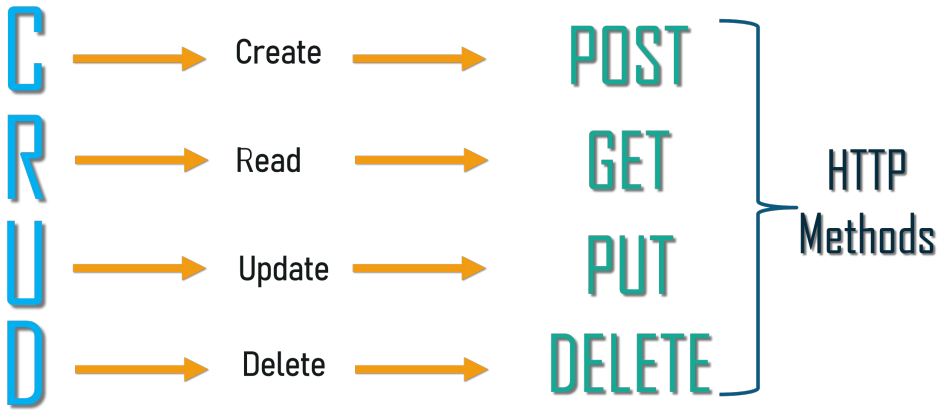
Faculty of Applied Science
Information Technology
2023-24 Fall Semester

