson->setBirthday('1935-02-14');

$movie = new Movie(1234, 'Seven', 'seven');
$movie->addDirector($fincher);
$movie->addDirector($kopelson);
```

```
<?xml version="1.0"?>
<movie>
  <!-- ... -->
  <directors>
    <id>973463</id>
    <name>David Fincher</name>
    <birthday>1962-05-10</birthday>
    <deathday/>
  </directors>
  <directors>
    <id>783237</id>
    <name>Arnold Kopelson</name>
    <birthday>1935-02-14</birthday>
    <deathday/>
  </directors>
</movie>
```

```
{
  "genre":{
    "id":42,
    "slug":"thriller",
    "title":"Thriller"
  },
  "id":1234,
  "title":"Seven",
  "duration":130,
  "released":true,
  "slug":"seven",
  "description":"A brilliant thriller!",
  "release_date":"1996-01-31",
  "directors":[
    {
      "id":973463,
      "name":"David Fincher",
      "birthday":"1962-05-10",
      "deathday":null
    },
    {
      "id":783237,
      "name":"Arnold Kopelson",
      "birthday":"1935-02-14",
      "deathday":null
    }
  ]
}
```

Many to Many Bidirectional Relationship

```php
class Role
{
    private $id;
    private $character;
    private $movie;
    private $actor;

    function __construct($id, Movie $movie, Actor $actor, $character)
    {
        $this->id = $id;
        $this->movie = $movie;
        $this->actor = $actor;
        $this->character = $character;
    }
}
```

The « Role » instance keeps a reference to the « Movie » that also keeps references to « roles » played by actors.

```php
$movie = new Movie(1234, 'Seven', 'seven');

// ...

$pitt = new Actor();
$pitt->setId(328470);
$pitt->setName('Brad Pitt');
$pitt->setBirthday('1963-12-18');

$freeman = new Actor();
$freeman->setId(329443);
$freeman->setName('Morgan Freeman');
$freeman->setBirthday('1937-06-01');

$mills = new Role(233, $movie, $pitt, 'David Mills');
$sommerset = new Role(328, $movie, $freeman, 'William Sommerset');

$movie->addRole($mills);
$movie->addRole($sommerset);

$serializer->serialize($movie, 'json');
```

PHP Fatal error: Uncaught exception 'Symfony\Component\Serializer\Exception\CircularReferenceException' with message 'A circular reference has been detected (configured limit: 1).' in /Volumes/Development/Sites/Serializer/vendor/symfony/serializer/Normalizer/AbstractNormalizer.php:221

Handling Circular References

Handling Circular References

```php
$normalizer = new ObjectNormalizer(null, $converter);
$normalizer->setIgnoredAttributes([ 'storageKey' ]);

// Return the object unique identifier instead of the
// instance to stop a potential infinite serialization loop.

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

Circular references support has been introduced in Symfony 2.6.

```
{
  ...
  "roles":[
    {
      "actor":{
        "id":328470,
        "name":"Brad Pitt",
        "birthday":"1963-12-18",
        "deathday":null
      },
      "character":"David Mills",
      "id":233163,
      "movie":1234
    },
   ...
  ]
}
```

Using Callback Normalizers.

Actors, Directors and Movies now stores date representations as « DateTime » objects. These instance must be serialized too.

```php
$movie = new Movie(1234, 'Seven', 'seven');
$movie->setReleaseDate(new \DateTime('1996-01-31'));

$pitt = new Actor();
$pitt->setBirthday(new \DateTime('1963-12-18'));

$fincher = new Director();
$fincher->setBirthday(new \DateTime('1962-05-10'));

$serializer->serialize($movie, 'json');
```

Without custom serializer to handle « DateTime » instance, the Serializer serializes any date object as follows:

```
<release_date>
  <last_errors>
    <warning_count>0</warning_count>
    <warnings/>
    <error_count>0</error_count>
    <errors/>
  </last_errors>
  <timezone>
    <name>Europe/Paris</name>
    <location>
      <country_code>FR</country_code>
      <latitude>48.86666</latitude>
      <longitude>2.33333</longitude>
      <comments></comments>
     </location>
   </timezone>
   <offset>3600</offset>
   <timestamp>823042800</timestamp>
</release_date>
```

