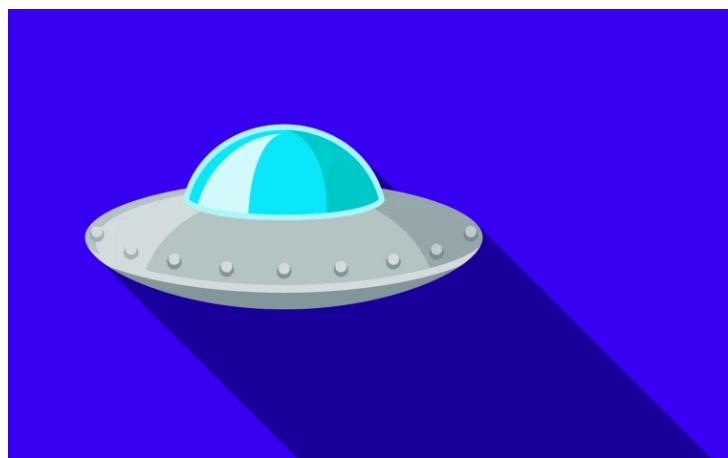


OOP (Course 2): Services, Dependency Injection and Containers



Chapter 1: Service Classes

Well hey! Welcome back! It's time to put our new object-oriented skills into practice. We're working on the same out of this world project: it has ships, you choose them, then they engage in epic battle!

In an editor, far far away, you'll see a simple application that runs this: `index.php` is the homepage and `battle.php` does the magic and shows the results. Last time, we created a single class called `Ship`, which describes all its properties - it's like a container for one ship's details:

```
lib/Ship.php
1 // ... lines 1 - 2
2
3 class Ship
4 {
5     private $name;
6
7     private $weaponPower = 0;
8
9     private $jediFactor = 0;
10
11    private $strength = 0;
12
13    private $underRepair;
14
15 // ... lines 14 - 115
16 }
```

We used this to replace these big associative arrays. Now we deal with cute `Ship` objects:

functions.php

```
↔ // ... lines 1 - 4
5 function get_ships()
6 {
7     $ships = array();
8
9     $ship = new Ship('Jedi Starfighter');
10    // $ship->setName('Jedi Starfighter');
11    $ship->setWeaponPower(5);
12    $ship->setJediFactor(15);
13    $ship->setStrength(30);
14    $ships['starfighter'] = $ship;
↔ // ... lines 15 - 33
34     return $ships;
↔ // ... lines 35 - 126
```

Remove all the Flat Functions!

Having a huge list of flat functions in `functions.php` is not a good recipe for staying organized. But in just a few minutes, we'll use some new classes to give our app a whole new level of sophistication. We'll get rid of `battle()` first.

Look at `Ship`: this is a class that basically just holds data - some people call that "state", but I'll say "data" - and I'm talking about the values on a `Ship` object's properties. So a `Ship` object holds data, but it doesn't really do any work. Sure, it has some methods on it, but these just return that data, after doing some small logic at best.

Reason #1 for creating a class is this: we need some organized unit to hold data.

But there's a second big reason to create a class: because you need to do some work. For example, in `functions.php`, the `battle()` function *does* work: we give it 2 Ships, it does some calculations, executes logic to see how different strengths affect each other and ultimately returns the result of that work.

And we're all familiar with creating functions like this. And here's the secret for OO: whenever you get the urge to create a flat function like `battle()`, don't. Instead, create a class and with a method inside of it.

Create the BattleManager Service Class

Let's do this! Since this function is all about battling, let's create a new class called

BattleManager:

lib/BattleManager.php

```
1 <?php
2
3 class BattleManager
4 {
5 // ... lines 5 - 57
58 }
```

Be as creative as you want with naming: I want to describe that methods in this class will do things related to battling.

Go copy and remove the flat `battle()` function: paste it into **BattleManager**. Put `public` in front of `function`. Remember, `public` means that code *outside* of this class will be able to call this:

lib/BattleManager.php

```
// ... lines 1 - 2
3 class BattleManager
4 {
5     /**
6      * Our complex fighting algorithm!
7      *
8      * @return array With keys winning_ship, losing_ship &
9      * used_jedi_powers
10     */
10    public function battle(Ship $ship1, $ship1Quantity, Ship $ship2,
11    $ship2Quantity)
11    {
12        // ... lines 12 - 51
13        return array(
14            'winning_ship' => $winningShip,
15            'losing_ship' => $losingShip,
16            'used_jedi_powers' => $usedJediPowers,
17        );
18    }
19 }
```

And yes, you don't *have* to add `public`: functions default to `public` if you say nothing, but let's keep things clear!

That's all you need to change: functions work the same inside or outside of a class: they have arguments, they return stuff.

But we do need to change code where we call this function - in `battle.php`. So how can we call this? Well, when we want to call a method on `Ship`, we need to have a `Ship` object first. The same is true here: we need a `BattleManager` object first. Start with a new variable called `$battleManager` and create a new `BattleManager` object:

```
battle.php
↑ // ... lines 1 - 28
29 $battleManager = new BattleManager();
↓ // ... lines 30 - 98
```

And now say `$battleManager`, the arrow, then `battle()`:

```
battle.php
↑ // ... lines 1 - 28
29 $battleManager = new BattleManager();
30 $outcome = $battleManager->battle($ship1, $ship1Quantity, $ship2,
    $ship2Quantity);
↓ // ... lines 31 - 98
```

Let's give this a shot! Refresh `battle.php`. Oh no! Class `BattleManager` not found! Epic fail!

Not really - at the top of `functions.php`, we have access to the `Ship` class because we're requiring it. Do the same for `BattleManager`:

```
functions.php
1 <?php
2
3 require_once __DIR__ . '/lib/Ship.php';
4 require_once __DIR__ . '/lib/BattleManager.php';
↓ // ... lines 5 - 73
```

There *is* a way where you can reference classes like `BattleManager` *without* needing to worry about the require statements. It's called autoloading, it's really common, and you'll learn how to master it in a future episode. But until then: if you have a class, `require` it.

Go back and refresh!

Cool - totally working.

Now we have 2 reasons to create a class. First, if you have some data - like properties that describe a ship, creating a class for that is nice. You'll create a `Ship` object whenever you have a set of that data. In `get_ships()`, we create 4 `Ship` objects. These types of classes are sometimes called models, because they model something, like a ship.

Second, if you need to make a function that does some work: create a class and put a method in it, like `BattleManager`. Or, you may put multiple methods inside one class - as long as they are all thematically similar.

You'll create one of these objects - like `BattleManager` - just one time, before you need to call a method on it. These are sometimes called service classes, because they perform work or service. Organizing your code to use service classes can be tricky, but we'll learn all about that.

Chapter 2: An Army of Service Classes

Yay! We got rid of a flat function. Woh - not so fast: inside `battle()`, we're *calling* a flat function: `didJediDestroyShipUsingTheForce()`:

```
lib/BattleManager.php
1 // ... lines 1 - 2
2
3 class BattleManager
4 {
5
6 // ... lines 5 - 9
7
8 public function battle(Ship $ship1, $ship1Quantity, Ship $ship2,
9 $ship2Quantity)
10
11 {
12 // ... lines 12 - 18
13
14         if (didJediDestroyShipUsingTheForce($ship1)) {
15 // ... lines 20 - 23
16
17         }
18 // ... lines 25 - 56
19
20     }
21
22 }
```

No bueno!

Refactoring to private Functions

This lives at the bottom of `functions.php`. In our app, this is *only* called from inside `battle()`, and since it obviously relates to battles, let's move it into `BattleManager`. Make it a `private function`:

```
lib/BattleManager.php
```

```
↔ // ... lines 1 - 2
3 class BattleManager
4 {
↔ // ... lines 5 - 58
59     private function didJediDestroyShipUsingTheForce(Ship $ship)
60     {
61         $jediHeroProbability = $ship->getJediFactor() / 100;
62
63         return mt_rand(1, 100) <= ($jediHeroProbability*100);
64     }
65 }
```

Why did I make it private? Well, do we need use this function from outside of this class? No - the only code using it is up in `battle()`, so this is a perfect candidate to be `private`.

Above in `battle()`, update the calls to be

`$this->didJediDestroyShipUsingTheForce()`. The "force" of our app is happy again:

```
lib/BattleManager.php
```

```
↔ // ... lines 1 - 2
3 class BattleManager
4 {
↔ // ... lines 5 - 9
10    public function battle(Ship $ship1, $ship1Quantity, Ship $ship2,
11                           $ship2Quantity)
12    {
13        // ... lines 12 - 18
14        if ($this->didJediDestroyShipUsingTheForce($ship1)) {
15            // ... lines 20 - 23
16            if ($this->didJediDestroyShipUsingTheForce($ship2)) {
17                // ... lines 26 - 29
18            }
19        }
20        // ... lines 31 - 56
21    }
22    // ... lines 58 - 64
23
24 }
```

Now, if someday we *did* want to use this function from outside of `BattleManager`, *then* we could change it to `public`. Ok, so why not just make everything `public` - isn't that more flexible? Yes, but making this private is *nice*: it means that if I want to change this function - add arguments or even change what it returns - I know that the *only* code that will be affected will be right inside this class. If it's public, who knows what code I might break in my app?

Start with `private`, make it `public` only if you need. The same rule goes for `protected` - something we'll talk about later with inheritance.

Let's make sure we didn't bust things. Refresh!

Yes!

Service 2: ShipLoader

In `functions.php`, only the flat `get_ships()` function remains. You guys know what do to: move it into a class!

Should we move it into `BattleManager`? No - it doesn't relate to battles. Instead, create a new class for this - how about `ShipLoader`:

```
lib/ShipLoader.php
1 // ... lines 1 - 2
2
3 class ShipLoader
4 {
5
6 // ... lines 5 - 36
7 }
```

Let's work our magic: go grab `get_ships()` and move it into `ShipLoader`. Remove the old commented code and make the function `public`. Also, rename it from `get_ships()` to `getShips()` - that's a more common naming standard for methods in a class:

```
lib/ShipLoader.php
1 // ... lines 1 - 2
2
3 class ShipLoader
4 {
5     public function getShips()
6     {
7         $ships = array();
8
9         $ship = new Ship('Jedi Starfighter');
10
11     }
12
13     return $ships;
14
15 }
16
17
18 }
```

Yep, that's great! Now we need to update the code that *calls* this function. But first, open `functions.php` and `require` the new `ShipLoader.php`:

```
functions.php
1 // ... lines 1 - 4
2
3
4
5 require_once __DIR__.'/lib/ShipLoader.php';
```

`getShips()` is used in `battle.php` and `index.php` - start there. To call the method, create a `$shipLoader` variable and create a new `ShipLoader()` object. Now, just `$shipLoader->getShips()`:

```
index.php
1 <?php
2 require __DIR__.'/functions.php';
3
4 $shipLoader = new ShipLoader();
5 $ships = $shipLoader->getShips();
6
7 // ... lines 6 - 119
```

Do the same thing in `battle.php`:

```
battle.php
1 <?php
2 require __DIR__.'/functions.php';
3
4 $shipLoader = new ShipLoader();
5 $ships = $shipLoader->getShips();
6
7 // ... lines 6 - 99
```

I think it's time to try it. Click to create a new battle. Looks pretty good. Setup a new battle and, Engage. Ok! `battle.php` works too!