Web Technologies
05 - Request and Blade Templating

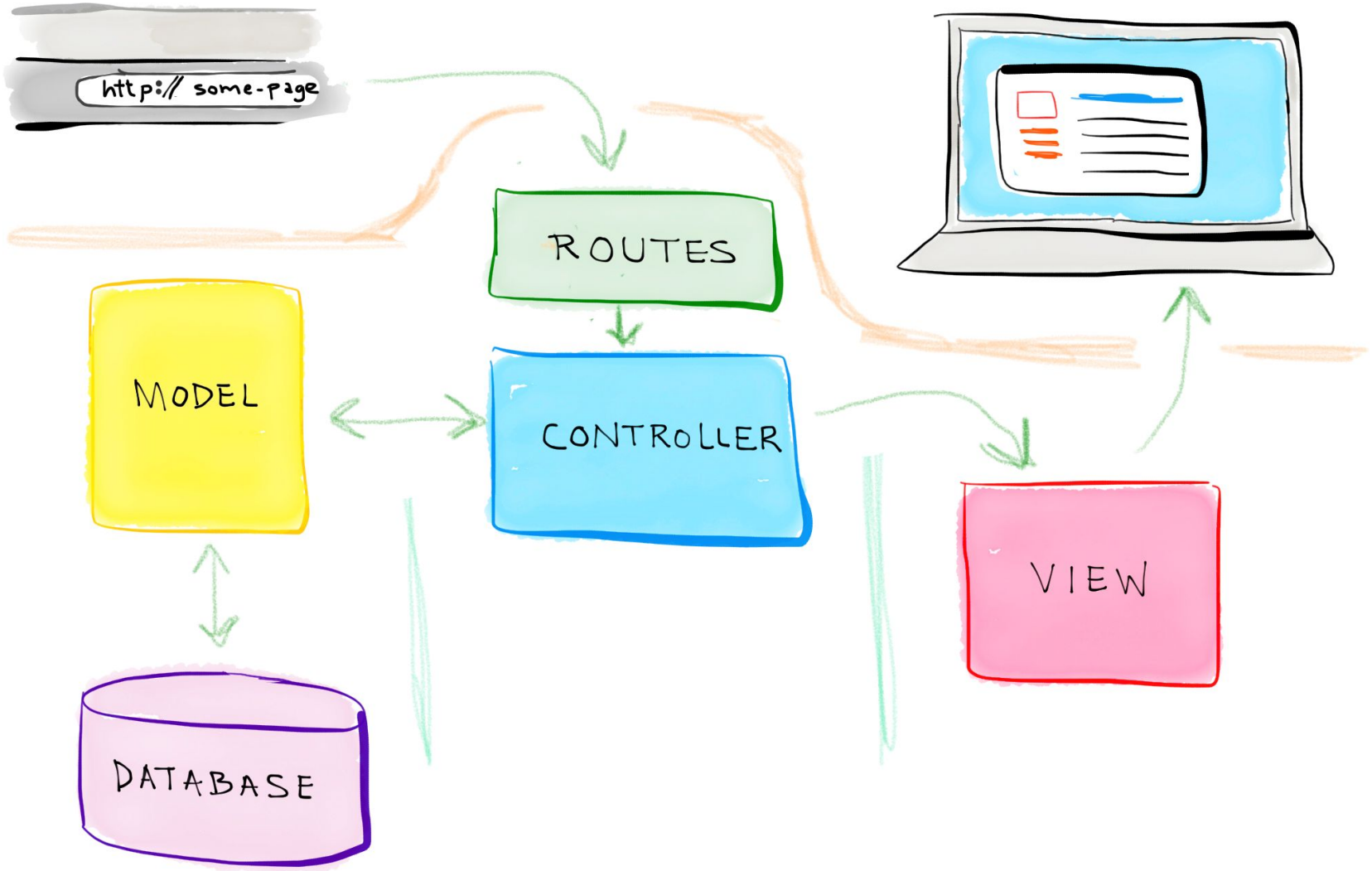
Previous Lecture

- Pass Request Data to Views
- What Is MVC?
- HTTP Verbs
- REST
- Controllers



Content

- Collecting and Handling User Data
- Validation
- Blade Templating
- Template Inheritance


http:// some-page
ROUTES
MODEL
CONTROLLER
VIEW
DATABASE

Collecting and Handling User Data

Websites that benefit from a framework like Laravel often don't just serve static content. Many deal with complex and mixed data sources, and one of the most common (and most complex) of these sources is user input in its basic forms: URL paths, query parameters, POST data, and file uploads.

Laravel provides a collection of tools for gathering, validating, normalizing, and filtering user-provided data. We'll look at those here.

Collecting and Handling User Data

The most common tool for accessing user data in Laravel is injecting an instance of the *Illuminate\Http\Request* object. It provides easy access to all of the ways users can provide input to your site: POST, posted JSON, GET (query parameters), and URL segments.

```
public function store(Request $request){

}
```

Collecting and Handling User Data

$request->all()

Just like the name suggests, $request->all() gives you an array containing all of the input the user has provided, from every source. Let's say, for some reason, you decided to have ...

Collecting and Handling User Data

$request->except() and ***$request->only()***

$request->except() provides the same output as $request->all(), but you can choose one or more fields to exclude—for example, _token. You can pass it either a string or an array of strings.

$request->only() is the inverse of ***$request->except()***, you will only get the field that you specify in the ***only()*** method.

Collecting and Handling User Data

$request->has()

With ***$request->has()*** you can detect whether a particular piece of user input is available to you.

$request->method()

returns the HTTP verb for the request, and ***$request- >isMethod()*** checks whether it matches the specified verb

Validation

Laravel has quite a few ways you can validate incoming data. We'll cover form requests in the next section, so that leaves us with two primary options: validating manually or using the ***validate()*** method on the Request object. Let's start with the simpler, and more common, validate().

Validation

validate() on the Request Object

The Request object has a ***validate()*** method that provides a convenient shortcut for the most common validation workflow.

```
public function store(Request $request)
{
    $request->validate([
        'title' => 'required|unique:recipes|max:125',
        'body' => 'required'
    ]);

    // Recipe is valid; proceed to save it
}
```

Validation

We only have four lines of code running our validation here, but they're doing a lot.

First, we explicitly define the fields we expect and apply rules **(here separated by the pipe character, |)** to each individually.

Next, the validate() method checks the incoming data from the $request and determines whether or not it is valid.

If the data is valid, the validate() method ends and we can move on with the controller method, saving the data or whatever else.

Validation

But if the data isn’t valid, it throws a ValidationException. This contains instruc tions to the router about how to handle this exception.

In our examples here we’re using the **“pipe”**

syntax: 'fieldname': 'rule|otherRule|anotherRule'.

But you can also use the array syntax to do the same thing:

'fieldname': ['rule' , 'otherRule' , 'anotherRule'].

