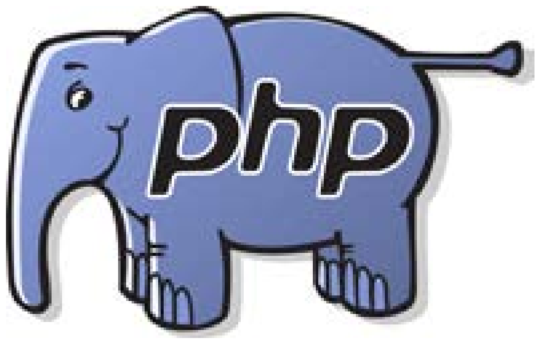
WEB 306 – PHP: Databases and Framework

Day 8 – Composer and Libraries

1. Virtual Hosts



1.1 About Virtual Hosts

In order to work with many large web development frameworks and projects locally, it is necessary for you to set up what is known as a "virtual host" on your computer. A virtual host lets you store a project anywhere on your computer and lets you create an alias for it which looks like a web address i.e. `project.local`, `coolsite.test`. This means that if you have a project with a long address you do not need to enter it to view your site. In my `C:/` drive, I have set up a folder called Sandbox which contains ALL of my web development projects and uses a virtual host called `sandbox.local`.

PHP Resources

1. PHP Website
2. XAMPP
3. Documentation
4. W3Schools
5. CSS Tricks
6. Scotch.io
7. Stack Overflow

1.2 Setting Up a Virtual Host

Creating a virtual host requires editing core files on your computer so you must have administrator priviledges when you open these files. There are two files which you need to edit to set up a virtual host using XAMPP. The first one is stored in `C:/xampp/apache/conf/extra` and is called `http-vhosts.conf`. You can open this file in VS Code or Atom like any other text file. Even though you already have access to the address `localhost` for viewing local files you MUST define it as one of your hosts using the following code before you define any new hosts.

```
<VirtualHost *:80>
  DocumentRoot "C:/xampp/htdocs"
  ServerName localhost
</VirtualHost>
```

The `DocumentRoot` setting points to the path your project is stored in on your computer. The `ServerName` setting is the address which you want to be able to access your project from in your browser. It is important that this address does not end in a real TLD (top-level domain name) like `.com` or `.net`.

Developers used to use the TLD `.dev` but this address has been registered as a real TLD by Google and they have begun to sell `.dev` domains so it is not recommended to use this as a fake TLD anymore. Instead it is recommended that you use either `.local` or `.test` which has been excluded from being eligible for being a real TLD.

The `ServerAlias` setting is the same address prefaced with `www.` like a web address. The path to your project must also be included in the opening `Directory` tag.

To add a new host, make sure that you have all of the following settings in your `httpd-vhosts.conf` file.

```
<VirtualHost *:80>
    DocumentRoot "C:/Sandbox/web-306-day-8-composer-libraries/public"
    ServerName elephants.local
    ServerAlias www.elephants.local
    <Directory "C:/Sandbox/web-306-day-8-composer-libraries/public">
      Options All
      Require all granted
      Options Indexes FollowSymLinks Includes execCGI
      AllowOverride All
      Order Allow,Deny
      Allow From All
    </Directory>
</VirtualHost>
```

The second file which you must edit is stored in `C:\Windows\System32\drivers\etc` and is simply called `hosts`. This is a Windows system file so to edit this file you must open Notepad ***with administrator priviledges*** and then go to **File > Open...**, navigate to `C:\Windows\System32\drivers\etc` and open the file called `hosts`. This file lists IP addresses and their associated virtual hosts on your computer. To add a new host enter a line which reflects the format of the following.

```
::1             elephants.local
```

I believe that it is easier to create virtual hosts using MAMP but can not speak to this from experience.

2. Namespaces

2.1 About Namespaces

Classes and objects are frequently used in third-party libraries because they allow code to be very modular due to encapsulation, intuitive due to abstraction, expandable due to inheritance and flexible due to polymorphism.

However, this presents a challenge because many libraries and projects need to use similar classes which could very easily have the same class names. For instance, there are likely many libraries with classes like `Database`, `Model` or `Controller`.

In the case of a Car class, there could be a database of vehicles which might need a class for a road Car and a rail Car and could then have a conflict. People used to deal with this by giving their class names a prefix to differentiate them like `Road_Car` and `Rail_Car` but this creates longer, more complex class names which are ugly and a bit of a pain to use repeatedly. If we use enough classes it can also be a pain to require them all separately using the correct file path whenever we need to use them. PHP introduced a concept called "namespaces" to address these issues. Namespaces are like file paths but aren't directly correlated with a file's path relative to a given file. They are used to give classes unique "virtual" addresses which can be used to access them, kind of like a domain name for a class.