No More `functions.php`

AND, all the flat functions are gone! Object-orient all the things! So if you look in `functions.php`, well, there aren't any functions here: just `require` statements, and even those we'll get rid of eventually. To celebrate, give this a more appropriate name: `bootstrap.php`. Update this in `battle.php`:

```
battle.php
```

```
1 <?php
2 require __DIR__. '/bootstrap.php';
3 // ... lines 3 - 99
```

and `index.php`:

```
index.php
```

```
1 <?php
2 require __DIR__. '/bootstrap.php';
3 // ... lines 3 - 119
```

Refresh once more! Let's keep going.

Chapter 3: Sharpening the Battle Result with a Class

The most obvious time you should create a class is when you are passing around an associative array of data. Check out the `battle()` function: it returns an associative array - with `winning_ship`, `losing_ship` and `used_jedi_powers` keys:

```
lib/BattleManager.php
1 // ... lines 1 - 2
2
3 class BattleManager
4 {
5 // ... lines 5 - 9
6
7     public function battle(Ship $ship1, $ship1Quantity, Ship $ship2,
8     $ship2Quantity)
9     {
10 // ... lines 12 - 51
11
12         return array(
13             'winning_ship' => $winningShip,
14             'losing_ship' => $losingShip,
15             'used_jedi_powers' => $usedJediPowers,
16         );
17     }
18 // ... lines 58 - 64
19
20 }
```

We use this in `battle.php`, set it to an `$outcome` variable, then reference all those keys to print stuff further down:

```

battle.php
  ↑ // ... lines 1 - 30
31 $outcome = $battleManager->battle($ship1, $ship1Quantity, $ship2,
  ↑ $ship2Quantity);
  ↑ // ... lines 32 - 77
78             <?php if ($outcome['winning_ship'] == null): ?>
79                 Both ships destroyed each other in an epic battle
  to the end.
80             <?php else: ?>
81                 The <?php echo $outcome['winning_ship']-
  >getName(); ?>
82             <?php if ($outcome['used_jedi_powers']): ?>
83                 used its Jedi Powers for a stunning victory!
84             <?php else: ?>
85                 overpowered and destroyed the <?php echo
  $outcome['losing_ship']->getName() ?>s
86             <?php endif; ?>
87             <?php endif; ?>
  ↑ // ... lines 88 - 99

```

Ah man, I *hate* this kind of stuff. It's not obvious at all what's inside this `$outcome` variable or whether the keys it has now might be missing or different in the future. When you see questionable code like this, you need to be thinking: this is perfect for a class.

Creating the BattleResult Model Class

Let's create one! Now, what to call this new class. Well, this information summarizes a battle result - let's use that - a new class called `BattleResult`:

```

lib/BattleResult.php
  ↑ // ... lines 1 - 2
3 class BattleResult
4 {
  ↑ // ... lines 5 - 14
15 }

```

Ok, let's think about this: it'll need to hold data for the winning ship, the losing ship and whether jedi powers were used. So, let's create 3 private properties called `$usedJediPowers`, `$winningShip` and `$losingShip`:

lib/BattleResult.php

```
↔ // ... lines 1 - 2
3 class BattleResult
4 {
5     private $usedJediPowers;
6     private $winningShip;
7     private $losingShip;
↔ // ... lines 8 - 14
15 }
```

Look at `Ship`: our other model-type class that holds data. There are two ways we can set the data. One way is by making a `__construct()` function. Here, we're saying: "Hey, when you create a new `Ship` object, you need to pass in the name as an argument":

lib/Ship.php

```
↔ // ... lines 1 - 2
3 class Ship
4 {
5     private $name;
↔ // ... lines 6 - 14
15     public function __construct($name)
16     {
17         $this->name = $name;
↔ // ... lines 18 - 19
20     }
↔ // ... lines 21 - 115
116 }
```

For the other properties, we created public functions - like `setStrength()`, `setWeaponPower()` and `getJediFactor()`:

lib/Ship.php

```
↔ // ... lines 1 - 2
3 class Ship
4 {
↔ // ... lines 5 - 36
37     public function setStrength($number)
38     {
39         if (!is_numeric($number)) {
40             throw new \Exception('Strength must be a number, duh!');
41         }
42
43         $this->strength = $number;
44     }
↔ // ... lines 45 - 100
101    /**
102     * @param int $weaponPower
103     */
104    public function setWeaponPower($weaponPower)
105    {
106        $this->weaponPower = $weaponPower;
107    }
108
109    /**
110     * @param int $jediFactor
111     */
112    public function setJediFactor($jediFactor)
113    {
114        $this->jediFactor = $jediFactor;
115    }
↔ // ... lines 116 - 117
```

Both ways are fine - but I like to use the `__construct()` strategy for any properties that are required. You *must* give your ship a name - it doesn't make sense to have a nameless Ship fighting battles. How will they know who to write songs about?

A `BattleResult` only makes sense with *all* of this information - that's perfect for setting via the constructor! Create a new `public function __construct()` with `$usedJediPowers`, `$winningShip` and `$losingShip`. These argument names don't need to match the properties, it's just nice. Now, assign each property to that variable:
`$this->usedJediPowers = $usedJediPowers`,
`$this->winningShip = $winningShip` and `$this->losingShip = $losingShip`:

```
lib/BattleResult.php
1 // ... lines 1 - 2
2
3 class BattleResult
4 {
5     private $usedJediPowers;
6     private $winningShip;
7     private $losingShip;
8
9     public function __construct($usedJediPowers, $winningShip,
10                                $losingShip)
11    {
12        $this->usedJediPowers = $usedJediPowers;
13        $this->winningShip = $winningShip;
14        $this->losingShip = $losingShip;
15    }
16}
```

Ok, this little data wrapper is done.

Passing BattleResult around

So let's use it inside `battle()`: instead of returning that array, return a new `BattleResult` and pass it `$usedJediPowers`, `$winningShip` and `$losingShip`:

```
lib/BattleManager.php
1 // ... lines 1 - 2
2
3 class BattleManager
4 {
5
6 // ... lines 5 - 9
7
8     public function battle(Ship $ship1, $ship1Quantity, Ship $ship2,
9                           $ship2Quantity)
10    {
11
12 // ... lines 12 - 51
13
14        return new BattleResult($usedJediPowers, $winningShip,
15                                $losingShip);
16    }
17
18 // ... lines 54 - 60
19
20}
```

But hey, we're referencing a class, so make sure you require it in `bootstrap.php`:

```
bootstrap.php
1 // ... lines 1 - 5
2
3 require_once __DIR__ . '/lib/BattleResult.php';
```

So where is `battle()` being called? It's at the top of `battle.php` - and this `$outcome` variable used to be that associative array - now it's a fancy `BattleResult` object:

```
battle.php
↑ // ... lines 1 - 30
31 $outcome = $battleManager->battle($ship1, $ship1Quantity, $ship2,
    $ship2Quantity);
↓ // ... lines 32 - 99
```

This means that our code below - the stuff that treats `$outcome` like an array - should blow up.:

```
battle.php
↑ // ... lines 1 - 70
71             <?php if ($outcome['winning_ship']): ?>
72                 <?php echo $outcome['winning_ship']->getName(); ?>
73             <?php else: ?>
74                 Nobody
75             <?php endif; ?>
↓ // ... lines 76 - 99
```

Let's see some fireworks! Boom error!

```
Cannot use object of type BattleResult as array on line 71.
```

But we *do* need to get the winning ship from the `BattleResult` object. Is that possible right now? No - the `$winningShip` property is private. If we want to access it from outside the class, we need a *public* function that returns it for us. We did this same thing in `Ship` with methods like `getName()`.

Type-Hinting Arguments

But before we add some methods - think about the 3 arguments. What are they? Well, `$usedJediPowers` is a boolean and the other two are `Ship` objects. And whenever you have an argument that is an object, you can choose to type-hint it by putting the name of the class in front of it:

```
lib/BattleResult.php
1 // ... lines 1 - 2
2
3 class BattleResult
4 {
5 // ... lines 5 - 13
6
7     public function __construct($usedJediPowers, Ship $winningShip, Ship
8     $losingShip)
9     {
10         $this->usedJediPowers = $usedJediPowers;
11         $this->winningShip = $winningShip;
12         $this->losingShip = $losingShip;
13     }
14 // ... lines 20 - 43
15 }
16
17 }
```

But this doesn't change any behavior - it just means that if you pass something that's *not* a `Ship` object on accident, you'll get a really nice error. And there's one other benefit - auto-completion in your editor! PhpStorm now knows what these variables are.

Adding Getter Methods

Ok, back to what we were doing. We need to access the private properties from *outside* this class. To do that, we'll create some *public* functions. Start with

`public function getWinningShip()`. This will just `return $this->winningShip`:

```
lib/BattleResult.php
1 // ... lines 1 - 2
2
3 class BattleResult
4 {
5 // ... lines 5 - 31
6
7     public function getWinningShip()
8     {
9         return $this->winningShip;
10    }
11 // ... lines 36 - 43
12 }
13
14 }
```

We'll do this for *each* property. But actually, I can make PhpStorm write these methods for me! Suckers! Delete `getWinningShip()`, then right-click, go to "Generate" and select "Getters". Select all 3 properties, say abracadabra, and let it work its magic.

It even added some PHPDoc above each with an `@return mixed` - which basically is PhpStorms' way of saying "I don't know what this method returns". So let's help it - the first returns a `boolean` and the other two return a `Ship` object:

```
lib/BattleResult.php
1 // ... lines 1 - 2
2
3 class BattleResult
4 {
5
6 // ... lines 5 - 20
7
8
9 /**
10  * @return boolean
11 */
12 public function isUsedJediPowers()
13 {
14     return $this->usedJediPowers;
15 }
16
17 /**
18  * @return Ship
19 */
20 public function getWinningShip()
21 {
22     return $this->winningShip;
23 }
24
25 /**
26  * @return Ship
27 */
28 public function getLosingShip()
29 {
30     return $this->losingShip;
31 }
32
33 }
```

This comment stuff is optional - but it helps other developers read our code *and* gives us auto-completion when we call these methods.

