DRUPALCON

Portland

MAY 20-24 2013

Dependency Injection in D8

Kat Bailey

Building Bridges, Connecting Communities

Intro

- *Rudimentary* understanding of OOP assumed
- Big changes in D8

Agenda

- DI as a design pattern
- DI from a framework perspective
- Symfony-style DI
- DI in Drupal 8

Agenda

- DI as a design pattern

} Why?

- DI from a framework perspective
- Symfony-style DI
- DI in Drupal 8

} How?

Why Dependency Injection?

Goal: we want to write code that is...

- ✔ Clutter-free
- ✔ Reusable
- ✔ Testable

Doing It Wrong

1. An example in procedural code

```
function my_module_func($val1, $val2) {
  module_load_include('module_x', 'inc');
  $val1 = module_x_process_val($val1);
  return $val1 + $val2;
}
```

```
function my_module_func($val1, $val2) {
  module_load_include('module_x', 'inc');
  $val1 = module_x_process_val($val1);
  return $val1 + $val2;
}
```

- ✗ Clutter-free
- ✗ Reusable
- ✗ Testable

Doing It Wrong

1. An example in procedural code
2. An example in Object Oriented code

```
class Notifier {
  private $mailer;

  public function __construct() {
    $this->mailer = new Mailer();
  }

  public function notify() {
    ...
    $this->mailer->send($from, $to, $msg);
    ...
  }
}
```

```
class Notifier {
  private $mailer;

  public function __construct() {
    $this->mailer = new Mailer('sendmail');
  }

  public function notify() {
    ...
    $this->mailer->send($from, $to, $msg);
    ...
  }
}
```

```
class Notifier {
  private $mailer;

  public function __construct(Mailer $m) {
    $this->mailer = $m;
  }

  public function notify() {
    ...
    $this->mailer->send($from, $to, $msg);
    ...
  }
}
```

```
class Notifier {
  private $mailer;

  public function __construct(Mailer $m) {
    $this->mailer = $m;
  }

  public function notify() {
    ...
    $this->mailer->send($from, $to, $msg);
    ...
  }
}

$mailer = new Mailer();
$notifier = new Notifier($mailer);
```

Goal: we want to write code that is...

- ✔ Clutter-free
- ✔ Reusable
- ✔ Testable

Goal: we want to write code that is...

- ✔ Clutter-free
- ✔ Reusable
- ✔ Testable

} Ignorant

The less your code knows, the more reusable it is.

```
class Notifier {
  private $mailer;

  public function __construct(Mailer $m) {
    $this->mailer = $m;
  }

  public function notify() {
    ...
    $this->mailer->send($from, $to, $msg);
    ...
  }
}

$mailer = new Mailer();
$notifier = new Notifier($mailer);
```

```
class Notifier {
  private $mailer;

  public function
     __construct(MailerInterface $m) {
     $this->mailer = $m;
  }

  public function notify() {
     ...
     $this->mailer->send($from, $to, $msg);
     ...
  }
}

$mailer = new SpecialMailer();
$notifier = new Notifier($mailer);
```

```
class Notifier {
  private $mailer;

  public function
     __construct(MailerInterface $m) {
     $this->mailer = $m;
  }

  public function notify() {
     ...
     $this->mailer->send($from, $to, $msg);
     ...
  }
}

$mailer = new SpecialMailer();
$notifier = new Notifier($mailer);
```

Constructor Injection

Dependency Injection

Declaratively express dependencies in the class definition rather than instantiating them in the class itself.

Constructor Injection is not the only form of DI

Setter Injection

```
class Notifier {
  private $mailer;

  public function
   setMailer(MailerInterface $m) {
    $this->mailer = $m;
  }

  public function notify() {
    ...
    $this->mailer->send($msg);
  }
}

$mailer = new Mailer();
$notifier = new Notifier();
$notifier->setMailer($mailer);
```

```
class Notifier {
  private $mailer;

  public function
   setMailer(MailerInterface $m) {
    $this->mailer = $m;
  }

  public function notify() {
    ...
    $this->mailer->send($msg);
  }
}
```

Setter Injection

```
$mailer = new Mailer();
$notifier = new Notifier();
$notifier->setMailer($mailer);
```

Interface Injection

Like setter injection, except there is an interface for each dependency's setter method.

