Laravel

Databases and Web Applications Laboratory (LBAW)
Bachelor in Informatics Engineering and Computation (L.EIC)

Sérgio Nunes
Dept. Informatics Engineering
FEUP · U.Porto

Based on Laravel: Up & Running, Matt Stauffer (2019)
Based on Laravel Official Documentation (10.x)

Outline

- Laravel Overview
- Laravel Setup
- Routing and Controllers
- Blade Templating
- Database and Eloquent
- Laravel at LBAW

Laravel Overview

Frameworks

- Libraries and packages handle isolated components.
- A framework groups a collection of components together with settings and configurations, directory structure, code templates and skeletons, etc.
- Frameworks include decisions not only on which components but on how these components work together.
- Alternative: decide on each individual component; setup on how they work together; maintain component's evolution.
- Frameworks bring tested configurations, consistency, robustness, communities.
- Frameworks also bring commitment, reduced freedom, vendor lock-in, third-party dependency.

History of PHP Frameworks

- Ruby on Rails (2004) is a landmark in the development of web application frameworks. Popular concepts introduced by RoR include: MVC, RESTful JSON APIs, convention over configuration, Active-Record.

- Notable PHP web frameworks:
  - CakePHP (2005), cakephp.org
  - Symfony (2005), symfony.com
  - CodeIgniter (2006), codeigniter.com
  - Slim (2010), slimframework.com
  - Laravel (2011), laravel.com

Laravel Highlights

- Laravel is a backend web development framework, highlighting:
- **Rapid application development framework**, i.e. fast learning curve, support for most common tasks, consistent API, predictable structure, large tools and packages ecosystem.
- **Convention over configuration**, i.e. default configurations out of the box.
- **Simplicity**, i.e. start simple approach philosophy, and then bring more complex solutions if needed; use of simpler PHP syntax and coding practiced.

Laravel Community

- Laravel has a strong and active community.
- A large number of open and freely available resources.
- Official documentation, laravel.com/docs
- From the community
  - Laracasts, laracasts.com
  - Laravel News, laravel-news.com
  - Laravel Podcast, laravelpodcast.com
  - Discussion Forums, laracasts.com/discuss

  - Awesome Laravel, github.com/chiraggude/awesome-laravel

Laravel Origins: A PHP Documentary

https://www.youtube.com/watch?v=127ng7botO4

Laravel Setup

PHP Composer

- getcomposer.org
- Composer is PHP’s package and dependency manager.
- It works on a per-project basis, i.e. there are no global installations.
- Packages are installed in a ‘vendor’ directory inside the project.

- Declare which packages are needed for a project
- Composer finds and downloads package versions and dependencies that need to be installed.
- Packages updates are managed with the update command.

Composer Basic Usage

- Composer uses a JSON file — composer.json — per-project to manage its dependencies.
- Best practice is to build the **composer.json** file using composer.

- Packages can be found at Packagist, packagist.org
- Example: install phpunit, PHP unit testing framework:
  - `composer require phpunit/phpunit`
  - A composer.json file will be created (or edited) and the package will be downloaded.

- A **composer.lock** file is used to fix the versions used in a project.
  - composer install will not get the latest versions but the ones specified in composer.lock.
  - composer.lock should be committed to the version control system.

Laravel “Hello World”

Laravel “Hello World” Example

- Build a simple Laravel web application that shows posts stored in a SQLite database.

- Receive a GET request at /posts/{slug}.
- Read post information from the SQLite database.
- Present the result as an HTML page.

Install Laravel 10.x

- Requirements:
  - PHP (>= 8.0)
  - Composer, getcomposer.org
- Use Composer to start a new Laravel project
  - `composer create-project laravel/laravel example-app 10`
- Change to the ‘example-app’ directory
  - `cd example-app`
- Start a local web server (using PHP’s **or** Laravel’s):
  - `php -S localhost:8000 -t public`
  - `php artisan serve`
- View at http://localhost:8000

Database Schema

- `database/example-app.sql`

```
DROP TABLE IF EXISTS posts;

CREATE TABLE posts (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    date DATE NOT NULL DEFAULT CURRENT_TIMESTAMP,
    slug TEXT NOT NULL UNIQUE,
    title TEXT NOT NULL,
    body TEXT
);

INSERT INTO posts (slug, title, body) VALUES ('hello-world', 'Hello world!', 'This is a first text.');
INSERT INTO posts (slug, title, body) VALUES ('test', 'Testing', 'We are testing the system.');
INSERT INTO posts (slug, title, body) VALUES ('fox', 'Fox?', 'The quick brown fox jumps over the lazy dog.');
```

Create Database

- Define the SQL schema (in previous slide)
  - `database/example-app.sql`
- Create a new SQLite database
  - `touch database/example-app.sqlite`
- Create the database schema from the SQL file.
  - `sqlite3 database/example-app.sqlite < database/example-app.sql`

Database Connection

- Setup database access in the environment configuration file
  - `.env`
- Remove previous DB_* settings and add:
  - `DB_CONNECTION=sqlite`
  - `DB_DATABASE={ full path }/database/example-app.sqlite`
- To test the database connection you can open a CLI to the database:
  - `php artisan db`
- And then, in SQLite, run:
  - `.tables`

Automate Database Creation (1)

- Laravel's seeding functions can be used to create the database from a SQL file.
- Change [`database/seeders/DatabaseSeeder.php`]

```php
<?php

namespace Database\Seeders;

// use Illuminate\Database\Console\Seeds\WithoutModelEvents;
use Illuminate\Database\Seeder;
use Illuminate\Support\Facades\DB;

class DatabaseSeeder extends Seeder
{
    /**
     * Run the database seeds.
     *
     * @return void
     */
    public function run()
    {
        $path = 'database/example-app.sql';
        DB::unprepared(file_get_contents($path));
        $this->command->info('Database seeded!');
    }
}
```

Automate Database Creation (2)

- The database can be dropped with:
  - `php artisan db:wipe`
- And re-created with:
  - `php artisan db:seed`

Models

- Laravel models control database access.
- Create a Laravel model for ‘posts’ using Artisan
  - `php artisan make:model Post`
- A new file will be created at
  - `app/Models/Post.php`
- Disable automatic timestamps by editing the model file.

```
class Post extends Model
{
    use HasFactory;

    // Don't add create and update timestamps in database.
    public $timestamps  = false;
}
```