2.2 Defining Namespaces

To define a namespace, use the namespace keyword, followed by the name of your class's namespace. Depending on how complex the needs of your project are, or the number of libraries you have, you might have one or more sub-namespaces as well.

To separate a sub-namespace from another namespace use the backslash. Even though namespaces don't directly mirror file paths, it is still necessary for our class to be stored in a folder structure which roughly mirrors our namespace. The namespace ***MUST*** be the first line in your class file.

```php
<?php
namespace Utilities;

class Sanitizer {
```

```php
<?php
namespace Utilities;

class Connector {
```

```php
<?php
namespace Models;

require '../utilities/Sanitizer.php';

class Elephant {
```

```php
<?php
namespace Controllers;

require '../utilities/Connector.php';
require '../models/Elephant.php';

class ElephantController {
```

2.3 Using Namespaces

Now that we have created our namespace, we can use it by using the **use** keyword followed by the name of our namespace name.

```php
<?php
namespace Models;

require '../utilities/Sanitizer.php';

```

```
use \Utilities\Sanitizer;

class Elephant {
```

You can alternatively simply use your namespace by putting it in front of your class name when you create a new object or reference the class.

```
public function set_name(string $value) {
   $this->name = \Utilities\Sanitizer::format_name($value);
}
```

In order for our namespace to work, we still need to require our class file though. You might be thinking that this is actually less convenient than it was before, and you would be right. That's why people do what is called "autoloading" and create a class which automatically loads all of their classes into their files when a new object of that class is created.

3. Autoloading

In order for our namespaces to work we need to create a class with a method which will ensure that whenever we create a new object our object's class will be automatically loaded into a given file so that we don't need to require it.

This class will be called `AutoLoader` and it will be a singleton so it will need to define a private static variable which is set to null.

```
class AutoLoader {
   private static $loader = null;
```

It will have a method called `class_loader()` and will take a paramter which will be our namespace and class name. We're going to take our namespace and our class name and convert them into a file path. Use the `str_replace()` function to replace the backslashes in our namespace and class name with forward slashes so that we can convert our namespace into a file path. Then concatenate `'.php'` on the end to complete the file name.

Use the `realpath()` function and use `__DIR__` concatenated with `'/../'` in order to get the absolute (full) path name of our current file and go up one directory level to the root. Then concatenate that with `$class_path`.

```
    public static function class_loader($namespace_class_name) {
        $class_path = str_replace('\\', '/', $namespace_class_name) .
            '.php';

        $full_class_path = realpath(__DIR__ . '/../') .  $class_path;
```

We will then check to see if the corresponding PHP file exists and if so, require `$full_class_path`.

```
        if (file_exists($full_class_path)) {
            require $full_class_path;
        }
    }
```

Because this is a singleton, we need a `get_loader()` method which creates a new instance of the class and returns it.

```
    public static function get_loader() {
        if (self::$loader == null) {
            self::$loader = new AutoLoader();
        }
```

However, unlike our previous singleton, in order to have this class automatically load our classes it is necessary to use the spl_autoload_register function before returning the instance. This function takes an array as the first parameter which holds the class name of an autoloading method we want to call, followed by the method name.

```
        spl_autoload_register(array('AutoLoader', 'class_loader'),
            true, false);
        return self::$loader;
    }
  }
```

The second and third parameters are boolean values. The first specifies whether the function should throw exceptions (errors) if the class_loader() function can not be registered and should be set to true. The second specifies that this autoloader method should be moved to the top of the queue if there are multiple autoloaders and should be set to false as we will want this autoloader to come after another autoloader we will be importing.

4. Index

In order for our AutoLoader class to work we need to require it into every page which needs access to our classes which kind of defeats the purpose of using an autoloader.

To avoid needing to require this file in every file we will require it in an index.php file which we wil place in a public directory which is located in the root directory of our app and we will use this file to import every page. Then to initialize the AutoLoader class we will call the get_loader() method. We will also be requiring a thid-party autoloader in this file which will allow us to use external libraries.

```
  <?php
  $root = realpath(__DIR__ . '/..');

  $composer_autoload = $root . '/vendor/autoload.php';
  $custom_autoloader = $root . '/utilities/AutoLoader.php';
  $routes = $root . '/controllers/routes.php';

  require $composer_autoload;
  require $custom_autoloader;

  AutoLoader::get_loader();

  require $routes;
```

To import every page in our application into this file we will require our routes.php file at the bottom of this file.

