www.phparch.com

April 2017
VOLUME 16 - Issue 4

php[architect]

SPRING RENEWAL

PSR-7 HTTP Messages In the Wild

Integrating With APIs

Demystifying Multi-Factor Authentication

Free Sample

ALSO INSIDE

Education Station:
Rock Your Deployments With Rocketeer

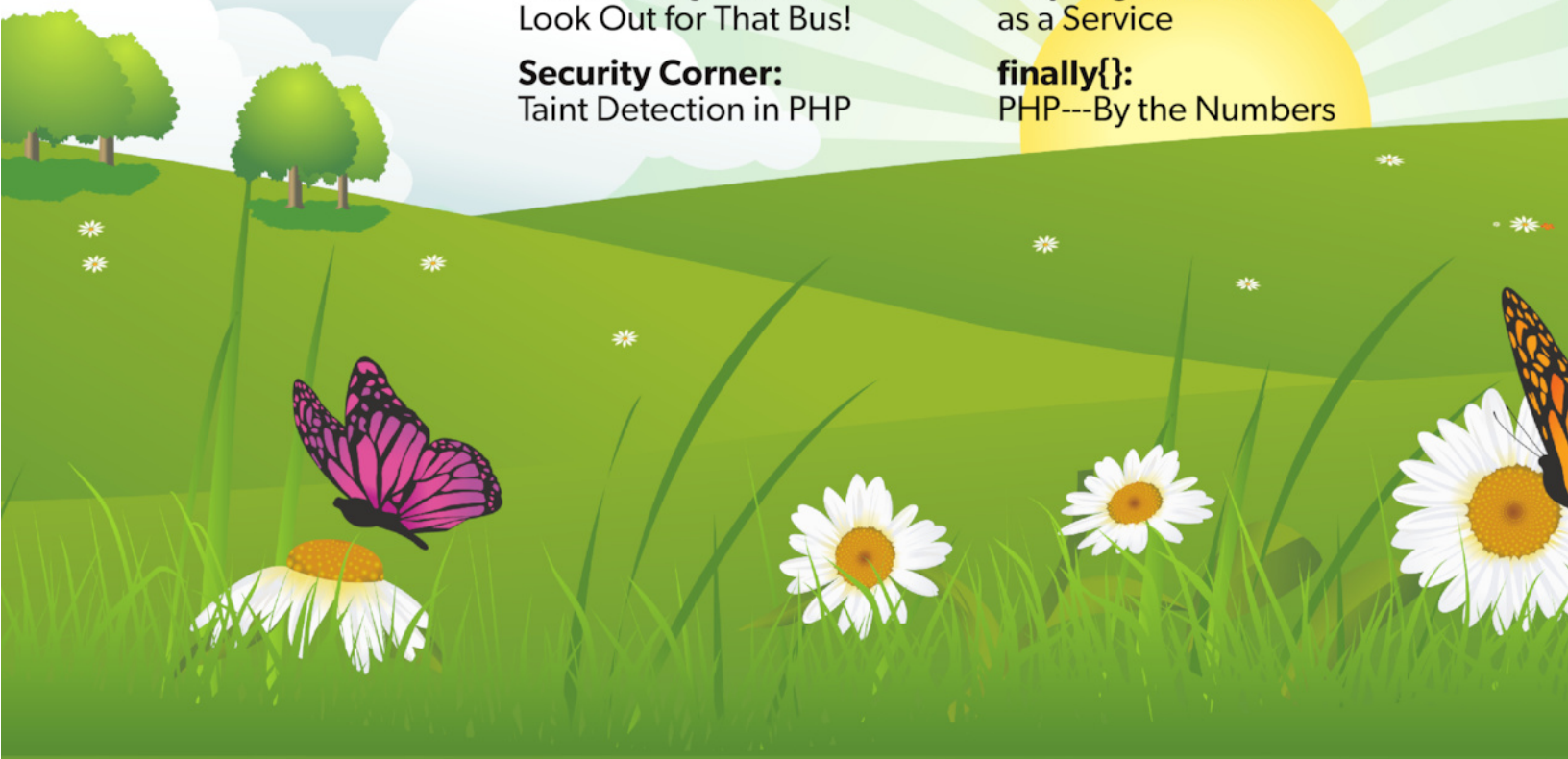
Community Corner:
Look Out for That Bus!

