

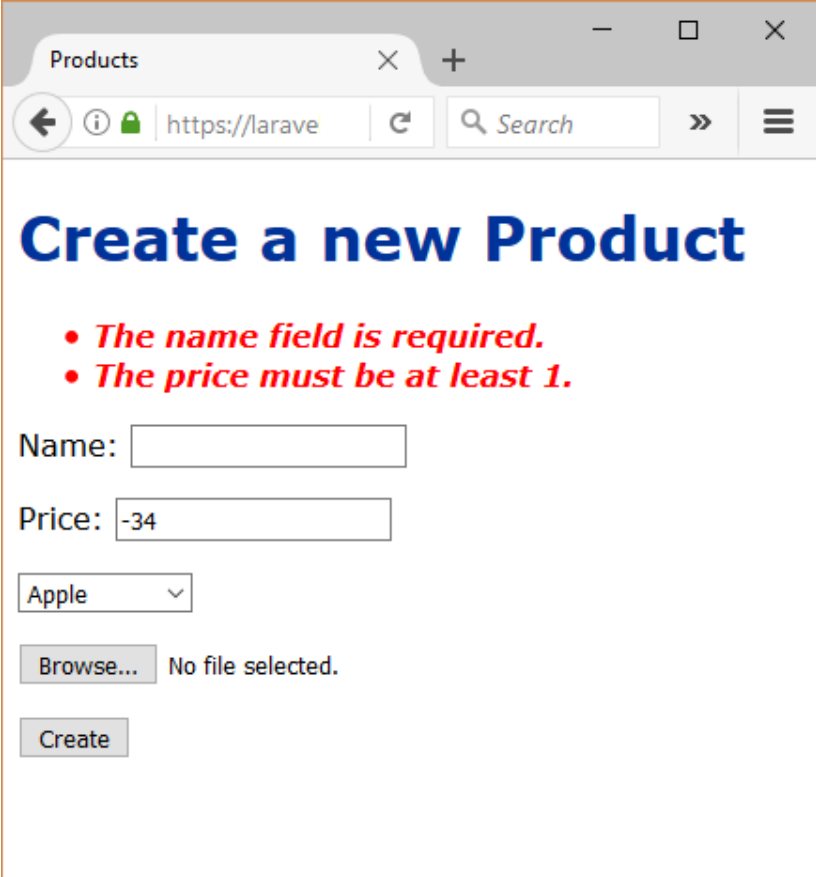
# Validation and User Authentication

- Input validation with rules
- State maintenance
- User authentication
- Model – Many to many relationship



# Input Validation with Rules

- Laravel greatly simplifies the implementation of input validation.
- Instead of writing validation logic, we only need to specify the **validation rules**, and Laravel will perform the validation for us.
- The validation functionality is provided by the **validation()** function implemented by the base controller.
- Since all controllers should extend base controllers, so we can use validation in our controller class by **`$this->validate()`**.



The screenshot shows a web browser window with a single tab titled 'Products'. The address bar displays 'https://larave' with a search icon to the right. The main content area has a heading 'Create a new Product' in blue. Below the heading, there are two red error messages: '• The name field is required.' and '• The price must be at least 1.'. The form contains the following fields: a text input for 'Name:' which is empty, a text input for 'Price:' containing '-34', a dropdown menu currently showing 'Apple', a file upload section with a 'Browse...' button and the text 'No file selected.', and a 'Create' button at the bottom.

E.g.: **store()** in **ProductController**:

```
public function store(Request $request){  
    $this->validate($request, [  
        'name' => 'required|max:255',  
        'price' => 'required|numeric',  
        'manufacturer' => 'exists:manufacturers,id'  
    ]);  
    $product = new Product();  
    $product->name = $request->name;  
    $product->price = $request->price;  
    $product->manufacturer_id = $request->manufacturer;  
    $product->save();  
    return redirect("product/$product->id");  
}
```

- The call to **validate()** takes two parameters:
  1. The request object.
  2. The validation rule.
- The validation rule is an associate array, where the name of the input is the key and the rule is the value. In the above example:

- **Name** is required and can be at most 255 characters.
- **Price** is required and must be numeric.
- **Manufacturer** with that id must exist in the manufacturers table.
- Multiple rules can be specified for an input. | separates the rules.
- Some rules have parameters, e.g.:
  - max:value
  - exists:table,column
- Other rules include:
  - Date
  - Email
  - Image (filename must have an image extension)
  - Integer
  - Etc.
- A complete list of rules is provided in the Laravel documentation:
  - <http://laravel.com/docs/validation>
- If the validation fails, the rest of the store() will not be execution, the execution will be redirect back to the **form page**.
- If the validation passes, the rest of the store() will be executed.

## Displaying Error Message and Old Values

- When a validation error occurs, the form page needs to display the error and the old values (the ones submitted for validation), to allow user to correct the error.
- Laravel will generate the error messages and store them in **\$errors**.
- **\$errors** is available to all views in the **web** middleware group.
- **\$errors->all()** returns an array of strings containing all of the error messages.
- To get all of the error messages for a particular field use (note that a field can produce multiple errors):

```
$errors->get('name')
```

- To get the first message for a field use:

```
$errors->first('name')
```

- The value entered by the user is stored in the session variable and can be retrieved by using the helper function **old()**. E.g. `old('name')` to retrieve the previously entered name.
- The old value can take a 2<sup>nd</sup> parameter. If there is no old value, it will return value from the 2<sup>nd</sup> parameter, e.g. `old('name', $product->name)`, where it returns the old name, if there is one, otherwise, it returns the value in `$product->name`.