Models Usage

- Models can be used to interact with the database.
- Test the ‘Post’ model with Artisan’s tinker tool (CLI).
  - `php artisan tinker`

```
>>> use App\Models\Post;
>>> Post::find(1);
=> App\Models\Post {#4626
     id: 1,
     date: "2022-11-03 19:03:07",
     slug: "hello-world",
     title: "Hello world!",
     body: "This is a first text.",
   }

>>> Post::first->slug;
=> "hello-world"
```

```
>>> Post::all()
=> Illuminate\Database\Eloquent\Collection {#4402
     all: [
       App\Models\Post {#4360
         id: 1,
         date: "2022-11-03 19:03:07",
         slug: "hello-world",
         title: "Hello world!",
         body: "This is a first text.",
       },
       App\Models\Post {#4629
         id: 2,
         date: "2022-11-03 19:03:07",
         slug: "test",
         title: "Testing",
         body: "We are testing the system.",
       },
       App\Models\Post {#4628
         id: 3,
         date: "2022-11-03 19:03:07",
         slug: "fox",
         title: "Fox?",
         body: "The quick brown fox jumps over the lazy dog.",
       },
     ],
   }
```

Controllers

- Laravel controllers are where each resource logic is implemented.
- To create a controller for the Post model use:
  - `php artisan make:controller PostController`
- A new file will be created at:
  - `app/Http/Controllers/PostController.php`

Post Controller

- The Post controller defines a show function that receives a ‘slug’ and:
  - If found, show the associated post;
  - If not, show a not found message.

- `app/Http/Controllers/PostController.php`

```php
namespace App\Http\Controllers;

use App\Models\Post;
use Illuminate\Http\Request;

use Illuminate\Database\Eloquent\ModelNotFoundException;

class PostController extends Controller
{
    /**
     * Show the Post for a given slug.
     *
     * @param  $slug
     * @return \Illuminate\Http\Response
     */
    public function show($slug)
    {
        try {
            // Show the 'posts.show' view based on the slug in the route.
            return view('posts.show', [
                'post' => Post::where('slug', $slug)->firstOrFail()
            ]);
        }
        catch(ModelNotFoundException $e) {
            // TBD: redirect to a search page based on the not found slug text.
            return "Post not found.";
        }
    }
}
```

Views

- Laravel views are used to define the user interface.
- Views are written in Blade and defined under:
  - `resources/views`
- To create a new post.show view, create a new folder:
  - `resources/views/posts`
- And inside create a new file:
  - `resources/views/posts/show.blade.php`
- This view is called from the controller with:
  - `return view('posts.show', [ 'post' => … ]);`

Post View

- Laravel views are written in Blade.
- `resources/views/posts/show.blade.php`

```
<!DOCTYPE html>
<html>

<head>
    <title>{{ $post->title }}</title>
</head>

<body>
    <h1>{{ $post->title }}</h1>
    <p>{{ $post->body }}</p>
</body>

</html>
```

Routes

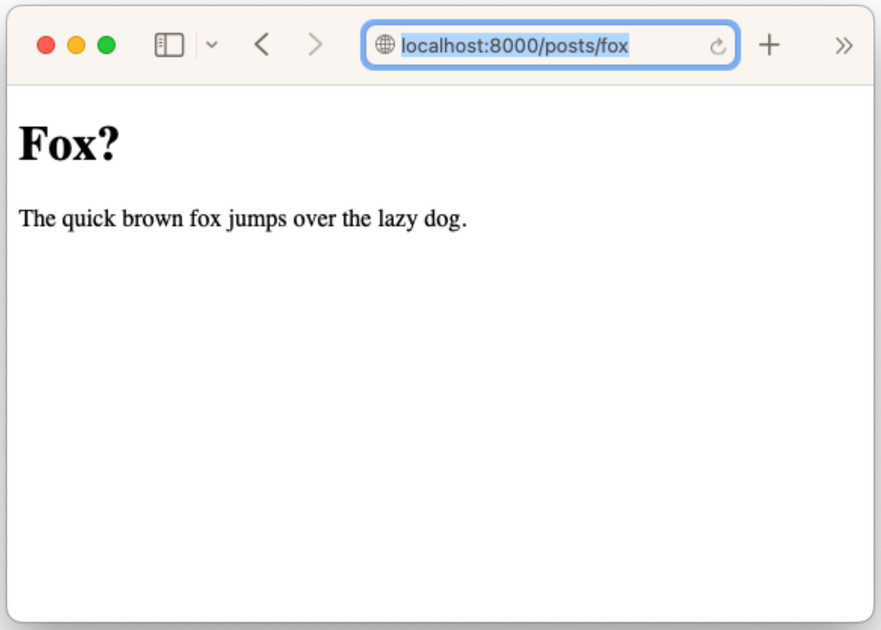
- Laravel routes define how web requests are processed.
- Web routes are defined in the file:
  - `routes/web.php`
- Create a new route for a GET /posts/{slug} to be handled by the show function in the PostController.

```php
use App\Http\Controllers\PostController;

Route::get('posts/{slug}', [PostController::class, 'show']);
```

Test

- Start Laravel's built-in web server:
  - `php artisan serve`
- Test URLs
  - localhost:8000/posts/fox
  - localhost:8000/posts/not



Laravel Details

Configuration

- Configuration files are stored in the `config` directory.
- Current configuration options can be shown with:
  - `php artisan about`
- The `.env` file can be used to set up configurations that differ between environments, e.g. local development, production server.
  - Typical configurations: application settings (name, debug, url, …), database (host, password, …).
  - APP_DEBUG is used to enable or disable debugging mode.

Directory Structure

- `app`, core code of the application, including models and controllers.
- `bootstrap`, framework bootstrapping files, namely app.php, and a cache directory.
- `config`, configuration files and settings.
- `database`, database migrations, model factories, and seeds.
- `lang`, language files.
- `public`, contains index.php, the entry point for all requests, and assets (images, CSS, JavaScript).
- `resources`, contains views and uncompiled assets (CSS, JavaScript).
- `routes`, all routes definitions, including web and api routes.
- `storage`, caches, logs, and compiled Blade templates.
- `tests`, automated unit and integrated tests.
- `vendor`, composer dependencies.

Request Lifecycle

- All requests are first handled by `public/index.php`
- The first step is to obtain an instance of the Laravel application.
- The incoming request is sent to a **kernel**, typically the HTTP kernel (handles web requests).
  - Configure error handling, logging, set up the app environment.
- The request goes through a set of **middleware** (multilayers) for session handling, security, etc.
- The request then moves through **service providers**, which bootstrap various components, e.g. database, validation, routing.
  - An important service provider is the Route Service Provider, which manages the routes, and dispatches requests to the specific controller methods.
- The response sent by the controller goes back to the route's middleware, and is finally sent to the client.

