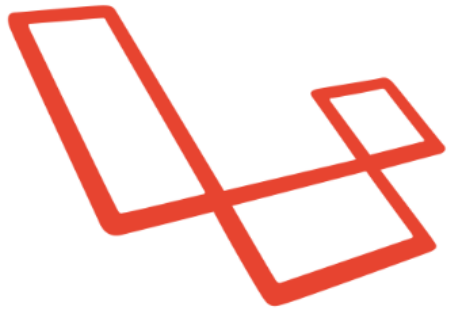


Laravel

#2
Handling
Requests and Events

write@robertogallea.com | http://www.robertogallea.com

Worksop outline

- Model binding
- Authorization scaffolding
- Custom Request handlers
- Middlewares
- Using Events

Model binding

Automatically injecting models in controllers' actions

Model binding

- If route parameter is type hinted, its value considered as the row id key, and model is automatically loaded

```php
public function destroy($post)
{
  $post = Post::find($post);
  $post->delete();
  return redirect('/');
}
```

```php
public function destroy(Post $post) {
    $post->delete();

    return redirect('/');
}
```

Model binding

- What if… I don’t want to show id in links?
- Example: use Uuid (uuid4) → random alphanumeric string, can (and should) supposed as universally unique*
- Use a key different than primaryKey, by overriding model method **getRouteKeyName()**

Add migration for uuid

php artisan make:migration add_uuid_to_posts

```
class AddUuidToPosts extends Migration
{
    public function up()
    {
        Schema::table('posts', function($table) {
            $table->string('uuid',36)->unique();
        });
    }

     public function down()
    {
        Schema::table('posts', function($table) {
            $table->dropColumn('uuid');
        });
    }
}
```

```php
// inside PostController@store
$post->uuid = Uuid::uuid4();

// inside post list view
<form method="post" action="/post/{{ $post->uuid }}">

// inside Post.php
public function getRouteKeyName()
{
  return 'uuid';
}
```

Authorization scaffolding

Gate definition

Define access to resources

- Out-of-the-box authorization layer
- Gates
  - Test if certain user is allowed to access some resource
- **Example:** a post can be deleted only by its creator
  - Condition: author_id === authenticated_user_id

Gate definition

- For being used, Gates must be defined
- Generally, they are defined in Service Providers

***Service Provider**: Piece of code that «bootstraps» application core components*

- Define Gates in AuthServiceProvider

```
use Illuminate\Support\Facades\Gate;

public function boot()
{
    $this->registerPolicies();

    Gate::define('delete-post', function ($user, $post) {
       return $user->id === $post->user_id;
    });
}
```

Gate usage

- Use them inside Controllers to test if user can access the resource

```php
public function destroy(Post $post)
{
    if (Gate::forUser(Auth::user())->allows('delete-post', $post)) {
        $post->delete();
        return redirect('/');
    }
    else {
        redirect('/')->withErrors(['msg' => 'Not allowed']);
    }
}
```

- Note: For testing current authenticated user
  Gate::allows('delete-post', $post)
  Auth::user()->can('delete-post',$post)
- Note: *Gate::denies('delete-post',$post)* is the inverse

Gate usage

- Conditionally render view according to gates

```
{{-- inside post list table --}}
<td>
    @can('delete-post',$post)
        <form method="post" action="/post/{{ $post->uuid }}">
            {{ csrf_field() }}
            {{ method_field('DELETE') }}
            <button type="submit">Delete post</button>
        </form>
    @else
        <button type="button" disabled>Delete post</button>
    @endcan
</td>
```

Custom Requests

Decoupling input validation from controllers

Custom requests

- Used for moving data validation logic outside of controller
- Cleaner!

Defining custom request

- Run *php artisan make:request StorePostRequest*
- Run *php artisan make:request DeletePostRequest*

- Creates app/Http/Requests/StorePostRequest.php and DeletePostRequest.php
- Four methods:
  - authorize() – to determine whether the user can make the request
  - rules() – rules to validate the inputs
  - messages() – message the error messages for input validation
  - attributes() – how to name attributes' names in validation errors

StorePostRequest

- StorePostRequest:

```php
class StorePostRequest extends FormRequest
{
    public function authorize()
    {
        return true;
    }

    public function rules()
    {
        return [
            'title' => 'required|max:100',
            'content' => 'required',
        ];
    }
}
```