- An extract from **views/products/create\_form.blade.php**:

```
@section('content')
<h1>Create a new Product</h1>
@if (count($errors) > 0)
  <div class="alert">
    <ul>
      @foreach ($errors->all() as $error)
        <li>{{ $error }}</li>
      @endforeach
    </ul>
  </div>
@endif
<form method="POST" action="product">
  {{csrf_field()}}
  <p><label>Name: </label><input type="text" name="name" value="{{
old('name') }}"></p>
  <p><label>Price: </label><input type="text" name="price" value="{{ old('price')
}}"></p>
```

```
<p><select name="manufacturer">
@foreach ($manufacturers as $manufacturer)
  @if($manufacturer->id == old('manufacturer'))
    <option value="{{ $manufacturer->id }}" selected="selected">{{ $manufacturer-
>name}}</option>
  @else
    <option value="{{ $manufacturer->id }}">{{ $manufacturer->name}}</option>
  @endif
@endforeach
</select></p>
<input type="submit" value="Create">
</form>
@endsection
```

# State Maintenance

- HTTP is a stateless protocol.
- Each request/response is independent from other request/responses.
- However, applications often need to maintain state between page requests.
- For example, if a user adds an item to a shopping cart, the application must remember the cart's state even while the user browses for other items.
- There are four mechanisms which can be used to maintain state between pages:

- URL parameters
- Hidden components
- Cookies
- Session variables

## URL parameters

- State can be maintained using URL parameters by ensuring that all state is passed to every subsequent page request.
- This means that all links on a page must be dynamically generated to include state information.
- The guest book application with pagination uses URL parameters to maintain state of the current page being viewed.
- The problem with using URL parameters is that the user may access the site through a URL which



doesn't have the state parameters, e.g. by visiting a bookmark, or going back in the browser's history.

## Hidden components

- State can be maintained in hidden form fields, such as the id.
- The data in hidden form fields will be sent to the server with the other form field data.
- Hidden form fields are useful for updating item data.
- However, they can not be used to maintain state when links are clicked.

## Cookies

- Cookies are a way for a web application to store data in the browser.

- Cookie data is associated with a URL and will be sent back to the server when the browser accesses the URL again.
- When a browser accesses a web application a request is sent, e.g:

```
GET /index.html HTTP/1.1  
Host: www.example.org
```

browser -> server

- The server sends back a Set-Cookie header with a name-value pair which will be stored in the browser, e.g:

```
HTTP/1.1 200 OK  
Content-type: text/html  
Set-Cookie: name=value
```

server -> browser

- The browser will store the string "value" against the name "name" for the URL: ww.example.org/index.html

- When the browser accesses the URL again it will send back the cookies that the server set:

```
GET /index.html HTTP/1.1  
Host: www.example.org  
Cookie: name=value
```

browser -> server

- The server now has access to the data that it stored in the browser.

## Cookie attributes

- The server may specify additional attributes in the Set-Cookie header:
- Domain and Path
  - By default the domain and path of the cookie is the same as what was in the request.

- This can be overridden by specifying domain and path parameters.
- For example, all pages on the website may track the user's name, in this case we want the website to be the domain and the path to be "/", e.g:

```
Set-Cookie: name=value;  
Domain=www.example.org; Path=
```

- All pages accessed below `www.example.org/` will be sent the cookie.
- Expires and Max-Age
  - An expiry date can be set on a cookie.
  - The browser must delete the cookie after the expiry date.
  - The expiry form is: "Wdy, DD Mon YYYY HH:MM:SS GMT" e.g:

```
Set-Cookie: name=value; Expires=Tue, 15 Jan 2013 21:47:38 GMT; Path=/
```

- Alternatively a Max-Age can be set as an interval of seconds in the future from when the browser received the cookie.
- If no expiry date is set, the cookie is only valid for a session.
- A session ends when the browser quits.
- Secure and HTTPOnly
  - These options do not have values.
  - Secure indicates that the cookie will only be sent on secure connections, e.g. HTTPS.
  - HTTPOnly indicates that the cookie will only be sent on HTTP connections.
  - Cookies can be accessed through JavaScript (document.cookie), HTTPOnly prevents cookie access from JavaScript.