Request Lifecycle

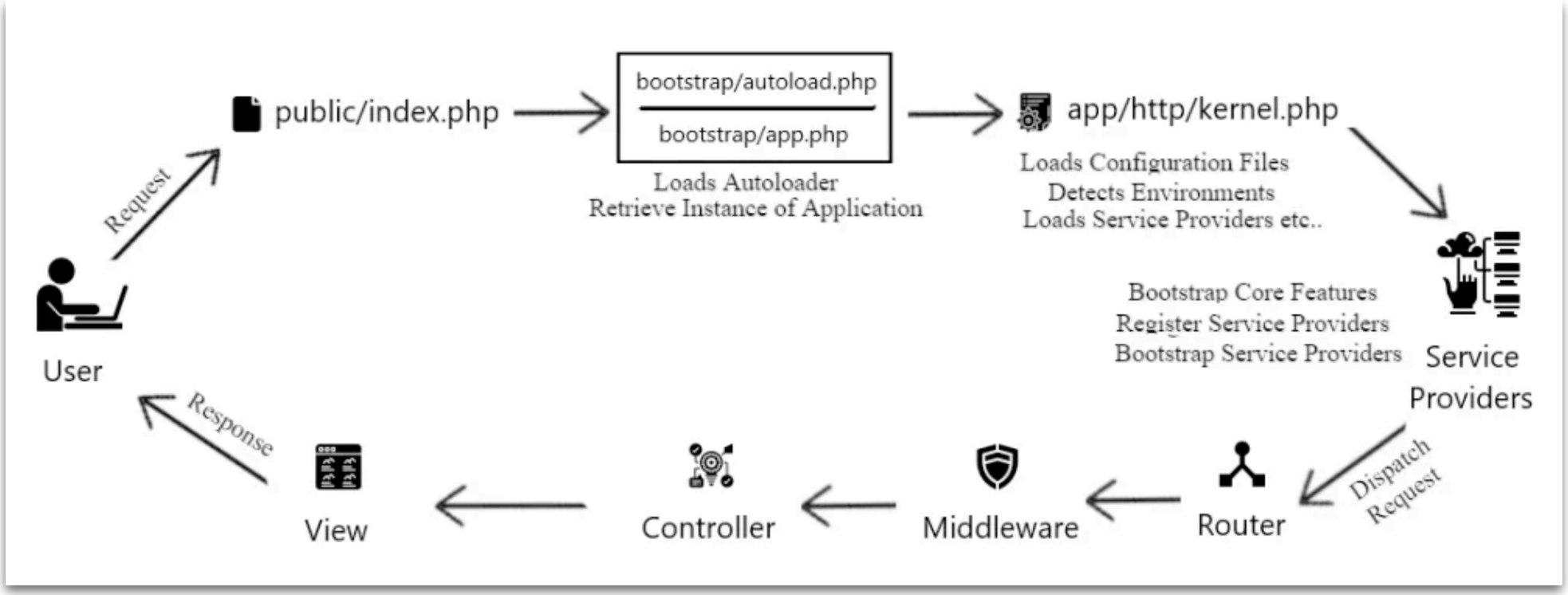
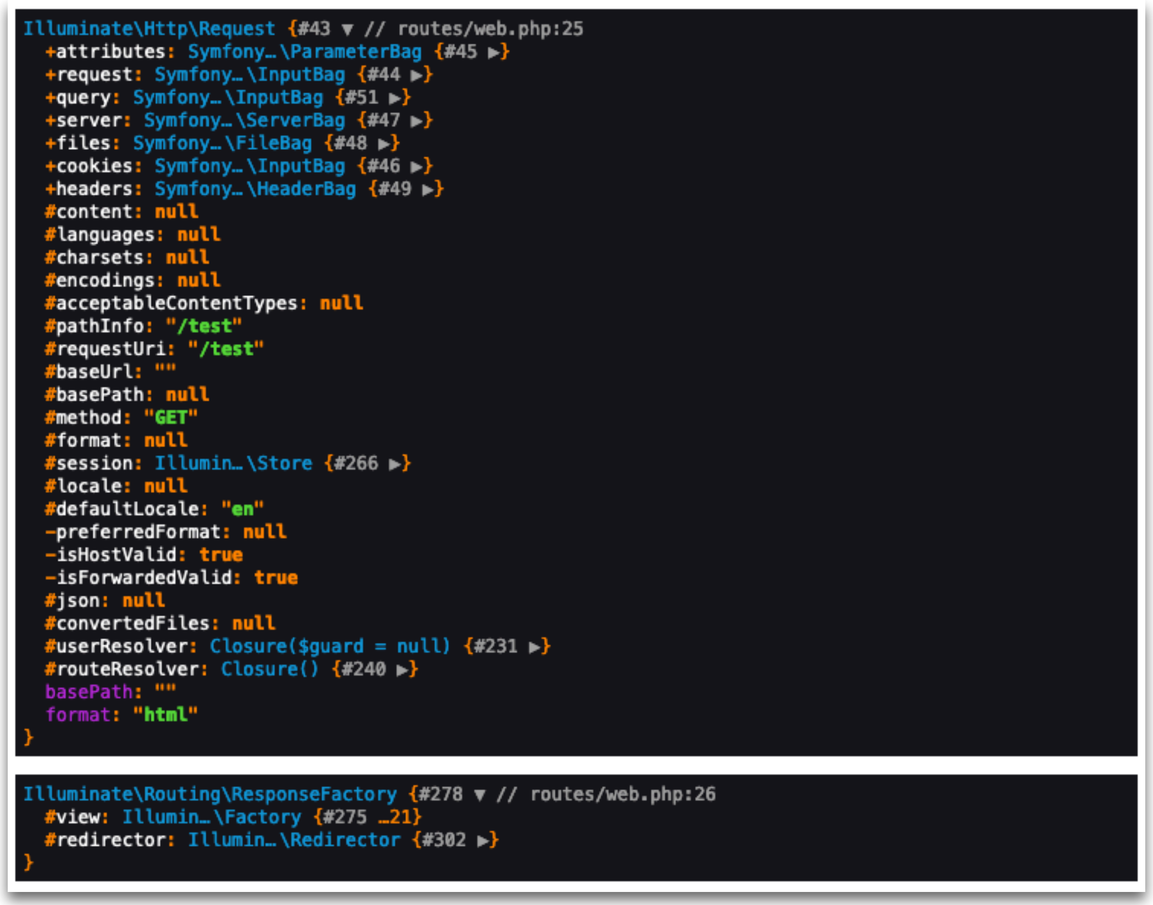


Image from: Parixit Patel, Laravel Request Lifecycle (2020)
https://dev.to/patelparixit07/laravel-request-lifecycle-195e

Request and Response Objects

- You can inspect the request and response objects with specific methods.

```php
Route::get('test', function () {
    dump(request());
    dump(response());
});
```



Routing and Controllers

Routes

Route Definitions

- Web routes are defined in `routes/web.php`
- API routes are defined in `routes/api.php`
- Currently defined routes can be listed with:
  - `php artisan route:list`
- Two basic route definitions.
  - GET /
  - GET /about

```php
use Illuminate\Support\Facades\Route;


Route::get('/', function () {
    // Return a string.
    return 'Hello, World!';
});

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

Route Methods

- The router allows the registration of routes responding to any HTTP verb.

