

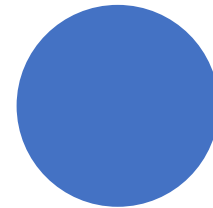


# Service Container & Auto Resolution

- We have already looked at many parts of Laravel:
  - Databases
    - Migrations
    - Seeds
    - Factories
  - Models
  - Controllers
  - Views
- Today we will be looking at a core (yet somewhat advanced and powerful) concept:
  - Service Containers

---

# Laravel Core Concepts



# Design Pattern

- You are used to programming custom code to express logic that calls re-usable libraries. The libraries take care of generic tasks.
- For example, we might use a serialization library to read a file, but we write most of the logic ourselves in the main method.
- The Service Container is Laravel's implementation of a design pattern called IoC – Inversion of Control
  - You write custom classes which receive the flow of control from the framework
  - For example, you provide a route and corresponding method, these hook into the Laraval framework to create a new web page
- The Service Container is also known as the IoC container

# What is the Service Container?

- The service container is essentially a normal class that implements the Singleton Design Pattern.
  - This means there is only ever one instance of the class
- We can place tools, classes, services, etc into the container and pre-configure them
- Since it is a singleton we can retrieve and use these tools easily at any point in the code.

# Example 1

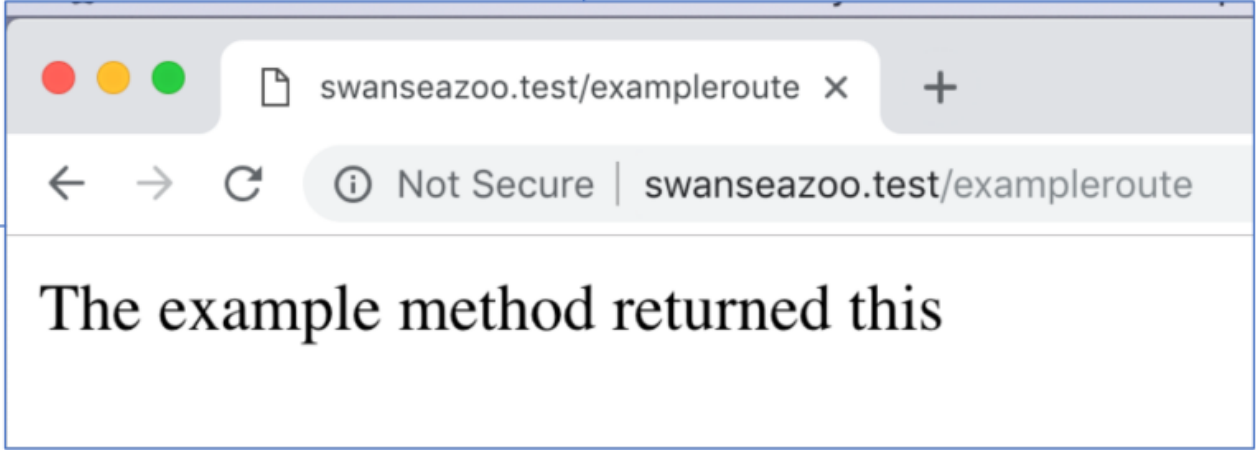
---

# Swansea Zoo: Test Route

```
Route::get('exampleroute', 'AnimalController@exampleMethod');
```

```
class AnimalController extends Controller
{
    public function exampleMethod() {
        return "The example method returned this";
    }
    ...
}
```

We have setup a single route being serviced by a method in a controller.



The screenshot shows a web browser window with a single tab titled 'swanseazoo.test/exampleroute'. The address bar shows the URL 'swanseazoo.test/exampleroute' with a 'Not Secure' warning. The page content displays the text 'The example method returned this'.

The example method returned this

# Let's Add a Parameter to the Method

```
public function exampleMethod($foo) {  
    return "The example method returned this";  
}
```

Kind of obvious right?  
We didn't provide the  
parameter.

