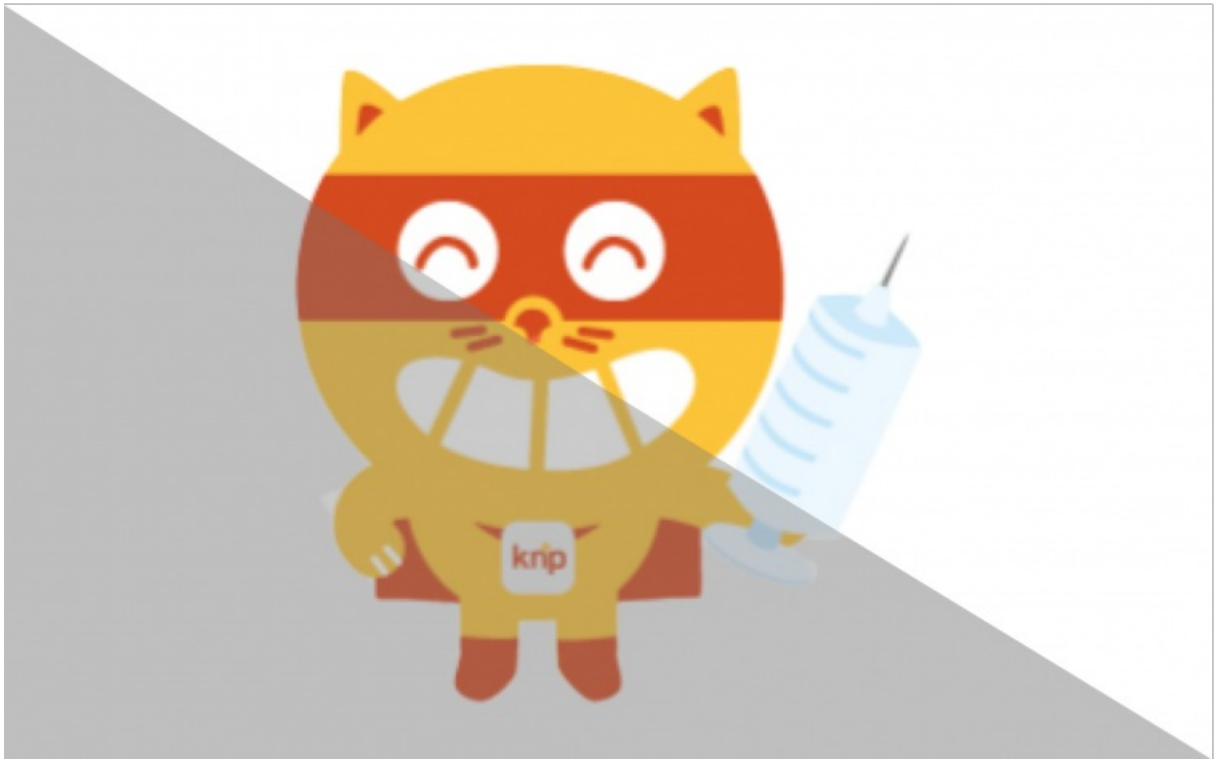
Dependency Injection and the art of services and containers



With <3 from SymfonyCasts

Chapter 1: Dependency Injection

Hi guys! In this tutorial, we're going to talk about dependency injection, services, and dependency injection containers by looking at a simple one called Pimple. The great news is that understanding these things isn't hard, but it can dramatically increase the quality and maintainability of the code you write.

As always, we'll be coding with a real example. Recently, we noticed that a lot of really nice rich people have been emailing us trying to give away their money. In this tutorial, we're going to create a simple app to help these fine people, we're calling it SendMoneyToStrangers.com.

I've already bootstrapped a small app, which you can download. It uses an Sqlite database, so make sure you have it installed, then `chmod 777` the data directory and run a script that creates some dummy data for us:

```
$ chmod -R 777 data/
$ php data/setupDb.php
```

The app is really simple:

