8

APIs
with Laravel

If you don't fail at least 90 percent of the time, you're not aiming high enough

- Alan Kay

Contents

How To Use This Document

Bits of text in magenta are links and should be clicked at every opportunity. Bits of text in `monospaced yellow` represent code. Most the other text is just text: you should probably read it.

Copying and pasting code from a PDF can mess up indentation. For this reason large blocks of code will usually have a [View on GitHub] link underneath them. If you want to copy and paste the code you should follow the link and copy the file from GitHub.

Taking Notes

In earlier cohorts I experimented with giving out notes in an editable format. But I found that people would often unintentionally change the notes, which meant that the notes were then wrong. I've switched to using PDFs as they allow for the nicest formatting and are also immune from accidental changes.

Make sure you open the PDF in a PDF viewing app. If you open it in an app that converts it into some other format (e.g. Google Docs) you may well miss out on important formatting, which will make the notes harder to follow.

I make an effort to include all the necessary information in the notes, so you shouldn't need to take any additional notes. However, I know that this doesn't work for everyone. There are various tools that you can use to annotate PDFs:

- Preview (Mac)
- Edge (Windows)
- Hypothes.is
- Google Drive (*not* Google Docs)
- Dropbox

Do not use a word processor to take programming notes! (e.g. Google Docs, Word, Pages). Word processors have the nasty habit of converting double-quotes into "smart-quotes". These can be almost impossible to spot in a text-editor, but will completely break your code.

Chapter 1

Classes in PHP

A class is an abstract representation of an object that you want to create. For example, you might have a class `Person` that allows you to create lots of object **instances** representing different people.

Here's a class that represents a person:

```php
<?php

class Person
{
  // declare class properties at the top
  private $firstName;
  private $lastName;
  private $dob;

  // the constructor method
  public function __construct($firstName, $lastName, $dob)
  {
    // we can use $this to reference object properties
    $this->firstName = $firstName;
    $this->lastName = $lastName;
    $this->dob = $dob;
  }

  public function getName()
  {
    // use $this to read properties
    return "{$this->firstName} {$this->lastName}";
  }
```

```
  public function getAge()
  {
    // do some date stuff
    $date = new DateTime($this->dob);
    $now = new DateTime();
    return $now->diff($date)->y;
  }
}
```

[View code on GitHub]

As you can see, it's much the same as a JavaScript class: we have the `class` keyword followed by the name of the class and there are some functions inside the class.

Closing Tags

If you're been writing procedural PHP, you might be used to adding `?>` at the end of any PHP that you write. This allows you to go between HTML and PHP easily. However, when we're writing PHP with classes we'll never mix HTML and PHP.

It's considered good practice to never use a PHP closing tag.

From now on code examples won't include the opening tag, but you will need to add it as the first line in all your files.

However, there are some things we've not seen in JavaScript: the words `private` and `public`; and we declare our properties outside of the constructor method. We'll look at these in more detail shortly.

Here's how we'd use our class:

```
// as in JavaScript, use the "new" keyword to create
// a new object instance
$jim = new Person("Jim", "Henson", "1936-09-24");

// create another Person
$frank = new Person("Frank", "Oz", "1944-03-25");

// each object has its own properties
// so getAge() will return different values for each one
```

```
$jim->getAge();
$frank->getAge();
```

[View code on GitHub]

You can see that where we'd write a dot in JavaScript (`jim.getAge()`), we write an arrow in PHP (`$jim->getAge()`), but otherwise it's almost identical in usage.

As in JavaScript, we call functions that belong to an object **methods** and the values **properties**.

Using `new` Objects

Unlike in JavaScript you can't immediately use a created object:

```
// won't work
new Person("Jim", "Henson", "1936-09-24")->getAge();
```

However, if you really want to, you can get around this with a pair of brackets around the `new` statement:

```
// will work
// but you don't have a reference to the object anymore
(new Person("Jim", "Henson", "1936-09-24"))->getAge();
```

1.1 Properties

In PHP we declare all of the properties that our class uses at the top of the class. This makes it easy to see which values are available. It also allows us to set default values easily:

```
class DatabaseConnection
{
  // properties with sensible defaults
  private $engine = "mysql";
  private $host = "127.0.0.1";
  private $port = 3306;
```

```
  // properties that don't have sensible defaults
  private $username;
  private $password;

  // ... code to do database stuff
}
```

[View code on GitHub]

1.2 $this

Inside our classes we can use the `$this` keyword to access properties and methods that belong to the current object instance. It works in much the same way as JavaScript except that it's much more reliable: `$this` in PHP *always* refers to the current object and has no meaning elsewhere.

```
class Address
{
  private $street;
  private $town;
  private $country;

  // a setter function
  public function setStreet($street)
  {
    // $this represents whichever object we're working on
    $this->street = $street;
    return $this;
  }

  // ...setTown and setCountry as above

  // a getter function
  public function getAddress()
  {
    // you can call methods too
    return implode(", ", $this->assigned());
  }

  private function assigned()
```

```
  {
    return array_filter([
      $this->street,
      $this->town,
      $this->country,
    ]);
  }
}
```

[View code on GitHub]

1.2.1 Returning `$this`

If your method doesn't have anything to return, for example if it just sets a value, then you can return `$this`: it will give back the current object instance to the user, meaning that they can **chain** such methods together:

```
class Address
{
  private $street;
  private $postcode;

  // a setter function
  public function setStreet($street)
  {
    // $this represents whichever object we're working on
    $this->street = $street;
    return $this;
  }

  public function setPostcode($postcode)
  {
    // $this represents whichever object we're working on
    $this->postcode = $postcode;
    return $this;
  }
}


// elsewhere
$address = new Address();
```

```
// because setStreet returns $this we can chain methods
$address->setStreet("1630 Revello Drive")->setPostcode("BS3 9BR");
```

[View code on GitHub]

PSR-2: Coding Style Guide

You've possibly noticed that in all the examples above the opening curly brace { for classes and methods is on its own line. This is part of the PSR-2: Coding Style Guide spec.

If you do an `if` statement (or other control structure) then the opening curly brace, obviously, goes on *the same* line.

You're probably thinking that this doesn't make the slightest bit of sense. And you'd be right. PSR-2 was created by sending round a questionnaire about coding style to 30 or so of most prolific PHP programmers and they just went with whatever the majority said for each point.

But it's the style that everyone uses now. You'll get used to it.

1.3 Visibility

Methods and properties in PHP can have three levels of visibility: **public**, **private**, and **protected**.

A **public** method can be called anywhere in the PHP code. A **public** property can be read and changed from anywhere in the PHP code.

A **private** method can only be called within the class it is declared in. A **private** property can only be read and changed within the class it belongs to.

```
class DatabaseConnection
{
  // properties with sensible defaults
  private $engine = "mysql";
  private $host = "127.0.0.1";
  private $port = 3306;
```

```
  // properties that don't have sensible defaults
  private $db;

  // set the database
  public function setDB($db)
  {
    $this->db = $db;
    return $this;
  }

  // returns the DB connection
  public function connection($username, $password)
  {
    return new PDO(
      $this->pdoSettings(),
      $username,
      $password
    );
  }

  // this method is private - it's only useful inside the class
  private function pdoSettings()
  {
    $parts = implode(";", [
      "host={$this->host}",
      "port={$this->port}",
      "dbname={$this->db}",
    ]);

    return "{$this->engine}:{$parts}";
  }
}
```

[View code on GitHub]

A **protected** property/method can only be used within the class it is declared in and any class that inherits from it (we'll look at inheritance later).

In most instances it's good practice to make all of your properties private and then use "getter" and "setter" methods if you need to be able to change the values outside the class.

1.4 Static Methods & Properties

Classes are primarily used for creating object instances. But sometimes it's useful to write some functionality about the object type instead of object instances.

For example, if we have a `Person` class we might want to write a bit of functionality that gives us an alphabetic array of last names. We could write a `lastNames()` function, but then it's not associated with the `Person` class.

Instead, we will write a `static` method: a method that belongs to the class itself rather than to an object instance.

```
class Person
{
  // static methods (and properties) at top
  public static function lastNames($people)
  {
    $lastNames = array_map(function ($person) {
      // best to use methods to get values
      return $person->lastName();
    }, $people);

    return sort($lastNames);
  }

  // ...non-static methods and properties at bottom
}

// elsewhere, pass in an array of Person objects
Person::lastNames([$oli, $pete, $audrey, $tom]);
```

[View code on GitHub]

Now it is clear that the `lastNames()` method has something to do with `Person` objects.

Paamayim Nekudotayim

The `::` symbol is also known as the "Paamayim Nekudotayim", which is Hebrew for "double colon". This can lead to the somewhat mystifying error:

```
PHP expects T_PAAMAYIM_NEKUDOTAYIM
```

All it's saying is you need a `::` somewhere.

The Zend Engine, which was behind PHP 3.0 and all subsequent releases, was originally developed at the Israel Institute of Technology.

Sometimes it's useful to be able to refer to the class you're currently working in. We can use the `static` keyword to do this:

```
class Person
{
  public static function lastNames($people)
  {
    // call the getLastNames method of this class
    $lastNames = static::getLastNames($people);
    return sort($lastNames);
  }

  // a private static method
  private static function getLastNames($people)
  {
    return array_map(function ($person) {
      return $person->lastName();
    }, $people);
  }

  // ...non-static methods and properties at bottom
}
```

[View code on GitHub]

`static` vs `self`

You will sometimes see `self` instead of `static` to reference the current class. Using `static` in this way was only added in PHP 5.3, so a lot of older code uses `self`.

If you're not using inheritance, then it doesn't make any difference which one you use. If you do then `self` refers to class that it is written in and `static` refers to the class it is called in (which might be different from where it was written if you're using inheritance).

1.5 Additional Resources

- PHP Apprentice: Classes
- Laracasts: Classes 101
- Laracasts: Classes
- PHP Apprentice: Static

Chapter 2

Namespaces

So far we've had to put all of our classes into a single file, when ideally we'd like *one class per file*[1].

2.1 `require_once`

We can import one PHP file into another using the `require_once` keyword:

```
// include the DatabaseConnection class
// assuming it's in the same directory
require_once "DatabaseConnection.php";

// create a new DatabaseConnection object
$db = new DatabaseConnection();

// get the connection
$connection = $db->setDB("tests")->connection("tests", "secret");
```

[View code on GitHub]

We give `require_once` a relative file path and it will be as if the contents of that file are included in place.

[1] It makes it much easier to find what you're looking for

`require` and `include`

There is also the `require` command: this does the same as `require_once`, except you could accidentally load the same file in more than once - which you almost never want to do.

There is also `include` and `include_once` which do the same as the `require` equivalents, except if it can't find the file it will keep running and just show an error message. However, normally if we're trying to include a file we wouldn't want the code to run at all if it can't find the file, so `require` is preferred.

2.2 Naming Collisions

Large PHP apps can have hundreds (or even thousands) of classes. It's not uncommon for two classes to end up with the same name. For example, in a blog app you might have a `Post` class which deals with the data for each post on the site. But you might also have a `Post` class which posts an update to Slack each time a post is added.

We could, of course, be careful about the naming of each class, calling one `BlogPost` and the other `SlackPost`, but in large apps it can be tricky keeping track of every class name that you've used - and it becomes practically impossible when you have multiple developers working on the same app.

Even if we're really careful naming our classes, we don't have any control over the names of classes in PHP libraries that other people have written. It would be unrealistic to make sure that none of the class names you've used clash with those in any libraries that you might use.

The Bad Old Days

I lied just then. Back in the before-times, when PHP was still trying to find itself, you *did* have to use unique names for every single class - including the ones in libraries (which you had no control over). To get around this issue you would pick an almost definitely unique prefix (like your company name) and add it to the front of every single class in the app: `SmallHadronCollider_BlogPost`, `SmallHadronCollider_SlackPost`. Needless to say, this made the code where the classes were used very messy.

2.3 Namespaces

Namespaces were added to PHP 5.3 to avoid this problem. The most everyday use of namespaces is the file system on your computer: you can have two files called exactly the same thing *as long as they're in separate directories.*

Namespacing in PHP is much the same idea. We assign each class to a namespace and then we can have two classes with the same name, *as long as they're in separate namespaces*. This means that when we use the class we need to tell PHP which namespace we are talking about.

We assign a namespace by adding a `namespace` declaration at the top of the file:

```
namespace Blog\Data;

class Post { ... }
```

Now, when we want to use this class we'll need to use the namespace:

```
new Blog\Data\Post();
```

This might not seem any better than the old way of doing things (i.e. using `BlogPost`), but PHP also gives us the `use` keyword.

We can put a `use` statement at the top of a PHP file to tell it to always use a specific namespaced class:

```
use Blog\Data\Post;

// further down the file
new Post(); // actually new Blog\Data\Post()

// we can use the other namespaced Post class
// we just need to use the full namespace
new Services\Slack\Post();
```

Generally we'll use the same class multiple times inside a file, so this saves a lot of typing.

If the class you want to use is in the *same* namespace as the current class you don't even need a `use` statement.

You can **alias** a class to give it a different name in the file you're working in. This can be particularly if you have two classes which share the same class name but are in different namespaces:

```
use Blog\Data\Post;
use Services\Slack\Post as SlackPost;

new Post(); // actually new Blog\Data\Post()
new SlackPost(); // actually new Services\Slack\Post()
```

2.4 Autoloading

When I said earlier that PHP apps can contain thousands of classes you might have thought "Well that's going to be an awful lot of `require_once` statements". And, in fact, historically that's exactly what you'd have: a file called something like `load.php` which listed thousands of files. Every time you wanted to add a class you'd need to write it and then make sure you added it to the massive list.

Thankfully, things have moved on since then and PHP supports **autoloading**. This lets us tell PHP where to find a specific class based on its name and namespace. However, writing this code ourselves is unnecessary because we'd be much better using the Composer package manager to do it for us.

2.5 Additional Resources

- PHP Namespaces Explained
- Auto-Loading Classes

Chapter 3

Composer

Composer is PHP's **package manager** (like `npm` is for JavaScript). It lets us easily add code written by other people to our projects.

One of Composer's responsibilities is to set up autoloading of any libraries that it adds, that way you don't have to manually link to all the files that you use in your project. It's also very easy to setup Composer so it will autoload any classes that *you've* created.

3.1 Initialising

First we need to add Composer. Run the following in the project directory that you want to add Composer to:

```
# the -n bit stops it asking you a bunch of questions
composer init -n
```

This will add a `composer.json` file to your project.

3.2 PSR-4 Autoloading

Next we need to tell Composer to load our classes for us. We're going to use the PSR-4 namespace standard. Basically, this means that we pick a directory to be the "root" of our namespace, and everything from that point on is just based on directory names.

First, create a directory called `app`. Then edit the `composer.json` so that it looks like this:

```
{
    "autoload": {
       "psr-4": {"App\\": "app/"}
    },
    "require": {}
}
```

Here we've told Composer that any namespace starting with the root `App` should look for files in the `app` directory. We could use anything as the root namespace or directory name.