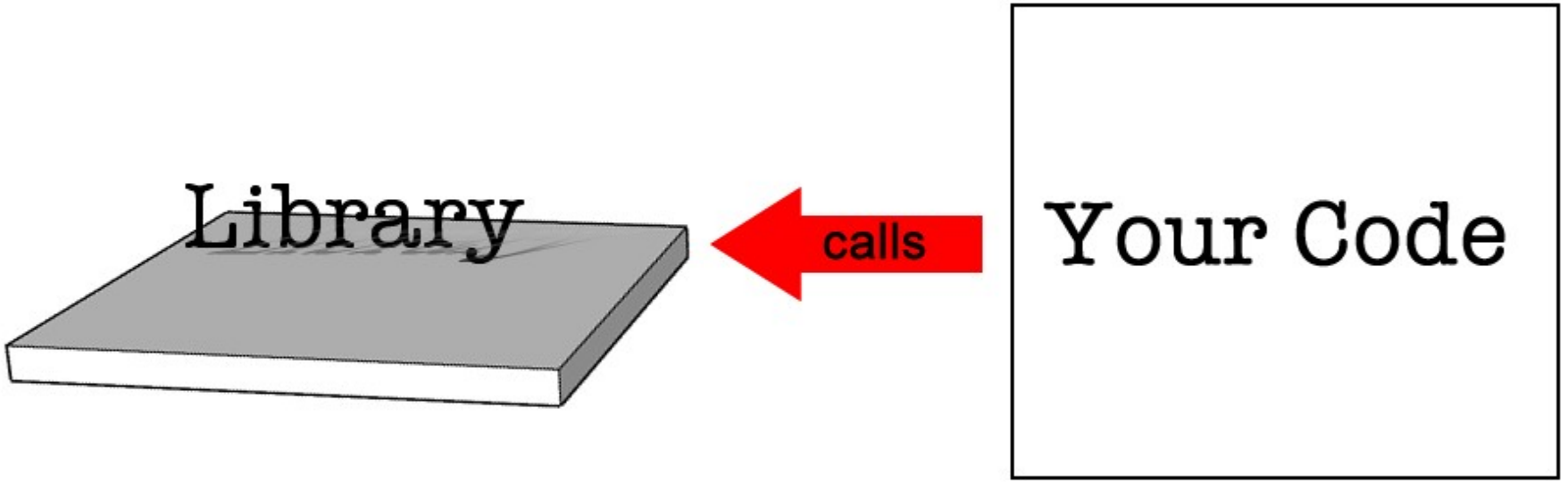
Very verbose

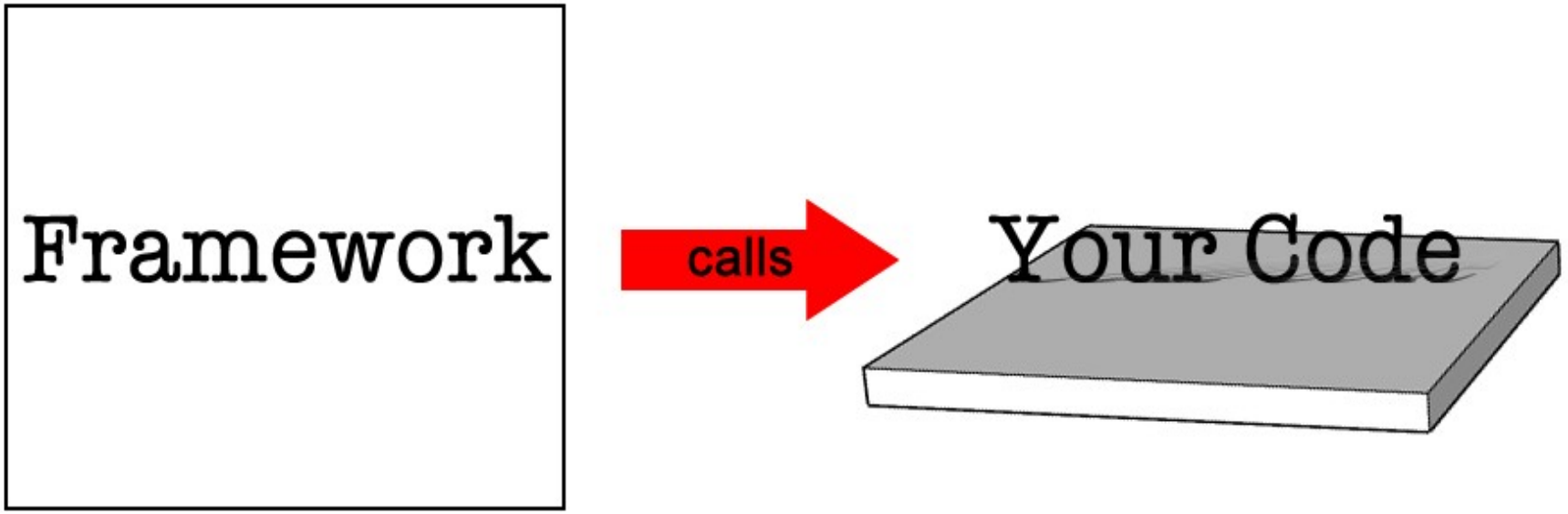
Not very common

Dependency Injection

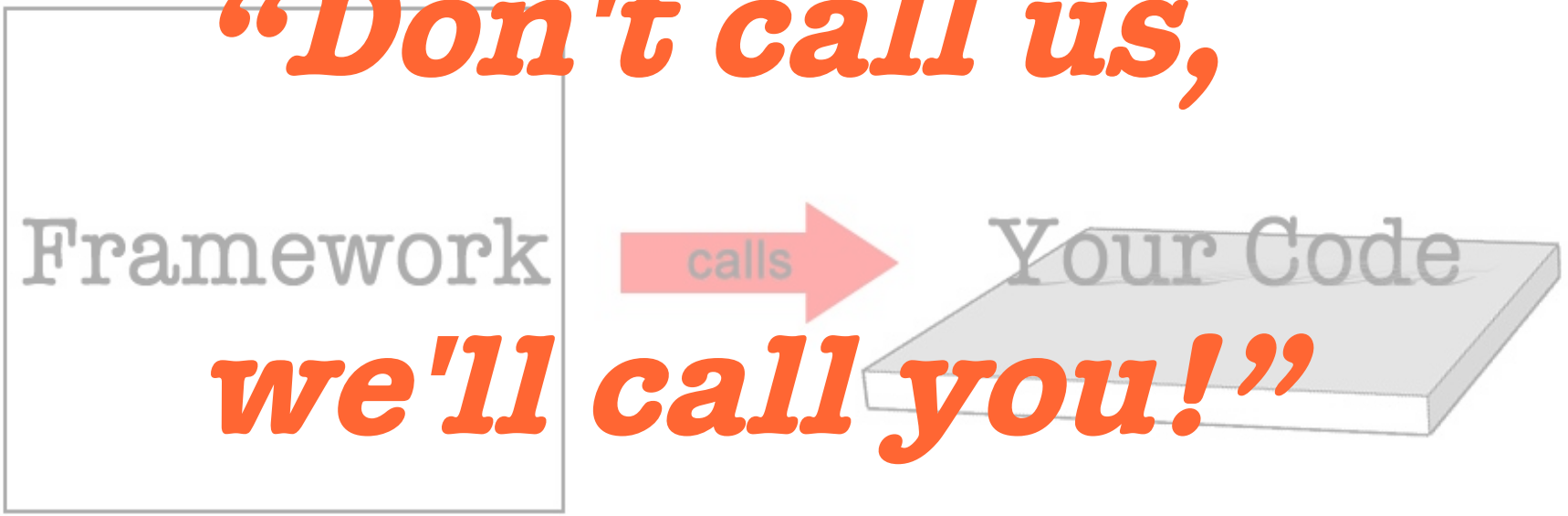
==

Inversion of Control


Library
calls
Your Code


Framework
calls
Your Code

"Don't call us, we'll call you!"



(The Hollywood Principle)

```
class Notifier {
  private $mailer;

  public function
     __construct(MailerInterface $m) {
     $this->mailer = $m;
  }

  public function notify() {
     ...
     $this->mailer->send($from, $to, $msg);
     ...
  }
}

$mailer = new Mailer();
$notifier = new Notifier($mailer);
```

```
class Notifier {
  private $mailer;

  public function
     __construct(MailerInterface $m) {
     $this->mailer = $m;
  }

  public function notify() {
     ...
     $this->mailer->send($from, $to, $msg);
     ...
  }
}
```

```
$mailer = new Mailer();
$notifier = new Notifier($mailer);
```

?

Where does injection happen?

Where does injection happen?