Security Corner:
Taint Detection in PHP

Leveling Up:
Evaluating Value Objects

Artisanal:
Easy Vagrant Environments as a Service

finally{}:
PHP---By the Numbers



PHP[TEK] 2017

The Premier PHP Conference
12th Annual Edition

May 24-26 — ATLANTA

Keynote Speakers:

Gemma Anible
WonderProxy

Keith Adams
Slack

Alena Holligan
Treehouse

Larry Garfield
Platform.sh

Mitch Trale
PucaTrade

Samantha Quiñones
Etsy

Sponsored By:

php women · ShootProof · Zeamster · Drupal Sourcing Mental Illness · platform.sh · stickermule

tek.phparch.com

Demystifying Multi-Factor Authentication

Brian Retterer

Account security is a hot topic among developers and software users. No dev wants to be responsible for the next big "user accounts breached" headline. With every new headline, users are becoming more concerned about their security, and it's our responsibility to create products our users can trust. But trust between your application and your users can be difficult to obtain. One way we gain that trust is by providing a secure browsing experience via an SSL certificate on your server. We also gain trust by securing user passwords using industry standards such as bcrypt.

As developers, we know that one of the best ways to assure account security and grow user trust is by offering or enforcing Multi-Factor Authentication. A frequent recommendation is to "enable two-factor authentication your account." In this article, we'll look at what this means and how you can implement it in your applications.

What Is Multi-Factor Authentication?

There are many different forms of Multi-Factor Authentication. Even the words used to describe it have a few different versions. You may hear people talk about multi factor, 2-factor, MFA, factored authentication, or more. These are all the same idea; you must have something you know and something you have to authenticate and gain access. The **something you know** part comes from your username and password combination where the **something you have** comes in many different forms. This can be a physical device, such as your cell phone, or even a one-time-use code that is emailed to you.

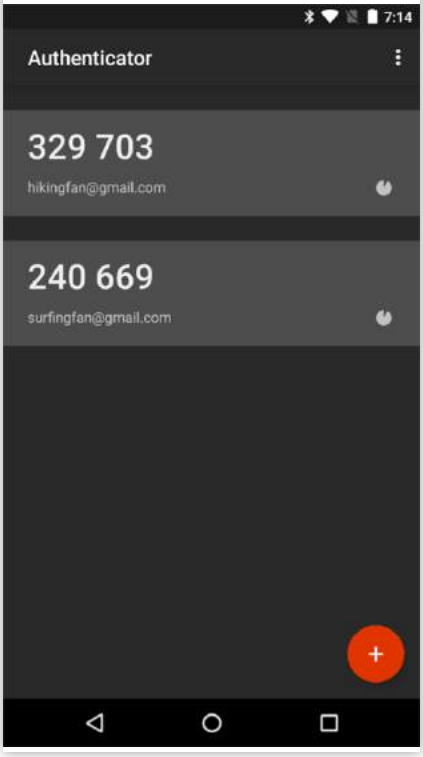
The majority of you have likely experienced Multi-Factor Authentication in some form or another. This is a safe bet for me to make because Multi-Factor Authentication is all around you. If you remember the two main requirements, something you know and something you have, we can apply them to everyday life. A transaction at an ATM fits our criteria. You know your pin number, and you have your debit card.

In the technology world, we have a few options available to us. The aforementioned case of an email being sent with a one-time-use code is one of the most common ways of implementing MFA. Although this is the method I'll be showing you how to set up, I'd be remiss if I didn't cover the other secure options that you have as a developer. A newer standard for handling MFA is to allow the use of a Time-Based one-Time Password, or TOTP. The most common method of implementation would be to use Google Authenticator.

