7

Server-Side Programming
with Laravel

There is no such thing as perfect security, only varying levels of insecurity.

- Salman Rushdie

Contents

How To Use This Document

Bits of text in red are links and should be clicked at every opportunity. Bits of text in `monospaced green` 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:

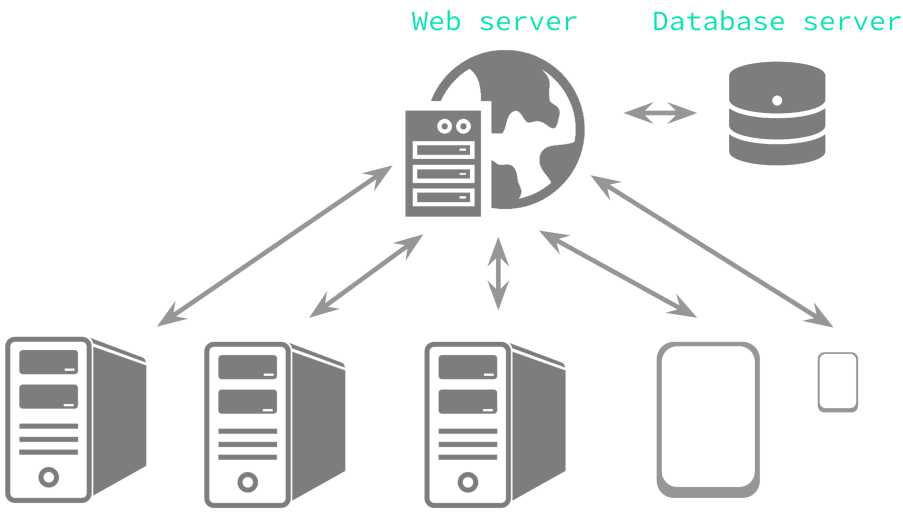
- Preview (Mac)
- Edge (Windows)
- Hypothes.is
- Google Drive (*not* Google Docs)
- Dropbox

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

Server-Side Programming

Server-side code runs on a web server, as opposed to in a web-browser (**client-side**). Common things that server-side code is used for include sending email, working with files, and interacting with a database.



Server-side Architecture

Probably the two biggest concerns of server-side programming are **performance** and **security**: making sure code runs quickly, controlling who can access files, storing data securely, &tc.

We'll be using PHP for most of our server-side code,[1] but you can use almost any programming language for the same purpose: Ruby, Python, Java, Haskell, and even JavaScript.

[1] It's free, popular, and you spent all last week learning it

1.1 Web Servers

PHP doesn't have to be run on a web server, as you have experienced by running PHP using the command-line, but to build apps that can be accessed by other people from a web-browser, we need a web server.

A web server is just a piece of software that runs on a computer that is connected to the internet and "serves" files to a user who requests them. Every time you visit a website you're talking to a web server. We'll look into precisely how this works when we look at HTTP later in the course.

Common web servers are NGINX ("Engine X") and Apache on Linux, or IIS (Internet Information Services) on Windows.

LAMP/LEMP

LAMP/LEMP is most common server **stack** in the world. More than 70% of websites are built with it:

Linux	Operating system
Apache / **E**ngine X	Web server
MySQL / **M**ariaDB	Database
PHP / **P**erl / **P**ython	Programming language

LAMP is the older of the two and, as such, still the most heavily used. But LEMP is becoming increasingly popular as NGINX was designed specifically for high performance with modern web applications.

1.2 Additional Resources

- MDN: What is a web server?
- NGINX vs. Apache (a bit biased)

Chapter 2

Vagrant

When we're developing server-side code we need to run it somewhere. Obviously we'll want to run it on our own computer, but every computer has a slightly different setup. For example, different versions of macOS have different versions of PHP and Windows doesn't have built-in PHP support.

Ideally we want to develop on something that's as close to the production server as possible. Otherwise there might be bugs in our code that are **environment** specific. Setting up our local machines to match the server might not be possible: the specific versions may not be available for your particular setup and, unless you use Linux, the operating system will be completely different.

2.1 Virtual Machines

Vagrant allows us to run a **virtual machine** (VM) on our computer. A virtual machine is a computer made entirely out of software. As far as the VM is concerned it's a real computer, but in reality it's all just `0`s and `1`s. It's the Matrix basically.[1]

> **Terminology**
>
> A virtual machine is often referred to as the **guest** and the computer it runs on as the **host**.

Vagrant does all sorts of clever tricks to make it so that the guest VM has access

[1] There is no spoon

to the files on the host computer. It does a bunch more clever tricks to make it so that the host computer can access the guest VM. If we're still going for protracted Matrix references, it's sort of like Morpheus.[2]

2.2 Configuring Vagrant

When working with Vagrant boxes we start off with a file called `Vagrantfile` that has the configuration details of the virtual machine we're creating.

We generally have one `Vagrantfile` per project, as each project will probably have a slightly different setup and should be kept separate.

There are various key bits of configuration:

- `config.vm.box`: The **box** to build the Vagrant instance from. A box is just a pre-made VM setup: usually an operating system with some pre-installed and configured software. See Vagrant Cloud for a list of available boxes.
- `config.vm.network`: The IP address of the guest VM. Useful for communicating with the VM. If running lots of VMs at once, they should have different IP addresses.
- `config.vm.hostname`: The local domain name to use for the VM (resolves to the IP address above) if the vagrant-hostsupdater plugin is installed.
- `config.vm.synced_folder`: Which folder on the host to sync into the guest virtual machine. This is how you get your files "onto" the VM.

2.3 Using Vagrant

We control our Vagrant machine using the command-line, and must first navigate to the directory that has our `Vagrantfile` of the machine we wish to control (or one of its subdirectories).

2.3.1 Starting a Vagrant Machine

To start a Vagrant VM:

```
vagrant up
```

[2] Fine, I'll stop. It'll be B*Witched lyrics next...

This can take quite a long time the first time you run it, particularly if you've not used the box before, as it will need to download it (and they tend to be quite large - it's an entire computer in software after all) and once it's up and running it may well **provision** it (run the initial configuration).

If a VM has been run before then this process can still take a few minutes, depending on whether you've left it running previously.

> **`.vagrant` Directory**
>
> When a Vagrant VM is created a hidden `.vagrant` directory is created in the same directory as the `Vagrantfile`. This keeps track of the VM.
>
> For that reason you should be careful moving or restructuring your project folder once the VM is created.

2.3.2 Stopping a Vagrant Machine

While a Vagrant VM is running it is using your computer's resources (CPU and RAM), so when not needed it should be turned off:

```
vagrant halt
```

A stopped Vagrant VM still uses hard-drive space.

2.3.3 Deleting a Vagrant Machine

```
vagrant destroy
```

This command will delete the VM and its virtual hard drive. Any data that is on the VM will be lost. Usually this is just the databases as most files will be on the host machine and these are not removed.

2.4 Scotch Box

Scotch Box is a Vagrant box that includes PHP and MySQL running on an Ubuntu Linux operating system, perfect for LAMP/LEMP development.

It also includes other useful tools for server-side developers, like MailHog that helps us test email sending from our websites and apps.

2.4.1 Getting Started

Scotch Box includes a ready made `Vagrantfile`, so all we need to do is clone it and then run `vagrant up`:

```
# clone scotch box into `my-project`
git clone https://github.com/scotch-io/scotch-box my-project

# go into `my-project` directory
cd my-project

# run vagrant up
vagrant up
```

Once that's finished, we're ready to start using our shiny new virtual machine.

2.5 Getting In

It's often necessary to get onto the VM to run a few commands. First, make sure your machine is running, then run:

```
vagrant ssh
```

If it asks you for a password, it's `vagrant`.

Once you've run this command you'll notice that your command-line probably looks a bit different. That's because you're now seeing the command-line of the VM guest. Anything you run now will be running on the VM.

Luckily all the commands are the same,[3] so you should be able to work your way around. Just remember that the directory structure is going to be a bit different. This is quite dependent on which box you built your VM from, so you may need to look at the relevant documentation to know where things live.

[3] Macs are built on-top of Unix – which is what Linux is based on – and Windows users are using WSL, which *is* Linux

2.5.1 Getting Out

You can get back to your own machine by running:

```
exit
```

This should take you back to the command-line you're familiar with on your own machine.

2.6 Additional Resources

- Vagrant
- Scotch Box
- Vagrant Boxes

Chapter 3

MySQL

Databases are pieces of software that store structured data for us.