The built-in normalizers allow to set PHP callbacks to handle custom serialization steps.

```php
$normalizer = new Normalizer\ObjectNormalizer(...);

$callback = function ($dateTime) {
    return $dateTime instanceof \DateTime
        ? $dateTime->format(\DateTime::ISO8601)
        : '';
};

$normalizer->setCallbacks([
    'releaseDate' => $callback,
    'birthday' => $callback,
    'deathday' => $callback,
]);
```

```
<?xml version="1.0"?>
<movie>
  <!-- ... -->
  <release_date>1996-01-31T00:00:00+0100</release_date>
  <directors>
    <id>973463</id>
    <name>David Fincher</name>
    <birthday>1962-05-10T00:00:00+0100</birthday>
    <deathday/>
  </directors>
  <directors>
    <id>783237</id>
    <name>Arnold Kopelson</name>
    <birthday>1935-02-14T00:00:00+0000</birthday>
    <deathday/>
  </directors>
</movie>
```

```
{
  "genre":{
    "id":42,
    "slug":"thriller",
    "title":"Thriller"
  },
  "id":1234,
  "title":"Seven",
  "duration":130,
  "released":true,
  "slug":"seven",
  "description":"A brilliant thriller!",
  "release_date":"1996-01-31T00:00:00+0000",
  "directors":[
    {
      "id":973463,
      "name":"David Fincher",
      "birthday":"1962-05-10T00:00:00+0000",
      "deathday":null
    },
    {
      "id":783237,
      "name":"Arnold Kopelson",
      "birthday":"1935-02-14T00:00:00+0000",
      "deathday":null
    }
  ]
}
```

Using the Custom Normalizer.

Adding the Custom Normalizer

The built-in « Custom » normalizer is responsible for automatically calling the « normalize() » and « denormalize() » methods of your objects if they implement the corresponding interface.

```
$normalizers[] = new Normalizer\CustomNormalizer();
```

```php
use Symfony\Component\Serializer\Normalizer\NormalizableInterface;
use Symfony\Component\Serializer\Normalizer\NormalizerInterface;

class Role implements NormalizableInterface
{
    private $id;
    private $character;
    private $movie;
    private $actor;

    function normalize(NormalizerInterface $normalizer,
        $format = null, array $context = [])
    {
        return [
            'id' => $this->id,
            'character' => $this->character,
            'actor' => $this->actor,
        ];
    }
}
```

Serialization Groups

Annotation Configuration

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

class Movie
{
    /** @Groups({"admins"}) */
    private $id;

    /** @Groups({"admins", "publishers", "users" }) */
    private $title;

    /** @Groups({"admins", "publishers" }) */
    private $slug;

    /** @Groups({"admins", "publishers", "users" }) */
    private $releaseDate;

    /** @Groups({ "admins", "publishers", "users" }) */
    public function isReleased()
    {
        return new $this->releaseDate <= new \DateTime();
    }
}
```

YAML Configuration

```
Movie:
  attributes:
    id:
      groups: [ admins ]
    title:
      groups: [ admins, publishers, users ]
    slug:
      groups: [ admins, publishers ]
    releaseDate:
      groups: [ admins, publishers, users ]
    released:
      groups: [ admins, publishers, users ]
```

XML Configuration