25 lines | app.php

```
... lines 1 - 2
require __DIR__.'/vendor/autoload.php';

use DiDemo\Mailer\SmtpMailer;

$dsn = 'sqlite:'.__DIR__.'/data/database.sqlite';
$pdo = new PDO($dsn);

$mailer = new SmtpMailer('smtp.SendMoneyToStrangers.com', 'smtpuser', 'smtppass', '465');

$sql = 'SELECT * FROM people_to_spam';
foreach ($pdo->query($sql) as $row) {
    $mailer->sendMessage(
        $row['email'],
        'Yay! We want to send you money for no reason!',
        sprintf(<<<EOF
Hi %s! We've decided that we want to send you money for no reason!

Please forward us all your personal information so we can make a deposit and don't ask any questions!
EOF
        , $row['name']),
        'YourTrustedFriend@SendMoneyToStrangers.com'
    );
}
```

It queries the database, then delivers emails to each person using some `SmtpMailer` class:

60 lines | src/DiDemo/Mailer/SmtpMailer.php

```
... lines 1 - 2
namespace DiDemo\Mailer;

/**
 * Sends emails via SMTP
 */
class SmtpMailer
{
    private $hostname;

    private $user;

    private $pass;

    private $port;

    public function __construct($hostname, $user, $pass, $port)
    {
        $this->hostname = $hostname;
        $this->user = $user;
        $this->pass = $pass;
        $this->port = $port;
    }

... lines 26 - 33
    public function sendMessage($recipientEmail, $subject, $message, $from)
    {
... lines 36 - 58
    }
}
```

You could use any mailer library here, and I've made this class fake the sending of emails for simplicity. Instead, it just logs details to a file:

60 lines | src/DiDemo/Mailer/SmtpMailer.php

```
... lines 1 - 33
    public function sendMessage($recipientEmail, $subject, $message, $from)
    {
        // dummy implementation - this class is just used as an example

        // hack - just log something so we can see it
        $logPath = __DIR__.'/../../../logs/mail.log';
        $logLines = array();
        $logLines[] = sprintf(
            '[%s][%s:%s@%s:%s][From: %s][To: %s][Subject: %s]',
            date('Y-m-d H:i:s'),
            $this->user,
            $this->pass,
            $this->hostname,
            $this->port,
            $from,
            $recipientEmail,
            $subject
        );
        $logLines[] = '---------------';
        $logLines[] = $message;
        $logLines[] = '---------------';

        $fh = fopen($logPath, 'a');
        fwrite($fh, implode("\n", $logLines)."\n");
        // end hack
    }
```

Tip

We're using Composer for autoloading files in our `src/` directory with the following `composer.json`:

11 lines | composer.json

```
{
... lines 2 - 6
    "autoload": {
        "psr-4": { "": "src/" }
    }
}
```

Tail the log file:

```
$ tail -f logs/mail.log
```

Then run the app via `php app.php` from the command line:

```
$ php app.php
```

You'll see two emails are sent to two lucky people.

Chapter 2: Services and Dependency Injection

Our app is small now, but as it grows, the `app.php` file will get harder and harder to read. The best way to fix this is to separate each different chunk of functionality into different PHP classes and methods. Each of these classes is called a "service" and the whole idea is sometimes called Service-Oriented Architecture.

Create a new file in `src/DiDemo` called `FriendHarvester.php`, which will be responsible for sending the email to every lucky person in the database:

```
11 lines | src/DiDemo/FriendHarvester.php
... lines 1 - 2
namespace DiDemo;

class FriendHarvester
{
... lines 7 - 10
}
```

Add the namespace so that it follows the directory structure and give it an `emailFriends` method:

```
11 lines | src/DiDemo/FriendHarvester.php
... lines 1 - 6
    public function emailFriends()
    {

    }
```

Copy in all of our logic into this new method:

```
28 lines | src/DiDemo/FriendHarvester.php
... lines 1 - 10
        $mailer = new SmtpMailer('smtp.SendMoneyToStrangers.com', 'smtpuser', 'smtppass', '465');

        $sql = 'SELECT * FROM people_to_spam';
        foreach ($pdo->query($sql) as $row) {
            $mailer->sendMessage(
                $row['email'],
                'Yay! We want to send you money for no reason!',
                sprintf(<<<EOF
Hi %s! We've decided that we want to send you money for no reason!

Please forward us all your personal information so we can make a deposit and don't ask any questions!
EOF
                , $row['name']),
                'YourTrustedFriend@SendMoneyToStrangers.com'
            );
        }
... lines 27 - 28
```

Go Deeper!

To learn more about PHP namespaces, check out our free PHP Namespaces in 120 Seconds tutorial

Tip

The namespace follows the directory structure so the the class is automatically autoloaded by Composer's autoloader. For more on how this all works, see Autoloading in PHP and the PSR-0 Standard.