Name the Methods Awesomely

Check out the first method - `getUsedJediPowers()`. Is it clear what the method returns? It's kind of bad English, and that's a shame. This method will return whether or not Jedi powers were used to win this battle. Let's give it a name that says that - how about `wereJediPowersUsed()`?

```
lib/BattleResult.php
1 // ... lines 1 - 2
2
3 class BattleResult
4 {
5
6 // ... lines 5 - 20
7
8 /**
9  * @return boolean
10 */
11
12 public function wereJediPowersUsed()
13 {
14
15     return $this->usedJediPowers;
16
17 }
18
19 // ... lines 28 - 43
20
21 }
```

Using `get` and then the method name is a good standard, but you can name these methods however you want.

Using BattleResult for Battle #Wins

Now we can *finally* go back to `battle.php` and start using these public methods. Start by renaming `$outcome` to `$battleResult` - it's more clear this is a `BattleResult` object:

```
battle.php
1 // ... lines 1 - 30
2
3 $battleResult = $battleManager->battle($ship1, $ship1Quantity, $ship2,
4 $ship2Quantity);
5
6 // ... lines 32 - 99
```

Below, use `$battleResult->getWinningShip()`:

```
battle.php
1 // ... lines 1 - 30
2
3 $battleResult = $battleManager->battle($ship1, $ship1Quantity, $ship2,
4 $ship2Quantity);
5
6 // ... lines 32 - 70
7
8 <?php if ($battleResult->getWinningShip()): ?>
9
10 // ... lines 72 - 99
```

Except, where's my auto-completion on that method? This will work, but PhpStorm is highlighting the method like it's wrong. It doesn't know that `$battleResult` is a `BattleResult` object.

Why? Look at `battle()`. We are returning a `BattleResult`, but oh no, the `@return` above this method still advertises that this method returns an array. Fix that with `@return BattleResult`:

```
lib/BattleManager.php
1 // ... lines 1 - 2
2
3 class BattleManager
4 {
5     /**
6      * Our complex fighting algorithm!
7      *
8      * @return BattleResult
9      */
10    public function battle(Ship $ship1, $ship1Quantity, Ship $ship2,
11                           $ship2Quantity)
12    {
13        //
14    }
15
16    // ... lines 12 - 52
17
18    // ... lines 54 - 60
19
20 }
```

Ok, now PhpStorm is acting friendly - the angry highlighting on the method is gone. Now update the other spots: `$battleResult->getWinningShip()->getName()`: thank you auto-complete. Use that same method once more, and in the `if` statement, use that nice `wereJediPowersUsed()` method. Finish with `$battleResult->getLosingShip()`:

battle.php

```
↔ // ... lines 1 - 70
71             <?php if ($battleResult->getWinningShip()): ?>
72                 <?php echo $battleResult->getWinningShip()->
73             >getName(); ?>
74                 <?php else: ?>
75                     Nobody
76                 <?php endif; ?>
↔ // ... lines 76 - 77
78             <?php if ($battleResult->getWinningShip() == null): ?>
79                 Both ships destroyed each other in an epic battle
80                 to the end.
81             <?php else: ?>
82                 The <?php echo $battleResult->getWinningShip()->
83             >getName(); ?>
84             <?php if ($battleResult->wereJediPowersUsed()): ?>
85                 used its Jedi Powers for a stunning victory!
86             <?php else: ?>
87                 overpowered and destroyed the <?php echo
$battleResult->getLosingShip()->getName() ?>s
88             <?php endif; ?>
89             <?php endif; ?>
↔ // ... lines 88 - 99
```

I think we're done. Refresh to try it! Ship it!

And gone are the days of needing to use weird associative arrays:

`BattleManager::battle()` returns a nice `BattleResult` object. And we're in full control of what public methods we put on that.

Chapter 4: Optional type-hinting & Semantic Methods

I need to show you something - so start another battle between some Jedi Star Fighters. It works... but if I refresh enough times... come on... yes! It blows up!

```
Argument 2 passed to BattleResult::__construct() must be an instance
of Ship, null given.
```

In `BattleResult` - because we're good programmers - we type-hinted the two `Ship` arguments. Buuuuut, if you look at the `battle()` function, there's a case where the ships can destroy each other. And when that happens, there is no winning or losing ship - they're both null. Since - news flash `null` is *not* a `Ship` object, PHP gets angry and casts down this big error.

When you type-hint an argument, the value *must* be that class - not even `null` is ok. But sometimes you *do* have a spot where an argument might be a specific object, or it might be null. To support this, make the argument optional - add an `= null` after it:

```
lib/BattleResult.php
1 // ... lines 1 - 2
2
3 class BattleResult
4 {
5
6 // ... lines 5 - 13
7
8     public function __construct($usedJediPowers, Ship $winningShip = null,
9     Ship $losingShip = null)
10    {
11        $this->usedJediPowers = $usedJediPowers;
12        $this->winningShip = $winningShip;
13        $this->losingShip = $losingShip;
14    }
15
16 // ... lines 20 - 53
17
18 }
```

I don't have to, but I'll update `@return` on the methods to be `Ship|null`:

```
lib/BattleResult.php
1 // ... lines 1 - 2
2
3 class BattleResult
4 {
5
6 // ... lines 5 - 36
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37 /**
38  * @return Ship|null
39 */
40 public function getLosingShip()
41 {
42     return $this->losingShip;
43 }
44
45 /**
46 * Was there a winner? Or did everybody die :(
47 */
48 /**
49 * @return bool
50 */
51 public function isThereAWinner()
52 {
53     return $this->getWinningShip() !== null;
54 }
```

PhpStorm will still give me auto-completion - but this is a signal to other developers not to blindly call this method and *always* assume it will return a `Ship` object. We're already coding safely in `battle.php`: we check to make sure `getWinningShip()` returns something before calling a method on it. Cool.

Adding a Semantic isThereAWinner Method

To check if a `BattleResult` has a winner, you can see if `getWinningShip()` returns null. But we can do even better. Go to `BattleResult` and make a new public method called `isThereAWinner()`. Here, return `$this->getWinningShip != null`:

```
lib/BattleResult.php
1 // ... lines 1 - 2
2
3 class BattleResult
4 {
5 // ... lines 5 - 43
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45 /**
46 * Was there a winner? Or did everybody die :(
47 *
48 * @return bool
49 */
50 public function isThereAWinner()
51 {
52     return $this->getWinningShip() !== null;
53 }
54 }
```

There's at least two great things about this. First, code outside of this class doesn't need to know *how* to figure out whether or not there was a winner: that code can be dumb and just call this method. Second, if something happens in the future and the logic used to figure out if there is a winner changes, we only need to update the code in this *one* spot: no need to run around the code base trying to figure out where we have the old logic for seeing if there was a winner.

Update `battle.php` to use this. The first `if` statement is *really* trying to figure out whether or not there was a winner. Update this to `$battleResult->isThereAWinner()`. Use that again right below:

```
battle.php
1 // ... lines 1 - 70
2
3 // ... line 71
4 <?php if ($battleResult->isThereAWinner()): ?>
5
6 // ... line 72
7 <?php else: ?>
8
9 // ... line 73
10 <?php endif; ?>
11
12 // ... lines 74 - 75
13 <?php if (! $battleResult->isThereAWinner()): ?>
14
15 // ... line 76
16 <?php else: ?>
17
18 // ... lines 77 - 80
19 <?php endif; ?>
20
21 // ... lines 81 - 87
22 <?php endif; ?>
23
24 // ... lines 88 - 99
```

Go back and refresh! You'll have to trust me that if we refresh this 1000 times, it'll always work - our bug is gone - and we have a nifty new helper method in `BattleResult`.

Chapter 5: Objects are Passed by Reference

Start another battle - how about 3 CloakShape fighters against 4 RZ-1 A-wing interceptors.

Behind the scenes: each ship has a `strength`. The `battle()` function uses this as the ship's health, and as they battle each other, that health gets lower and lower until one hits zero.

We need to add a new feature: after the battle: display the final health of the battling ships. One will be zero or negative, but how much health did the other have left?

In `battle()`, those "ship health" variables are *not* returned in `BattleResult`. So we *don't* have access to this information. We could add it to `BattleResult`, but I want to do something more interesting.

After fighting a battle, let's *update* the strength of each ship with their new health: like

`$ship1->setStrength($ship1Health)` and the same for `$ship2`:

```
lib/BattleManager.php
 1 // ... lines 1 - 9
 2
 3     public function battle(Ship $ship1, $ship1Quantity, Ship $ship2,
 4     $ship2Quantity)
 5     {
 6
 7         // ... lines 12 - 16
 8         while ($ship1Health > 0 && $ship2Health > 0) {
 9
10             // ... lines 18 - 31
11             // now battle them normally
12             $ship1Health = $ship1Health - ($ship2->getWeaponPower() *
13             $ship2Quantity);
14             $ship2Health = $ship2Health - ($ship1->getWeaponPower() *
15             $ship1Quantity);
16
17         }
18
19         // update the strengths on the ships, so we can show this
20         $ship1->setStrength($ship1Health);
21         $ship2->setStrength($ship2Health);
22
23     }
24
25     // ... lines 40 - 56
26
27 }
28
29 // ... lines 58 - 66
```

After all, in real life - if a `$ship` is almost defeated, it's probably pretty broken - so it's `$strength` should reflect that.

Check this out by dumping `$ship1->getStrength()` and `$ship2->getStrength()` and die. Refresh! We have -14 and 116, 130 and 0 and so on.

Ok, working nicely, and that's simple. Actually, we just did something really important. Until now, this function has only *read* data from our ships. But now, we've *changed* those objects. In other words, in `battle.php`, we start with two `Ship` objects and pass them into `battle()`:

```
battle.php
26 // ... lines 1 - 25
27 $ship1 = $ships[$ship1Name];
28 $ship2 = $ships[$ship2Name];
29
30 $battleManager = new BattleManager();
31 $battleResult = $battleManager->battle($ship1, $ship1Quantity, $ship2,
32 $ship2Quantity);
33 // ... lines 32 - 106
```

Once that finishes running, those *same* two objects are different now: their data has changed.

This is *totally* different than how arrays work: if `$ship1` were an array, and the `battle()` function changed one of its keys internally, that would have *no* effect here: `$ship1` would still be the same array with the same original values.

Objects are passed by reference: it means that there is only *one* `$ship1` object in existence and when we pass it to a function, we're passing that *one* object. But when you pass an array or a string to a function, you're actually passing a copy of the original value. If that value changes inside the function, it has no affect on the original variable.

Some of you may be familiar with adding an `&` symbol before an argument: this does the same thing: it makes that argument pass by reference. For objects, that's not needed, because this is *always* true.