StorePostRequest

- Change store() action in PostController:

```
public function store(StorePostRequest $request) {
    $post = new Post();
    $post->uuid = Uuid::uuid4();
    $post->title = $request->title;
    $post->content = $request->content;
    $post->author_id = Auth::user()->id;
    $post->save();

    return redirect('/');
}
```

DeletePostRequest

- DeletePostRequest:

```php
public function authorize()
{
    return $this->route('post') && $this->user()->can('delete-post', $this->route('post'));
}

public function rules()
{
    return [
        //
    ];
}
```

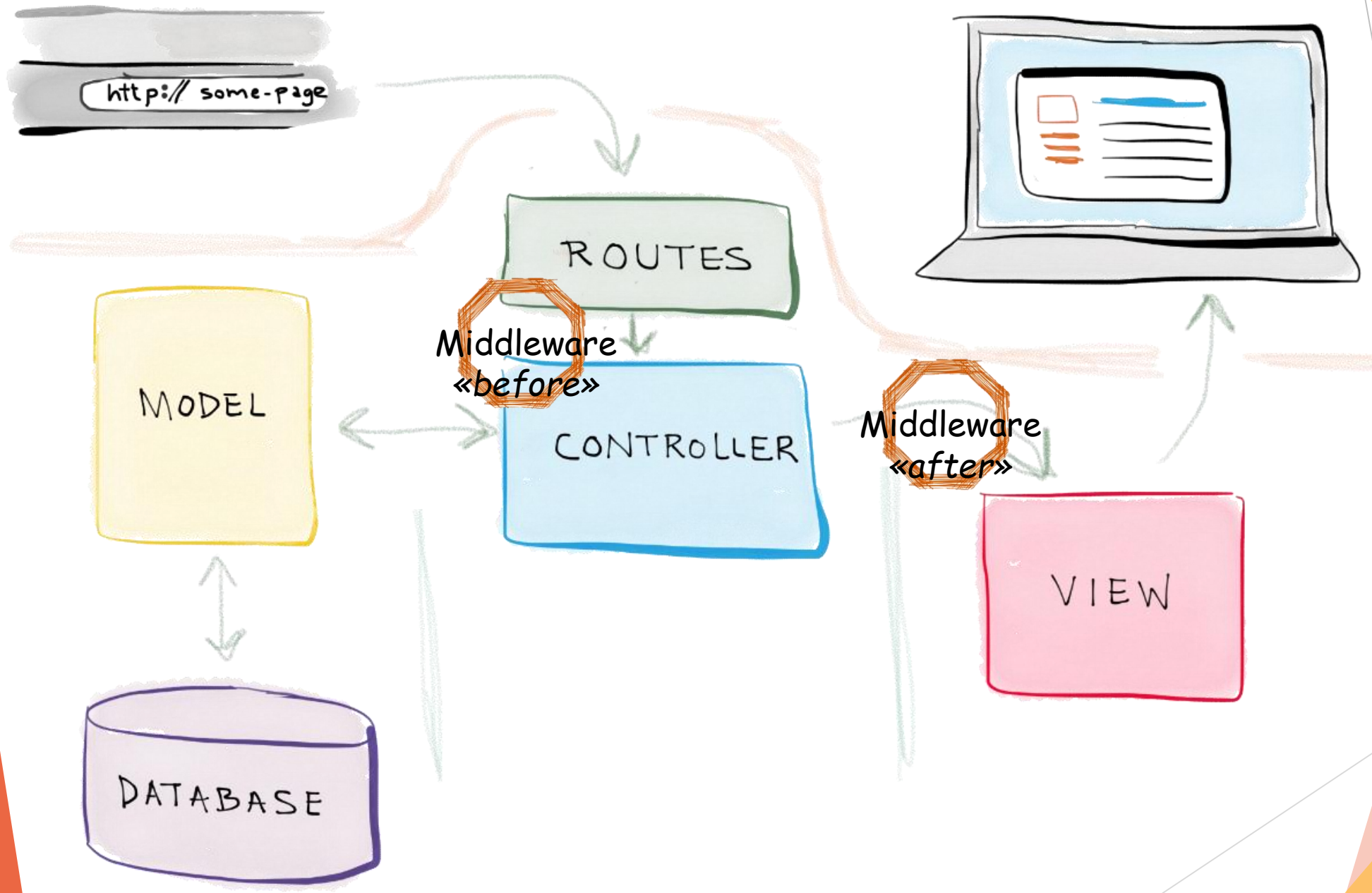
DeletePostRequest

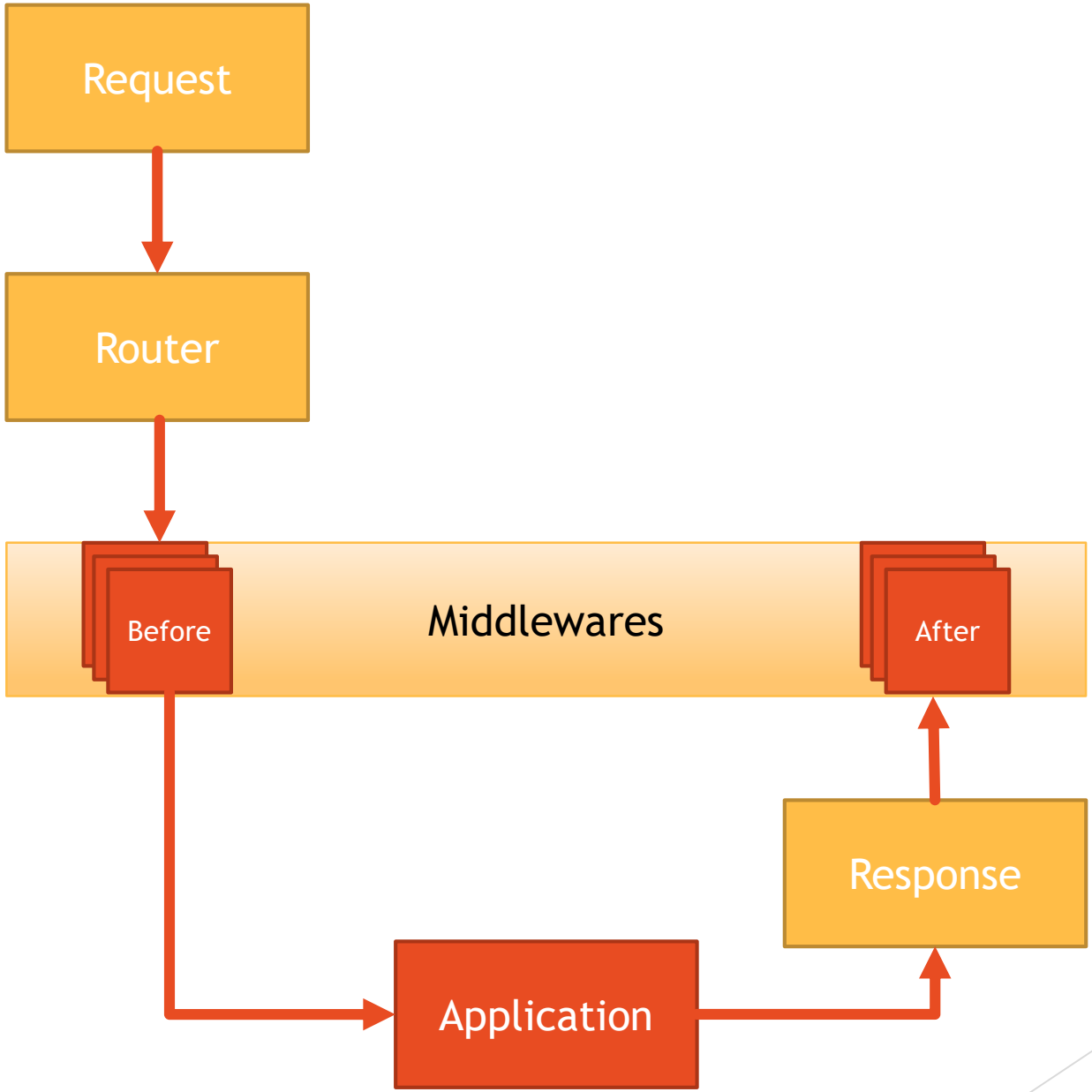
- Change destroy() action in PostController:

```php
public function destroy(DeletePostRequest $request, Post $post)
{
    $post->delete();
    return redirect('/');
}
```

Middlewares

Filtering requests and responses


http:// some-page
ROUTES
Middleware
«before»
MODEL
CONTROLLER
Middleware
«after»
VIEW
DATABASE