Symfony \ Component \  
Debug \ Exception \  
FatalThrowableError  
(E\_RECOVERABLE\_ERROR)

Too few arguments  
to function  
App\Http\Controll  
ers\AnimalController  
::exampleMethod(),  
0 passed and  
exactly 1 expected



/home/vagrant/Laravel/swanseazoo/app/Http/Controllers/AnimalController.php

```
2.  
3. namespace App\Http\Controllers;  
4.  
5. use App\Animal;  
6. use App\Enclosure;  
7. use Illuminate\Http\Request;  
8.  
9. class AnimalController extends Controller  
10. {  
11.  
12.     public function exampleMethod($foo) {  
13.         return "The example method returned this";  
14.     }  
15.  
16.     /**  
17.      * Display a listing of the resource.
```

# Let's Add a Parameter to the Method: Type Hinting

This time lets hint at the type we expect the parameter to be:

```
public function exampleMethod(Animal $foo) {  
    dd($foo);  
    return "The example method returned this";  
}
```



What do you think happens now?

We still haven't provided a argument anywhere ...



# Let's Add a Parameter to the Method: Type Hinting

- But it works!
- We get an instance of Animal.
- Laravel saw that we wanted an instance of animal and made a new instance for us.
  - Notice how the attributes are empty.
- This is the main idea: when we need instances then the framework **provides (resolves)** them and we don't have to **instantiate** them manually.

```
Animal {#211 ▼  
  #connection: null  
  #table: null  
  #primaryKey: "id"  
  #keyType: "int"  
  +incrementing: true  
  #with: []  
  #withCount: []  
  #perPage: 15  
  +exists: false  
  +wasRecentlyCreated: false  
  #attributes: []  
  #original: []  
  #changes: []  
  #casts: []  
  #dates: []  
  #dateFormat: null  
  #appends: []  
  #dispatchesEvents: []  
  #observables: []  
  #relations: []  
  #touches: []  
  +timestamps: true  
  #hidden: []  
  #visible: []  
  #fillable: []  
  #guarded: array:1 [▶]  
}
```

# Example 2

---

# Service Container - Accessing


- As it turns out the `/routes/web.php` file is a good place to play around in Laravel.
  - Code we add to it is run each time we run the application.
- We can access the single instance of the Service Container via the `app()` function:
- So lets add these lines to `routes/web.php` at the top:

```
dd(app());
```

```
Application {#2 ▼  
  #basePath: "/home/vagrant/Laravel/swanseazoo"  
  #hasBeenBoostrapped: true  
  #booted: false  
  #bootingCallbacks: array:1 [▶]  
  #bootedCallbacks: []  
  #terminatingCallbacks: []  
  #serviceProviders: & array:22 [▶]  
  #loadedProviders: array:22 [▶]  
  #deferredServices: array:102 [▶]  
  #databasePath: null  
  #storagePath: null  
  #environmentPath: null
```

# Service Container - Binding

- Lets see how to bind data into the service container.
  - We will be able to retrieve what we bind later on.
- First lets create a new class to play with – just a plain class like in Java – not a model.
  - Twitter – supposed to represent some object that can send tweets.
- We could not instansiate this in our controllers and use it as normal.



```
Twitter.php x
1 <?php
2
3 namespace App;
4
5 class Twitter
6 {
7 }
```

# Service Container – Binding and Resolving

In the routes/web.php file we play with binding:

```
use App\Twitter;  
  
app()->bind('twitter', function ($app) {  
    return new Twitter();  
});  
  
$t = app()->make('twitter');  
dd($t);
```

Please bind this function to the key 'twitter'.

The function returns a new instance of the Twitter class

Resolve, via the service container, the object with the key 'twitter'.

```
Twitter {#131}
```

We successfully got a new instance

# Service Container – Resolving Twice

- What if we resolve twice?

```
$t = app()->make('twitter');  
$t2 = app()->make('twitter');  
dd($t, $t2);
```

```
Twitter {#131}  
Twitter {#132}
```

We got instances.

The grey numbers represent a human readable unique ID of the objects.

So the function that we bound to 'twitter' in the service container was run twice.

# Service Container – Singletons

- The Service Container can also register singletons:

```
app()->singleton('twitter', function ($app) {  
    return new Twitter();  
});
```

- Look what happens now:

```
$t = app()->make('twitter');  
$t2 = app()->make('twitter');  
dd($t, $t2);
```

```
Twitter {#131}
```

```
Twitter {#131}
```

We resolved the same instance twice.