Validation Rules

https://laravel.com/docs/10.x/validation

Validation

The ***validate()*** method on requests (and the withErrors() method on redirects that it relies on) flashes any errors to the session. These errors are made available to the view you're being redirected to in the ***$errors*** variable. And remember that as a part of Laravel's magic, that ***$errors*** variable will be available every time you load the view, even if it's just empty, so you don't have to check if it exists with ***isset()***.

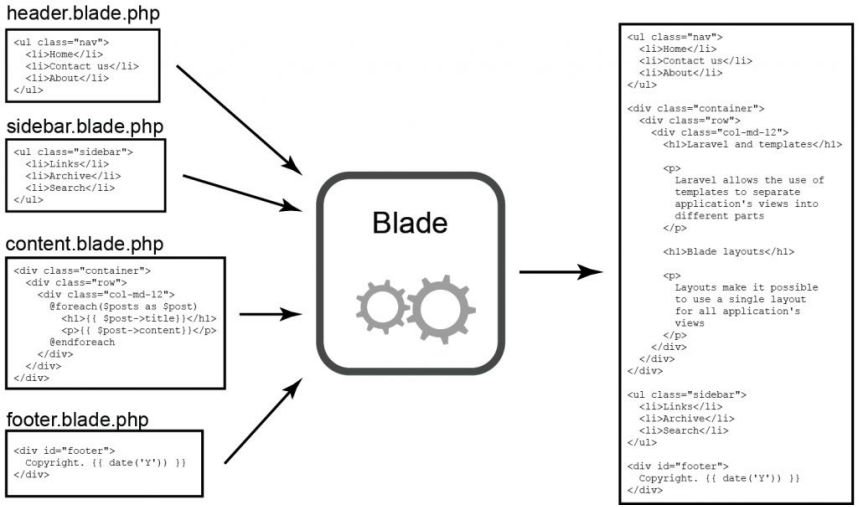
Validation

```
@if ($errors->any())
    <ul id="errors">
        @foreach ($errors->all() as $error)
            <li>{{ $error }}</li>
        @endforeach
    </ul>
@endif
```

Break Time!

Blade Templating

Laravel offers a custom templating engine called Blade, which is inspired by .NET's Razor engine. It boasts a concise syntax, a shallow learning curve, a powerful and intuitive inheritance model, and easy extensibility.



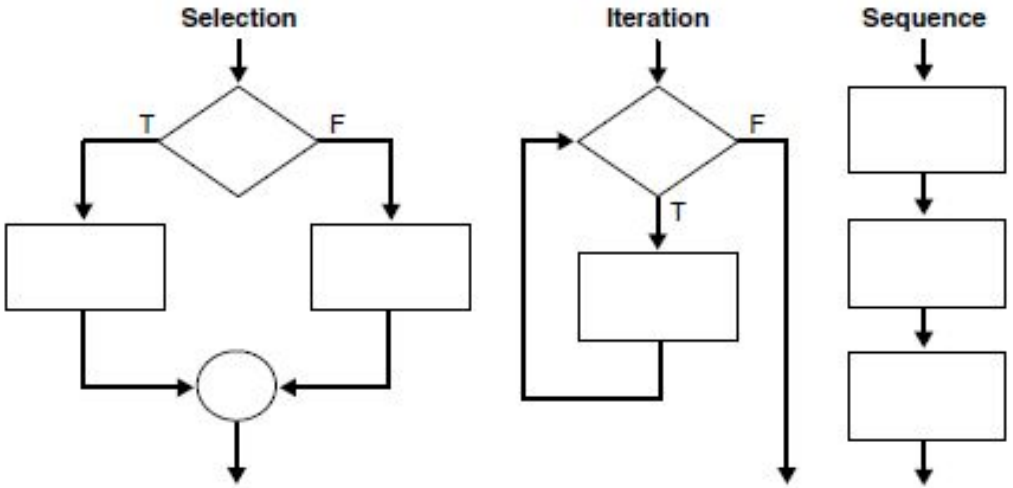
Echoing Data

Blade uses curly braces for its “echo” {{ and }} are used to wrap sections of PHP that you’d like to echo. **{{ $variable }}** is similar to **<?php echo $variable ?>** in plain PHP.

```
<h1>{{ $group->title }}</h1>
{!! $group->heroImageHtml() !!}
```

Control Structures

Most of the control structures in Blade will be very familiar. Many directly echo the name and structure of the same tag in PHP. There are a few convenience helpers, but in general, the control structures just look cleaner than they would in PHP.



