

CLOSE ENCOUNTERS

with
PHP

Level 4

Composer & Autoloading

Package Management

We Need Better Validation

Our validation works, but is lacking in features. Let's review what we might want.

- Validate the existence of each, but if one is missing we will need to report this to the user
- If the date is not formatted correctly, we will need to inform the user
- If the email is an invalid format, we will need to report this to the user as well

Why Packages?

What is a library, why do we need it, and what is Composer?

- A library (or package) is a collection of code that is meant to serve a single purpose and to be reusable
- Packages are open source, which means any number of developers can contribute, so the package can evolve quickly
- PHP uses a package management tool called Composer
- Composer will allow us to define our libraries for each project and use them almost anywhere in our code

How Do We Install Composer?

The best way to install Composer is to use the command line.

- In the terminal you will use these commands:

```
→ ~ php -r "copy('https://getcomposer.org/installer', 'composer-setup.php');" 
```

copy downloads the installer and renames it!

We are running php as a command-line tool

All these commands can be found at
<https://go.codeschool.com/composer-install>

How Do We Install Composer?

The best way to install Composer is to use the command line.

- In the terminal you will use these commands:

→ `~ php -r "copy('https://getcomposer.org/installer', 'composer-setup.php');"`

→ `~ php -r "if (hash_file('SHA384', 'composer-setup.php') === 115a8dc7871f15d8531`

This long command verifies our installer

All these commands can be found at
<https://go.codeschool.com/composer-install>

How Do We Install Composer?

The best way to install Composer is to use the command line.

- In the terminal you will use these commands:

→ `~ php -r "copy('https://getcomposer.org/installer', 'composer-setup.php');"`

→ `~ php -r "if (hash_file('SHA384', 'composer-setup.php') === 115a8dc7871f15d8531`

→ `~ php composer-setup.php`

→ `~ php -r "unlink('composer-setup.php');"`

Now we will run our installer, then delete it

All these commands can be found at
<https://go.codeschool.com/composer-install>

How Do We Install Composer?

The best way to install Composer is to use the command line.

- In the terminal you will use these commands:

→ `~ php -r "copy('https://getcomposer.org/installer', 'composer-setup.php');"`

→ `~ php -r "if (hash_file('SHA384', 'composer-setup.php') === 115a8dc7871f15d8531`

→ `~ php composer-setup.php`

→ `~ php -r "unlink('composer-setup.php');"`

→ `~ mv composer.phar /usr/local/bin/composer`

Move the Composer file to our /usr/local/bin folder

All these commands can be found at
<https://go.codeschool.com/composer-install>

Finding Packages

We can use the Composer command to search for packages.

- In the terminal you will use these commands

→ `~ composer search validation`  *search followed by our query: validation*

`illuminate/validation` The Illuminate Validation package.

`respect/validation` The most awesome validation engine ever created for PHP

`siriusphp/validation` Data validation library. Validate arrays, array objects, domain models etc using a simple API. Easily add your own validators on top of the already dozens built-in validation rules

`intervention/validation` Additional Validator Functions for the Laravel Framework


Installing the Validation Package

Using the Composer CLI, we will install the respect/validation package.

- This command will be run in the terminal at the root of our project:

→ `~ composer require respect/validation`

*require will add the package to our
composer.json file and install it*



Using version ^1.1 for respect/validation

./composer.json has been created

Loading composer repositories with package information

Updating dependencies (including require-dev)

– Installing respect/validation (1.1.4)

Writing lock file

Generating autoload files

Composer Folder Structure

Inside a vendor folder, at the root of the project, will be where our packages go.



app



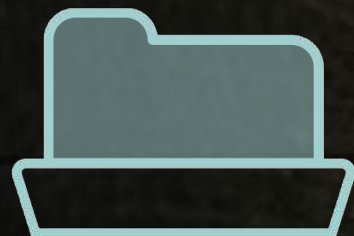
composer.json

The composer.json file defines what packages are needed for the project

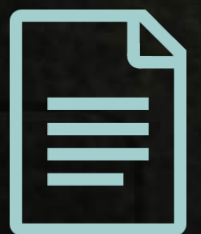


public

All new packages will be installed into the vendor folder



vendor



autoload.php

*The validation package will be installed in the respect folder.
This is related to the package name respect/validation.*



respect

Looking at composer.json

The composer.json file is where our project dependencies are managed.