Next, run `composer dump-autoload`: this, somewhat confusingly, generates an autoload file for us[1] in a directory called `vendor` (this is where Composer installs any libraries we might want to use).

We'll also need to create a file in the root of our project that looks like this:

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

// ... code that uses the classes
```

Now, as long as we stick to the following rules, we won't need to require any other files manually:

1. One class per file, where the file name is the same as the class name (case sensitive)
2. Put all of our classes in the `App` root namespace
3. If we add directories inside the `app` directory (for extra organisation), they add an extra level to the namespace

Because namespaces and classes in PHP are usually capitalised and, with PSR-4, the directory and filenames match the namespace/class naming, all the files and directories inside `app` will also be capitalised.

[1]Technically, it gets rid of the existing autoload file and then creates a new one - hence the name

For example, if we had a class called `Post` that just sat inside the `app` directory, it should be in a file called `Post.php` and have the namespace `App\Post`. If we had a class `Post` that did something with Slack we could create a directory `app/Slack` and then put the file `Post.php` in it with the namespace `App\Slack\Post`.

3.3 Libraries

As well as handling autoloading for us, Composer's main purpose is to let us use bits of code other people have written. Let's take a look at some useful libraries.

You can search for libraries on Packagist or Libraries.io.

> **Composer & Git**
>
> You don't want to add the `vendor` directory to your Git repositories, as it can be easily recreated by running `composer install`. So make sure you add `vendor/` to your `.gitignore` file.

3.3.1 `symfony/var-dumper`

We install this with `composer require symfony/var-dumper`. We then have access to the `dump()` function, which is a more useful version of `echo`:

```
dump([1, 2, 3, 4]);
/*
    array:4 [
      0 => 1
      1 => 2
      2 => 3
      3 => 4
    ]
*/

dump(new Person("Zazu"));
/*
    App\Person {
      -name: "Zazu"
    }
*/
```

It also adds the `dd()` function (for "debug and die"), which dumps the result and then immediately stops the PHP. This can be useful if you want to check something half-way through a process. Be careful, if you use `dd()` nothing after it will run.

See the VarDumper documentation for more information.

3.3.2 `Illuminate\Support\Collection`

This is part of Laravel, which we'll be covering later. It basically lets us handle arrays in a way which isn't utterly horrible[2].

We install it by running `composer require illuminate/support`. It's got tonnes of really useful methods, but we'll just look at four of them here: our old friends `map()`, `filter()`, and `reduce()`, as well as a very useful one called `pluck`.

Generally collection methods return a new collection object. You can turn a `Collection` back into a standard array by calling its `all()` method.

`filter`

Filter is almost identical to JavaScript: we pass it an anonymous function that takes each item in the array and returns a boolean value. It returns a new `Collection` containing all the items for which the function returned `true`:

```
use Illuminate\Support\Collection;

$numbers = new Collection([1, 2, 3, 4, 5]);

$even = $numbers->filter(function ($n) {
  return $n % 2 === 0;
});

dump($even->all()); // [2, 4]
```

[View code on GitHub]

[2] PHP's built-in array handling functions are utterly inconsistent "Because that's how PHP rolls: Like a square wheel".

`map`

Map is also very similar to JavaScript: we pass it an anonymous function that takes each item in the array and transforms the value somehow. It returns a new `Collection` where each item has been transformed:

```
// we can also use the collect function to create a new collection
$numbers = collect([1, 2, 3, 4, 5]);

$squared = $numbers->map(function ($n) {
  return $n * $n;
});

dump($squared->all()); // [1, 4, 9, 16, 25]
```

[View code on GitHub]

`reduce`

Again, reduce is very similar to JavaScript: we pass it an anonymous function that takes the accumulated value and each item in the array. The return value is passed in as the accumulator value for the next iteration. It returns the final accumulated value:

```
$numbers = collect([1, 2, 3, 4, 5]);

// remember, reduce takes two arguments
// the accumulator and each value in turn
// the initial value for $acc is null
// so make sure you set it
$sum = $numbers->reduce(function ($acc, $val) {
  return $acc + $val;
}, 0);

dump($sum); // 15
```

[View code on GitHub]

Make sure you pass in an initial value for the accumulator, otherwise it will be `null`, which might cause problems.

`pluck`

We've not come across `pluck` before, but it's very useful. It assumes your collection contains either associative arrays or objects all with the same structure. You pass it a key value and it extracts a new `Collection` contain just that key/property from each item in the collection:

```
$goodWatchin = collect([
  ["id" => 1, "name" => "Unbreakable Kimmy Schmidt"],
  ["id" => 2, "name" => "The Leftovers"],
  ["id" => 3, "name" => "Game of Thrones"],
]);

// [1, 2, 3]
dump($goodWatchin->pluck("id")->all());

// ["Unbreakable Kimmy Schmidt", "The Leftovers", "Game of Thrones"]
dump($goodWatchin->pluck("name")->all());
```

[View code on GitHub]

3.3.3 nesbot/carbon

The Carbon library makes working with dates in PHP much easier.[3] We install it by running `composer require nesbot/carbon`[4]. Once it's installed we have access to the `Carbon\Carbon` class.

```
use Carbon\Carbon;

// how old I am
dump(Carbon::createFromDate(1984, 4, 16)->age);
// the next summer olympics
dump((new Carbon("2016"))->addYears(4));
```

[View code on GitHub]

There are many more features listed in the Carbon documentation.

[3] I have yet to come across a programming language that has really good built-in date functionality
[4] If you've already installed `illuminate/support` this isn't necessary, as it's a dependency for that package

3.4 Additional Resources

- More about PSR-4
- Composer

Chapter 4

Object-Oriented Programming

So far, most of the code you've written has been "procedural": start at the top of a file, run through it, maybe call a few functions as you go, and then finish at the end. This is fine for simple programs or when we're just working inside an existing system (e.g. WordPress), but it doesn't really scale to larger applications.

The problem comes because we need to manage **state**: keeping track of all the values in our code. For a large app you could easily have thousands of values that need storing. Naming and keeping track of all these variables would become a nightmare if they were all in the same global scope.

Object-Oriented Programming[1] (OOP) is one way to make this easier. The key idea behind OOP is **encapsulation**: we keep functions and variables that are related to each other in one place (an object) and then use visibility to limit which bits of code can access and change them.

An object is effectively a black box: objects can send **messages** to each other (by calling methods), but they need not have any knowledge of the inner workings of other objects.

[1] "Oriented" not "Orientated".

The Unusual History of PHP

PHP has a long and complicated history. The first version of PHP wasn't even a programming language, it was just simple templating language that allowed you to re-use the same HTML code in multiple files.

> *I don't know how to stop it, there was never any intent to write a programming language … I have absolutely no idea how to write a programming language, I just kept adding the next logical step on the way.*
>
> - Rasmus Lerdorf, Creator of PHP

Over the years PHP morphed into a simple programming language and then into a modern object-oriented programming language. However, it wasn't really until 2009, with the release of PHP 5.3, that PHP could truly be considered a fully object-oriented language.

Because of this gradual change, older PHP frameworks and systems (such as WordPress) were originally written using non-OO code, which is why they still contain a large amount of procedural code.

PHP gets a lot of flack for not being a very good programming language and a few years ago that was perhaps a valid criticism. But in recent years, particularly with the release of PHP 7, it's just not true anymore. It certainly still has some issues, but nothing that a few libraries can't get around.

> *There are only two kinds of languages: the ones people complain about and the ones nobody uses*
>
> - Bjarne Stroustrup, Creator of C++

4.1 Encapsulation

Say that our app includes some code to send an email. If we were using procedural code we would probably have a function called `sendMail` that we can pass various values to:

```
// a sendMail function
function sendMail($to, $from, $message)
{
  // ... send email
}

// elsewhere
sendMail(
  "bob@bob.com",
  "hello@wombat.io",
  "Welcome to the best app for finding wombats near you"
);
```

[View code on GitHub]

But we might want to be able to customise more than just the to, from, and message parts of the email. Which means we'd either need to have a lot of optional arguments (which becomes unwieldy quickly) or rely on global variables:

```
// setup global variables
$to = null;
$from = null;
$message = null;
$subject = null;
$characterSet = "utf8";

function sendMail()
{
  // ... use the global variables
}


// elsewhere
$to = "bob@bob.com";
$from = "hello@wombat.io";
$message = "Welcome to the best app for finding wombats near you";
```

```
$subject = "A Wombat Welcome";

sendMail();
```

[View code on GitHub]

But this is truly horrible: we have no way of preventing other parts of our code from changing these values and we would start having to use long variable names to avoid ambiguity in bigger apps.

So, we want to store the variables and the functionality together in one place and in such a way that values can't be accidentally changed. This is where objects come in:

```
class Mail
{
  private $to;
  private $from;
  private $characterSet = "utf8";

  public function to($address)
  {
    $this->to = $address;
    return $this;
  }

  public function from($address)
  {
    $this->from = $address;
    return $this;
  }

  public function send($subject, $message)
  {
    // ... code to send mail
    // we can use $this to access the values
  }
}


// elsewhere
$mail = new Mail();
$mail->to("bob@bob.com")
```

```
    ->from("hello@wombat.io")
    ->send(
      "A Wombat Welcome",
      "Welcome to the best app for finding wombats near you"
    );
```

[View code on GitHub]

Now if we need to add additional fields, we can just add a property and setter method

4.2 (Almost) Pure OO

Many object oriented languages *only* use objects. For example in Java everything lives inside a class and you specify which class your app should create first when you run it.

Because of PHP's history we always need a little bit of procedural code to get our objects up and running. This is often called the **bootstrap** file.

```
class App
{
  public function start()
  {
    // code that makes the app do stuff
  }
}

// the bootstrap code
$app = new App();
$app->start();
```

[View code on GitHub]

Once we've created our first object the idea of object-oriented programming is that we use objects from that point onwards.

4.3 Object Types & Type Declarations

We looked at "types" with JavaScript: numbers, strings, booleans, arrays, &c. PHP has a similar set of built-in types. In OOP we generally talk of objects of having

the type of their class: e.g. a `Person` object instance is of the type `Person`.

PHP supports **type declarations**.[2] These let us say that the arguments passed to a function or method must be of a certain type. For example, say we had a `MailingList` class with a `sendWith()` method. It would be useful to say that we can only pass `Mail` objects into this method[3]:

```
class MailingList
{
  private $emails = [];
  private $subject;
  private $message;
  private $from;

  // mailing list code
  // to set from, subject, and message
  // and to add email addresses

  // use the Mail type-declaration to
  // only allow Mail classes
  public function sendWith(Mail $mailer)
  {
    // setup the from address
    $mailer->from($this->from);

    // for each email send with the passed in $mailer
    foreach ($this->emails as $email) {
      $mailer->to($email)->send($this->subject, $this->message);
    }
  }
}
```

[View code on GitHub]

Before accepting the `$mailer` parameter, we add the type declaration/hint of `Mail`. Now, if the user of that class tries to pass in something that isn't an instance of `Mail`, PHP will throw an error.

```
// create a new mailing list
$mailinglist = new MailingList();
```

[2] In previous versions of PHP these were called type hints
[3] This is an example of **dependency injection**.

```
// ... code to set subject, from, add email addresses

// create a new mail object and pass in
$mail = new Mail();
$mailinglist->sendWith($mail);

// try to send with a Person
// won't work!
$mae = new Person("Mae", "Zimmerman");
$mailinglist->sendWith($mae); // TypeError!
```

[View code on GitHub]

4.4 Polymorphism

You might look at the above example and think, "What's the point of passing in the `Mail` object?" And you'd be right. If we can only pass in a `Mail` object, then we may as well just create it in the `MailingList` class. What if we wanted to be able to sometimes send our mailing list emails using the server's built-in mail program and sometimes using MailChimp? We can't type-hint two different classes, but if we don't type-hint at all then you could accidentally pass in something that will break your code completely.

This is where the idea of **polymorphism** comes in. Polymorphism is when two *different* types of object share enough in common that they can take each other's place in a specific context.

There are two ways to achieve this in most OO languages:

- **Inheritance**: when an object can inherit methods/properties from another object, creating a hierarchy of object types.
- **Interfaces**: when an object implements a defined set of methods.

4.5 Inheritance

Inheritance lets us create a hierarchy of object types, where the **children** types inherit all of the methods and properties of the **parent** classes. This allows reuse of methods and properties but allowing for different behaviours.

For example, say that we want to create a `Mail` and a `MailChimp` class. Both of these will be responsible for sending an email, so they will have some things in common, but their inner workings will be different. We could create a parent `Mailer` class:

```
class Mailer
{
  protected $to;
  protected $from;
  protected $characterSet = "utf8";

  public function to($address)
  {
    $this->to = $address;
    return $this;
  }

  public function from($address)
  {
    $this->from = $address;
    return $this;
  }
}
```

[View code on GitHub]

We move all of the shared code into the `Mailer` class. We don't put the `send()` method into the `Mailer` class, as it will be different for each implementation. We can then **extend** the `Mailer` class to copy its behaviour into `Mail` and `MailChimp`:

```
// extends Mailer
// so it has the same properties and methods
class Mail extends Mailer
{
  public function send($subject, $message)
  {
    // send using local mail server
  }
}

// extends Mailer
// so it has the same properties and methods
class MailChimp extends Mailer
```

```
{
  public function send($subject, $message)
  {
    // send using mail chimp
  }
}
```

[View code on GitHub]

Now, in the `MailingList` class we can use `Mailer` as the type declaration. That means that depending on our mood,[4] we can send messages with either the local mail server or with MailChimp:

```
class MailingList
{
  // ...etc.