```
Route::get($uri, $callback);
Route::post($uri, $callback);
Route::put($uri, $callback);
Route::patch($uri, $callback);
Route::delete($uri, $callback);
Route::options($uri, $callback);
```

- Also allows the registration of routes that match multiple HTTP verbs (any, match).

```
Route::match(['get', 'post'], '/', function () {
    //
});

Route::any('/', function () {
    //
});
```

Redirect Routes

- The redirect method can be used to define a route redirection.
- By default, the redirect method uses a 302 status code.
- A specific HTTP status code can be defined as the third optional argument.

```php
// By default a 302 status code is used.
Route::redirect('/here', '/there');

// The specific HTTP code can be defined.
Route::redirect('/here', '/there', 301);

// This returns a 301 status code.
Route::permanentRedirect('/here', '/there');
```

View Routes

- The view method provides a shortcut for simple view routes.
- The first argument is the URI of the route.
- The second argument is the view to use.
- The third, optional, argument is an array of data to pass to the view.

```
Route::view('/about', 'about');

Route::view('/welcome', 'welcome', ['name' => 'Taylor']);
```

Route Parameters

- Route parameters allow the definition of variable segments within a URI.
- Multiple route parameters can be required in a single route.
- Parameters can be defined as optional with a ?.

```php
// Route parameters are defined with curly brackets.
Route::get('/user/{id}', function ($id) { ... });

// Multiple parameters can be used in a single route.
Route::get('/posts/{post}/comments/{comment}', function ($postId, $commentId) { ... });

// Parameters can be set as optional.
Route::get('/user/{name?}', function ($name = 'John') { ... });
```

Route Regular Expressions

- Regular expressions can be defined to establish constrains in routes.
- Requests will only be handled by the route if they match the pattern.
- If no matches are found, a 404 HTTP response is returned.

```php
Route::get('/user/{name}', function ($name) {
    //
})->where('name', '[A-Za-z]+');

Route::get('/user/{id}', function ($id) {
    //
})->where('id', '[0-9]+');

Route::get('/user/{id}/{name}', function ($id, $name) {
    //
})->where(['id' => '[0-9]+', 'name' => '[a-z]+']);
```

Named Routes

- Named routes are convenient for generating URLs and handling redirects.
- A name can be specified for a route by chaining the name() method onto the route.
- Names can be used in redirects.

```php
// Name a route as 'profile'.
Route::get('/user/profile', function () {
    //
})->name('profile');


// Generate a redirect to 'profile' route.
return redirect()->route('profile');
```

Route Behavior

- Route behavior can be defined with a closure (i.e. anonymous function).

```php
Route::get('/greeting', function () {
    return 'Hello World';
});
```

- The **standard approach** is to handle it through a controller method.
- Routes can be grouped to share the came controller.

```php
// Route /users is handle by the index method defined in the UserController.
Route::get('/user', [UserController::class, 'index']);

// Two routes grouped to use different methods from the same controller.
Route::controller(OrderController::class)->group(function () {
    Route::get('/orders/{id}', 'show');
    Route::post('/orders', 'store');
});
```

Controllers

Controllers

- Controllers organize the logic of one or more related routes under the same class.
- Controllers are grouped under
  - `app/Http/Controllers`.
- Artisan provides commands to create a controller template.
  - `php artisan make:controller UserController`

```php
<?php

namespace App\Http\Controllers;

use App\Models\User;

class UserController extends Controller
{
    /**
     * Show the profile for a given user.
     *
     * @param  int  $id
     * @return \Illuminate\View\View
     */
    public function show($id)
    {
        return view('user.profile', [
            'user' => User::findOrFail($id)
        ]);
    }
}
```

Resource Controller

- Creating methods for all standard CRUD operations is a common use case.
- Laravel provides mechanisms to streamline this scenario.
- First, create a standard resource controller with Artisan:
  - php artisan make:controller PhotoController --resource
- This will create methods all for standard HTTP CRUD operations.
- These methods can be registered as routes with a single command:
  - Route::resource('photos', PhotoController::class);

Actions Handled By Resource Controller

Verb	URI	Action	Route Name
GET	`/photos`	index	photos.index
GET	`/photos/create`	create	photos.create
POST	`/photos`	store	photos.store
GET	`/photos/{photo}`	show	photos.show
GET	`/photos/{photo}/edit`	edit	photos.edit
PUT/PATCH	`/photos/{photo}`	update	photos.update
DELETE	`/photos/{photo}`	destroy	photos.destroy

Data Validation

Data Validation

- Laravel includes many features to support validation of incoming data.
- The standard approach is to use the `validate` method in all incoming HTTP requests.
- Validation rules are passed into the `validate` method. If it fails, a validation exception will be thrown.

```php
// Store a new blog post.
public function store(Request $request)
{
    $validated = $request->validate([
        'title' => ['required', 'unique:posts', 'max:255'],
        'body' => ['required'],
    ]);
    // The blog post is valid...
}
```

- Official documentation:
  - Validation, laravel.com/docs/10.x/validation
  - Validation rules, laravel.com/docs/10.x/validation#available-validation-rules

Blade Templating

Views

- Views enable the separation of the presentation layer (from the model and controller layers), and are defined under the `resource/views` directory.
- Views can be nested within subdirectories and then referenced using “dot notation”.

```php
// Calls the view stored at resources/views/about.blade.php
return view('about', $data);


// Calls the view stored at resources/views/admin/profile.blade.php
return view('admin.profile', $data);
```

Passing Data to Views

- Data can be passed directly as an array of key / value pairs.
- Or, the `with` function can be used with the `view` method.

```php
// Data can be passed as an array of key / value pair.
return view('greetings', ['name' => 'Victoria']);


// Individual pieces of data can be passed using with.
return view('greeting')
    ->with('name', 'Victoria')
    ->with('occupation', 'Astronaut');
```

Blade Templates

- Blade is the templating engine included with Laravel.