composer.json

```
{  
  "require": {  
    "respect/validation": "^1.1"  
  }  
}
```

At a minimum, we want version 1.1

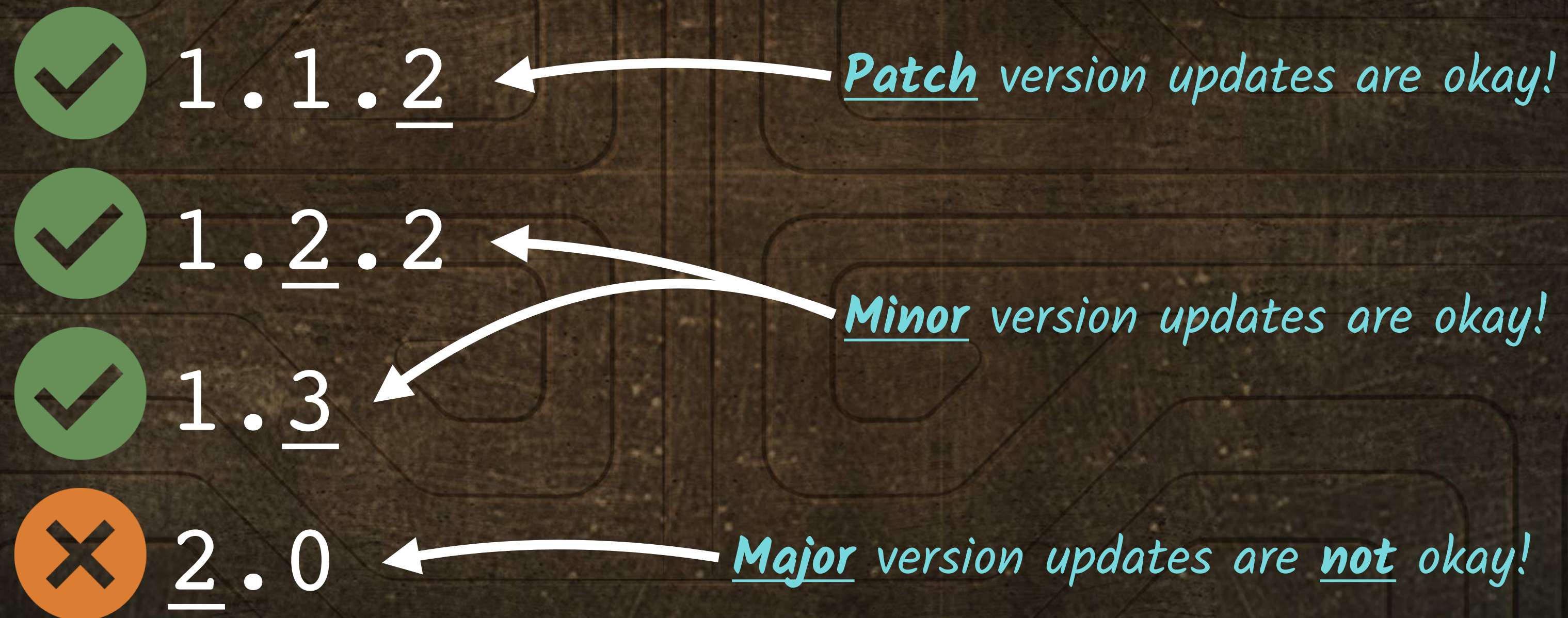
The ^ symbol is a wildcard for next significant release

So far we only have one package required

Semantic Versioning Requirements

Using the ^ symbol, what will we allow if the package gets updated?

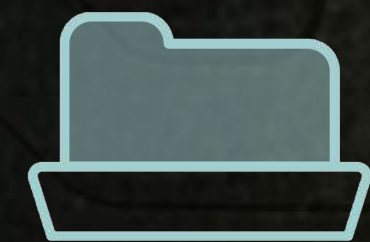
```
"respect/validation": "^1.1"
```



So, download anything newer than version 1.1, but not version 2 or higher!

Composer Provides an Autoloader

Inside a vendor folder, at the root of the project, Composer provides the autoload.php file.

