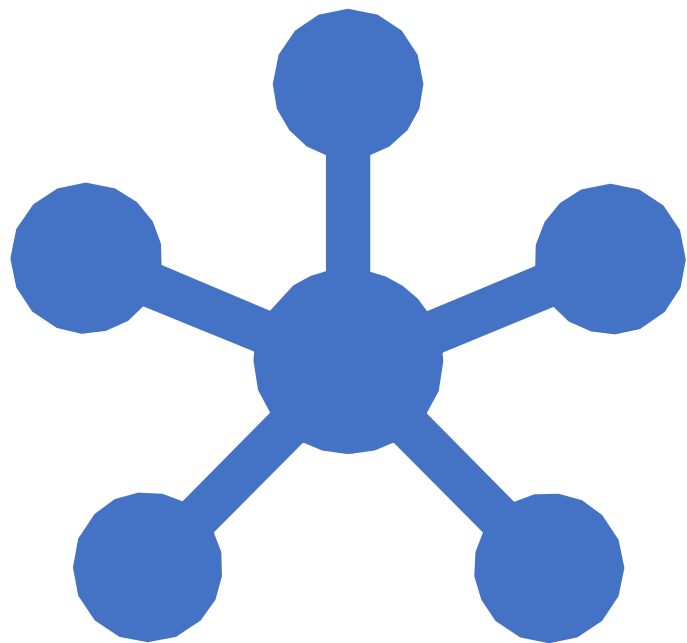


Middleware and Applying Authentication

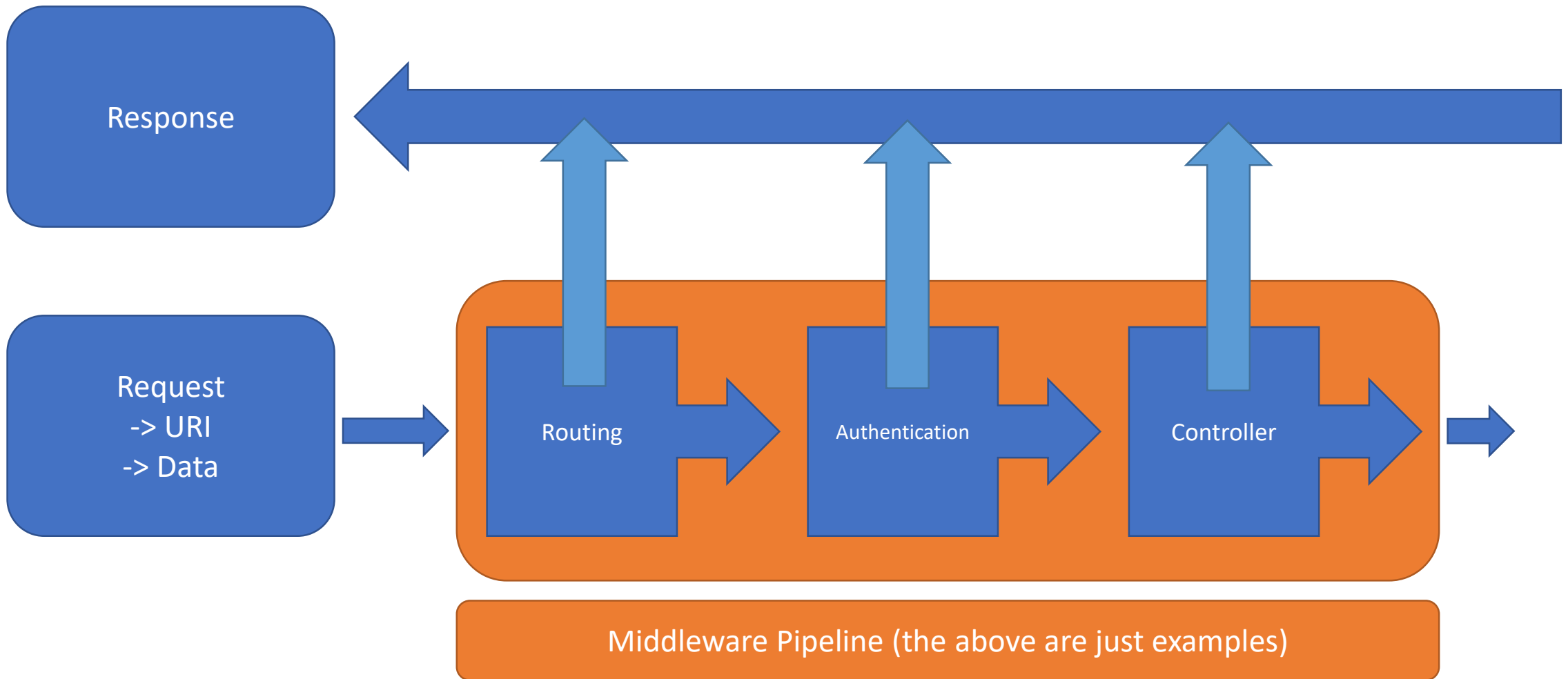


Authentication

- Laravel comes with great authentication out of the box with almost zero work needed
- In seconds you will get a full authentication system
- The question is how do we use this system to protect particular routes
 - ...so that only authenticated users can access them
- We could put logic in each controller method
- But this would end up with lots of copied code, and we would be mixing the authentication logic with our business logic
- Instead we use middleware.