The most common form of database is the **relational database**. To get data in and out of a relational database we use a **Structured Query Language** (SQL).

SQL comes in many flavours: MySQL, MariaDB, PostgreSQL, SQL Server, SQLite. We'll be using MySQL.

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. 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

3.1 Database Structure

It's easiest to think of the structure of databases with an example. Let's say that we're going to create a blog which consists of lots of articles. Each article has a title and content (the text of the article) as well as a creation date.

We usually have a single SQL database per project: it stores all of the data for that specific project.

A database is made up of **tables**, which are in turn made up of **rows**, which have one or more **columns**.

A table represents a collection of some sort of *thing*, e.g. blog articles. A row is a single instance of that thing, e.g. an article. And a column represents a specific piece of information about that thing, e.g. an article title:

id	title	content	created_at
1	Where Do Eels Come From?	Nobody knows exactly	2020-04-16
2	Do Moose Get Drunk?	Maybe	2020-04-17

3.2 Column Types

Each column stores a specific type of data. Here are a few of the most common types (although there are many more):

Type	Purpose	Notes
`VARCHAR`	Short pieces of text	Fixed length
`TEXT`	Long pieces of text	Flexible length, with performance hit
`INT`	Whole numbers	Signed (`+`/`-`) and unsigned
`TINYINT`	Small whole numbers	Often used to store boolean values (`0` and `1`)
`DECIMAL`	Decimal point numbers	Need to specify a precision
`DATE`	Dates	`YYYY-MM-DD` format
`DATETIME`	Storing times	If you don't need to support time-zones
`TIMESTAMP`	Storing accurate times	If you do need to support time-zones

3.3 IDs

A row in an SQL database always has to have one **unique** way of identifying it – otherwise there's no way to select/update a single row. For our purposes we'll just use a numeric ID. MySQL can automatically generate these for us.

These unique IDs are called the `PRIMARY KEY` and are often set to `AUTO_INCREMENT`, so the first record has ID `1`, the next is assigned `2` and so on.

If you're adding data to a database that already exists or that is shared with other people your IDs may not start at `1` and they might have gaps – that's not anything to worry about it.

> **Beyond IDs**
>
> More complex systems (particularly ones spread across multiple servers) might need to use a more complex system such as **UUID**s (Universally Unique Identifiers).
>
> The unique aspect of a row can also be a combination of multiple columns. This can be very useful with complex relationships.

3.4 SQL

3.4.1 CREATE DATABASE

We can use `CREATE DATABASE` to add a new database.

```
CREATE DATABASE database_name;
```

3.4.2 SHOW

We can use `SHOW` to see what databases and tables we have.

```
SHOW DATABASES;

SHOW TABLES (IN database_name);
```

3.4.3 USE

Once we've created a database we need to tell MySQL that we want to work with that database:

```
USE database_name;
```

To `USE` or not to `USE`

If you don't run `USE` then you'll need to include the database name in all the SQL queries you run. As we often only work within one database at a time this adds typing.

Without `USE`:

```
SELECT * FROM database_name.articles;
SELECT * FROM database_name.comments;
SELECT * FROM database_name.tags;
```

With `USE`:

```
USE database_name;
SELECT * FROM articles;
SELECT * FROM comments;
SELECT * FROM tags;
```

3.4.4 CREATE TABLE

This is probably the most complex of the commands as we need to define our table structure. Line breaks can make this easier to follow. Until you have a semi-colon at the end of the line MySQL won't try to interpret what you've written.

```
CREATE TABLE articles (
    # add an auto-incrementing ID
    id INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
    # add a title with max length of 100
    title VARCHAR(100),
    # add a body that can be as long as you like
    body TEXT,
    # add a created_at column that gets set automatically
    created_at DEFAULT CURRENT_TIMESTAMP,
    # add an updated_at column that gets updated automatically
    updated_at DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
);
```

Table and Column Naming

Although it's not necessary, you should **always use snake case for your table and column names**: all lower-case with underscores in place of spaces.

e.g. `user_roles`, `created_at`, `date_of_birth`

If you don't do this you will run into all sorts of issues when it comes to working with your database.

3.4.5 SELECT

We can use `SELECT` to get information out of a specific table:

```
SELECT * FROM articles;
```

If we're only interested in some of the columns we can specify this:

```
SELECT id, title, body FROM articles;
```

WHERE

If we're only interested in some of the records we can filter using `WHERE`.

To get articles written before Jan 14th 2020:

```
SELECT * FROM articles WHERE created_at < "2020-01-14";
```

LIKE

To search the database for parts of a field we use `LIKE`:

```
SELECT * FROM articles WHERE title LIKE '%wombats%';
```

Note: the `%` is a **wildcard** and allows us to look for articles that have "wombat" in the title at the start, end, or middle.

3.4.6 INSERT

To insert a new row of data (record) into your table:

```
INSERT INTO articles (title, body) VALUES ('Welcome to my blog!', 'In
↪  this essay I will...');
```

3.4.7 UPDATE

To update fields in a record:

```
UPDATE `articles` SET `title` = 'Welcome to my blog!' WHERE id = 1;
```

Note: the `WHERE` is very important, else all records in the table will have their title updated to the same thing!

3.4.8 ALTER

To change a table after its been created:

```
ALTER TABLE articles ADD COLUMN featured TINYINT(1) DEFAULT 0;
```

3.4.9 DELETE

To delete a row (record) from a table:

```
DELETE FROM TABLE articles WHERE id = 1;
```

Note: the `WHERE` is very important, else all records in the table will be deleted!

3.4.10 DROP

We can use `DROP` to remove a table:[1]

```
DROP TABLE table_name;
```

We can also use `DROP` to remove an entire database:[2]

```
DROP DATABASE database_name;
```

[1] Use with caution!
[2] Use with *extreme* caution!

Joins

We've not actually covered the "relational" part of relational databases. We can use the `JOIN` command to get related data from multiple tables in a very efficient manner.

We'll be covering the *structure* of these relationships later in the course, but we won't actually be using MySQL queries.

3.5 Using MySQL with Vagrant

SSH into your Vagrant machine (with `vagrant ssh`), then run:

```
# login to mysql
# for Scotch Box username and password are "root"
mysql -u <username> -p
```

And once you're done:

```
# in MySQL
# go from MySQL prompt back to command-line on the Vagrant machine
exit
```

You may need to run `exit` a *second* time to leave the Vagrant VM.

3.6 Additional Resources

- MySQL Data Types
- A Visual Explanation of SQL Joins

Chapter 4

Laravel

Laravel is a *modern* object-oriented 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 a site up and running in a few minutes.

4.1 Key Features

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 for server-side code.

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 back-ends might be written completely differently. There are also certain essential bits of tooling, like database migrations, that are not currently 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.

Node is really 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). But if you want to quickly create a web app or API, you'll be ready to go much quicker using something like Laravel.

4.2 Setup

Laravel provides an installer app that creates the app scaffolding for you. 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.

4.3 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).

4.4 Additional Resources

- Laravel Installation
- Laracasts
- PHP in 2019

Chapter 5

Homestead

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:

```
# inside your project folder (cd project-name first!) run
composer require laravel/homestead
```

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
```

You'll also need to edit the `.env` file to use the default Homestead database setup:

```
DB_DATABASE=homestead
DB_USERNAME=root
DB_PASSWORD=secret
```

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.

The `.env` File

There are certain configuration options that rather than being application specific are **environment** (i.e. the machine that the app is running on) specific.

For example, when you're developing your application you can probably get away with using `root` as the root MySQL password, as it's only on your Vagrant VM. You might also want to show full error messages when something goes wrong.

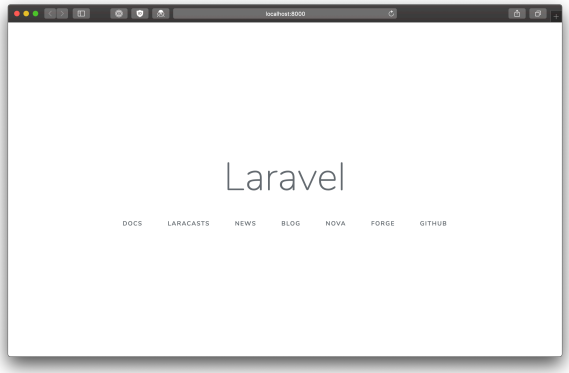
However, on a production server you might need to use a better MySQL password and maybe a different database name. You probably *don't'* want to output full errors, as this can be a big security risk: giving away how your application is built exposes it to known vulnerabilities.

These sort of configuration options go in the `.env` file. This file is specific to each computer that it runs on, so should not be copied or added to version control. Instead, when running on a new machine, you should copy the `.env.example` file and edit that. This will make sure that you don't accidentally copy across settings that are specific to another machine.

5.1 Getting Started

Once Vagrant has finished loading:

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



Laravel up and running

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:

```
# reload vagrant and provision it
vagrant reload --provision # re-run provisioning scripts

# generate a new app key
php artisan key:generate
```

5.2 Additional Resources

- Per-project Homestead Setup
- Why .env?
- Environmental Variables

Chapter 6

Artisan

`artisan` is Laravel's command-line interface. It can do all sorts of things:

- Create most of the files that you'll need for you
- Help setup your databases
- Tell you information about your project
- Let you try out bits of code
- Optimize your code
- Put your site in/out of maintenance mode
- Inspire you to become a better person

And a bunch of other stuff too. We'll be using `artisan` *a lot*.

It's important to run all of your `artisan` commands on the Vagrant machine. 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` on 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.

6.1 Available Commands

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

6.2 Additional Resources

- Artisan

Chapter 7

Database Migrations

Most Laravel sites are going to be **database driven**: storing and retrieving data from a database.

Realistically, you're never going to know the complete structure of your database when you start building a site. 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 site. 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 wombats, fish, and spoons.[1]

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 (e.g. 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. Laravel will handle all of this for us.

7.1 Creating a Migration

The Homestead box creates a database called `homestead` which Laravel has been setup to use by the provisioning script.[2] The `homestead` database is empty by default, so we'll need to add the structure that we need for our site.

We're going to build a simple blog 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 `content` columns.

Make sure you're inside the Vagrant VM 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`).

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;
```

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

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

    /**
     * Reverse the migrations.
     *
     * @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 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->id();
        $table->string("title", 100);
        $table->text("content");
        $table->timestamps();
    });
}
```

[View code on GitHub]

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 `content` column with the "text" type – this is for storing arbitrarily long bits of text.

Table and Column Naming

Remember, you should **use snake-case for table and column names**: all lower-case with an underscore instead of a space. If you don't you're opening yourself up to a lot of issues later on.

[4] It doesn't use `varchar` because Laravel supports lots of different SQL engines, not all of which use the same types as MySQL.

7.2 Running Migrations

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.

You will need to run `artisan migrate` whenever you add new database migrations or if you pull down code that someone else has written.

Migration Batches

Laravel creates a `migrations` table in your database to keep track of which migrations have already run.

When you run `artisan migrate` it will run *all* of the migrations that haven't been run yet in one go.

When we refer to a migration "batch" we mean one or more migrations that ran at the same time.

7.3 Rollback

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

If you want to rollback a single migration, as opposed to the last batch, then you can use:

```
# rollback the last individual migration
artisan migrate:rollback --step=1
```

[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`.

If you make changes to a migration file you will need to run that migration again using `migrate:rollback` followed by `migrate`.

When migrations go bad

Sometimes database migrations go wrong: you might mistype a table name or something, then when you try to run the migrations you just get a bunch of errors. Often no amount of uping and downing can get you out of this. There are two useful commands to try:

```
# runs all of the down methods, then runs all the ups again
artisan migrate:refresh

# wipes the database, then runs all the ups again
artisan migrate:fresh
```

Be aware that running either of these command will wipe *all* data in your database.

7.4 Changing Existing Tables

Sometimes you'll need to update database tables that already exist.

First, you'll need to create a new migration. Say we wanted to add users to our `articles` table to keep track of who wrote what article. We'd use:

```
aristan make:migration add_users_to_articles_table --table=articles
```

It's important to give your migration a descriptive name (using snake-case). Notice that you can use the `--table` option to specify a table: this will automatically generate much of the migration code for us.

You can write the migration much the same as before, except anything you do will be working with the existing table.

For example, to add a `user_id` column to `articles`:

```
public function up()
{
    Schema::table("articles", function (Blueprint $table) {
        // add a new column
        $table->foreignId("user_id");
    });
}
```

Be careful to update the `down()` method to undo *everything* that you write in the `up()` method:

```
public function down()
{
    Schema::table("articles", function (Blueprint $table) {
        // drop the column added in up()
        $table->dropColumn("user_id");
    });
}
```

Changing Existing Migration Files

If your database doesn't contain any important data and you've not yet shared the migration with anyone else then sometimes it can be easier to go back and edit your original migration file and use `artisan migrate:refresh`.

If you're working on the project with someone else and you've already shared the migration with them you shouldn't do this unless you've agreed in advance that it's the best thing to do. In this instance, make sure everyone runs `artisan migrate:refresh`, otherwise your database structures will get out of sync.

7.4.1 Changing Existing Columns

To change existing *columns* of a table (as opposed to adding or removing columns of an existing table), you'll need to install an additional package:

```
composer require doctrine/dbal
```

You can now use the `change()` method:

```
Schema::table('articles', function (Blueprint $table) {
    $table->string("title", 50)->change();
});
```

7.5 Additional Resources

- Database Migrations

Chapter 8

Eloquent ORM

Now we've got a database structure in place, we want to be able to access the database from Laravel. 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 and opens up a plethora of security concerns. But it 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. SQLite can be orders of magnitude faster than MySQL for data that doesn't need to persist – which, as we'll see later, is very useful for testing code.

That's where **Object Relational Mapper**s (ORM) come in. An ORM lets you work with plain old PHP objects,[1] but under the hood it's actually writing and running database queries for you. Laravel's ORM library is called **Eloquent**.[2]

8.1 Models

Models are PHP classes that extend Eloquent's `Model` class. By default they live in the `App` namespace. Each model represents a different type of thing that your app works with. It usually maps to a table in the database.

For example, when we ran `artisan make:model` earlier, we created an `Article` model class. Have a look in `app/Article.php`:

[1] Like the one's you spent all of last week writing and using
[2] Good name.

```
namespace App;

use Illuminate\Database\Eloquent\Model;

class Article extends Model
{
    //
}
```

[View code on GitHub]

All the work is being done by that `extends Model` bit. It's Eloquent that is going to do almost all the work for us.

8.2 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->content = "Today I went shopping. I bought some spoons.";
$article->save();
```

[View code on GitHub]

We create a new instance of the `Article` class and then set properties that match the column names on our `articles` database table. Then we just need to call the inherited `save()` method.

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 does all of the databasey stuff for us. We don't need to worry about writing MySQL 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]

8.3 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`[3] back with all the articles in it:

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

Each row in the database is returned as an instance of the `Article` class.

You can also run `Article::find()` to find an article with a specific `id`:

[3] Remember these from last week?

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

This will return an instance of the `Article` class with the properties set to the corresponding values from the database row (or `null` if it doesn't find anything).

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

8.4 Model Methods

We can add our own methods to models in order to make them more useful. For example, we might want a way to get a truncated version of each article's `content` to display on a summary page.

We could add this to the `Article` model as follows:

```
use Illuminate\Support\Str;

class Article extends Model
{
    public function truncate()
    {
        // use the Laravel Str::limit method
        // to limit to 20 characters
        return Str::limit($this->content, 20);
    }
}
```

We've added a method called `truncate` to the `Article` class. The `$this->content` property will be set to the `content` value for whichever article the object instance represents. Now all instances of that class will have a `truncate` method.

```
// get the article with ID 1
$article = Article::find(1);

// call the truncate method we've added to the class
$article->truncate(); // truncated version of the content
```

Another useful method that we might add to our `Article` model is one that tells us, in human-readable form, how long ago a blog-post was posted:

```
public function relativeDate()
{
    return $this->created_at->diffForHumans();
}
```

We get the `created_at` property, which is automatically created for us by Laravel. This property is also automatically turned into a `Carbon` date object, so we can just call the Carbon methods on it directly, such as `diffForHumans()`.

```
// get the article with ID 1
$article = Article::find(1);

// call the relativeDate method we've added to the class
$article->relativeDate(); // something like "1 day ago"
```

8.5 Additional Resources

- Eloquent

Chapter 9

Blade

Laravel has its own **templating language** called "Blade".