And just like that, you've created your first service! Roughly speaking, a service is any PHP class that performs an

action. Since this sends emails to our new soon-to-be-rich friends, it's a service.

Tip

An example of a PHP class that's *not* a service would be something that simply holds data, like a `Blog` class, with `title`, `author` and `body` fields. These are sometimes called "Model objects".

The `app.php` code gets pretty simple now: just instantiate the `FriendHarvester` and call the method:

13 lines | app.php

```
... lines 1 - 5
use DiDemo\FriendHarvester;
... lines 7 - 10
$friendHarvester = new FriendHarvester();
$friendHarvester->emailFriends();
```

But when we try it:

```
$ php app.php
```

We get a huge error!

Once we've moved the code, we don't have access to the PDO object anymore. So how can we get it?

Accessing External Objects from a Service

This is our first important crossroads. There are a few cheating ways to do this, like using the dreaded global keyword:

30 lines | src/DiDemo/FriendHarvester.php

```
... lines 1 - 8
    public function emailFriends()
    {
        // NOOOOOOOO!!!!
        global $pdo;
... lines 13 - 28
    }
```

Don't use this. You could also make the `$pdo` variable available statically, by creating some class and then reference it:

19 lines | app.php

```
... lines 1 - 10
class Registry
{
    static public $pdo;
}
Registry::$pdo = $pdo;

$friendHarvester = new FriendHarvester();
... lines 18 - 19
```

30 lines | src/DiDemo/FriendHarvester.php

```
... lines 1 - 8
    public function emailFriends()
    {
        // Still NOOOOOOOO!!!!
        $pdo = \Registry::$pdo;
... lines 13 - 28
    }
```

The problem with both approaches is that our `FriendHarvester` has to assume the `$pdo` variable has actually been set and is available. Or to say it differently, when you use this class, you need to make sure any global or static

variables it needs are setup. And the only way to know what the class needs is to scan the file looking for global or static variable calls. This makes `FriendHarvester` harder to understand and maintain, and much harder to test.

Our Friend Dependency Injection

Let's get rid of all of that and do this right.

Since `FriendHarvester` needs the PDO object, add a `__construct()` method with it as the first argument. Set the value to a new private property and update our code to use it:

35 lines | src/DiDemo/FriendHarvester.php

```
... lines 1 - 6
class FriendHarvester
{
    private $pdo;

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

    public function emailFriends()
    {
... lines 18 - 20
        foreach ($this->pdo->query($sql) as $row) {
... lines 22 - 32
        }
    }
}
```

The `FriendHarvester` now makes a lot of sense: whoever instantiates it *must* pass us a `$pdo` variable. Inside this class, we don't care *how* this will happen, we just know that it will, and we can make use of it.

Tip

You can also type-hint the argument, which is a great practice. We'll talk more about this later:

```
public function __construct(\PDO $pdo)
```

This very simple idea is called Dependency Injection, and you just nailed it! Dependency injection means that if a class needs an object or some configuration, we force that information to be passed into that class, instead of reaching outside of it by using a global or static variable.

Back in `app.php`, we now need to explicitly pass the PDO object when instantiating the `FriendHarvester`:

13 lines | app.php

```
... lines 1 - 10
$friendHarvester = new FriendHarvester($pdo);
... lines 12 - 13
```

Run it:

```
$ php app.php
```

Everything works exactly like before, except that we've moved our logic into a service, which makes it testable, reusable, and much more understandable for two reasons.

First, the class and method names (`FriendHarvester::emailFriends()`) serve as documentation for what our code does. Second, because we're using dependency injection, it's clear what our service might do, because we can see what outside things it needs.

Chapter 3: Injecting Config & Services and using Interfaces

We've already created our first service *and* used dependency injection, we're even closer to getting this money out! One problem with the `FriendHarvester` is that we've hardcoded the SMTP configuration inside of it:

```
35 lines | src/DiDemo/FriendHarvester.php
... lines 1 - 15
    public function emailFriends()
    {
        $mailer = new SmtpMailer('smtp.SendMoneyToStrangers.com', 'smtpuser', 'smtppass', '465');
... lines 19 - 33
    }
```

What if we want to re-use this class with a different configuration? Or what if our beta and production setups use different SMTP servers? Right now, both are impossible!