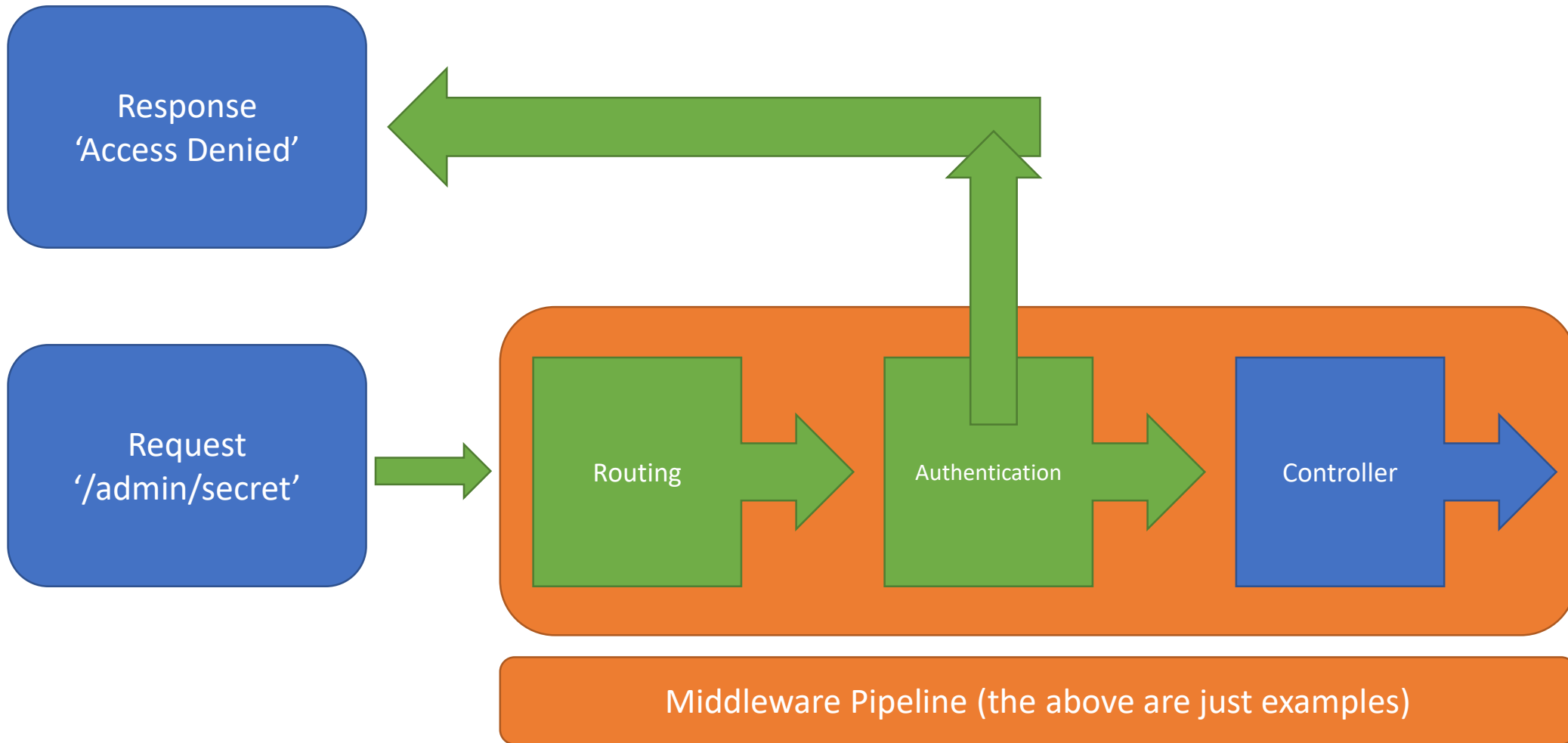


Middleware Recap



A request enters the middleware pipeline (through index.php) and is passed along until one of the middleware packages creates a response.

If a request gets all the way through the pipeline with no request generated an error is sent to the user.