A templating language lets you easily combine the data in your app with some other format to create a final document. In the case of Blade, it lets us easily generate the final HTML output of our app, making it easy to:

- Reuse repeated markup
- Display data
- Conditionally show content
- Loop over Collections

Blade templates have a `.blade.php` extension and they live inside the `resources/views` directory.

If you have a look at `resources/views/welcome.blade.php` you'll see the initial "Larvel" page that you get whenever you first create a Laravel project.[1] We're going to be gradually changing this file show a list of articles from our database.

First, let's strip it back to basics:

```
<!doctype html>
<html class="no-js" lang="en">
  <head>
    <meta charset="utf-8" />
```

[1] In the next section we'll look at **routing** which allows us to change which view we get

```
    <title>My Amazing Blog</title>
  </head>
  <body>
    <div class="container">
      <nav>
        <a href="/">My Amazing Blog</a>
      </nav>

      <main>
        <a href="#">
          <h5>Tea</h5>
          <small>1 Day Ago</small>
          <p>These teas are the bees knees</p>
        </a>
      </main>
  </body>
</html>
```

[View code on GitHub]

Now reload the site.

We've got a title and a some filler text for a blog post. And it looks like a bag of arse. Bootstrap to the rescue. In your `<head>`:

```
<link
  rel="stylesheet"
  href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.min.css"
  integrity="sha384-ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQUOhcWr7x9JvoRxT2MZw1T"
  crossorigin="anonymous"
/>
```

And your `<body>`:

```
<div class="container">
  <nav class="mt-4 navbar navbar-light bg-light">
    <a class="navbar-brand" href="/">My Amazing Blog</a>
  </nav>

  <main class="mt-4">
    <div class="list-group">
      <a href="#" class="list-group-item list-group-item-action">
        <div class="d-flex w-100 justify-content-between">
```

```
          <h5 class="mb-1">Tea</h5>
          <small>1 Day Ago</small>
        </div>
        <p class="mb-1">These teas are the bees knees</p>
      </a>
    </div>
  </main>
</div>
```

[View code on GitHub]

What a lot of `class`es! But it looks much nicer if you load it in the browser.

9.1 Extending

You'll probably notice that some of the material in our template is fairly generic markup that we'd want to use on every page of our site (e.g. the `<head>` and `<nav>`), whereas some of it would only be relevant for the homepage (e.g. the blog post list).

Blade lets us separate these two different aspects. We'll create a file called `app.blade.php` for the markup that's the same on every page:

```
<!doctype html>
<html class="no-js" lang="en">
  <head>
    <meta charset="utf-8" />
    <title>My Amazing Blog</title>

    <link rel="stylesheet" ...etc. />
  </head>
  <body>
    <div class="container">
      <nav class="mt-4 navbar navbar-light bg-light">
        <a class="navbar-brand" href="/">My Amazing Blog</a>
      </nav>

      <main class="mt-4">
          @yield("content")
      </main>
    </div>
  </body>
</html>
```

[View code on GitHub]

You'll notice that we `@yield("content")`[2] where the page specific content is going to go.

Next we need to **extend** this template in `welcome.blade.php`:

```
@extends("app")

@section("content")
  <div class="list-group">
    <a href="#" class="list-group-item list-group-item-action">
      <div class="d-flex w-100 justify-content-between">
        <h5 class="mb-1">Tea</h5>
        <small>1 Day Ago</small>
      </div>
      <p class="mb-1">These teas are the bees knees</p>
    </a>
  </div>
@endsection
```

[View code on GitHub]

There are two key things:

- `@extends`: we are extending the `app.blade.php` file (we don't need to include the file extension)
- `@section("content")`: we have created a named section called "content", which the `app` template `yield`s

Now, reload the site and everything should be as before. But we've modularised our code to make it much more reusable.

9.2 Partials

Sometimes it can be useful to split up your templates further. For example, as our app gets more complicated the `app.blade.php` file might get quite large. We

[2] These `@` commands are known as "Blade Directives"

could break it up into **partials**: small bits of template that we can include from another file.

Let's create a file `resources/views/partials/nav.blade.php` and move our navigation markup into it:

```
<nav class="mt-4 navbar navbar-light bg-light">
  <a class="navbar-brand" href="/">My Amazing Blog</a>
</nav>
```

[View code on GitHub]

Now, in `app.blade.php` we can use this file:

```
<div class="container">
    @include("partials/nav")

  <main class="mt-4">
```

You'll notice that we're not extending or using sections in the partial. That's because a partial is always `@include`d from another file.

9.3 Loops & Moustaches

It's often useful to repeat a certain bit of markup multiple times. Let's use a loop to update our articles list to show multiple articles.

First, make sure you've got multiple articles added to your database. Use `artisan tinker` to add some more if you've not got many.

Now update `welcome.blade.php` with the following:

```
@extends("app")

@section("content")
  <div class="list-group">

    { /* loop over all of the articles */ }
    { /* each article object goes into $article */ }
```

```
    @foreach (App\Article::all() as $article)
      <a href="/articles/{{ $article->id }}" class="list-group-item
      ↪  list-group-item-action">
        <div class="d-flex w-100 justify-content-between">

          { /* output the article title */ }
          <h5 class="mb-1">{{ $article->title }}</h5>

          { /* use the relativeDate() method */ }
          <small>{{ $article->relativeDate() }}</small>
        </div>

        { /* output the truncated content */ }
        <p class="mb-1">{{ $article->truncate() }}</p>
      </a>
    @endforeach
  </div>
@endsection
```

[View code on GitHub]

We use the `@foreach` directive to loop over all of our articles. Each one gets passed in as `$article`. We can then use **moustaches** (double curly-braces) to interpolate PHP values into our template.

9.4 Passing Data

Let's put the code for a single list item into its own partial.

In `resources/views/partials/articles/list-item.blade.php`:

```
<a href="/articles/{{ $article->id }}" class="list-group-item
↪  list-group-item-action">
  <div class="d-flex w-100 justify-content-between">
    <h5 class="mb-1">{{ $article->title }}</h5>
    <small>{{ $article->relativeDate() }}</small>
  </div>
  <p class="mb-1">{{ $article->truncate() }}</p>
</a>
```

[View code on GitHub]

Now we can include this from `welcome.blade.php` and pass in the `$article`:

```
@foreach (App\Article::all() as $article)
  {{ /* pass-through $article as "article" */ }}
  @include("partials/articles/list-item", ["article" => $article])
@endforeach
```

We can pass an associative-array as the second argument to `@include` and that will then be available inside the partial with the variable name that we gave it (`"article"` in this case).

9.5 Additional Resources

- Laravel: Blade Templates

Chapter 10

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 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. If Laravel finds a match it then runs the code that the route points to.

If you have a look in `routes/web.php`, you'll see that it already contains a route:

```
<?php

use Illuminate\Support\Facades\Route;

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

We use a static method of the `Route` class to say that if the user visits `/` (the **root**[1] URL – e.g. the homepage of our site) we should return a **view** called "welcome". This is why, if you visit your Laravel site, you'll see the `welcome.blade.php` template. But if you go to any other URL you'll get a 404.

Let's add another route for an "About" page:

[1] Yes, it's the root route

```
Route::get('/about', function () {
    return view('about');
});
```

Next, create a template `resources/views/about.blade.php`:

```
@extends("app")

@section("content")
  <div class="card">
    <h2 class="card-header">About</h2>
    <article class="card-body">
      <p>This is my amazing blog.</p>
      <p>It's amazing.</p>
    </article>
  </div>
@endsection
```

[View code on GitHub]

Now, visit `/about` if your browser and you should see the new page.

10.1 URL Parameters

Next, let's add a route for an article.

First create the file `resources/views/article.blade.php`:

```
@extends("app")

@section("content")
  <div class="card">
    <h2 class="card-header">Unflattering</h2>
    <article class="card-body">
      <p>This wombat's hat makes him look fat</p>
    </article>
  </div>
@endsection
```

[View code on GitHub]

And then add another route to `routes/web.php`:

```
Route::get('/articles/{article}', function () {
    return view('article');
});
```

Now if you visit the URL `/articles/1` in your browser, you should see the article view that you just created. It will also work if you go to `articles/2`. In fact it will work for `articles/anything`. That's because the `{article}` bit is a **URL parameter**: a placeholder for *any* value. We'll look at how we can work with this value later.

10.2 Additional Resources

- Laravel: Routing

Chapter 11

Controllers

As our app gets more complicated, we don't want to have to put all of the logic that works out what to show to the user in the routes file: it would quickly become unmanageable.

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.

First, we'll need a class. Run the following inside your Vagrant box to create an Articles controller:

```
artisan make:controller Articles
```

This will create a file (`app/Http/Controllers/Articles.php`), which contains the boilerplate for a controller.

11.1 Controller Methods

You can define your own methods in your controller for handling different requests.

First, let's update our routes to use the `Articles` controller:

```
Route::get('/', "Articles@index");
```

This is telling Laravel to run the `index` method of the `Articles` controller when the user visits the root route (`/`).

If you try loading the site now you'll get a server error. We'll need to add the method to `Articles`:

```
class Articles extends Controller
{
    public function index()
    {
        return view("welcome");
    }
}
```

This might just seem like we've made everything much more complicated for no obvious reason. But there are many advantages, one of which is that we can pass in data to the view much more easily.

First, tell the `Articles` controller where to find your `Article` model class:

```
use App\Article;
```

Then update your `index` method:

```
public function index()
{
    return view("welcome", [
        // pass in all the articles
        "articles" => Article::all(),
    ]);
}
```

Now we can update our `welcome.blade.php` template so that we don't have to fetch the articles in the template itself:

```
@foreach ($articles as $article)
```

Now we can potentially reuse the `welcome.blade.php` template for *any* page that needs to show a list of articles, not just the homepage. So we should probably rename the file `articles.blade.php` and update the `Articles` controller:

```
public function index()
{
    return view("articles", [
        // pass in all the articles
        "articles" => Article::all(),
    ]);
}
```

Next, let's get our page to show a specific article working properly (i.e. show the relevant article, rather than just some filler text).
The data (properties) from the Article model are passed into the view with the variable name `article`.

First update the route to use a method instead of putting all the code in the routes file:

```
Route::get("/articles/{article}", "Articles@show");
```

Next, add the `show` method to the `Articles` controller:

```
public function show($id)
{
    $article = Article::find($id);

    return view("articles/show", [
        "article" => $article
    ]);
}
```

The value from the URL parameter (the `{article}` bit in the route), gets passed in as the first argument to the method. So we can use this to find the relevant article and then pass it in.

Now we can update our `article.blade.php` template:

```
@extends("app")

@section("content")
  <div class="card">
    <h2 class="card-header">{{ $article->title }}</h2>
    <article class="card-body">
      {{ $article->content }}
    </article>
  </div>
@endsection
```

Now, if we visit `/articles/1` (assuming you have an article with the ID `1`), you should see the actual article text.

However, if you visit `/articles/384938439`, you'll get an error (unless you got really carried away adding articles). That's because currently we don't check to see if we actually get an `Article` back from our call to `Article::find()`. So, sometimes, we get back `null`, that gets passed into the template, and then things go very wrong.

Luckily, Laravel offers an elegant solution to this issue: **Route Model Binding**.

11.1.1 Route Model Binding

If, instead of just having the `$id` parameter, we replace it with `Article $article` Laravel is going to do some super-sneaky witchcraft:

```
public function show(Article $article)
{
    return view("articles/show", [
        "article" => $article
    ]);
}
```

In standard PHP code, this wouldn't have any right to work, but in Laravel it does. That's because Laravel uses **reflection**: that's where rather than just running your code, it actually *reads* your code and alters its behaviour accordingly. This is the sort of trick that should not be used too often: it takes more processing power and it's almost magical – and magical code is almost impossible to understand.

So what's it actually doing? It reads your code, sees that you've asked for an `Article` and notices that it's in the place where you'd normally accept the value from the URL parameter. It then joins this up and, instead of just passing in the parameter value as a string, looks for an `Article` with a matching ID. If it finds one then it passes it in (so that your type declaration is still valid). If it doesn't then it doesn't even bother trying to run your controller method and just displays a 404 page instead.

One thing to note, it's really important that the `{article}` bit in the route's URL parameter matches the model name, otherwise Route Model Binding won't be enabled.

View All Routes

A useful command to see all routes setup, the HTTP methods for them, which controllers and which methods will be run:

```
artisan route:list
```

11.2 Additional Resources

- Laravel: Controllers
- How to Structure Routes in Large Laravel Apps

Chapter 12

Forms

If we want to submit data to the site we'll need a form.

Let's create a simple form for creating a new blog post.

In `resources/views/form.blade.php`:

```
@extends("app")

@section("content")
    <form class="form card">
        <h2 class="card-header">Create Article</h2>

        <fieldset class="card-body">
            <div class="form-group">
                <label for="title">Title</label>
                <input id="title" name="title" class="form-control" />
            </div>

            <div class="form-group">
                <label for="content">Content</label>
                <textarea id="content" name="content"
                ↪  class="form-control"></textarea>
            </div>
        </fieldset>

        <div class="card-footer text-right">
            <button class="btn btn-success">Create</button>
        </div>
    </form>
@endsection
```

[View code on GitHub]

And add a route so that we can view it:

```
Route::group(["prefix" => "articles"], function () {
    // add *above* route with URL parameter
    // otherwise 'create' will get included in that
    Route::get('create', "Articles@create");
    Route::get('{article}', "Articles@show");
});
```

Then in your `Articles` controller:

```
public function create()
{
    return view("form");
}
```

Now visit `/articles/create` and you should see the form we just created.

12.1 Form `method`

If you add some data and then try to submit the form you'll notice that the page refreshes and the URL now includes a lot of extra gumpf after a `?` character.

```
/articles/create?title=Tea&content=These+teas+are+the+bees+knees
```

This is known as a `query string` and you'll see that it contains the data that you've just submitted.

When data is submitted in this way we call it a `GET` request: the data is submitted as part of the URL. This is very useful for things like search forms, where each search query creates a different URL. However, it's no good for submitting lots of data as URLs have a limited length. It's also no good for submitting sensitive data, as all of the data in the URL gets added to your browser history, which has various security implications.

That's where `POST` comes in. If we tell a browser to submit a form using `POST` it will submit the data as the **body** of the request. We'll look at exactly what this means later in the course, but for now all you need to know is that that data gets sent to the server, but it doesn't get added to the URL.

Method	How it submits data	Example usage
`GET`	Data is added to the URL	A search form
`POST`	Data in the body of the request	Submitting complex/sensitive data

By default a `<form>` will submit data as a `GET` request. To get it to use `POST` we'll need to add the `method` attribute to our template:

```
<form class="form" method="post">
```

Form `action`

When we submit data to a form the browser needs to submit the data *somewhere*. In a Laravel app it is standard to submit data to the same URL as the page, we can do this because rendering the page is a `GET` request, whereas submitting the form is usually a `POST` request. We do not need to specify an `action` attribute in this case.

12.2 `POST` Requests

Now that we're `POST`ing our data, we need to do something with it. This is where everything we've learnt so far finally comes together!

First, we need to create a `POST` route in the `articles` block:

```
Route::group(["prefix" => "articles"], function () {
    Route::get('create', "Articles@create");

    // a *post* request
    // needs to go to a different controller method
    Route::post('create', "Articles@createPost");

    Route::get('{article}', "Articles@show");
});
```

Next, we need to create the `createPost` method in our `Articles` controller:

```
public function createPost()
{
}
```

Now, if you try submitting your form you'll get a rather cryptic[1] "419 | Page Expired" page.

This is our first (of many) security blocks that Laravel puts in place for us.

12.2.1 Cross-Site Request Forgery

By default Laravel won't let you do things that might cause security problems.

Imagine that you go to the Blackwells website to buy a book. You log in, add some items to your basket, buy some books and then go to another website having not logged out of Blackwells. A wee while later you visit some super-dodgy website. Because you've got a cookie for Blackwells, the dodgy site could now make a `POST` request to Blackwells and it's going to think that it's you making the request. This is a **Cross-Site Request Forgery**.

Laravel doesn't want you to accidentally open your site up to this sort of attack. As such it won't let you make a form request unless the form has a hidden CSRF **token**. This is a randomly generated string of nonsense that Laravel knows about, but which another site trying to make a request to your site couldn't possibly guess. Laravel can then check this value whenever a request is made: if it's correct then everything works as it should, if it's not then you get the `419` error we got earlier.

To add a CSRF token we simple use the `@csrf` Blade shortcut somewhere in our form:

```
<form method="post" class="form card">
    @csrf
```

Now if we submit the form you'll refresh the current page. We need to update our `createPost` method to do something.

[1] Yet beautiful

12.2.2 Handling Data

Update the `createPost` method to the following:

```
// accept the Request object
// this gives us access to the submitted data
public function createPost(Request $request)
{
    // get all of the submitted data
    $data = $request->all();

    // create a new article, passing in the submitted data
    $article = Article::create($data);

    // redirect the browser to the new article
    return redirect("/articles/{$article->id}");
}
```

As you can see, first we need to accept the `Request` object: this represents the request that the user made, including the data they submitted. If we ask Laravel for this it will give it to us using similar witchcraft to Route Model Binding.

Then we get all the submitted data from the `Request` object. Next, we create an `Article` using that data, which will add an article to the database with the data that the user submitted. Finally we **redirect** the browser to URL of the newly created article.

View or Redirect?

As a general rule of thumb, for a Blade based site:

Request Method	Return Type
`GET`	`view()`
`POST`	`redirect()`

If a `POST` request returns a `view()` the user can refresh the page, which will resubmit any data, possibly leading to data duplication (or worse). If instead it returns a `redirect()`, when the user refreshes the page it will reload the page you've redirected to.

But, if you try to submit the form, you'll get an error.

12.2.3 Mass Assignment Vulnerability

We now come to our second common security issue. Accepting whatever the user gives you and sending it straight to the database is one of those cases: known as a **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 request directly (either by editing the from in Developer Tools or with a tool like Postman or `curl`). They add an `admin = true` property to the data 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. This is a bad thing.

To stop this from happening, if you pass an Eloquent model an array of properties, it will only use 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
{
    protected $fillable = ["title", "content"];

    // ... other Article code
}
```

Now try submitting the form. Hopefully it should submit and you get redirected to the new article.

12.3 Additional Resources

- Cross Site Request Forgery (CSRF)
- Mass Assignment Vulnerability

Chapter 13

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.

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.

13.1 Form Requests

As with most things 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 add support for user logins, 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 `name` of the submitted datum 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 submitted value
  public function rules()
  {
    return [
      "title" => ["required", "string", "max:100"],
      "content" => ["required", "string"],
    ];
  }
}
```

[View code on GitHub]

Next 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 `createPost` method to use the `ArticleRequest` class instead of `Request`:

```
public function createPost(ArticleRequest $request)
{
  // ... create post code
}
```

[View code on GitHub]

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

Now, if the validation passes everything will work as before, but if it *fails* the user will be sent back to the form. However, currently they won't know what they did wrong and all of their submitted data will get lost!

13.2 Updating Templates

13.2.1 Old Input

First, let's update our template so that the data the user submitted doesn't get lost if there's a validation error. To do this we can use the `old()` helper function.

To the `title` `<input>`:

```
<input
    id="title"
    name="title"
    class="form-control"
    value="{{ old('title') }}"
/>
```

To the `content` `<textarea>`:

```
<textarea
    id="content"
    name="content"
    class="form-control"
>{{
    old('content')
}}</textarea>
```

With a `textarea`, make sure you only put spaces/line-breaks *inside* the template curly braces, otherwise they'll get added to the content of the `<textarea>`.

13.2.2 Errors

Next, let's get error messages and styling working.

At the bottom of the `title` `<fieldset>`:

```
@error('title')
    <p class="invalid-feedback">
        {{ $message }}
    </p>
@enderror
```

If there are any, this will output an error message for `title` with appropriate Bootstrap classes. Don't forget to add one for `content` too.

We can also add relevant styling to our form fields:

```
class="form-control @error('title') is-invalid @enderror"
```

This will add the Bootstrap `is-invalid` class *if* the `title` field has an error.

13.3 Additional Resources

- Laravel: Validation
- Laravel: Working with Error Messages
- Bootstrap Form Validation Classes

Chapter 14

One-to-Many Relationships

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.

[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!

14.1 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 represent 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.

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

- **One-to-Many**: when some[5] 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 ex-

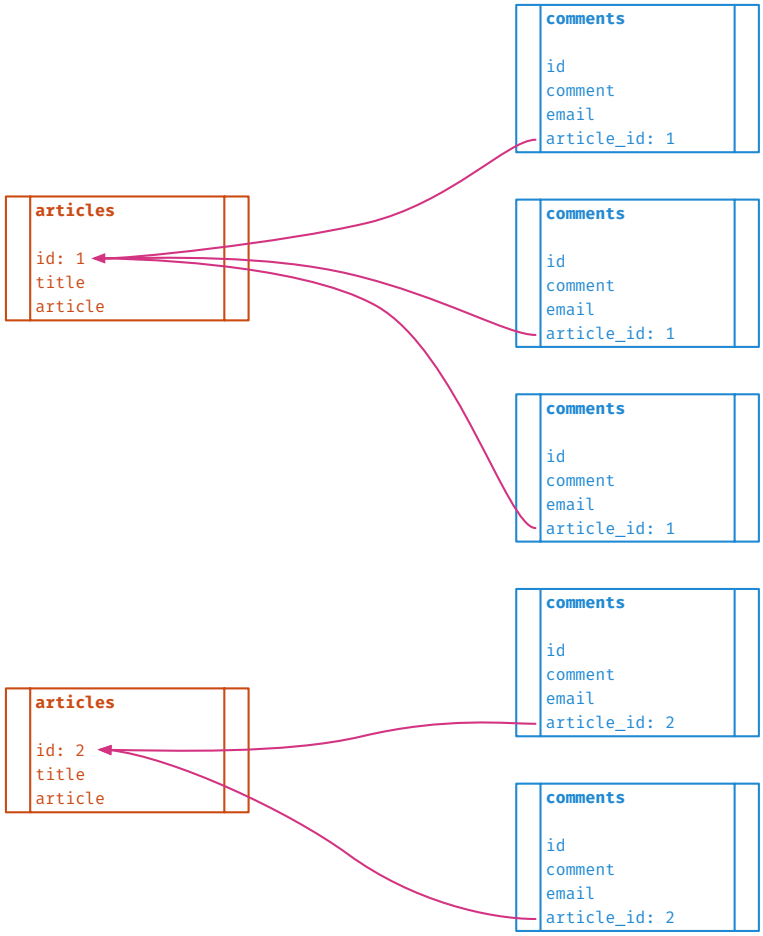
[5] Zero or more

press 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.

14.2 One-to-Many Relationships

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

14.2.1 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->id();
```

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

    // create the article_id column
    $table->foreignId("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.

14.2.2 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.[6]

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
```

[6] 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]

14.3 Adding Comments

First, we'll need a form. Create the following in `resources/views/comments/form.blade.php`:

```
<form method="post" class="form card mt-4 mb-4">
  <h4 class="card-header">Add Comment</h4>
  <fieldset class="card-body">
    @csrf

    <div class="form-group">
      <label for="email">Email</label>
      <input
        id="email"
        name="email"
        type="email"
        class="form-control @error('email') is-invalid @enderror"
        value="{{ old("email", $article ? $article->email : "") }}"
      />

      @error('email')
        <p class="invalid-feedback">
          {{ $message }}
        </p>
      @enderror
    </div>

    <div class="form-group">
      <label for="comment">Comment</label>
      <textarea
        id="comment"
        name="comment"
        class="form-control @error('comment') is-invalid @enderror"
      >{{
        old("comment", $article ? $article->comment : "")
      }}</textarea>
```

```
      @error('comment')
        <p class="invalid-feedback">
          {{ $message }}
        </p>
      @enderror
    </div>

  </fieldset>

  <div class="card-footer text-right">
    <button class="btn btn-success">Add Comment</button>
  </div>
</form>
```

[View code on GitHub]

You'll notice we've put all of the CSRF and validation logic in there already.

Next, include this in the article template:

```
<article class="card">
    {{ /* article code */ }}
</article>

<hr />

<h3>Comments</h3>

@include("comments/form")
```

When we submit this form it's going to `POST` to `/articles/article`, so next we'll need to setup a route:

```
// existing show route
Route::get('{article}', "Articles@show");

// add the *post* route below
Route::post('{article}', "Articles@commentPost");
```

Now we'll need to update the `Articles` controller. First, let the controller know about the `Comment` model:

```
use App\Comment;
```

Next, let's add the controller method:

```
// 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 commentPost(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 redirect("/articles/{$article->id}");
}
```

[View code on GitHub]

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. Then we store it in the database using the article model.

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]

14.3.1 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 `Articles` 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 commentPost(CommentRequest $request, Article $article)
{
  // ...store code
}
```

[View code on GitHub]

14.3.2 Displaying the Comments

Finally, let's update our article page to display the comments:

```
<article class="card">
  {{ /* article code */ }}
</article>

<hr />

<h3>Comments</h3>

{{ /* if an article has comments list them */ }}
@if($article->comments->isNotEmpty())
  <div class="list-group">
    @foreach ($comments as $comment)
      <div class="d-flex w-100 justify-content-between">
        <h5 class="mb-1">{{ $comment->email }}</h5>
      </div>

      <p class="mb-1">{{ $comment->comment }}</p>
    @endforeach
  </div>
@else
  <p class="alert alert-secondary">No comments found</p>
@endif

@include("articles/comments/form")
```

[View code on GitHub]

14.4 Additional Resources

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

Chapter 15

Unit Testing

Unit Testing is when we test small parts of our code to check that it works as we expect. This can save us having to dump bits of code to see what's going on.