Injecting Configuration

When we realized that `FriendHarvester` needed the PDO object, we injected it via the constructor. The same rule applies to configuration. Add a second constructor argument, which will be an array of SMTP config and update the code to use it:

```
43 lines | src/DiDemo/FriendHarvester.php
... lines 1 - 6
class FriendHarvester
{
    private $pdo;

    private $smtpConfig;

    public function __construct($pdo, array $smtpConfig)
    {
        $this->pdo = $pdo;
        $this->smtpConfig = $smtpConfig;
    }

    public function emailFriends()
    {
        $mailer = new SmtpMailer(
            $this->smtpConfig['server'],
            $this->smtpConfig['user'],
            $this->smtpConfig['password'],
            $this->smtpConfig['port']
        );
... lines 27 - 41
    }
}
```

Back in `app.php`, pass the array when creating `FriendHarvester`:

```
18 lines | app.php
... lines 1 - 10
$friendHarvester = new FriendHarvester($pdo, array(
    'server' => 'smtp.SendMoneyToStrangers.com',
    'user' => 'smtpuser',
    'password' => 'smtppass',
    'port' => '465'
));
... lines 17 - 18
```

When we try it:

```
$ php app.php
```

It still works! Our class is more flexible now, but, let's level up again!

Injecting the Whole Mailer

We can now configure the `FriendHarvester` with different SMTP settings, but what if we wanted to change how mails are sent entirely, like from SMTP to sendmail? And what if we needed to use the mailer object somewhere else in our app? Right now, we would need to create it anywhere we need it, since it's buried inside `FriendHarvester`.

In fact, `FriendHarvester` doesn't really care *how* we're sending emails, it only cares that it has an `SmtpMailer` object so that it can call `sendMessage()`. So like with the `PDO` object, it's a dependency. Refactor our class to pass in the whole `SmtpMailer` object instead of just its configuration:

36 lines | src/DiDemo/FriendHarvester.php

```
... lines 1 - 6
class FriendHarvester
{
    private $pdo;

    private $mailer;

    public function __construct($pdo, $mailer)
    {
        $this->pdo = $pdo;
        $this->mailer = $mailer;
    }

    public function emailFriends()
    {
... line 21
        foreach ($this->pdo->query($sql) as $row) {
            $this->mailer->sendMessage(
... lines 24 - 32
            );
        }
    }
}
```

Update `app.php` to create the mailer object:

20 lines | app.php

```
... lines 1 - 10
$mailer = new SmtpMailer(
    'smtp.SendMoneyToStrangers.com',
    'smtpuser',
    'smtppass',
    '465'
);

$friendHarvester = new FriendHarvester($pdo, $mailer);
... lines 19 - 20
```

Try it out to make sure it still works:

```
$ php app.php
```

We would hate for our friends to miss this opportunity!

Once again, this makes the `FriendHarvester` even more flexible and readable, and will also make re-using the mailer possible. As a general rule, it's almost always better to inject a service into another than to create it internally.

When you're in a service, think twice before using the `new` keyword, unless you're instantiating a simple object that exists just to hold data as opposed to doing some job (i.e. a "model object").

Type-Hinting

One thing we've neglected to do is type-hint our two constructor arguments. Let's do it now:

```
36 lines | src/DiDemo/FriendHarvester.php
... lines 1 - 6
class FriendHarvester
{
... lines 9 - 12
    public function __construct(\PDO $pdo, SmtpMailer $mailer)
... lines 14 - 35
}
```

This is totally optional, but has a bunch of benefits. First, if you pass something else in, you'll get a much clearer error message. Second, it documents the class even further. A developer now knows exactly what methods she can call on these objects. And third, if you use an IDE, this gives you auto-completion! Type-hinting is optional, but I highly recommend it.

Adding an Interface

Right now we're injecting an `SmtpMailer`. But in reality, `FriendHarvester` only cares that the mailer has a `sendMessage()` method on it. But even if we had another class with an identical method, like `SendMailMailer`, for example, we couldn't use it because of the specific type-hint.