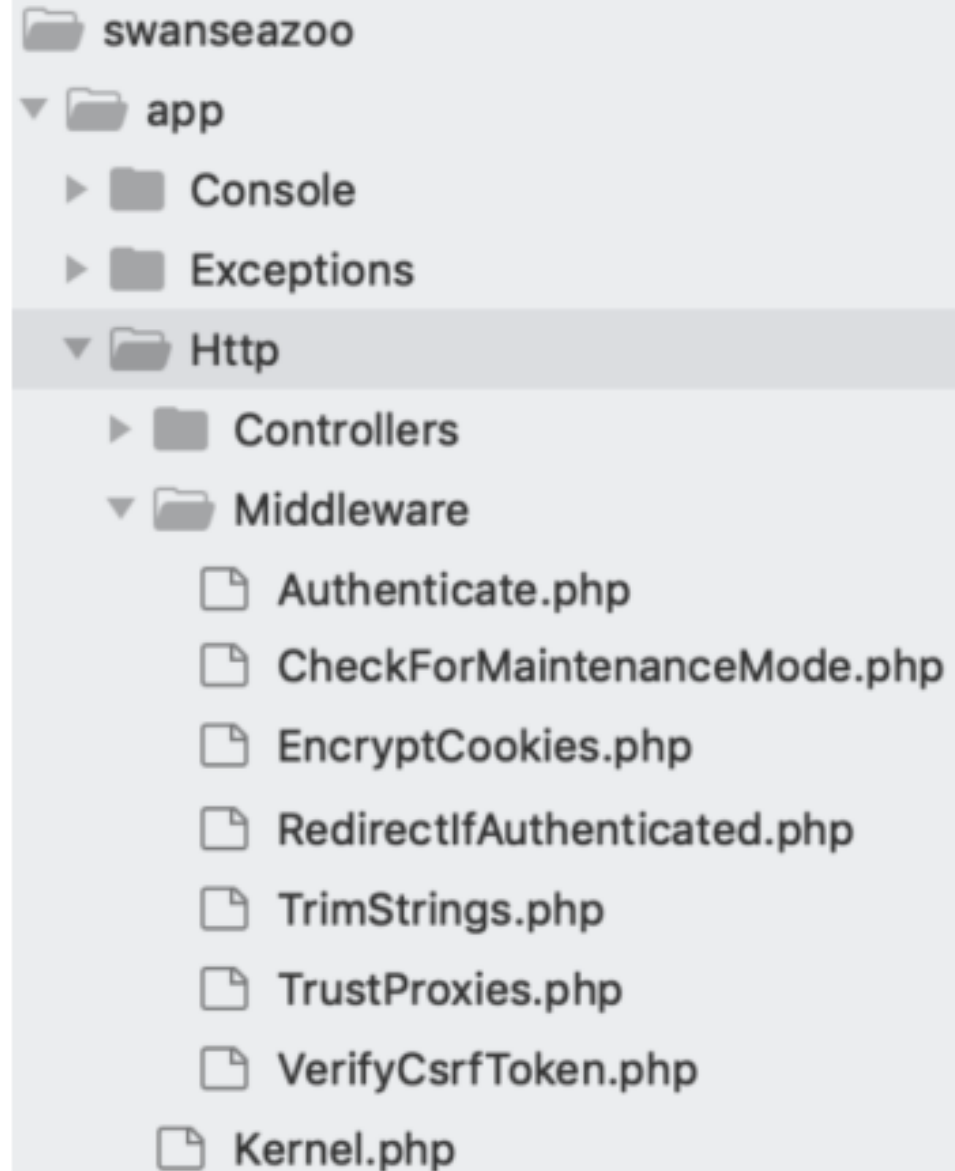


For example, if you request a URI you are not authorized for your request will get as far as authentication then a 'Access Denied' response will be created and sent back to you.

...or it might pass the request on to a 'login' controller which gives a 'please login' type response.

Middleware

- Middleware lives in
App\Http\Middleware
- Kernel.php defines the available
middleware.



Kernel.php

- Kernel.php provides a few arrays of middleware:

\$middleware:

```
protected $middleware = [  
    \App\Http\Middleware\CheckForMaintenanceMode::class,  
    \Illuminate\Foundation\Http\Middleware\ValidatePostSize::class,  
    \App\Http\Middleware\TrimStrings::class,  
    \Illuminate\Foundation\Http\Middleware\ConvertEmptyStringsToNull::class,  
    \App\Http\Middleware\TrustProxies::class,  
];
```

This list of middleware is applied to each and every request.

Kernel.php (Cont.)

- Route Middleware:

```
protected $routeMiddleware = [  
    'auth' => \App\Http\Middleware\Authenticate::class,  
    'auth.basic' => \Illuminate\Auth\Middleware\AuthenticateWithBasicAuth::class,  
    'bindings' => \Illuminate\Routing\Middleware\SubstituteBindings::class,  
    'cache.headers' => \Illuminate\Http\Middleware\SetCacheHeaders::class,  
    'can' => \Illuminate\Auth\Middleware\Authorize::class,  
    'guest' => \App\Http\Middleware\RedirectIfAuthenticated::class,  
    'signed' => \Illuminate\Routing\Middleware\ValidateSignature::class,  
    'throttle' => \Illuminate\Routing\Middleware\ThrottleRequests::class,  
    'verified' => \Illuminate\Auth\Middleware\EnsureEmailIsVerified::class,  
];
```

Key –
short
name


Path to actual
middleware

- See these as optional middleware you can selectively apply to routes.
 - Notice 'auth' guess what that does?

So What Does a Middleware Look Like?

- All middleware contains a 'handle' method. This gets called when the middleware should do its job.
- It is passed the HTTP request and the next middleware to call.
- The handle method can change the request, return an early http response or pass the request to the next middleware.

```
class MyMiddleware
{
    public function handle($request, Closure $next)
    {
        ...
        ...
        ...
        return $next($request);
    }
}
```



Pass the request to next middleware

The Auth Middleware

Protecting Routes

- There are two ways of protecting routes (developers can't decide which is best unfortunately) :
 - In the constructor of the controller
 - Explicitly in the routes file
- I would recommend putting it in the routes file
 - It keeps your controller classes clean and concerned with just controller related stuff
 - You can see all your routes and which are protected or not in a single file which makes it easier to maintain

Protecting Routes Via Constructor in Controller

- Lets take a look at the constructor of HomeController

```
public function __construct()  
{  
    $this->middleware('auth');  
}
```

- When we construct an instance of HomeController the Auth middleware is inserted automatically.
 - This means that all methods in this controller now require authentication.
 - If you are not authenticated then you get redirected to login.