Because unit tests are bits of code, we can run them as often as we like. This is much more efficient than manually testing/re-testing your code.

Laravel makes it very easy to unit test code. All our unit tests live in the `tests/Unit` directory. You can see an example test called `ExampleTest.php` already in there.

15.1 Running Tests

To run all of your unit tests run the following:

```
vendor/bin/phpunit --testsuite Unit
```

You should get a message saying:

```
OK (1 test, 1 assertion)
```

If you don't include the `--testsuite Unit` bit, it will also run **Feature Tests**, which let you test broader functionality, but we're not interested in this at the moment.

You can now delete the `tests/Unit/ExampleTest.php` file, as we'll create our own tests next.

15.2 Generating Tests

Laravel can automatically generate unit tests for us:

```
artisan make:test ArticleTest --unit
```

There should now be a file called `ArticleTest` in the unit tests directory.

15.3 Writing Tests

Let's have a look at the test class:

```
namespace Tests\Unit;

use PHPUnit\Framework\TestCase;

class ArticleTest extends TestCase
{
    public function testExample()
    {
        $this->assertTrue(true);
    }
}
```

Test class names need to *end* with `Test` and to extend `TestCase` – this is a class provided by PHPUnit that does most of the heavy-lifting for us.

Test methods need to *start* with `test`. You can do whatever you need to inside a test method, but importantly you should always **assert** something.

15.3.1 Asserting

"Asserting" something just means saying "This should be the case". So if we assert that something is true, we're saying "This value should be true". An assertion can be correct, in which case the test passes, or incorrect, in which case the test fails.

PHPUnit provides a huge number of possible assertions and Laravel provides some of its own too, but for now we'll stick to some basics:

- `assertTrue`: assert that something is `true`

- `assertSame`: assert that two things are the same type and value

Technically speaking you only really need `assertTrue`, but the other methods provide useful shorthands.

Let's write some assertions about an `Article`:

```
public function testFillable()
{
    $article = new Article([
        "title" => "Hello",
        "content" => "Blah blah blah",
    ]);

    // title should be set, as it's in $fillable
    $this->assertTrue($article->title === "Hello");
}
```

We create a new `Article` and then assert it's true that it's `title` property is equal to `"Hello"`.

We could of course write this using `assertSame`:

```
$this->assertSame("Hello", $article->title);
```

Using `assertSame` is better, as if the test fails PHPUnit will tell you what it was expecting to happen. Notice that we put the **expected** value first and the **actual** value second, if it's the other way round the PHPUnit output will be backwards.

Make sure to re-run your tests every time you make any changes:

```
vendor/bin/phpunit --testsuite Unit
```

`Call to member function connection() on null`

If you're not doing anything involving databases this *shouldn't* be necessary, but if you are testing a model and that model has a property that represents a *date*, Eloquent needs to know what database type you are using. In order to do this we need to setup the Laravel app with all of its environment and configuration settings.

To do this we need to extend the *Laravel* `TestCase` class. Replace the `use PHPUnit\Framework\TestCase` line with:

```
use Tests\TestCase;
```

Ideally we wouldn't need to do this, as it slows down our tests.

We should also test that values not in `$fillable` are not added:

```
public function testFillable()
{
    $article = new Article([
        "title" => "Hello",
        "content" => "Blah blah blah",
        "danger" => "Aaaargh!",
    ]);

    // title should be set, as it's in $fillable
    $this->assertSame("Hello", $article->title);

    // danger shouldn't be set, as it's not in $fillable
    $this->assertSame(null, $article->danger);
}
```

Let's test our `truncate` method to our `Article` model by adding the following to `ArticleTest`:

```
public function testTruncate()
{
    $article = new Article([
```

```
        "title" => "Hello",
        "content" => "Blah blah blah",
    ]);

    // doesn't need truncating
    $this->assertSame("Blah blah blah", $article->truncate());

    $article = new Article([
        "title" => "Hello",
        "content" => "Blah blah blah blah blah blah blah",
    ]);

    // should be truncated
    $this->assertSame("Blah blah blah blah...", $article->truncate());
}
```

15.3.2 setUp

You'll notice that we use the same `Article` object in both tests, we could set this up in the `setUp` method, which is automatically run before each test:

```
private $article;

public function setUp() : void
{
    // make sure we call the parent's setUp() method
    parent::setUp();

    // setup the article
    $this->article = new Article([
        "title" => "Hello",
        "content" => "Blah blah blah",
    ]);
}

public function testTitle()
{
    // use the article *property*
    $this->assertSame("Hello", $this->article->title);
}

public function testTruncate()
{
```

```
    // use the article *property*
    $this->assertSame("Blah blah blah", $this->article->truncate());

    // ...rest of test
}
```

15.4 Testing with a Database

Sometimes we need to test things that involve the database. However, we don't want to pollute our normal database with test data. If we run the same test hundreds of times, we don't want to end up with the same test article hundreds of times. It's also harder to write tests if we don't know the state of the database before we start.

For this reason we usually use a *separate* database for testing. By default Laravel uses an **in-memory SQLite** database. You don't really need to worry about what that means as Laravel does all the work for us, but basically it's temporary and stored in memory (as opposed to on the hard-drive), so it's really fast.

Laravel automatically sets this up for us in the `phpunit.xml` file.

15.4.1 Using the Test Database in Tests

We need to make a few changes to our test before we can work with the database.

If you haven't already, you'll need to extend the *Laravel* `TestCase` class. Replace the `use PHPUnit\Framework\TestCase` line with:

```
use Tests\TestCase;
```

We can also tell Laravel to automatically refresh (clear and then migrate) the database for each test we run. This makes the database state more predictable.

Add the following `use` statement:

```
use Illuminate\Foundation\Testing\RefreshDatabase;
```

Then, as the very first line in your class add the following:

```
class ArticleTest extends TestCase
{
    use RefreshDatabase;

    // ...rest of class code
}
```

This is called a `trait`: it effectively adds a method to your class.

Now we can write tests involving the database.

15.4.2 Writing Database Tests

Now we can now write code that uses the database just as we normally would:

```
public function testDatabase()
{
    // add an article to the database
    Article::create([
        "title" => "Hello",
        "content" => "Blah blah blah",
    ]);

    // get the first article back from the database
    $articleFromDB = Article::all()->first();

    // check the titles match
    $this->assertSame("Hello", $articleFromDB->title);
}
```

Because the database is wiped before each test, we can be sure that the first article in the database is the one we just added.

The test above isn't particularly useful, but when it comes to testing our controllers it can be.

> **`Cannot add a NOT NULL column with default value NULL`**
>
> SQLite doesn't allow you to add new columns to an existing database table with-

out making them `nullable` or giving them a `default` value. If you get the error above when you run your tests, you'll need to update your migrations to either set a default (`->default($value)`) value or make the column nullable (`->nullable()`).

15.5 Testing Controllers

Technically, this is blurring the lines between Unit tests and Feature tests, but it's very useful to be able to check that our controllers are doing what we expect.