## Working with cookies in PHP

- Cookies are set using the **setcookie()** function.
- The setcookie() function sets up the correct header for the cookie.
- The setcookie() parameters are:

**setcookie**(name, value, expire, path, domain, secure)

- All the arguments except name are optional.
- Cookie example:

```
<?php  
$strvalue = "Hello, this is a cookie";  
Setcookie("TestCookie", $strValue);  
Echo "<p>Cookie set</p>"  
?>
```

- When this PHP script is accessed from a browser it will return a response with a Set-Cookie header, e.g:

```
HTTP/1.1 200 OK  
Content-type: text/html  
Set-Cookie: TestCookie=Hello, this is a cookie Cookie set  
<p>Cookie set</p>
```

- Upon receiving this response the browser will store the TestCookie cookie and its value against the URL of the PHP page.
- When the page is accessed again the browser will send the cookie as follows:

```
GET /spec.html HTTP/1.1
Host: www.example.org
Cookie: TestCookie=Hello, this is a cookie
```

- The cookies sent to a PHP page can be accessed through the `$_COOKIE` array using the name of the cookie:

```
<?php
$strCookieData = $_COOKIE["TestCookie"];
echo "$strCookieData";
?>
```

- Another script that creates and uses cookies is in WebDev-Examples:
  - WebDev-Examples/week9/cookies.php

## Session variables

- Cookies store all of the data in the browser and send it back to the server.
- Another approach is to store a unique session identifier in a cookie but store all of the data related to that session on the server, this is the concept of *session variables*.
- Working with session variables requires three steps:
  - A. Setup the session using `session_start()`.

- B. Check if the required session variable has been set, if not give it an initial value.
- C. Use or modify the session variable.
- **session\_start()** takes no parameters and takes care of extracting the session identifier from the **\$\_COOKIE** array and also setting the session identifier using **setcookie()**.
- Therefore no direct access of cookies is required.
- Accessing and modifying session variables happens through the **\$\_SESSION** array.
- The following example increments a counter every time the page is accessed:

```
<?php
session_start();
if (!$_SESSION['intCount'])
    $_SESSION['intCount'] = 1;
else
    $_SESSION['intCount']++;

echo "<p>You have accessed this page $_SESSION['intCount'] times</p>";
?>
```

- The contents of the **\$\_SESSION** array are stored on the server and not sent to the browser.
- **session\_start()** will initialise the contents of the **\$\_SESSION** array based on the session id that is received from the browser in a cookie.
- On the server a session file is created for each session using the session id as the file name.
- The contents of the **\$\_SESSION** array is stored in the session file.
- A version of the cookie example using session variables is available in:
  - WebDev-Examples/week9/sessions.php

### What happens when cookies are disabled?

- If cookies are disabled the PHP session infrastructure can be used but the session id must be passed as a URL parameter, e.g:

```
<?php
session_start();
// Generate a session-specific URL
$orderURL = "order.php?PHPSESSID=" . session_id();
?>
<a href="<?= $orderURL ?>">Create order</a>
```

- To simplify the process PHP sets the contents of the global **SID** to be a string of the form PHPSESSID= be1231231acaeda123aada12309809a7.
- The above fragment can then be written more simply as:

```
<?php
session_start();
?>
<a href="order.php?<?= SID ?>">Create Order</a>
```



# Laravel Sessions

- Laravel includes a lot of session infrastructure which is already used for existing elements of the web framework.
- For example, a random string is generated and stored in the session for each form to prevent Cross Site Request Forgeries.
- As a result, there is no need to call `session_start()` as that will already be done for you. Instead of using session functionalities provided by Laravel.
- There are two primary ways to work with session in Laravel: Request and global session helper.

## Session via Request

- A request object has access to session object. E.g.: in a controller

```
namespace App\Http\Controllers;
use App\Http\Controllers\Controller;
use Illuminate\Http\Request;

class UserController extends Controller{
    // show user profile
    public function show(Request $request, $id){
        $value = $request->session()->get('key');
    }
}
```

- **get()** of session will return the value in the session associated with that 'key' value.
- **all()** of session will return all session data, it can be used instead of `get()`.

- **put()** of session stores data into session. It takes two arguments, the key and the value.

```
$request->session()->put('my_name', 'John Smith');
```

- **forget()** of session deletes data from a session. E.g.:

```
$request->session()->forget('my_name');
```

## Global session helper

- The global **session()** can be used to retrieve or store value to/from session.
- Call session() with one argument to retrieve the value associated with the key:

```
$value = session('key');
```

- Call it with key and value or an array to store into session:

```
$value = session('key', 'default');
```

```
session(['key' => 'value']);
```

# User Authentication

- Websites allow user's to login for various reasons:
  - To view confidential information
  - To perform transactions
  - To post information

## User Authentication

- User authentication consists of a number of states depending on whether the user is logged in and whether they have access to a particular page:

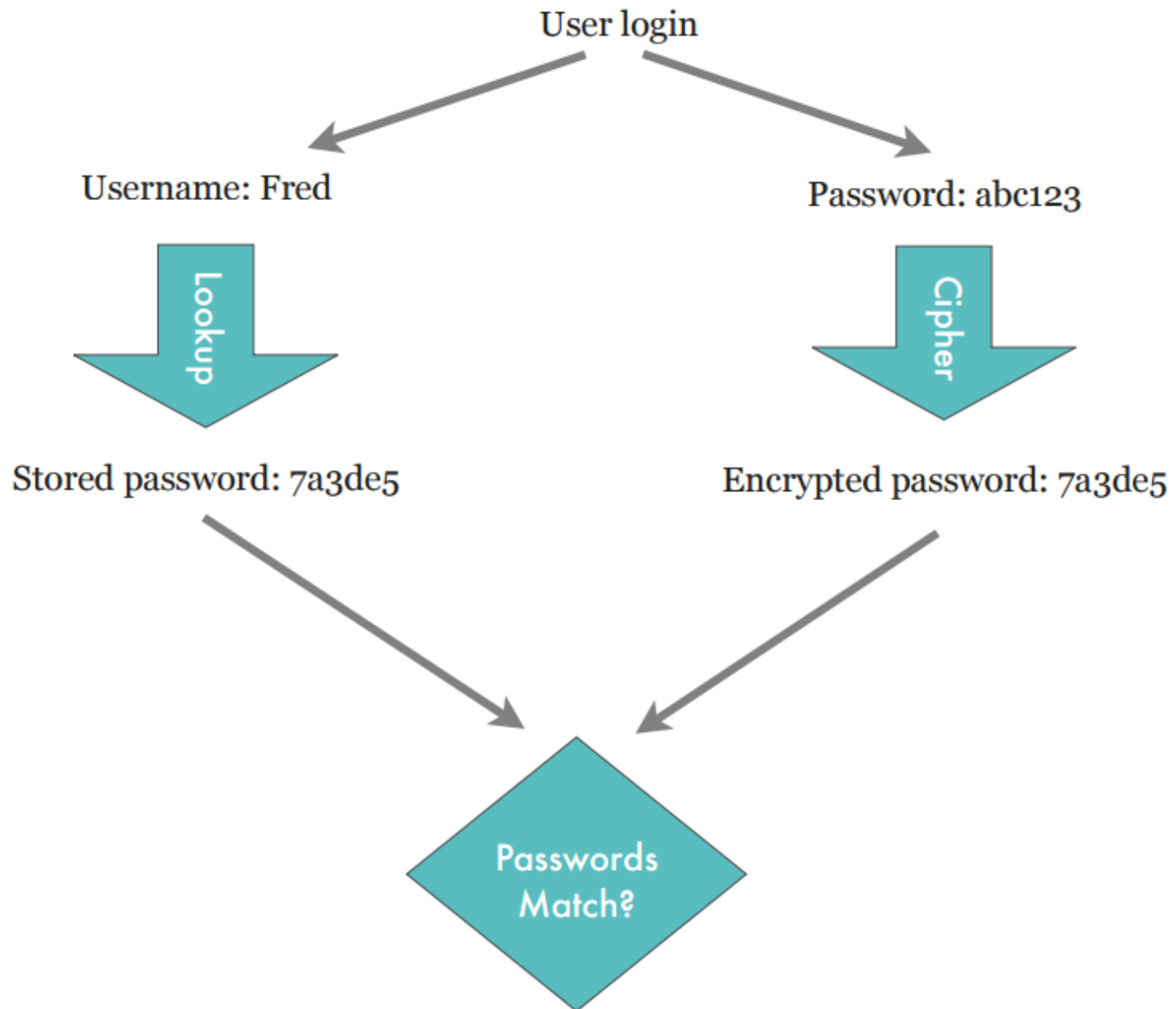
- **If the user is logged out:**
  - Some pages will not be accessible and may instead display an error or redirect to another page.
  - Some pages will be accessible but not show the same information that is visible if the user is logged in.
  - Most pages will either provide a link to allow the user to log in or form fields on each page to allow the user to directly log in.
- **When the user attempts to log in:**
  - If they entered the wrong username or password, must indicate an error and allow user to attempt to log in again.
  - If they logged in successfully should return them to the page where they initiated the log in.

- **If the user is logged in:**
  - No sign in button or log in form is shown.
  - Username should be displayed on each page.
  - Log out button should be displayed on each page.

## **Password Authentication**

- Users are usually authenticated using their username (or email address) and password.
- All these information need to be stored in the database.
- Password is a secret that only the user knows.
- For security purpose, passwords **must be encrypted** before stored into the database. Only encrypted password should be stored in database.
- Even if the database is breached, the passwords are not leaked.
- The safest form of password encryption is a one-way cipher.
- A one-way cipher can never be decrypted.
- How do we know if the user has entered the correct password if the stored password can't be decrypted?

## User Authentication Process



## Laravel's Authentication Scaffold Installation

- As authentication is a common feature in web applications, Laravel provides authentication scaffolds that we could use.
- We will use the simpler authentication scaffold called Laravel Breeze.
- The scaffold provides code for creating the users table (database), functionalities including UI for user registration, login, reset password etc.
- To install Breeze:
  1. Install Breeze using composer:

```
composer require laravel/breeze --dev
```
  2. Run Breeze install:

```
php artisan breeze:install blade
```
  3. Update database:

```
php artisan migrate
```
  4. Install the front-end code:

```
npm install
```
- We will now look at the details of the authentication scaffold from migration/seeders, models, controllers, views, to routes.

## Authentication Scaffold details

### Migration – Users table

- Laravel provides a migration file for creating users table, an extract from this

```
Schema::create('users', function (Blueprint $table) {  
    $table->id();  
    $table->string('name');  
    $table->string('email')->unique();  
    $table->timestamp('email_verified_at')->nullable();  
    $table->string('password');  
    $table->rememberToken();  
    $table->timestamps();  
});
```

- In Laravel's implementation, email is used instead of username. The **unique()** function forces email to be unique.
- Passwords are stored as strings.
- The **remember\_token** is used for functionality that Laravel provides to allow the logged in user to be remembered. You will need this column even if you don't use the remember user feature otherwise it will crash when logging out.

