



October 27th - 31st 2008
Mainz, Germany

Decouple your PHP code for reusability

Fabien Potencier

Who am I?

- Founder of Sensio
 - Web Agency
 - Since 1998
 - About 50 people
 - Open-Source Specialists
 - Big corporate customers
- Creator and lead developer of symfony
 - PHP framework

Coupling?

Coupling is the degree to which each program module relies on each one of the other modules.

[http://en.wikipedia.org/wiki/Coupling_\(computer_science\)](http://en.wikipedia.org/wiki/Coupling_(computer_science))

Low Coupling

With low coupling, a change in one module will not require a change in the **implementation** of another module.

Low Coupling

A module can interact with another module but through a **stable interface**

Why does it matter?

Testability

The ability to instantiate a class
in a test demonstrates
the level of decoupling
of your code

Readability

Code spends more time
being read and maintained
than being created

Maintainability

A change "here"
must not cause
a problem "there"

Extensibility

Smaller loosely coupled modules
makes your code
easier to extend

Reusability

The programming world
is evolving fast... very fast

But the ultimate goal of these
changes is the same

Build better tools to avoid reinventing the wheel

Build on existing code / work / libraries

Specialize for your specific needs

That's why Open-Source wins

Better tools are possible because
we base our work on existing work

We learn from other projects

We can adopt their ideas

We can make them better

This is the process of evolution

Reusability

Configuration

Customization / Extension

Documentation / Support

Dependency Injection

A real world « web » example

In most web applications,
you need to manage the user preferences

- The user language
- Whether the user is authenticated or not
- The user credentials
- ...

This can be done with a User object

- `setLanguage()`, `getLanguage()`
- `setAuthenticated()`, `isAuthenticated()`
- `addCredential()`, `hasCredential()`
- ...

The User information
need to be persisted
between HTTP requests

We use the PHP session for the Storage

```
class SessionStorage
{
    function __construct($cookieName = 'PHP_SESS_ID')
    {
        session_name($cookieName);
        session_start();
    }

    function set($key, $value)
    {
        $_SESSION[$key] = $value;
    }

    // ...
}
```

```
class User
{
    protected $storage;

    function __construct()
    {
        $this->storage = new SessionStorage();
    }

    function setLanguage($language)
    {
        $this->storage->set('language', $language);
    }

    // ...
}
```

```
$user = new User();
```

I want to change the session cookie name

```
class User
```

```
{
```

```
    protected $storage;
```

```
    function __construct()
```

```
    {
```

```
        $this->storage = new SessionStorage('SESSION_ID');
```

```
    }
```

```
    function setLanguage($language)
```

```
    {
```

```
        $this->storage->set('language', $language);
```

```
    }
```

```
    // ...
```

```
}
```

Hardcode it in
the User class

```
$user = new User();
```

```
class User
{
    protected $storage;

    function __construct()
    {
        $this->storage = new SessionStorage(STORAGE_SESSION_NAME);
    }
}
```

Add a global configuration?

```
define('STORAGE_SESSION_NAME', 'SESSION_ID');
```

```
$user = new User();
```

```
class User
{
    protected $storage;

    function __construct($sessionName)
    {
        $this->storage = new SessionStorage($sessionName);
    }
}
```

Configure via
User?

```
$user = new User('SESSION_ID');
```

```
class User
{
    protected $storage;

    function __construct($storageOptions)
    {
        $this->storage = new
        SessionStorage($storageOptions['session_name']);
    }
}
```

Configure with
an array

```
$user = new User(
    array('session_name' => 'SESSION_ID')
);
```

I want to change the session storage engine

Filesystem

MySQL

SQLite

...

```
class User
{
    protected $storage;

    function __construct()
    {
        $this->storage = Registry::get('session_storage');
    }
}
```

Use a global
registry object?

```
$storage = new SessionStorage();
Registry::set('session_storage', $storage);
$user = new User();
```

The User now depends on the Registry

Instead of hardcoding
the Storage dependency
in the User class

Inject the Storage dependency
in the User object

```
class User
{
    protected $storage;

    function __construct($storage)
    {
        $this->storage = $storage;
    }
}
```



Constructor
argument

```
$storage = new SessionStorage('SESSION_ID');
$user = new User($storage);
```

Use different Storage strategies

```
class User
{
    protected $storage;

    function __construct($storage)
    {
        $this->storage = $storage;
    }
}
```

Use a different
Storage engine

```
$storage = new MySQLSessionStorage( 'SESSION_ID' );
$user = new User($storage);
```

