ICOM 6034
Website engineering

Dr. Roy Ho
Department of Computer Science, HKU

Session 5: The server side - Part I

Web 2.0: summary

- Functionality: Rich Internet Applications or RIAs with complicated UI and logic
- Socialization: tools for publishing and sharing user-generated data and metadata
- Data: abilities to (re)use remote data to form new information or added value

Simplified by web development frameworks [covered in Part 2 of this course]

Simplified by standard data formats and Web API protocols – also supported by some dev. frameworks [covered in Part 3]

- These three elements can further be combined to form powerful and user-friendly web 2.0 applications.
  - An RIA can be built by fetching data from multiple websites, some of these websites might deliver user-generated data or metadata

Part 2: “Functionality” and “socialization”

“Functionality” in Web 2.0

- RIA user interfaces (Session 4):
  - AJAX
  - Client-side libraries/frameworks, e.g., jQuery
- Sophisticated application logic (Sessions 5-6):
  - Server-side frameworks, e.g., Laravel and Ruby on Rails

“Socialization” in Web 2.0

- User-generated contents (Session 6):
  - Drupal
  - Wiki

Objectives

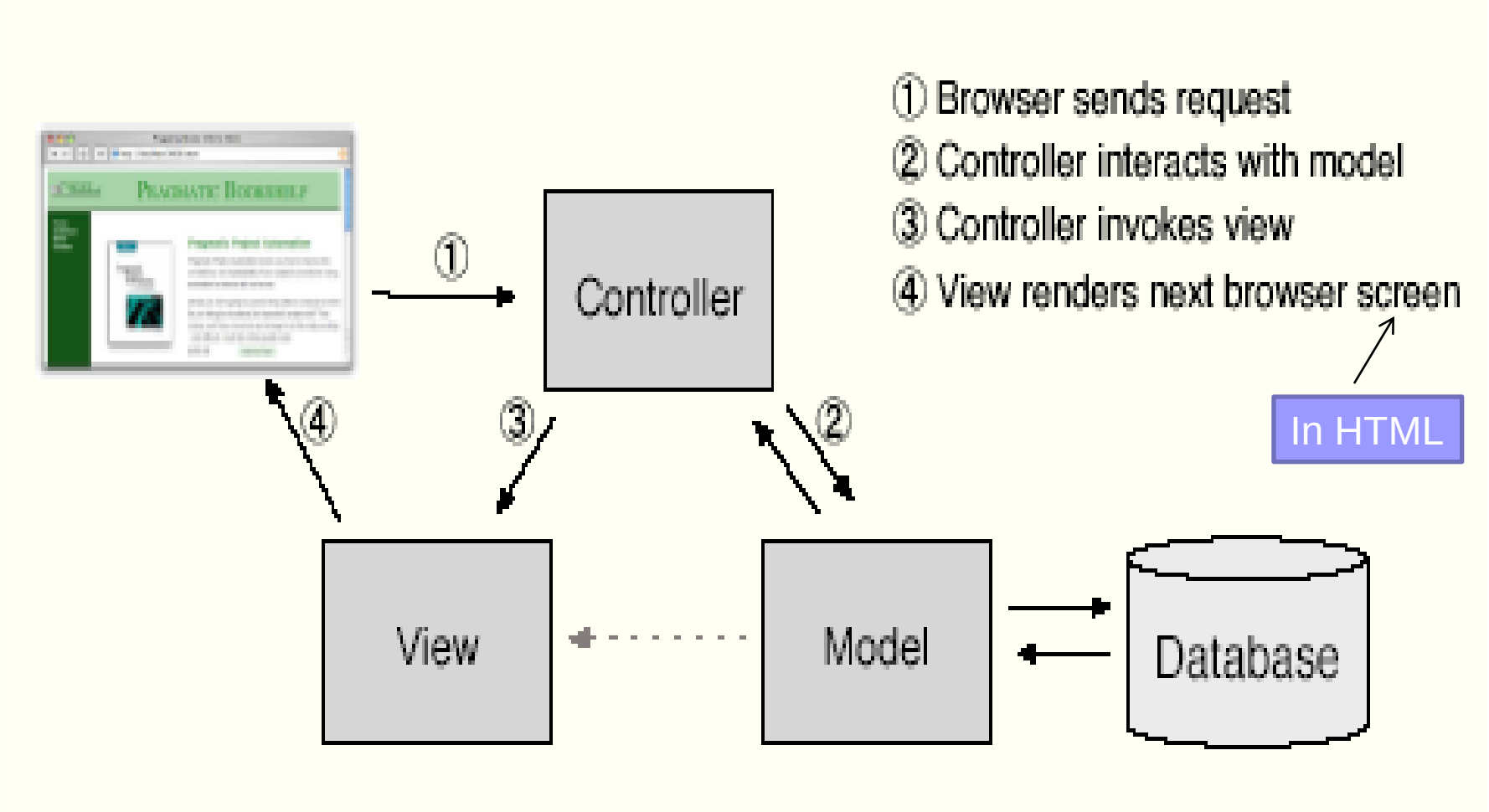
- The Model-View-Controller (MVC) design pattern of web applications
- Introduction to Laravel: a server-side web application framework
  - Routing and controllers
  - The Blade templating engine
  - Database access, Eloquent Model and ORM
  - Form validations
  - User authentication
- Lab 3A: Laravel Framework

The MVC design pattern

Model – View – Controller (MVC) design pattern

- A pattern for organizing program codes into three components
  - Commonly used for designing interactive applications (i.e., mostly with GUIs)
- Model
  - Programs that access and manage the data
  - Connections and manipulations of databases
  - Representation of real world objects (e.g., “students”, “teachers”, etc. in an intranet application) based on the low-level data (e.g., database records)
- View
  - Programs and templates (or “layouts”) for presenting/formatting the data in the browsers (i.e., HTML/JS/CSS and the programs for generating these)
- Controller
  - Receives all input events (requests) from user
  - Decides what they mean & what to do
  - Your application logic
  - Connects model and view

MVC


① Browser sends request
② Controller interacts with model
③ Controller invokes view
④ View renders next browser screen
In HTML
Controller
View
Model
Database
①
②
③
④

MVC benefits

- **Separation** of concerns at the server side
  - Application logic -> Controller
  - Data access routines and other data-related logic (e.g., data modelling) -> Model
  - Presentational/formatting details -> View

- Separations facilitate application extensibility. Examples:
  - Easy to add a new view (e.g., for phones/tablets/smart-watches, etc.) for the same application in the future
  - Can add new model info (e.g., HKU SPACE students in addition to HKU students), while old views still work
  - Can modify application logic (e.g., Controllers) without affecting the other components
    - E.g., Model for DB access (which may depend on/affect backend systems), View for display, etc.