Google Authenticator is a trust between a device and the authentication server. Three items are working together to create a successful login. A shared secret encoded with the Base32 algorithm that your device and the authentication server know, the current time, and a signing function. Once your application server creates the secret, and allows the user to input, or scan a QR code, the user will be given a 6-digit code which is only good for one minute (plus or minus a leeway) that can be used as part of their login.

Another form of MFA you may be more familiar with is a code being sent over SMS. This method has come under scrutiny over the last few years as it is becoming less and less secure, given someone could intercept the code over SMS or pose as your service provider. However, it works essentially the same as all of the other methods. A code

Figure 1.



which is associated with the user trying to log in is sent to the user's registered cell phone number. During the login, they are asked for this code. Once the code has successfully been used, or unsuccessfully used after a set amount of time (usually one to five minutes) the code is removed, and a new one will be generated at the next login.

> **Why is SMS less secure?** *A determined attacker could hijack the messages. They could socially engineer your service provider to send them to a different phone. Government friendly telecom companies could hand them over to the authorities, and fake cell phone towers could intercept them. For more see So Hey You Should Stop Using Texts for Two-Factor Authentication*[1]

How to Implement MFA In Your Project

Now that you know what MFA is and some of the types available, let's look at how you can implement it in your application. The most universal system for MFA is a one-time-use code that is emailed to the user after a matched username and password combination. This article is going to be using a basic scaffolding of the Laravel framework. We will be using the `make:auth` command to provide some basics and will modify from there. If you are unfamiliar with Laravel, the general concept will be the same, but you may need to research how to integrate it with your specific framework. The final code for this project can be cloned from bretterer/email-multi-factor-authentication[2].

To get started, let's create a new Laravel project:

```
composer create-project laravel/laravel \
        email-multi-factor-authentication
```

Once the Laravel project has been created, we need to initialize the authentication system:

```
php artisan make:auth
```

Now, if you were to visit the site in the web browser, you would see our login and register views. We will need to set up some database connections as well as run our migrations.

In your `.env` file:

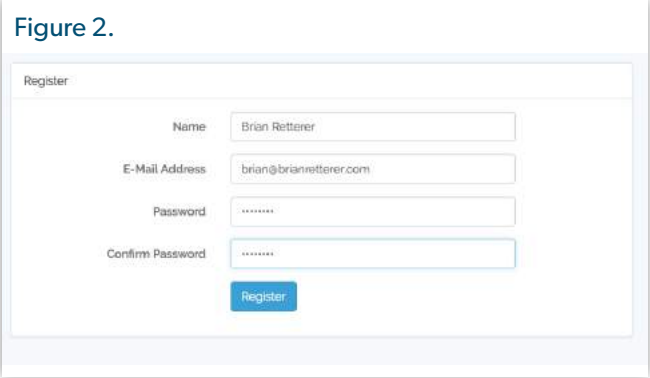
```
DB_CONNECTION=sqlite
MAIL_DRIVER=log
```

Then run:

```
php artisan migrate
```

Once you have all of that setup, you should be able to go to `/register` and register for a new account. You'll see the login form in Figure 2.

Figure 2.



Since Laravel uses the email address as the default for the username field, we have everything we need to continue with our modifications to require a second factor for authentication. Looking back, what we need to have for a factored authentication with email is: (a) something the user knows and (b) something the user has. The user knows their username and password, so we need to provide a code to their email, something they have, so they can fully log in. Since we want the codes to expire after a set amount of time, let's say five minutes, we'll need to create a new table to store all the codes. Let's create and run a new migration as in Listing 1.

Listing 1.

```
1. public function up()
2. {
3.     Schema::create('mfaCodes', function (Blueprint $table) {
4.         $table->increments('id');
5.         $table->integer('user_id');
6.         $table->string('code');
7.         $table->timestamp('expires');
8.         $table->timestamps();
9.     });
10. }
```

First, stub out the migration class file, which will be in the `database/migrations` directory. Then update the `up` method.

```
php artisan make:migration create_mfa_codes_table
```

Then, run the migration:

```
php artisan migrate
```

The remainder of this article assumes you know how to use artisan to scaffold any code we need. If stuck, refer to the project on GitHub.