Configuration becomes natural

```
class User
{
    protected $storage;

    function __construct($storage)
    {
        $this->storage = $storage;
    }
}
```

```
$storage = new MySQLSessionStorage( 'SESSION_ID' );
$user = new User($storage);
```



Configuration
is natural

Wrap third-party classes (Interface / Adapter)

```
class User
{
    protected $storage;

    function __construct(ISessionStorage $storage)
    {
        $this->storage = $storage;
    }
}
```



Add an
interface

```
interface ISessionStorage
{
    function get($key);

    function set($key, $value);
}
```

Mock the Storage object (for testing)

```
class User
{
    protected $storage;

    function __construct(ISessionStorage $storage)
    {
        $this->storage = $storage;
    }
}

class SessionStorageForTests implements ISessionStorage
{
    protected $data
    function set($key, $value)
    {
        self::$data[$key] = $value;
    }
}
```

Mock the
Session

Use different Storage strategies

Configuration becomes natural

Wrap third-party classes (Interface / Adapter)

Mock the Storage object (for testing)

Easy without changing the User class

That's Dependency Injection

Nothing more

« Dependency Injection is where components are given their dependencies through their constructors, methods, or directly into fields. »

<http://www.picocontainer.org/injection.html>

```
$storage = new SessionStorage();
```

```
// constructor injection
```

```
$user = new User($storage);
```

```
// setter injection
```

```
$user = new User();
```

```
$user->setStorage($storage);
```

```
// property injection
```

```
$user = new User();
```

```
$user->storage = $storage;
```

A slightly more complex web example

```
$request = new WebRequest();  
$response = new WebResponse();  
  
$storage = new  
    FileSessionStorage('SESSION_ID');  
$user = new User($storage);  
  
$cache = new FileCache(  
    array('dir' => dirname(__FILE__) . '/cache')  
);  
$routing = new Routing($cache);
```

```
class Application
{
    function __construct()
    {
        $this->request = new WebRequest();
        $this->response = new WebResponse();

        $storage = new FileSessionStorage( 'SESSION_ID' );
        $this->user = new User($storage);

        $cache = new FileCache(
            array( 'dir' => dirname(__FILE__) . '/cache' )
        );
        $this->routing = new Routing($cache);
    }
}

$application = new Application();
```

Back to square 1

```
class Application
{
    function __construct()
    {
        $request = new WebRequest();
        $response = new WebResponse();

        $storage = new FileSessionStorage('SESSION_ID');
        $user = new User($storage);

        $cache = new FileCache(
            array('dir' => dirname(__FILE__) . '/cache')
        );
        $routing = new Routing($cache);
    }
}

$application = new Application();
```

We need a Container

Objects
and relationships description

Configuration

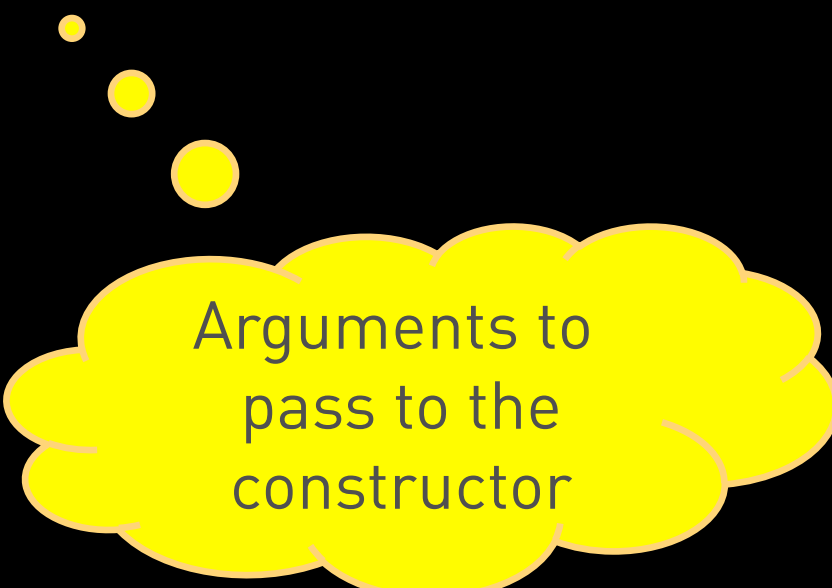
Instantiation

Service
description

```
$storageDef = new ServiceDefinition(  
    'FileSessionStorage'  
);
```

