Introducing

Symfony

The most revolutionary thing in France since 1789

Allow me to introduce myself

Claudio Beatrice
@omissis

6+ years of
experience on PHP

Organizing Drupal
events since 2009

drupal italia ASSOCIAZIONE

PHP, Drupal &
Symfony consulting

agavee DEVELOPMENT

GOOM RADIO

Web Radio

Skylogic
a eutelsat company

Telecommunications

What's Symfony2

A reusable set of standalone, decoupled, and cohesive PHP 5.3 components

A full-stack web framework 

A Request/Response framework built around the HTTP specification

A promoter of best practices, standardization and interoperability 

An awesome community!

Leave The STUPID Alone

As time went by, habits and practices that once seemed acceptable have proven that they were making our code harder to understand and maintain. In other words, STUPID.

But what makes code such a thing?

- **S**ingleton
- **T**ight coupling
- **U**ntestability
- **P**remature optimization
- **I**ndescriptive naming
- **D**uplication

Singleton

It's a design pattern that restricts the creation of an object to one instance(think of a DB connection).

It does introduce **undesirable limitations**(what if we'll need TWO DB connections?), **global state** and **hardcoded dependencies** which are all making code more difficult to test and more coupled.

Singleton

```
class DB
{
    private static $instance;

    private function __construct()
    {
        // ... code goes here ...
    }

    public static function getInstance()
    {
        if (empty(self::$instance)) {
            self::$instance = new self;
        }
        return self::$instance;
    }

    // ... more code goes here ...
}
```

Tight Coupling

It happens when classes are put in relation by using **type hints**, **static calls** or **direct instantiation**.

Introduces **hardcoded dependencies** between classes, which complicates:

- **code reuse**
- **unit testing**
- **integration**
- **modifications**

Tight Coupling

```php
// An example of tight coupling
class House {
    public function __construct() {
        $this->door   = new Door();
        $this->window = new Window();
    }
}

// And a possible solution
class House {
    // Door and Window are interfaces
    public function __construct(Door $door, Window $window) {
        $this->door   = $door;
        $this->window = $window;
    }
}
```

Untestability

If classes are **complex**, **tightly coupled** or **trying to do too much**, then there's a good chance that it's also quite hard if not impossible to test it in isolation.

The lack of proper test coverage will make code harder to maintain and change, as it becomes very difficult to tell if any modification is actually breaking something.

Premature Optimization

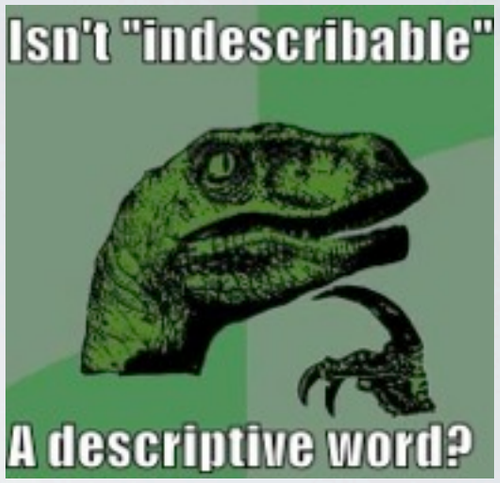
Most of the time major performance issues are caused by small portions of code(80/20 rule).

It is easier to **optimize correct code** than to **correct optimized code**.

Performance are not always a concern, therefore **optimize when it's a proved problem**, you'll save time and raise productivity.

Indescriptive Naming

There are two hard things in computer science: cache invalidation, naming things and off-by-one errors.
-- Phil Karlton, variated by the Interwebs