To make this more awesome, create a new `MailerInterface.php` file, which holds an interface with the single send method that all mailers must have:

```
8 lines | src/DiDemo/Mailer/MailerInterface.php
... lines 1 - 2
namespace DiDemo\Mailer;

interface MailerInterface
{
    public function sendMessage($recipientEmail, $subject, $message, $from);
}
```

Update `SmtpMailer` to implement the interface and change the type-hint in `FriendHarvester` as well:

```
60 lines | src/DiDemo/Mailer/SmtpMailer.php
... lines 1 - 7
class SmtpMailer implements MailerInterface
{
... lines 10 - 59
}
```

```
36 lines | src/DiDemo/FriendHarvester.php
... lines 1 - 4
use DiDemo\Mailer\MailerInterface;

class FriendHarvester
{
... lines 9 - 12
    public function __construct(\PDO $pdo, MailerInterface $mailer)
    {
... lines 15 - 16
    }
... lines 18 - 35
}
```

When you're finished, try the application again:

```
$ php app.php
```

Everything should still work just fine. And with any luck you will find a place for all of that annoying money.

Just like with every step so far, this has a few great advantages. First, `FriendHarvester` is more flexible since it now accepts any object that implements `MailerInterface`. Second, it documents our code a bit more. It's clear now exactly what small functionality `FriendHarvester` actually needs. Finally, in `SmtpMailer`, the fact that it implements an interface with a `sendMessage()` method tells us that this method is particularly important. The class could have other methods, but `sendMessage()` is probably an especially important one to focus on.

Chapter 4: Dependency Injection Container

Our project now has services, an interface, and is fully using dependency injection. Nice work! One of the downsides of DI is that all the complexity of creating and configuring objects is now your job. This isn't so bad since it all happens in one place and gives you so much control, but it is something we can improve!

If you want to make this easier, the tool you need is called a dependency injection container. A lot of DI containers exist in PHP, but let's use Composer to grab the simplest one of all, called Pimple. Add a `require` key to `composer.json` to include the library:

12 lines | composer.json

```
{
... lines 2 - 3
    "require": {
... line 5
        "pimple/pimple": "1.0.*"
    },
... lines 8 - 10
}
```

Make sure you've downloaded Composer, and then run `php composer.phar install` to download Pimple.

Go Deeper!

If you're new to Composer, check out our free The Wonderful World of Composer Tutorial.

Pimple is both powerful, and tiny. Kind of like having one on prom night. It is just a single file taking up around 200 lines. That's one reason I love it!

Create a new Pimple container. This is an object of course, but it looks and acts like an array that we store all of our service objects on:

24 lines | app.php

```
... lines 1 - 7
$container = new Pimple();
... lines 9 - 24
```

Start by adding the `SmtpMailer` object under a key called `mailer`. Instead of setting it directly, wrap it in a call to `share()` and in an anonymous function. We'll talk more about this in a second, but just return the mailer object from the function for now:

24 lines | app.php

```
... lines 1 - 9
$container['mailer'] = $container->share(function() {
    return new SmtpMailer(
        'smtp.SendMoneyToStrangers.com',
        'smtpuser',
        'smtppass',
        '465'
    );
});
... lines 18 - 24
```

To access the `SmtpMailer` object, use the array syntax again:

```
24 lines | app.php
... lines 1 - 21
$friendHarvester = new FriendHarvester($pdo, $container['mailer']);
... lines 23 - 24
```

It's that simple! Run the application to spam... I mean send great opportunities to our friends!

```
$ php app.php
```

Shared and Lazy Services

We haven't fully seen the awesomeness of the container yet, but there are already some cool things happening. First, wrapping the instantiation of the `mailer` service in an anonymous function makes its creation "lazy":

```
24 lines | app.php
... lines 1 - 9
$container['mailer'] = $container->share(function() {
... lines 11 - 16
});
... lines 18 - 24
```

This means that the object isn't created until much later when we reference the `mailer` service and ask the container to give it to us. And if we never reference `mailer`, it's never created at all - saving us time and memory.

Second, using the `share()` method means that no matter how many times we ask for the `mailer` service, it only creates it once. Each call returns the original object:

```
$mailer1 = $container['mailer'];
$mailer2 = $container['mailer'];

// there is only 1 mailer, the 2 variables hold the same one
$willBeTrue = $mailer1 === $mailer2;
```

Tip

The `share()` method is deprecated and removed since Pimple 2.0. Now, you simply need to use bare anonymous functions instead of wrapping them with `share()`:

```
$container['session'] = function() {
    return new Session();
};
```

This is a very common property of a service: you only ever need just one. If we need to send many emails, we don't need many mailers, we just need the one and then we'll call `send()` on it many times. This also makes our code faster and less memory intensive, since the container guarantees that we only have one mailer. This is another detail that we don't need to worry about.