The function only ran once. After the first run, Laravel kept track of the new instance of Twitter and will use it for each further resolution.

Why is this so powerful?

We can register a key and use it to provide a preconfigured instance

Anywhere in the code we can resolve it without needing to reconfigure it

In the above example that means we write the logic to interact with the twitter API once and then we can assess it in any controller or view we like!



# Back To Twitter Class

- Lets assume our Twitter Class has some config data provided during construction

```
class Twitter
{
    private $apiKey;

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

- Main idea:
  - Twitter is provided an API Key when constructed.
  - This API Key is used to access the service
    - It is like a password.

# Whoops it Broke

```
app()->singleton('twitter', function ($app) {  
    return new Twitter();  
});  
  
$t = app()->make('twitter');  
$t2 = app()->make('twitter');  
dd($t, $t2);
```

It Broke:

- Our function didn't provide the key!

Symfony \ Component \  
Debug \ Exception \  
**FatalThrowableError**  
(E\_RECOVERABLE\_ERROR)

Too few arguments  
to function  
App\Twitter::\_\_con  
struct(), 0 passed in  
/home/vagrant/Lar  
avel/swanseazoo/r  
outes/web.php on  
line 17 and exactly  
1 expected

/home/vagrant/Laravel/swanseazoo/app/Twitter.php

```
1. <?php  
2.  
3. namespace App;  
4.  
5. class Twitter  
6. {  
7.     private $apiKey;  
8.  
9.     public function __construct($apiKey) {  
10.         $this->apiKey = $apiKey;  
11.     }  
12. }
```

Arguments

# Fixing it: Let's Provide the Key

- Now we provide the key in the function that we bind to the service container.
- Each time we resolve we get our single – preconfigured – twitter instance.

```
app()->singleton('twitter', function ($app) {  
    return new Twitter('Some Secret Key');  
});  
  
$t = app()->make('twitter');  
$t2 = app()->make('twitter');  
dd($t, $t2);
```

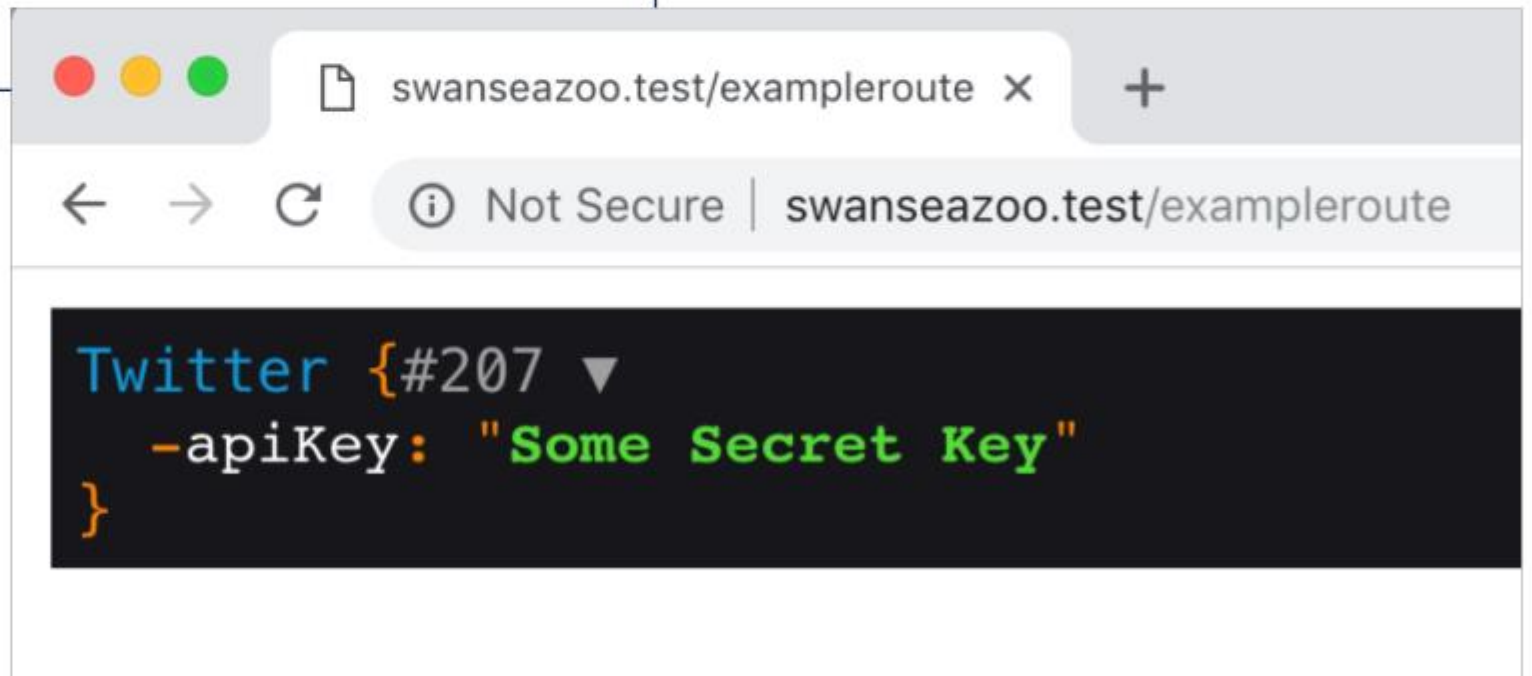
```
Twitter {#131 ▼  
    -apiKey: "Some Secret Key"  
}
```