- Supplement the client-side separations we have discussed before:
  - CSS: "presentation details" separated from document structure
  - JavaScript libraries/frameworks (e.g., jQuery): "behaviors" separated from document structure
  - (Client-side) data validation/presentation routines for i18n separated from document structure

Result: MVC structures application so that it is scalable and maintainable

Popular MVC frameworks at the server side

- PHP: Laravel, Symfony, CodeIgniter, CakePHP, etc.
- Java: Java Server Faces, Apache Struts, Spring Framework, etc.
- JavaScript: Express on Node.js
- Ruby: Ruby on Rails
- Microsoft: ASP.NET MVC
- Many others…

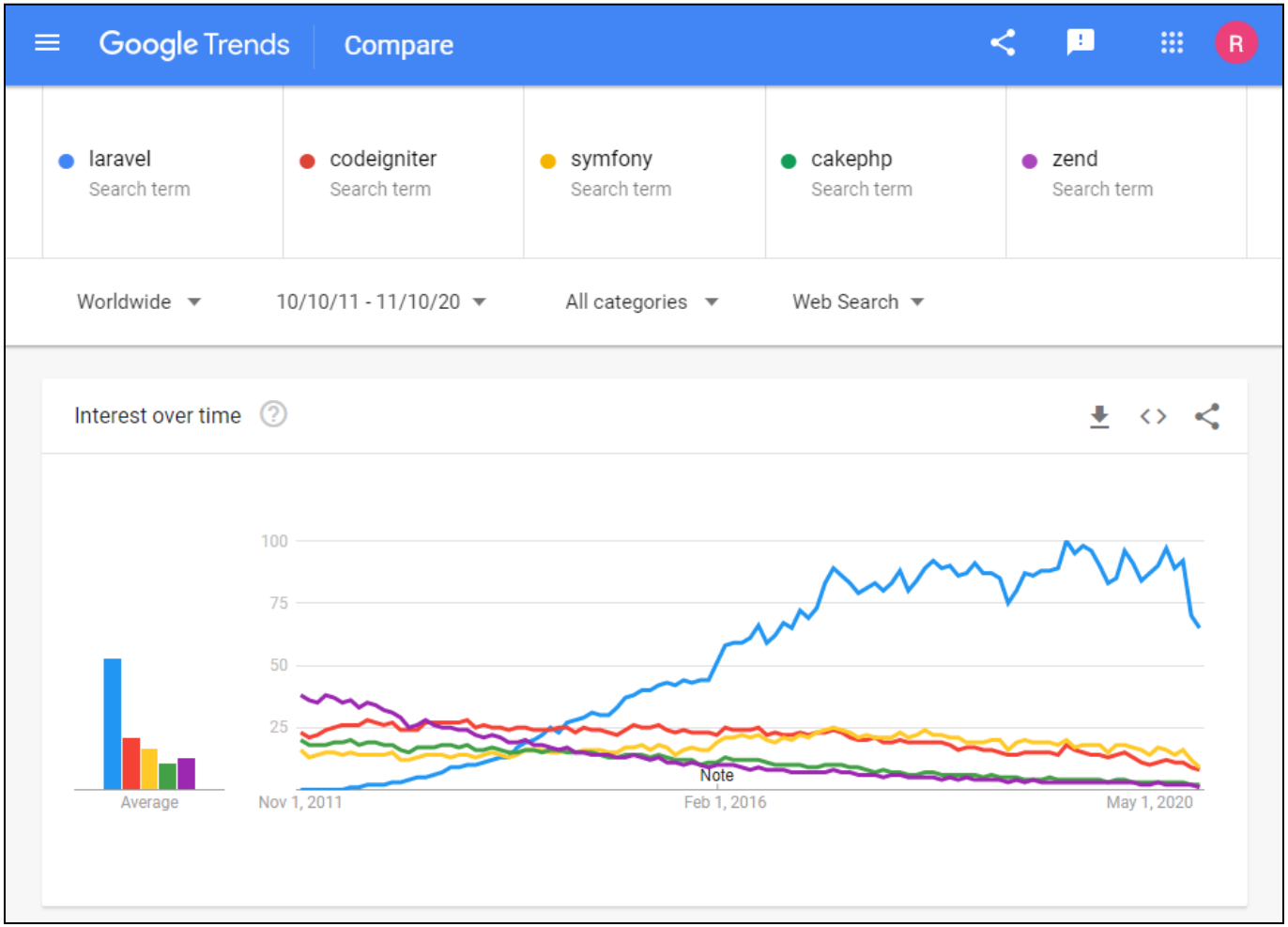
Laravel: the framework, routing and Controllers

(Some examples were taken from M. Stauffer, Laravel: Up and Running, O'Reilly Media 2016)

Laravel Framework

- An open source, MVC-based PHP framework
  - First released in 2011
  - Quickly became popular in just 4 years after its launch
  - Current version is 8
    - But Version 6 is the latest version with **LTS** (long-term support) and is more stable, so we will use Version 6 in the labs/assignment. Version 8 is similar.
  - Developed on top of Symfony, another open-source PHP framework
- Designed for rapid development of websites/apps
  - Easy to learn, productive, clean, extensible
  - Many tools and libraries available
  - Makes common and tedious tasks extremely simple, e.g., form validation, user authentication, etc.
    - E.g., simple user authentication can be implemented with only a few lines of code

Laravel Framework



- An active developer community
- Can also make use of the PHP community and other tools

Installation

- Requirements: PHP >= 7.2 with PDO and other essential PHP extensions, e.g., OpenSSL
  - Have been installed with XAMPP

- Laravel can be installed by Composer, a package management tool for PHP
  - Once Composer is installed, the following command can install the Laravel installer:
    - composer global require laravel/installer

  - Then, the "laravel new" command can be used to create the skeleton of a new Laravel application for development:
    - laravel new projectName
    - (A new folder "projectName" would be created in the current directory together with skeleton files for the new application)

  - Alternatively, the following single command will install Laravel **and** create an application called projectName in the current directory:
    - composer create-project --prefer-dist laravel/laravel projectName