The takeaway is that if you change an object, you're changing that object *everywhere*. To prove this, take our `$ship1` and `$ship2` - which are *not* returned by the `battle()` function - and add a new section that prints the finished strength. Add a `dl` element to make them a little pretty:

```
battle.php
34 // ... lines 1 - 33
34 <html>
34 // ... lines 35 - 53
54     <body>
55         <div class="container">
54 // ... lines 56 - 67
68             <div class="result-box center-block">
54 // ... lines 69 - 88
89                 <h3>Remaining Strength</h3>
90                 <dl class="dl-horizontal">
54 // ... lines 91 - 94
95                 </dl>
96             </div>
54 // ... lines 97 - 102
103         </div>
104     </body>
105 </html>
```

First, echo `$ship1->getName()` and then `$ship1->getStrength()`:

```
battle.php
34 // ... lines 1 - 33
34 <html>
34 // ... lines 35 - 53
54     <body>
55         <div class="container">
54 // ... lines 56 - 67
68             <div class="result-box center-block">
54 // ... lines 69 - 88
89                 <h3>Remaining Strength</h3>
90                 <dl class="dl-horizontal">
91                     <dt><?php echo $ship1->getName(); ?></dt>
92                     <dd><?php echo $ship1->getStrength(); ?></dd>
54 // ... lines 93 - 94
95                 </dl>
96             </div>
54 // ... lines 97 - 102
103         </div>
104     </body>
105 </html>
```

Do the same thing for `$ship2`:

```
battle.php
34 // ... lines 1 - 33
34 <html>
34 // ... lines 35 - 53
54     <body>
55         <div class="container">
54 // ... lines 56 - 67
68             <div class="result-box center-block">
54 // ... lines 69 - 88
89                 <h3>Remaining Strength</h3>
90                 <dl class="dl-horizontal">
91                     <dt><?php echo $ship1->getName(); ?></dt>
92                     <dd><?php echo $ship1->getStrength(); ?></dd>
93                     <dt><?php echo $ship2->getName(); ?></dt>
94                     <dd><?php echo $ship2->getStrength(); ?></dd>
95                 </dl>
96             </div>
54 // ... lines 97 - 102
103         </div>
104     </body>
105 </html>
```

We're missing auto-complete because we have some bad PHPDoc somewhere. We'll fix that in a bit.

Time to try it! Since objects are passed by reference, we should see the new, modified strength values - not the originals. Absolutely perfect.

Now let's get really wild and start fetching our ships from a database.

Chapter 6: Fetching Objects from the Database

Getting our Ship objects is easy: create a `ShipLoader` and call `getShips()` on it. We don't care *how* `ShipLoader` is getting these - that's *its* problem.

Hardcoding is so 1990, let's load objects from the database! We need to get these ships to their battlestations!

Database Setup

At the root of your project, open up a `resources` directory. Copy `init_db.php` out of there to the root of your project and open it up:

`init_db.php`

```
1 <?php
2
3 /*
4  * SETTINGS!
5 */
6 $databaseName = 'oo_battle';
7 $databaseUser = 'root';
8 $databasePassword = '';
9
10 /*
11  * CREATE THE DATABASE
12 */
13 $pdoDatabase = new PDO('mysql:host=localhost', $databaseUser,
14 $databasePassword);
15 $pdoDatabase->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
16 $pdoDatabase->exec('CREATE DATABASE IF NOT EXISTS oo_battle');
17 // ... lines 16 - 57
```

💡 Tip

In order for this to work, make sure that you have MySQL installed and running on your machine. There are various ways to install MySQL in different environments - if you have any questions, let us know in the comments!

This script will create a database and add a `ship` table with columns for `id`, `name`, `weapon_power`, `jedi_factor`, `strength` and `is_under_repair`:

```
init_db.php
↑ // ... lines 1 - 25
26 $pdo->exec('CREATE TABLE `ship` (
27   `id` int(11) NOT NULL AUTO_INCREMENT,
28   `name` varchar(255) COLLATE utf8mb4_unicode_ci NOT NULL,
29   `weapon_power` int(4) NOT NULL,
30   `jedi_factor` int(4) NOT NULL,
31   `strength` int(4) NOT NULL,
32   `is_under_repair` tinyint(1) NOT NULL,
33   PRIMARY KEY (`id`)
34 ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci');
↑ // ... lines 35 - 57
```

At the bottom, it inserts 4 rows into that table for the 4 ships we have hardcoded right now:

```
init_db.php
↑ // ... lines 1 - 38
39 $pdo->exec('INSERT INTO ship
40   (name, weapon_power, jedi_factor, strength, is_under_repair) VALUES
41   ("Jedi Starfighter", 5, 15, 30, 0)'
42 );
43 $pdo->exec('INSERT INTO ship
44   (name, weapon_power, jedi_factor, strength, is_under_repair) VALUES
45   ("CloakShape Fighter", 2, 2, 70, 0)'
46 );
47 $pdo->exec('INSERT INTO ship
48   (name, weapon_power, jedi_factor, strength, is_under_repair) VALUES
49   ("Super Star Destroyer", 70, 0, 500, 0)'
50 );
51 $pdo->exec('INSERT INTO ship
52   (name, weapon_power, jedi_factor, strength, is_under_repair) VALUES
53   ("RZ-1 A-wing interceptor", 4, 4, 50, 0)'
54 );
↑ // ... lines 55 - 57
```

If we run this file, it should get everything powered up. Head to your browser and run it there:

```
http://localhost:8000/init_db.php
```

If you see - Ding! - you know it worked. If you see a terrible error, check the database credentials at the top - make sure the user can create a new database.

If you want to check the database with something like phpMyAdmin, you'll see one `ship` table with 4 rows.

Querying for Ships

You look ready to query, copy the two lines that create the PDO object in `init_db` and head into `ShipLoader`. Keep things simple: `getShips()` needs to make a query. So for now, paste the PDO lines right here. Update the database name to be `oo_battle` and I'll fill in `root` as the user with no password:

```
lib/ShipLoader.php
```

```
1 // ... lines 1 - 2
2
3 class ShipLoader
4 {
5     public function getShips()
6     {
7         $pdo = new PDO('mysql:host=localhost;dbname=oo_battle', 'root');
8         $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
9
10    // ... lines 9 - 41
11
12    }
13
14 }
```

Ok, query time! Create a `$statement` variable and set it to `$pdo->prepare()` with the query inside - `SELECT * FROM ship`:

```
lib/ShipLoader.php
```

```
1 // ... lines 1 - 4
2
3     public function getShips()
4     {
5         $pdo = new PDO('mysql:host=localhost;dbname=oo_battle', 'root');
6         $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
7         $statement = $pdo->prepare('SELECT * FROM ship');
8
9
10    // ... lines 10 - 41
11
12    }
13
14 // ... lines 43 - 44
```

If PDO or prepared statements are new to you, don't worry - they're pretty easy. And besides, using PDO is another chance to play with objects!

Run `$statement->execute()` to send the query into hyperdrive and create a new `$shipsArray` that's set to `$statement->fetchAll()` with an argument:

`PDO::FETCH_ASSOC`. `var_dump` this variable:

lib/ShipLoader.php

```
↔ // ... lines 1 - 4
 5  public function getShips()
 6  {
 7      $pdo = new PDO('mysql:host=localhost;dbname=oo_battle', 'root');
 8      $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
 9      $statement = $pdo->prepare('SELECT * FROM ship');
10      $statement->execute();
11      $shipsArray = $statement->fetchAll(PDO::FETCH_ASSOC);
12      var_dump($shipsArray); die;
↔ // ... lines 13 - 41
42  }
↔ // ... lines 43 - 44
```

This queries for every row and returns an associative array. The `PDO::FETCH_ASSOC` part is a class constant - a nice little feature of classes we'll talk about later.

Let's see what this looks like! Head back to the homepage and refresh! AND... I was not expecting an error: "Unknown database `oo_battles`". The database *should* be called `oo_battle` - silly me! Refresh again!

Ok! 4 rows of data.

Private Functions are Awesome

Of course, what we *need* are objects, not arrays. But first, a quick piece of organization. Copy all this good `PDO` stuff and at the bottom, create a new

`private function queryForShips()`. Paste here and return that `$shipsArray`:

```
lib/ShipLoader.php
```

```
↔ // ... lines 1 - 2
3 class ShipLoader
4 {
↔ // ... lines 5 - 39
40     private function queryForShips()
41     {
42         $pdo = new PDO('mysql:host=localhost;dbname=oo_battle', 'root');
43         $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
44         $statement = $pdo->prepare('SELECT * FROM ship');
45         $statement->execute();
46         $shipsArray = $statement->fetchAll(PDO::FETCH_ASSOC);
47
48         return $shipsArray;
49     }
50 }
```

Head back up, call this method, then remove the original code:

```
lib/ShipLoader.php
```

```
↔ // ... lines 1 - 2
3 class ShipLoader
4 {
5     public function getShips()
6     {
7         $ships = array();
8
9         $shipsData = $this->queryForShips();
10        var_dump($shipsData);die;
↔ // ... lines 11 - 37
38    }
↔ // ... lines 39 - 49
50 }
```

Make sure things still work - cool! Now, why did we do this? Well, we had a chunk of code that did something - it made a query. Moving it into its own function has two advantages. First, we can re-use it later if we need to. But more importantly, it gives the code a name:

`queryForShips()`. Now it's easy to see what it does - a lot easier than when this was stuck in the middle of other code.

So, creating private functions to help split code into small chunks is awesome.

Give me Objects!

Back to the ship factory to create ship objects from the array we have now.

In `getShips()`, I'll rename the variable to `$shipsData` - it sounds cool to me. Now, loop over `$shipsData` as `$shipData`. Each time we loop, we'll create a `Ship` object: `$ship = new Ship()` and pass `$shipData['name']` as the only argument:

```
lib/ShipLoader.php
```

```
↔ // ... lines 1 - 4
 5     public function getShips()
 6     {
 7         $ships = array();
 8
 9         $shipsData = $this->queryForShips();
10
11         foreach ($shipsData as $shipData) {
12             $ship = new Ship($shipData['name']);
↔ // ... lines 13 - 17
18         }
↔ // ... lines 19 - 20
21     }
↔ // ... lines 22 - 33
```

Next, we can use the public functions to set the other data: `$ship->setWeaponPower()` and pass it `$shipData['weapon_power']`. Do the same for the `jedi_factor` and `strength` columns: `$ship->setJediFactor()` from the `jedi_factor` key and `$ship->setStrength()` from the `strength` key. The last column - `is_under_repair` we'll save that one for later. Can't have all the fun stuff at once! Finish the loop by putting `$ship` into the `$ships` array:

lib/ShipLoader.php

```
↔ // ... lines 1 - 4
5     public function getShips()
6     {
7         $ships = array();
8
9         $shipsData = $this->queryForShips();
10
11         foreach ($shipsData as $shipData) {
12             $ship = new Ship($shipData['name']);
13             $ship->setWeaponPower($shipData['weapon_power']);
14             $ship->setJediFactor($shipData['jedi_factor']);
15             $ship->setStrength($shipData['strength']);
16
17             $ships[] = $ship;
18         }
19
20         return $ships;
21     }
↔ // ... lines 22 - 33
```

Wasn't that easy? Now get rid of *all* of the hardcoded `Ship` objects. We have less code than we started. That's always my preference.