```
<!DOCTYPE html>
<html lang="{{ app()->getLocale() }}">
  <head>
    <!-- CSRF Token -->
    <meta name="csrf-token" content="{{ csrf_token() }}">

    <title>{{ config('app.name', 'Laravel') }}</title>

    <!-- Styles -->
    <link href="{{ asset('css/milligram.min.css') }}" rel="stylesheet">
    <link href="{{ asset('css/app.css') }}" rel="stylesheet">
 </head>
  <body>
    <main>
      <header>
        <h1><a href="{{ url('/cards') }}">Thingy!</a></h1>
        @if (Auth::check())
        <a class="button" href="{{ url('/logout') }}"> Logout </a> <span>{{ Auth::user()->name }}</span>
        @endif
      </header>
      <section id="content">
        @yield('content')
      </section>
    </main>
  </body>
</html>
```

Displaying Data

- Data can be displayed by using curly brackets to wrap a variable.
- Results of PHP functions can also be displayed inside a Blade template.
- The `@verbatim / @endverbatim` directives can be be used to display raw text.

```
Route::get('/', function () {
    return view('welcome', ['name' => 'Samantha']);
});
```

```
// Displaying a variable.
Hello, {{ $name }}.

// Displaying the result of a PHP function.
The current UNIX timestamp is {{ time() }}.
```

```
@verbatim
    <div class="container">
        Hello, {{ name }}.
    </div>
@endverbatim
```

Conditional Directives (if)

- Blade provides shortcuts to common PHP control structures, e.g. if statements.

```
@if (count($records) === 1)
    I have one record!
@elseif (count($records) > 1)
    I have multiple records!
@else
    I don't have any records!
@endif



@unless (Auth::check())
    You are not signed in.
@endunless
```

```
@isset($records)
    // $records is defined and is not null...
@endisset

@empty($records)
    // $records is "empty"...
@endempty
```

Conditional Directives (environment)

- Blade also includes conditional authentication directives and environment directives.

```
@auth
    // The user is authenticated...
@endauth

@guest
    // The user is not authenticated...
@endguest
```

```
@production
    // Production specific content...
@endproduction



@env('staging')
    // The application is running in "staging"...
@endenv

@env(['staging', 'production'])
    // The application is running in "staging" or "production"...
@endenv
```

Loop Directives

- Blade provides directives to work with PHP's loop structures.
- The loop variable can be used to further customize the template.

```
@for ($i = 0; $i < 10; $i++)
    The current value is {{ $i }}
@endfor

@foreach ($users as $user)
    <p>This is user {{ $user->id }}</p>
@endforeach

@forelse ($users as $user)
    <li>{{ $user->name }}</li>
@empty
    <p>No users</p>
@endforelse

@while (true)
    <p>I'm looping forever.</p>
@endwhile
```

```
@foreach ($users as $user)
    @if ($loop->first)
        This is the first iteration.
    @endif

    @if ($loop->last)
        This is the last iteration.
    @endif

    <p>This is user {{ $user->id }}</p>
@endforeach
```

Including Subviews

- The `@include` directive can be used to include a Blade view inside another view.
- All variables available to the parent view are also made available to the included view.
- Conditional includes are possible with `@includeIf`, `@includeWhen`, `@includeUnless`.

```
<div>
    @include('shared.errors')

    <form>
        <!-- Form Contents -->
    </form>
</div>
```

```
@includeWhen($boolean, 'view.name', ['status' => 'complete'])

@includeUnless($boolean, 'view.name', ['status' => 'complete'])
```

Building Layouts

- Template inheritance can be used to break-up complex designs.
- The `@section` directive defines a section of content.
- The `@yield` directive is used to display the contents of a given section.

```
<!-- resources/views/layouts/app.blade.php -->

<html>
    <head>
        <title>App Name - @yield('title')</title>
    </head>
    <body>
        @section('sidebar')
            This is the master sidebar.
        @show

        <div class="container">
            @yield('content')
        </div>
    </body>
</html>
```

```
@extends('layouts.app')

@section('title', 'Page Title')

@section('sidebar')
    @parent

    <p>This is appended to the master sidebar.</p>
@endsection

@section('content')
    <p>This is my body content.</p>
@endsection
```

Database and Eloquent

Database

Database

- Laravel provides a suit of tools for interacting with databases.
- Database interaction is supported through the use of:
  - raw SQL, a query builder, and Eloquent ORM.
- Laravel provides first-party support for:
  - MariaDB, MySQL, PostgreSQL, SQLite, SQL Server.
- Configuration is done through environment variables defined in .env files.
  - These settings are used in /config/database.php (which is not expected to be changed).

Raw SQL Queries

- Once the database connection is configured, SQL queries are run using the DB facade methods: select, update, insert, delete, and statement.

```php
use Illuminate\Support\Facades\DB;


// Basic example.
$users = DB::select('select * from users');

foreach ($users as $user) {
    echo $user->name;
}


// Using named bindings in a select SQL statement.
$results = DB::select('select * from users where id = :id', ['id' => 1]);

// Insert SQL statement.
DB::insert('insert into users (id, name) values (?, ?)', [1, 'Marc']);

// General SQL statement.
DB::statement('drop table users');
```

SQL Query Builder

- Laravel query builder can be used to interact with the database through a convenient method-based interface.
- Extensive support for:
  - aggregates
  - joins
  - where clauses
  - ordering, grouping, limit, offset
  - etc.
- Query builder documentation
  - laravel.com/docs/10.x/queries

```php
use Illuminate\Support\Facades\DB;

// Basic example using query builder.
$users = DB::table('users')->get();

foreach ($users as $user) {
    echo $user->name;
}


// Retrieving a single row and column.
$user = DB::table('users')->where('name', 'John')->first();

return $user->email;


// Retrieving a list of column values
$titles = DB::table('users')->pluck('title');

foreach ($titles as $title) {
    echo $title;
}
```

Eloquent ORM

Eloquent ORM

- Laravel includes Eloquent, an object-relational mapper (ORM).
- Using Eloquent, each database table has a corresponding model that is used to interact with that table.
- Model classes can be generated using Artisan, e.g.:
  - `php artisan make:model Flight`
- This is the approach adopted in LBAW.

Eloquent Naming Conventions

- Table names:
  - Eloquent will assume that a `Flight` model will link to a `flights` table; while an `AirTrafficController` table will store records in an `air_traffic_controller` table.
  - If the naming does not fit this convention, the `$table` property on the model can be used to specify the table name.
- Primary keys:
  - Eloquent will assume that each model's corresponding database table has a primary key column named `id`.
  - The `$primaryKey` property cab be used to specify a different column name.
  - Eloquent also assumes that primary keys are automatically incrementing integers.

Timestamps

- Eloquent expect two timestamps columns,
  - `created_at` and `updated_at`
  - Laravel uses these to manage migrations.
  - To disable this, set `$timestamps` to false.

- These are **not used** in LBAW (database migrations are not used).

Retrieving Models

- Once models are created, they can be used to retrieve data.
- Each model will work as a specialized query builder.