Running and testing

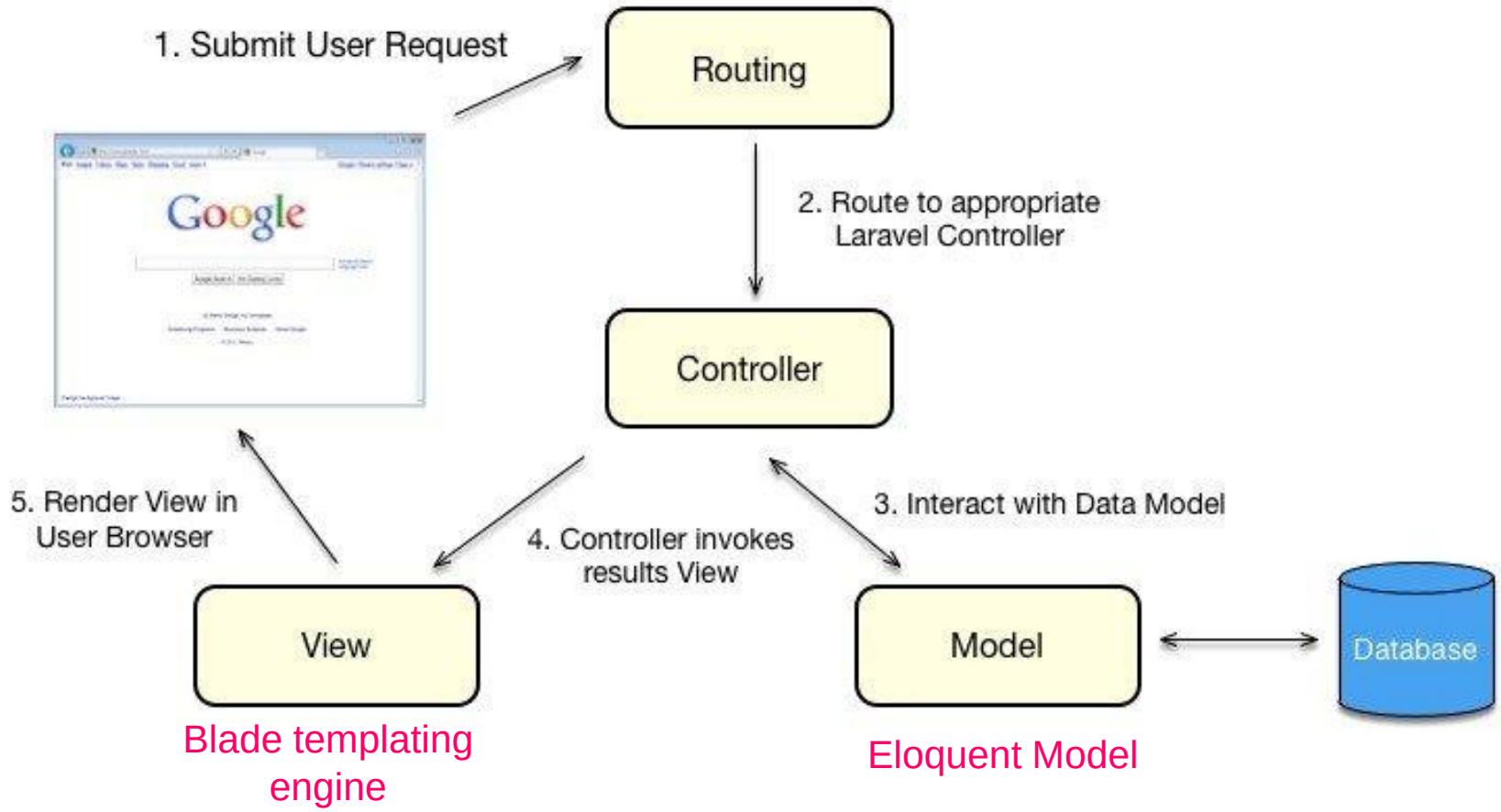
- Running/testing your application
  - If a web server (e.g., Apache in XAMPP) is available, you may point its DocumentRoot (by modifying httpd.conf) to the “public” folder of your Laravel application, then start the web server
  - Alternatively, PHP has a lightweight web server for testing. Simply go to your Laravel application folder and type:
    - php artisan serve
  - Artisan is the command-line interface of Laravel, which can be used to create skeletons of Controllers, Models, etc., among other functions.

The welcome page of a newly-created application.

The folder structure

app	Our application workspace: controllers, models, ...
bootstrap	Bootstrap files like classes auto-loading
config	Application configuration: authentication, database, ...
database	Database migrations and seeds
public	Public items, like *robots.txt* and *index.php*
resources	Views, template's assets and i18n files
routes	Application's routes: web, api, ...
storage	Storage folder for sessions, files, etc.
tests	Application tests: unit, integration, ...
.env.example	Environment parameters
artisan	Swiss army knife for multiple purposes
composer.json	Project dependencies (PHP packages)
gulpfile.js	Task-manager
package.json	Project dependencies (Javascript packages)
phpunit.xml	Configuration for testing environment
server.php	Local server for development only

MVC in Laravel



Simple routing (without a controller)

routes/web.php is the route definition file.

You define all your routes for web requests here.

Destination of requests

All GET requests sent to "/" (e.g., http://domain.com/) would be handled by this route

:: (double colon) is PHP's scope resolution operator, which enables access to static properties or methods of a class.

```
// routes/web.php
Route::get('/', function () {
    return 'Hello, World!';
});
```

HTTP method

Can be "get", "post", "put", "delete", etc.

More on "put" and "delete" in Part 3.

An anonymous function, also called a "closure" in PHP

This function simply prints "Hello, World!" to the user

Result: a GET request sent to http://domain.com/ would obtain 'Hello, World!'.