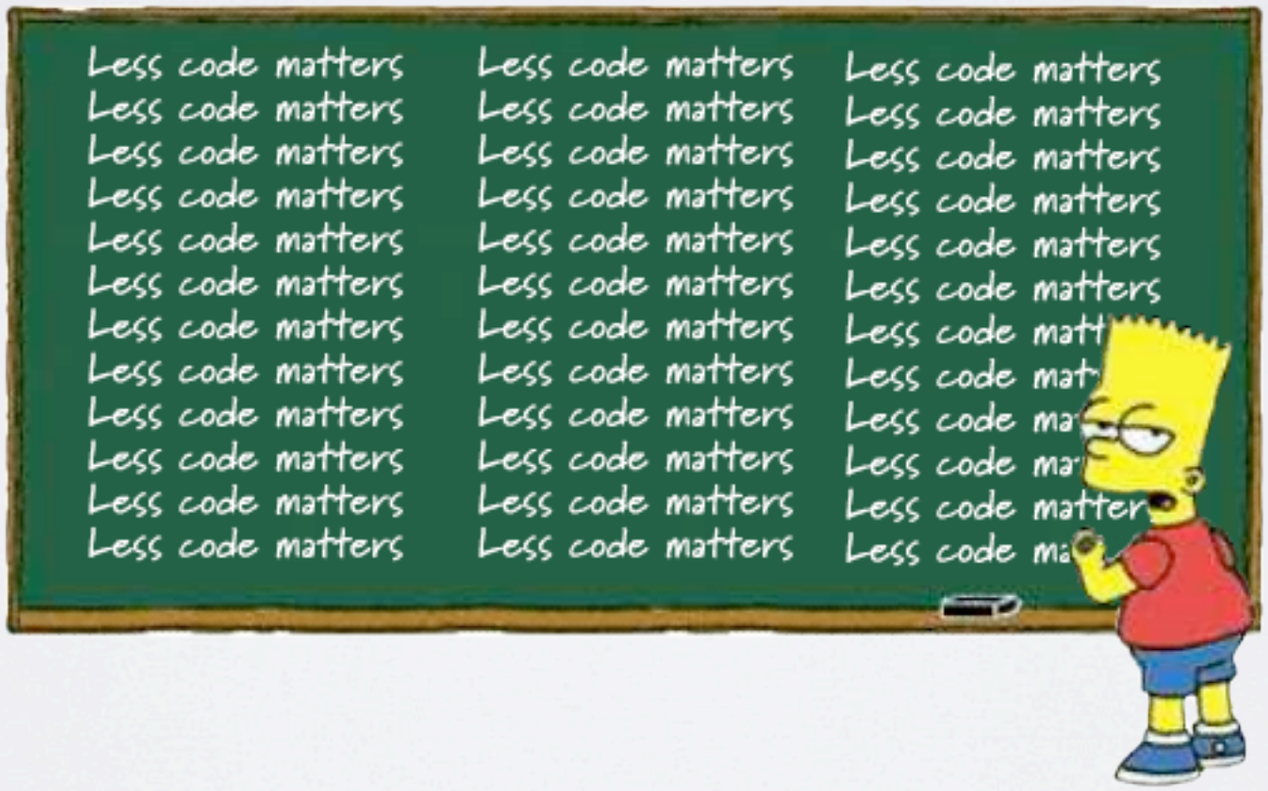


Even if **hard**, **naming is** a **fundamental** part of the job and should be considered part of the documentation, therefore remember to:

- **communicate intents**
- **favor clarity over brevity**
- think that **code is read far more often than written**, so it's more convenient to ease "reads" over "writes"

Duplication

How many times did they tell you to not repeat yourself?



BE SOLID!

What alternatives to write STUPID code do we have?

Another acronym to the rescue: SOLID!

It encloses five **class design principles**:

- **S**ingle responsibility principle
- **O**pen/closed principle
- **L**iskov substitution principle
- **I**nterface segregation principle
- **D**ependency inversion principle

Single Responsibility

There should never be more than one reason for a class to change.

Every class should have a single responsibility and fully encapsulate it.

If change becomes localized, complexity and cost of change are reduced, moreover there's less risk of ripple effects.