```php
// Using the Flight model to query the database.
use App\Models\Flight;

foreach (Flight::all() as $flight) {
    echo $flight->name;
}

// Each model serves as a query builder.
$flights = Flight::where('active', 1)
    ->orderBy('name')
    ->limit(10)
    ->get();
```

Inserting Models

- To insert a new record to the database, a new model is instantiated.
- The `save` method is used to store the instance in the database.

```php
use App\Models\Flight;

// Create a new Flight instance, set data, and store it.
$flight = new Flight;

$flight->name = "John Smith";

flight->save();



// Single statement call.
$flight = Flight::create([
    'name' => 'London to Paris',
]);
```

Eloquent Relationships

- To navigate between related records, using each model's relationships, Eloquent allows for the definition of relationships.
- This allows for chaining additional query constraints, e.g.:
  - `$post->comments()->where('title', 'foo')->first();`
  - `$comment->post->title;`
- Example, a Post has many Comments (one-to-many relationship).
  - Post hasMany Comments
  - Comment belongsTo Post

```php
class Post extends Model
{
    // Get the comments for the blog post.
    public function comments() {
        return $this->hasMany(Comment::class);
    }
}
```

```php
class Comment extends Model
{
    // Get the post that owns a comment.
    public function post() {
        return $this->belongsTo(Post::class);
    }
}
```

- See Eloquent Relationships for details on how to map
  - laravel.com/docs/10.x/eloquent-relationships

Database Relationships in Laravel

One-to-One Relationship

- User [1] —has— [1] Profile

```sql
-- Database Schema

CREATE TABLE users (
    id INTEGER PRIMARY KEY,
    name TEXT NOT NULL
);

CREATE TABLE profiles (
    id INTEGER PRIMARY KEY,
    user_id INTEGER UNIQUE REFERENCES users,
    bio TEXT
);
```

```php
# Laravel

class User extends Model {
    public function profile() {
        return $this->hasOne(Profile::class);
    }
}

class Profile extends Model {
    public function user() {
        return $this->belongsTo(User::class);
    }
}
```

```php
# Usage

// Access profile from user
$userBio = $user->profile->bio;

// Access user from profile
$userName = $profile->user->name;
```

One-to-Many Relationship

- Post [1] —has— [N] Comment

```sql
-- Database Schema

CREATE TABLE posts (
    id INTEGER PRIMARY KEY,
    title TEXT NOT NULL,
    content TEXT
);

CREATE TABLE comments (
    id INTEGER PRIMARY KEY,
    post_id INTEGER REFERENCES posts,
    content TEXT,
    created_at TIMESTAMP
);
```

```php
# Laravel

class Post extends Model {
    public function comments() {
        return $this->hasMany(Comment::class);
    }
}

class Comment extends Model {
    public function post() {
        return $this->belongsTo(Post::class);
    }
}
```

```php
# Usage

// Get all comments for a post
foreach ($post->comments as $comment) {
    echo $comment->content;
}

// Get post for a comment
$postTitle = $comment->post->title;
```

Many-to-Many Relationship

- Student [N] —enrollment— [N] Course

```sql
-- Database Schema

CREATE TABLE students (
    id INTEGER PRIMARY KEY,
    name TEXT NOT NULL
);

CREATE TABLE courses (
    id INTEGER PRIMARY KEY,
    name TEXT NOT NULL
);

CREATE TABLE course_student (
    student_id INTEGER REFERENCES students,
    course_id INTEGER REFERENCES courses,
    PRIMARY KEY (student_id, course_id)
);
```

```php
# Laravel

class Student extends Model {
    public function courses() {
        return $this->belongsToMany(Course::class);
    }
}

class Course extends Model {
    public function students() {
        return $this->belongsToMany(Student::class);
    }
}
```

```php
# Usage

// Get all courses for a student
foreach ($student->courses as $course) {
    echo $course->name;
}

// Attach/detach courses
$student->courses()->attach($courseId);
$student->courses()->detach($courseId);
```

Many-to-Many with Properties (Association Class)

- User [N] —[permissions]— [N] Role

```sql
-- Database Schema

CREATE TABLE roles_users (
    user_id INTEGER REFERENCES users,
    role_id INTEGER REFERENCES roles,
    permissions JSON,
    created_at TIMESTAMP,
    PRIMARY KEY (user_id, role_id)
);
```

```php
# Laravel

class User extends Model {
    public function roles() {
        return $this->belongsToMany(Role::class)
            ->withPivot('permissions');
    }
}
```

```php
# Usage

// Access pivot data
$permissions = $user->roles->first()->pivot->permissions;

// Attach with pivot data
$user->roles()->attach($roleId, ['permissions' => $permissions]);
```

Generalization Relationships in Laravel

Recall: Approaches to Map Generalization

- **1. Single Table Inheritance [superclass]**
  - One table for all entities
  - Discriminator column to distinguish types
  - All attributes in same table
- **2. Class Table Inheritance [entity-relationship]**
  - Separate table for each concrete class
  - Common attributes in parent table
  - Foreign key relationship to parent
- **3. Concrete Table Inheritance [object-oriented]**
  - Separate independent tables
  - All attributes (including inherited) in each table
  - No database relationship between tables

1. Single Table Inheritance

```sql
-- Database Schema

CREATE TABLE media (
    id INTEGER PRIMARY KEY,
    type VARCHAR(10) NOT NULL, -- 'book' or 'dvd'
    title VARCHAR(255),

    -- Common attributes
    price DECIMAL(10,2),
    publisher VARCHAR(255),

    -- Book specific
    author VARCHAR(255),
    isbn VARCHAR(13),
    pages INTEGER,

    -- DVD specific
    director VARCHAR(255),
    duration INTEGER,
    rating VARCHAR(5)
);
```

```php
# Laravel

-- in App\Models\Media.php
class Media extends Model
{
    protected $table = 'media';
}

-- in App\Models\Book.php
class Book extends Media
{
    protected static function booted()
    {
        static::creating(function ($book) {
            $book->type = 'book';
        });
    }

        static::addGlobalScope('type', function ($query) {
            $query->where('type', 'book');
        });
    }
}

-- in App\Models\DVD.php
class DVD extends Media
{
    protected static function booted()
    {
        static::creating(function ($dvd) {
            $dvd->type = 'dvd';
        });
    }

        static::addGlobalScope('type', function ($query) {
            $query->where('type', 'dvd');
        });
    }
}
```