5. Routes

Because of some changes we are making to the way our app works we need to change some things in our routes file.

Instead of calling methods using an address with a `$_GET` key of "action" we will get the URI (the subdirectory which follows our virtual host name i.e. the `/add` in `www.elephants.local/add`). We can do this using `$_SERVER['REQUEST_URI']` and the `parse_url()` function.

First we need to require our `\Controllers\ElephantController` class and create a new instance of our Controller using its namespace. Then we will use a switch statement to check the value fo the URI key and call the corresponding method in reponse. Previously we only used our routes file to call the create, update and delete methods but we will also be using it to call `read()`, `add()` and `edit()` methods now.

```
  $controller = new \Controllers\ElephantController;

  $uri = parse_url($_SERVER['REQUEST_URI'], PHP_URL_PATH);

  switch ($uri) {
     case '/':
          echo $controller->read();
          break;
     case '/add':
          echo $controller->add();
          break;
     case '/edit':
          echo $controller->edit($_GET['id']);
          break;
     case '/create':
          $controller->create($_POST);
          break;
     case '/update':
          $controller->update($_POST);
          break;
     case '/delete':
          $controller->delete($_POST);
          break;
     default:
          echo $controller->read();
  }
```

If the key does not match any of the methods then the file will just redirect back to the "view" page.

6. .htaccess Files

In order to have the above addresses direct to the index.php file we need to use a file called `.htaccess` in our public directory to rewrite our paths. The `.htaccess` file is a configuration file which is used on Apache web servers to adjust settings such as redirecting URLs and protecting passwords. Below is the exact code used the the `.htaccess` file which the MVC framework, Laravel uses. Basically what it does is tell every address to redirect to the `index.php` file where it will load a page by calling a method based on the address.

```
<IfModule mod_rewrite.c>
    <IfModule mod_negotiation.c>
        Options -MultiViews -Indexes
    </IfModule>

    RewriteEngine On

    # Handle Authorization Header
    RewriteCond %{HTTP:Authorization}
.
    RewriteRule .* - [E=HTTP_AUTHORIZATION:%{HTTP:Authorization}]

    # Redirect Trailing Slashes If Not A Folder...
    RewriteCond %{REQUEST_FILENAME} !-d
    RewriteCond %{REQUEST_URI} (.+)/$
    RewriteRule ^ %1 [L,R=301]

    # Handle Front Controller...
    RewriteCond %{REQUEST_FILENAME}
!-d
    RewriteCond %{REQUEST_FILENAME}
!-f
    RewriteRule ^ index.php [L]
</IfModule>
```

VS Code Extension

Apache Conf

To add syntax highlighting for HTACCESS files in VS Code, install the Atom package ***Apache Conf***. To install Apache Conf click on the fifth icon in the left-hand menu or press **ctrl+shift+X** and search for `apache conf`.

Atom Package

Language htaccess

To add syntax highlighting for HTACCESS files in Atom, install the Atom package ***Language htaccess***. To install Language htaccess go to File > Settings, then Install and search for `language-htaccess`.

7. Intro to CLI

CLI stands for **Command Line Interface**. It is also known as the “terminal.” A CLI allows you to type commands to tell your computer to perform various actions.

7.1 About Git BASH

Unix, Linux and Mac operating systems use the same CLI commands. Windows uses its own unique CLI commands. Some of them are the same as the Linux commands but many of them are different. However, Git Bash allows us to use Linux terminal commands, which are more popular and commonly referenced in documentation, on Windows.

Git Bash is a CLI which is designed specifically to use Linux commands with Git on Windows. We will not be using the command line for Git because the GitHub GUI is much easier to use but we will be using Linux commands on Windows. If you are using a Mac then you do not need Git Bash.

You can download Git and Git Bash for Windows at gitforwindows.org.

7.3 Intro to Shell Script

The commands which are entered into a Linux terminal make up a programming language known as Shell Script. Shell Script allows us to enter commands which will do things such as list all files in the current directory, change directories, and make a new directory.

```
$ ls
$ cd /c/some_directory
$ mkdir some_new_directory
```

An `.sh` file is a Shell file which is used to run a collection of several Linux terminal commands at once. You can leave comments in `.sh` files to explain your code.

```
# This is a Shell comment
```

Like PHP, we can use the echo keyword to display text in the terminal.

```
echo "Intro to Shell"
```

In our `.sh` file we can run commands like `ls` to check what is in our directory.

```
ls
```

To run all of the commands listed in the `.sh` file use the `sh` command followed by the name of the `.sh` file.

```
$ sh intro-shell.sh
```

Shell Script Commands

There are many commands to learn but these are the primary ones we will be using:

1. **`echo "Hello"` — display text**
2. **`ls` — list files in current directory**
3. **`cd` — change directory**
4. **`cd /` — go to root directory**
5. **`cd ../` — go up one directory level**
6. **`cd some/name` — go to some/name**
7. **`mkdir some_name` — make directory**
8. **`sh file.sh` — run .sh file**
9. **`sudo` — execute as superuser (required for MacOS)**

For a more comprehensive list of commands go here.