Single Responsibility

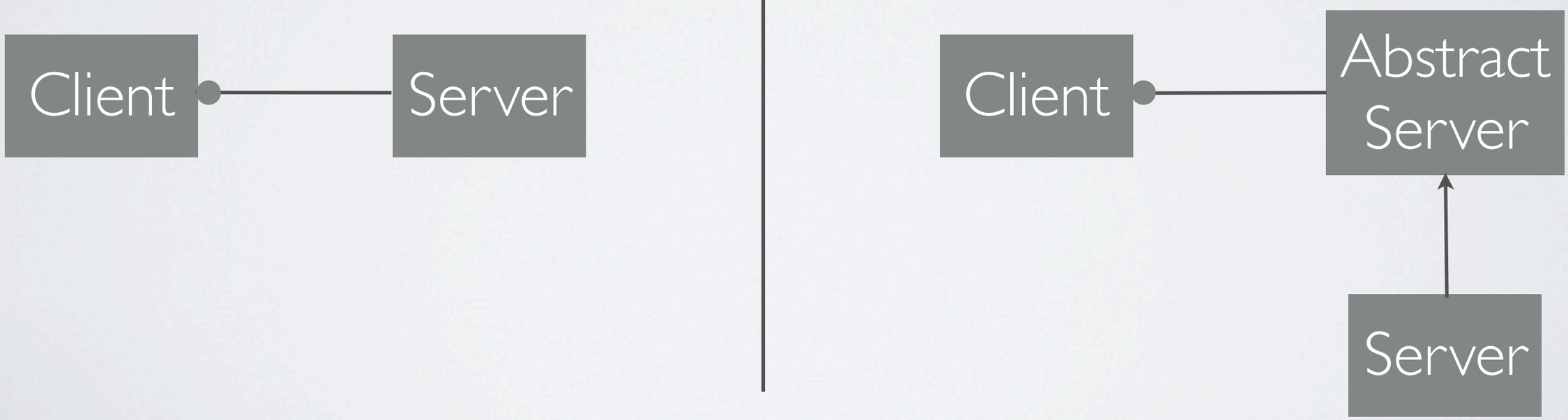
```
interface Modem
{
    function dial($phoneNumber);
    function hangup();
    function send($message);
    function receive();
}
```

The above interface shows two responsibilities: connection management and data communication, making them good candidates for two separate interfaces/implementations.

Open/Closed

Software entities (classes, functions, etc) should be open for extension, but closed for modification.

This principle states that the source code of software entities shouldn't ever be changed: those entities must be derived in order to add the wanted behaviors.



Liskov Substitution

Objects in a program should be replaceable with instances of their subtypes without altering any of the desirable properties of that program, such as correctness and performed task.

It intends to guarantee semantic interoperability of object types in a hierarchy.

Liskov Substitution

```php
class Rectangle {
  protected $width;
  protected $height;

  function setWidth($width) {...}
  function getWidth() {...}
  function setHeight($height) {...}
  function getHeight() {...}
}

class Square extends Rectangle {
  function setWidth($width) {
    $this->width = $width;
    $this->height = $width;
  }
  function setHeight($height) {
    $this->width= $height;
    $this->height = $height;
  }
}
```

```php
function draw(Rectangle $r) {
  $r->setWidth(5);
  $r->setHeight(4);

  // is it correct to assume that
changing the width of a Rectangle
leaves is height unchanged?
  assertEquals(
    20,
    $r->setWidth() * $r->setHeight()
  );
}
```

Liskov Substitution

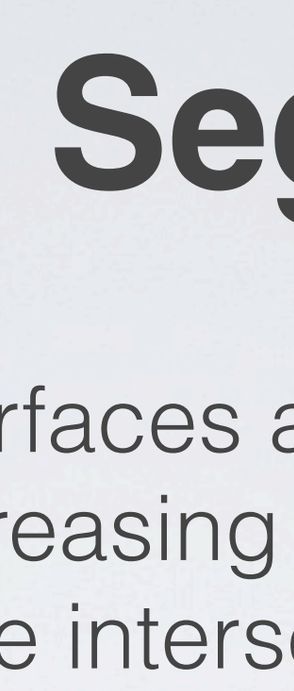
The flaw in the Rectangle-Square design shows that even if conceptually a square is a rectangle, a *Square* object is not a *Rectangle* object, since **a *Square* does not behave as a *Rectangle***.

As a result, the public behavior the clients expect for the base class must be preserved in order to conform to the LSP.

Interface Segregation

Many client-specific interfaces are better than one big one. This principle helps decreasing the coupling between objects by minimizing the intersecting surface.

```
interface MultiFunctionPrinter
{
    function print(...);
    function scan(...);
    function fax(...);
}
```