Conditionals

```
@if (count($talks) === 1)
    There is one talk at this time period.
@elseif (count($talks) === 0)
    There are no talks at this time period.
@else
    There are {{ count($talks) }} talks at this time period.
@endif
```

Loops

```
@for ($i = 0; $i < $talk->slotsCount(); $i++)
    The number is {{ $i }}<br>
@endfor

  @while ($item = array_pop($items))
      {{ $item->orSomething() }}<br>
  @endwhile
```

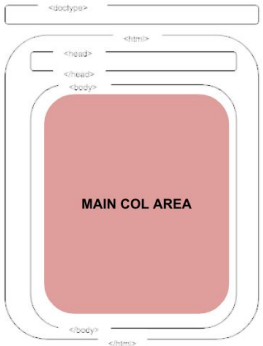
Loops

```
@foreach ($talks as $talk)
    • {{ $talk->title }} ({{ $talk->length }} minutes)<br>
@endforeach
```

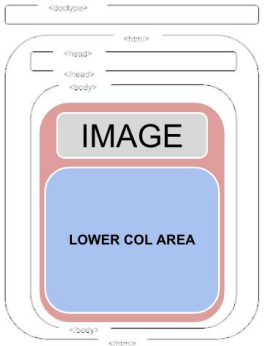
Template Inheritance

Two of the primary benefits of using Blade are template inheritance and sections. To get started, First, we will examine a "master" page layout. Since most web applications maintain the same general layout across various pages, it's convenient to define this layout as a single Blade view:

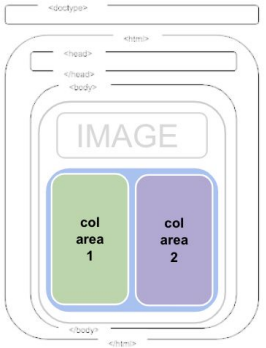
Parent template



Child template



Sub-child template



```
<!-- Stored in 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>
```

Template Inheritance

As you can see, this file contains typical HTML markup. However, take note of the @section and @yield directives. The @section directive, as the name implies, defines a section of content, while the @yield directive is used to display the contents of a given section.

Extending A Layout

When defining a child view, use the Blade @extends directive to specify which layout the child view should "inherit".

Views which extend a Blade layout may inject content into the layout's sections using @section directives. Remember, as seen in the example above, the contents of these sections will be displayed in the layout using @yield:

```
<!-- Stored in resources/views/child.blade.php -->

@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
```

Extending A Layout

In this example, the sidebar section is utilizing the @parent directive to append (rather than overwriting) content to the layout's sidebar. The @parent directive will be replaced by the content of the layout when the view is rendered.

For more details on topics of this lecture:
Read Chapter 04

Lab Task

- Make a controller with resources(all 7 functions)
- Make a route (in accordance with REST) for each function.
- Make a view for creating a new resource and write the necessary code for the form.
- Write the necessary code in the store() function to validate the data from the form and show the error messages if there were any validation errors.
- Send all the data that was received from the form to another view and show all the data with some styling.

Activities and Next Week Topics

This Week:

- Read Chapter 03 of Laravel: Up & Running, for more information on MVC,HTTP Verbs and REST.
- Practice different route and passing data through routes.
- Recreate last weeks assignment with controllers and use the blade templating elements.
- Create a Github Student Account.
- Download and install git-fork.com

Next Week:

- Databases and Eloquent.

References / Further Readings

- Matt Stauffer, 2019. Laravel: Up & Running: A Framework for Building Modern PHP Apps. O'Reilly Media.
- Laravel.com : Laravel's official Documentation.
- Dayle Rees, 2016. Laravel: Code Smart.