```
<?xml version="1.0" ?>
<serializer ...>
    <class name="Movie">
        <attribute name="id">
            <group>admins</group>
        </attribute>
        <attribute name="title">
            <group>admins</group>
            <group>publishers</group>
            <group>users</group>
        </attribute>
        <attribute name="slug">
            <group>admins</group>
            <group>publishers</group>
        </attribute>
    </class>
</serializer>
```

Load Groups Metadata

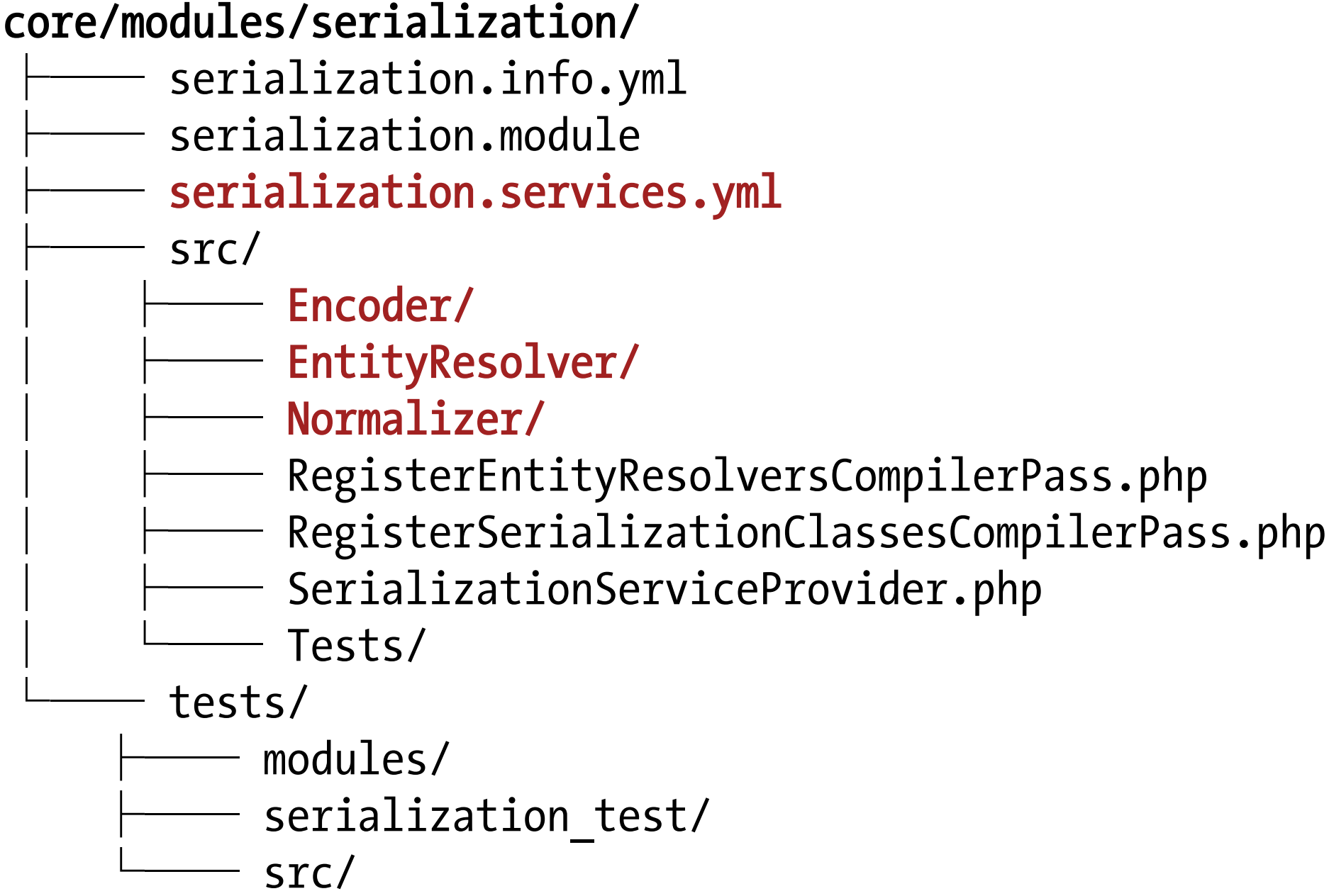
```php
use Symfony\Component\Serializer\Mapping\Loader\AnnotationLoader;
use Symfony\Component\Serializer\Mapping\Loader\XmlFileLoader;
use Symfony\Component\Serializer\Mapping\Loader\YamlFileLoader;
use Symfony\Component\Serializer\Mapping\Factory\ClassMetadataFactory;
use Doctrine\Common\Annotations\AnnotationReader;
use Doctrine\Common\Cache\ArrayCache;

// Setup a loader
$loader = new AnnotationLoader(new AnnotationReader());
$loader = new YamlFileLoader(__DIR__.'/config/serializer.yml');
$loader = new XmlFileLoader(__DIR__.'/config/serializer.xml');
$cache = new ArrayCache();

// Setup the normalizers
$factory = new ClassMetadataFactory($loader, $cache);
$normalizer = new Normalizer\ObjectNormalizer($factory, $converter);

// ...
```

Serialization Groups

```
$serializer->serialize($movie, 'xml', [
    'xml_root_node_name' => 'movie',
    'groups' => [ 'users' ],
]);