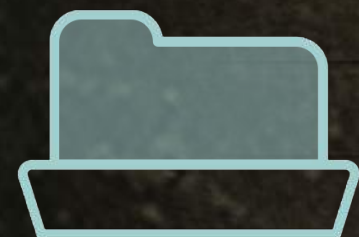


Vendor

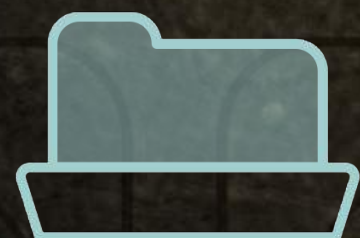


autoload.php

*Inside our vendor folder, we will find the autoload.php file.
This file alone will load all packages in the vendor folder.*



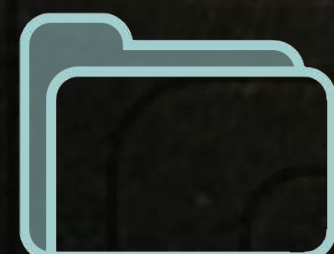
Respect



Validation



docs



library

*The files we will be needing are actually down here, in the
library folder, and autoload will take care of all that!*

CLOSE
ENCOUNTERS
with
PHP

Adding Autoloading From Composer

Requiring the autoload file in our project will give us access to all of the packages.

Add the autoload.php file from our vendor directory.

```
require __DIR__ . '/../vendor/autoload.php';
```



The autoload.php file will automatically give us access to all of the packages within the Composer vendor directory.

Adding Autoloading

app/src/app.php

```
<?php
require __DIR__ . '/../../vendor/autoload.php';
require __DIR__ . '/validation.php';
...
if (!empty($date) && !empty($email) && !empty($description)) {
    echo validate_date($date);

    if (filter_var($email, FILTER_VALIDATE_EMAIL)) {
        echo "<p>Email: $email</p>";
    }
    echo '<p>' . htmlspecialchars($description) . '</p>';
}
}
```

Add the autoload.php file from our vendor directory

We can now use packages anywhere below, including in our validation file

Using Respect/Validation

The use command is how we are able to load libraries.

app/src/validation.php

```
<?php
use Respect\Validation\Validator;

function validate_date($date_string)
{
    if ($time = strtotime($date_string)) {
        return date('F jS Y', $date_string);
    } else {
        return $date_string . ' does not look valid.';
    }
}
```

*Using the namespace of Respect\Validation,
we can access the Validator class*



Using Respect/Validation

The use command is how we are able to load libraries.

app/src/validation.php

```
<?php
use Respect\Validation\Validator;
```

```
$v = new Validator;
```

The new keyword creates a Validator object named \$v

```
function validate_date($date_string)
{
    if ($time = strtotime($date_string)) {
        return date('F jS Y', $date_string);
    } else {
        return $date_string . ' does not look valid.';
    }
}
```


Using Respect/Validation

The use command is how we are able to load libraries.


app/src/validation.php

```
<?php
use Respect\Validation\Validator;
```

```
$v = new Validator;
```

```
var_dump($v);
```

*Let's see what the \$v object looks like
with a var_dump!*



```
function validate_date($date_string)
{
    if ($time = strtotime($date_string)) {
        return date('F jS Y', $date_string);
    } else {
        return $date_string . ' does not look valid.';
    }
}
```


Var Dump of Our Validator

The Validator is an object type, with a protected array of rules. What is all this?!

```
/var/www/hello/app/src/app.php:8:  
object(Respect\Validation\Validator)[3]  
  protected 'rules' =>  
    array (size=0)  
      empty  
  protected 'name' => null  
  protected 'template' => null
```

- We can run validation commands with the validator to test against custom rules
- Each instance of a validator can have a unique name
- Each instance of a validator can also have a template that allows us to customize our error strings
- We can now add some rules to our empty rules array on the Validator

Composer & Autoloading Review

Let's take a quick look back over this lesson in review.

- Composer is a package manager for PHP
- We used the Composer CLI to search and install packages to our application
- We gained access to the package through the use of the **autoload.php** file
- The **use** keyword allows us to access a class through a Namespace/ClassName pattern
- We create new validator instances with the **new** keyword

CLOSE ENCOUNTERS

with
PHP