More examples of routing

Route parameters

```
Route::get('users/{id}/friends', function ($id) {
    //
});
```

Routing to the "show" method of the "MembersController" class (more on this later…)

```
// Defining a route with name in routes/web.php:
Route::get('members/{id}', 'MembersController@show')->name('members.show');

// Link the route in a view using the route() helper
<a href="<?php echo route('members.show', ['id' => 14]); ?>">
```

Route names can be used elsewhere in your application to reproduce the path of a route. In this example: http://domain.com/members/14

Defining the name of this route

A route group sharing the same path prefix:

```
Route::group(['prefix' => 'api'], function () {
    Route::get('/', function () {
        // Handles the path /api
    });
    Route::get('users', function () {
        // Handles the path /api/users
    });
});
```

Sub-domain routing

```
Route::group(['domain' => 'api.myapp.com'], function () {
    Route::get('/', function () {
        //
    });
});
```

Controllers and action methods

- Controllers (or Controller classes):
  - The application's "endpoints" for receiving requests
  - Contain the main "logic" of your application
  - Usually, a controller groups all actions related to a particular object (i.e., "noun") together
    - E.g., a "BookController" class might have actions like "show", "edit", "create", "destroy", etc.
  - Interact with Models and provide data to Views
  - Return the appropriate Views to the clients

- A controller action method is a method of a controller class that handles a specific request

- After controllers (and their action methods) are defined, routes can be defined for correct routing of requests:

In routes/web.php:

```
Route::get('/', 'TasksController@home');
```

Result: all "GET" requests sent to / will be routed to the home method of the TasksController, and will receive "Hello, World!".

Instead of a string (e.g., "Hello, World!"), an action method normally returns a View (e.g., return view('home');) to clients

```
php artisan make:controller TasksController
```

Create a skeleton of the TasksController class:

app/Http/Controllers/TasksController.php

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

use App\Http\Requests;

class TasksController extends Controller
{
}
```

Let's add a "home()" action method…

```
class TasksController extends Controller
{
    public function home()
    {
        return 'Hello, World!';
    }
}
```

Views

- "Views" provide HTML to the clients; there are three types of "views" in Laravel:
  - Static HTML/CSS/JS – (files stored in public/)
  - Plain PHP – (resources/views/{view_name}.php)
  - Blade templates – (resources/views/{view_name}.blade.php)
    - A templating engine inspired by .NET
    - Much more concise syntax than plain PHP
    - Simplifies the creation of extensible, unified layouts for your application

- Common ways to return a view:

```
Route::get('/', function() {
    return \File::get(public_path() . '/home.html');
});
```

Return a static HTML file.

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

Laravel will search for home.blade.php **or** home.php in the resources/views folder, execute it and return the result to the client.

```
Route::get('/', function()
{
   return View::make('hello');
});
```

Same as above – Laravel will search for hello.blade.php **or** hello.php in the resources/views folder, execute it, and return the result to the client.

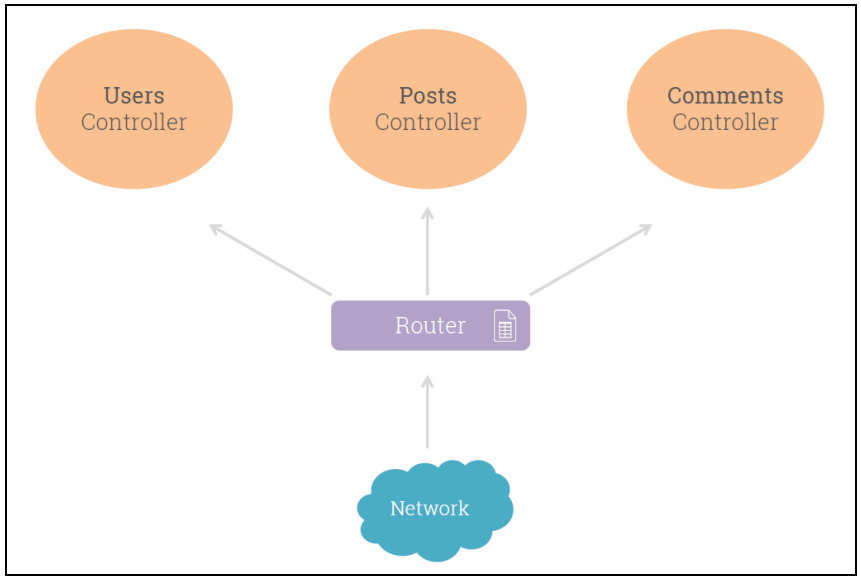
But "View::make(…)" is less commonly used than "return view(…)" in recent versions of Laravel.

- Views can be returned directly from routes (like above), or from a Controller class, within action methods

More on controllers

Defining routes to multiple controllers in routes/web.php:

```
Route::get('/users', 'UsersController@show);
Route::get('/posts', 'PostsController@show);
Route::get('/comments', 'CommentsController@show);
```



Route parameters are automatically passed to controllers, e.g., {id} can be retrieved by using $id within update():

```
Route::get('user/{id}, 'UserController@update');
```

Reading user inputs (e.g., from HTML forms) through the Request object:

```
public function store(Request $request)
{
    $name = $request->input('name');

    //
}
```

The Request object will then be "injected" to the store() method automatically.

If Request is injected, and you have a route parameter at the same time, the route parameter would have to be declared explicitly:

```
public function update(Request $request, $id)
{
    //
}
```

MVC hello world

The model (data) (app/Math.php):

```
<?php
class Math
{
    public function sum($val1,$val2)
    {
            return $val1 + $val2;
    }
}
?>
```

The view (presentation) (resources/views/hello.php):

```
<?php
   echo 'sum is ' . $result;
?>
```

The controller (application logic) (app/Http/Controllers/HelloController.php):

```
<?php
    class HelloController extends Controller
    {
        public function home()
        {
            $math = new Math();
            $sum = $math->sum(5,10);
            return view('hello', ['result' => $sum]);
        }
    }
?>
```

1. Obtain the data

Note: this is just an example - in real-world applications, the model class should pull data from a backend database or from another website through web APIs.

2. Control would then be passed automatically to the “view”

Another way to share data with View:
`return view('hello')->with('result', $sum);`

Resource Controllers

- Usually, a controller groups all actions related to a particular object (i.e., “noun”) together.
- Such an object is normally called a “resource”; most resources (especially real-world objects like a “book”) support typical operations like “show”, “edit”, “create”, “destroy”, etc.
  - The typical CRUD (create, read, update, delete) operations in database
  - E.g., a Book resource (i.e., the “BookController” class) might have actions like “show”, “edit”, “create”, “destroy”, etc.
- Creating these operations and their routes is tedious; Laravel supports Resource Controllers to simplify the development

```
php artisan make:controller TasksController --resource
```

Verb	URL	Controller method	Name	Description
GET	tasks	index()	tasks.index	Show all tasks
GET	tasks/create	create()	tasks.create	Show the create task form
POST	tasks	store()	tasks.store	Accept form submission from the create task form
GET	tasks/{task}	show()	tasks.show	Show one task
GET	tasks/{task}/edit	edit()	tasks.edit	Edit one task
PUT/PATCH	tasks/{task}	update()	tasks.update	Accept form submission from the edit task form
DELETE	tasks/{task}	destroy()	tasks.destroy	Delete one task

The generated skeleton – you will need to implement the logic for each action.