Now witness the Geek-Awesomeness of this fully armed and operational Container!

Let's keep going and add our other services to the container. But first, I'll add some comments to separate which part of our code is building the container, and which part is our actual application code:

28 lines | app.php

```
... lines 1 - 7
/* START BUILDING CONTAINER */

$container = new Pimple();

$container['mailer'] = $container->share(function() {
    return new SmtpMailer(
        'smtp.SendMoneyToStrangers.com',
        'smtpuser',
        'smtppass',
        '465'
    );
});

$dsn = 'sqlite:'.__DIR__.'/data/database.sqlite';
$pdo = new PDO($dsn);

/* END CONTAINER BUILDING */
... lines 25 - 28
```

Let's add `FriendHarvester` to the container next:

32 lines | app.php

```
... lines 1 - 20
$container['friend_harvester'] = $container->share(function() {
    return new FriendHarvester($pdo, $container['mailer']);
});
... lines 24 - 32
```

That's easy, except that we somehow need access to the `PDO` object and the container itself so we can get two required dependencies. Fortunately, the anonymous function is passed an argument, which is the Pimple container itself:

35 lines | app.php

```
... lines 1 - 20
$container['friend_harvester'] = $container->share(function(Pimple $container) {
    return new FriendHarvester($container['pdo'], $container['mailer']);
});
... lines 24 - 35
```

To fix the missing `PDO` object, just make it a service as well:

35 lines | app.php

```
... lines 1 - 24
$container['pdo'] = $container->share(function() {
    $dsn = 'sqlite:'.__DIR__.'/data/database.sqlite';

    return new PDO($dsn);
});
... lines 30 - 35
```

Now we can easily update the `friend_harvester` service configuration to use it:

35 lines | app.php

```
... lines 1 - 20
$container['friend_harvester'] = $container->share(function(Pimple $container) {
    return new FriendHarvester($container['pdo'], $container['mailer']);
});
... lines 24 - 35
```

With the new `friend_harvester` service, update the application code to just grab it out of the container:

35 lines | app.php

```
... lines 1 - 32
$friendHarvester = $container['friend_harvester'];
$friendHarvester->emailFriends();
```

Now that all three of our services are in the container, you can start to see the power that this gives us. All of the logic of exactly which objects depend on which other object is abstracted away into the container itself. Whenever we need to use a service, we just reference it: we don't care how it's created or what dependencies *it* may have, it's all handled elsewhere. And if the constructor arguments for a service like the `mailer` change later, we only need to update one spot in our code. Nobody else knows or cares about this change.

Remember also that the services are constructed lazily. When we ask for the `friend_harvester`, the `pdo` and `mailer` services haven't been instantiated yet. Fortunately, the container is smart enough to create them first, and then pass them into the `FriendHarvester` constructor. All of that happens automatically, behind the scenes.

Configuration

But a container can hold more than just services, it can house our configuration as well. Create a new key on the container called `database.dsn`, set it to our configuration, and then use it when we're creating the PDO object:

35 lines | app.php

```
... lines 1 - 11
$container['database.dsn'] = 'sqlite:'.__DIR__.'/data/database.sqlite';
... lines 13 - 26
$container['pdo'] = $container->share(function(Pimple $container) {
    return new PDO($container['database.dsn']);
});
... lines 30 - 35
```

We're not using the `share()` method or the anonymous function because this is just a scalar value, and we don't need to worry about that lazy-loading stuff.

We can do the same thing with the SMTP configuration parameters. Notice that the name I'm giving to each of these parameters isn't important at all, I'm just inventing a sane pattern and using the name where I need it:

39 lines | app.php

```
... lines 1 - 11
$container['database.dsn'] = 'sqlite:'.__DIR__.'/data/database.sqlite';
$container['smtp.server'] = 'smtp.SendMoneyToStrangers.com';
$container['smtp.user'] = 'smtpuser';
$container['smtp.password'] = 'smtp';
$container['smtp.port'] = 465;

$container['mailer'] = $container->share(function(Pimple $container) {
    return new SmtpMailer(
        $container['smtp.server'],
        $container['smtp.user'],
        $container['smtp.password'],
        $container['smtp.port']
    );
});
... lines 26 - 39
```

When we're all done, the application works exactly as before. What we've gained is the ability to keep all our configuration together. This would make it very easy to change our database to use MySQL or change the SMTP password.