```
Twitter {#131 ▼  
    -apiKey: "Some Secret Key"  
}
```

- We can resolve in our controllers:

```
public function exampleMethod(Animal $foo) {  
    $t = app()->make('twitter');  
    dd($t);  
    return "The example method returned this";  
}
```

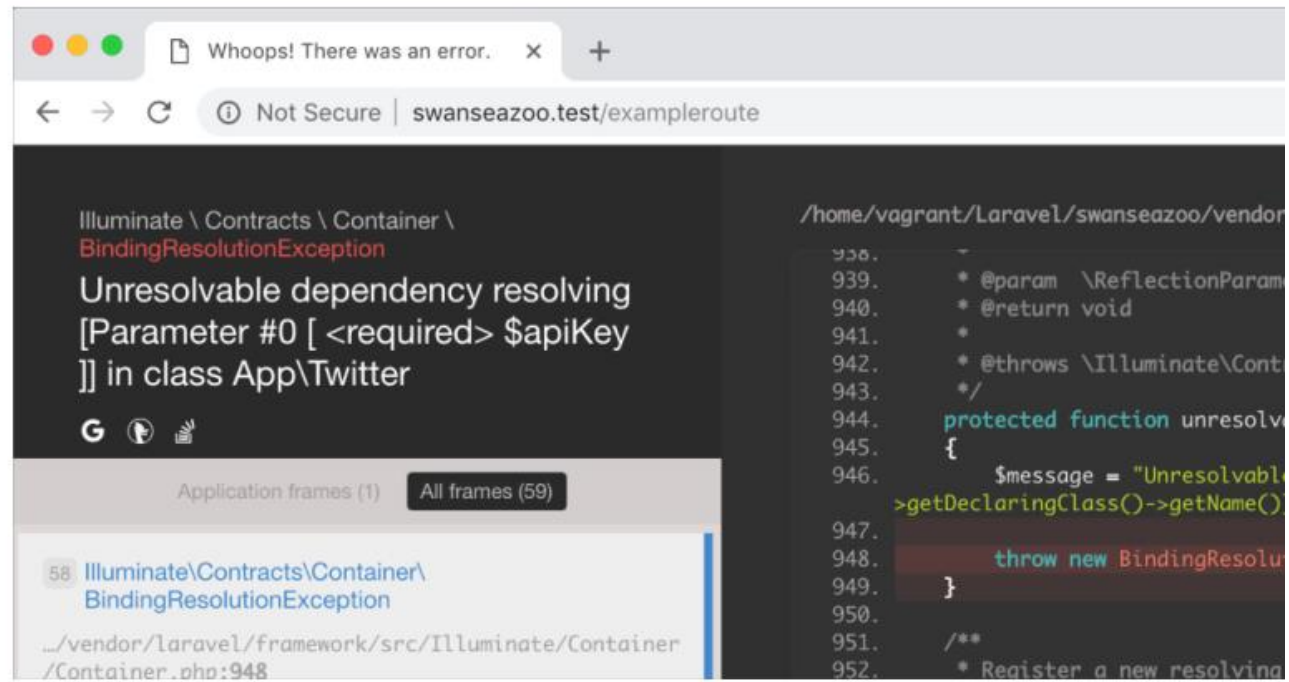
- But we can go further use type hinting too.
  - Next Slide.



# Twitter Using Type Hinting – Attempt 2

- Lets access the Twitter instance through type hinting.
- Boom it failed:
  - Of course it did.
  - We have not configured a Key of 'Twitter'.
  - So Laravel tried to create a new one (and couldn't provide a key).