```
// routes/web.php
Route::resource('tasks', 'TasksController');
```

A single route definition for all the supported methods.

Redirections in Laravel

Simple redirection:

```
Route::redirect('/here', '/there');
```

Redirecting a client to his previous page (like the browser's "back" button):

```
return redirect()->back();
```

Redirecting to a specific path (3 possible ways):

```
// Using the global helper to generate a redirect response
Route::get('redirect-with-helper', function () {
    return redirect()->to('login');
});

// Using the global helper shortcut
Route::get('redirect-with-helper-shortcut', function () {
    return redirect('login');
});

// Using the facade to generate a redirect response
Route::get('redirect-with-facade', function () {
    return Redirect::to('login');
});
```

Redirecting to a named route:

```
Route::get('redirect', function () {
    return redirect()->route('conferences.index');
});
```

Passing parameters to the redirected page:

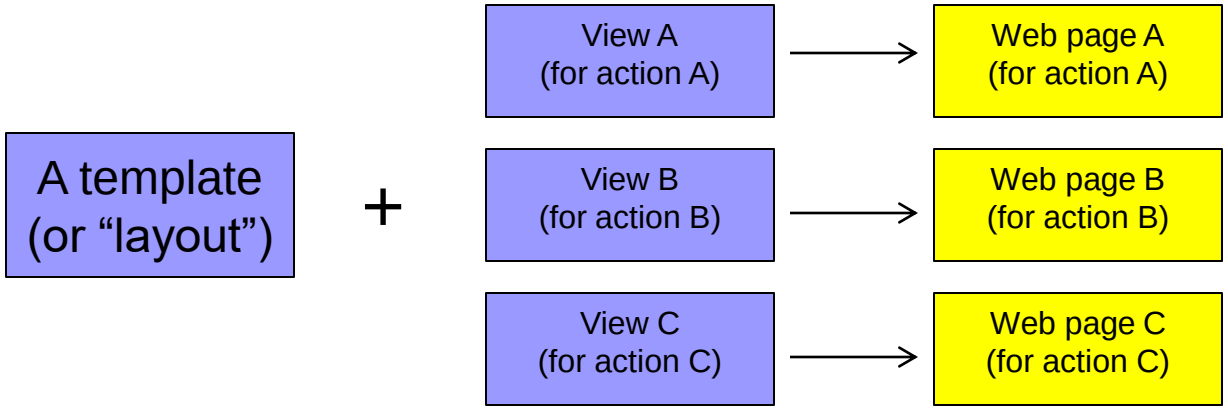
```
Route::get('redirect-with-key-value', function () {
    return redirect('dashboard')
        ->with('error', true);
});
```

The Blade templating engine

The “View”

What is a template?

- A template is a "layout" of a page containing variables and sections to fill in to form complete web pages
  - E.g., the company's logo, navigation section, footer, copyright message, disclaimers, etc., which are displayed in every page in a website, can be put in the template
  - A template "inserts" the information (e.g., page title, main content, etc.) provided by the "View"'s of individual web pages
- Most websites have one or only a few templates that provide a unified "look and feel" of the entire website



Blade

- PHP itself is a powerful templating language, but its syntax is just too ugly, with <?php … ?> all over the place, which are difficult to read/maintain as well.
- Blade is Laravel's templating engine built on PHP, with a clean and concise syntax.

Regular PHP:

```
<?php if ($user->isLogged()) { ?>
    Welcome back, <?= $user->name; ?>
<?php } ?>
```

A Blade view script:

```
@if ($user->isLogged())
    Welcome back, {{ $user->name }}
@endif
```

Blade's special directives

{{ … }} will echo the evaluated expression, with escaping done automatically.

I.e., {{ $variable }} =
<?= htmlentities($variable) ?>

(Note: see Session 3 on Cross-Site Scripting attacks and the purpose of htmlentities())

Template inheritance

Laravel uses '.' to represent folders; layouts.master means layouts/master

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

```
<!-- resources/views/layouts/master.blade.php -->
<html>
    <head>
        <title>My Site | @yield('title', 'Home Page')</title>
    </head>
    <body>
        <div class="container">
            @yield('content')
        </div>
        @section('footerScripts')
            <script src="app.js"></script>
        @show
    </body>
</html>
```

+

```
<!-- resources/views/dashboard.blade.php -->
@extends('layouts.master')

@section('title', 'Dashboard')

@section('content')
    Welcome to your application dashboard!
@endsection

@section('footerScripts')
    @parent

    <script src="dashboard.js"></script>
@endsection
```

Result:

```
<html>
    <head>
        <title>My Site | Dashboard</title>
    </head>
    <body>
        <div class="container">
            Welcome to your application dashboard!
        </div>
            <script src="app.js"></script>
            <script src="dashboard.js"></script>
    </body>
</html>
```

@parent means to also include the "default" content.

Other Blade control structures

if … elseif … endif:

```
@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:

Example 4-4. @for and @endfor

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

Example 4-5. @foreach and @endforeach

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

Example 4-6. @while and @endwhile

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

The special $loop variable:

```
@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
```