Next, need to "hijack" the login Laravel does for us since we don't want to set any of the cookies for the user until they successfully input the code that was emailed to them. To do so, we will need to create our own login method which validates the user credentials, and if it is valid, sends an email to the user with a random code which we store in our database for later use (See Listing 2).

1 *So Hey You Should Stop Using Texts for Two-Factor Authentication:* http://phpa.me/wired-sms-2fa

2 *bretterer/email-multi-factor-authentication:* https://github.com/bretterer/email-multi-factor-authentication

Listing 2.

```
1. public function login(Request $request)
2. {
3.     $this->validateLogin($request);
4.
5.     if ($this->hasTooManyLoginAttempts($request)) {
6.         $this->fireLockoutEvent($request);
7.
8.         return $this->sendLockoutResponse($request);
9.     }
10.
11.    $credentials = $this->credentials($request);
12.    $loginCheck = $this->guard()->validate(
13.        $credentials
14.    );
15.
16.    if ($loginCheck) {
17.        $code = $this->createOneTimeUseCode($credentials);
18.        $this->emailOneTimeUseCode($credentials, $code);
19.
20.        return redirect('/login/challenge');
21.    }
22.
23.    $this->incrementLoginAttempts($request);
24.
25.    return $this->sendFailedLoginResponse($request);
26. }
```

The majority of the login method will remain the same, but where we need to take over is using the `validate()` method on the `guard` class instead of the `attempt()` method. You will see these changes on line 11. You will also see we need a new method to save a new code into our table which will be used to email, as well as a method to actually email the user. Now, both of these methods (see Listing 3) are inside of the `LoginController`, but there is no reason they have to live there. You can place these anywhere in your application.

Listing 3.

```
1. private function createOneTimeUseCode($credentials) {
2.     $randomNumber = random_int(100000,999999);
3.
4.     $user = User::where('email', $credentials['email'])
5.                 ->first();
6.
7.     DB::table('mfaCodes')->insert([
8.        'user_id' => $user->id,
9.        'code' => $randomNumber,
10.       'expires' => time() + 300
11.    ]);
12.
13.    return $randomNumber;
14. }
15.
16. private function emailOneTimeUseCode($credentials, $code) {
17.    Mail::to($credentials['email'])
18.        ->send(new LoginCode($code));
19. }
```

Listing 4.

```
1. class LoginCode extends Mailable
2. {
3.     use Queueable, SerializesModels;
4.
5.     /**
6.      * Create a new message instance.
7.      *
8.      * @return void
9.      */
10.    public function __construct($code) {
11.        $this->code = $code;
12.    }
13.
14.    /**
15.     * Build the message.
16.     *
17.     * @return $this
18.     */
19.    public function build() {
20.        return $this->view('mail.logincode')
21.            ->with([
22.               'code' => $this->code
23.            ]);
24.    }
25. }
```