We've only changed this *one* file, but we're ready! Refresh! Welcome to our dynamic application in under 10 minutes. Ship it!

Chapter 7: Handling the Object Id

Ships are loading dynamically, buuuuuut, I've got some bad news: we broke our app. Start a battle - select the Jedi Starfighter as one of the ships and engage.

Huh, so instead of the results, we see:

Don't forget to select some ships to battle!

Pretty sure we selected a ship... But the URL has a `?error=missing_data` part, `index.php` is reading this. It all comes from `battle.php` and it happens if we POST here, but we are missing `ship1_name` or `ship2_name`. In other words, if we forget to select a ship. But we *did* select a ship! Somehow, these select menus are broken. Check out the code: we're looping over `$ships` and using `$key` as the option value:

```
index.php
// ... lines 1 - 90
91           <select class="center-block form-control btn drp-
dwn-width btn-default btn-lg dropdown-toggle" name="ship1_name">
// ... line 92
93           <?php foreach ($ships as $key => $ship): ?>
94           <?php if ($ship->isFunctional()): ?>
95           <option value="<?php echo $key; ?>"><?
96           php echo $ship->getNameAndSpecs(); ?></option>
97           <?php endif; ?>
98           <?php endforeach; ?>
// ... lines 99 - 119
```

In `getShips()`, the key was a nice, unique string. But now it's just the auto-increment index. The page fails because the 0 index looks like an empty string in `battle.php`.

Adding a Ship id Property

We *still* need something unique so that we can tell `battle.php` exactly which ships are fighting. Fortunately, the `ship` table has exactly that: an auto-incrementing primary key `id`

column. If we use this as the option value, we can query for the ships using that in `battle.php`. Blast off! I mean, we should totally do that.

In `ShipLoader`, we could put the `id` as the key of the array. But instead, since `id` is a column on the `ship` table, why not also make it a property on the `Ship` class? Open up `Ship` and add a new `private $id`:

```
lib/Ship.php
1 // ... lines 1 - 2
2
3 class Ship
4 {
5     private $id;
6
7 // ... lines 6 - 133
8
9 }
```

And at the bottom, right click, then make the getter and setter for the `id` property. Update the PHPDoc to show that `$id` is an integer. Optional, but nice:

```
lib/Ship.php
1 // ... lines 1 - 2
2
3 class Ship
4 {
5
6 // ... lines 5 - 118
7
8     /**
9      * @return int
10     */
11
12     public function getId()
13     {
14         return $this->id;
15     }
16
17     /**
18      * @param int $id
19      */
20
21     public function setId($id)
22     {
23         $this->id = $id;
24     }
25 }
```

Now when we get our `Ship` objects, we need to call `setId()` to populate that property: `$ship->setId()` and `$shipData['id']`

lib/ShipLoader.php

```
↔ // ... lines 1 - 2
3 class ShipLoader
4 {
5     public function getShips()
6     {
↔ // ... lines 7 - 10
11         foreach ($shipsData as $shipData) {
12             $ship = new Ship($shipData['name']);
13             $ship->setId($shipData['id']);
↔ // ... lines 14 - 17
18             $ships[] = $ship;
↔ // ... lines 19 - 21
22         }
↔ // ... lines 23 - 33
34 }
```

Head over to `index.php` to use the fancy new property. Remove the `$key` in the `foreach` - no need for that. And instead of the key, print `$ship->getId()`. Also change the `select` name to be `ship1_id` so we don't get confused about *what* this value is:

index.php

```
↔ // ... lines 1 - 90
91             <select class="center-block form-control btn drp-
dwn-width btn-default btn-lg dropdown-toggle" name="ship1_id">
↔ // ... line 92
93             <?php foreach ($ships as $ship): ?>
94                 <?php if ($ship->isFunctional()): ?>
↔ // ... lines 95 - 96
97                 <?php endforeach; ?>
98             </select>
↔ // ... lines 99 - 119
```

Make the same changes below: update the select name, remove `$key` from the loop, and finish with `$ship->getId()`:

```
index.php
  ↪ // ... lines 1 - 102
103           <select class="center-block form-control btn drp-
  ↪ dwn-width btn-default btn-lg dropdown-toggle" name="ship2_id">
  ↪ // ... line 104
105           <?php foreach ($ships as $ship): ?>
  ↪ // ... line 106
107           <option value="<?php echo $ship-
  ↪ >getId(); ?>"><?php echo $ship->getNameAndSpecs(); ?></option>
  ↪ // ... line 108
109           <?php endforeach; ?>
  ↪ // ... lines 110 - 119
```

Ok, before we touch battle, try this out. No errors! And the select items have values 1, 2, 3 and 4 - the auto-increment ids in the database. Success!

Querying for One Ship

We've renamed the `select` fields and we're sending a database id. Let's update `battle.php` for this. First, we need to change the `$_POST` keys: look for `ship1_id` and `ship2_id`. Update the variables names too - `$ship1Id` and `$ship2Id`. That'll help us not get confused. Update the variables in the first `if` statement

```
battle.php
  ↪ // ... lines 1 - 6
  7 $ship1Id = isset($_POST['ship1_id']) ? $_POST['ship1_id'] : null;
  ↪ // ... line 8
  9 $ship2Id = isset($_POST['ship2_id']) ? $_POST['ship2_id'] : null;
  ↪ // ... lines 10 - 11
12 if (!$ship1Id || !$ship2Id) {
13     header('Location: /index.php?error=missing_data');
14     die;
15 }
  ↪ // ... lines 16 - 106
```

Before, we got *all* the `$ships` then used the array key to find the right ones. That won't work anymore - the key is just an index, but we have the id from the database.

Instead, we can use that id to query for a single ship's data. Where should that logic live? In `ShipLoader`! It's *only* job is to query for ship information, so it's perfect.

Create a new public function `findOneById()` with an `$id` argument. Copy *all* the query logic from `queryForShips()` and put it here. For now don't worry about all this ugly code duplication. Update the query to be `SELECT * FROM ship WHERE id = :id` and pass that value to `execute()` with an array of `id` to `$id`:

```
lib/ShipLoader.php
1 // ... lines 1 - 2
2
3 class ShipLoader
4 {
5 // ... lines 5 - 23
6
7     public function findOneById($id)
8     {
9         $pdo = new PDO('mysql:host=localhost;dbname=oo_battle', 'root');
10        $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
11        $statement = $pdo->prepare('SELECT * FROM ship WHERE id = :id');
12        $statement->execute(array('id' => $id));
13
14 // ... lines 30 - 32
15
16     }
17
18 // ... lines 34 - 45
```

If this looks weird to you - it's a prepared statement. It runs a normal query, but prevents SQL injection attacks. Change the variable below to be `$shipArray` and change `fetchAll()` to just `fetch()` to return the *one* row. Dump this at the bottom:

```
lib/ShipLoader.php
1 // ... lines 1 - 23
2
3     public function findOneById($id)
4     {
5         $pdo = new PDO('mysql:host=localhost;dbname=oo_battle', 'root');
6         $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
7         $statement = $pdo->prepare('SELECT * FROM ship WHERE id = :id');
8         $statement->execute(array('id' => $id));
9         $shipArray = $statement->fetch(PDO::FETCH_ASSOC);
10
11         var_dump($shipArray);die;
12     }
13
14 // ... lines 34 - 45
```

Ok, back to `battle.php`! Let's use this. Now,

```
$ship1 = $shipLoader->findOneById($ship1Id). And
$ship2 = $shipLoader->findOneById($ship2Id). And I need to move this code
further up above the bad_ships error message. We'll use it in a second:
```

```
battle.php
```

```
16 // ... lines 1 - 16
17 $ship1 = $shipLoader->findOneById($ship1Id);
18 $ship2 = $shipLoader->findOneById($ship2Id);
19 // ... lines 19 - 106
```

Try it! Fight some Starfighters against a Cloakshape Fighter. There's the dump for just *one* row! Sweet, let's finish this!

Going from Array to Ship Object

The last step is to take this array and turn it into a `Ship` object. And good news! We've already done this in `getShips()`! And instead of repeating ourselves, this is another perfect spot for a `private function`. Create one called `createShipFromData` with an array `$shipData` argument:

```
lib/ShipLoader.php
```

```
16 // ... lines 1 - 2
17
18 class ShipLoader
19 {
20
21 // ... lines 5 - 32
22
23     private function createShipFromData(array $shipData)
24     {
25
26 // ... lines 35 - 41
27
28     }
29
30 // ... lines 43 - 54
31
32 }
33
34 // ... lines 56 - 57
```

Copy all the `new Ship()` code and paste it here. Return the `$ship` variable:

lib/ShipLoader.php

```
↔ // ... lines 1 - 32
33     private function createShipFromData(array $shipData)
34     {
35         $ship = new Ship($shipData['name']);
36         $ship->setId($shipData['id']);
37         $ship->setWeaponPower($shipData['weapon_power']);
38         $ship->setJediFactor($shipData['jedi_factor']);
39         $ship->setStrength($shipData['strength']);
40
41         return $ship;
42     }
↔ // ... lines 43 - 57
```

Now, anyone inside `ShipLoader` can call this, pass an array from the database, and get back a fancy new `Ship` object.

Back in `getShips()`, remove all that code and just use `$this->createShipFromData()`. Do the same thing in `findOneById()`:

lib/ShipLoader.php

```
↔ // ... lines 1 - 4
5     public function getShips()
6     {
↔ // ... lines 7 - 10
11         foreach ($shipsData as $shipData) {
12             $ships[] = $this->createShipFromData($shipData);
13         }
↔ // ... lines 14 - 15
16     }
↔ // ... line 17
18     public function findOneById($id)
19     {
20         $pdo = new PDO('mysql:host=localhost;dbname=oo_battle', 'root');
21         $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
22         $statement = $pdo->prepare('SELECT * FROM ship WHERE id = :id');
23         $statement->execute(array('id' => $id));
24         $shipArray = $statement->fetch(PDO::FETCH_ASSOC);
↔ // ... lines 25 - 29
30         return $this->createShipFromData($shipArray);
31     }
↔ // ... lines 32 - 57
```

In `battle.php`, `$ship1` and `$ship2` should now be `Ship` objects. The next if statement is a way to make sure that *valid* ship ids were passed: maybe someone is messing with our form!