## Seeding Users

- Laravel does not provide a default seeder for users. If required, a seeder file can be created using the artisan command.
- Below is an extract from **UsersTableSeeder.php**:

```
public function run() {  
    DB::table('users')->insert([  
        'name' => "Bob",  
        'email' => 'Bob@gmail.com',  
        'password' => bcrypt('123456'),  
    ]);  
    DB::table('users')->insert([  
        'name' => "Fred",  
        'email' => 'Fred@gmail.com',  
        'password' => bcrypt('123456'),  
    ]);  
}
```

- Note the password is encrypted before storing into the database.
- Since this is for testing purpose, keep the password simple, and easy to remember.



## Automatic Seeding Users via Factory

- Sometimes a lot of test data is required. Laravel's **Factory** class can be used to generate test data.
- With Factory, we can define a set of default attributes for each of our Eloquent models using model factories.
- Laravel comes with UserFactory. The following line in DatabaseSeeder will generate 10 test users:

```
\App\Models\User::factory(10)->create();
```

- We can also look at the code to see how we can define Factory.
  - The class **UserFactory** contains the code on how to generate fake users. It is located in: *database/factories/UserFactory.php*
  - In the model class *app/Models/user.php*, the HasFactory module is including so the User class has access to factory().
- For more information regarding Factory, see: <https://laravel.com/docs/database-testing>

## Routes

- The scaffold inserted the following line to routes/web.php:  

```
require __DIR__.'./auth.php';
```
- Auth.php contains may predefined routes for link to authentication controllers.

## Auth Controllers

- The scaffold adds controllers for registering new user, login, and password reset. These controllers are located in **app/Http/Controllers/Auth/**.
- The one of the main Auth Controller we should pay attention to is the **RegisterUserController**. This controller handles the registration of new users. An extract from **RegisterUserController.php**

```
$request->validate([
    'name' => 'required|string|max:255',
    'email' => 'required|string|email|max:255|unique:users',
    'password' => ['required', 'confirmed', Rules\Password::defaults()],
]);
$user = User::create([
    'name' => $request->name,
    'email' => $request->email,
    'password' => Hash::make($request->password),
]);
```

- User input in `$request` is validated with the supplied rules.
- **User::create()** creates a new user. Note that **Hash::make()** is used to encrypt the password before storing into the database.
- If we need to customise User, e.g. store extra information about user, then in addition to changing the migration file, we will also need to change RegisterUserController.
- The controller for handling user login and logout is the **AuthenticatedSessionController**.
  - It generates the login form in **create()**.
  - Once the login form is submitted, **store()** is called.
    - It use **authenticate()** defined in **App\Http\Requests\Auth\LoginRequest**;
    - The actual authentication is done with **Auth::attempt()**
    - Session is regenerated to prevent *session fixation attack*.
- After registration or authentication, the following redirect is executed:

```
return redirect()->intended(RouteServiceProvider::HOME);
```

- RouteServiceProvider lives in **App\Providers**. The **HOME** constant can be changed to redirect the user to a different location after login, e.g. redirect to the root of the application:

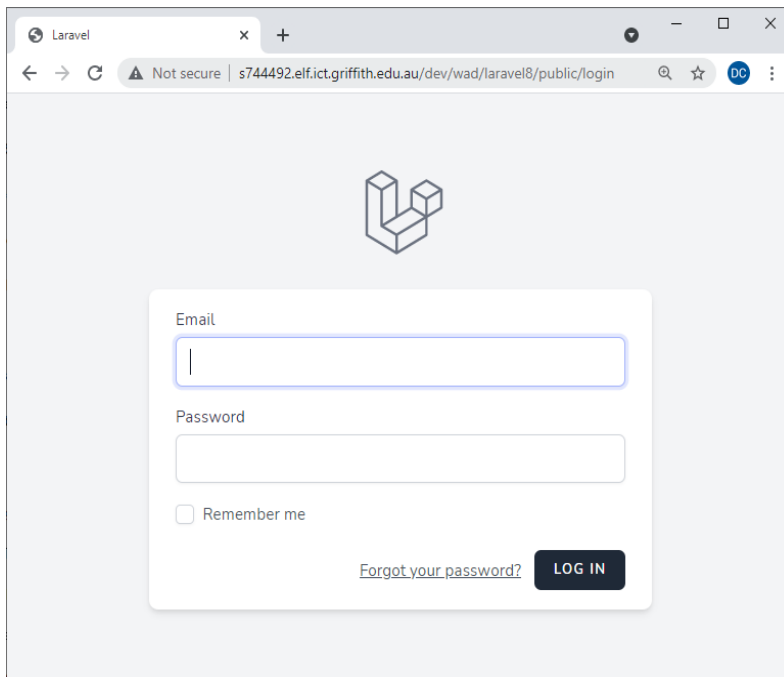
```
public const HOME = '/';
```

## User Model

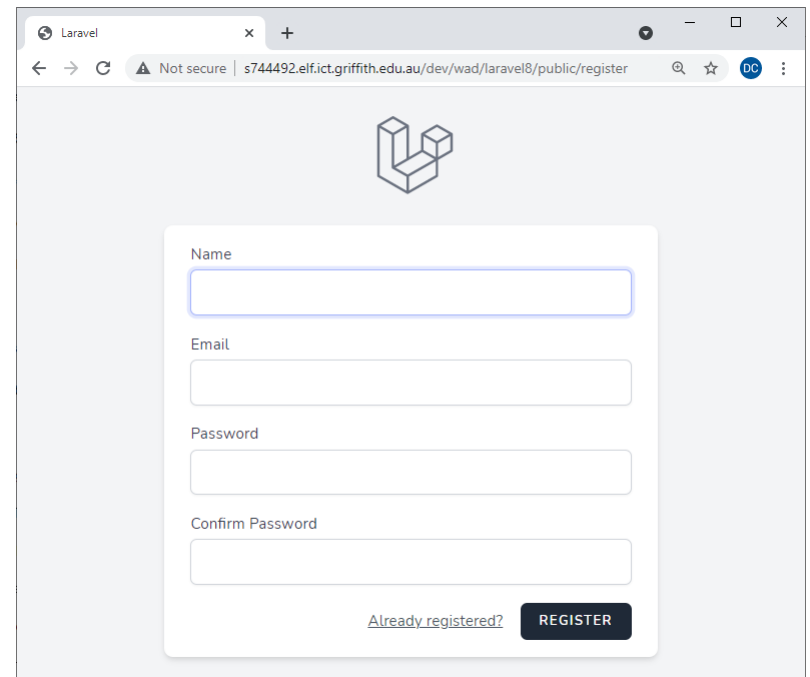
- The scaffold makes use of the model **User**, which is located in **app/Models/User.php**.
- We are free to use this class as a model to implement relationships between the users table and other tables.

## View

- The scaffold creates views in **resources/views/auth** for user registration, login, email password and password reset.
- The scaffold provide ready to use functionality. All we need to do is provide links to routes.



A screenshot of a web browser displaying the Laravel login page. The browser's address bar shows the URL `s744492.elf.ict.griffith.edu.au/dev/wad/laravel8/public/login`. The page features a light gray background with a white login form in the center. At the top of the form is the Laravel logo. Below it, there are input fields for 'Email' and 'Password'. A 'Remember me' checkbox is located below the password field. At the bottom of the form, there is a link for 'Forgot your password?' and a dark blue 'LOG IN' button.



A screenshot of a web browser displaying the Laravel registration page. The browser's address bar shows the URL `s744492.elf.ict.griffith.edu.au/dev/wad/laravel8/public/register`. The page features a light gray background with a white registration form in the center. At the top of the form is the Laravel logo. Below it, there are input fields for 'Name', 'Email', 'Password', and 'Confirm Password'. At the bottom of the form, there is a link for 'Already registered?' and a dark blue 'REGISTER' button.

## Scaffold Styling - Tailwind

- Laravel's auth scaffold use Tailwind CSS for styling: <https://tailwindcss.com/>
- Tailwind requires more hands-on work compared to Bootstrap.
- As the focus of this course is backend, we don't intend to cover Tailwind in this course.
- If you are interested in font-end development, Tailwind is a worth learning (if you agree with the Tailwind approach).
- If you are not already familiar with Tailwind, we suggest you use your own styling or Bootstrap for this course's exercise/assignments.

## Authentication Functions

- Laravel provide several authentication functions and directive.
- To check if the user is logged in, the **Auth::check()** function can be used. E.g.:

```
use Illuminate\Support\Facades\Auth;
```

```
if (Auth::check()) {  
    // The user is logged in...  
}
```

- To retrieve the currently authenticated user:

```
$user = Auth::user();
```

from the user object you can retrieve their details.

- To retrieve the currently authenticated user's ID:

```
$id = Auth::id();
```

- **Auth::guest()** is the opposite of **check()**. **guest()** returns true if the user is not logged in.

- Authentication functions can also be used in view. E.g.

```
@if (Auth::guest())
    <li><a href="{{ route('login') }}">Login</a></li>
    <li><a href="{{ route('register') }}">Register</a></li>
@else
    <!-- user is logged in --->
    ....
```

- Laravel also provide **directives** for use in views: @auth @endauth, @guest @endguest.

```
@auth <!-- user is logged in --->
    {{Auth::user()->name}}
    <form method="POST" action= "{{url('/logout')}}">
        {{csrf_field()}}
        <input type="submit" value="Logout">
    </form>
@else <!-- user is not logged in --->
    ...
@endauth
```

- In the above example, if user is logged in, then display username, and a button for logout.

## Authentication Middleware

- Middleware filters HTTP request entering our application.
- The authentication middleware checks if the user is authenticated before the request reaches our application logic.
- If the user is not authentication, then it redirects the user to the login screen, otherwise it lets the request proceed.
- Middleware can be attached to route, e.g.:

```
Route::get('profile', function () {  
    // Only authenticated users may enter...  
})->middleware('auth');
```

So the profile route can only be executed when the user is authenticated. Otherwise, the user is redirected to the login page.

- The authentication Middleware can also be attached to a controller, which require user to be authenticated before any function in this controller can be executed.