Move Configuration into a Separate File

Now that we have this flexibility, let's move the configuration and service building into separate files altogether. Create a new `app/` directory and `config.php` and `services.php` files. Require each of these from the `app.php` script right after creating the container:

16 lines | app.php

```
... lines 1 - 4
/* START BUILDING CONTAINER */

$container = new Pimple();

require __DIR__.'/app/config.php';
require __DIR__.'/app/services.php';

/* END CONTAINER BUILDING */
... lines 13 - 16
```

Next, move the configuration logic into `config.php` and all the services into `services.php`. Be sure to update the SQLite database path in `config.php` since we just moved this file:

7 lines | app/config.php

```
... lines 1 - 2
$container['database.dsn'] = 'sqlite:'.__DIR__.'/../data/database.sqlite';
$container['smtp.server'] = 'smtp.SendMoneyToStrangers.com';
$container['smtp.user'] = 'smtpuser';
$container['smtp.password'] = 'smtp';
$container['smtp.port'] = 465;
```

21 lines | app/services.php

```
... lines 1 - 2
use DiDemo\Mailer\SmtpMailer;
use DiDemo\FriendHarvester;

$container['mailer'] = $container->share(function(Pimple $container) {
    return new SmtpMailer(
        $container['smtp.server'],
        $container['smtp.user'],
        $container['smtp.password'],
        $container['smtp.port']
    );
});

$container['friend_harvester'] = $container->share(function(Pimple $container) {
    return new FriendHarvester($container['pdo'], $container['mailer']);
});

$container['pdo'] = $container->share(function(Pimple $container) {
    return new PDO($container['database.dsn']);
});
```

Skinny Controllers and Service-Oriented Architecture

Awesome! We now have configuration, service-building and our actual application code all separated into different files. Notice how clear our actual app code is now - it's just one line to get out a service and another to use it.

If this were a web application, this would live in a controller. You'll often hear that you should have "skinny controllers" and a "fat model". And whether you realize it or not, we've just seen that in practice! When we started, `app.php` held all of our logic. After refactoring into services and using a service container, `app.php` is skinny. The "fat model" refers to moving all of your logic into separate, single-purpose classes, which are sometimes referred to collectively as "the model". Another term for this is service-oriented architecture.

In the real world, you may not always have skinny controllers, but always keep this philosophy in your mind. The skinnier your controllers, the more readable, reusable, testable and maintainable that code will be. What's better, a 300 line long chunk of code or 5 lines that use a few well-named and small service objects?

Auto-completion with a Container

One of the downsides to using a container is that your IDE and other developers don't exactly know what type of object a service may be. There's no perfect answer to this, since a container is very dynamic by nature. But what you *can* do is use PHP documentation whenever possible to explicitly say what type of object something is.

For example, after fetching the `friend_harvester` service, you can use a single-line comment to tell your IDE and other developers exactly what type of object we're getting back:

19 lines | app.php

```
... lines 1 - 15
/** @var FriendHarvester $friendHarvester */
$friendHarvester = $container['friend_harvester'];
$friendHarvester->emailFriends();
```

This gives us IDE auto-complete on the `$friendHarvester` variable. Another common tactic is to create an object or sub-class the container and add specific methods that return different services and have proper PHPDoc on them. I won't show it here, but imagine we've sub-classed the `Pimple` class and added a `getFriendHarvester()` method which has a proper `@return` PHPDoc on it.

Chapter 5: A Container in your Project

Ok, time to get to emailing! No matter what framework or system you work on, you can start applying these principles immediately. You may already have a dependency injection container available to you, and if so, great! If not, don't worry! Even without a container, you can start applying the principles of moving code into new service classes and using dependency injection. If you have to instantiate these service objects manually when you need them, that's still a huge step forward!

You can also bring a container into your project. Pimple is the simplest and easiest, but there are also others such as Symfony's DependencyInjection Component, Aura Di, and Zend\Di. These are more feature-rich and also contain speed optimizations.

Somewhere early in your bootstrap process, simply create the container, configure it, and make it available to your controllers or page code.

If you have any questions or comments, post them! Have fun, and we'll see you next time!