7.4 Intro to Windows CMD and BATCH

CMD is the Windows CLI. Even though we can use Linux commands with Git Bash some of the time it's easier to use the Windows CMD. We can also use CMD commands from Git Bash by entering the command `cmd`.

A `.bat` file is a BATCH file which is for running a batch of CMD commands at once.

`@echo OFF` is used to stop terminal commands in our BAT file from being displayed in CMD when they are run.

```
@echo OFF
```

Windows/CMD Commands

There are many commands to learn but these are the primary ones we will be using:

1. **`echo.Hello` — display text**
2. **`@echo OFF` — don't display text**
3. **`echo ON` — do display text**
4. **`dir` — directory information**
5. **`cd` — change directory**
6. **`cd \` — go to root directory**
7. **`cd ..\` — go up one directory level**
8. **`cd directory\name` — go to directory/name**
9. **`mkdir` — make directory**
10. **`file.bat` — run .bat file**
11. **`exit` — end CMD session**

For a more comprehensive list of commands go here.

```
REM This is a BAT comment
REM stands for REMARK. Not the band!
:: This is another way of writing BAT comments
```

We can use `echo` without the `@` symbol and then followed by a . and a phrase to display it.

```
echo.Intro to CMD
```

In our `.bat` file we can run commands like `dir` to check what is in our directory.

```
dir

echo ON
```

To run all of the commands listed in the `.bat` file by entering the name of the `.bat` file into CMD.

```
$ intro-cmd.bat
```

8. Intro to Package Managers

A package manager or package management system is a system which automates the installation, updating and removal of software from a computer or server. In modern development, most major programming languages have one or more of their own package managers which are used to install, update and remove “packages” of software and libraries which are written in, used with or specific to that programming language.

Package managers are used through the CLI, `.shell` or `.bat` files. A specific set of commands are used for installing, updating or removing packages. These commands and the packages which are available would differ depending on which package manager is being used.

9. Intro to Composer

Composer is a package manager for PHP. You can write commands in the terminal (Git Bash) to download PHP packages using it. To download and install Composer on Windows, go to the Composer website, download the installation wizard, go through the steps and be sure to add Composer to the PATH variable of your computer. To check if Composer installed successfully, open Git Bash and simply enter the command `composer`. If Composer is installed then you should see this:

```
   ______
  / ____/___  ____ ___  ____  ____  ________  _____
 / /   / __ \/ __ `__ \/ __ \/ __ \/ ___/ _ \/ ___/
/ /___/ /_/ / / / / / / /_/ / /_/ (__  )  __/ /
\____/\____/_/ /_/ /_/ .___/\____/____/\___/_/
                    /_/
```

Whenever you enter the command `composer` with the command `require` to install a package all of the files will be downloaded into a folder in the root directory of you project called `vendor`. A JSON file called **composer.json** will also be created in your project's root directory which will list all of the required packages for your project.

```
{
    "require": {
        "php": "^7.2",
        "vlucas/phpdotenv": "^2.5",
        "eftec/bladeone": "^3.16",
        "phpdocumentor/phpdocumentor": "^2.9"
    }
}
```

10. Dotenv

We are going to use Composer to install a package called phpdotenv in our app's root directory. Navigate to the root directory of your app in Git Bash and then enter the these commands:

```
composer require vlucas/phpdotenv
```

Once you run this command a **vendor** folder will automatically be created in the root directory. This folder is for external libraries for PHP and should always go in the root directory.

VS Code Extension

DotENV

To add syntax highlighting for Dotenv files in VS Code, install the VS Code extension ***DotENV***. To install DotENV click on the fifth icon in the left-hand menu or press **ctrl+shift+X** and search for **dotenv**.