Class name

```
$storageDef = new ServiceDefinition(  
    'FileSessionStorage',  
    array('SESSION_ID')  
);
```



Arguments to
pass to the
constructor

```
$container = new ServiceContainer(array(  
    'storage' => $storageDef,  
));
```

Each object has a
unique identifier

```
$userDef = new ServiceDefinition('User',  
    array(new ServiceReference('storage'))  
);
```



Reference to
another object

```
$storageDef = new ServiceDefinition(
    'FileSessionStorage',
    array('SESSION_ID')
);

$userDef = new ServiceDefinition('User',
    array(new ServiceReference('storage'))
);

$container = new ServiceContainer(array(
    'storage' => $storageDef,
    'user'    => $userDef,
));
```

```
$user = $container->getService( 'user' );
```

Get the configuration for the user object

The User constructor must be given a storage object

Get the storage object from the container

Create a User object by passing the constructor arguments

```
$user = $container->getService( 'user' );
```

is roughly equivalent to

```
$storage = new SessionStorage( 'SESSION_ID' );  
$user = new User($storage);
```

A container
is able to manage
any object (POPO)

The objects do not need to know
they are managed
by a container

Implementation tips

```
public function getService ($id)
{
    $def = $this->definitions[$id];

    $r = new ReflectionClass($def->getClass());

    if (is_null($r->getConstructor()))
    {
        return $r->newInstance();
    }

    $args = $def->getArguments();
    $args = $this->evaluateArguments($args);

    return $r->newInstanceArgs($args);
}
```

```
public function evaluateArguments($arg)
{
    if (is_array($arg))
    {
        return array_map(
            array($this, 'evaluateArguments'), $arg
        );
    }

    if (
        is_object($arg)
        && $arg instanceof ServiceReference
    )
    {
        return $this->getService($arg);
    }

    return $arg;
}
```

}

Towards a real implementation

Scope

Each time you get a service:

Do you want a new object

or

Does the container must return the same object?

```
$userDef->setGlobal(true);
```

```
$feedReaderDef->setGlobal(false);
```

Service definition

```
$container = new ServiceContainer(array(
    'storage' => new ServiceDefinition('FileSessionStorage'),
    'user'    => new ServiceDefinition('User',
        array(new ServiceReference('storage'))
    ),
));
```

PHP

```
<service id="storage" class="FileSessionStorage" />
```

XML

```
<service id="user" class="User">
  <constructor_arg type="service" id="storage" />
</service>
```

Service configuration

```
<parameter key="storage.class">
    SQLiteSessionStorage
</parameter>
<parameter key="storage.session_name">
    SESSION_ID
</parameter>
```

Configuration is decoupled from the definition

```
<service id="storage" class="%storage.class%">
    <constructor_arg>
        %storage.session_name%
    </constructor_arg>
</service>
```

Service configuration

```
[storage]
  class = SQLiteSessionStorage
  session_name = SESSION_ID
```

```
<service id="storage" class="%storage.class%">
  <constructor_arg>
    %storage.session_name%
  </constructor_arg>
</service>
```

```
// Access parameters
```

```
$sessionName = $container['storage.session_name'];
```

```
// Access services
```

```
$user = $container->user;
```

Implementations in PHP

- Crafty: <http://phpcrafty.sourceforge.net/>
- Garden: <http://garden.tigris.org/>
spring
- Stubbles: <http://www.stubbles.net/wiki/Docs/IOC>
Google Guice
- symfony 2 will have its dependency injection framework
spring

Remember, most of the time,
you don't need a Container
to use Dependency Injection

You can start to use and benefit
from Dependency Injection today

by implementing it
in your projects

by using external libraries
that already use DI
without the need of a container

symfony
Zend Framework
ezComponents

Doctrine
Swift Mailer

...

Let's add some i18n fun

```
class User
{
    protected $storage, $i18n;

    function __construct($storage, $i18n)
    {
        $this->storage = $storage;
        $this->i18n = $i18n;
    }

    function setLanguage($language)
    {
        $this->storage->set('language', $language);

        $this->i18n->setLanguage($language);
    }
}
```

// ...

