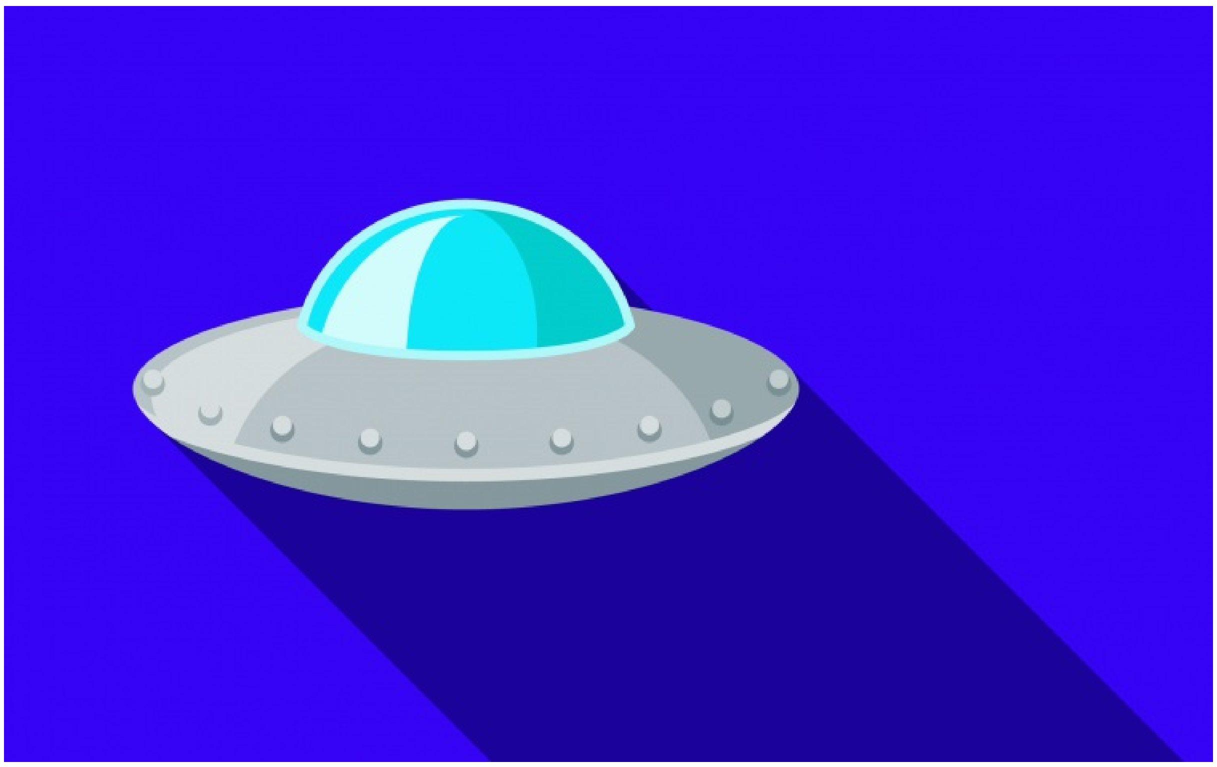
OOP (Course 2): Services, Dependency Injection and Containers



With <3 from SymfonyCasts

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:

117 lines lib/Ship.php

```
... lines 1 - 2
class Ship
{
private $name;
private $weaponPower = 0;
private $jediFactor = 0;
private $strength = 0;
private $underRepair;
... lines 14 - 115
}
```

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

126 lines functions.php

```
... lines 1 - 4
function get_ships()
{
$ships = array();
$ship = new Ship('Jedi Starfighter');
//$ship->setName('Jedi Starfighter');
$ship->setWeaponPower(5);
$ship->setJediFactor(15);
$ship->setStrength(30);
$ships['starfighter'] = $ship;
... lines 15 - 33
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:

59 lines lib/BattleManager.php

```
<?php
class BattleManager
{
... lines 5 - 57
}
```

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:

59 lines lib/BattleManager.php

```
... lines 1 - 2
class BattleManager
{
/**
* Our complex fighting algorithm!
*
* @return array With keys winning_ship, losing_ship & used_jedi_powers
*/
public function battle(Ship $ship1, $ship1Quantity, Ship $ship2, $ship2Quantity)
{
... lines 12 - 51
return array(
'winning_ship' => $winningShip,
'losing_ship' => $losingShip,
'used_jedi_powers' => $usedJediPowers,
);
}
}
```

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:

98 lines battle.php