With these tough ships in my database I should hope not.

I still want this check, so back in `ShipLoader`, add one more thing. If the `id` is invalid - like 10 or the word "pirate ship" - then `$shipArray` will be `null`. So, `if (!$shipArray)` then just `return null`:

```
lib/ShipLoader.php
1 // ... lines 1 - 17
2
3     public function findOneById($id)
4     {
5
6 // ... lines 20 - 23
7     $shipArray = $statement->fetch(PDO::FETCH_ASSOC);
8
9
10    if (!$shipArray) {
11        return null;
12    }
13
14    return $this->createShipFromData($shipArray);
15
16
17 // ... lines 32 - 57
```

The method now returns a `Ship` object or null. Back in `battle.php`, update the if to say if `!$ship1 || !$ship2`:

```
battle.php
1 // ... lines 1 - 16
2
3     $ship1 = $shipLoader->findOneById($ship1Id);
4     $ship2 = $shipLoader->findOneById($ship2Id);
5
6
7 if (!$ship1 || !$ship2) {
8     header('Location: /index.php?error=bad_ships');
9     die;
10}
11
12 // ... lines 24 - 106
```

And that should do it!

Go back and load the homepage fresh. And start a battle. When we submit, we'll be POST'ing these 2 ids to `battle.php`. And it works!

Thanks to `ShipLoader`, everyone is talking to the database, but nobody has to really worry about this.

PHPDoc for Autocomplete!

Let's fix one little thing that's bothering me. In `index.php`, we call `getShips()`. But when we loop over `$ships`, PhpStorm acts like all of the methods on the `Ship` object don't exist: `getName` not found in class.

If you look above `getShips()`, there's *no* PHP documentation. And so PhpStorm has *no* idea what this function returns. To fix that, add the `/**` above it and hit enter to generate some basic docs. Now it says `@return array`. That's true, but it doesn't tell it what's *inside* the array. Change it to `@return Ship[]`:

```
lib/ShipLoader.php
1 // ... lines 1 - 2
2
3 class ShipLoader
4 {
5     /**
6      * @return Ship[]
7      */
8     public function getShips()
9     {
10    // ... lines 10 - 18
11
12    }
13    // ... lines 20 - 61
14
15    }
16    // ... lines 63 - 64
```

This says: "I return an array of `Ship` objects". And when we loop over something returned by `getShips()`, we get happy code completion. Do the same thing above `findOneById()` - it returns just *one* `Ship` or null:

lib/ShipLoader.php

```
↔ // ... lines 1 - 2
3 class ShipLoader
4 {
↔ // ... lines 5 - 20
21     /**
22      * @param $id
23      * @return Ship
24      */
25     public function findOneById($id)
26     {
27         $pdo = new PDO('mysql:host=localhost;dbname=oo_battle', 'root');
28         $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
29         $statement = $pdo->prepare('SELECT * FROM ship WHERE id = :id');
30         $statement->execute(array('id' => $id));
31         $shipArray = $statement->fetch(PDO::FETCH_ASSOC);
32
33         if (!$shipArray) {
34             return null;
35         }
36
37         return $this->createShipFromData($shipArray);
38     }
↔ // ... lines 39 - 61
62 }
↔ // ... lines 63 - 64
```

Chapter 8: Making only one DB Connection with a Property

I can't stand it any longer. The app is small, but our database credentials are already duplicated *and* hidden inside this one class. What if we added a second table - like `battle` - and a `BattleLoader` class? At this rate, we'd be copying and pasting the database password *there* too. Gross.

Isolate the PDO Creation in ShipLoader

Enough is enough. Let's fix this little by little. First, I don't want to duplicate the `new PDO` code twice in this class. To fix that, create a `private function getPDO()` - private because - at least so far - we only want to call this from inside `ShipLoader`. Copy the `new PDO` line and the one below it and put them here. Return `$pdo` and let's even add some nice PHPDoc:

```
lib/ShipLoader.php
1 // ... lines 1 - 2
2
3 class ShipLoader
4 {
5 // ... lines 5 - 48
6
7     /**
8      * @return PDO
9      */
10     private function getPDO()
11     {
12         $pdo = new PDO('mysql:host=localhost;dbname=oo_battle', 'root');
13         $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
14
15         return $pdo;
16     }
17 // ... lines 59 - 67
18 }
19 // ... lines 69 - 70
```

You know what's next: use this above with: `$pdo = $this->getPDO()`. Repeat this in the other spot:

```
lib/ShipLoader.php
```

```
↔ // ... lines 1 - 2
3 class ShipLoader
4 {
↔ // ... lines 5 - 23
24     /*
25     public function findOneById($id)
26     {
27         $statement = $this->getPDO()->prepare('SELECT * FROM ship WHERE id
28 = :id');
↔ // ... lines 28 - 35
36     }
↔ // ... lines 37 - 59
60     private function queryForShips()
61     {
62         $statement = $this->getPDO()->prepare('SELECT * FROM ship');
↔ // ... lines 63 - 66
67     }
68 }
↔ // ... lines 69 - 70
```

Head back to the homepage! Ha! Nothing broken yet.

Prevent Multiple PDO Objects

Ok, a little bit better. Here's the next problem: what if a single page calls `findOneById()` multiple times? Well, `getPDO()` would be called twice, two `PDO` objects would be created and this would mean that two database connections would be made. Such waste! We only need one connection and we only need one `PDO` object.

How can we guarantee that only one `PDO` object is created?

By using a property! But in a way that we haven't seen yet. Up until now, we've only put properties on our model classes - like `Ship` - and that has been to hold data about the object, like `name`, `weaponPower`, etc.

In service classes - any class whose main job is to do *work* instead of hold data - you use properties for two reasons: to hold options about *how* the class should behave. And to hold other tools - like a `PDO` object.

Create a `private $pdo` property:

```
lib/ShipLoader.php
```

```
↔ // ... lines 1 - 2
3 class ShipLoader
4 {
5     private $pdo;
↔ // ... lines 6 - 71
72 }
↔ // ... lines 73 - 74
```

Now, we can use a little trick thanks to OO! Down in `getPDO()`, add an `if` statement to check if the `pdo` property is equal to `null`. Why of course it is! So far, nothing is setting it, so it's *always* null. But now, if it *is* null, move the `new PDO()` code into this and then assign this to the `pdo` property. Finish by returning `$this->pdo`:

```
lib/ShipLoader.php
```

```
↔ // ... lines 1 - 2
3 class ShipLoader
4 {
5     private $pdo;
↔ // ... lines 6 - 53
54     private function getPDO()
55     {
56         if ($this->pdo === null) {
57             $this->pdo = new PDO('mysql:host=localhost;dbname=oo_battle',
58             'root');
59             $this->pdo->setAttribute(PDO::ATTR_ERRMODE,
60             PDO::ERRMODE_EXCEPTION);
61         }
62     }
↔ // ... lines 63 - 71
72 }
↔ // ... lines 73 - 74
```

The first time you call this, `$this->pdo` is null so we create a new `PDO` object and set the property. Then, if someone calls this during the same request, the `pdo` property will already be an object, so it'll skip creating a second one and just return it. Boom!

This is the first time we've seen a service class - something that does work for us - have a property. And in service classes, properties aren't about holding data that describe something - like a `Ship` - they're used to store options about how the class should work or other useful objects that class needs.

We shouldn't notice *any* difference - so refresh to try it. Yes! Think about it: thanks to objects, we were able to reduce the number of database connections being created by touching one file and not breaking anything.

Chapter 9: OO Best Practice: Centralizing Configuration

Ok, next problem: at the bottom of `ShipLoader`, our database connection information is hardcoded. That's a problem for two reasons. First, if this works on my computer, it probably won't work on production, unless everything matches up. And second, what if we need a database connection inside some other class? Right now, we'd just have to copy and paste those credentials into yet *another* spot. Eww.

Here's the goal: move the database configuration *out* of this class to somewhere more central so it can be re-used. And good news: the way you do this is *fundamentally* important to using object-oriented code correctly.

How to Make the OO Kittens Sad

But first, let me tell you what you *shouldn't* do. You *shouldn't* just move this configuration to another file and then use some `global` keywords to get that information here. You *will* see this kind of stuff - heck you might see it all the time depending on your project. The problem is that your code gets harder to read and maintain: "Hey, where the heck is this `$dbPassword`" variable created? And what if you wanted to re-use this class in another project? It better have global variables with the exact same names.

Learning the better way is the difference between an "ok" object-oriented developer and a great one: and even though this is only episode 2, you're about to learn it.

The Secret: Pass Objects the Config they Need

The secret is this: if a service class - like `ShipLoader` - needs information - like a database password - we need to pass that information to `ShipLoader` instead of expecting it to use a global keyword or some other method to "find" it on its own. The most common way to do this is by creating a constructor.

Create a Constructor for Options

Create a `public function __construct()` and make an argument for each piece of configuration this class needs. `ShipLoader` needs *three* pieces of configuration. First, the database DSN - which is the connection parameter, thing `mysql:host=localhost`. It also needs the `$dbUser` and the `$dbPassword`:

```
lib/ShipLoader.php
1 // ... lines 1 - 2
2
3 class ShipLoader
4 {
5 // ... lines 5 - 10
6
7     public function __construct($dbDsn, $dbUser, $dbPass)
8     {
9 // ... lines 13 - 15
10
11     }
12
13 // ... lines 17 - 82
14
15 }
16
17 // ... lines 84 - 85
```

And just like any class, you'll set each of these on a private property. Create a `private $dbDsn`, `$dbUser` and `$dbPass`. In `__construct()`, assign each argument to the property. I made my arguments - like `$dbUser` the same as my property name - but that's not needed, it's just nice for my own sanity:

```
lib/ShipLoader.php
1 // ... lines 1 - 2
2
3 class ShipLoader
4 {
5 // ... lines 5 - 6
6
7     private $dbDsn;
8
9     private $dbUser;
10
11     private $dbPass;
12
13     public function __construct($dbDsn, $dbUser, $dbPass)
14     {
15         $this->dbDsn = $dbDsn;
16         $this->dbUser = $dbUser;
17         $this->dbPass = $dbPass;
18     }
19
20 // ... lines 17 - 82
21
22 }
23
24 // ... lines 84 - 85
```

If this feels silly, pointless or you don't get it yet. That's GREAT. Keep watching. Thanks to this change, whoever creates a `new ShipLoader()` is forced to pass in these 3 configuration arguments. We don't care who creates `ShipLoader`, but when they do, we store the configuration on three properties and can use that stuff in our methods below.