10.2 Creating an ENV File

Make a new file and save it as simply **.env**. This is an environment file where we can store sensitive data like usernames and passwords without putting them in the code so that we can work on database projects collaboratively and share our code publicly in places like GitHub without displaying our credentials. Enter your database login info here:

Atom Package

Language Dotenv

To add syntax highlighting for Dotenv files in Atom, install the Atom package ***Language Dotenv***. To install Language Dotenv go to **File > Settings**, then **Install** and search for **language-dotenv**.

```
DB_HOST="localhost"
DB_USER="root"
DB_PASSWORD=
DB_NAME="elephants"
```

When using XAMPP, our database host is localhost and unless we create a new database account in PHPMyAdmin, then our username is "root" and our password is blank by default.

10.3 Loading Database Credentials with Dotenv

We will import our database credentials into **Connector** class. In order to do so, we need to use the **\Dotenv** namespace to access the **Dotenv** class. Now that we are using autoloading it is also necessary for us to import PDO using the use keyword. We will also create a new private property to hold our Dotenv object.

```
namespace Utilities;

use \Dotenv\Dotenv;
use PDO;

class Connector {
  private static $instance = null;
  private $dotenv;
  private $host;
  private $name;
  private $user;
  private $password;
```

Once we have accessed the `Dotenv` class, we need to create a new instance of it and use the directory where our `.env` file is saved as the parameter, which in this case is our root. We will then load our credentials from the `.env` file using the `load()` method. Now that we have loaded our database credentials, we can save them as PHP variables using the `getenv()` function.

```
  private function __construct() {
      $this->dotenv = new Dotenv(__DIR__ . '/../');
      $this->dotenv->load();

      $this->host = getenv('DB_HOST');
      $this->name = getenv('DB_NAME');
      $this->user = getenv('DB_USER');
      $this->password = getenv('DB_PASSWORD');

      $pdo = new PDO(
          'mysql:host=' . $this->$host .
          ';dbname=' . $this->$name,
          $this->$user,
          $this->$password
      );
  }

  public static function get_instance() {
      if (self::$instance == null) {
          self::$instance = new Connector;
      }
      return self::$instance;
  }

  public static function get_connection() {
      return $this->instance;
  }
}
```

11. Blade Templates

11.1 About Blade Templates

Despite the fact that PHP was originally designed specifically to be an HTML templating language, some people do not like the default HTML templating syntax and have created enhanced templating languages for the PHP templating language. The MVC framework, Laravel uses their own templating language called Blade which makes creating HTML templates with PHP much easier.

11.2 BladeOne

Despite the fact that the Blade templating language was designed specifically to be used with Laravel, it has become so popular that there have been several adaptations of it which can be used outside of Laravel. BladeOne is a templating engine which was designed after Blade and which shares most of the same features and syntax. We will use BladeOne, with our existing pure PHP CRUD app to compare the difference between the regular PHP templating language and Blade templates.

To install BladeOne using Composer enter the following Composer command.

```
composer require eftec/bladeone
```

11.3 Base Blade Template

11.3.1 Creating Parent Templates

Unlike with regular PHP templating where we would normally create separate header and footer files, Blade templates allow you to create a “parent” template which can be applied to multiple child templates. We will create a parent template file called `base.blade.php` which we will save in a folder called `templates` which will be in the `views` folder. This file will start as a normal HTML page.

```
<!DOCTYPE html>
<html lang=”en”>
  <head>
      <meta charset=”UTF-8”>
```

VS Code Extension

Laravel-Blade

To add syntax highlighting for Blade template files in VS Code, install the VS Code extension ***laravel-blade***.To install laravel-blade click on the fifth icon in the left-hand menu or press **ctrl+shift+X** and search for `laravel-blade`.

You must save your Blade template files with the format `filename.blade.php`.

Some VS Code syntax themes may not support laravel-blade highlighting. To change your syntax theme go to go to **File > Preferences > Color Theme (Code > Preferences > Color Theme on macOS)**

Atom Package

Language Blade

To add syntax highlighting for Blade template files in Atom, install the Atom package ***Language Blade***. To install Language Blade go to **File > Settings**, then **Install** and search for `language-blade`.

You must save your Blade template files with the format `filename.blade.php`.

Some Atom syntax themes may not support Language Blade highlighting. To change your syntax theme go to **File > Settings**, then **Themes**.

```
        <meta name="viewport" content="width=device-width,
            initial-scale=1.0">
        <link rel="stylesheet" href="css/main.css">
    </head>
```

11.3.2 Using Variables and Functions in Templates

We can echo variables which we have passed to our Blade template using our Controller by simply wrapping them in double curly braces: `{{ some_variable }}`

```
    <body>
        <header>
            <h1>{{ $title }}</h1>
        </header>

        <footer>
            <p><small>Copyright {{ date('Y') }}
                <em>{{ $title }}<em><small></p>
        </footer>
    </body>
  </html>
```

Blade automatically does HTML escaping using `htmlspecialchars()` for us so it is not necessary for us to do escaping manually.