```
... lines 1 - 28
$battleManager = new BattleManager();
... lines 30 - 98
```

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

98 lines battle.php

```
... lines 1 - 28
$battleManager = new BattleManager();
$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:

73 lines functions.php

```
<?php
require_once __DIR__.'/lib/Ship.php';
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():

59 lines lib/BattleManager.php

```
... lines 1 - 2
class BattleManager
{
... lines 5 - 9
public function battle(Ship $ship1, $ship1Quantity, Ship $ship2, $ship2Quantity)
{
... lines 12 - 18
if (didJediDestroyShipUsingTheForce($ship1)) {
... lines 20 - 23
}
... lines 25 - 56
}
}
```

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:

66 lines lib/BattleManager.php

```
... lines 1 - 2
class BattleManager
{
... lines 5 - 58
private function didJediDestroyShipUsingTheForce(Ship $ship)
{
$jediHeroProbability = $ship->getJediFactor() / 100;
return mt_rand(1, 100) <= ($jediHeroProbability*100);
}
}
```

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:

66 lines lib/BattleManager.php

```
... lines 1 - 2
class BattleManager
{
... lines 5 - 9
public function battle(Ship $ship1, $ship1Quantity, Ship $ship2, $ship2Quantity)
{
... lines 12 - 18
if ($this->didJediDestroyShipUsingTheForce($ship1)) {
... lines 20 - 23
}
if ($this->didJediDestroyShipUsingTheForce($ship2)) {
... lines 26 - 29
}
... lines 31 - 56
}
... lines 58 - 64
}
```

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:

37 lines lib/ShipLoader.php

```
... lines 1 - 2
class ShipLoader
{
... lines 5 - 36
}
```

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:

37 lines lib/ShipLoader.php

```
... lines 1 - 2
class ShipLoader
{
public function getShips()
{
$ships = array();
$ship = new Ship('Jedi Starfighter');
... lines 10 - 33
return $ships;
}
}
```

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:

6 lines functions.php

```
... lines 1 - 4
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():

119 lines index.php

```
<?php
require __DIR__.'/functions.php';
$shipLoader = new ShipLoader();
$ships = $shipLoader->getShips();
... lines 6 - 119
```

Do the same thing in battle.php:

99 lines battle.php

```
<?php
require __DIR__.'/functions.php';
$shipLoader = new ShipLoader();
$ships = $shipLoader->getShips();
... 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:

99 lines battle.php

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

and index.php:

119 lines index.php

```
<?php
require __DIR__.'/bootstrap.php';
... 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 associatve array - with winning_ship, losing_ship and used_jedi_powers keys:

66 lines lib/BattleManager.php

```
... lines 1 - 2
class BattleManager
{
... lines 5 - 9
public function battle(Ship $ship1, $ship1Quantity, Ship $ship2, $ship2Quantity)
{
... lines 12 - 51
return array(
'winning_ship' => $winningShip,
'losing_ship' => $losingShip,
'used_jedi_powers' => $usedJediPowers,
);
}
... lines 58 - 64
}
```

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

99 lines battle.php

```
... lines 1 - 30
$outcome = $battleManager->battle($ship1, $ship1Quantity, $ship2, $ship2Quantity);
... lines 32 - 77
<?php if ($outcome['winning_ship'] == null): ?>
Both ships destroyed each other in an epic battle to the end.
<?php else: ?>
The <?php echo $outcome['winning_ship']->getName(); ?>
<?php if ($outcome['used_jedi_powers']): ?>
used its Jedi Powers for a stunning victory!
<?php else: ?>
overpowered and destroyed the <?php echo $outcome['losing_ship']->getName() ?>s
<?php endif; ?>
<?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:

16 lines lib/BattleResult.php

```
... lines 1 - 2
class BattleResult
{
... lines 5 - 14
}
```

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:

16 lines lib/BattleResult.php

```
... lines 1 - 2
class BattleResult
{
private $usedJediPowers;
private $winningShip;
private $losingShip;
... lines 8 - 14
}
```

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":

117 lines lib/Ship.php

```
... lines 1 - 2
class Ship
{
private $name;
... lines 6 - 14
public function __construct($name)
{
$this->name = $name;
... lines 18 - 19
}
... lines 21 - 115
}
```

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

117 lines lib/Ship.php