- ➔ Manual injection
- ➔ Use a factory
- ➔ Use a container / injector

Using DI in a Framework

Dependency Injector

==

Dependency Injection Container (DIC)

==

IoC Container

==

Service Container

The Service Container

- Assumes responsibility for constructing object graphs (i.e. instantiating your classes with their dependencies)
- Uses configuration data to know how to do this
- Allows infrastructure logic to be kept separate from application logic

Objects as Services

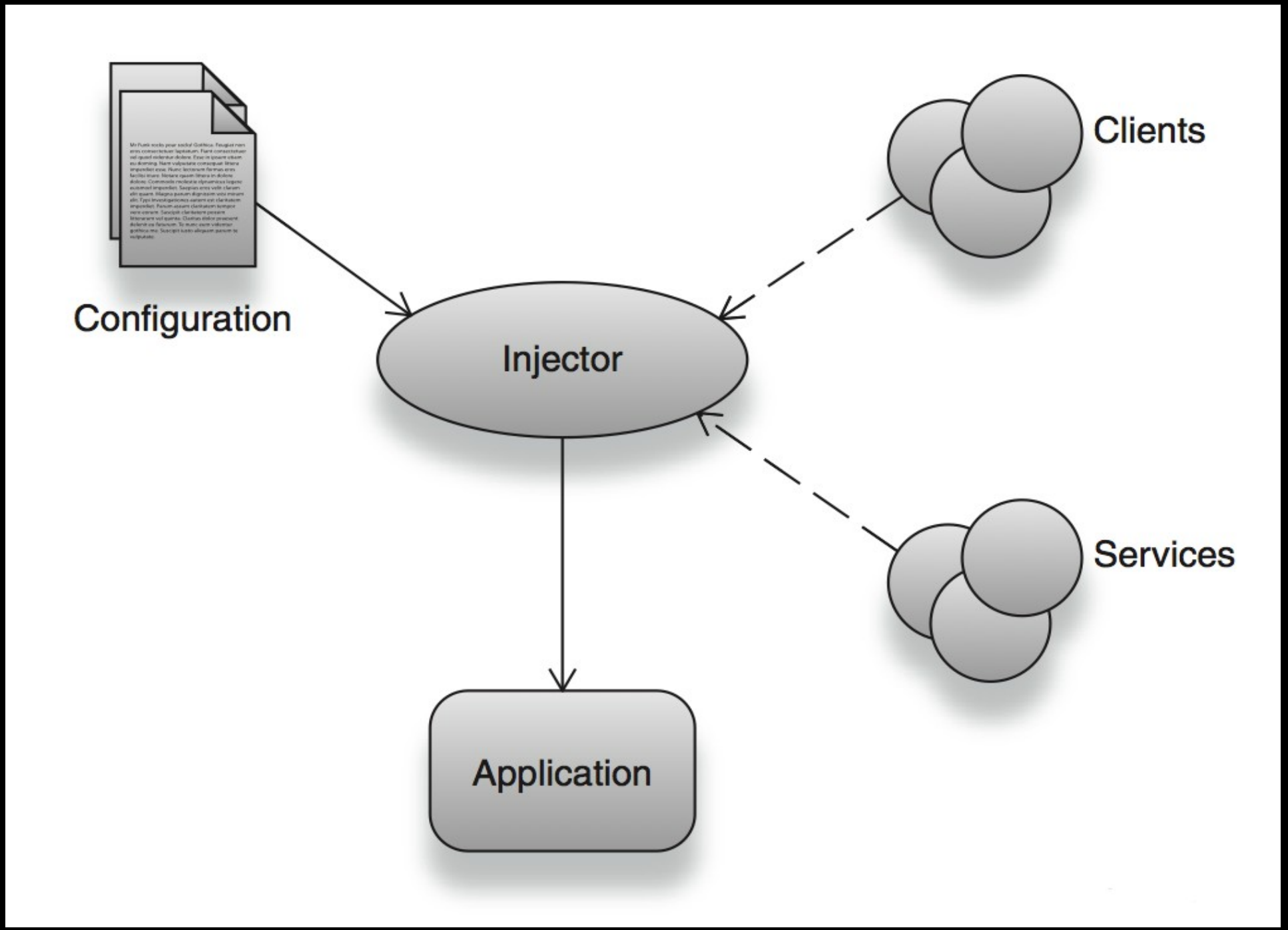
A service is an object that provides some kind of globally useful functionality