First, let's create a new test, putting it in an appropriate namespace:

```
artisan make:test Http\\Controllers\\ArticlesTest --unit
```

Then we'll edit the test file:

```
namespace Tests\Unit\Http\Controllers;

// setup to use database
use Illuminate\Foundation\Testing\RefreshDatabase;
use Tests\TestCase;

// use classes we'll need
use App\Article;

class ArticlesTest extends TestCase
{
    // the database migrations trait
    // ensures the database is cleared between tests
    use RefreshDatabase;

    // a test for the controller's createPost() method
    public function testCreatePost()
    {
        // use the "call" method to fake a
        // POST request to /articles/create
        // then get the original content
        $response = $this->call('POST', '/articles/create', [
            "title" => "foo",
            "content" => "bar",
        ]);
```

```
        // test to see if and article has been added to the database
        // the database is wiped each test, so should be the first one
        $article = Article::all()->first();
        $this->assertSame("foo", $article->title);
        $this->assertSame("bar", $article->content);
    }
}
```

First, we do the usual database test setup bits and pieces. Then in our test we make a fake `POST` call to `/articles/create`. Then we test to see if it was added to the database.

As always, don't forget to run your tests:

```
vendor/bin/phpunit --testsuite Unit
```

If all tests pass we can be sure that the `createPost` method is doing what we expect: taking the data from the request and storing it in the database.

15.5.1 When Controller Test Go Wrong

Controller tests might not always work and it can be hard to work out why.

If your fake request doesn't work for some reason (e.g. validation fails), then any assertions will also fail.

The easiest way to work out what's going on is to make a *real* request in the browser with the same data and see what happens. You should get an error which you can then work with.

15.6 Additional Resources

- PHPUnit
- Laravel: Testing - Getting Started
- Nyan Cat PHPUnit Result Printer

Chapter 16

Auth

Laravel makes it very easy to add user authentication to our app, including pre-built login, registration, and password reset forms. We can then authorise that user to do different things.

Authentication vs Authorisation

Authentication means checking that a user is who they say they are. This normally involves asking for a password that only that user could know.

Authorisation is allowing a user to do different things based on who they are. For example, editing a blog post that they wrote, but not one that someone else wrote.

The two together are referred to as "auth". It's very easy to get them muddled and people frequently do.

16.1 Auth Scaffolding

First add the UI package which will generate the forms for us:

```
composer require laravel/ui --dev
```

Then add the bootstrap Auth UI:

```
artisan ui bootstrap --auth
```

You'll need to update all the views in `resources/views/auth` to use `@extends('app')` (instead of `layouts.app`). You can then remove the `resources/views/layouts` directory.

The above process will have added the necessary routes to our app. But we probably want to disable the user registration route. So update `routes/web.php`:

```
Auth::routes(['register' => false]);
```

Now you can go to `/login` and see the login form. However, you won't be able to login just yet as you haven't created a `User`.

16.2 `User` Model

The `User` model already exists inside the `app` directory and the relevant migrations were there when you did you first database migrations, so the tables are already in place too.

The easiest way to create a new user is to use `artisan tinker`:

```
$user = new User();
$user->name = "Your Name"
$user->email = "your@email.horse"
$user->password = Hash::make("password")
$user->save()
```

Now if you visit `/login` you should be able to login to your app.

Hashing Passwords

It's really important not to store a user's password in plain text. If someone got their hands on the database – a "hacker" or just a malicious employee – they could see the password and then try it on other websites. Because most people reuse the same password on lots of sites, this works surprisingly well.

We could **encrypt** a password: that means obscuring it in some way that is reversible. We could then store then encrypted version and decrypt it when necessary. However, because this is reversible, you still have the potential for someone to decrypt the password and view it.

Hashing is the process of taking a string and turning it into a very-almost-certainly unique different string. For example "fish" might become:

```
64875fcccaac069fcb3e0e201e7d5b9166641608
```

As long as the hashing algorithm always gives the same output given the same input we don't need to ever reverse the process to see what the user's password is, we can just compare the hashed version in the database with a hashed version of what they typed into the login form.

There are lots of different ways to hash a string. Some older ones like MD5 and SHA1 have known vulnerabilities that make them insecure.

If you know the hashing algorithm used it's possible to build a **Rainbow Table**: a database of the hashes of all combinations of letters and digits up to a certain length. Then you can simply compare the hashed version to the results in the database and work out the password.

Salting was invented to get around the issue of Rainbow Tables. If you append a long random string (the "salt") to the beginning of the user's password and keep track of this, you can make sure the string being hashed is longer than a Rainbow Table could conceivably store (more than about 30 characters becomes unrealistic for storage/performance). Then to check the password you just concatenate the password and the salt and check the result.

However, even this has issues, as computers get faster over time and what's com-

putationally expensive today might be cheap and easy in a few years. Modern hashing techniques, such as bcrypt, get around this by hashing the password repeatedly for a given amount of time. As computers get faster, the number of times it gets hashed will go up and up, meaning that it's *always* computationally expensive to try and generate a Rainbow Table. Bcrypt also automatically salts the password and keeps track of this in the hash itself along with the specific hashing algorithm that was used and how many times it was hashed:

```
$2y$10$Ng9xo7lGx8OFciETjR38auk9BgGM98.UqLjrakvt9Vu2REL1KiOXG
 1  2  3                      4
```

1. `2y`: It's the fixed version of the blowfish hashing algorithm
2. `10`: It's been hashed 2^{10} times
3. `Ng9xo7lGx8OFciETjR38au`: The salt (128 bits – 22 characters)
4. `k9BgGM98.UqLjrakvt9Vu2REL1KiOXG`: The hash (184 bits – 31 characters)

In conclusion, use Laravel's `Hash` class and you'll be fine.

16.3 Authentication

Now that our app understands the concept of a logged-in user, we can do various things with the `Auth` class in our controllers.

First make sure to `use` it:

```
use Auth;
```

Then we can use various static methods:

```
Auth::user() // gives back the User model
Auth::id() // gives back the user's id
```

16.4 Authorisation

We can make sure that some of the parts of our app are only visible to logged-in users, by using the `auth` **middleware**:

```
Route::group(["prefix" => "articles"], function () {
    // put behind the auth middleware
    // need to be logged in to use
    Route::group(["middleware" => "auth"], function () {
        Route::get('create', "Articles@create");
        Route::post('create', "Articles@createPost");
        Route::get('{article}/edit', "Articles@edit");
        Route::post('{article}/edit', "Articles@editPost");
    });

    // don't need to be logged in to view or comment
    Route::get('{article}', "Articles@show");
    Route::post('{article}', "Articles@commentPost");
});
```

We wrap the routes that we want to protect using the `middleware` group option. Now if we try to visit any of these routes without being logged in we'll be taken to the login form.

We can use Gates and Policies to limit access even further – for example, only allowing a user to edit articles they've written – but this is beyond the scope of this chapter.

16.5 Under the Hood

Laravel does all of the hard-work for us, but it's useful to know a little bit about what's going on.

16.5.1 Cookies

It can be useful to keep track of a user on a website. For example, if they login to the website, we want to keep track of them between pages.

We can use a **cookie** to store a simple bit of text on the user's computer. This is sent back to the server every time that a request is made. The server can then use this to work out who the user is.

16.5.2 Sessions

To use cookies we need to create a **session**. Laravel does this for us automatically. A session ID is a randomised string that gets generated when a user first visits your website. This is then stored in a cookie and used to identify the user on future requests. It's important that the session ID be hard to guess, otherwise anyone could manually create a cookie and guess at sessions IDs hoping to trick the server into thinking that it was you.

More advanced session IDs take into account things like the user's IP address, browser, and other identifying features. If any of these suddenly change then that suggests someone else is trying to use the session ID and the session ID is revoked. Although this adds additional security, it's also quite CPU intensive, as such sessions tend to only be used for requests that are unlikely to be made in quick succession, such as a user browsing a website. For things like API requests, which can sometimes be made many times a second, alternative methods of authentication are required.

16.6 Additional Resources

- Laravel: Authentication
- Laravel: Authorisation
- Laravel: Gates and Policies
- Wikipedia: Cryptographic Hash Functions
- Wikipedia: Bcrypt
- Wikipedia: HTTP Cookies
- Wikipedia: Sessions

Glossary

- **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.
- **Foreign Key**: A relationship between two tables in MySQL that is enforced by the database.
- **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.
- **Relational Database Management Systems**: A database made up of tables (that represent one type of thing) and the relationships between items in those tables.
- **Server-Side**: Code that runs on a server. As opposed to client-side, which runs on the user's computer.
- **Stack**: A set of software that is commonly used together (e.g. the LEMP server stack)
- **Web Server**: A piece of software that "serves" files in response to a request over the internet.

Colophon

Created using TEX

Fonts

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

Written by Mark Wales & Oli Ward

∞

May 8, 2020