```
... lines 1 - 2
class Ship
{
... lines 5 - 36
public function setStrength($number)
{
if (!is_numeric($number)) {
throw new \Exception('Strength must be a number, duh!');
}
$this->strength = $number;
}
... lines 45 - 100
/**
* @param int $weaponPower
*/
public function setWeaponPower($weaponPower)
{
$this->weaponPower = $weaponPower;
}
/**
* @param int $jediFactor
*/
public function setJediFactor($jediFactor)
{
$this->jediFactor = $jediFactor;
}
... 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:

16 lines lib/BattleResult.php

```
... lines 1 - 2
class BattleResult
{
private $usedJediPowers;
private $winningShip;
private $losingShip;
public function __construct($usedJediPowers, $winningShip, $losingShip)
{
$this->usedJediPowers = $usedJediPowers;
$this->winningShip = $winningShip;
$this->losingShip = $losingShip;
}
}
```

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:

62 lines lib/BattleManager.php

```
... lines 1 - 2
class BattleManager
{
... lines 5 - 9
public function battle(Ship $ship1, $ship1Quantity, Ship $ship2, $ship2Quantity)
{
... lines 12 - 51
return new BattleResult($usedJediPowers, $winningShip, $losingShip);
}
... lines 54 - 60
}
```

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

7 lines bootstrap.php

```
... lines 1 - 5
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:

99 lines battle.php

```
... lines 1 - 30
$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.:

99 lines battle.php

```
... lines 1 - 70
<?php if ($outcome['winning_ship']): ?>
<?php echo $outcome['winning_ship']->getName(); ?>
<?php else: ?>
Nobody
<?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:

45 lines lib/BattleResult.php

```
... lines 1 - 2
class BattleResult
{
... lines 5 - 13
public function __construct($usedJediPowers, Ship $winningShip, Ship $losingShip)
{
$this->usedJediPowers = $usedJediPowers;
$this->winningShip = $winningShip;
$this->losingShip = $losingShip;
}
... lines 20 - 43
}
```

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:

45 lines lib/BattleResult.php

```
... lines 1 - 2
class BattleResult
{
... lines 5 - 31
public function getWinningShip()
{
return $this->winningShip;
}
... lines 36 - 43
}
```

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:

45 lines lib/BattleResult.php

```
... lines 1 - 2
class BattleResult
{
... lines 5 - 20
/**
* @return boolean
*/
public function isUsedJediPowers()
{
return $this->usedJediPowers;
}
/**
* @return Ship
*/
public function getWinningShip()
{
return $this->winningShip;
}
/**
* @return Ship
*/
public function getLosingShip()
{
return $this->losingShip;
}
}
```

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()?

45 lines lib/BattleResult.php

```
... lines 1 - 2
class BattleResult
{
... lines 5 - 20
/**
* @return boolean
*/
public function wereJediPowersUsed()
{
return $this->usedJediPowers;
}
... lines 28 - 43
}
```

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:

99 lines battle.php

```
... lines 1 - 30
$battleResult = $battleManager->battle($ship1, $ship1Quantity, $ship2, $ship2Quantity);
... lines 32 - 99
```

Below, use $battleResult->getWinningShip():

99 lines battle.php

```
... lines 1 - 30
$battleResult = $battleManager->battle($ship1, $ship1Quantity, $ship2, $ship2Quantity);
... lines 32 - 70
<?php if ($battleResult->getWinningShip()): ?>
... 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:

62 lines lib/BattleManager.php

```
... lines 1 - 2
class BattleManager
{
/**
* Our complex fighting algorithm!
*
* @return BattleResult
*/
public function battle(Ship $ship1, $ship1Quantity, Ship $ship2, $ship2Quantity)
{
... lines 12 - 52
}
... lines 54 - 60
}
```

Ok, now PhpStorm is acting friendly - the angry highightling 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():

99 lines battle.php

```
... lines 1 - 70
<?php if ($battleResult->getWinningShip()): ?>
<?php echo $battleResult->getWinningShip()->getName(); ?>
<?php else: ?>
Nobody
<?php endif; ?>
... lines 76 - 77
<?php if ($battleResult->getWinningShip() == null): ?>
Both ships destroyed each other in an epic battle to the end.
<?php else: ?>
The <?php echo $battleResult->getWinningShip()->getName(); ?>
<?php if ($battleResult->wereJediPowersUsed()): ?>
used its Jedi Powers for a stunning victory!
<?php else: ?>
overpowered and destroyed the <?php echo $battleResult->getLosingShip()->getName() ?>s
<?php endif; ?>
<?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:

55 lines lib/BattleResult.php