- It makes sense that the User class depends on a Storage class
 - We can't use the User object without a Storage
- It does not make sense to tie the User class to the I18n class
 - We want to be able to use the User with non internationalized websites

```

class User
{
    protected $storage, $i18n;

    function __construct($storage, $i18n = null)
    {
        $this->storage = $storage;
        $this->i18n = $i18n;
    }

    function setLanguage($language)
    {
        $this->storage->set('language', $language);

        if (!is_null($this->i18n))
        {
            $this->i18n->setLanguage($language);
        }
    }
}

```

```
class User
{
    protected $storage;

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

    function setI18n($i18n)
    {
        $this->i18n = $i18n;
    }

    function setLanguage($language)
    {
        $this->storage->set('language', $language);

        if (!is_null($this->i18n))
        {
            $this->i18n->setLanguage($language);
        }
    }
}
```

The Observer pattern

- The Observer pattern is a way to allow changes in an object to update many other objects
- It is a powerful mechanism to extend applications without having to change the object itself
- Associated words: hook, listener, event, subject,
...

- Used by a lot of applications for:
 - Plugins (Wordpress, ...)
 - Bridges between applications
 - As a master application:
 - Gallery2 (90% events / hooks for user / group create / update / delete, logout, login, ..missing: configuration changes)
 - Xaraya (100% events / hooks for user / group create / update / delete, configuration changes, logout, login, ..)
 - Wordpress (90% events / hooks for user / update / delete, rewrites, logout, login, ..)
 - Drupal (80% maybe?)
 - Joomla (100% joomla 1.5; login, logout, create, update, delete user, block, activation, system before and after start)Typo3 (50% maybe?, e.g. no unified create user event / hook)
 - As a slave application:
 - Gallery 2
 - Phorum.org

http://codex.gallery2.org/Gallery2:Embedding:Event-Based_Loose-Coupled_Integration

Implementations in PHP

- PEAR_Dispatcher
 - http://pear.php.net/package/Event_Dispatcher
- symfony implementation
 - <http://svn.symfony-project.com/branches/1.1/lib/event/>
 - http://www.symfony-project.org/book/1_1/17-Extending-Symfony
 - Based on the Cocoa notification center
 - Simple and powerful
 - Decoupled from symfony
 - Simple to use

```
$i18n = new I18n();  
$dispatcher = new sfEventDispatcher();  
$listener = array($i18n, 'listenToChangeCultureEvent');  
$dispatcher->connect('user.change_language', $listener);
```

```
$storage = new FileSessionStorage();  
$user = new User($dispatcher, $storage);
```

```
class User
{
    protected $storage, $dispatcher;

    function __construct($dispatcher, $storage)
    {
        $this->dispatcher = $dispatcher;
        $this->storage = $storage;
    }

    function setLanguage($language)
    {
        $this->storage->set('language', $language);

        $event = new sfEvent(
            $this,
            'user.change_language',
            array('language' => $language)
        );

        $this->dispatcher->notify($event);
    }
}
```

Notifiers

Dispatcher

Listeners

1

I18n listens
to `user.change_culture`

2

User notifies
`user.change_culture`

Calls
all listeners

I18n callback
is called

The User object knows nothing about the I18n one

The I18n object knows nothing about the User one

They communicate through the Dispatcher object

Any class can listen to the 'user.change_culture' event and acts accordingly

Notifiers

Dispatcher

Listeners

1

118n listens
to `user.change_culture`

Your class listens
to `user.change_culture`

2

User notifies
`user.change_culture`

Calls
all listeners

118n callback
is called

Your class callback
is called

Notifiers

Dispatcher

User notifies
`user.change_culture`



Calls
nothing

Very small
overhead

In this example, the implementation is very simple

No interface to implement

No need to create an event class for each event

An event is just

a unique identifier (user.change_culture)

some conventions (parameter names)

Advantages

very simple to use

easy to add new arguments to the listener

very fast

very easy to add a new event, just notify it with a unique name

Disadvantages

the contract between the listener and the notifier is quite loose

Implementation

```
function connect($name, $listener)
{
    if (!isset($this->listeners[$name]))
    {
        $this->listeners[$name] = array();
    }

    $this->listeners[$name][] = $listener;
}

function notify(sfEvent $event)
{
    if (!isset($this->listeners[$name]))
    {
        foreach ($this->listeners[$event->getName()] as $listener)
        {
            call_user_func($listener, $event);
        }
    }
}
```

- 3 types of notification

- Notify : all listeners are called in turn, no feedback to the notifier
 - Logging, ...
- Notify Until : all listeners are called until one has « processed » the event. The listener that has processed the event can return something to the caller
 - Exceptions, ...
- Filter : Each listener filter a given value and can change it. The filtered value is returned to the caller
 - HTML content, ...

Questions?

Contact

Fabien Potencier
fabien.potencier@sensio.com

<http://www.sensiolabs.com/>

<http://www.symfony-project.org/>