$serializer->deserialize($movie, 'Movie', 'xml', [
    'xml_root_node_name' => 'movie',
    'groups' => [ 'users' ],
]);
```

Serializer Integration into Drupal 8

The « Serialization »
Core module integrates
the Symfony Serializer
into Drupal. »


core/modules/serialization/
serialization.info.yml
serialization.module
serialization.services.yml
src/
Encoder/
EntityResolver/
Normalizer/
RegisterEntityResolversCompilerPass.php
RegisterSerializationClassesCompilerPass.php
SerializationServiceProvider.php
Tests/
tests/
modules/
serialization_test/
src/

Built-in Normalizers

```
core/modules/serialization/src/Normalizer/
 ├── ComplexDataNormalizer.php (default)
 ├── ConfigEntityNormalizer.php
 ├── ContentEntityNormalizer.php
 ├── EntityNormalizer.php
 ├── ListNormalizer.php
 ├── NormalizerBase.php
 ├── NullNormalizer.php
 └── TypedDataNormalizer.php
```

By default, the Drupal « Serializer » only uses its custom made normalizers.

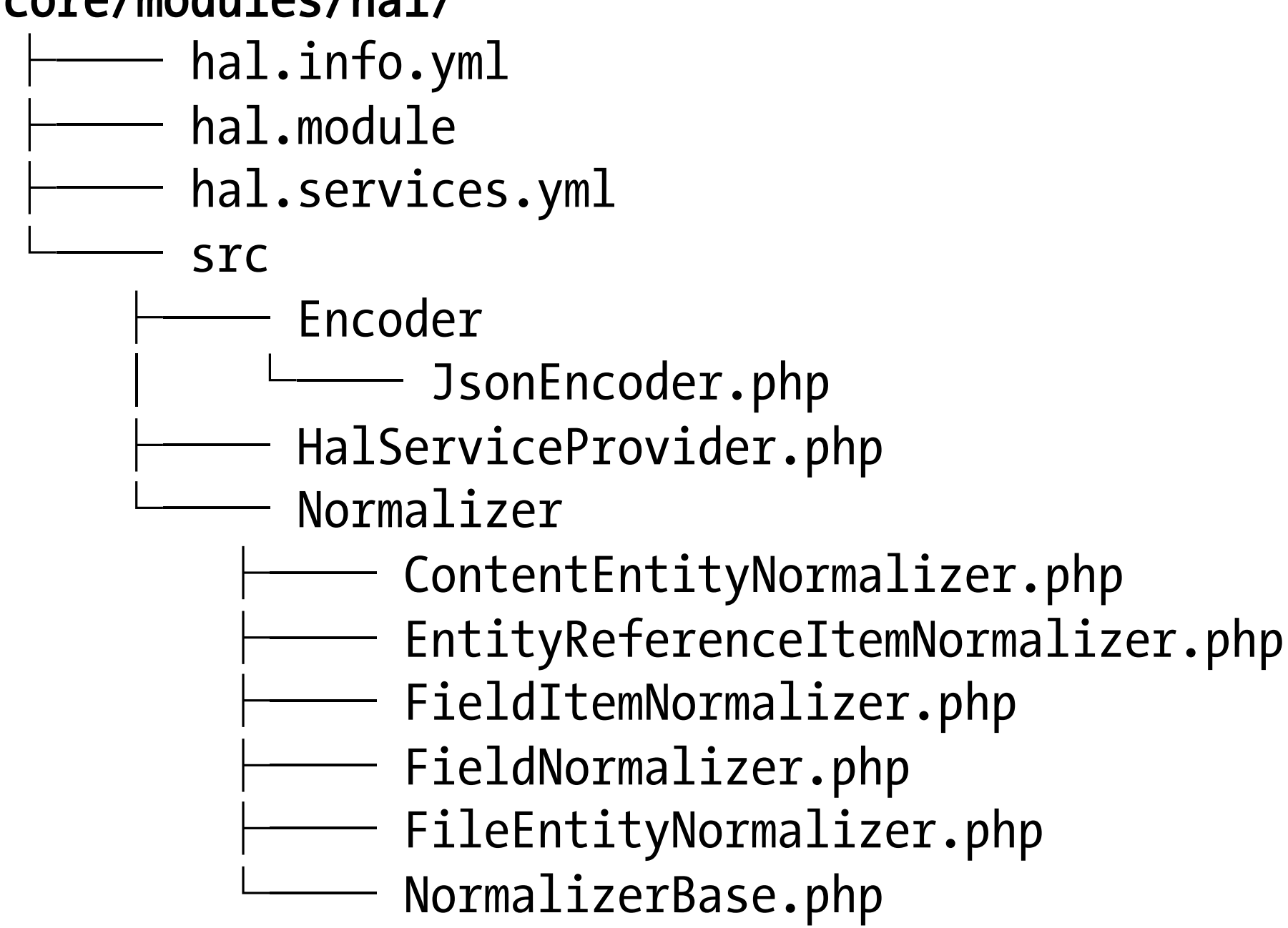
Registering Serialization Services