```
... lines 1 - 2
class BattleResult
{
... lines 5 - 13
public function __construct($usedJediPowers, Ship $winningShip = null, Ship $losingShip = null)
{
$this->usedJediPowers = $usedJediPowers;
$this->winningShip = $winningShip;
$this->losingShip = $losingShip;
}
... lines 20 - 53
}
```

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

55 lines lib/BattleResult.php

```
... lines 1 - 2
class BattleResult
{
... lines 5 - 36
/**
* @return Ship|null
*/
public function getLosingShip()
{
return $this->losingShip;
}
/**
* Was there a winner? Or did everybody die :(
*
* @return bool
*/
public function isThereAWinner()
{
return $this->getWinningShip() !== null;
}
}
```

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:

55 lines lib/BattleResult.php

```
... lines 1 - 2
class BattleResult
{
... lines 5 - 43
/**
* Was there a winner? Or did everybody die :(
*
* @return bool
*/
public function isThereAWinner()
{
return $this->getWinningShip() !== null;
}
}
```

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:

99 lines battle.php

```
... lines 1 - 70
<?php if ($battleResult->isThereAWinner()): ?>
... line 72
<?php else: ?>
... line 74
<?php endif; ?>
... lines 76 - 77
<?php if (!$battleResult->isThereAWinner()): ?>
... line 79
<?php else: ?>
... lines 81 - 86
<?php endif; ?>
... 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:

66 lines lib/BattleManager.php

```
... lines 1 - 9
public function battle(Ship $ship1, $ship1Quantity, Ship $ship2, $ship2Quantity)
{
... lines 12 - 16
while ($ship1Health > 0 && $ship2Health > 0) {
... lines 18 - 31
// now battle them normally
$ship1Health = $ship1Health - ($ship2->getWeaponPower() * $ship2Quantity);
$ship2Health = $ship2Health - ($ship1->getWeaponPower() * $ship1Quantity);
}
// update the strengths on the ships, so we can show this
$ship1->setStrength($ship1Health);
$ship2->setStrength($ship2Health);
... lines 40 - 56
}
... 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():

106 lines battle.php

```
... lines 1 - 25
$ship1 = $ships[$ship1Name];
$ship2 = $ships[$ship2Name];
$battleManager = new BattleManager();
$battleResult = $battleManager->battle($ship1, $ship1Quantity, $ship2, $ship2Quantity);
... 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:

106 lines battle.php

```
... lines 1 - 33
<html>
... lines 35 - 53
<body>
<div class="container">
... lines 56 - 67
<div class="result-box center-block">
... lines 69 - 88
<h3>Remaining Strength</h3>
<dl class="dl-horizontal">
... lines 91 - 94
</dl>
</div>
... lines 97 - 102
</div>
</body>
</html>
```

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

106 lines battle.php

```
... lines 1 - 33
<html>
... lines 35 - 53
<body>
<div class="container">
... lines 56 - 67
<div class="result-box center-block">
... lines 69 - 88
<h3>Remaining Strength</h3>
<dl class="dl-horizontal">
<dt><?php echo $ship1->getName(); ?></dt>
<dd><?php echo $ship1->getStrength(); ?></dd>
... lines 93 - 94
</dl>
</div>
... lines 97 - 102
</div>
</body>
</html>
```

Do the same thing for $ship2:

106 lines battle.php

```
... lines 1 - 33
<html>
... lines 35 - 53
<body>
<div class="container">
... lines 56 - 67
<div class="result-box center-block">
... lines 69 - 88
<h3>Remaining Strength</h3>
<dl class="dl-horizontal">
<dt><?php echo $ship1->getName(); ?></dt>
<dd><?php echo $ship1->getStrength(); ?></dd>
<dt><?php echo $ship2->getName(); ?></dt>
<dd><?php echo $ship2->getStrength(); ?></dd>
</dl>
</div>
... lines 97 - 102
</div>
</body>
</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:

57 lines init_db.php