Examples of Services

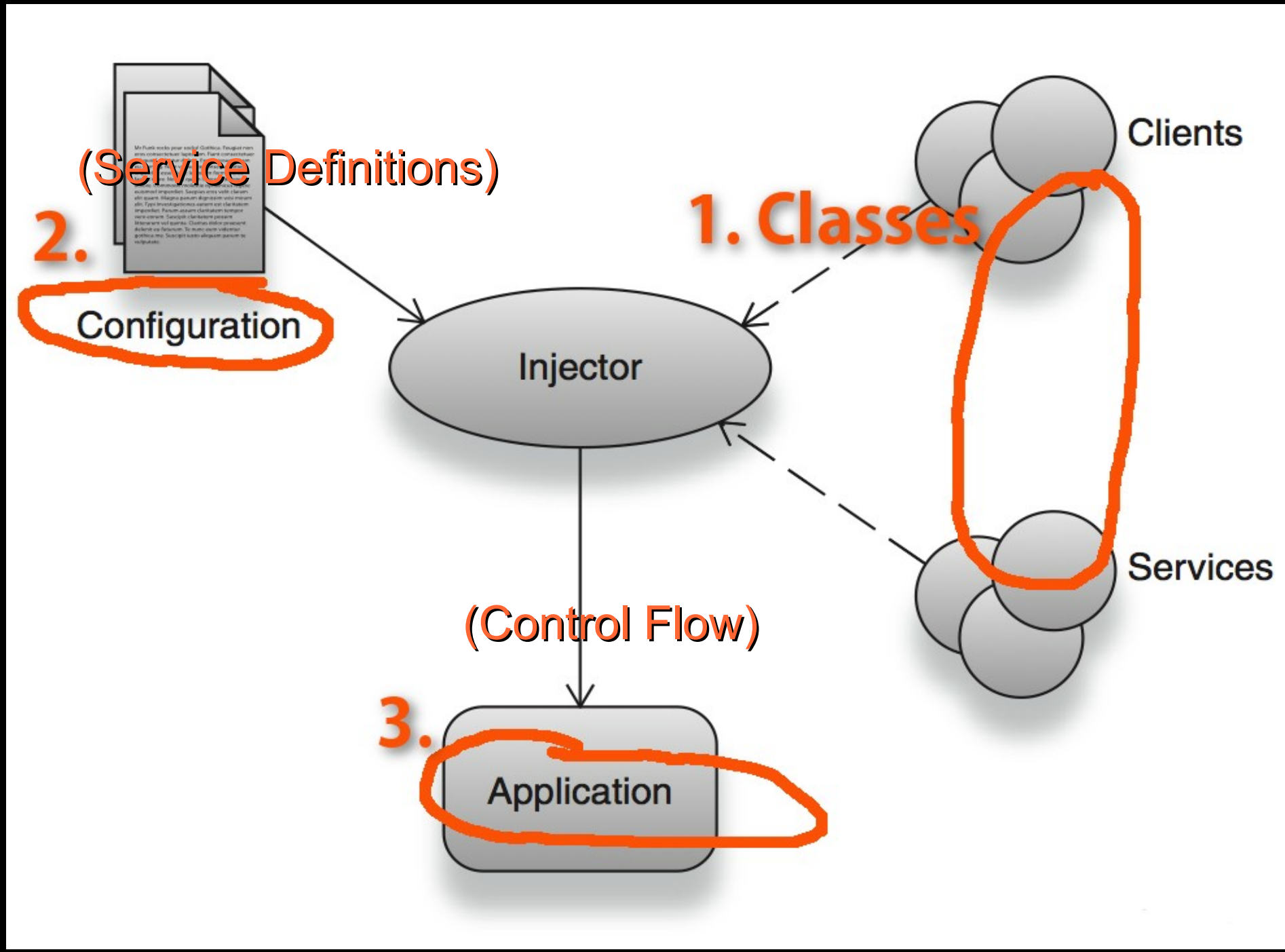
- Cache Backend
- Logger
- Mailer
- URL Generator