```php
# Usage

$books = Book::all();
$books = DVD::where('duration', '>', 120)->get();
```

2. Class Table Inheritance

```sql
-- Database Schema

CREATE TABLE media (
    id INTEGER PRIMARY KEY,
    title VARCHAR(255),
    price DECIMAL(10,2),
    publisher VARCHAR(255)
);

CREATE TABLE books (
    media_id INTEGER PRIMARY KEY REFERENCES media,
    author VARCHAR(255),
    isbn VARCHAR(13),
    pages INTEGER
);

CREATE TABLE dvds (
    media_id INTEGER PRIMARY KEY REFERENCES media,
    director VARCHAR(255),
    duration INTEGER,
    rating VARCHAR(5)
);
```

```php
# Laravel

# in App/Models/Media.php
class Media extends Model
{
    protected $table = 'media';
    protected $fillable = ['title', 'price', 'publisher'];
}

# in App/Models/Book.php
class Book extends Media
{
    protected $table = 'books';
    protected $primaryKey = 'media_id';

    public function mediaDetails()
    {
        return $this->belongsTo(Media::class, 'media_id');
    }
}

# in App/Models/DVD.php
class DVD extends Media
{
    protected $table = 'dvds';
    protected $primaryKey = 'media_id';

    public function mediaDetails()
    {
        return $this->belongsTo(Media::class, 'media_id');
    }
}
```

3. Concrete Table Inheritance

```sql
-- Database Schema

CREATE TABLE books (
    id INTEGER PRIMARY KEY,
    title VARCHAR(255),
    price DECIMAL(10,2),
    publisher VARCHAR(255),
    author VARCHAR(255),
    isbn VARCHAR(13),
    pages INTEGER
);

CREATE TABLE dvds (
    id INTEGER PRIMARY KEY,
    title VARCHAR(255),
    price DECIMAL(10,2),
    publisher VARCHAR(255),
    director VARCHAR(255),
    duration INTEGER,
    rating VARCHAR(5)
);
```

```php
# Laravel

# in App/Models/Media.php
abstract class Media extends Model
{
    // Common methods for both types
    abstract public function getType();
}

# in App/Models/Book.php
class Book extends Media
{
    protected $table = 'books';

    public function getType()
    {
        return 'book';
    }
}

# in App/Models/DVD.php
class DVD extends Media
{
    protected $table = 'dvds';

    public function getType()
    {
        return 'dvd';
    }
}
```

Database Migrations

- Database migrations are a solution to manage database schema evolutions.
  - Migrations can be compared to version control for a database.
- The migration class contains two methods, `up` and `down`.
  - The `up` method is used to add new tables, columns, or indexes.
  - The `down` method is used to reverse the operations performed by the `up` method.

- **Migrations are not used in LBAW.**
  - Database design is not made through Laravel.

Database Migrations Example

- The following migration creates a `flight` table.

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

return new class extends Migration
{
    // Run the migrations.
    public function up()
    {
        Schema::create('flights', function (Blueprint $table) {
            $table->id();
            $table->string('name');
            $table->string('airline');
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::drop('flights');
    }
};
```

Authentication and Authorization

Authentication

Authentication

- Laravel includes a built-in authentication service.
  - Official documentation at laravel.com/docs/10.x/authentication
- LBAW's template-laravel already includes authentication features that should be adapted to each project.
- See "Manually Authenticating Users" in Laravel's documentation.
  - laravel.com/docs/10.x/authentication#authenticating-users
- Alternatively, a pre-configured quick starter kit is available with Laravel Breeze, which includes a simple implementation of all authentication features, including login, registration, password reset, email verification, and password confirmation.
  - laravel.com/docs/10.x/starter-kits#laravel-breeze

Manual Authentication Setup

- A default Laravel project includes a User model.
  - Disable timestamps in the database with:
  - `public $timestamps  = false;`
  - Add any necessary relationships.
- Required database changes:
  - You need to add a column named `remember_token` to store a token for the “remember me” option. This column must be nullable and support 100 character strings.
- Create LoginController
  - See “Manually Authenticating Users”, laravel.com/docs/10.x/authentication#authenticating-users
- Add authentication routes.

template-laravel Setup

- Authentication is already set up in template-laravel.
  - Under `app/Http/Controllers/Auth`
- You may need to make changes if your table or column names differ from the defaults:
  [users, username, and password].
  - In the User model, set the `$table` property for a different table name.
  - In the LoginController, define a `username` function to return the field name for the username field.
  - In the User model, define a `getAuthPassword` function to return the field name for the password field.
- To support the “Remember me” feature you need to add a `remember_token` column to your users’ table schema.

Authorization

Authorization

- Laravel also provides features to authorize user actions against given resources.
- Two main solutions exist, and both can be used in a single application,
  - **Gates**, can be compared to routes, i.e. closure-based approach.
  - **Policies**, can be compared to controllers, i.e. define logic associated to a model or resource.

- Gates are most applicable to actions not related to any model or resource, e.g. view the admin panel.
- Policies are used to authorize actions for a particular model or resource, and offer a more robust and fine-grained control to define authorization.
- LBAW's template-laravel makes use of policies.

Use Cases

- **Guards (Authentication)**
  - Handle user login/session
  - "Who is the user?"
- **Gates (Global Authorization)**
  - Application-wide rules
  - Not tied to models
  - "Can user do X?"
  - Example: access-admin-panel, manage-settings
- **Policies (Model Authorization)**
  - Model-specific or resource-specific rules
  - CRUD operations
  - "Can user do X to this model?"
  - Example: update a Post, delete a Comment

Gates

Gates

- Gates are a simple way to handle yes/no authorization decisions using closure functions.
- Gates must be registered in `app/Providers/AuthServiceProvider`, the central location for all authorization rules.
- Gates provide a centralized system to manage application permissions, making it easy to modify and maintain authorization rules in one place.
- Gates are ideal for application-wide rules that aren't specifically tied to database models (unlike Policies).
- Gates work like security checkpoints: they receive a user and optionally other parameters, and return true or false to allow or deny access.