```
<?php
/*
* SETTINGS!
*/
$databaseName = 'oo_battle';
$databaseUser = 'root';
$databasePassword = '';
/*
* CREATE THE DATABASE
*/
$pdoDatabase = new PDO('mysql:host=localhost', $databaseUser, $databasePassword);
$pdoDatabase->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
$pdoDatabase->exec('CREATE DATABASE IF NOT EXISTS oo_battle');
... 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:

57 lines init_db.php

```
... lines 1 - 25
$pdo->exec('CREATE TABLE `ship` (
`id` int(11) NOT NULL AUTO_INCREMENT,
`name` varchar(255) COLLATE utf8mb4_unicode_ci NOT NULL,
`weapon_power` int(4) NOT NULL,
`jedi_factor` int(4) NOT NULL,
`strength` int(4) NOT NULL,
`is_under_repair` tinyint(1) NOT NULL,
PRIMARY KEY (`id`)
) 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:

57 lines init_db.php

```
... lines 1 - 38
$pdo->exec('INSERT INTO ship
(name, weapon_power, jedi_factor, strength, is_under_repair) VALUES
("Jedi Starfighter", 5, 15, 30, 0)'
);
$pdo->exec('INSERT INTO ship
(name, weapon_power, jedi_factor, strength, is_under_repair) VALUES
("CloakShape Fighter", 2, 2, 70, 0)'
);
$pdo->exec('INSERT INTO ship
(name, weapon_power, jedi_factor, strength, is_under_repair) VALUES
("Super Star Destroyer", 70, 0, 500, 0)'
);
$pdo->exec('INSERT INTO ship
(name, weapon_power, jedi_factor, strength, is_under_repair) VALUES
("RZ-1 A-wing interceptor", 4, 4, 50, 0)'
);
... 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:

44 lines lib/ShipLoader.php

```
... lines 1 - 2
class ShipLoader
{
public function getShips()
{
$pdo = new PDO('mysql:host=localhost;dbname=oo_battle', 'root');
$pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
... lines 9 - 41
}
}
```

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

44 lines lib/ShipLoader.php

```
... lines 1 - 4
public function getShips()
{
$pdo = new PDO('mysql:host=localhost;dbname=oo_battle', 'root');
$pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
$statement = $pdo->prepare('SELECT * FROM ship');
... lines 10 - 41
}
... 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:

44 lines [lib/ShipLoader.php]

```
... lines 1 - 4
public function getShips()
{
$pdo = new PDO('mysql:host=localhost;dbname=oo_battle', 'root');
$pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
$statement = $pdo->prepare('SELECT * FROM ship');
$statement->execute();
$shipsArray = $statement->fetchAll(PDO::FETCH_ASSOC);
var_dump($shipsArray);die;
... lines 13 - 41
}
... 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:

50 lines [lib/ShipLoader.php]

```
... lines 1 - 2
class ShipLoader
{
... lines 5 - 39
private function queryForShips()
{
$pdo = new PDO('mysql:host=localhost;dbname=oo_battle', 'root');
$pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
$statement = $pdo->prepare('SELECT * FROM ship');
$statement->execute();
$shipsArray = $statement->fetchAll(PDO::FETCH_ASSOC);
return $shipsArray;
}
}
```

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

50 lines lib/ShipLoader.php

```
... lines 1 - 2
class ShipLoader
{
public function getShips()
{
$ships = array();
$shipsData = $this->queryForShips();
var_dump($shipsData);die;
... lines 11 - 37
}
... lines 39 - 49
}
```

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:

33 lines lib/ShipLoader.php

```
... lines 1 - 4
public function getShips()
{
$ships = array();
$shipsData = $this->queryForShips();
foreach ($shipsData as $shipData) {
$ship = new Ship($shipData['name']);
... lines 13 - 17
}
... lines 19 - 20
}
... 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:

33 lines lib/ShipLoader.php

```
... lines 1 - 4
public function getShips()
{
$ships = array();
$shipsData = $this->queryForShips();
foreach ($shipsData as $shipData) {
$ship = new Ship($shipData['name']);
$ship->setWeaponPower($shipData['weapon_power']);
$ship->setJediFactor($shipData['jedi_factor']);
$ship->setStrength($shipData['strength']);
$ships[] = $ship;
}
return $ships;
}
... 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:

119 lines index.php

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

135 lines lib/Ship.php

```
... lines 1 - 2
class Ship
{
private $id;
... lines 6 - 133
}
```