11.3.3 Including Partials

We can still simply include template files into our Blade templates using the `@include()` function. Template files which are included using Blade are known as "partials" and should be kept in a separate folder in the `views` folder called `partials`.

Use the directory of the partial file as a parameter of the function. Files in folders can be accessed using a dot between the folder and file name.

```
  @include('partials.navigation')
```

11.3.4 Yielding Sections

Blade provides a structure called sections. Sections can be created in parent templates using the `@yeild()` function which is used as placeholders for content which could be different for every page and then referenced in child tempaltes. Use a name for the section as the parameter of this function.

```
  @yield('content')
```

11.4 Blade Child Templates

11.4.1 Extending Parent Templates

Child templates use a parent template as a basis but overwrite any sections which were yielded in the parent template. We will create a child template called `add-elephant.blade.php` which will be saved in the main `views` folder. To specify a parent template to base a child template on, use the `@extends()` function with a string with the path to the parent template as a parameter.

```
@extends('templates.base')
```

11.4.2 Overwriting Content Blocks

In the child template you can use sections with the same names as any of the yields from the parent. Any content which is included inside of these sections will overwrite the yields in the parent template.

```
@extends('templates.base')

@section('content')
<div class="container">
   <h2><span class="fa fa-plus"></span> Add Elephant</a></h2>
</div>
@endsection
```

11.4.3 Conditionals

If, if else and if else if statements use a different syntax in Blade templates. They work the same way that they do normally except that they require an `@endif` keyword to close the statement.

```
@if (isset($_GET['add']) && $_GET['add'] == 'error')
<div class="error"><span class="fa fa-exclamation-triangle"></span>
Sorry, we need more information about this elephant!</div>
@endif
```

11.4.4 Add Form Action

The Add Elephant page will have a form on it with an action attribute which points to the address ***"/create"*** in order to call the `create()` method in our Routes file.

```
<form action="/create" method="post">
   <label for="name">Name:</label>
   <input type="text" name="name" value="" id="name">

   <label for="age">Age:</label>
   <input type="number" name="age" value="" id="age">
```

11.4.5 Loops

All loops can be used in Blade templates using a slightly different syntax. They work the same way that loops work in PHP except that they require an `@endforeach` (if you're using a foreach loop) keyword to close the loop. We could loop through the options in our select field.

```
        <select name=”image” id=”image”>
             @foreach ($image_files as $file_name)
             <option value=”{{ $file_name }}">{{ $file_name }}</option>
             @endforeach
        </select>

        <button type="submit"><span class="fa fa-plus"></span>
             Add Elephant</button>
   </form>
```

```
  </div>
  @endsection
```

11.4.6 Setting Values

In order to set values for variables in Blade templates it is necessary to use the `@set()` function.

```
  @foreach ($elephants as $elephant_num => $elephant_row)
    @set($elephant = $elephant_row['object'])
    @set($id = $elephant_row['id'])
    ?>
    <div class="elephant">
         <h2>{{ $elephant->name) }}</h2>
         <h3>{{ $elephant->age }}</h3>

         <p><img src="/views/img/{{ $elephant->image }}"
              alt="{{ $elephant->name }}"></p>
```

11.4.7 Delete Form Action

In order for the delete button to work we need to create a form. The form will point to the address ***"/ delete"*** in order to call the `delete()` method in our Routes file.

```
         <form action="/delete" method="post">

              <a href="edit.php?id={{ $id }}"
    class="button"><span class="fa fa-edit"></span>Edit</a>

              <input type="hidden" name="id" value="{{ $id }}" id="id">

              <button type="submit"><span class="fa fa-trash"></span>
                   Delete</button>
         </form>
    </div>
    @endforeach
  </div>
```

11.4.8 Edit Form Action

The Edit Elephant page will have a form on it with an action attribute which points to the address ***"/ update"*** in order to call the `update()` method in our Routes file.

The name attribute of each input or select field must match with the key we used to retreive the data from the `$_POST` array in our model. The value of each input tag will be filled in with the elephant's currently defined properties by echoing them into the value attribute.