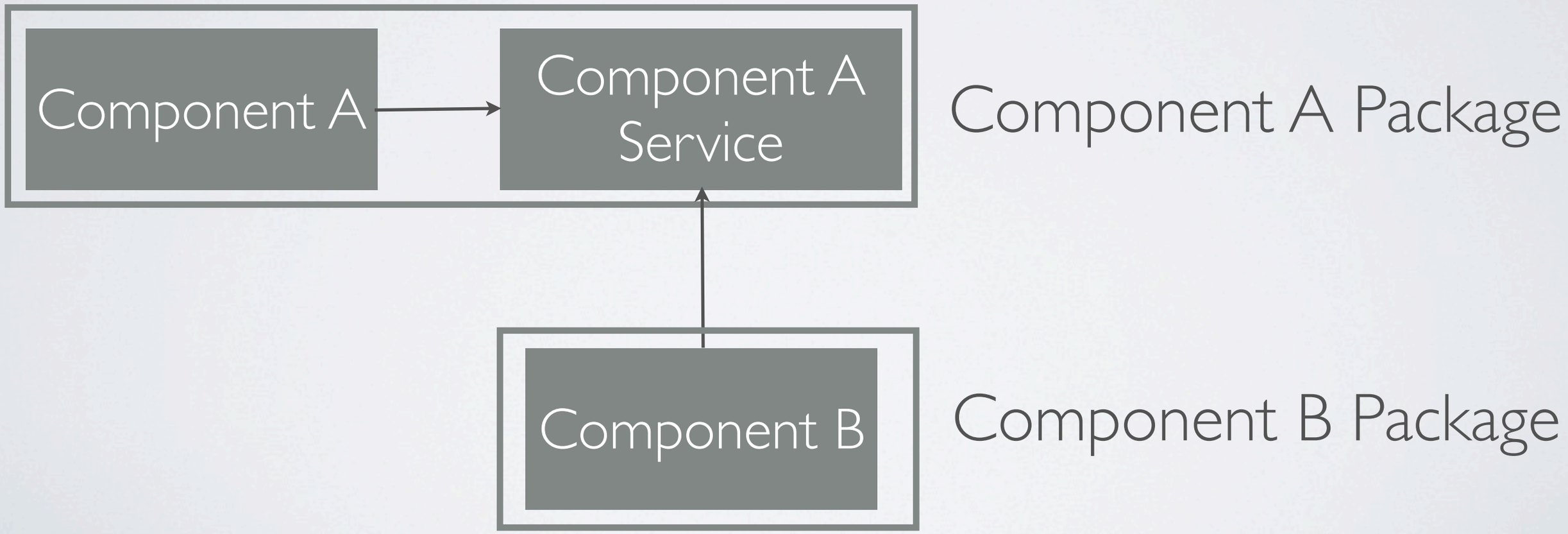
```
interface Printer
{
    function print(...);
}

interface Scanner
{
    function print(...);
}

interface Fax
{
    function print(...);
}
```

Dependency Inversion

- High-level entities should not depend on low-level entities and both should depend on abstractions.
- Abstractions should not depend upon details: details should depend upon abstractions.



Dependency Inversion

```
class Vehicle {
    protected $tyres;

    public function __construct() {
        $this->tyres = array_fill(0, 4, new Tyre(50));
    }
}
class Tyre {
    private $diameter;

    public function __construct($diameter) {
        $this->setDiameter($diameter);
    }

    public function setDiameter($diameter) {
        $this->diameter = $diameter;
    }

    public function getDiameter() {
        return $this->diameter;
    }
}
```

Dependency Inversion

```php
namespace Vehicle;

class Vehicle {
    protected $tyres;

    function addTyre(AbstractTyre $tyre) {
        $this->tyres[] = $tyre;
    }
}
```

```php
namespace Vehicle;

abstract class AbstractTyre {
    private $diameter;

    function __construct($diameter) {...}

    function setDiameter($diameter) {...}

    function getDiameter() {...}
}
```

```php
namespace Tyre;

use Vehicle\AbstractTyre;

class RaceTyre extends AbstractTyre {
    private $compound;

    function setCompound($compound) {...}

    function getCompound() {...}
}
```

How About Symfony Now?

Being aware of the principles of software development mentioned earlier allow us to better understand some of the choices that have been made for the framework as well as some of the tools that have been made available, such as:

- Class Loader
- Service Container
- Event Dispatcher
- HTTP Foundation
- HTTP Kernel

Class Loader

It loads your project's classes automatically if they're following a standard PHP convention aka **PSR-0**.

- **Doctrine\Common\IsolatedClassLoader**
  => /path/to/project/lib/vendor/**Doctrine/Common/IsolatedClassLoader.php**
- **Symfony\Core\Request**
  => /path/to/project/lib/vendor/**Symfony/Core/Request.php**
- **Twig_Node_Expression_Array**
  => /path/to/project/lib/vendor/**Twig/Node/Expression/Array.php**

It's a great way to get out of the require_once hell while gaining better interoperability and lazy loading at the same time.