At the bottom - let's do that. Copy the long database DSN string from `new PDO()` and replace it with `$this->dbDsn`. Make the second argument `$this->dbUser` and the third `$this->dbPass`:

```
lib/ShipLoader.php
1 // ... lines 1 - 2
2
3 class ShipLoader
4 {
5 // ... lines 5 - 64
65     private function getPDO()
66     {
67         if ($this->pdo === null) {
68             $this->pdo = new PDO($this->dbDsn, $this->dbUser, $this-
69             >dbPass);
70             $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
71         }
72         return $this->pdo;
73     }
74 // ... lines 74 - 82
75 }
76 // ... lines 84 - 85
```

And this class is done!

Passing Configuration to the Class

But now, when we create `ShipLoader`, we need to pass arguments. In `index.php`, PhpStorm is angry - `required parameter $dbDsn` - we're missing the first argument. We could just paste our database credentials right here. But we'll probably want them somewhere central.

Open `bootstrap.php` and create a new `$configuration` array. We'll use this now as sort of a "global configuration" variable. Put the 3 database credential things here - `db_dsn` - then paste the string - `db_user` is `root` and `db_pass` is an empty string:

bootstrap.php

```
↔ // ... lines 1 - 2
3 $configuration = array(
4     'db_dsn' => 'mysql:host=localhost;dbname=oo_battle',
5     'db_user' => 'root',
6     'db_pass' => null,
7 );
↔ // ... lines 8 - 13
```

Since we're requiring this from `index.php`, we can just use it there:

`$configuration['db_dsn']` is the first argument then use `db_user` as the second argument and `db_pass` to finish things off:

index.php

```
↔ // ... line 1
2 require __DIR__ . '/bootstrap.php';
3
4 $shipLoader = new ShipLoader(
5     $configuration['db_dsn'],
6     $configuration['db_user'],
7     $configuration['db_pass']
8 );
↔ // ... lines 9 - 123
```

Yes! Now the app's configuration is all in one file. In `index.php`, we pass this stuff to `ShipLoader` via its `__construct()` method. Then `ShipLoader` doesn't have *any* hardcoded configuration. Anything that was hardcoded before was replaced by a `__construct()` argument and a private property.

Make sure our ships are still battling. Refresh! *Still* not broken!

The Big Important Rule

Here's the rule to remember: don't put configuration inside of a service class. Replace that hardcoded configuration with an argument. This allows anyone using your class to pass in whatever *they* want. The hardcoding is gone, and your class is more flexible.

Oh, and by the way - this little strategy is called dependency injection. Scary! It's a tough concept for a lot of people to understand. If it's not sinking in yet, don't worry. Practice makes perfect.

Chapter 10: OO Best Practice: Centralizing the Connection

Ready for the next problem? Our `PDO` object is configurable, but we're still creating it inside of `ShipLoader`. What's going to happen if we add a `battle` table and a `BattleLoader`? Will it *also* need to create *its* own `PDO` object? Right now - yea. So if we have 50 tables, that means 50 separate connections. The horror!

I want *one* connection that *every* class uses.

Here's the goal: move the `new PDO()` call *out* of `ShipLoader` so that it can be created in a central location and used by everyone. How? By using the same strategy we just learned with configuration. If you want to move something out of a service class, add it as a `__construct()` argument and pass it in.

Adding a `$pdo` `__construct` Argument

Let's do it! Instead of passing in the 3 database options, we need to pass in the *whole* `PDO` object. Replace the 3 arguments with just one: `$pdo`. Give it a type-hint to be great programmers. Next, remove the three configuration properties. And back in `__construct()`, we already have a `$pdo` property, so set that with `$this->pdo = $pdo`.

```
lib/ShipLoader.php
1 // ... lines 1 - 2
2
3 class ShipLoader
4 {
5     private $pdo;
6
7     public function __construct(PDO $pdo)
8     {
9         $this->pdo = $pdo;
10    }
11
12 }
13 // ... lines 11 - 71
14
15 }
16 // ... lines 73 - 74
```

Time to simplify the `getPDO()` function. We don't need to worry about creating the object anymore. Instead, just return the property:

```
lib/ShipLoader.php
1 // ... lines 1 - 2
2
3 class ShipLoader
4 {
5 // ... lines 5 - 58
6
7     private function getPDO()
8     {
9         return $this->pdo;
10    }
11
12 // ... lines 63 - 71
13
14 }
15 // ... lines 73 - 74
```

Again: big picture: if you need to remove something from a service class - whether it's configuration or an object - remove it, and add it as an argument to the `__construct()` function.

Creating PDO

But now, we need go to `index.php` and change the arguments we're passing to the `new ShipLoader()`. We're not passing these three configuration pieces anymore. Copy those. Above this, create the `PDO` object. `$pdo = new PDO()` and paste in the arguments:

```
index.php
1 // ... lines 1 - 3
2
3 $pdo = new PDO(
4     $configuration['db_dsn'],
5     $configuration['db_user'],
6     $configuration['db_pass']
7 );
8
9 // ... lines 9 - 126
```

Below, pass `$pdo` as the only argument to `new ShipLoader()`:

index.php

```
↔ // ... lines 1 - 3
4 $pdo = new PDO(
5     $configuration['db_dsn'],
6     $configuration['db_user'],
7     $configuration['db_pass']
8 );
9 $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
10
11 $shipLoader = new ShipLoader($pdo);
↔ // ... lines 12 - 126
```

Ok, let's try it! Still works. Geez - we're unstoppable today.

Unfortunately, this isn't the only place we need this. Copy the `$pdo` and `$shipLoader` code and paste it into `battle.php`:

battle.php

```
↔ // ... lines 1 - 3
4 $pdo = new PDO(
5     $configuration['db_dsn'],
6     $configuration['db_user'],
7     $configuration['db_pass']
8 );
9 $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
10
11 $shipLoader = new ShipLoader($pdo);
↔ // ... lines 12 - 114
```

Choose some ships to battle and.... Engage. And *that* still works too!

The Big Important Takeaway

Ready for the big important takeaway? Don't include configuration or create new service objects from within a service. Even though the `PDO` class comes from PHP, it *is* a service class: it does work. If we create that service object from within a class, we can't easily share it or control it.

Instead, create all of your service objects in *one* place and then pass them into each other. This stuff is hard - a lot of systems violate the heck out of these rules! And that's ok - I want you to learn to become a *great* object-oriented developer, so we're looking at the *best* way to do things.

The downside is that the code to create the service objects is getting a bit complicated. *And it's duplicated!* Dang it - it's not right yet. Let's fix that next by learning another awesome strategy.

Chapter 11: Service Container

Good news: we've got great flexibility! Bad news: we have to create the service objects by hand and this stuff is duplicated. We need to centralize what we've got here.

Creating a Service Container

To do that, we'll create *one* special class whose only job is to create these service objects. This class is called a service container, ya know, because it's basically a container for all the service objects. You'll see.

In `lib/` create a new file called `Container.php`. Inside create a class called `Container`:

`lib/Container.php`

```
1 <?php
2
3 class Container
4 {
5 // ... lines 5 - 24
25 }
```

In `battle.php` and `index.php`, we create a new `PDO` object. Let's have `Container` do that instead. Create a new `public function getPDO()` inside `Container`. Copy the code to make this and paste it here. Hmm, we need the `$configuration` variable, so copy that from `bootstrap.php` and put it here temporarily. Return `$pdo` at the bottom and perfect the method by adding some PHPDoc:

lib/Container.php

```
↔ // ... lines 1 - 2
3 class Container
4 {
5     /**
6      * @return PDO
7      */
8     public function getPDO()
9     {
10         $configuration = array(
11             'db_dsn' => 'mysql:host=localhost;dbname=oo_battle',
12             'db_user' => 'root',
13             'db_pass' => null,
14         );
15
16         $pdo = new PDO(
17             $configuration['db_dsn'],
18             $configuration['db_user'],
19             $configuration['db_pass']
20         );
21         $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
22
23         return $pdo;
24     }
25 }
```

Using the Container

Ok, nobody needs to do this work by hand anymore. Go to `index.php`. At the top, create a `$container` variable and set it to `new Container()`. Below that, replace the `new PDO()` stuff with just `$container->getPDO()`:

index.php

```
↔ // ... lines 1 - 3
4 $container = new Container();
5 $pdo = $container->getPDO();
↔ // ... lines 6 - 122
```

Copy those lines and repeat this in `battle.php`:

battle.php

```
↔ // ... lines 1 - 3
4 $container = new Container();
5 $pdo = $container->getPDO();
↔ // ... lines 6 - 110
```

Before trying this, don't forget to go to `bootstrap.php`: we need to require the file so we can access the new class:

bootstrap.php

```
↔ // ... lines 1 - 8
9 require_once __DIR__.'/lib/Container.php';
↔ // ... lines 10 - 14
```

Hey, let's give it a shot! Refresh! No problems.

Centralizing Configuration

Ok, we've started removing duplication. But I made us go one step backwards: once again, our configuration is buried inside a class - I'd rather have that somewhere central. Fix this like we always do when we want to remove some details from a class: create a

`public function __construct()` with a `$configuration` argument. Add the `$configuration` property and assign it in the construct function:

lib/Container.php

```
↔ // ... lines 1 - 2
3 class Container
4 {
5     private $configuration;
6
7     public function __construct(array $configuration)
8     {
9         $this->configuration = $configuration;
10    }
↔ // ... lines 11 - 25
26 }
```

Down in `getPDO()`, let's celebrate! Remove the `$configuration` variable and reference the property instead:

lib/Container.php

```
↔ // ... lines 1 - 2
3 class Container
4 {
5     private $configuration;
↔ // ... lines 6 - 14
15     public function getPDO()
16     {
17         $pdo = new PDO(
18             $this->configuration['db_dsn'],
19             $this->configuration['db_user'],
20             $this->configuration['db_pass']
21         );
↔ // ... lines 22 - 24
25     }
26 }
```

This is an easy change - `bootstrap.php` already holds the central `$configuration` array. In `battle.php` pass `$configuration` to the Container:

battle.php

```
↔ // ... lines 1 - 3
4 $container = new Container($configuration);
↔ // ... lines 5 - 110
```

And do the same thing for `index.php`:

index.php

```
↔ // ... lines 1 - 3
4 $container = new Container($configuration);
5 $pdo = $container->getPDO();
↔ // ... lines 6 - 122
```

Time for a sanity check! Refresh! Oh no!

```
PDOException on Container.php line 21
```

Put on your debugging cap! That's the line that creates the new `PDO` object. Hmm, we didn't change anything - this is fishy. Dump `$this->configuration` and refresh. Ah, it's `null`. Well, clearly that's not right. I see it. Silly mistake: in `__construct()`, I wasn't assigning the property. Make sure you have `$this->configuration = $configuration`:

lib/Container.php

```
↔ // ... lines 1 - 2
3 class Container
4 {
5     private $configuration;
6
7     public function __construct(array $configuration)
8     {
9         $this->configuration = $configuration;
10    }
↔ // ... lines 11 - 25
26 }
```

We were passing in the configuration, but I had forgot to set it on my property. Try it again. Excellent!