Request
Router
Before
Middlewares
After
Response
Application

Middlewares

- Middlewares are bootstrapped with laravel core code
- Loaded middleware are listed in **app/Http/Kernel.php**

- **$middleware = […]** → Middlewares that are ALWAYS ran during every request
- **$middlewareGroups = […]** → Middlewares grouped under a common label to apply them all together at once
- **$routeMiddleware = […]** → Middleware that can be applied to routes

Some out-of-the box middlewares

- CheckForMaintenanceMode → allows to disable application (*php artisan down|up*)
- ValidatePostSize → Check max post size against php.ini settings
- TrimStrings → remove any empty spaces before and/or after strings
- ConvertEmptyStringsToNull → convert empty strings to null values
- TrustProxies → Handles trusted proxies (load balancers, etc…)

Applying route middlewares

- Chain ->middleware([…]) after each route
- Example, constrain a single route to be accessible only to authenticated users:

```
Route::get('/test', function () {
    return 'ok';
})->middleware(['auth']);
```

Defining «before» middlewares

- Define a middleware allowing to access resource only on seconds multiple of four (!!!)

```
php artisan make:middleware AllowEveryFourSeconds
```

- Add it to App/Http/Kernel.php

```
protected $routeMiddleware = [
    ...
    ‘four_seconds’ => \App\Http\Middleware\AllowEveryFourSeconds::class,
];
```

- Apply to test route

```
Route::get('/test', function () {
    return 'ok';
})->middleware(['four_seconds']);
```

```php
class AllowEveryFourSeconds
{
    /**
     * Handle an incoming request.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \Closure  $next
     * @return mixed
     */
    public function handle($request, Closure $next)
    {
        $time = Carbon::now();
        if (($time->second % 4) !== 0)
            abort(403,'Time is ' . $time->format('H:i:s'));
        return $next($request);
    }
}
```

Defining «after» middlewares

- Define a middleware inverting the content of the response

  php artisan make:middleware InvertResponse

- Add it to App/Http/Kernel.php

```
protected $routeMiddleware = [
    ...
    ‘invert’ => \App\Http\Middleware\InvertResponse::class,
];
```

- Apply to test route

```
Route::get('/test2', function () {
    return 'ok';
})->middleware([‘invert']);
```

```php
<?php

namespace App\Http\Middleware;

use Closure;

class InvertResponse
{
    /**
     * Handle an incoming request.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \Closure  $next
     * @return mixed
     */
    public function handle($request, Closure $next)
    {
        $response = $next($request);
        $response->setContent(strrev($response->getContent()));
        return $response;
    }
}
```

Using events

Subscriber/Listener Model

Events model

- Basic Observer implementation
- Suscribe/listen to events

- Definition of
  - Events – The event that are generated
  - Listeners – The observers for the events

- Decoupling of application logic (multiple listeners for same event)

Defining events

- Open App\Http\Providers\EventServiceProvider.php

```
protected $listen = [
    'App\Events\Event' => [
        'App\Listeners\EventListener',
    ],
];
```

- Define you events and listeners

php artisan event:generate

- Creates missing events and listeners under App\Events and App\Listeners respectively

Event Example

- On blog deletion write a log entry
- On blog deletion write its content to disk

```
protected $listen = [
    'App\Events\Event' => [
        'App\Listeners\EventListener',
    ],
    'App\Events\PostDeletedEvent' => [
        'App\Listeners\LogDeletedPost',
        'App\Listeners\BackupDeletedPost',
    ]
];
```

php artisan event:generate

- PostDeletedEvent

```
public function __construct(Post $post)
{
    $this->post = $post;
}
```

- LogDeletedPost

```
public function handle(PostDeletedEvent $event)
{
    Log::info('Post deleted: ' . $event->post);
}
```

- BackupDeletedPost

```
public function handle(PostDeletedEvent $event)
{
    Storage::put('post_' . $event->post->id . '.txt', $event->post->content);
}
```

Stopping event propagation

- Suppose you want to stop event propagation for posts having content longer than 10 characters

- Return false to stop propagation

```
public function handle(PostDeletedEvent $event)
{
    Log::info('Post deleted: ' . $event->post);

    if (strlen($event->post->content) > 10)
        return false;
}
```

Thank you!

What's next…

- ORM data manipulation
- Route definition techniques
- Url generation
- Internationalization
- Resources
- Model Policies
- …