Examples of Non-Services

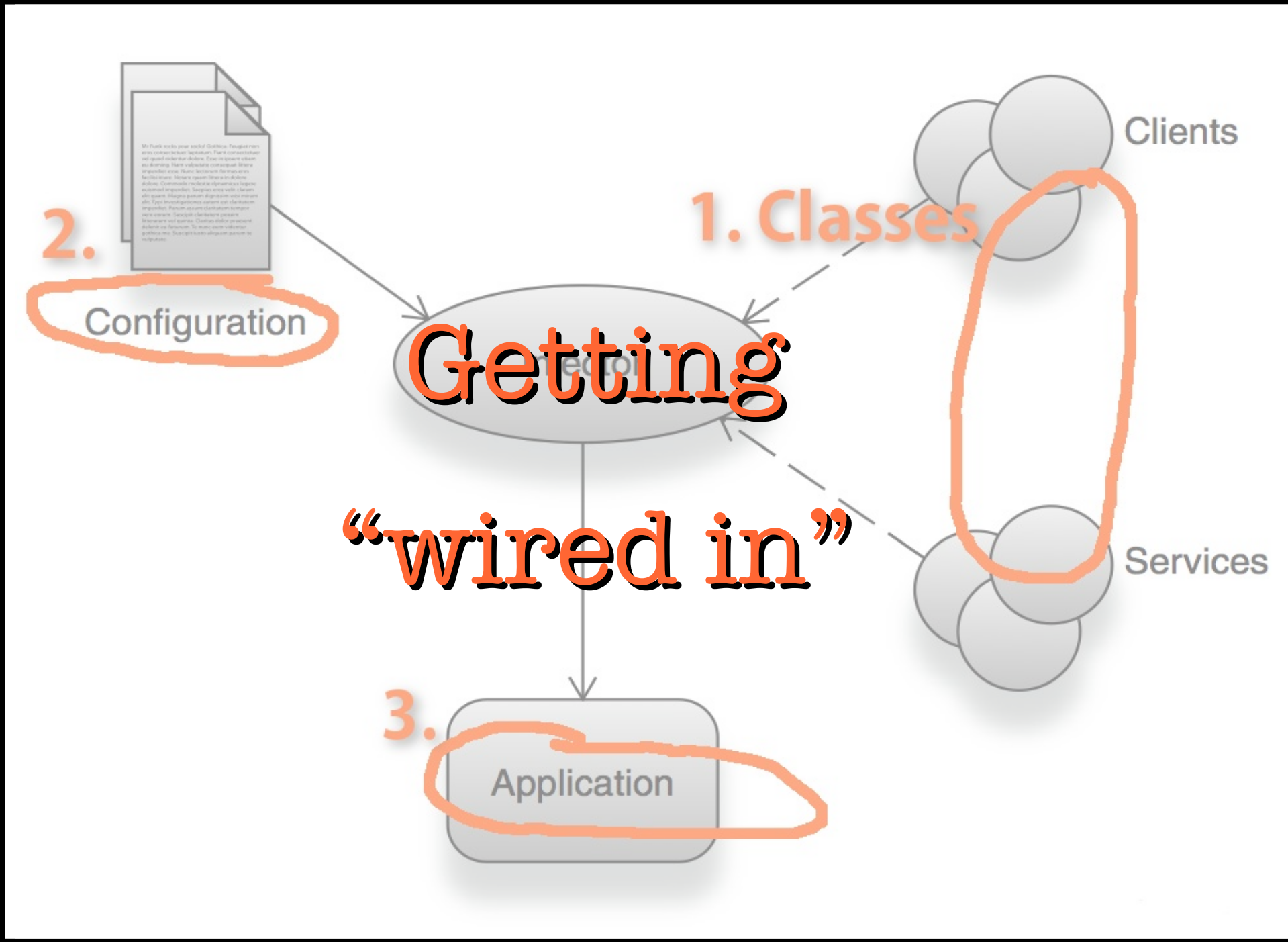
- Product
- Blog post
- Email message



Source: Dependency Injection by Dhanji R. Prasanna, published by Manning



Source: Dependency Injection by Dhanji R. Prasanna, published by Manning



Source: Dependency Injection by Dhanji R. Prasanna, published by Manning

Sample configuration

```
<services...>
  <service id="notifier" class="Notifier">
    <constructor-arg ref="emailer" />
  </service>
  <service id="emailer" class="Mailer">
    <constructor-arg ref="spell_checker" />
  </service>
  <service id="spell_checker"
       class="SpellChecker" />
</services>
```

How does it work?

- Service keys map to service definitions
- Definitions specify which class to instantiate and what its dependencies are
- Dependencies are specified as references to other services (using service keys)
- $container->getService('some_service')

Scope

The scope of a service is the context under which the service key refers to the same *instance*.

Symfony's Dependency Injection Component

Symfony's DI Component

- Service keys are strings, e.g. 'some_service'
- Service definitions, in addition to specifying which class to instantiate and what constructor arguments to pass in, allow you to specify additional methods to call on the object after instantiation

Symfony's DI Component

- Default scope: container
- Can be configured in PHP, XML or YAML
- Can be “compiled” down to plain PHP

Some Symfony Terminology

“Compiling” the container

It's too expensive to parse configuration on every request.

Parse once and put the result into a PHP class that hardcodes a method for each service.

“Compiling” the container

```
Class service_container extends Container {
  /**
   * Gets the 'example' service.
   */
  protected function getExampleService()
  {
    return $this->services['example'] = new
\Some\Namespace\SomeClass();
  }
}
```

“Synthetic” Services

A synthetic service is one that is not instantiated by the container – the container just gets told about it so it can then treat it as a service when anything has a dependency on it.

Compiler passes

Compiler passes are classes that process the container, giving you an opportunity to manipulate existing service definitions.