Service Container

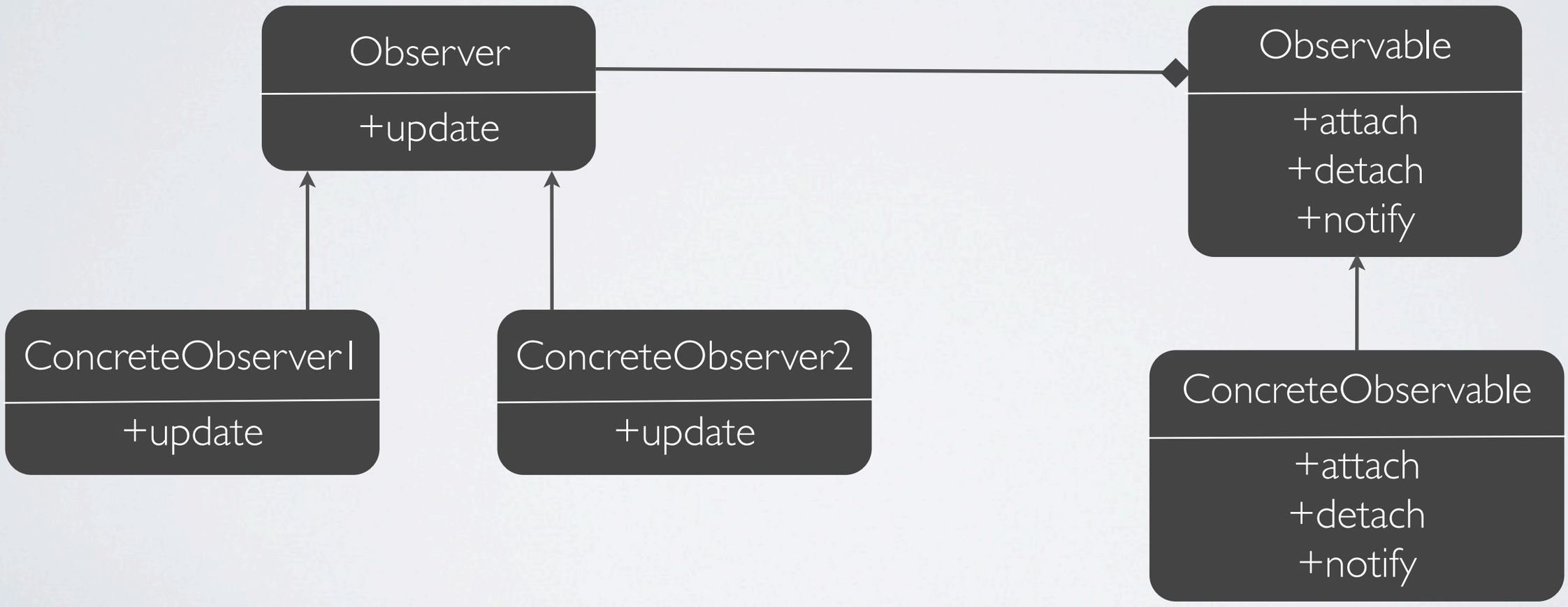
aka Dependency Injection Container

A **Service** is any PHP object that performs a “global” task: think of a Mailer class.

A **Service Container** is a special object (think of it as an **Array of Objects on Steroids**) that centralizes and standardizes the way objects are constructed inside an application: instead of directly creating Services, the developer configures the Container to take care of the task.

Event Dispatcher

A lightweight implementation of the **Observer Pattern**,
it provides a powerful and easy way to extend objects.



Event Dispatcher

```php
use Symfony\Component\EventDispatcher\EventDispatcher;

$dispatcher = new EventDispatcher();

$callable = function (Event $event) use ($log) {
    $log->addWarning('th3 n1nj4 d1sp4tch3r 1s 4ft3r y0u');
}
$dispatcher->addListener('foo.bar', $callable);

$dispatcher->dispatch('foo.bar', new Event());
```

HTTP Foundation

It replaces the PHP's global variables and functions that represent either requests or responses with a full-featured object-oriented layer for the HTTP messages.

```php
use Symfony\Component\HttpFoundation\Request;
// http://example.com/?foo=bar
$request = Request::createFromGlobals();
$request->query->get('foo'); // returns bar
// simulate a request
$request = Request::create('/foo', 'GET', array('name' => 'Bar'));


use Symfony\Component\HttpFoundation\Response;

$response = new Response('Content', 200, array(
    'content-type' => 'text/html'
));
// check the response is HTTP compliant and send it
$response->prepare($request);
$response->send();
```