  // use the Mailer type-declaration to
  // allows all children of Mailer
  public function sendWith(Mailer $mailer)
  {
    // setup the from address
    $mailer->from($this->from);

    // for each email send with the passed in $mailer
    foreach ($this->emails as $email) {
      $mailer->to($email)->send($this->subject, $this->message);
    }
  }
}

// elsewhere
$mailinglist = new MailingList();
$mailinglist->sendWith(new Mail());

// or maybe
$mailinglist->sendWith(new MailChimp());
```

[View code on GitHub]

[4] Or possibly something more concrete

4.5.1 Abstract Classes

But you can perhaps see a few issues with this. Firstly, you could create an instance of `Mailer` and pass that into `sendWith()`. This would cause an issue because the `Mailer` class doesn't have a `send()` method, so you'd get an error. Secondly, there's no guarantee that a child of `Mailer` has a `send()` method: we could easily create a child of `Mailer` but forget to add it. This would lead to the same issue:

```
$mailinglist = new MailingList();

// oops, won't work
// Mailer doesn't have a send method
// so we'll get an error when sendWith()
// tries to call it
$mailinglist->sendWith(new Mailer());
```

[View code on GitHub]

This is where **abstract classes** come in. These are classes that are not meant to be used to create object instances directly, but instead are designed to be the parent of other classes. We can setup properties and methods in them, but they can't actually be constructed, only extended.

We can also create `abstract` method signatures. These allow us to say that a child class *has* to implement a method with the given name and parameters. We'll get an error if we don't.

This gets rounds both issues: we won't be able to instance `Mailer` and we can make sure any of its children have a `send()` method:

```
// make the class abstract
// it cannot be instanced now, only extended
abstract class Mailer
{
  protected $to;
  protected $from;
  protected $characterSet = "utf8";

  public function to($address)
  {
    $this->to = $address;
    return $this;
```

```
  }

  public function from($address)
  {
    $this->from = $address;
    return $this;
  }

  // create an abstract method signature
  // all children of this class must implement the send method
  abstract public function send($subject, $message);
}
```

[View code on GitHub]

4.5.2 Overriding

Children classes can **override** methods and properties from parent classes. That means if we wanted to make it so that the `to()` method in the `MailChimp` class did something slightly different, then we could write a different `to()` method. As long as it has the same method signature (i.e. excepts the same parameters), this will work.

If necessary it's possible to stop a child class from overriding a method by adding the `final` keyword in front of it:

```
abstract class Mailer
{
  // ...etc.

  // not sure why you'd need to do this
  // but now a child class couldn't
  // change this method
  final public function to($address)
  {
    $this->to = $address;
    return $this;
  }
}
```

[View code on GitHub]

4.5.3 The Intuitive Appeal of Inheritance

You can probably see that there's an intuitive sort of appeal to inheritance. Programming books often give examples like creating an `Animal` class, which then has a child `Mammal` class, which then has a child `Primate` class, which has a child `Person` class, &c.

It also looks like we can save ourselves a lot of code by adding as many of the properties and methods as we can into the classes higher up the hierarchy: we only need to write the `to()` and `from()` methods *once* in the `Mailer` class, which save a lot of repetition.

4.6 Interfaces

Interfaces are the other way that we can take advantage of polymorphism. An **interface** is a list of **method signatures** that a class can say it conforms to. It is a contract: if a class implements an interface then we are guaranteed that it has a certain set of methods taking a specified set of arguments.

For example, rather than creating a `Mailer` abstract class, we could instead create an interface:

```
interface MailerInterface
{
  public function to($address);
  public function from($address);
  public function send($subject, $message);
}
```

[View code on GitHub]

You can see that we list out all of the methods that a class that implements this interface *must* implement. We would use it as follows:

```
// this time we use implements
// extend a class, implement an interface
class Mail implements MailerInterface
{
  private $to;
  private $from;
```

```
  private $characterSet = "utf8";

  public function to($address)
  {
    $this->to = $address;
    return $this;
  }

  public function from($address)
  {
    $this->from = $address;
    return $this;
  }

  public function send($subject, $message)
  {
    // send using local mail server
  }
}
```

[View code on GitHub]

We would use it in `MailingList` in exactly the same way:

```
class MailingList
{
  // ...etc.

  // use the MailerInteface type-declaration to
  // allows all classes that implement MailerInterface
  public function sendWith(MailerInterface $mailer)
  {
    // setup the from address
    $mailer->from($this->from);

    // for each email send with the passed in $mailer
    foreach ($this->emails as $email) {
      $mailer->to($email)->send($this->subject, $this->message);
    }
  }
}
```

[View code on GitHub]

Now we could only pass in classes that implement the `MailerInterface` interface.

This might at first not seem like much of an advantage over using an abstract class, but a class can only `extend` a single class, whereas it can `implement` as many interfaces as it likes:

```
class MailChimp extends
  MailerInterface,
  HttpInterface,
  EncryptionInterface,
  ThatOtherInterface
{
  // ...etc.
}
```

[View code on GitHub]

4.6.1 Message Passing

If you look at the `sendWith()` method you'll see that it only uses the `to()`, `from()`, and `send()` methods of the object that gets passed in. Therefore it's *guaranteed* that if the object passed in conforms to the `MailerInterface`, which defines all three of those methods, then it will work. That's not to say the implementation will necessarily work, but the `sendWith()` method has access to all the methods that it requires.

This is a core idea in OOP. But one that is often not talked about with beginner level books on OOP. Although it's called *Object*-Oriented Programming, it's not actually the objects that should get the focus, it's the **messages** that they can send to one another: the methods and their parameters.

The reason interfaces are such a powerful idea is because they focus solely on the messages and don't tell you anything about the implementation. This is important because of encapsulation. If we have to worry about how a specific object does something, then we can't treat it as a black box.

This is also why try to use only `private` properties: if a property is `public` then we need to know about how the internals of the class work.

Inheritance can break the idea of encapsulation because in order to use it you often need to know about the *internal* workings of the parent classes: Were these properties declared? Has another class in the hierarchy overriden anything? This makes it much harder to work with and it gets worse as the codebase gets bigger. What if you need to change a parent class: will it break any of the children?

"But," I hear you saying, "you've got to write the same `to` and `from` methods and repeat all the properties in `Mail` and `MailChimp`! And repeating code is bad."

This is true with these very simple class examples, but you generally find that with more complex classes you don't actually repeat all that much code between them. If you do need to repeat code between classes there are often cleaner ways to do it than using inheritance: a shared class, for example.

4.7 Inheritance Tax

> *The object-oriented version of "Spaghetti code" is, of course, "Lasagna code": too many layers*
>
> - Roberto Waltman

If you read any books about OOP they'll focus a lot of their time on inheritance. While inheritance can be very useful, all of this attention means that it's often the technique that programmers reach for when they need to add a bit of functionality. And it will almost always lead to much more complicated code.

That's not to say it isn't useful. It's totally fine to inherit code that frameworks/libraries provide, as in these cases you're generally adding one tiny bit of functionality to something that's much more complicated under-the-hood.[5] But if it's code that you've written, it's always worthwhile thinking "Do I really need to use inheritance?"

Sandi Metz suggests not using inheritance until you have *at least* three classes that are *definitely* all using exactly the same methods. You should never start out by writing an abstract class: write the actual use-cases first and only write an abstract class if you definitely need one. If you do use inheritance then try not to create layers and layers of it: maybe have a rule that you'll only ever inherit through one layer.

[5] Although some purists would say even in this case there are better alternatives: see the Active Record vs Data Mapper debate

4.7.1 Composition

> *Prefer composition over inheritance*
>
> - The Gang of Four, *Design Patterns*

Think back to the `Animal`, `Mammal`, `Primate` &c. example earlier. It intuitively has some appeal, but if you think about it for a while it starts to fall down. For example, if your mammal class had any functionality to do with giving birth to live young, then the `Monotreme` class is going to have issues. It turns out that despite all having the same common ancestor at some point in history, the only thing that *all* mammals have in common is something to do with jaw articulation.[6]

The idea of *composition over inheritance* is that rather than sharing behaviour with inheritance, we share it using interfaces and shared classes. Rather than having a `giveBirthToLiveYoung()` method that all `Mammal` objects have to inherit, instead have a `GiveBirthToLiveYoung` interface, which is implemented only by those mammals that need to. If a lot of those classes share the same implementation of that method then consider moving it into a separate class that they can all share. It might require a bit more code to get working, but it is much easier to make changes to.

The Law of Demeter

The "Law of Demeter" is a guideline for OOP about how objects should use other objects. Expressed succinctly:

> *Each object should only talk to its friends; don't talk to strangers*

In practice, this means that an object should only call methods on either itself or objects that it has been given. You should avoid calling a method which returns an object and then calling a method on that object: it requires too much knowledge about other objects.

```
// this is fine
$this->doThing();
```

[6] "Synapsids that possess a dentary–squamosal jaw articulation and occlusion between upper and lower molars with a transverse component to the movement", Kemp, T. S. (2005) *The Origin and Evolution of Mammals*

```php
// this is fine
$this->mailer->send("Hello");

// this is not good
// the object needs to know how libraries
// and how books work
// this is *not* chaining
// get back a different object each call
$this->library->firstBook()->author();

// even worse
// using properties directly
// should stick to method calling
$this->library->books[0]->author;
```

4.8 Additional Resources

- PHP Apprentice: Inheritance
- PHP Apprentice: Interfaces
- Wikipedia: Composition over Inheritance
- Wikipedia: The Law of Demeter
- Practical Object-Oriented Design in Ruby: An intermediate book about OOP. In Ruby, but all the key concepts work in PHP too.
- Wikipedia: The SOLID Principles of OOP: Quite advanced, but incredibly useful if you can get your head round it.

Chapter 5

HTTP

5.1 How the Web Works

HyperText Transfer Protocol is what the World Wide Web is built upon.

Fundamentally it's just sending some text[1], with a **well defined format** (or **standard**), that the browser knows how to interpret into a web page.

By default, a server will listen on **port 80** for anything that looks like HTTP. If it receives a valid HTTP request then it will return a HTTP response.

In fact, we can manually type in HTTP requests:

```
# make a direct connection to the CERN server on port 80
telnet info.cern.ch 80
```

Type in the following exactly and press return twice:

```
GET /hypertext/WWW/TheProject.html HTTP/1.1
Host: info.cern.ch
```

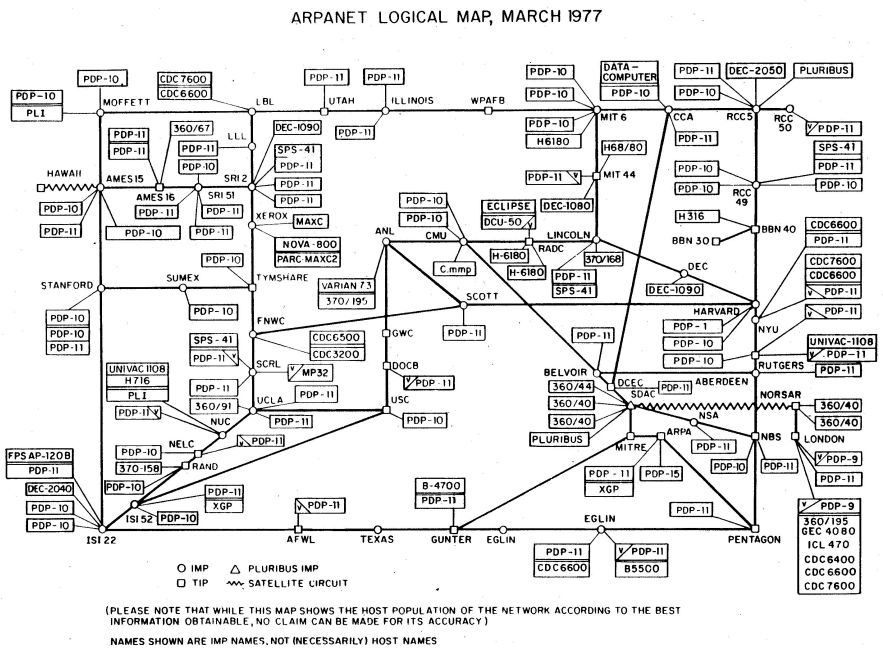
You should get back a valid HTTP response.

[1] This is no longer true with HTTP 2.0, which uses a binary format.

The Internet vs. The Web

The internet and the world wide web are, historically, not the same thing.

The internet started off as ARPANet in the late 60s. It was a way of connecting together government computers and universities in a way where it wasn't necessary that every "node" in the network needed to be connected directly.



The internet was so small in 1977 that you could map every computer connected to it on a piece of A4 (source)

Things like e-mail, IRC, newsgroups, and bulletin boards all existed before the web.

The "World Wide Web" was invented in 1989 by Sir Tim Berners Lee, who was working at CERN at the time. He wanted to create a way so that you could click on a reference in a research paper an be taken directly to the referenced paper - what we'd now call a link. He didn't expect this to have much use outside of academia.

5.2 Requests

An HTTP request, at its most minimal, will generally consist of the following:

```
GET /api/articles HTTP/1.1
Host: restful.training
Accept: application/json
Accept-Encoding: gzip, deflate
```

1. `GET`: the HTTP method
2. `/api/articles`: the URI fragment
3. `HTTP/1.1`: the HTTP version
4. `Host:`: the server hostname header
5. `Accept:`: the accept header - what format to return
6. `Accept-Encoding:`: the accept encoding header - can we handle gzipped files?

We can see all of this and more for every single request in the Network panel of Chrome Developer Tools.

5.2.1 Methods

The HTTP standard lists a number of **methods** that can be used:

- `OPTIONS`
- `GET`
- `HEAD`
- `POST`
- `PUT`
- `DELETE`
- `TRACE`
- `CONNECT`

Browsers typically only use the `GET` and `POST` methods: when you view a webpage you are making a `GET` request; when you submit a form you are making a `POST` request.

The other methods largely went unused until RESTful APIs came along.

5.3 Responses

A typical response:[2]

```
HTTP/1.1 200 OK
Content-Encoding: gzip
Content-Length: 183
Content-Type: application/json; charset=UTF-8
Connection: Closed

[
    {
        "id": 1,
        "name": "Alice",
        "won": 0,
        "lost": 0,
        "created_at": "2017-07-04 16:03:11",
        "updated_at": "2017-07-04 16:03:11"
    }
]
```

- `HTTP/1.1`: the HTTP version
- `200 OK`: the HTTP status code and message
- `Content-Encoding:`: the content encoding header
- `Content-Length:`: the content length header
- `Content-Type:`: the content type/character encoding header
- `Connection:`: the connection header

Then two line breaks, followed by the body (HTML or JSON).

[2] You can use `curl -i http://example.com` to see the response for any page.

5.3.1 Status Codes

The HTTP spec also defines a number of "status codes" which have specific meanings:

- `2xx`: Success
  - `200`: OK
  - `201`: Created
  - `204`: No Content
- `3xx`: Redirect
- `4xx`: Client Error
  - `400`: Bad Request
  - `401`: Unauthorised
  - `403`: Forbidden
  - `404`: Not Found
  - `405`: Method Not Allowed
  - `422`: Unprocessable Entity
  - `429`: Too Many Requests
  - `418`: I'm a Teapot[3]
- `5xx`: Server Error

5.4 Additional Resources

- The World Wide Web and Its Inventor
- Charles Proxy: view all the HTTP requests on your computer
- What is Internet Anyway? An Internaut's Guide

[3] As per RFC 2324 Hyper Text Coffee Pot Control Protocol: Any attempt to brew coffee with a teapot should result in the error code "418 I'm a teapot". The resulting entity body MAY be short and stout.

Chapter 6

RESTful APIs

Application Programming Interfaces allow programs to talk to one another[1].

But, programs are stupid: they can only deal with things that they're expecting.

So an API has to be **well-defined**: how we make requests to it and what it gives us back should be *predictable* (and preferably documented).

6.1 RESTful

We can design our API however we like, but if we stick to certain conventions it will be much easier to use the API from other programs.

The most popular way to design an API nowadays[2] is using **RESTful**[3]

The idea of RESTful is that we send our data as JSON using HTTP and use the already established HTTP methods/status codes.

Because most programming languages have built in functionality for working with HTTP and JSON, using RESTful APIs means that a program written in one language can talk to a program written in another language without any problems.

[1] "API" is sometimes also used to refer to the methods on objects, e.g. the `map` method of an array or the `.getElementById()` method of DOM nodes
[2] It's looking like GraphQL might become the new standard in a few years
[3] **RE**presentational **S**tate **T**ransfer - don't worry if you can't remember that, I just looked it up

Key Ideas

- Sent using HTTP
  - HTTP headers
  - HTTP methods
  - HTTP status codes
- Use URLs for different resources
- Use JSON to send/receive data

6.2 JSON

JSON is a way of storing and transferring data structures. It is used by most APIs on the internet. And it's based on JavaScript (it stands for **JavaScript Object Notation**) - so if you understand JS objects/arrays, you can understand JSON.