Use them to:

- Specify a different class for a given service id
- Process “tagged” services

Tagged Services

You can add tags to your services when you define them. This flags them for some kind of special processing (in a compiler pass).

For example, this mechanism is used to register event subscribers (services tagged with 'event_subscriber') to Symfony's event dispatcher

Bundles

Bundles are Symfony's answer to plugins or modules, i.e. prepackaged sets of functionality implementing a particular feature, e.g. a blog.

Each bundle includes a class implementing the BundleInterface which allows it to interact with the container, e.g. to add compiler passes.

Symfony's Event Dispatcher plays an important role in the application flow.

Symfony's Event Dispatcher

- Dispatcher dispatches events such as Kernel::Request
- Can be used to dispatch any kind of custom event
- Event listeners are registered to the dispatcher and notified when an event fires
- Event subscribers are classes that provide multiple event listeners for different events

Symfony's Event Dispatcher

- A compiler pass registers all subscribers to the dispatcher, using their service IDs
- The dispatcher holds a reference to the service container
- Can therefore instantiate “subscriber services” with their dependencies

A compiler pass iterates over the tagged services

```
class RegisterKernelListenersPass implements
    CompilerPassInterface {

  public function process(ContainerBuilder $container) {

    $definition = $container
      ->getDefinition('event_dispatcher');

    $services = $container
      ->findTaggedServiceIds('event_subscriber');
    foreach ($services as $id => $attributes) {
      $definition->addMethodCall('addSubscriberService',
array($id, $class));
    }
  }
}
```

Register the compiler pass

```
class CoreBundle extends Bundle {


  public function build(ContainerBuilder $container)
  {
    ...
    // Compiler pass for event subscribers.
    $container->addCompilerPass(new
        RegisterKernelListenersPass());
    ...
  }
}
```

Dependency Injection in Drupal 8

Some D8 Services

- The default DB connection ('database')
- The module handler ('module_handler')
- The HTTP request object ('request')

Services:

```
  database:
    class: Drupal\Core\Database\Connection
    factory_class: Drupal\Core\Database\Database
    factory_method: getConnection
    arguments: [default]

  path.alias_whitelist:
    class: Drupal\Core\Path\AliasWhitelist
    tags:
      - { name: needs_destruction }

  language_manager:
    class: Drupal\Core\Language\LanguageManager

  path.alias_manager:
    class: Drupal\Core\Path\AliasManager
    arguments: ['@database',
      '@path.alias_whitelist', '@language_manager']
```

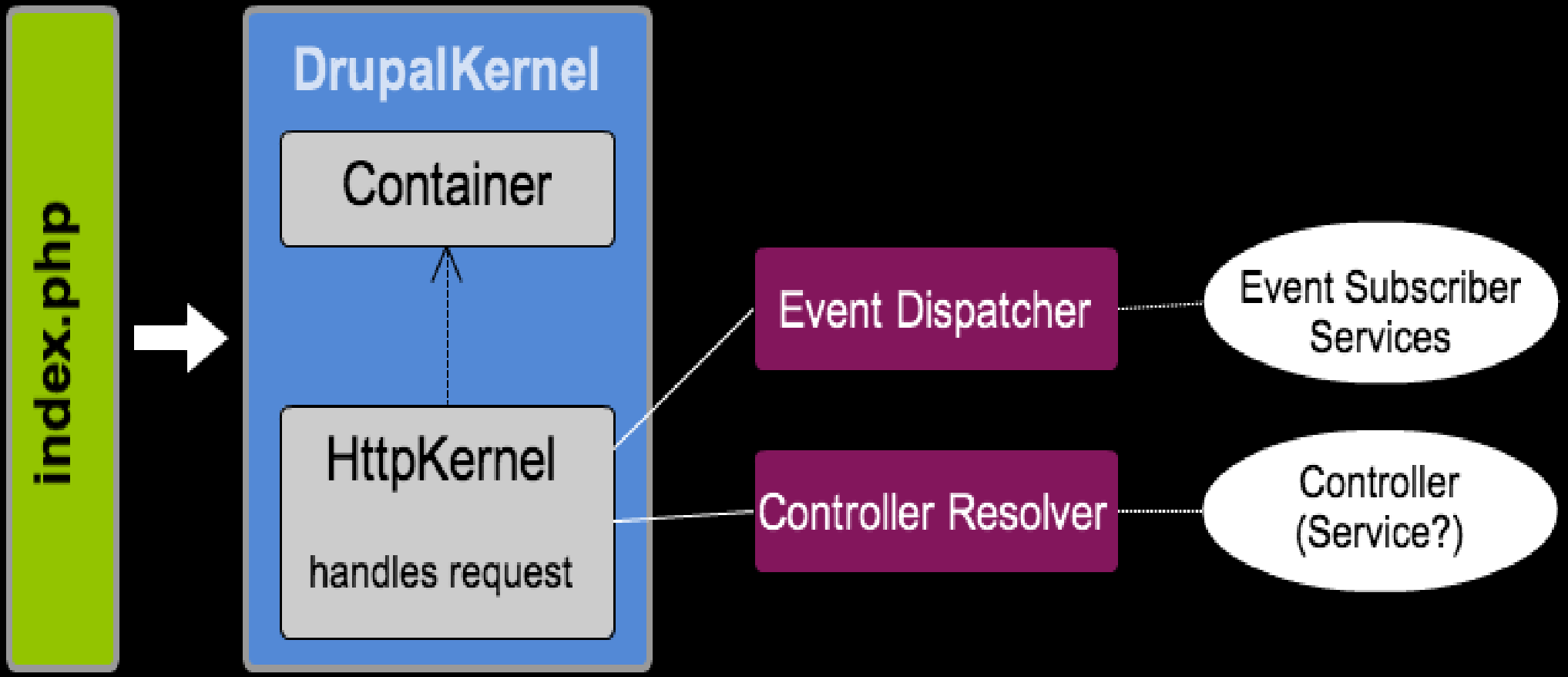
AliasManager's Constructor

```
class AliasManager implements AliasManagerInterface {
  ...
  public function __construct(Connection $connection,
      AliasWhitelist $whitelist, LanguageManager
      $language_manager) {
    $this->connection = $connection;
    $this->languageManager = $language_manager;
    $this->whitelist = $whitelist;
  }
...
}
```