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:

135 lines lib/Ship.php

```
... lines 1 - 2
class Ship
{
... lines 5 - 118
/**
* @return int
*/
public function getId()
{
return $this->id;
}
/**
* @param int $id
*/
public function setId($id)
{
$this->id = $id;
}
}
```

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

34 lines lib/ShipLoader.php

```
... lines 1 - 2
class ShipLoader
{
public function getShips()
{
... lines 7 - 10
foreach ($shipsData as $shipData) {
$ship = new Ship($shipData['name']);
$ship->setId($shipData['id']);
... lines 14 - 17
$ships[] = $ship;
... lines 19 - 21
}
... lines 23 - 33
}
```

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:

119 lines index.php

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

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

119 lines index.php

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

106 lines battle.php

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

45 lines lib/ShipLoader.php

```
... lines 1 - 2
class ShipLoader
{
... lines 5 - 23
public function findOneById($id)
{
$pdo = new PDO('mysql:host=localhost;dbname=oo_battle', 'root');
$pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
$statement = $pdo->prepare('SELECT * FROM ship WHERE id = :id');
$statement->execute(array('id' => $id));
... lines 30 - 32
}
... 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:

45 lines lib/ShipLoader.php

```
... lines 1 - 23
public function findOneById($id)
{
$pdo = new PDO('mysql:host=localhost;dbname=oo_battle', 'root');
$pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
$statement = $pdo->prepare('SELECT * FROM ship WHERE id = :id');
$statement->execute(array('id' => $id));
$shipArray = $statement->fetch(PDO::FETCH_ASSOC);
var_dump($shipArray);die;
}
... 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:

106 lines battle.php

```
... lines 1 - 16
$ship1 = $shipLoader->findOneById($ship1Id);
$ship2 = $shipLoader->findOneById($ship2Id);
... 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:

57 lines lib/ShipLoader.php

```
... lines 1 - 2
class ShipLoader
{
... lines 5 - 32
private function createShipFromData(array $shipData)
{
... lines 35 - 41
}
... lines 43 - 54
}
... lines 56 - 57
```

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

57 lines lib/ShipLoader.php

```
... lines 1 - 32
private function createShipFromData(array $shipData)
{
$ship = new Ship($shipData['name']);
$ship->setId($shipData['id']);
$ship->setWeaponPower($shipData['weapon_power']);
$ship->setJediFactor($shipData['jedi_factor']);
$ship->setStrength($shipData['strength']);
return $ship;
}
... 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():

57 lines lib/ShipLoader.php

```
... lines 1 - 4
public function getShips()
{
... lines 7 - 10
foreach ($shipsData as $shipData) {
$ships[] = $this->createShipFromData($shipData);
}
... lines 14 - 15
}
... line 17
public function findOneById($id)
{
$pdo = new PDO('mysql:host=localhost;dbname=oo_battle', 'root');
$pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
$statement = $pdo->prepare('SELECT * FROM ship WHERE id = :id');
$statement->execute(array('id' => $id));
$shipArray = $statement->fetch(PDO::FETCH_ASSOC);
... lines 25 - 29
return $this->createShipFromData($shipArray);
}
... 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:

57 lines lib/ShipLoader.php

```
... lines 1 - 17
public function findOneById($id)
{
... lines 20 - 23
$shipArray = $statement->fetch(PDO::FETCH_ASSOC);
if (!$shipArray) {
return null;
}
return $this->createShipFromData($shipArray);
}
... lines 32 - 57
```

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

106 lines battle.php

```
... lines 1 - 16
$ship1 = $shipLoader->findOneById($ship1Id);
$ship2 = $shipLoader->findOneById($ship2Id);
if (!$ship1 || !$ship2) {
header('Location: /index.php?error=bad_ships');
die;
}
... 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[]:

64 lines lib/ShipLoader.php

```
... lines 1 - 2
class ShipLoader
{
/**
* @return Ship[]
*/
public function getShips()
{
... lines 10 - 18
}
... lines 20 - 61
}
... 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:

64 lines lib/ShipLoader.php