Gate Definition

```php
// In App\Providers\AuthServiceProvider
public function boot()
{
    // Simple permission gate
    Gate::define('view-dashboard', function (User $user) {
        return $user->is_admin;
    });

    // Gate with additional parameters
    Gate::define('edit-post', function (User $user, Post $post) {
        return $user->id === $post->user_id;
    });

    // Gate for role-based permissions
    Gate::define('manage-users', function (User $user) {
        return $user->hasRole(['admin', 'moderator']);
    });
}
```

```php
# Use in Controllers

public function viewDashboard()
{
    // Using authorize helper
    if (Gate::allows('view-dashboard')) {
        return view('admin.dashboard');
    }

    abort(403);
}

public function editPost(Post $post)
{
    // Alternative denial check
    if (Gate::denies('edit-post', $post)) {
        abort(403, 'Unauthorized action.');
    }

    return view('posts.edit', ['post' => $post]);
}
```

```blade
@can('view-dashboard')
    <a href="/admin/dashboard">Admin Dashboard</a>
@endcan

@cannot('manage-users')
    <p>You don't have permission to manage users.</p>
@endcannot

@if (Gate::allows('edit-post', $post))
    <a href="{{ route('posts.edit', $post) }}">Edit Post</a>
@endif
```

Policies

Policies

- Policies are defined within the `app/Policies` directory.
- Policies are organized around specific models or resources.
  - E.g., `app/Policies/PostPolicy`
- New policies can be created by hand or automatically created using Artisan.
  - `php artisan make:policy PostPolicy`
  - `php artisan make:policy PostPolicy --model=Post`
- The use of the `model` parameter instructs Artisan to create skeletons for methods related to viewing, creating, updating, and deleting resources.

Registering Policies

- Policies need to be registered to inform Laravel which policy to use when authorizing against a given model type.
- This is done in the `app/Providers/AuthServiceProviders.php` file, setting the `policies` property.
- The following configuration instructs Laravel to use the `PostPolicy` class when authorizing actions against the `Post` Eloquent model.

```
class AuthServiceProvider extends ServiceProvider
{
    /**
     * The policy mappings for the application.
     *
     * @var array
     */
    protected $policies = [
        Post::class => PostPolicy::class,
    ];
...
```

Policy Auto-Discovery

- Instead of manually registering model policies, Laravel can automatically discover policies as long as model and policy follow standard naming conventions.
  - Policies must be placed in the Policies directory.
  - The policy name must match the model name and have “Policy” as a suffix.
  - E.g., `app/Models/Post` and `app/Policies/PostPolicy`.

Policy Methods

- The methods defined in each policy class, define the actions to be authorized.
- For example, an `update` method on the `PostPolicy` class determines if a given `User` can update a given `Post` instance.
  - The `update` method receives a `User` and a `Post` instance as arguments.
  - And should return `true` or `false`.
  - The following code verifies if the user's `id` matches the `user_id` on the post.

```php
public function update(User $user, Post $post)
{
    // A user can update a post if she is the author of that post.
    return $user->id === $post->user_id;
}
```

Authorizing Actions

- Using policies, actions can be authorized at different levels, specifically via:
- The user model, using `can` and `cannot` methods, e.g.:
  - `if ($request->user()->can('update', $post)) { ... }`
- Controller helpers, using the `authorize` method, e.g.:
  - `$this->authorize('create', $item);`
  - `$this->authorize('update', $post);`
- Blade templates, using `@can` and `@cannot` directives, e.g.:
  - `@can('update', $post)`
  - `@elsecan('create', App\Models\Post::class)`

template-laravel example

```php
// app/Http/Controllers/ItemController.php

class ItemController extends Controller
{
  /**
   * Creates a new item.
   *
   * @param  int  $card_id
   * @param  Request request containing the description
   * @return Response
   */
  public function create(Request $request, $card_id)
  {
    $item = new Item();
    $item->card_id = $card_id;

    // Throws an exception if the user is not authorized to create this item.
    $this->authorize('create', $item);

    $item->done = false;
    $item->description = $request->input('description');
    $item->save();

    return $item;
  }
```

```php
//  app/Policies/ItemPolicy.php

use Illuminate\Auth\Access\HandlesAuthorization;

class ItemPolicy
{
    use HandlesAuthorization;

    public function create(User $user, Item $item)
    {
      // User can only create items in cards they own.
      return $user->id == $item->card->user_id;
    }
```

Laravel Ecosystem

Laravel Ecosystem

- Laravel has a mature and dynamic ecosystem.
- Many tools, integrations and packages, both from first-parties and third-parties.

- Laravel Vapor, serverless deployment platform
  - https://vapor.laravel.com
- Laravel Conference
  - https://laracon.eu
- Laravel Bootcamp
  - https://bootcamp.laravel.com
- Laravel Jobs
  - https://larajobs.com

Useful Packages

- **Laravel Debugbar**
  - Debugging toolbar with detailed insights.
  - https://github.com/barryvdh/laravel-debugbar
  - `composer require barryvdh/laravel-debugbar`
- **IDE Helper Generator for Laravel**
  - Generates helper files that enable your IDE to provide accurate autocompletion.
  - https://github.com/barryvdh/laravel-ide-helper
- **Pest | PHP testing framework**
  - https://pestphp.com

Additional Topics

- Much more to explore and use. Some highlights.
- Artisan Console
  - https://laravel.com/docs/10.x/artisan
- Data Validation
  - https://laravel.com/docs/10.x/validation
- Testing
  - https://laravel.com/docs/10.x/testing
- Telescope Package
  - https://laravel.com/docs/10.x/telescope

LBAW ‘template-laravel’

LBAW ‘template-laravel’

- A sample Laravel project is provided with template-laravel.
  - https://gitlab.up.pt/lbaw/template-laravel
- This project implements a simple task manager called Thingy!
- Is expected to be used as a starting point for the development of the prototype.
  - User registration and authentication is already included and can be adapted.
  - Thingy! specific features need to be removed.

Publishing the Project

- The project is published using Docker.
  - A container is built using the provided Dockerfile.
  - The container is uploaded to the group's Gitlab Container Registry
    - https://gitlab.up.pt/lbaw/lbawYYYY/lbawYYGG/container_registry

- The script `upload_image.sh` handles the details:
  - https://gitlab.up.pt/lbaw/template-laravel/-/blob/master/upload_image.sh

Project Container Dockerfile

```
FROM ubuntu:24.04

# Install dependencies
ENV DEBIAN_FRONTEND=noninteractive
RUN apt-get update
RUN apt-get install -y --no-install-recommends libpq-dev vim nginx php8.3-fpm php8.3-mbstring php8.3-xml php8.3-pgsql
php8.3-curl ca-certificates

# Copy project code and install project dependencies
COPY --chown=www-data . /var/www/

# Copy project configurations
COPY ./etc/php/php.ini /usr/local/etc/php/conf.d/php.ini
COPY ./etc/nginx/default.conf /etc/nginx/sites-enabled/default
COPY .env.production /var/www/.env
COPY docker_run.sh /docker_run.sh

# Start command
CMD sh /docker_run.sh
```

References

- Laravel: Up & Running, Matt Stauffer. O'Reilly, 2019
- Laravel Official Documentation (version 10.x), https://laravel.com/docs/10.x