PHP code block:

```
@php

    //

@endphp
```

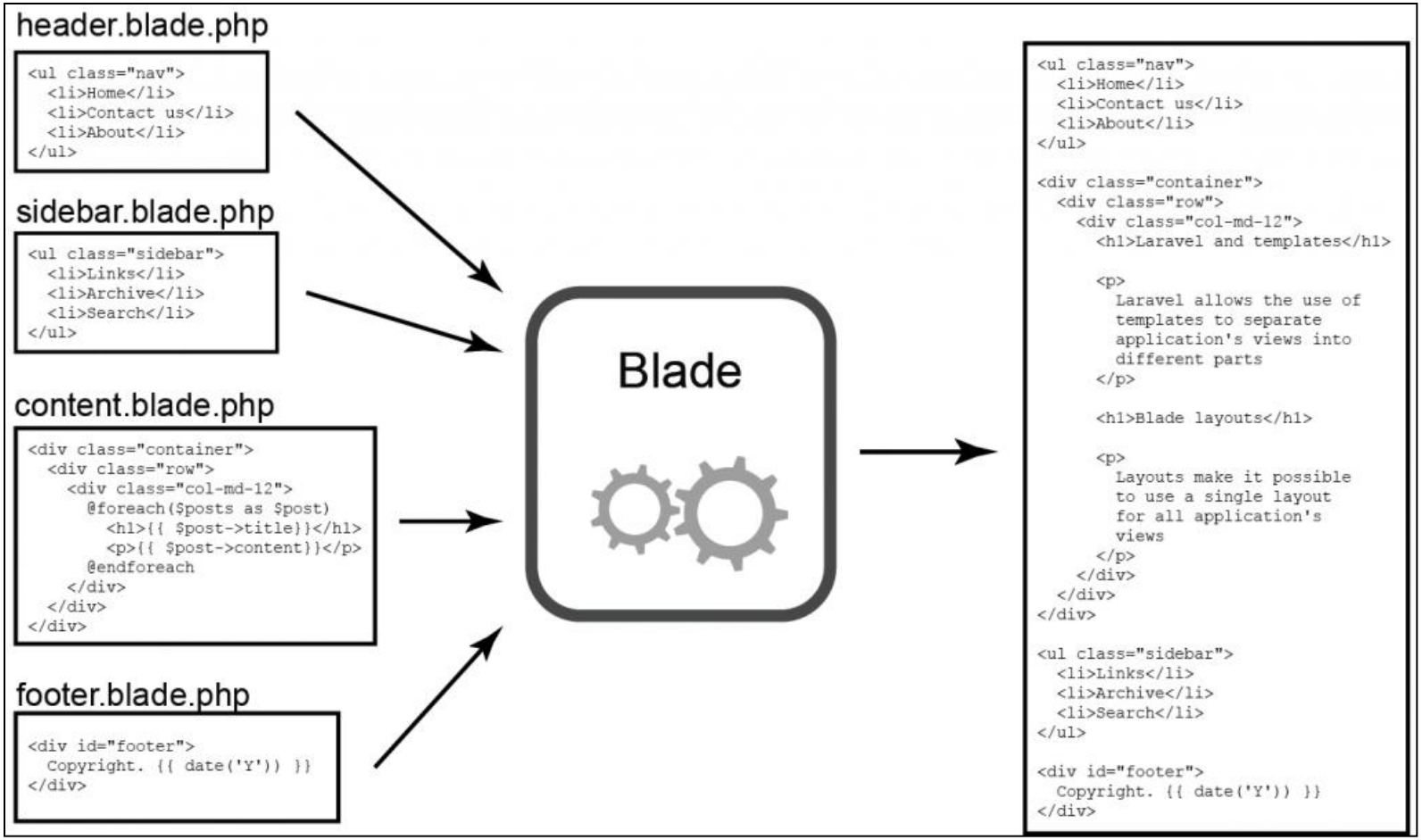
Include another blade script:

```
<div>

    @include('shared.errors')


    <form>

        <!-- Form Contents -->

    </form>

</div>
```

Comments:

```
{{-- This comment will not be present in the rendered HTML --}}
```

Typical usage of @include()



Combining multiple "view partial"'s to form a complete HTML page.

Database access, Eloquent Model and ORM

The “Model”

Database access

- Database access in Laravel is very simple and flexible
- Laravel supports MySQL, SQLite, PostgreSQL, and SQL Server
- The DB connections are defined in *config/database.php*

A SQLite connection:

```
'sqlite' => [
    'driver'   => 'sqlite',
    'database' => database_path('database.sqlite'),
    'prefix'   => '',
],
```

To specify the default DB connection in *config/database.php*:

```
'default' => env('DB_CONNECTION', 'mysql'),
```

A MySQL/MariaDB connection:

```
'mysql' => [
    'driver'    => 'mysql',
    'host'      => env('DB_HOST', 'localhost'),
    'database'  => env('DB_DATABASE', 'forge'),
    'username'  => env('DB_USERNAME', 'forge'),
    'password'  => env('DB_PASSWORD', ''),
    'charset'   => 'utf8',
    'collation' => 'utf8_unicode_ci',
    'prefix'    => '',
    'strict'    => false,
    'engine'    => null,
],
```

- There are three ways to interact with database tables in Laravel: raw SQL, query builder, and Eloquent ORM

Raw SQL and query builder

- You may access (select, insert, update, delete) the DB anywhere in your application without creating a Model, by using raw SQL or the query builder
- Query builder allows for creating queries through function chaining

Raw SQL:

Parameter binding to prevent SQL injection attacks (see Session 3)

```
$usersOfType = DB::select(
    'select * from users where type = ?',
    [$type]
);
```

```
DB::insert(
    'insert into contacts (name, email) values (?, ?)',
    ['sally', 'sally@me.com']
);
```

```
$countUpdated = DB::update(
    'update contacts set status = ? where id = ?',
    ['donor', $id]
);
```

```
$countDeleted = DB::delete(
    'delete from contacts where archived = ?',
    [true]
);
```

Query builder:

Query builder performs parameter binding automatically

```
$usersOfType = DB::table('users')
    ->where('type', $type)
    ->get();
```

```
$emails = DB::table('contacts')
    ->select('email', 'email2 as second_email')
    ->get();
```

```
$id = DB::table('contacts')->insertGetId([
    'name' => 'Abe Thomas',
    'email' => 'athomas1987@gmail.com',
]);
```

Or using insert() if $id is not needed

```
DB::table('contacts')
    ->where('points', '>', 100)
    ->update(['status' => 'vip']);
```

Carbon is a PHP extension for handling DateTime

```
DB::table('users')
    ->where('last_login', '<', Carbon::now()->subYear())
    ->delete();
```

Eloquent Model and ORM

- In Laravel, each DB table can be “mapped” into an in-memory Model class that “represents” the table in your code => “object-relational mapping” (ORM)

- Eloquent is Laravel’s implementation of ORM for object-oriented database access

- Why ORM and Eloquent (instead of using raw SQL or query builder)?
  - Full benefits of using object orientation => better maintainability & extensibility
  - The code is far more readable (object-oriented) than raw SQL / query builder
  - A table can be accessed as a whole, by using the Model class
    - e.g., `Users::all()` can return all users from the database table
  - Each object instance represents a table row (e.g., `$fred = new User`)
  - An object can manage its own persistence, e.g., `$fred->save()`, `$fred->delete()`, etc.