HTTP Kernel

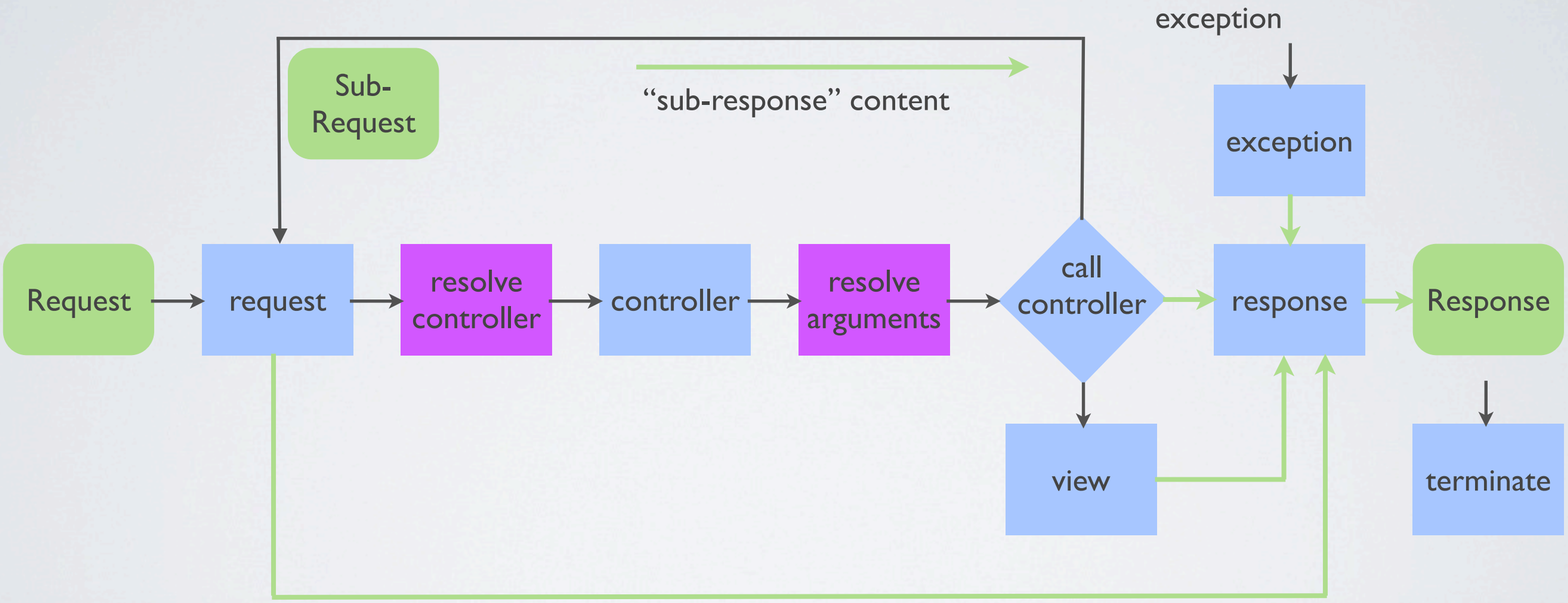
The Kernel is the core of Symfony2: it is built on top of the HttpFoundation and its main goal is to “convert” a Request object into a Response object using a Controller, which in turn can be any kind of PHP callable.

```
interface HttpKernelInterface
{
    const MASTER_REQUEST = 1;
    const SUB_REQUEST = 2;

    /**
     * ...
     * @return Response A Response instance
     * ...
     * @api
     */
    function handle(Request $request, $type = self::MASTER_REQUEST,
$catch = true);
}
```

HTTP Kernel

Workflow



Symfony is not enough

Let's take a look at some of the most important third-party libraries

Doctrine2

The Doctrine Project is made of a selected set of PHP libraries primarily focused on providing persistence services and related functionality:

- Common
- Database Abstraction Layer
- Object Relational Mapper
- MongoDB Object Document Mapper
- CouchDB Object Document Mapper
- PHPCR Object Document Mapper
- Migrations

Doctrine2

```php
namespace Drupal\Bundle\NodeBundle\Document;

use Doctrine\ODM\MongoDB\Mapping\Annotations as MongoDB;
use Doctrine\Common\Persistence\PersistentObject;

/**
 * @MongoDB\Document(collection="node")
 */
class Node extends PersistentObject
{
    /**
     * @MongoDB\Id
     */
    protected $id;

    /**
     * @MongoDB\String
     */
    protected $title;

    // accessor and mutators
}
```

Twig

A flexible, fast and secure template engine for PHP.