```
# core/modules/serialization/serialization.services.yml
services:
  serializer:
    class: Symfony\Component\Serializer\Serializer
    arguments: [{  }, {  }]

  serializer.normalizer.list:
    class: Drupal\serialization\Normalizer\ListNormalizer
    tags:
      - { name: normalizer }

  serializer.encoder.json:
    class: Drupal\serialization\Encoder\JsonEncoder
    tags:
      - { name: encoder, format: json }
```

Built-in Services

```
serializer

# Normalizers
serializer.normalizer.password_field_item
serializer.normalizer.config_entity
serializer.normalizer.content_entity
serializer.normalizer.entity
serializer.normalizer.complex_data
serializer.normalizer.list
serializer.normalizer.typed_data

# Encoders
serializer.encoder.json
serializer.encoder.xml

# Entity Resolvers (for HAL REST web services)
serializer.entity_resolver
serializer.entity_resolver.uuid
serialization.entity_resolver.target_id
```

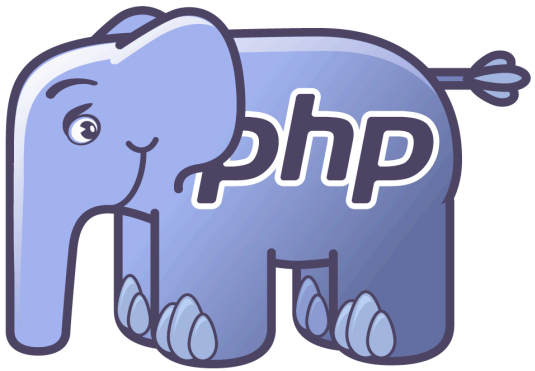
The « Hal » Core module also integrates the Symfony Serializer into Drupal. »


core/modules/hal/
hal.info.yml
hal.module
hal.services.yml
src
Encoder
JsonEncoder.php
HalServiceProvider.php
Normalizer
ContentEntityNormalizer.php
EntityReferenceItemNormalizer.php
FieldItemNormalizer.php
FieldNormalizer.php
FileEntityNormalizer.php
NormalizerBase.php

Built-in Services

```
services:
  serializer.normalizer.entity_reference_item.hal:
    class: Drupal\hal\Normalizer\EntityReferenceItemNormalizer
    arguments: [@rest.link_manager, @serializer.entity_resolver]
    tags:
      - { name: normalizer, priority: 10 }

  serializer.normalizer.entity.hal:
    class: Drupal\hal\Normalizer\ContentEntityNormalizer
    arguments: [@rest.link_manager, @entity.manager, @module_handler]
    tags:
      - { name: normalizer, priority: 10 }

  serializer.encoder.hal:
    class: Drupal\hal\Encoder\JsonEncoder
    tags:
      - { name: encoder, priority: 10, format: hal_json }
```

Going Further with Data Serialization



JMS Serializer Library

- Yaml / XML / Json Serialization
- Advanced Serialization Mapping
- Handle Circular References gracefully
- Advanced Metadata Configuration
- Integrates with Doctrine / Symfony / ZF...
- Versionning support
- Extensible at will

```
http://jmsyst.com/libs/serializer
```

Some Final Thoughts

- Serializing data is « mostly » easy to achieve!
- Deserializing is not easy at all!

- For simple use cases, use the Symfony Serializer!
- For advanced use cases, use the JMS Serializer!

```
http://jmsyst.com/libs/serializer
```

Thank You!

Hugo Hamon
hugo.hamon@sensiolabs.com
@hhamon

Symfony

SensioLabs