```
{
    "people": [{
        "name": "Alice",
        "age": 52,
        "address": {
            "address1": "221b Baker Street",
            "city": "London"
        }
    }, {
        "name": "Bob",
        "age": 34,
        "address": {
            "address1": "13 Morse Street",
            "city": "Exeter"
        }
    }]
}
```

JSON **is not** JavaScript: JavaScript is a programming language that gets interpreted into machine code and runs, JSON is a data format for representing data so that it can be stored/transferred.

JSON has to be understood by programming languages other than JavaScript which means it is stricter: you can't have stray commas and property keys *have* to be wrapped with **double quotes**.

6.3 Methods

With RESTful the HTTP methods are given specific meanings:

Method	Meaning
`GET`	displaying data, shouldn't change anything (e.g. the database)
`POST`	creating new things
`PUT`	updating things
`PATCH`[4]	updating parts of things
`DELETE`	deleting things

6.4 Status Codes

Some status codes are also given specific meanings:

Status Code	Name	Meaning
`201`	Created	A new item has been created
`422`	Unprocessable Entity	Validation error

6.5 Naming Routes

By convention we use something like the following for naming our routes:

Method	Example URL	Meaning
`GET`	`/articles`	Show all articles
`POST`	`/articles`	Create new article
`GET`	`/articles/1`	Show article with ID 1
`PUT`	`/articles/1`	Update article with ID 1
`DELETE`	`/articles/1`	Delete article with ID 1
`GET`	`/articles/1/comments`	Show comments for article with ID 1

Notice that we use the pluralised version of names in the URL.

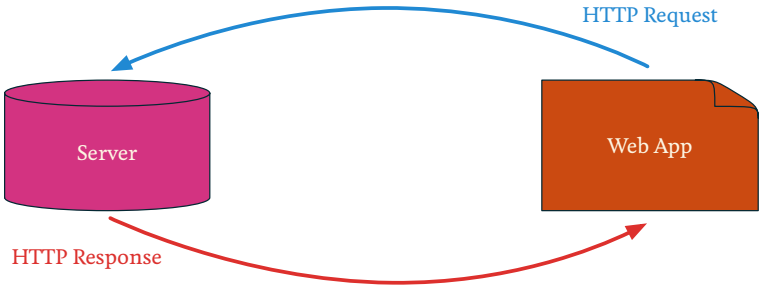
[4] Added to the spec in 2010

6.6 RESTful Architecture

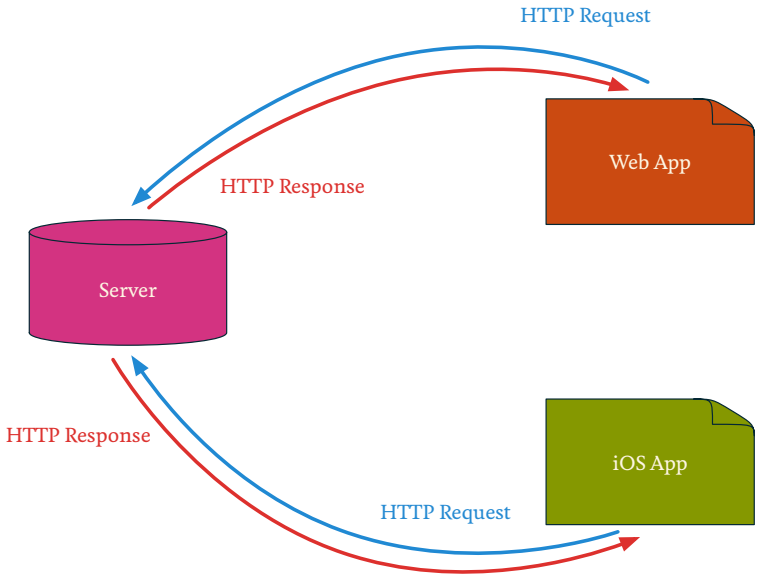
With a traditional website everything comes from the same server: the database, backend code, and frontend code.

An **API driven** web app splits the database and backend from the frontend. The backend code can be on a server in one bit of the world and the frontend code can be running on another server or an app.

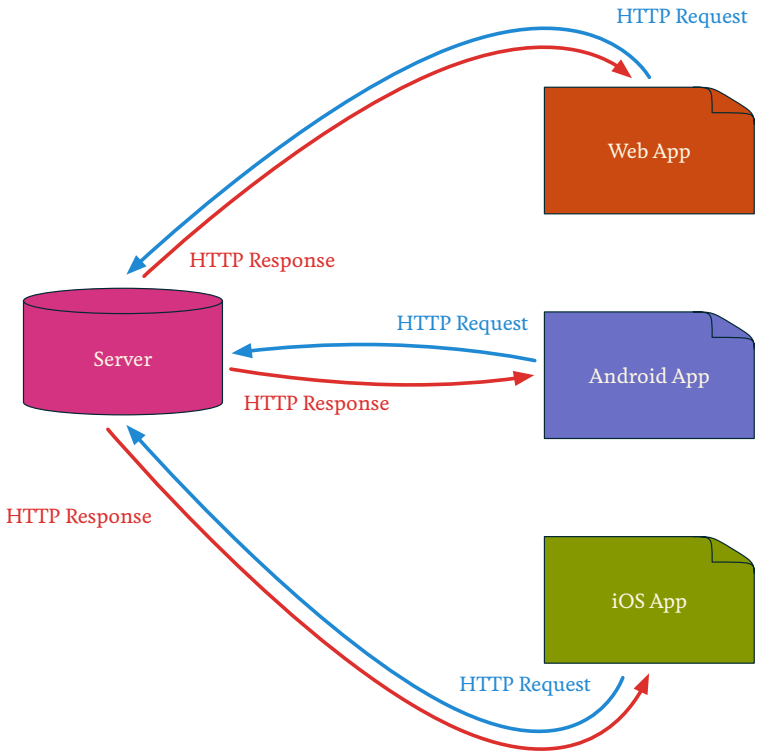
This means that you only have to write the main bit of your app once in the backend: the various frontends are just UIs using API requests to do everything.



A web app talks to the API using HTTP requests



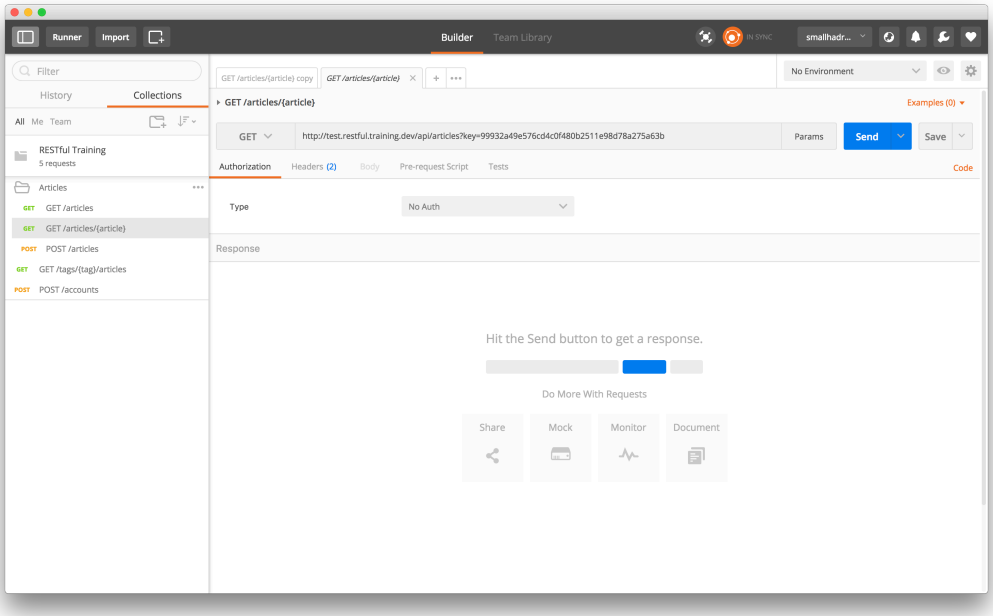
A completely separate iOS app can use the same API



And an Android app can use the same API too

6.7 Postman

Postman is a free app designed for testing out RESTful APIs[5]



Postman

APIs are not designed for people, so when you're testing them with Postman you need to get used to writing/reading JSON. Remember, it's just a slightly stricter JS object/array syntax.

Key things to remember:

- Make sure you set the `Accept: application/json` header
- Make sure you use "raw" and then "JSON (application/json)" for the Body
- Use Collections to save routes you're likely to use more than once

6.8 Additional Resources

- Any API: documentation and interactive consoles for tonnes of public APIs

[5] For Mac there is an alternative app called Paw, which is much better, but not free.

Chapter 7

Laravel: An Introduction

7.1 What is Laravel?

Laravel is a *modern* PHP framework designed for building database-driven websites and APIs. The "modern" bit is important, as many popular PHP frameworks have codebases that go back to the pre-2010 PHP days, when everything was horrible.

Laravel was created by a chap called Taylor Otwell, who was previously a .NET developer. .NET had a lot of nice ideas, but at the time you had to pay a lot of money to use any of it. So he decided to create a PHP framework based on some of the better ideas.

One of the core ideas behind Laravel is that it should be built on pre-existing libraries. Rather than having to write *all* of the code from scratch, Laravel just joins together the best libraries and provides a wrapper for them so that everything is consistent. That means the more complicated bits of code (like file systems management or routing) are written by specialists in those areas.

The other key idea behind Laravel is that it should make building websites and APIs as frictionless as possible. If there's a common thing that lots of sites need to do, then Laravel will have a quick and easy way to do it. This means that, once you get used to it, you can get an API up and running in a few minutes.

Here are some of the key features of Laravel:

- **Homestead**: a Vagrant configuration specifically designed for Laravel development
- **Eloquent ORM**: an incredibly simple way to work with databases

- `artisan`: a command line tool for doing all of the most common Laravel tasks
- **Database migrations**: a quick and easy way to create and update database structures
- **Scheduling** and **Job Queues**: easily setup tasks to run at a later point
- Very good documentation: well written documentation is really important when using a framework
- Laracasts: hundreds of hours of videos and an active community of developers

Why Not Node?

It's become very trendy to use Node to make APIs.

However, there is no established framework (or combination of libraries) in the Node ecosystem that provide anything close to what Laravel offers. This means that two seemingly similar Node APIs might be written completely differently. There are also certain essential bits of tooling, like database migrations, that are not well supported. In fact, Node is in a very similar sort of position to where PHP was pre-Laravel: lots of options, but none of them quite offering everything that you need. Give it a few more years and this will almost certainly change.

So why is Node so popular for building APIs? Companies think "We've got front-end developers who can already write JavaScript, let's use them to write the back-end too!". This is wrong-headed for two reasons: firstly, back-end code has very little in similar with front-end code. Secondly, if you really have to do this, you'd be much better getting developers to learn a new programming language, like PHP, Ruby, or C#, and using frameworks like Laravel, Ruby on Rails, or .NET. It turns out most the energy comes with learning the framework rather than the programming language, so you'd be better using a framework that does most the work for you.

That's not to say Node doesn't have its uses. It's great for doing stuff with web-sockets and streaming data (trying to use PHP for those is horrible). And if you enjoy functional programming then Node lets you write largely functional code (again not something PHP is good for). Personally, if I'm going to be writing object-oriented code anyway, I'd always go for Laravel.

7.2 Setup

Laravel provides an installer app that lets you create the scaffolding for a Laravel app. You'll need to install it using the following command:

```
composer global require laravel/installer
```

This adds the Laravel installer package to your computer so that you can run it from any directory. You only need to set this up on your computer once.

7.3 Creating a Project

To create a new Laravel project run:

```
laravel new project-name
```

Obviously you should call your project something more descriptive (and make sure it's a directory-friendly name - no spaces, all lowercase).

7.4 Running Laravel

There are various ways to get Laravel up and running. If you're on a Mac you can use Laravel Valet. But we'll be using **Homestead**, Laravel's pre-configured Vagrant box.

There are two ways to setup Homestead: globally and locally. The global way is more efficient in terms of hard drive usage, but it's also a bit of a pain to get setup properly. For this reason I recommend going with the local setup.

First, we need to add the Homestead files to our project. In the newly created project directory run:

```
composer require laravel/homestead --dev
```

This should add a couple of new Composer packages. If it looks like it's installed lots of packages then make sure you're definitely in the project directory.

Composer only adds files to the `vendor` directory, which Vagrant doesn't know anything about. So next, we need to copy the relevant Homestead files into the project directory:

```
vendor/bin/homestead make
```

This should have added a `Homestead.yaml` and `Vagrantfile` to your project.

> **`Homestead.yaml`**
>
> The `Homestead.yaml` file that is generate is specific to the directory that you run it in. If you move the project to a different directory you'll want to run the `homestead make` command again or update the folders mapping in the `Homestead.yaml` file.

By default Homestead is setup to use 2GB of RAM. For our purposes this is overkill (and will make computer with less than 8GB of RAM feel really sluggish). So, change the second line of `Homestead.yaml` so it just uses 512MB:

```
memory: 512
```

Now you can get Vagrant up and running:

```
vagrant up
```

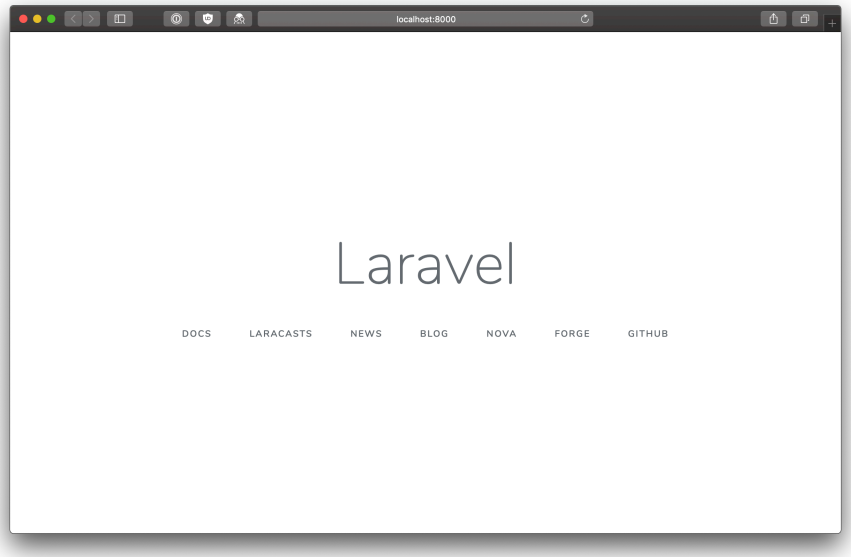
This can take a while the first time as it will need to download the appropriate Homestead box[1], then get the VM up and running, and then **provision** (in this case, run the Laravel setup scripts) the VM.

[1] This can be *really* slow. Unfortunately the Homestead box is hosted on a really under-powered server. If you get a download time of several hours it can sometimes be worth cancelling and retrying a few minutes later.

7.5 Getting Started

Once Vagrant has finished loading:

- On Mac: http://homestead.test
- On Windows: http://localhost:8000



Laravel up and running

500 Server Errors on First Run

Occasionally provisioning doesn't quite work as planned. This often happens if you moved the project directory and forgot to update `Homestead.yaml`. You can re-run the provisioning scripts with `vagrant provision`. However, when you run them the second time it can skip a few stages. So, if you get a 500 error on first run, run the following in the project directory:

```
vagrant provision # re-run provisioning scripts
php artisan key:generate # generate a new app key
```

7.6 Additional Resources

- Laravel Installation
- Per-project Homestead Setup

Chapter 8

Eloquent ORM

If we're going to create a database driven API, we'll need to interact with a database. There are two key parts to this:

- Setting up and updating the structure of the database
- Reading/writing data from the database

Laravel makes it very easy to do both of these using the `artisan` command.

`artisan`

We'll be using `artisan` *a lot* as it will save us writing all the boilerplate code.

It's important to run all of your `artisan` commands inside the Vagrant box. Some of the commands work if you run them on your own machine, but not anything to do with databases. To avoid getting any errors *always* run `artisan` inside Vagrant.

```
vagrant ssh # login box  - password is vagrant if needed
cd code # Homestead puts everything inside the code directory
artisan <artisan-command> # artisan command you wish to run
```

It's probably easiest to have a terminal tab/window open that you just keep logged into the Vagrant box at all times.

If you run just `artisan` it will list all of the commands it supports.

8.1 Database Migrations

The Homestead box creates a database called `homestead` which Laravel has been setup to use by the provisioning script.[1] The `homestead` database is empty by default, so we'll need to add the structure that we need for our API.

Realistically, you're never going to know the complete structure of your database when you start building an API. Specifications will change over time and features will be added and removed. The structure you end up with at the end might be very different from what you started with.

For this reason, we need a way to keep the structure of the database up-to-date with the rest of the app: it's no good writing code that tries to access a non-existent table or column.

Keeping the database structure up-to-date becomes particularly tricky if there are multiple developers working on the same codebase. In the bad ole days developers would share a "dump" of the database: i.e. a complete copy of all of the tables, columns, and data. But this is bad for a couple of reasons:

- Say you have two developers working on the API. One of them, Asha, is working on stuff to do with money and the various related tables. The other developer, Jiro, is working on stuff to do with users and the tables related to that. If Asha does a dump of her database and passes it over to Jiro, then it's not going to have all the user tables that Jiro has created, so he'd need to recreate them manually.
- Sharing the structure of the database is desirable, but we probably don't want to share the test data that other developers have been putting in. The things that people type in as test data are probably best kept to themselves. Anyone looking at a database I've populated manually would think I was obsessed with monkeys, fish, and spoons.[2]

This is where **database migrations** come in. A database migration is a file that contains instructions on how to update a database: for example, create a new table called `spoons` with `id INT`, `type VARCHAR(50)`, and `runcible TINYINT` columns. This file can then be run in order to update the structure of the database. A migration only needs to run once (the table only needs creating a single time), so there also needs to be a way to keep track of which database migration files have run and which haven't.

[1] Take a look in the `.env` file if you want to see.
[2] And I really don't care for fish.

8.1.1 Creating a Migration

We're going to build a simple blog API which will allow us to create articles which have a title and an article body. So we want to create a table called `articles` with `id`, `title`, and `article` columns.

Make sure you're inside the Vagrant box and in the `code` directory and then run:

```
artisan make:model Article -m
```

This creates an Article model class[3] (`app/Article.php`) as well as a database migration (`database/migrations/<timestamp>_create_articles_table.php`). It is possible to create database migrations *without* creating a model (`artisan make:migration`), but we won't be needing that.

Look inside the database migration and you'll see the following:

```
<?php

use Illuminate\Support\Facades\Schema;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;

class CreateArticlesTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('articles', function (Blueprint $table) {
            $table->increments('id');
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
```

[3] We'll look at models shortly.

```
     *
     * @return void
     */
    public function down()
    {
        Schema::dropIfExists('articles');
    }
}
```

[View code on GitHub]

As you can see, a migration consists of an `up` method and a `down` method. The `up` method's job is to make the changes that we want to the database. The `down` method's job is to reverse the changes that `up` made: this allows us to "rollback" a migration if we made a mistake.

You can see that both methods are already partly written for us. The `up` method is creating an `articles` table with an `id` column and the `down` method will remove the `articles` table. In fact, we won't need to change the `down` method at all.

Timestamps

You'll notice that that `up` method also includes `$table->timestamps()`. This adds `created_at` and `updated_at` columns, which Laravel will automatically keep up-to-date for us.

These columns can be very useful when working with more complex data tables. Later in the week we'll look at a case when they are perhaps not necessary, but for now, it's best to keep them.

However, we do need to update the `up` method to include the other columns:

```
public function up()
{
    Schema::create('articles', function (Blueprint $table) {
        $table->increments('id');
        $table->string("title", 100);
        $table->text("article");
        $table->timestamps();
```

```
    });
}
```

We've added the `title` column as a "string" with maximum length of 100: in MySQL this is the same as `VARCHAR(100)`.[4] We've also added a `article` column with the "text" type - this is for storing arbitrarily long bits of text.

We've now created our database migration, but until we run it nothing will happen.[5] We run all migrations that haven't yet been run with:

```
artisan migrate
```

If you look at the database now you'll see that the `articles` table has been created.

If you made a mistake, you can run `artisan migrate:rollback` to undo the last set of migrations that you ran, make any changes, and then run `artisan migrate` again.

8.2 Models

Next, we want to read and write data from the database. Back in the day we'd have done this by writing MySQL queries as strings in PHP and then using `mysqli` to run them.

Having to write queries out as strings is pretty horrible at the best of times. But this would also limit our app to only being able to use MySQL, which would mean we couldn't easily switch it over to SQLite or PostgreSQL if we needed to.

That's where **Object Relational Mapper**s (ORM) come in. An ORM lets you work with plain old PHP objects, but under the hood it's actually writing and running database queries for you. Laravel's ORM library is called **Eloquent**.[6]

[4] It doesn't use `varchar` because Laravel supports lots of different SQL engines, not all of which use the same types as MySQL.
[5] It's important to note that database migrations are not part of the Laravel app: they do not run whenever the app loads, only when you run them with `artisan`.
[6] Good name.

8.2.1 Writing Data

We can use the `artisan tinker` command to run arbitrary bits of PHP code in our app environment. Let's create a new article:

```
$article = new Article();
$article->title = "My amazing blog post";
$article->article = "Today I went shopping. I bought some spoons.";
$article->save();
```

[View code on GitHub]

Have a look in the database now and you'll see that a new row has been added with the data that we added to the `$article` properties.

The `Article` class was created when we ran `artisan make:model` earlier. If you take a look at the class it doesn't seem to do very much:

```
namespace App;

use Illuminate\Database\Eloquent\Model;

class Article extends Model
{
    //
}
```

[View code on GitHub]

All of the work here is being done by Eloquent's `Model` class, which `Article` is extending. The `Model` class does all of the databasey stuff for us. We just create a new `Article` class, set some properties, and then call the `save()` method. We don't need to worry about writing queries, setting up IDs, or updating the timestamps - Eloquent does it all for us.

Once you've called the `save()` method, you can take a look at the `id` property:

```
$article->id; // 1 - assuming this is the first thing you've added
$article->created_at; // timestamp from just now
$article->updated_at; // timestamp from just now
```

[View code on GitHub]

The `id` property now has a value because the article has been stored in the database. The `created_at` and `updated_at` properties also have values.

If you make another change to the article and then save it again you'll see that the `updated_at` property has been updated:

```
$article->title = "Spoonarama!";
$article->save();

$article->id; // same as before
$article->created_at; // same as before
$article->updated_at; // updated timestamp from just now
```

[View code on GitHub]

8.2.2 Reading Data

We can also find articles using the `Article` class. It inherits various `static` methods from `Model`. For example, you can run `Article::all()` to get a `Collection` back with all the articles in it, and you can run `Article::find(1)` to find the article with the `id` 1.

```
// all the articles
Article::all();

// the article with id 1
Article::find(1);
```

You can also build up more complex `WHERE`-style queries, all just with Eloquent methods. See the Eloquent docs for more information.

8.3 Additional Resources

- Database Migrations
- Eloquent

Chapter 9

Basic API

Now we've got somewhere to store our data and a way to easily access it, we can start to build the public facing part of our API.

Remember, a typical RESTful API uses URLs to decide what "resource" we are working with and the HTTP method to decide what to do with it. So we need a way to say if the user goes to the URL `/articles` and they do a `POST` request, we should look at the JSON data they've sent and create a new article in the database.

9.1 Routing

Routing is the process of determining what code should be run based on the given URL. This is a concept you'll come across a lot when developing websites. In fact, we'll come across it again next week when we look at React.

When a user loads a page in a Laravel app, after setting everything up, Laravel has a look to see if the URL matches any of the defined **routes** (which live in the `routes` directory). A route represents a specific URL and HTTP method combination. If Laravel finds a match it then runs the code that the route points to.

First, let's add a route to handle `POST` events sent to `/articles`. This will point to code that handles creating a new article.

Add a `POST` route for `/articles` to `routes/api.php`:

```
// when the user makes a POST request to the URL /articles
// call the store method of the Articles controller
$router->post("articles", "Articles@store");
```

We use the `$router` object, which is provided to use by Laravel. It has methods for each of the HTTP methods (e.g. `$router->get()`, `$router->post()`, etc.). These methods take a URL fragment as the first argument and a controller method as a second argument (we'll get to these in a second). So the code above says "If the user makes a `POST` request to `/articles`, run the `store()` method of the `Articles` controller.

9.2 Controllers

Controller is another term that comes up a lot in software development. A controller's job is to bring together all the other bits of code that make up an app: to *control* what happens. In Laravel, controllers are called by the router: they take the request, do whatever it is that needs doing, and then return a response.

Run the following inside your Vagrant box to create an Articles controller:

```
artisan make:controller Articles --api
```

This will create a file (`app/Http/Controllers/Articles.php`), which contains the boilerplate for a controller. The `--api` flag adds all the standard API methods to the controller for us.[1]

Laravel has added a `store` method to `Articles` for us:

```
public function store(Request $request)
{
  // handle post request
}
```

[1]This isn't always necessary if you're only adding a few routes to the controller, but in this case we'll be adding all the standard methods.

9.3 The API

We now know almost all we need to know to build a fully functioning API. Let's go through each of the routes in turn.

9.3.1 Storing

We need to create new articles, so we'll be using the `Article` model in our controller. But that lives in the `App` namespace, so we'll need to tell the `Articles` controller where to find it:

```
use App\Article;
```

Then we need to update our `store` function to create an article using the data from the request:

```
public function store(Request $request)
{
  // get all the request data
  // returns an array of all the data the user sent
  $data = $request->all();

  // create article with data and store in DB
  // and return it as JSON
  // automatically gets 201 status as it's a POST request
  return Article::create($data);
}
```

[View code on GitHub]

The first thing to notice is the `Request` object getting passed in. This represents the request that the user made. It stores all sorts of useful information, but in this case the only thing we're interested in is the JSON data that the user submitted (the URL and HTTP method have already been dealt with by the router).

We use the `$request->all()` method to get back an associative array of all the data the use submitted. If we were to log this with `dd($request->all())` we'd get something like:

```
array:2 [
  "title" => "My Thoughts on Monkeys"
  "article" => "Are they related to Donkeys?"
]
```

We then use the `create` method from the `Article` class (inherited from Eloquent's `Model` class). This takes an associative array of values, creates a new instance of the model (in this case an article), sets the properties that were provided, and then stores it in the database. In other words, it does everything we need to do to create a new article! It also returns the newly created article instance, which we immediately `return`.

Whatever we `return` from a controller method gets sent back as a response to the user. Laravel is smart enough to output it in the format the user asks for. So, as long as an `Accept: application/json` header has been set, the user will get back the newly created article in JSON format.

Now you can try doing a `POST` request with Postman… but it won't work just yet.

Mass Assignment Vulnerability

By default Laravel won't let you do things that might cause security problems. Accepting whatever the user gives you and sending it straight to the database is one of those cases: known as the **mass assignment vulnerability**.

Imagine you have a `users` table that has an `admin` column for keeping track of whether a user is an admin or not. This is pretty common, so a hacker might try to exploit it. Your user sign-up form probably doesn't have an `admin` option, but a hacker can easily make a sign-up API request directly (with Postman or `curl`). They add an `"admin": true` property to the JSON that they send. Now, if your controller just takes *everything* that they sent and tries to update the database, the hacker has just been given admin rights.

To stop this from happening, if you pass an Eloquent model and array of properties to update, it will only update the properties that you've listed in the model's `$fillable` property. If you've not set this property and try to pass in an array of values you'll get a 500 error.

We need to update the `Article` model to tell it which fields to expect:

```
class Article extends Model
{
  // Only allow the title and article field to get updated via mass
  ↪  assignment
  protected $fillable = ["title", "article"];
}
```

[View code on GitHub]

Now try making the request in Postman. You should get back a newly created article in JSON format.

9.3.2 Listing

Next let's setup the `GET` request for `/articles`. When this request is made we'll want to give back a JSON array of all the articles that have been added to the database so far.

First, add the route:

```
$router->get("articles", "Articles@index");
```

[View code on GitHub]

The `index` method already exists because we created the model with the `--api` flag.

We already know we can get all the articles back from the database with `Article::all()` and that Laravel will automatically format whatever we return into JSON. We also don't need to get any data from the request, as we're just showing *all* articles. So our `index` method is simply:

```
public function index()
{
  // get all the articles
  return Article::all();
}
```

[View code on GitHub]

That's all there is to it: Laravel does all the hard work for us.

9.3.3 Reading

Next we want to add a route for a `GET` request to something like `/articles/1`, where the user wants a specific article back.

As usual, we'll add a route first. This one is going to be a bit different because it needs to allow for *any* valid ID after the `/articles/` bit of the URL. To do this we use a URL parameter: a placeholder for any text in the URL:

```
// {article} is a url parameter representing the id we want
$router->get("/articles/{article}", "Articles@show");
```

[View code on GitHub]

You'll also notice that we've not got three routes, all with the same `articles` prefix in the URL. It can be convenient to group these: not only does it save a bit of typing, the indentation also makes the groupings of related URLS much clearer:

```
$router->group(["prefix" => "articles"], function ($router) {
  // create an article
  $router->post("", "Articles@store");
  // show all articles
  $router->get("", "Articles@index");

  // show a specific article
  $router->get("{article}", "Articles@show");
});
```

[View code on GitHub]

We've wrapped our routes with a call to `$router->group()`, which we've given a `prefix` property of `articles`. All of the routes in this group now have this as the first part of their URL, so we need to remove that prefix from the routes we're putting into the group.

Now we need to add the code for our controller method. The value of the URL parameter automatically gets passed into the method as the first argument, so we

can use that along with `Article::find()` to get the appropriate article from the database:

```
// the id gets passed in for us
public function show($id)
{
  return Article::find($id);
}
```

[View code on GitHub]

9.3.4 Updating

Next we'll need a `PUT` route for URLs like `/articles/1`.

Again, we'll add the route first:

```
$router->put("{article}", "Articles@update");
```

[View code on GitHub]

Make sure you add this *inside* the prefixed group we created earlier.

`PUT` requests are the most complex sort of request we'll be dealing with:

- They need to get the JSON data that the user sent with the request
- They need to get the existing item out of the database, so they need to know the relevant ID
- They need to update the item and then save the changes back to the database

But, of course, Laravel makes it all quite easy:

```
// request is passed in because we ask for it with type hinting
// and the URL parameter is always passed in
public function update(Request $request, $id)
{
    // find the article with the given id
    $article = Article::find($id);
```

```
    // get the request data
    $data = $request->all();

    // update the article using the fill method
    // then save it to the database
    $article->fill($data)->save();

    // return the updated version
    return $article;
}
```

[View code on GitHub]

We use the `fill()` method that Eloquent models inherit. This lets us pass in an associative array of properties, which then get updated on the model (taking the `$fillable` property into account). We need to call `save()` once we've done updating the model to make sure the changes are persisted in the database.

Four lines of code to do quite a complex set of computations. A lot I know! Don't worry, we'll get rid of some of them later.

9.3.5 Deleting

Finally, let's add a `DELETE` route.

We'll need a URL parameter again, as we want to delete specific articles:

```
$router->delete("{article}", "Articles@destroy");
```

[View code on GitHub]

Again, make sure this goes in the prefixed group we created.

Now update the `destroy` method in `Articles`:

```
public function destroy($id)
{
  $article = Article::find($id);
  $article->delete();
```

```
  // use a 204 code as there is no content in the response
  return response(null, 204);
}
```

[View code on GitHub]

We use `find()` to get the article object. We then call the `delete()` method on the object to remove it from the database. Finally we return a `response()` with no value and a `204: No Content` response code, as there's nothing sensible to give back to the user.

∞

And that's it. We've created a fully functional API in about fifteen lines of code.

9.4 Additional Resources

- Routing
- Controllers

Chapter 10

API Details

Now, while we have got a basic API up and running, it's lacking some fairly important features:

- **404s**: if we request an article that doesn't exist we'll get a 500 error
- **Responses**: the responses contain data we might not want to share
- **CORS Support**: due to the security requirements of modern web browsers, they won't be able to use our API
- **Validation**: we can submit absolutely anything to the API at the moment, which will lead to all sorts of MySQL errors

10.1 Route Model Binding

What if we try to update `/articles/34849`? Currently we'll get a 500 error as there is no article with ID 34849. But a 500 error is of no use to the client making API requests: it just tells them that you've got a bug in your code.

If an article doesn't exist then we should return a 404 status. The client *can* do something with this as it has a very specific meaning: the thing you requested doesn't exist.

This is such a common issue when building websites/APIs, that Laravel has support for it baked-in. We can use **Route Model Binding** to ask Laravel to find the appropriate article for us and, if it doesn't find one, return a 404 status automatically.

We do it using PHP type-hinting. Rather than accepting the standard URL parameter argument, we use type-hinting to say to Laravel "Give us an `Article` instead".

Laravel does a little bit of magic[1] and tries to find the article. If it finds it then it passes in the article object instead of the URL parameter; if it can't find it then it returns a 404 response immediately and doesn't even run the controller code.

To get this working, all we need to do is update all the places in our controller where we accepted the URL parameter `$id` to ask for an `Article` instead. We can then get rid of all the calls to `Article::find()`, as Laravel will have already done that for us:

```
// accept the type-hinted article instead of $id
public function show(Article $article)
{
  // and just return it
  return $article;
}

// accept the article
public function update(Request $request, Article $article)
{
  // no need to find it anymore
  $data = $request->only(["title", "article"]);
  $article->fill($data)->save();
  return $article;
}

// accept the article
public function destroy(Article $article)
{
  // no need to find it anymore
  $article->delete();
  return response(null, 204);
}
```

[View code on GitHub]

[1] People often complain if frameworks do too much "magic". Rails is famous for it. The type-hinting trick is the only case I'm aware of where Laravel can really be accused of it. It's using Reflection to do this, which is when a programming language looks at its own code and works things out from it. It's probably not something you'll need to use much.

10.2 Resources

It would also be good if we had some control over how our data comes back. For example when we're listing all the articles we probably don't want to send the full article text: it would be a massive JSON output if there were lots of really long articles. And we might not want to send everything from the database back to the user: for example, the `created_at` and `updated_at` properties probably aren't that useful.

This is where **Resources** come in. They let us control the format of the JSON output. We create a `Resource` class which tells Laravel how to format a specific model. Then, before returning a response in the controller, we pass it through the resource.

First, let's create one to simplify the output of an article so it only show the `id`, `title`, and `article` fields. As usual, we'll use `artisan` to do most the work for us:

```
artisan make:resource ArticleResource
```

This will create a file in `app/Http/Resources` with the boilerplate code in place. We need to update the `toArray()` method to return the format that we'd like. In the `Resource` class we can access the properties of a model using `$this`:

```
public function toArray($request)
{
  // just show the id, title, and article properties
  // $this represents the current article
  return [
    "id" => $this->id,
    "title" => $this->title,
    "article" => $this->article,
  ];
}
```

[View code on GitHub]

Next, we need to update the `Articles` controller to use the resource for output.

First, tell the controller where to find the `ArticleResource` class:

```
use App\Http\Resources\ArticleResource;
```

Then we need to update all of the methods that return an article to use the resource:

```
public function store(Request $request)
{
    // ... store code

    // return the resource
    // automatically uses the right status code
    return new ArticleResource($article);
}

public function show(Article $article)
{
    // return the resource
    return new ArticleResource($article);
}

public function update(Request $request, Article $article)
{
    // ... update code

    // return the resource
    return new ArticleResource($article);
}
```

[View code on GitHub]

If you do an API call to any of these routes now you'll notice that the response is also wrapped in an outer `data` property by default. It is possible to change this, but it's actually quite useful once you start to deal with more complex APIs (e.g. so you can have a separate property to track pagination).

We also want to make the `list` method only show the `id` and `title` properties. We'll need to create a separate resource for this as we're outputting a different format.

Again, we'll use `artisan` to make a resource:

```
artisan make:resource ArticleListResource
```

Then update the `toArray()` method to output just the `id` and `title` properties:

```
public function toArray($request)
{
  // just the id and title properties
  return [
    "id" => $this->id,
    "title" => $this->title,
  ];
}
```

[View code on GitHub]

Finally we'll need to update the controller.

First let the controller know about `ArticleListResource`:

```
use App\Http\Resources\ArticleListResource;
```

Then update the `index` method. We're returning a collection of articles, so we'll need to use the `Resource`'s `collection` method:

```
public function index()
{
  // needs to return multiple articles
  // so we use the collection method
  return ArticleListResource::collection(Article::all());
}
```

[View code on GitHub]

10.3 CORS

Although we can make requests to our API using Postman and the like, we won't be able to do it from a modern web browser - which is generally where we'd like to be making the requests from. By default browsers won't allow JS from domain *x* to get data from domain *y* as they have different **origins**, a potential security issue. So, if our API is on a separate domain, which it more than likely will be, we need to be able to make cross-origin requests.

Modern browsers use something called **Cross-Origin Resource Sharing** to handle this. They expect certain HTTP headers to be set on API responses to say who can and can't use the API.

We don't want to write the code for this ourselves as it's fairly complicated. Luckily, someone else has done it for us.

Install the CORS library with `composer`:

```
composer require barryvdh/laravel-cors
```

Then run the following command:

```
php artisan vendor:publish --provider="Barryvdh\Cors\ServiceProvider"
```

This copies the necessary config files out of the `vendor` directory.

We need to tell Laravel to load the CORS library, so add the following line to the `providers` property in `config/app.php`:

```
'providers' => [
  // ...other providers
  Barryvdh\Cors\ServiceProvider::class,
],
```

[View code on GitHub]

We also need to tell Laravel that it should use the CORS library as **middleware** for every HTTP request that gets made. Middleware is code that runs before/after the

controller code to modify the request/response in some way.

To add the CORS middleware, in `app/Http/Kernel.php`:

```
protected $middleware = [
  // ...other middleware...
  \Barryvdh\Cors\HandleCors::class,
];
```

[View code on GitHub]

10.4 Validation

You should always validate any data that gets submitted to your site on the server-side. For a good user-experience you should probably also have validation on the client-side using JavaScript, but it's perfectly possible that a user might have JavaScript disabled or that a malicious party might be making a direct request.

The idea of validation is to turn a certain class of `500` errors into `4xx` errors. Remember, client-side code can't do anything useful with `500` errors, but `4xx` errors have specific meanings.

To avoid any MySQL errors we need to at minimum validate the following:

- `required`: any database fields that cannot be `null` should have the `required` validation
- `max:255`: if you're storing data in a `VARCHAR` then make sure you have max length validation that matches the `VARCHAR` length
- `date`/`integer`/`string`: check formats before inserting into MySQL. These are not the database migration types, but validation specific rules - so `VARCHAR` and `TEXT` both count as `string` for validation. You will also need the `nullable` validation if the field is not required.

As well as the cases above there are plenty of other bits of validation that you can do easily with Laravel.

As with everything in Laravel, we'll need to create a new type of class. In this case it's a `FormRequest` class. The `FormRequest` class actually inherits everything from the `Request` class that we're already using in our controller when we want to work with the request the user made.

Use `artisan` to create a `FormRequest` class:

```
artisan make:request ArticleRequest
```

This will create a file in `app/Http/Requests`. The `ArticleRequest` class has two methods. We first need to return `true` from the `authorize` method: this method can be very useful when you support user logins in your API, but for now we'll just assume everyone can make requests. The `rules` method needs to return an associative array, where the property name is the JSON property that needs validating and its value is an array of validation rules:

```
class ArticleRequest extends FormRequest
{
  // always return true
  // unless you add user logins
  public function authorize()
  {
    return true;
  }

  // an array of validation rules for each JSON property
  public function rules()
  {
    return [
      "title" => ["required", "string", "max:100"],
      "article" => ["required", "string", "min:50"],
    ];
  }
}
```

[View code on GitHub]

Finally we need to update the `Articles` controller to use our validated request instead of the standard `Request` object. First let it know where the `ArticleRequest` class lives:

```
use App\Http\Requests\ArticleRequest;
```

And then update the type-hinting on the `store` and `update` methods to use the `ArticleRequest` class instead of `Request`:

```
public function store(ArticleRequest $request)
{
  // ... store code
}

public function update(ArticleRequest $request, Article $article)
{
  // ... update code
}
```

[View code on GitHub]

This works because `ArticleRequest` is a descendant of `Request`, so it has all of the same methods and properties.

10.5 Additional Resources

- Route Model Binding
- API Resources
- Laravel CORS Library
- Middleware
- Validation

Chapter 11

Advanced Database Structures

It's very common that we'll need to store relationships between different types of things. For example an article can have comments and tags. And while comments belong to a specific article, tags can belong to multiple articles. We need some way to store these relationships in our database and to deal with them in Laravel.

A naive approach[1] to storing comments for an article might be to add a `comments` column and then store an array of comments. But there isn't an `ARRAY` type in SQL, so you'd have to **serialize**[2] it somehow. This is possible, but it will lead to a world of trouble later on.

Another wrong-footed approach[3] would be to create a bunch of columns called `comment_1`, `comment_2`, &c. This is arguably worse than the serialisation approach as it means the database structure limits the number of comments that can be stored - and we'd have to change the database structure if we wanted to store more comments[4]. If we wanted to store more than just the comment text we'd also need *multiple* columns for each comment: `comment_text_1`, `comment_email_1`, &c.

11.1 Relational Databases

SQL is designed to be used with **Relational Database Management Systems** (RDBMS). The key word being *relational*. Although each table in the database should repre-

[1] This was what I did my first time building a database. I then learnt the proper way to do it and spent the following two months rewriting the whole thing in secret and updating the site without telling the client.
[2] Turning a data structure into data that can be stored/transferred. This is actually what WordPress does for quite a lot of its data.
[3] Which I've seen in production software
[4] As I said, I've seen this in production software. And they charged the client for such database changes too!

sent one type of thing, the database allows us to map relationships between rows in one table to rows in another table. This is based on some pretty rigorous mathematical logic, so it's got well known performance characteristics.

Why Not NoSQL?

> *It's a curious thing about our industry: not only do we not learn from our mistakes, we also don't learn from our successes*
>
> - Keith Braithwaite

It's become very trendy to use NoSQL databases to make API driven apps[5]. The most popular being MongoDB.

MongoDB is a "document" database: it's designed for storing arbitrary data, as opposed to specific data types. If that's all you use it for then it's really efficient - although such features are now common in SQL databases like PostgreSQL. But people got carried away and tried to use it to store *relational* data and ended up getting themselves into lots of issues - do a search for "MongoDB to PostgreSQL" for many an article on the subject.

It turns out that *most* data you'll ever want to store and work with is relational, so you should probably stick to a relational database most the time.

That's not to say that NoSQL has no place in the world: it's really good for specific uses cases and more often than not should be used *alongside* SQL databases.

For example it's very common to pair up ElasticSearch with an SQL database: searching an SQL database for key terms is either very inefficient or involves a lot of extra tables and code. ElasticSearch does it all for us with very little effort. However, you'd still want to store your main data in an SQL database.

Another case is using a graph database (such as Neo4j) alongside SQL. Although SQL is good at storing specific relationships, it's very bad at exploring those relationships: for example, Facebook finding all of your friends' friends. In fact the relationships don't need to get very complicated before it would take SQL millennia to find every possible relationship. Graph databases are designed specifically to find relationships and can do this very quickly. Again, you'd probably want to store your main data in an SQL database

There are various types of relationship that we can store in SQL, but the two most common are:

- **One-to-Many**: when some[6] of one type of thing belong to another type of thing, for example articles have lots of comments, but comments belong to a specific article.
- **Many-to-Many**: when some of one type of thing related to some of another type of thing, for example articles can have lots of tags, and tags belong to lots of articles.

Other Types of Relationships

We won't be looking at the following types of relationship in any detail, but they're worth knowing about:

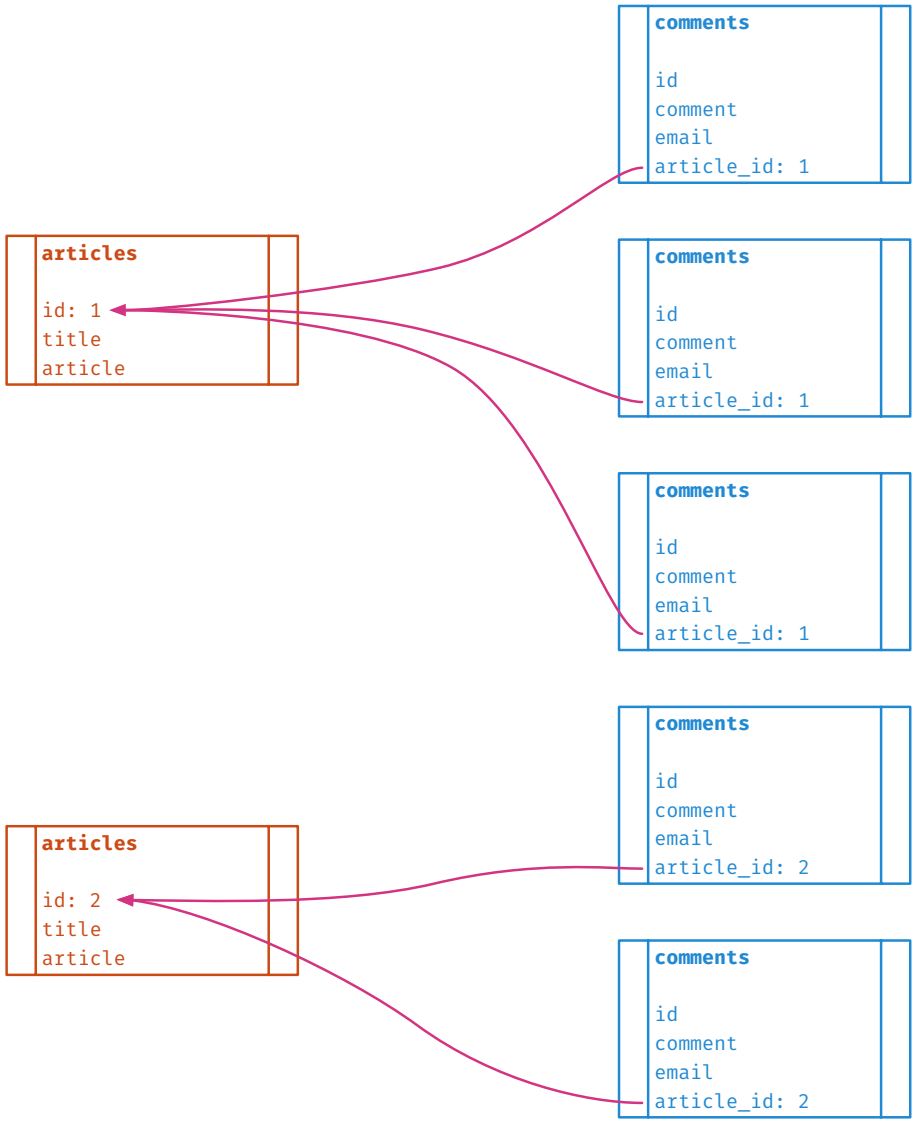
- **One-to-One**: when two types of thing are linked directly. For example a `people` table might have a one-to-one link to an `addresses` table. However, in many cases these are actually one-to-*many* relationships: e.g. two different people might share the same address.
- **Has-Many-Through**: when some of one thing relate to another type of thing *via* a third type of thing. For example if our blog had users we could find all the comments on articles that the user wrote: the comments belong to the article and the article belongs to a user. This isn't technically a new type of relationship: it's just two relationships strung together.
- **Polymorphic Relationships**: sometimes it's useful to represent a hierarchy. Say that we had various type of blog post (e.g. `articles`, `links`, `videos`) and we stored each type in its own table as they require different values to be stored. We could have a parent `posts` table which stores the common information and allows us to get all of them out with one query. Relational algebra doesn't express this sort of relationship well, so it is not usually part of the RDBMS. This means it isn't brilliant from a performance perspective and the way it's stored in the database is dependent on the DB library that you're using.

[5] You might be noticing a pattern here.
[6] Zero or more

11.2 One-to-Many Relationships: Comments

Each article can have **many** comments, but each comment can only belong to **one** article. So this is a one-to-many relationship. One-to-many relationships are asymmetrical, in that one sort of thing effectively belongs to another sort of thing.

We can store this relationship by creating an `article_id` column on the `comments` table that references the ID of the article each comment belongs to. Under the hood, MySQL can really efficiently use this structure to join together related data.



One-to-many relationships

11.2.1 Database Migration

First, we'll need to create a new database migration. We should create a `Comment` model at the same time as we'll need to work with comments in the Laravel code. Run the following `artisan` command:

```
artisan make:model Comment -m
```

A comment belongs to an article and has an email address and the comment text. As well as adding the columns, we should also tell MySQL that the `article_id` column points to the `id` column of the `articles` table. We do this by setting up a **foreign key** constraint.

> **Foreign Keys**
>
> Setting up a foreign key constraint tells MySQL that a column on one table points to a column on another table (or even the same table).
>
> You *could* create the `article_id` column without creating a foreign key and everything would still work. However, by adding the foreign key we get certain data integrity guarantees:
>
> - It's not possible to set an `article_id` that doesn't exist
> - If an article is removed from the database, MySQL can remove all related comments automatically (this is known as **cascading** the delete operation)
> - We can also cascade update operations, which can be useful if you aren't using `AUTO_INCREMENT` for IDs
>
> Data integrity is really important, so it's always worthwhile spending the extra bit of effort to create a foreign key.

Update the `up` method of the newly created database migration:

```
public function up()
{
  Schema::create("comments", function (Blueprint $table) {
    // create the basic comments columns
    $table->increments("id");
```

```
    $table->string("email", 100); // use a VARCHAR
    $table->text("comment"); // could be any length
    $table->timestamps();

    // create the article_id column
    $table->integer("article_id")->unsigned();

    // set up the foreign key constraint
    // this tells MySQL that the article_id column
    // references the id column on the articles table
    // we also want MySQL to automatically remove any
    // comments linked to articles that are deleted
    $table->foreign("article_id")->references("id")
          ->on("articles")->onDelete("cascade");
  });
}
```

[View code on GitHub]

Don't forget to run `artisan migrate` once you've saved the migration file.

11.2.2 Eloquent Models

Now we've updated the database structure we need to tell our Eloquent models about the relationship between articles and comments. Then Eloquent can do its ORM magic and join up the different models.

Let's update our Article model to let it know that it can have comments:

```
class Article extends Model
{
  // ...etc.

  // plural, as an article can have multiple comments
  public function comments()
  {
    // use hasMany relationship method
    return $this->hasMany(Comment::class);
  }
}
```

[View code on GitHub]

Now we can easily access a collection of `Comment` objects for an article object instance using its new `comments` property.[7]

We need to setup the other side of the relationship in the `Comment` model (which we created earlier):

```
class Comment extends Model
{
  // setup the other side of the relationship
  // use singular, as a comment only has one article
  public function article()
  {
    // a comment belongs to an article
    return $this->belongsTo(Article::class);
  }
}
```

[View code on GitHub]

Now all of our `Comment` object instances will have an `article` property that gives back the related `Article` object.

We can use `artisan tinker` to get a bit of a better idea of how the ORM relationships work:

```
// first create a comment
$comment = new Comment();
$comment->email = "malala@example.com";
$comment->comment = "A pleasant and life-affirming comment";
$comment->article_id = 1; // assuming you have an article with id 1
$comment->save();

// now try using the article property
$article = $comment->article; // should give you back an article object
$article->id; // 1
$article->title; // the title of the article with ID 1
$article->article; // the article body of the article with ID 1

// now try getting the article's comments
```

[7] Even though we created a method - Eloquent creates the property for us

```
$sameArticle = Article::find(1); // find the article with ID 1
$sameArticle->comments; // a collection containing the comment above
```

[View code on GitHub]

11.2.3 Routing & Controller Logic

Now that we've got the `Comment` model setup we can add the necessary controller logic to the API. We're going to want two new routes: one to add a comment to an article and one for showing all the comments that belong to a specific article. Both of these routes work with a specific article, so we'll need to get the ID as a URL parameter.

We'll setup the following routes:

- `POST /articles/<article_id>/comments`: create a comment for the given article ID
- `GET /articles/<article_id>/comments`: get all comments for the given article ID

We could add two methods to the `Articles` controller to handle these, but to keep things tidy let's create a new `Comments` controller:

```
artisan make:controller Comments --api
```

Now we can add the routes:

```
$router->group(["prefix" => "articles"], function ($router) {
    // ...article routes

    // comment routes
    $router->post("{article}/comments", "Comments@store");
    $router->get("{article}/comments", "Comments@index");
});
```

[View code on GitHub]

Now we'll need to update the `Comments` controller. First, let the controller know about the `Comment` *and* `Article` models:

```
use App\Article;
use App\Comment;
```

We'll do the `GET` request first as it's very simple. We'll use route model binding to get the correct article object passed in for us (or a 404 if it's not found). And then we just use the `comments` property to get the collection of all that article's comments:

```
// use route model binding on the URL parameter
// then we'll get the article passed in for us
public function index(Article $article)
{
  // use the comments property of the article model
  return $article->comments;
}
```

[View code on GitHub]

Next, let's handle the `POST` request. This one is a little more involved, as we'll need to deal with the request JSON. Again, we'll use route model binding to get the article from the URL. Then we need to get the request data and create a new comment using it. However, we can't use the `Comment::create()` method, as we need to make sure it has a valid `article_id` property, otherwise MySQL won't let us store it. Instead we'll pass the data in as the first argument when we create a new `Comment` object: this assigns the properties, as with `create()`, but doesn't save it to the database. We'll store it in the database using the article model:

```
// we need to accept Request first, using type hinting
// and then use route model binding to get the relevant
// article from the URL parameter
public function store(Request $request, Article $article)
{
  // create a new comment, passing in the data from the request JSON
  $comment = new Comment($request->all());

  // this syntax is a bit odd, but it's in the documentation
  // stores the comment in the DB while setting the article_id
```

```
  $article->comments()->save($comment);

  // return the stored comment
  return $comment;
}
```

[View code on GitHub]

Passing the data in when we create the comment counts as mass assignment, so we need to make sure we update the `Comment` model's `fillable` property so we don't get a mass assignment vulnerability error:

```
protected $fillable = ["email", "comment"];
```

[View code on GitHub]

11.2.4 Validation

We *always* need to add validation to any data sent to the server. First, use `artisan` to create a `CommentRequest`:

```
artisan make:request CommentRequest
```

Then update the `authorize()` and `rules()` methods:

```
class CommentRequest extends FormRequest
{
  // set to true so the request goes through
  public function authorize()
  {
    return true;
  }

  public function rules()
  {
    return [
      // required, use email validation rule
      // and VARCHAR(100), so make sure no longer than 100
```

```
      "email" => ["required", "email", "max:100"],

      // required and a string
      "comment" => ["required", "string"],
    ];
  }
}
```

[View code on GitHub]

You'll also need to update the `Comments` controller to use the validated request. First let the controller know where to find the `CommentRequest` class:

```
use App\Http\Requests\CommentRequest;
```

Then update the type-hinting to use `CommentRequest` instead of `Request`:

```
// use CommentRequest instead of Request
public function store(CommentRequest $request, Article $article)
{
  // ...store code
}
```

[View code on GitHub]

11.2.5 Resource

Finally, we'll need to update how our comments get formatted in JSON. Again, we don't need to sent the timestamps columns and we also don't need the `article_id` column: if you've made an API request to get the comment then you must already know the article ID.

Use `artisan` to create a resource for comments:

```
artisan make:resource CommentResource
```

Then update the `toArray()` method to return only the columns we want:

```
public function toArray($request)
{
  return [
    "id" => $this->id,
    "email" => $this->email,
    "comment" => $this->comment,
  ];
}
```

[View code on GitHub]

Now to use it in the controller. Tell the `Comments` controller where to find the resource class:

```
use App\Http\Resources\CommentResource;
```

Then update the `index` and `store` methods:

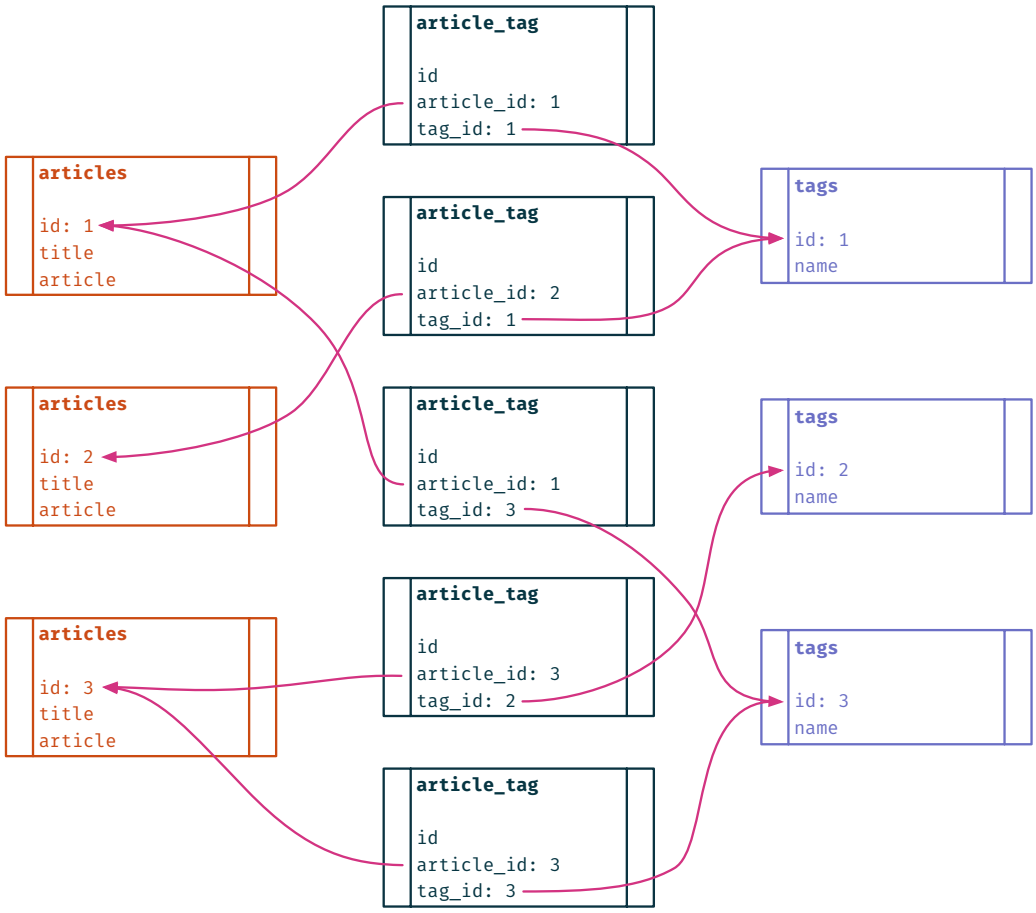
```
public function index(Article $article)
{
  // return a collection of comments
  return CommentResource::collection($article->comments);
}

public function store(CommentRequest $request, Article $article)
{
  // ...store code