For the select tag we will loop through and check each option to see if it matches the value of the elephant's `$image` property using an if statement. If it does, we will use the `selected` attribute in the tag.

```
  <form action="/update" method="post">
    <input type="hidden" name="id" value="{{ $id }}" id="id">

    <label for="name">Name:</label>
    <input type="text" name="name" value="{{ $this_elephant->name }}"
         id="name">

    <label for="age">Age:</label>
    <input type="number" name="age" value="{{ $this_elephant->age }}"
         id="age">

    <select name=”image” id=”image”>
         @foreach ($image_files as $file_name)
         <option value=”{{ $file_name }}" @if ($file_name ==
              $elephant->image) selected @endif>{{ $file_name }}
              </option>
         @endforeach
    </select>

    <button type="submit"><span class="fa fa-plus"></span>
         Edit Elephant</button>

  </form>
```

12. Abstract Classes

12.1 About Abstract Classes

Abstract classes are classes which are not supposed to be, and cannot be, instantiated, meaning that you can not create a new instance of them. Abstract classes are used as parent classes which would have child classes extended from them. Those child classes could then be instantiated.

Some MVC frameworks, such as Laravel, use abstract parent classes to define the general properties and methods for all of their major components such as Models and Controllers. These classes would include any properties and methods which all subclasses of the abstract class would require. Custom versions of these classes would then be created as child classes of the abstract classes which would inherit all of the properties and methods of the parent class.

12.2 The Abstract Model

12.2.1 Defining the Abstract Model

Some of our properties and methods could be moved from our Elephant Model to an abstract Model class which could be applied to any future Models. An example of something all of our Model classes would share would be the `__get()` magic method which is used as a universal getter method which gets the value of any private or protected property.

```
  namespace Models;

  use \Utilities\Connector;
  abstract class Model {
    public function __get($property) {
        if (property_exists($this, $property)) {
            return $this->$property;
        }
    }
```

Because the model is supposed to represent the data of your application, in practice, if it is following MVC and OOP principles closely then it should also contain most of the data processing encapsulated inside of the Model class as methods. This includes things such as converting and encoding objects as serialized objects or JSON.

```
    public function to_encoded() {
        $object = serialize($object);
        $object = base64_encode($object);
        return $object;
    }
```

This would also include things such as running SQL queries which are related to a particular Model.

```
    public function insert($table, $column, $value) {
        $db = Connector::get_instance();
        $pdo = $db->get_pdo();
        $sql = "INSERT INTO `{$table}` (`{$column}`) VALUES
            ('{$value}')";
        return $pdo->prepare($sql);
    }

    public function update($table, $column, $value, $id) {
        $db = Connector::get_instance();
        $pdo = $db->get_pdo();
        $sql = "UPDATE `{$table}` SET {$column}='{$value}' WHERE
            id={$id}";
        return $pdo->prepare($sql);
    }
```

We shouldn't need to create a new instance of a class to delete or select data so we could make those methods static.

```
    public static function delete($table, $id) {
        $db = Connector::get_instance();
        $pdo = $db->get_pdo();
        $sql = "DELETE FROM `{$table}` WHERE id={$id}";
        return $pdo->prepare($sql);
    }

    public static function get($table) {
        $db = Connector::get_instance();
        $pdo = $db->get_pdo();
        $sql = "SELECT * FROM `{$table}`";
        $query = $pdo->prepare($sql);
        $query->execute();
```

```
            return $query->fetchAll();
        }
        public static function where($table, $column, $value) {
            $db = Connector::get_instance();
            $pdo = $db->get_pdo();
            $sql = "SELECT * FROM `{$table}` WHERE {$column}={$value}";
            $query = $pdo->prepare($sql);
            $query->execute();
            return $query->fetchAll();
        }
    }
```

These methods would then act as a more encapsulated and abstracted interface which can be referenced by the Controller.

12.2.2 Extending the Abstract Model

We would then need to use our abstract Model class' namespace in our Elephant Model and extend it from the parent class. We can also create aliases for class names when we use them using the `as` keyword. In this case we are defining our `Model` class as `BaseModel`.

```
namespace Models;

use \Utilities\Sanitizer;
use \Models\Model as BaseModel;

class Elephant extends BaseModel {
```

12.3 The Abstract Controller

12.3.1 Defining the Abstract Controller

Now that we are using Blade templates we also have a reason to use an abstract Controller. Our abstract `Controller` class will import the `BladeOne` object using its namespace in order to activate our Blade templates.

```
namespace Controllers;

use \eftec\bladeone\BladeOne;
```

This class needs four properties which will hold data which will be parameters of the constructor method of the `BladeOne` class.

```
class Controller {
    private $root;
    private $views;
    private $cache;
    private $blade;
```