```
... lines 1 - 2
class ShipLoader
{
... lines 5 - 20
/**
* @param $id
* @return Ship
*/
public function findOneById($id)
{
$pdo = new PDO('mysql:host=localhost;dbname=oo_battle', 'root');
$pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
$statement = $pdo->prepare('SELECT * FROM ship WHERE id = :id');
$statement->execute(array('id' => $id));
$shipArray = $statement->fetch(PDO::FETCH_ASSOC);
if (!$shipArray) {
return null;
}
return $this->createShipFromData($shipArray);
}
... lines 39 - 61
}
... 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:

70 lines lib/ShipLoader.php

```
... lines 1 - 2
class ShipLoader
{
... lines 5 - 48
/**
* @return PDO
*/
private function getPDO()
{
$pdo = new PDO('mysql:host=localhost;dbname=oo_battle', 'root');
$pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
return $pdo;
}
... lines 59 - 67
}
... lines 69 - 70
```

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

70 lines lib/ShipLoader.php

```
... lines 1 - 2
class ShipLoader
{
... lines 5 - 23
*/
public function findOneById($id)
{
$statement = $this->getPDO()->prepare('SELECT * FROM ship WHERE id = :id');
... lines 28 - 35
}
... lines 37 - 59
private function queryForShips()
{
$statement = $this->getPDO()->prepare('SELECT * FROM ship');
... lines 63 - 66
}
}
... 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:

74 lines lib/ShipLoader.php

```
... lines 1 - 2
class ShipLoader
{
private $pdo;
... lines 6 - 71
}
... 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:

74 lines lib/ShipLoader.php

```
... lines 1 - 2
class ShipLoader
{
private $pdo;
... lines 6 - 53
private function getPDO()
{
if ($this->pdo === null) {
$this->pdo = new PDO('mysql:host=localhost;dbname=oo_battle', 'root');
$this->pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
}
return $this->pdo;
}
... lines 63 - 71
}
... 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:

85 lines lib/ShipLoader.php

```
... lines 1 - 2
class ShipLoader
{
... lines 5 - 10
public function __construct($dbDsn, $dbUser, $dbPass)
{
... lines 13 - 15
}
... lines 17 - 82
}
... 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:

85 lines lib/ShipLoader.php

```
... lines 1 - 2
class ShipLoader
{
... lines 5 - 6
private $dbDsn;
private $dbUser;
private $dbPass;
public function __construct($dbDsn, $dbUser, $dbPass)
{
$this->dbDsn = $dbDsn;
$this->dbUser = $dbUser;
$this->dbPass = $dbPass;
}
... lines 17 - 82
}
... 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:

85 lines lib/ShipLoader.php

```
... lines 1 - 2
class ShipLoader
{
... lines 5 - 64
private function getPDO()
{
if ($this->pdo === null) {
$this->pdo = new PDO($this->dbDsn, $this->dbUser, $this->dbPass);
$pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
}
return $this->pdo;
}
... lines 74 - 82
}
... 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:

13 lines bootstrap.php

```
... lines 1 - 2
$configuration = array(
'db_dsn' => 'mysql:host=localhost;dbname=oo_battle',
'db_user' => 'root',
'db_pass' => null,
);
... 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:

123 lines index.php

```
... line 1
require __DIR__.'/bootstrap.php';
$shipLoader = new ShipLoader(
$configuration['db_dsn'],
$configuration['db_user'],
$configuration['db_pass']
);
... 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.

74 lines lib/ShipLoader.php

```
... lines 1 - 2
class ShipLoader
{
private $pdo;
public function __construct(PDO $pdo)
{
$this->pdo = $pdo;
}
... lines 11 - 71
}
... 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:

74 lines lib/ShipLoader.php

```
... lines 1 - 2
class ShipLoader
{
... lines 5 - 58
private function getPDO()
{
return $this->pdo;
}
... lines 63 - 71
}
... 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:

126 lines index.php

```
... lines 1 - 3
$pdo = new PDO(
$configuration['db_dsn'],
$configuration['db_user'],
$configuration['db_pass']
);
... lines 9 - 126
```

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

126 lines index.php

```
... lines 1 - 3
$pdo = new PDO(
$configuration['db_dsn'],
$configuration['db_user'],
$configuration['db_pass']
);
$pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
$shipLoader = new ShipLoader($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:

114 lines battle.php

```
... lines 1 - 3
$pdo = new PDO(
$configuration['db_dsn'],
$configuration['db_user'],
$configuration['db_pass']
);
$pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
$shipLoader = new ShipLoader($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:

26 lines lib/Container.php

```
<?php
class Container
{
... lines 5 - 24
}
```

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:

26 lines lib/Container.php

```
... lines 1 - 2
class Container
{
/**
* @return PDO
*/
public function getPDO()
{
$configuration = array(
'db_dsn' => 'mysql:host=localhost;dbname=oo_battle',
'db_user' => 'root',
'db_pass' => null,
);
$pdo = new PDO(
$configuration['db_dsn'],
$configuration['db_user'],
$configuration['db_pass']
);
$pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
return $pdo;
}
}
```

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():

122 lines index.php

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

Copy those lines and repeat this in battle.php:

110 lines battle.php

```
... lines 1 - 3
$container = new Container();
$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:

14 lines bootstrap.php

```
... lines 1 - 8
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:

27 lines lib/Container.php

```
... lines 1 - 2
class Container
{
private $configuration;
public function __construct(array $configuration)
{
$this->configuration = $configuration;
}
... lines 11 - 25
}
```

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

27 lines lib/Container.php

```
... lines 1 - 2
class Container
{
private $configuration;
... lines 6 - 14
public function getPDO()
{
$pdo = new PDO(
$this->configuration['db_dsn'],
$this->configuration['db_user'],
$this->configuration['db_pass']
);
... lines 22 - 24
}
}
```

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

110 lines battle.php

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

And do the same thing for index.php:

122 lines index.php

```
... lines 1 - 3
$container = new Container($configuration);
$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:

27 lines lib/Container.php

```
... lines 1 - 2
class Container
{
private $configuration;
public function __construct(array $configuration)
{
$this->configuration = $configuration;
}
... lines 11 - 25
}
```

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:

32 lines lib/Container.php

```
... lines 1 - 2
class Container
{
... lines 5 - 6
private $pdo;
... lines 8 - 16
public function getPDO()
{
if ($this->pdo === null) {
$this->pdo = new PDO(
$this->configuration['db_dsn'],
$this->configuration['db_user'],
$this->configuration['db_pass']
);
... lines 25 - 26
}
... line 28
return $this->pdo;
}
... lines 31 - 32
```

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:

32 lines lib/Container.php

```
... lines 1 - 18
if ($this->pdo === null) {
$this->pdo = new PDO(
$this->configuration['db_dsn'],
$this->configuration['db_user'],
$this->configuration['db_pass']
);
$this->pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
}
... 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():

46 lines lib/Container.php

```
... lines 1 - 2
class Container
{
... lines 5 - 8
private $shipLoader;
... lines 10 - 36
public function getShipLoader()
{
... lines 39 - 43
}
}
```

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:

46 lines lib/Container.php

```
... lines 1 - 2
class Container
{
... lines 5 - 8
private $shipLoader;
... lines 10 - 33
/**
* @return ShipLoader
*/
public function getShipLoader()
{
if ($this->shipLoader === null) {
$this->shipLoader = new ShipLoader($this->getPDO());
}
return $this->shipLoader;
}
}
```

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!

121 lines index.php

```
... lines 1 - 3
$container = new Container($configuration);
$shipLoader = $container->getShipLoader();
... lines 7 - 121
```

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

108 lines battle.php

```
... lines 1 - 3
$container = new Container($configuration);
$shipLoader = $container->getShipLoader();
... 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:

46 lines lib/Container.php

```
... lines 1 - 2
class Container
{
... lines 5 - 8
private $shipLoader;
... lines 10 - 33
/**
* @return ShipLoader
*/
public function getShipLoader()
{
if ($this->shipLoader === null) {
$this->shipLoader = new ShipLoader($this->getPDO());
}
return $this->shipLoader;
}
}
```

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:

60 lines lib/Container.php

```
... lines 1 - 2
class Container
{
... lines 5 - 10
private $battleManager;
... lines 12 - 47
/**
* @return BattleManager
*/
public function getBattleManager()
{
if ($this->battleManager === null) {
$this->battleManager = new BattleManager();
}
return $this->battleManager;
}
}
```

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

109 lines battle.php

```
... lines 1 - 26
$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:

14 lines bootstrap.php

```
... lines 1 - 8
require_once __DIR__.'/lib/Service/Container.php';
require_once __DIR__.'/lib/Model/Ship.php';
require_once __DIR__.'/lib/Service/BattleManager.php';
require_once __DIR__.'/lib/Service/ShipLoader.php';
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!