  // return a single comment
  return new CommentResource($comment);
}
```

[View code on GitHub]

11.3 Many-to-Many Relationships: Tags

Tags and articles have a more complex relationship than comments and articles. An article can have any number of tags, but a tag can also belong to any number of articles. This is a many-to-many relationship. These relationships are symmetrical: neither side *belongs* to to the other side.

We can't reference the article or tag ID from the other table in this case, as that way we could only reference a single item. In this case we need a **pivot table**. The pivot table *just* stores the relationship between an article and a tag using an `article_id` and a `tag_id` column:



Many-to-many relationships. Simples!

It is possible to store additional information about the relationship on a pivot table, which Eloquent makes available on the model's `pivot` property, but in our case this won't be necessary.

Termlists

It's quite common with databases to have tables which effectively just store a list of strings with an associated ID: e.g. countries, cities, genders, etc. These are sometimes referred to as **termlists** (because it's a list of terms):

id	country
1	Algeria
2	Burkina Faso
3	Cambodia
…	…

id	gender
1	Female
2	Male
3	Non-Binary
…	…

It's a good idea to use the same column name for the string column for all your termlists. For example rather than having a `country` column in the `countries` table and a `gender` column in the `genders` table, you should always call the column something like `name`:

id	name
1	Algeria
2	Burkina Faso
3	Cambodia
…	…

id	name
1	Female
2	Male
3	Non-Binary
…	…

You normally need to do similar sorts of things to all your termlists: e.g. list them, reorganise them, add items, remove items, etc. By giving them all the same structure you can reuse the same bits of code for all of them.

Generally it's not very useful to store timestamps on termlist tables.

11.3.1 Database Migration

We'll need a `Tag` model shortly, so let's create that at the same time as the database migration using `artisan`:

```
artisan make:model Tag -m
```

The database migration for the tags is going to be more complicated than for comments as we'll need to create *two* tables: a termlist table for tags and the pivot table. We can create more than one table in a single migration, just by calling `Schema::create()` twice:

```
public function up()
{
  // create the tags table
  // it's a termlist so call the string column name
  // don't need timestamps - not very useful here
  Schema::create("tags", function (Blueprint $table) {
    $table->increments("id");
    $table->string("name", 30);
  });

  // create the pivot table using the Eloquent naming convention
  Schema::create("article_tag", function (Blueprint $table) {
    // still have an id column
    $table->increments("id");

    // create the article id column and foreign key
    $table->integer("article_id")->unsigned();
    $table->foreign("article_id")->references("id")
          ->on("articles")->onDelete("cascade");

    // create the tag id column and foreign key
    $table->integer("tag_id")->unsigned();
    $table->foreign("tag_id")->references("id")
          ->on("tags")->onDelete("cascade");
  });
}
```

[View code on GitHub]

Notice that we've called the pivot table `article_tag`. That's because Eloquent can automatically find pivot tables if they're named using the singular version of the two tables being joined together, in alphabetical order, with an underscore between them. You can call it whatever you like, but you'd need to let Laravel know if you chose something else.

We're not done yet. If we create two tables we need to make sure we update the `down()` method so that both tables are removed if we rollback the migration. It's important that we remove the tables in the opposite order from how they were created: if we try and drop the `tags` table before the pivot table, the foreign key constraints in the pivot table would all fail (as there would be no valid `tag_id`s to point to):

```
// update the down method
public function down()
{
  // remove the pivot table first
  // otherwise all the tags foreign key contraints would fail
  Schema::dropIfExists("article_tag");

  // then drop the tags table
  Schema::dropIfExists("tags");
}
```

[View code on GitHub]

As a general rule, if you're creating multiple tables, you should always drop them in reverse order.

11.3.2 Eloquent Models

First, we need to tell the `Tag` model not to worry about setting the timestamp columns, otherwise we'll get a MySQL error when we try to add a tag. We do this by setting the `public` property `timestamps` to `false`:

```
class Tag extends Model
{
  // don't need timestamps
  // no idea why this one is public
  public $timestamps = false;
}
```

[View code on GitHub]

Next we should let Eloquent know about the relationship between articles and tags. In the `Tag` model add an `articles` method and use the `belongsToMany()` relationship method:

```
class Tag extends Model
{
  // ...etc.

  // using the belongsToMany() method
```

```
  // as it's a many-to-many relationship
  public function articles()
  {
    return $this->belongsToMany(Article::class);
  }
}
```

[View code on GitHub]

We'll also need to tell the `Article` model about tags. Again we'll use a plural name for the method. And many-to-many relationships are symmetrical, so we use the *same* relationship method:

```
class Article extends Model
{
  // ...etc.

  // use the belongsToMany() method again
  public function tags()
  {
    return $this->belongsToMany(Tag::class);
  }
}
```

[View code on GitHub]

The API we're creating takes an array of tag strings (e.g. `["Monkey", "Fish", "Spoons"]`) as part of the `POST`/`PUT` request when creating/updating an article. We'll need to make sure that we don't create the same tag more than once. Let's add a `static` method to the `Tag` class so that we can pass it an array of strings and it will return a collection of `Tag` objects: if the tag already exists it will find it in the database and if the tag doesn't exist it will create it for us. We'll call the method `fromStrings`:

```
class Tag extends Model
{
  // ...etc.

  // accepts the array of strings from the request
  public static function fromStrings(array $strings)
  {
```

```
    // turns into a collection and maps over
    return collect($strings)->map(function ($string) {
      // remove any blank spaces either side
      $clean = trim($string);

      // then call the make method
      return static::make($clean);
    });
  }

  // a method that takes a string and either
  // returns the existing tag from the database
  // or creates a new one if it doesn't exist
  private static function make($string)
  {
    // check if tag already exists
    // will be either a Tag object or null
    $exists = Tag::where("name", $string)->first();

    // if tag exists return it, otherwise create a new one
    return $exists ? $exists : Tag::create(["name" => $string]);
  }
}
```

[View code on GitHub]

Because we're using `create()`, which uses mass assignment, we'll also need to add a `$fillable` property to the `Tag` model:

```
class Tag extends Model
{
  // ...etc.

  // name should be fillable
  protected $fillable = ["name"];

  // ...etc.
}
```

[View code on GitHub]

Now, let's use `artisan tinker` to see if this works:

```
// try the parse method
// add in some spaces to check trimming
Tag::fromStrings(["Monkey", "Fish ", " Spoons "]);

// get all the tags
// a collection with "Monkey", "Fish", and "Spoons" Tags
Tag::all();

// try again, with some new tags and some old
Tag::fromStrings(["Fish", "Penguin", "Wombat "]);

// get all the tags
// a collection with "Monkey", "Fish", "Spoons"
// "Penguin" and "Wombat" tags
// Fish should only appear once
Tag::all();
```

[View code on GitHub]

11.3.3 Controller Logic

We won't need to add any new routes for tags because they get added as part of the article `POST`/`PUT` request. But we will need to update the `store` and `update` methods in the `Articles` controller.

First we'll need to let the `Articles` controller know where to find the `Tag` class:

```
use App\Tag;
```

Then we'll need to update the methods. First, we'll change the `$request->all()` call into a `$request->only()` call: this isn't strictly necessary (as the `$fillable` property will ignore `tags` unless we add it), but it makes it clear that we don't want the tags added just yet. Then we'll use the `Tag::fromStrings()` method we created earlier to get back a collection of tags that we want to set on the article. We then use the `sync()` method, which takes an array of `id`s, to automatically sort out all of the pivot table entries for us:

```
class Articles extends Controller
{
  // ...etc.

  public function store(ArticleRequest $request)
  {
    // only get the title and article fields
    $data = $request->only(["title", "article"]);
    $article = Article::create($data);

    // get back a collection of tag objects
    $tags = Tag::fromStrings($request->get("tags"));

    // sync the tags: needs an array of Tag ids
    $article->tags()->sync($tags->pluck("id")->all());

    return new ArticleResource($article);
  }

  // ...etc.

  public function update(ArticleRequest $request, Article $article)
  {
    // only get the title and article fields
    $data = $request->only(["title", "article"]);
    $article->fill($data)->save();

    // get back a collection of tag objects
    $tags = Tag::fromStrings($request->get("tags"));

    // sync the tags: needs an array of Tag ids
    $article->tags()->sync($tags->pluck("id")->all());

    return new ArticleResource($article);
  }

  // ...etc.
}
```

[View code on GitHub]

You'll notice we use the same logic for tags in both methods. It would be good to get rid of the repetition. We can add a method to our `Article` model:

```
class Article extends Model
{
  // ...etc.

  // just accept an array of strings
  // we don't want to pass request in as there's no
  // reason models should know about about the request
  public function setTags(array $strings)
  {
    $tags = Tag::fromStrings($strings);

    // we're on an article instance, so use $this
    $this->tags()->sync($tags->pluck("id")->all());

    // return $this in case we want to chain
    return $this;
  }
}
```

[View code on GitHub]

And then update the `Articles` controller:

```
class Articles extends Controller
{
  // ...etc.

  public function store(ArticleRequest $request)
  {
    $data = $request->only(["title", "article"]);

    // use the new method
    $article = Article::create($data)->setTags($request->get("tags"));

    return new ArticleResource($article);
  }

  // ...etc.
```

```
  public function update(ArticleRequest $request, Article $article)
  {
    $data = $request->only(["title", "article"]);
    $article->fill($data)->save();

    // use the new method - can't chain as save returns a bool
    $article->setTags($request->get("tags"));

    return new ArticleResource($article);
  }

  // ...etc.
}
```

[View code on GitHub]

11.3.4 Validation

We'll need to add validation rules for tags to `ArticleRequest`. We'll use the `array` validation type for the `tags` field. We can also use the `.*` notation to add validation for the items inside the `tags` array. In this case we want them to be strings with a maximum length of 30 (we used a `VARCHAR(30)` for tags):

```
public function rules()
{
    return [
       "title" => ["required", "string", "max:100"],
       "article" => ["required", "string", "min:50"],
       "tags" => ["required", "array"], // check tags is an array
       "tags.*" => ["string", "max:30"], // check members of tags are
       ↪  strings
    ];
}
```

[View code on GitHub]

11.3.5 Resource

Finally, let's update our article resources to include tags as an array. In `ArticleResource` and `ArticleListResource`:

```
public function toArray($request)
{
    return [
        // ... other properties

        // pluck the name property of each tag
        "tags" => $this->tags->pluck("name"),
    ];
}
```

[View code on GitHub]

11.4 Additional Resources

- Eloquent: One to Many Relationships
- Eloquent: Many to Many Relationships

Chapter 12

Deploying a Laravel API

Laravel is fairly easy to setup on a server, but it does require full control over your server's setup, which you generally won't have if you're using shared hosting. There are various options:

- You can use Heroku, which will host your site for free, but the server goes to sleep after 30 minutes of inactivity
- There's also AWS, which is often free for the first year of the most basic services. Just be warned, AWS can be *deeply* confusing and if you forget to turn off your server before the free period runs out they'll start charging you.
- Another option would be to use a VPS host like Digital Ocean, but that does require a certain amount of setup and will cost $5 a month. If you sign up you'll get $10 free credit, which is enough to get a server running for a couple of months (**full transparency**: I get some free credit if you end up sticking with Digital Ocean after using that link)

You can use Laravel Forge to make setting up all of the above servers easier. But it costs $12 a month **on top** of the hosting fees, so it's probably not worth it unless you really don't want to deal with setting up servers.

Moral Support

Setting up servers can be deeply scary early on (in fact, even after 10 years it's still anxiety-inducing at times). The only way to get better at it is to just keep doing it. It can take a long time, but it does all go in eventually.

Glossary

- **Class**: An abstract representation of an object instance
- **Client-Side**: Code that runs on the user's computer. As opposed to server-side, which runs on the server.
- **Controller**: A piece of code that is run for a specific route, whose job it is to get/update the relevant data and return a response to the user.
- **CORS**: Cross-origin resource sharing - a safety feature that permits browsers to make requests to APIs on different domains.
- **Dependency Injection** Rather than hard-coding a dependency on a specific class, taking advantage of polymorphism and passing in a class that implements a specific interface.
- **Foreign Key**: A relationship between two tables in MySQL that is enforced by the database.
- **Instance**: An object is an instance of a specific class with its own set of properties
- **Many to Many**: A relationship between two tables of a database where the items of table A can be linked to many items from table B, and the items from table B can be linked to many items in table A.
- **Middleware**: A piece of code that runs before/after the controller code to change the request/response in some way.
- **Namespace**: A set of classes where each class has a unique name. We can have two classes with the same name as long as they are in different namespaces.
- **One to Many**: A relationship between two tables of a database where the items of table A can be linked to many items from table B, but items from table B can only be linked to one item in table A.
- **ORM**: Object Relational Mapper - allows us to access data from a database using standard objects.

- **Pivot Table**: A table that stores a many-to-many relationship.
- **Provisioning**: Setting up a server/virtual machine so that it's ready to be used.
- **Relational Database Management Systems**: A database made up of tables (that represent one type of thing) and the relationships between items in those tables.
- **Resource**: Allows us to control the JSON output of our API.
- **Static**: A property or method that belongs to a class rather than an object instance

Colophon

Created using TEX

Fonts

- Feijoa by Klim Type Foundry
- Avenir Next by Adrian Frutiger, Akira Kobayashi & Akaki Razmadze
- Fira Mono by Carrois Apostrophe

Colour Palette

- Solarized by Ethan Shoonover

Written by Mark Wales
smallhadroncollider.com

∞

February 24, 2019