The attachment is done at the constructor:

```
public function __construct() {  
    $this->middleware('auth');  
}
```



- We can specify only certain functions in a controller are attached to the middleware:

```
public function __construct() {  
    $this->middleware('auth', ['only'=>['create', 'edit']]);  
}
```

only **create()** and **edit()** are attached to the authentication middleware.

- We can attach all functions to the middleware but specify exceptions:

```
public function __construct() {  
    $this->middleware('auth', ['except'=>['index']]);  
}
```

**index()** can be executed without the user being logged in.

- In **RegisterController.php**, instead of 'auth' middleware, 'guest' middleware is used:

```
public function __construct() {  
    $this->middleware('guest');  
}
```

so that the registration page is only displayed to users that are not logged in.

## More on Middleware

There is a lot more to middleware than what we can cover. Here we outline some other features Laravel's middleware provides. If you need these features, look up Laravel document to figure out how to implement these.

- **Define your own middleware.** *Auth* is one of the predefined middleware that comes with Laravel. You can also define your own middleware for your needs. E.g. you may define a middleware called *admin*, which allows the request through only if the user is an administrator.
- **Group middleware.** Multiple middleware can be grouped together into one middleware. E.g. it may be useful to group the *auth* and *admin* middleware, since *admin* has to be logged in.
- **Apply multiple middleware to a route.**
- **Apply one or more middleware to a group of routes.**

## Manually Authenticating Users

- Instead of using the scaffold, one can build their own login form and manually perform authentication using **Auth::attempt()**.
- To use this function, **Illuminate\Support\Facades\Auth** needs to be imported.
- **Auth::attempt()** takes user's email (or username) and password, and will look up the users table to see if passwords match. If so, it returns true.
- An example of using **Auth::attempt()**:

```
use Illuminate\Support\Facades\Auth;
public function authenticate() {
    if (Auth::attempt(['email' => $email, 'password' => $password])) {
        // Authentication passed...
        return redirect('/home');
    }
}
```

## Password Reset via Email

- Laravel also provides the feature to allow user to reset their password by emailing a reset link to the user.
- For testing purpose, we can simply write the emails to the log instead of sending it out.
- To configure email to be sent to log replace the following line in **.env** file:

```
MAIL_DRIVER=smtpt
```

With:

```
MAIL_DRIVER=log
```

- Then refresh the configuration cache with this command:

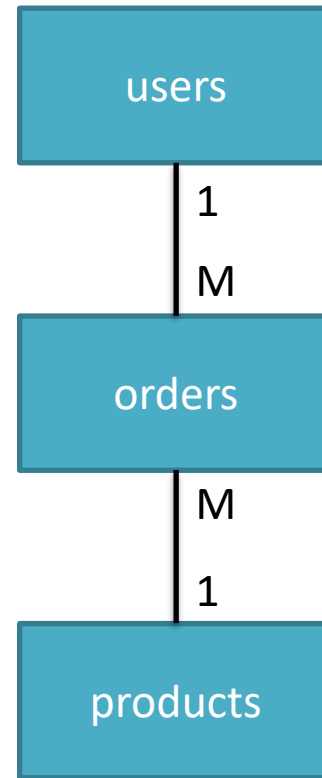
```
php artisan config:cache
```

- From now on, emails sent by this Laravel project will be written to the end of the log file at storage/logs/laravel.log

# Model – Many to many relationships

## Users-Orders-Products Example

- A user can order many products.
- A product can be ordered by many users.
- Therefore the relationship between users and products is many-to-many:



- The following code demonstrates how these tables can be created using the Schema class.

```
Schema::create('users', function (Blueprint $table) {  
    $table->increments('id');  
    $table->string('name');  
    $table->string('email')->unique();  
    $table->string('password');  
    $table->rememberToken();  
    $table->timestamps();  
});
```

- The users table is as provided by Laravel.

```
Schema::create('products', function(Blueprint $table){  
    $table->increments('id');  
    $table->string('name');  
    $table->float('price');  
    $table->integer('manufacturer_id');  
    $table->timestamps();  
});
```

- Products table is unchanged from the previous lecture.

```
Schema::create('orders', function($table) {  
    $table->increments('id');  
    $table->integer('user_id'); // foreign key  
    $table->integer('product_id'); // foreign key  
    $table->integer('quantity');  
    $table->timestamps();  
});
```

- The orders table contains foreign keys to products and users tables, and a quantity field.

### **Many-to-Many Relationships**

- As discussed previously, Eloquent can handle many-to-many relationships for us.
- For this to work the many-to-many table should be named with the singular table names in alphabetical order, e.g. product\_user
- In our example it is more logical to call the table orders.