Eloquent Model

To create an Eloquent model class:

```
php artisan make:model Contact
```

app/Contact.php:

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Contact extends Model
{
    //
}
```

Laravel assumes:

- table name = plural form, and small letters, of the class name, i.e., table name = "contacts" in this example
- there is an id column (the primary key) in the table
- name of object attributes = name of the table columns
- there are created_at & updated_at timestamp columns

No manual definition of the above items is required!

That's it! This model class would be mapped to the "contacts" table in the DB automatically.

You do not need to define the object attributes or table columns manually.

Once the model class is created, you can do something like:

```php
public function save(Request $request)
{
    // Create and save a new contact from user input
    $contact = new Contact();
    $contact->first_name = $request->input('first_name');
    $contact->last_name = $request->input('last_name');
    $contact->email = $request->input('email');
    $contact->save();

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

Eloquent DB queries

A typical way of passing Model data to View in a Controller

```
// ContactController
public function show($contactId)
{
    return view('contacts.show')
        ->with('contact', Contact::findOrFail($contactId));
}
```

SELECT

```
$allContacts = Contact::all();
```

```
$vipContacts = Contact::where('vip', true)->get();
```

INSERT

```
$contact = new Contact;
$contact->name = 'Ken Hirata';
$contact->email = 'ken@hirata.com';
$contact->save();
```

```
$contact = Contact::create([
    'name' => 'Keahi Hale',
    'email' => 'halek481@yahoo.com'
]);
```

create() ~= new() + save()

UPDATE

```
$contact = Contact::find(1);
$contact->email = 'natalie@parkfamily.com';
$contact->save();
```

```
Contact::where('created_at', '<', Carbon::now()->subYear())
    ->update(['longevity' => 'ancient']);
```

DELETE

```
$contact = Contact::find(5);
$contact->delete();
```

```
Contact::destroy([1, 5, 7]);
```

```
Contact::where('updated_at', '<', Carbon::now()->subYear())->delete();
```

Eloquent is powerful; read the documentation for more query options and functions. E.g., for security, you may use the fillable / guarded attributes to specify which table columns can be updated.

Migration support

- Modern frameworks (e.g., Laravel, Rails, etc.) support database migrations for defining and managing database structures in a code-driven way.
- With migrations, tables can be created and database schemas can be updated in code.
- => Any new database (e.g., after your website is migrated to another ISP, or a backup site upon network/hardware failures) can be brought to your application's schema in seconds.

```php
class CreateUsersTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('users', function (Blueprint $table) {
            $table->increments('id');
            $table->string('name');
            $table->string('email')->unique();
            $table->string('password', 60);
            $table->rememberToken();
            $table->timestamps();
        });
    }

    public function down()
    {
        Schema::drop('users');
    }
}
```

Every migration file has a up() and a down() method.

up() – to apply the migration.
down() – to un-do it.

Definition of table columns.

To create a migration file skeleton:

php artisan make:migrate {file_name}

To run a migration:

php artisan migrate

Form validations

Form validation in Controller

- Form validations can be the most tedious part of server-side programming; Laravel provides a validate() method that simplifies the task.

```php
// routes/web.php
Route::get('recipes/create', 'RecipesController@create');
Route::post('recipes', 'RecipesController@store');

// app/Http/Controllers/RecipesController.php
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class RecipesController extends Controller
{
    public function create()
    {
        return view('recipes.create');
    }

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

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

Provide the initial form to the client

Accept form submission

This View contains the initial form

'title' is required, unique in the recipes table and shorter than or equal to 125 characters

'body' is required

If validation succeeds, execution would continue

If validation fails, the user would be redirected to the previous page (i.e., the initial form).

User inputs and $errors would be provided to the redirected page, so developers can print the submitted values in the form, and the corresponding errors.

Displaying errors and previous user inputs

- If validation fails, in the redirected page (i.e., the form), you may want to display the validation errors and the previous user inputs - so that the user does not need to input everything again.
- These information can be retrieved by using $errors and the old() method

Whoops! There were some problems with your input.

- The first name must be at least 5 characters.
- The last name field is required.
- The email must be a valid email address.
- The mobileno must be a number.
- The password field is required.
- The confirm password field is required.
- The details field is required.

```
@if(count($errors))
        <div class="alert alert-danger">
                <strong>Whoops!</strong> There were some problems with your input.
                <br/>
                <ul>
                        @foreach($errors->all() as $error)
                        <li>{{ $error }}</li>
                        @endforeach
                </ul>
        </div>
@endif
```

```
<input  type="text"
        name="firstname"
        placeholder="Enter First Name"
        value="{{ old('firstname') }}">
```

"Enter First Name" is displayed on the initial form.

If there are validation errors and the form is displayed again, the previously-submitted firstname would be displayed.

Form requests

- Doing all validations in the controller can “clutter” the controller, which should be about application logic instead of validations (which are more for data integrity)
- Laravel’s form requests come to rescue, which separates validation logic from the controllers

To create a form request class:

```
php artisan make:request CreateCommentRequest
```

app/Http/Requests/CreateCommentRequest.php:

```
class CreateCommentRequest extends Request
{
    public function rules()
    {
        return [
            'body' => 'required|max:1000'
        ];
    }
```

In the controller class, simply **typehint** the request object with the form request class, Laravel will perform the validation using the rules specified.

The result is the same as doing validation in the controller.

In addition to form requests, Laravel also supports custom validation for defining application-specific validation logic. Read the documentation for details.

Using form request for validation:

```
Route::post('comments', function (App\Http\Requests\CreateCommentRequest $request) {
    // Store comment
});
```

User authentication

Authentication

- Authentication means verifying who someone is, and allowing them to act as that person in your application, e.g., the login process.

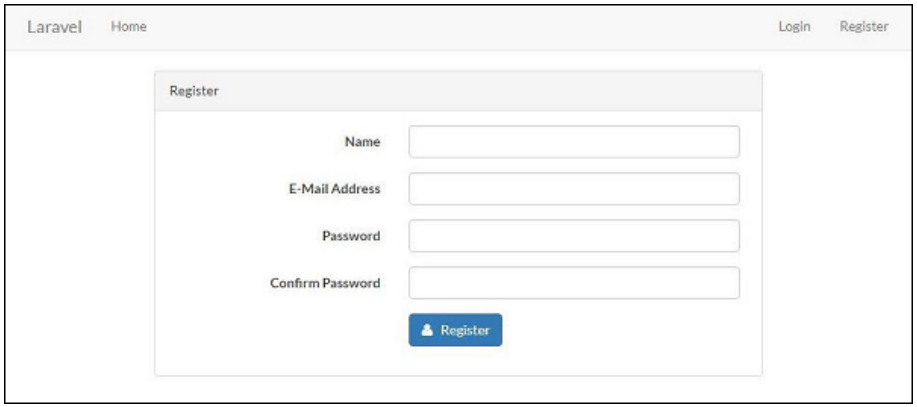
- A simple authentication system should support:
  - User registration - using the client's name, email address and password
  - User login and logout
  - Password reset using a valid email address
  - Controlling access to routes
  - A 'users' database table

- Laravel provides these out of the box, so that developers can focus on the application logic. For a new application, simply do:

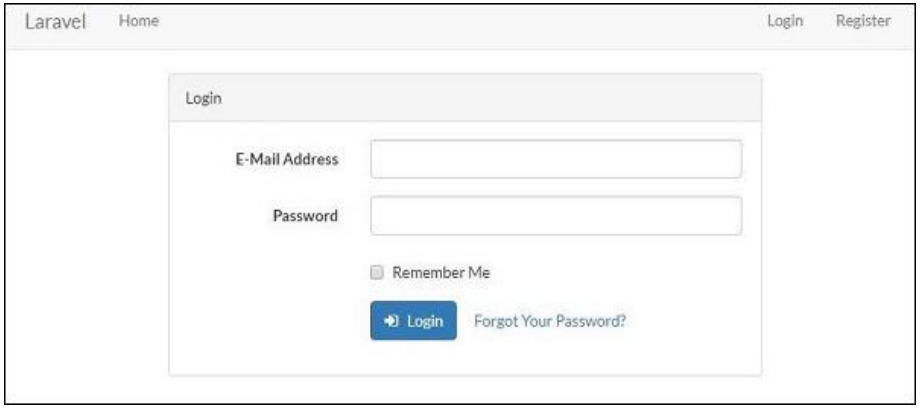
```
php artisan ui vue --auth
php artisan migrate
```

- That's it!

User registration form:



User login form:



Forgot password form:

Authentication

The generated routes – you don't need to add these yourself:

```
// Authentication Routes
$this->get('login', 'Auth\LoginController@showLoginForm');
$this->post('login', 'Auth\LoginController@login');
$this->post('logout', 'Auth\LoginController@logout');

// Registration Routes
$this->get('register', 'Auth\RegisterController@showRegistrationForm');
$this->post('register', 'Auth\RegisterController@register');

// Password Reset Routes
$this->get('password/reset', 'Auth\ForgotPasswordController@showLinkRequestForm');
$this->post('password/email', 'Auth\ForgotPasswordController@sendResetLinkEmail');
$this->get('password/reset/{token}', 'Auth\ResetPasswordController@showResetForm');
$this->post('password/reset', 'Auth\ResetPasswordController@reset');
```

/login – the login form
/register – the user registration form
/password/reset – forgot pw form
/password/reset/{token} – reset pw form

The generated files:

```
app/Http/Controllers/HomeController.php
resources/views/auth/login.blade.php
resources/views/auth/register.blade.php
resources/views/auth/passwords/email.blade.php
resources/views/auth/passwords/reset.blade.php
resources/views/layouts/app.blade.php
resources/views/home.blade.php
```

Using authentication

- Laravel uses middleware for authentication.
- A route (i.e., the corresponding Controller, Model and View) can be protected using the "auth" middleware

```
Route::get('dashboard', 'HomeController@dashboard')->middleware('auth');
```

Only authenticated users can enter this route. Guests will be redirected to the login page. The way how guests are handled can be changed by modifying app/Http/Middleware/Authenticate.php .

- You may also restrict pages to guests only (i.e., not authenticated users) by using the "guest" middleware:

```
Route::get('guestlanding', 'HomeController@guestlanding')->middleware('guest');
```

Only guests can enter this route.

- Middleware provides a powerful mechanism for filtering HTTP requests entering your application. You may create your own middleware (e.g., for checking clients' physical location, whether they are "administrators", etc.) – read the documentation for detail.

Summary

- The MVC pattern enforces separation of concerns of server-side code
  - Facilitates code maintainability
  - Ensures application extensibility
  - Supplements the separations of {document structure, presentation, behavior} at the client side

- MVC server-side frameworks (e.g., Laravel) simplify and better organize implementations of web 2.0 applications. Benefits:
  - The MVC pattern for clean separations
  - Write less code
    - The framework simplifies common (and tedious) tasks such as layout management (Blade), form validation, authentication, database access (Eloquent ORM) and many others
    - The third-party tools developed by the community can help perform almost all kinds of server-side development tasks – similar to how jQuery/Bootstrap simplify client-side development
    - Result: rapid development of maintainable, extensible server-side code

Post-class self-learning resources

- Learning Laravel
  - A lot of information & tutorials available online for Laravel
    - Simply search when you encounter any problems in programming
  - Laravel's official documentation
    - https://laravel.com/docs/6.x
  - Laracasts – tutorials in videos/screencasts ("Netflix for Laravel learners")
    - https://laracasts.com/
  - Laravel.io (the community, forum, etc.)
    - https://laravel.io/forum
- Take a brief look at some other server-side MVC frameworks and try their demos/tutorials (if available):
  - PHP: Symfony, CakePHP, CodeIgniter
  - Java: Java Server Faces, Apache Struts, Spring Framework
  - JavaScript: Express on Node.js
  - Ruby: Ruby on Rails
  - Microsoft: ASP.NET MVC
- See Moodle for the links

ICOM6034: Assignment

- Only one assignment: Assignment 1
- Assignment 1 [30%]:
  - On jQuery and Laravel
  - An extension of Labs 1, 2, 3A (today) and 3B (on Feb 5)
  - (but only part of the assignment is related to Lab 3B, so you may start doing it now)
  - Released today/tomorrow; due on March 17, 2021, 11:59pm
- Completing the labs before doing the assignment will save you quite a bit of time
- Late penalty: 1 point (out of 30) per day.
- Details of the group project will be announced in the next lecture