This keeps my requirement of a centralized configuration array *and* centralizing where we create service objects. But we still need to move a few more service objects in here and fix one more issue. Almost there!

Chapter 12: Container: Force Single Objects, Celebrate

Home stretch! Our goal is to make `Container` responsible for creating *every* service object: like `PDO`, but also `ShipLoader` and `BattleManager`.

Guaranteeing only One PDO Object

Here's our issue: if we called `$container->getPDO()` twice on the same request, we'd *still* end up with multiple PDO objects, and so, multiple database connections. Ok, if we're careful, we can avoid this. We can do better: let's *guarantee* that only one PDO object is ever created.

We did this before in `ShipLoader`. Create a `private $pdo` property at the top of `Container`. In `getPDO()`, add an `if` statement to see if the property is null. If it is, create the `new PDO()` object and set it on the property. Return `$this->pdo` at the bottom:

```
lib/Container.php
1 // ... lines 1 - 2
2
3 class Container
4 {
5 // ... lines 5 - 6
6
7     private $pdo;
8
9     public function getPDO()
10    {
11         if ($this->pdo === null) {
12             $this->pdo = new PDO(
13                 $this->configuration['db_dsn'],
14                 $this->configuration['db_user'],
15                 $this->configuration['db_pass']
16             );
17
18         }
19
20         return $this->pdo;
21     }
22
23 }
```

Again, the first time we call this: the `pdo` property is null, so we create the object and set the property. The second, third and fourth time we call this, the object is already there, so we just return it.

Oh, and while I'm here, I'll paste back one line I lost on accident earlier:

```
lib/Container.php
19     if ($this->pdo === null) {
20         $this->pdo = new PDO(
21             $this->configuration['db_dsn'],
22             $this->configuration['db_user'],
23             $this->configuration['db_pass']
24         );
25
26         $this->pdo->setAttribute(PDO::ATTR_ERRMODE,
27             PDO::ERRMODE_EXCEPTION);
28     }
29
30     // ... lines 28 - 32
```

This just sets up PDO to throw nice exceptions if something goes wrong so I can see them.

Move ShipLoader to the Container

Keep going! We don't want to instantiate a `ShipLoader` object manually in `battle.php` and `index.php`. Let's just do it inside `Container`.

Follow the same pattern: create a `private` property called `$shipLoader`, and a `public function getShipLoader()`:

```
lib/Container.php
1     // ... lines 1 - 2
2
3     class Container
4     {
5
6         // ... lines 5 - 8
7
8         private $shipLoader;
9
10        // ... lines 10 - 36
11
12        public function getShipLoader()
13        {
14
15            // ... lines 39 - 43
16
17            }
18
19        }
```

In here, add the same `if` statement: `if ($this->shipLoader === null)`, then `$this->shipLoader = new ShipLoader()`. Remember, *it* has a required argument for the PDO object. That's easy, just say `$this->getPDO()`. At the bottom return `$this->shipLoader` and add the PHPDoc above it:

```
lib/Container.php
1 // ... lines 1 - 2
2
3 class Container
4 {
5
6 // ... lines 5 - 8
7
8     private $shipLoader;
9
10 // ... lines 10 - 33
11
12 /**
13  * @return ShipLoader
14  */
15
16     public function getShipLoader()
17     {
18
19         if ($this->shipLoader === null) {
20             $this->shipLoader = new ShipLoader($this->getPDO());
21         }
22
23         return $this->shipLoader;
24     }
25 }
```

Use it! In `index.php`, say `$shipLoader = $container->getShipLoader()`. And I have a bonus for you! We don't need the `$pdo` variable anymore - we only did that to pass it to `ShipLoader`. Simplify!

```
index.php
1 // ... lines 1 - 3
2
3 $container = new Container($configuration);
4
5
6 $shipLoader = $container->getShipLoader();
7
8 // ... lines 7 - 121
```

Copy the new `$shipLoader` line and repeat this in `battle.php`:

```
battle.php
1 // ... lines 1 - 3
2
3 $container = new Container($configuration);
4
5
6 $shipLoader = $container->getShipLoader();
7
8 // ... lines 7 - 108
```

Ok, make sure this is all working. Refresh! Somebody make a sad trombone noise:

```
Call to a member function getShips() on a non-object index.php line 6.
```

Ok, trusty debugging cap back on. On line 6, we're calling `getShips()` on the `$shipLoader`, which is apparently null. So `$container->getShipLoader()` must *not* be returning the object for some reason. How rude.

Oh, and the problem is me! I added an extra `!` in my `if` statement so that it never got inside. Lame. Make sure your's looks like mine does now:

lib/Container.php

```
1 // ... lines 1 - 2
2
3 class Container
4 {
5 // ... lines 5 - 8
6
7     private $shipLoader;
8
9 // ... lines 10 - 33
10
11    /**
12     * @return ShipLoader
13     */
14
15    public function getShipLoader()
16    {
17        if ($this->shipLoader === null) {
18            $this->shipLoader = new ShipLoader($this->getPDO());
19        }
20
21        return $this->shipLoader;
22    }
23
24 }
25 }
```

Ok, now it works.

Move BattleManager to the Container

Only one more service to go! In `battle.php`, we create the `BattleManager`. Let's move it! Add the `private $battleManager` property and then the

`public function getBattleManager()`. Copy the ship loader code to save time... and so I don't mess up again. Update it for `battleManager`:

```
$this->battleManager = new BattleManager(); And return
$this->battleManager:
```

lib/Container.php

```
↔ // ... lines 1 - 2
3 class Container
4 {
↔ // ... lines 5 - 10
11     private $battleManager;
↔ // ... lines 12 - 47
48     /**
49      * @return BattleManager
50      */
51     public function getBattleManager()
52     {
53         if ($this->battleManager === null) {
54             $this->battleManager = new BattleManager();
55         }
56
57         return $this->battleManager;
58     }
59 }
```

Go use it in `battle.php`: `$battleManager = $container->getBattleManager();`:

battle.php

```
↔ // ... lines 1 - 26
27 $battleManager = $container->getBattleManager();
↔ // ... lines 28 - 109
```

Ok, let's try the *whole* thing! Start a battle... and Engage. Ok, the bad guys won, but our app still works. And the code behind it is so much more awesome.

Chapter 13: Container to the Rescue

Congratulations! What we just did is *incredible*. Every service object we have - meaning every object that does work like `BattleManager`, `PDO` and `ShipLoader` - is created by the `Container` class. This is its *only* job.

Adding Arguments? Simple

The benefits are huge. Here's one. Imagine we need to give `BattleManager` a few constructor arguments. Once we've done that, the *only* code we need to touch outside of `BattleManager` is right here inside Container. We *don't* need to go anywhere else - like `battle.php` - and change *anything*. We just say `$container->getBattleManager()` and the `Container` class will take care of all of the work to create that object.

Objects aren't Created Until/Unless Needed

But wait, there's more! Before, at the top of our files - like `index.php` - we created *all* of our objects. So if we had 50 different useful service objects, we'd create them all right here. How wasteful.

But with the `Container` idea, none of these objects are created until and *unless* you ask for them. For example, `index.php` never calls `$container->getBattleManager()`. So the `BattleManager` object is never created. We save precious CPUs and memory.

Containers: A Pattern

I didn't invent this Container idea - it's a well-known strategy called a dependency injection container. It's a special class and you always have just one.

Its only job is to create service objects. And in fact, if you do a good job, *all* service objects will be created here - you won't instantiate them *anywhere* else.

Model Classes versus Service Classes

Remember - *model* objects - like `Ship` and `BattleResult` - are classes that just hold data and don't really do much work. And you can create *these* whenever you need them - they're *not* created by the Container. So in `BattleManager` at the bottom of `battle()`, we needed a new `BattleResult` to be a container for our data. And in `ShipLoader`, whenever you query for a ship, we create a new `Ship` model object.

Model objects *can* be created anywhere in your code, whenever you need them. But these service objects - the ones that do work for you and don't really hold data - these should be created in a central spot. And the `Container` is a nice way to do that.

Reorganizing Models and Services

To make this more clear in our app, let's redecorate. Create a `lib/Service` directory and a `lib/Model` directory. Move `BattleManager`, `ShipLoader` and `Container` - it's a little different, but it's still technically a service - into `lib/Service`. And move `BattleResult` and `Ship` - our simple "model" objects into `lib/Model`:

```
mv lib/BattleManager.php lib/Service
mv lib/ShipLoader.php lib/Service
mv lib/Container.php lib/Service

mv lib/Ship.php lib/Model
mv lib/BattleResult.php lib/Model
```

To make this work, we just need to update the `require` paths in `bootstrap.php`:

```
bootstrap.php
 1 // ... lines 1 - 8
 2 require_once __DIR__ . '/lib/Service/Container.php';
 3 require_once __DIR__ . '/lib/Model/Ship.php';
 4 require_once __DIR__ . '/lib/Service/BattleManager.php';
 5 require_once __DIR__ . '/lib/Service/ShipLoader.php';
 6 require_once __DIR__ . '/lib/Model/BattleResult.php';
```

And yes, in a future episode, we're going to fully get rid of these. And it will be great.

Refresh! Still working!

Best Practices vs the Real World

In this episode, instead of learning more OO concepts, we went straight to the hard stuff and learned how to *organize* our code into model classes that hold data and service classes that do work. We also learned that when you're in a service class - like `ShipLoader` - instead of hardcoding configuration or creating other service objects inside, we can move those outside of the class and add anything we need as an argument to the `__construct()` function. Then, we'll pass that information to the class. That's dependency injection, and it's one of the harder things to grasp about OO. So if it doesn't totally make sense yet - stick with us - we'll keep practicing.

Now a quick warning. When you look at other projects, this idea of model objects -- that hold data but don't do anything - and service objects - that do work but don't really hold any data - is not always followed. Sometimes you'll see these mixed together you might have a class like `Ship` that has methods in it that do work - like `battle()` or even `save()` that would save the Ship's data to the database.

What I'm showing you are "best practices". When you get out into the wild, it's not always this clean. And that's ok - over time, you'll learn to bend the rules when it makes sense. But in your mind, keep these two *types* of classes separate and recognize if a class is a model, a service or both.

Ok guys - in the next episodes, we're going to dive into more great concepts of OO - like interfaces, abstract classes, and static calls. These will really take your mad-skills to the next level.

So join us, and I'll seeya guys next time!

With <3 from SymfonyCasts