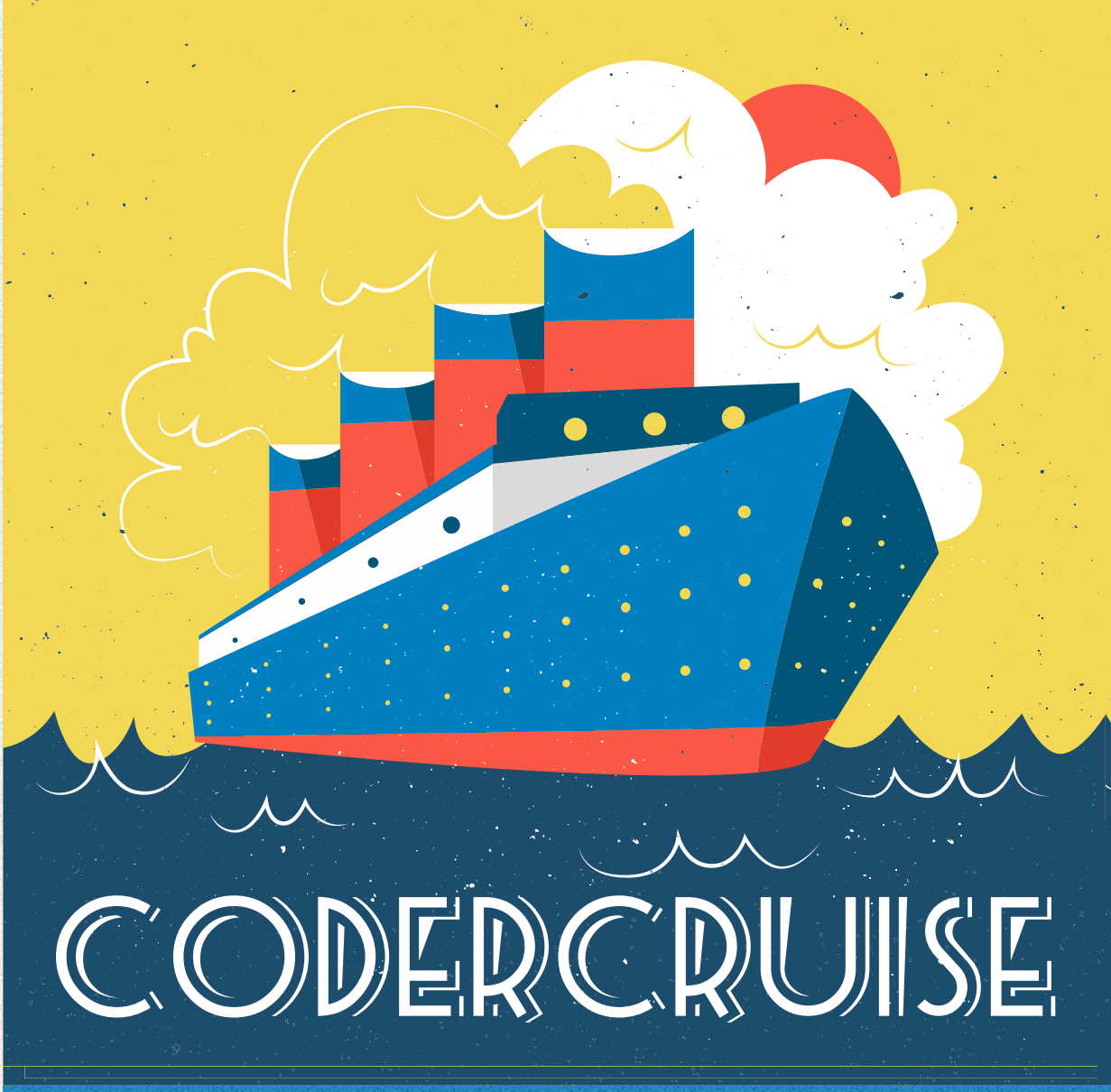
We also need to add a new route, as in Listing 4, for the `/login/challenge` endpoint where our form will live for the user to enter the code sent to their email. We also need a new `Mailable` class that will handle the email sending. If you are unsure what the Laravel `Mailer` class is, or how to use it, you can find more information in the Laravel documentation[3].

We are now in the home stretch of the setup for our emailed code factored authentication. The last step is handling a way for the user to enter the code we just sent them. We'll need a few additions to our login controller, a couple of routes, and a new view. Let's begin with the view, shown in Listing 5. Copy the current login view and modify it slightly since it is close to what we need. Take out the password field, and change the email field over to a code text field. Make sure to also change all the error checks over to a key of `code`. You should also remove any of the extra links on the page that are no longer needed, such as the "forgot password" link.