Protecting Routes Via Constructor in Controller (2)

- But what if we only wanted some methods in a controller.

- We can do

```
$this->middleware('auth')->except('x');
```

to apply auth to every method except x.

- We can also do

```
$this->middleware('auth')->only('x', 'y');
```

to apply auth to only methods except x and y.

The first example is white listing, by default each method is protected and we white list the ones we don't want protected. This is more secure – **why?**

In Routes File

- We can also apply middleware in the routes file.

```
Route::get('animals/create', 'AnimalController@create')  
    ->name('animals.create')->middleware('auth');
```

```
Route::post('animals', 'AnimalController@store')  
    ->name('animals.store')->middleware('auth');
```

- Now we can keep the middleware with the routes!

Guest Middleware

- There is also **guest** middleware.
- This does the opposite of auth.
- You have to not be logged in.
- Useful for the register form. Logged in users should not be able to see it, you have to be a guest.



Creating Custom Middleware

Creating Custom Middleware

- Our aim is to create some middleware which checks if the HTTP request asks please
- Specifically:
 - If the HTTP request has a query parameter named 'please' (with any value) then it is fine to proceed as normal
 - Else we return a message saying 'you have to ask nicely'
- Step 1: Create the middleware with artisan... 'artisan make:middleware CheckPolite'
- This creates a file `app\Http\Middleware\CheckPolite.php`

Let's Make Our Own Middleware (Cont.)

- Now we must write the handle method:

```
public function handle($request, Closure $next)
{
    $pleaseExists = $request->exists('please');

    if ($pleaseExists) {
        return $next($request);
    } else {
        return response("You must say the magic word.");
    }
}
```

- If please is present, then proceed to next middleware.
- Otherwise return an early response.

Registering Our Middleware

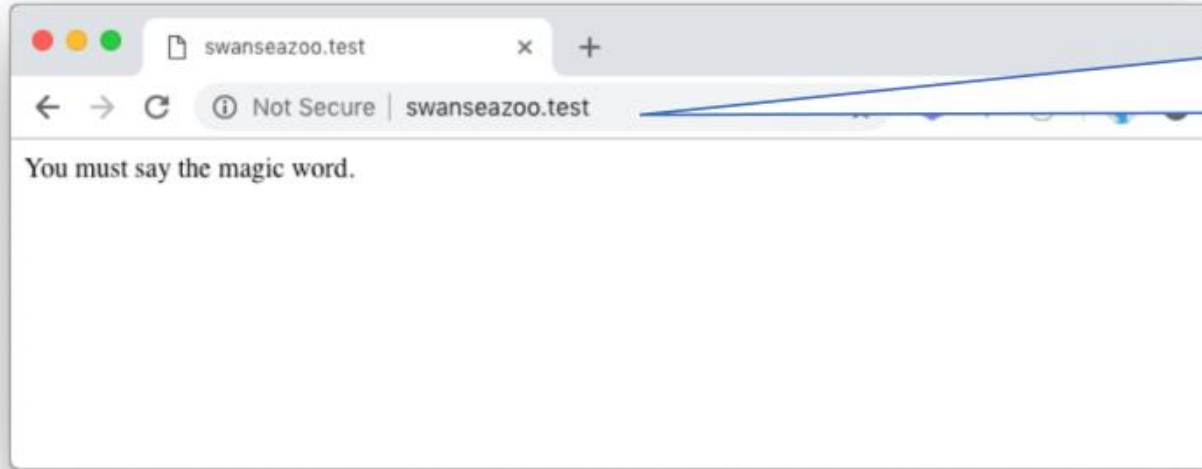
- First we have to register our middleware in Kernel.php:

```
protected $routeMiddleware = [  
    'auth' => \App\Http\Middleware\Authenticate::class,  
    ...  
    'polite' => \App\Http\Middleware\CheckPolite::class,  
];
```