2 ways you can use core's services

1. From procedural code, using a helper: Drupal::service('some_service')
2. Write OO code and get wired into the container

Drupal's Application Flow



Get wired in as an event subscriber

1. Implement EventSubscriberInterface

```
class MySubscriber implements
                        EventSubscriberInterface {

  static function getSubscribedEvents() {
    $events[KernelEvents::REQUEST][] =
        array('onKernelRequest', 200);
    return $events;
  }

  public function onKernelRequest(GetResponseEvent
$event) {
    ...
  }

}
```

2. Write a service definition and add the 'event_subscriber' tag

```
Services:

  ...

  my_subscriber:
    class: Drupal\mymodule\MySubscriber
    tags:
      - { name: event_subscriber }

  ...
```

How to get your controller wired in?

Controllers as Services?

- Controllers have dependencies on services
- Whether they should be directly wired into the container is a hotly debated topic in the Symfony community
- Recommended way in D8 is not to make controllers themselves services but to implement a special interface that has a factory method which accepts the container
- See book module for an example!

Don't inject the container!
Ever.
(Unless you absolutely must)

Where does it all happen?

- The Drupal Kernel:
  core/lib/Drupal/Core/DrupalKernel.php
- Services are defined in:
  core/core.services.yml
- Compiler passes get added in:
  core/lib/Drupal/Core/CoreBundle.php
- Compiler pass classes live here:
  core/lib/Drupal/Core/DependencyInjection/Compiler/...

What about modules?

- Services are defined in: *mymodule/mymodule.services.yml*
- Compiler passes get added in: *mymodule/lib/Drupal/mymodule/MymoduleBundle.php*
- All classes, including compiler pass classes, must live in *mymodule/lib/Drupal/mymodule/*

Easy testability with DI and PHPUnit

PHPUnit Awesomeness

```
// Create a language manager stub.
$language_manager = $this
  ->getMock('Drupal\Core\Language\LanguageManager');


$language_manager->expects($this->any())
  ->method('getLanguage')
  ->will($this->returnValue((object) array(
    'langcode' => 'en',
    'name' => 'English')));
```

PHPUnit Awesomeness

```
// Create an alias manager stub.
$alias_manager = $this
  ->getMockBuilder('Drupal\Core\Path\AliasManager')
  ->disableOriginalConstructor()
  ->getMock();

$alias_manager->expects($this->any())
  ->method('getSystemPath')
  ->will($this->returnValueMap(array(
    'foo' => 'user/1',
    'bar' => 'node/1',
  )));
```

Resources

These slides and a list of resources on DI and its use in Symfony and Drupal are available at

http://katbailey.github.io/

Questions?

DRUPALCON

Portland

MAY 20-24 2013

What did you think?

Evaluate this session at:
portland2013.drupal.org/schedule.

Thank you!

Building Bridges, Connecting Communities