Next, we add the routes:

```
1. Route::get(
2.     '/login/challenge',
3.     'Auth\LoginController@challenge'
4. );
5. Route::post(
6.     '/login/challenge', [
7.        'uses'=>'Auth\LoginController@validateChallenge',
8.        'as' => 'login.challenge'
9. ]);
```

3 *Laravel documentation:* https://laravel.com/docs/5.2/mail



CODERCRUISE

7 days at sea, 3 days of conference

Leaving from New Orleans and visiting
Montego Bay, Grand Cayman, and Cozumel

July 16-23, 2017 — *Tickets $295*

www.codercruise.com

Presented by One for All Events

And, finally, we add the methods in Listing 6 to our controllers.

The `validateChallenge` method is where all the checks happen. The first thing we do is remove any expired codes from the database to keep things clean. We then find a code in the database that matches the code submitted, also checking to make sure the code is not expired. If a code is found, we get the user ID associated with the code and log the user in using the built in Laravel functionality. If a code is not found, we send the user back to the challenge page with an error.

There are a few flawed items in this example. For instance, we are not asking for the email again from the user. This makes this system a little more vulnerable to brute force attacks, meaning attempts could be made against the challenge page over and over with different numbers until they get one that works. One way to fix this would be asking for the email address again for the code and seeing if it matches the expected user. Another option would be to set a cookie during the first step of the login with a unique code, and then in the email you send, have a link that takes the user to a URL with a parameter that has the same code. There are many other options available for helping solve this problem, and I leave it up to you to decide what is best for your application.

Listing 5.

```
1. @extends('layouts.app')
2.
3. @section('content')
4. <div class="container">
5.   <div class="row">
6.     <div class="col-md-8 col-md-offset-2">
7.       <div class="panel panel-default">
8.         <div class="panel-heading">Multi Factor Code</div>
9.         <div class="panel-body">
10.          <form class="form-horizontal" role="form"
11.                method="POST"
12.                action="{{ route('login.challenge') }}">
13.            {{ csrf_field() }}
```

For the full listing, see this month's code archive

Listing 6.

```
1. public function challenge() {
2.    return view('auth/challenge');
3. }
4.
5. public function validateChallenge(Request $request) {
6.    DB::table('mfaCodes')->where('expires', '<', time())
7.                         ->delete();
8.    $code = $request->get('code');
9.
10.   $codeEntry = DB::table('mfaCodes')
11.                  ->where('code', $code)
12.                  ->where('expires', '>', time())
13.                  ->first();
14.
15.   if ($codeEntry) {
16.       $this->guard()->loginUsingId($codeEntry->user_id);
17.       return $this->sendLoginResponse($request);
18.   }
19.
20.   return redirect()->back()
21.                    ->withErrors([
22.                       'code' => 'That code is not valid'
23.                    ]);
24. }
```

What's Next?

There are many places you can go with this. Although I used Laravel for the sample code, the same logic can be used in virtually any system with authentication.

The sample provided is what I like to call a poor man's version of Multi-Factor Authentication, mainly because it is not as secure as some other options out there. Setting up some of the other options, such as Google Authenticator, take much of the same processes as the sample above. To set those up requires a slightly different logic and some extra setup for the end user.

I'd love it if one day every application required some form of factored authentication to help keep my accounts safe. Now that you know the basics of setting this up, go out and re-vamp your applications to offer this extra step of security to all of your users.

Brian Retterer is a developer advocate at Okta, a Silicon Valley cloud-based identity service. He is focused on serving the PHP community, and always excited to represent and educate developers, especially in the Midwest. From his home base in Ohio, you can often find him advocating for best practices in account security and RESTful API design. You can follow Brian on Twitter at @bretterer

Borrowed this magazine?

Get php[**architect**] delivered to your doorstep or digitally every month!

Each issue of php[**architect**] magazine focuses on an important topic that PHP developers face every day.

We cover topics such as frameworks, security, ecommerce, databases, scalability, migration, API integration, devops, cloud services, business development, content management systems, and the PHP community.

php[**architect**]

Digital and Print+Digital Subscriptions Starting at $49/Year

http://phpa.me/mag_subscribe