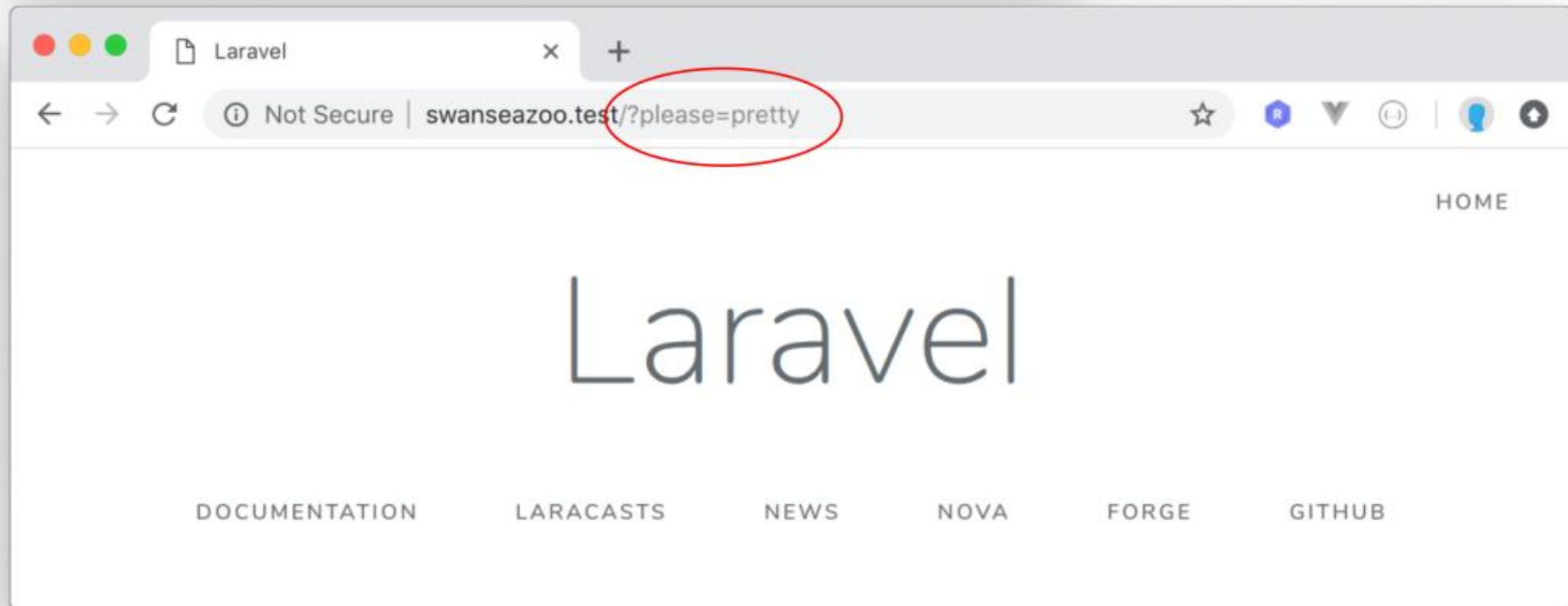
- Then we need to add it to a route in routes/web.php:

```
Route::get('/', function () {  
    return view('welcome');  
})->middleware('polite');
```

Testing Our Middleware



Middleware kicks out out.



Summary

- You now know about middleware and in particular the auth middleware.
- The auth middleware allows us to not write authentication logic all over our controllers.
 - We use middleware to protect the routes.
 - Very neat and clean idea.
- Note: We have not looked at authorization.