It offers a great set of features, a concise syntax and very good performances (it compiles to PHP and has an optional C extension); moreover it's super easy to extend and it's thoughtfully documented.

It gives the presentation layer a big boost in terms of expressiveness, making it more powerful and easier to use: prepare yourself for sweet hugs by front-end developers :)

Twig

```
{# Node list page #}
{% extends 'layout.html.twig' %}

{% macro node_render(node) %}
    <div id="node-{{node.id}}">
        <h2>{{ node.title|title }}</h2>
        <div>{{ node.creationDate|date('d/m/Y') }}</div>
        <div>{{ node.body }}</div>
        <div>{{ node.tags|join(', ') }}</div>
    </div>
{% endmacro %}

{% block body %}
    {% for node in nodes %}
        node_render(node);
    {% else %}
        {{ 'We did not find any node.'|trans }}
    {% endfor %}
{% endblock body %}
```

Tags

autoescape
block
do
embed
extends
filter
flush
for
from
if
import
include
macro
raw
sandbox
set
spaceless
use

Filters

abs
capitalize
convert_encoding
date
default
escape
format
join
json_encode
keys
length
lower
merge
nl2br
number_format
raw
replace
reverse
slice
sort
striptags
title
trim
upper
url_encode

Functions

attribute
block
constant
cycle
date
dump
parent
random
range

Tests

constant
defined
divisibleby
empty
even
iterable
null
odd
sameas

Operators

in
is
Math (+, -, /, %, /, *, **)
Logic (and, or, not, (), b-and, b-xor, b-or)
Comparisons (==, !=, <, >, >=, <=, ===)
Others (.., |, ~, ., [], ?:)

Assetic

An advanced asset management framework for PHP.

```
$css = new AssetCollection(array(
    new FileAsset('/path/to/src/styles.less', array(new LessFilter())),
    new GlobAsset('/path/to/css/*'),
), array(
    new Yui\CssCompressorFilter('/path/to/yuicompressor.jar'),
));

// this will echo CSS compiled by LESS and compressed by YUI
echo $css->dump();
```

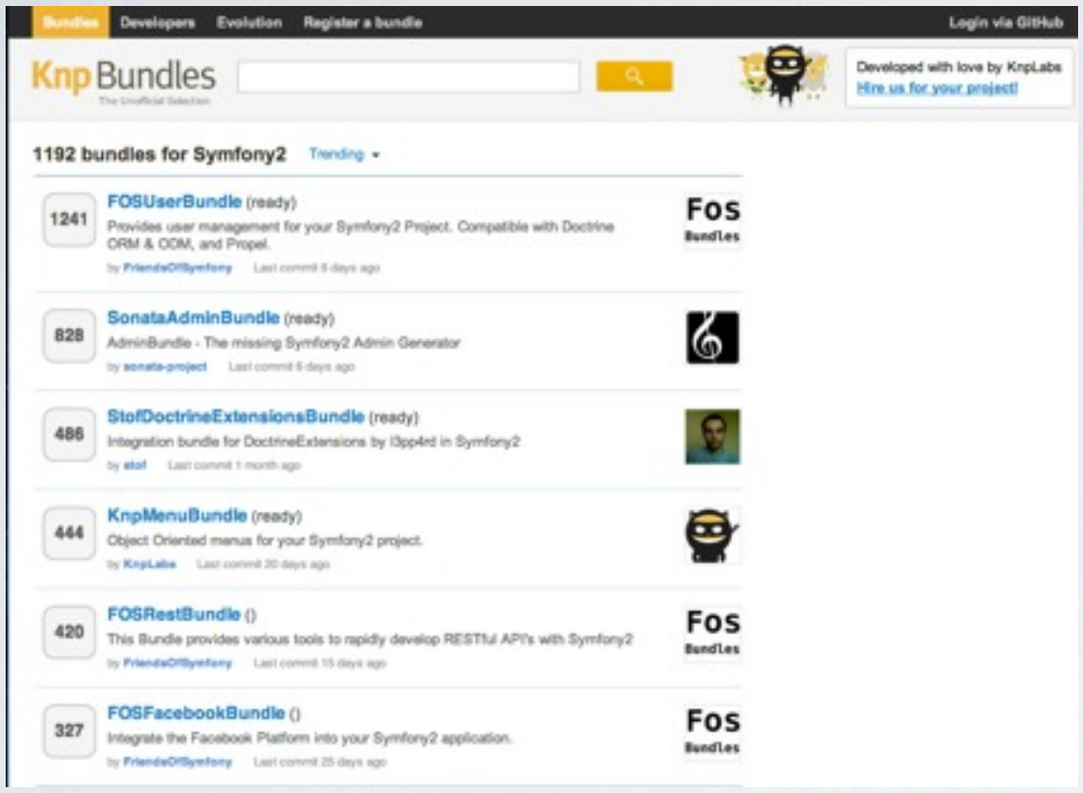
It ships with a strong set of filters for handling css, js, less, sass, compression, minifying and much more. Moreover, it's nicely integrated with Twig.