We will define these properties in the constructor method of our abstrct controller class. The `$root` property will just get the root directory. The `$views` property stores the folder where `BladeOne` will look for our Blade templates. The `$cache` property specifies where the regular PHP files which are compiled from our Blade templates are stored. We will store them in a `views` folder which is nested in a `storage` folder as this structure roughly mimics Laravel's folder structure. Finally we need to access the static constant of `MODE_AUTO` from the `BladeOne` class in order to automatically compile Blade templates.

```
    public function __construct() {
        $this->root = realpath(__DIR__ . '/..');
        $this->views = $this->root . '/views';
        $this->cache = $this->root . '/storage/views';
        $this->blade = new BladeOne($this->views, $this->cache,
            BladeOne::MODE_AUTO);
    }
```

We will also want to have a magic `__get()` method in the class.

```
    public function __get($property) {
        if (property_exists($this, $property)) {
            return $this->$property;
        }
    }
  }
```

12.3.2 Extending the Abstract Controller

We would then need to use our abstract Controller class' namespace in our Elephant Controller and extend it from the parent class.

```
  namespace Controllers;

  use \Models\Elephant;
  use \Controllers\Controller as BaseController;

  class ElephantController extends BaseController {
```

12.3.3 The Add Method

Now that we are using Blade templates we need to add some more methods to our Elephant Controller and revise some of the existing ones. We need an `add()` method to display the Add Elephant page. This method will return the `run()` method of the `BladeOne` object which will take the name of a Blade template to redirect to as the first parameter and an associative array of variables to pass to that Blade template as the second parameter. The name of each associative array key will become the name of a variable which can be used in the Blade template at a later point.

```
  public function add() {
    return $this->blade->run("add-elephant", [
       "title" => "Add Elephant",
       "image_files" => Elephant::$image_files
     ]);
  }
```

12.3.3 The Create Method

Our `create()` method will be mostly the same but will be simplified by using our abstract Model's methods.

```
    public function create(array $post) {
        if (!empty($post[‘name’]) && !empty($post[‘age’]) &&
            !empty($post[‘image’])) {

            $elephant = new Elephant($post);

            $query = $elephant->insert('elephants', 'object',
                $elephant->to_encoded());
```

It also needs to redirect to the new addresses of our pages.

```
            if ($query->execute()) {
                header('Location: /?add=success');
                exit();
            } else {
                header('Location: /add?db=error');
                exit();
            }
        } else {
            header('Location: /add?add=error');
            exit();
        }
    }
```

12.3.4 The Read Method

The `read()` method will be simplified by using our abstract Model's method.

```
    public function read() {
        $elephants = Elephant::get('elephants');

        foreach ($elephants as $elephant_num => $elephant_row) {
            $an_elephant = base64_decode($elephant_row);
            $an_elephant = unserialize($an_elephant);
            $elephants[$elephant_num]['object'] = $an_elephant;
        }
```

Instead of simply returning an array this method will return the `run()` method of the `BladeOne` object which will take the name of a Blade template to redirect to as the first parameter and an associative array of variables to pass to that Blade template as the second parameter. We need to pass the array of elephants to the Blade template as a variable.

```
        return $this->blade->run("add-elephant", [
            "title" => "View Elephants",
            "elephants" => $elephants
        ]);
    }
```

12.3.5 The Edit Method

We will also need an `edit()` method to display the Edit Elephant page. This method will be similar to the read method.

```
  public function edit($id) {
        $an_elephant_row = Elephant::where('elephants', 'id', $id);

        $an_elephant = base64_decode($an_elephant_row['object']);
        $an_elephant = unserialize($an_elephant);

        return $this->blade->run("add-elephant", [
             "id" => $id,
             "title" => "Edit {$an_elephant->name}",
             "elephant" => $an_elephant,
             "image_files" => Elephant::$image_files
        ]);
    }
```

12.3.6 The Update Method

The `update()` method is very similar to the create method with a couple of differences.

```
  public function update(array $post) {
        $id = $post['id'];
        if (!empty($post[‘name’]) && !empty($post[‘age’]) &&
             !empty($post[‘image’])) {

             $elephant = new Elephant($post);

             $query = $elephant->update('elephants', 'object',
                  $elephant->to_encoded(), $id);

             if ($query->execute()) {
                  header('Location: /?edit=success');
                  exit();
             } else {
                  header('Location: /edit?db=error&id=' . $id);
                  exit();
             }
        } else {
             header('Location: /edit?edit=error&id=' . $id);
             exit();
        }
    }
```

12.3.7 The Delete Method

The `delete()` method will be very simple now.

```
    public function delete(array $post) {
        $id = $post['id'];
        $query = Elephant::delete('elephants', $id);

        if ($query->execute()) {
            header('Location: /?delete=success');
            exit();
        } else {
            header('Location: /?delete=error');
            exit();
        }
    }
}
```