```
public function exampleMethod(Animal $foo, Twitter $t) {  
    dd($t);  
    return "The example method returned this";  
}
```



# Resolving

- When Laravel resolves it does the following:
  - If there something bound in the Service Container with the same name.
    - If so use that.
    - If not, then look if there is a normal Class with that name.
      - If so, try to instantiate it.
      - Otherwise, fail.



## Let's Change the Key in the Service Provider

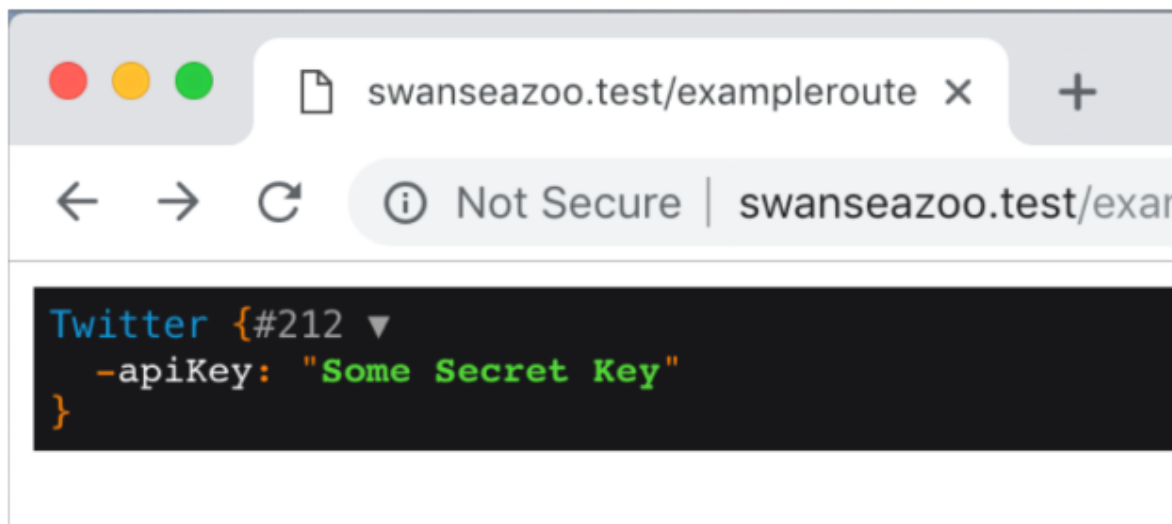
- If we use the full Class Name in the service provider then this would work:

```
app()->singleton('App\Twitter', function ($app) {  
    return new Twitter('Some Secret Key');  
});
```

# Let's Change the Key in the Service Provider (Cont.)

- Now for the route `exampleroote` (serviced by `exampleMethod`) we get:

```
public function exampleMethod(Animal $foo, Twitter $t) {  
    dd($t);  
    return "The example method returned this";  
}
```



Just by type hinting, we have been given a preconfigured instance of Twitter.

We can do this anywhere in the code.



# Why is this Powerful? (Again)

Using dependency injection (which is what the above pattern is called) means if you want to change (in our example) your Twitter implementation you just need to change the binding function for the service container

That means you can easily switch to a testing method which doesn't really tweet, or change the API key if you change twitter accounts with a single line of code

Also, imagine writing a multiplatform game (not web apps) – each platform has it's own achievement system. You can just write a general interface to achievements which is accessed using dependency injection, then set build options that change the achievement class depending on what platform you are targeting at that time.