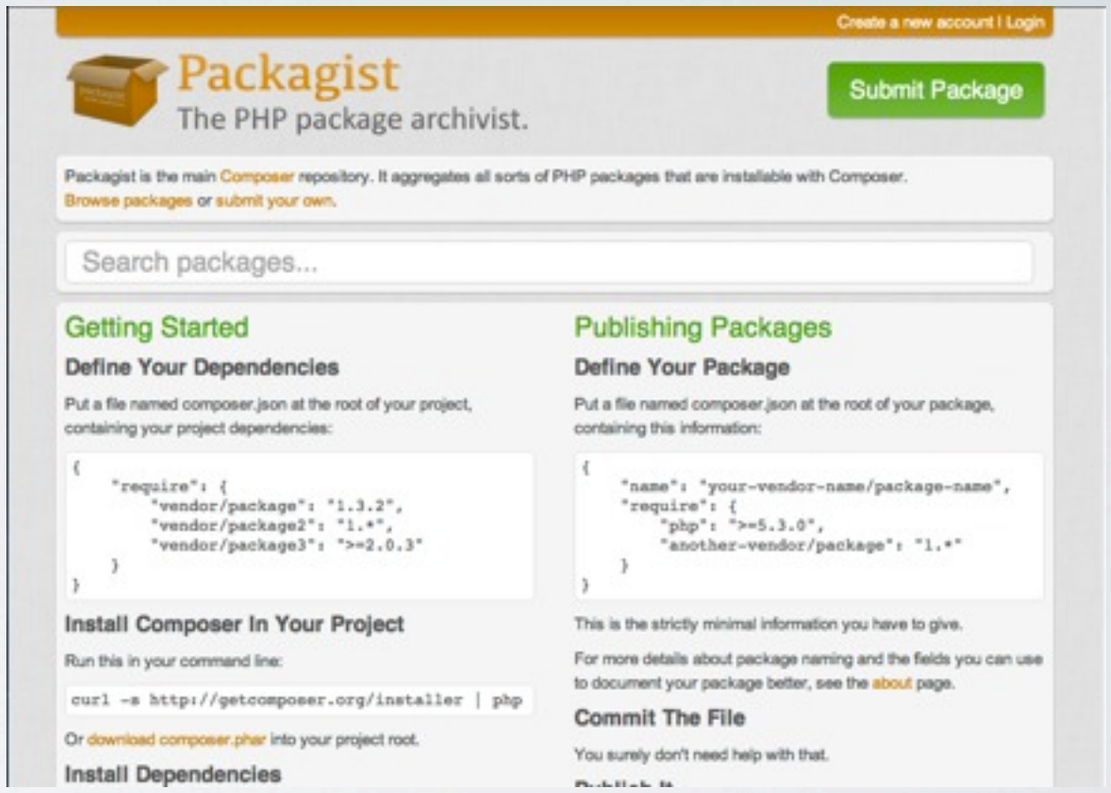
Giving The Devil His Due

Some resources I used to make these slides:

- http://nikic.github.com/
- http://fabien.potencier.org/
- http://symfony.com/
- http://www.slideshare.net/jwage/symfony2-from-the-trenches
- http://www.slideshare.net/weaverryan/handson-with-the-symfony2-framework
- http://www.slideshare.net/weaverryan/symony2-a-next-generation-php-framework
- http://martinfowler.com/articles/injection.html
- http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

And Many Moar!





Find many more Symfony2 Bundles and PHP Libraries at **knpbundles.com** and **packagist.org**! (and while you're at it, take a look at **Composer**! ;)

Thank You!

Claudio Beatrice
@omissis