### **Eloquent Automatic Handling of Many-to-Many Relationships**

- Eloquent can automatically join many-to-many tables for us.
- For this to work we need to create belongsToMany() functions in each of the models, e.g.:

```
class User extends Eloquent {  
    function products() {  
        return $this->belongsToMany('App\Models\Product', 'orders');  
    }  
}
```

- In the above example, because our table is not named `product_user`, we have passed in a second parameter with the table name `orders`, we can also pass in additional parameters listing the column names as well if they don't contain the model names, e.g.:

```
return $this->belongsToMany('App\Models\Product', 'orders', 'user_id',  
    'product_id');
```

- This allows us to query the products that a user has ordered using the dynamic property **products**, e.g:

```
$products = $user->products;
```

- If we want to filter the results further we can use the function **products()**, e.g.:

[illegible]



## Accessing the Pivot table

- Sometimes we want to access columns of the many-to-many table.
- For example orders has a quantity.
- Eloquent refers to the many-to-many table as a pivot table.
- To enable pivot columns to be accessed they need to be defined when the relationship is defined:

```
class User extends Eloquent {  
    function products() {  
        return $this->belongsToMany('App\Models\Product', 'orders')-  
        >withPivot('quantity');  
    }  
}
```

- If you also want the timestamps on the pivot table to be automatically maintained you need to specify this on the relationship as well:

```
class User extends Eloquent {  
    function products() {  
        return $this->belongsToMany('App\Models\Product', 'orders')-  
        >withPivot('quantity')->withTimestamps();  
    }  
}
```

- We can access the pivot table columns through the pivot property.
- For example, the following loops through all products that a user has ordered and also displays the quantity which is stored in the orders table:

```
@foreach ($products as $product)
  {{ $product->name }} {{ $product->pivot->quantity }}
@endforeach
```

- Note that the pivot property is only available on objects that were returned as a result of accessing a related table through one of the model's functions.
- Pivot columns can also be specified when creating a new many-to-many record.
- For example when a user places a new order they also need to specify the quantity, e.g.:

```
$user = User::find(1);
$product = Product::find(1);
$user->products()->attach($product->id, array('quantity' => $quantity));
```

- The above example attaches the product with id 1 to the user with id 1, by creating a new orders table record with the specified quantity.

## Queries on Joined Tables

- If we also include the **manufacturers** table from week 8 as having a one-to-many relationship with **products**, how would we return a list of products which are made by a manufacturer with a certain name?
- Note that we can't simply use **Product::whereRaw()** since the query string can only refer to columns from the Product table.
- Laravel provides a **whereHas()** function to allow you to query a related table.
- The solution to the previous example is:

```
$name = $input['name'];  
$products = Product::whereHas('manufacturer', function($query) use ($name){  
    return $query->whereRaw('name like ?', array("%$name%"));  
})->get();
```

- The code within the function operates on the related table.
- Note that to access a variable outside of the function it must be placed in the **uses()** clause.
- If you need to be able to query both manufacturer and product simultaneously you can add where clauses to the above example, e.g.:

```
$name = $input['name'];  
$products = Product::whereHas('manufacturer', function($query) use ($name){  
    return $query->whereRaw('name like ?', array("%$name%"));  
}) ->whereRaw('name like ?', array("%$name%"))->get();
```

- The above example requires that both product name **AND** manufacturer name match the \$name variable, if you want to perform an **OR** you would need to do the following:

```
$name = $input['name'];  
$products = Product::whereRaw('name like ?', array("%$name%"))  
    ->orWhereHas('manufacturer', function($query) use ($name){  
        return $query->whereRaw('name like ?', array("%$name%"));  
    }->get();
```

- The above example indicate that searching between tables using models can get quite complex.
- In these cases it is probably simpler to construct an SQL query and send it directly to the database. This is discussed at the end of this section.

## Different Relationship Function Names

- Eloquent uses the relationship function name to infer the foreign key name.
- If you want to use a different function name, you must explicitly specify the foreign key name, for example in Product we would have a relationship for **manufacturer()**:

```
function manufacturer(){  
    return $this->belongsTo('Manufacturer');  
}
```

- If we want to call this function **madeBy()** it would need to specify the foreign key:

```
function madeBy() {  
    return $this->belongsTo('Manufacturer', 'manufacturer_id');  
}
```

## Accessing the Database using Query Builder

- Even though Eloquent provides a none-SQL abstraction to access the database, some queries can be more complex when written in pure Eloquent.
- Another way to access database in Laravel is using Query Builder (QB).
  - See this link for details: <https://laravel.com/docs/queries>
  - Eloquent is build on top of QB. So all the QB methods can be used in an Eloquent query.
- QB queries are at a slightly higher abstraction (easier to understand?) than SQL.
- Example: Retrieve a list of products which are made by a manufacturer with a certain name?

```
$name = 'Apple';  
$products = DB::table('manufacturers')  
    ->where('manufacturers.name', '=', $name)  
    ->join('products', 'manufacturers.id', '=', 'products.manufacturer_id')  
    ->get();
```

## Advantage of QB over raw SQL

- Database/SQL independent.
- Handles sanitization automatically, don't need to do binding.
- QB queries are extendable/reusable. E.g.:
  - From the previous example, we could construct a query to retrieve the manufacturers of a given name:

```
$name = 'Apple';  
$query_manuf = DB::table('manufacturers')  
                ->where('manufacturers.name', '=', $name);  
$manufacturers = $query_manuf->get();
```

- We can build on top of `$query_manuf` to retrieve products:

[illegible]

## Accessing the Database via raw SQL

- You could always use raw SQL as we did earlier in this course.
- As a refresher, the `DB::select()` function can be used to send select queries to the database, e.g.:

```
$products = DB::select('select * from products');
```

- The result is an array of records, each record being an object with properties for each result column, e.g.:

```
$name = $products[0]->name;
```