Web Applications in PHP

On this page >

Laravel 10 Cheat Sheet / Summary

This cheat sheet is a quick reference for frequently used information in Laravel projects and acts as a summary of our course. For more in-depth information, we've included the relevant documentation links.

TIP

We have also made a similar page for Livewire with the same purpose, you can find that here: Livewire Cheat Sheet / Summary

If you're using the TALL stack (Tailwind, AlpineJS, Livewire & Laravel), check out these additional resources for the stack's other technologies :

- Tailwind Official Documentation
- Tailwind cheat sheet
- Alpine Official Documentation

General commands

shell

```shell
# Commands list
$ php artisan list

# Command help
$ php artisan help ...

# Clear application cache
php artisan cache:clear

# Clear view cache
php artisan view:clear

# Clear route cache
php artisan route:clear
```

```

# Clear config cache
php artisan config:clear
```

Blade

Blade is Laravel's concise and versatile templating engine, used for crafting dynamic **views, components, and layouts**.

The extension for Blade files is '.blade.php'.

In Laravel views are located in the **resources/views** directory. Regular views are mainly used for building out pages while components and layouts provide a way to create reusable UI elements.

Documentation

Blade - Display Data

blade

```blade
{{-- Display available data by using '{{ }}' --}}
<h1>Welcome, {{ $user->name }}!</h1>

{{-- Use {!! !!} to output without escaping --}}
<div>{!! $htmlOutput !!}</div>
```

Blade - Directives

blade

```blade
{{-- Dump (and die) variables to the browser --}}
@dump($perPage, $newGenre, $genres->toArray())
@dd($perPage, $newGenre, $genres->toArray())

{{-- If statement --}}
{{-- You make comments like this --}}

{{-- If statement --}}
@if($user->isAdmin)
    <p>You have admin privileges.</p>
@else
```

```
    <p>You are a regular user.</p>
@endif

{{-- Switch case --}}
@switch($role)
    @case('admin')
        <p>You are an admin.</p>
        @break
    @case('user')
        <p>You are a user.</p>
        @break
    @default
        <p>Your role is not defined.</p>
@endswitch

{{-- Check if defined and not null --}}
@isset($products)
    ...
@endisset

{{-- Check if empty --}}
@empty($records)
    ...
@endempty

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

{{-- Foreach loop --}}
@foreach($products as $product)
    <li>{{ $product->name }}</li>
@endforeach

{{-- Forelse loop --}}
@forelse($products as $product)
    <li>{{ $product->name }}</li>
@empty
    <p>No products found.</p>
@endforelse

{{-- While loop --}}
```

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

{{-- Determine if a user is authenticated or not.
     Add ('nameOfTheRole') to check for a specific role. --}}
@auth
    <p>You're logged in</p>
@endauth

@guest
    <p>Please login</p>
@endguest

{{-- Add raw php code --}}
@php
    $counter = 1;
@endphp

{{-- Show JSON data --}}
<pre>@json($products, JSON_PRETTY_PRINT)</pre>
```

Blade - Components & Layouts

Components are reusable pieces of your layout. Laravel expects Blade components to be present in the **resources/views/components** directory.

For class based components the classes are automatically discovered in the **app/View/Components** directory.

Layouts are a way to use the same general page layout across various pages. When using Blade components in Laravel, making a layout is done just the same as any other component and relies heavily on slots (it can also be done via template inheritance).

```shell
# Make a class based component
$ php artisan make:component Alert

# Make an anonymous component = No class
$ php artisan make:component Alert --view
```

```

# Create component in subdirectory
$ php artisan make:component Forms/Input
```

```blade
{{-- Render a component, use x- + name in kebab-case, root = components
<x-alert/>

{{-- Use '.' for subdirectories --}}
<x-forms.input/>

{{-- Pass attributes to a component, use ':' for dynamic data --}}
<x-alert type="error" :message="$message" class="mt-4"/>

{{-- Define Data properties in a component --}}
@props(['type' => 'info', 'message'])

{{-- Use attributes in a component --}}
<div {{ $attributes }}>
    <p>{{ $message }}</p>
</div>

{{-- Merge attributes in a component --}}
<div {{ $attributes->merge(['class' => 'alert']) }}>
    <p>My message</p>
</div>

{{-- Use attributes & data properties in a component --}}
<div {{ $attributes->merge(['class' => 'alert alert-'.$type]) }}>
    <p>{{ $message }}</p>
</div>
```

Blade - Slots & Stacks

Slots provide a mechanism for injecting dynamic content into Blade components

```blade
{{-- Define a named slot with $nameOfTheSlot --}}
<span class="alert-title">{{ $title }}</span>

{{-- Define the default slot with $slot --}}
<div class="alert alert-danger">
```

```blade
    {{ $slot }}
</div>

{{-- Pass content to the slots --}}
{{-- Anything between the component tags which is not in a named slot w
<x-alert>
    <x-slot name="title"> {{-- OR <x-slot:title> --}}
        Server Error
    </x-slot>

    <strong>Whoops!</strong> Something went wrong!
</x-alert>
```

A stack can best be described as a 'reference' on the page where you can later add additional code. They are especially useful to add page specific JavaScript.

```blade
{{-- Create a stack (in a component)--}}
@stack('script')

{{-- Push to a stack (when using the component) --}}
@push('script')
    <script>
        console.log('Script is working!')
    </script>
@endpush
```

Routes

Routes in Laravel define how HTTP requests are handled, connecting URLs to actions within your application. Routes can be found in the **routes** directory, for web applications we use the **routes/web.php** file for routing.

Documentation

shell

```shell
# Route list
$ php artisan route:list

# Hide routes defined by third-party packages
$ php artisan route:list --except-vendor
```

php

```php
// Get route
// When user surfs to /shop the shop.blade.php view located in recource
Route::get('shop', function () {
    return view('shop');
});

// Shorter notation when only returning a view
Route::view('shop', 'shop');

// Passing data to a view via routing
// Use an associative array
Route::get('shop/products', function (){
    $products = [ ... ];

    return view('shop.products.overview', [
        'products' => $products
    ]);
});

// Compact function is often used when possible
return view('shop.product.overview', compact('products'));

// Named routes
Route::view('/', 'welcome')->name('home');

// Group routes
// Prefix is added before every route in the group
// Name is added before each named route in the group
Route::prefix('admin')->name('admin.')->group(function () {
    //...
});

// Laravel provides options to respond to any HTTP verb
Route::get($uri, $callback);
Route::post($uri, $callback);
```

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

// Respond to multiple defined HTTP verbs
Route::match(['get', 'post'], '/', function () {
    // ...
});

// Respond to any HTTP verb
Route::any('/', function () {
    // ...
});

// Redirect a route to a different route
Route::redirect('/here', '/there');
```

Migrations

Migrations are mainly used for defining the structure of your database. Migrations are found in the **database/migrations** directory.

Documentation

Migrations - Commands

```shell
# Create a migration (use descriptive name with snake_case)
$ php artisan make:migration create_products_table
$ php artisan make:migration add_price_to_products_table

# Run the new migrations
$ php artisan migrate

# Rollback latest migration
php artisan migrate:rollback

# Rollback all migrations
php artisan migrate:reset

```

```
# Rollback all and re-migrate
php artisan migrate:refresh

# Rollback all, re-migrate and run all seeders
php artisan migrate:refresh --seed
```

Migrations - Syntax

Example of a create migration

```php
// up() --> What to add on migrate
Schema::create('products', function (Blueprint $table) {

    // auto-increment primary key
    $table->id();

    // created_at and updated_at
    $table->timestamps();

    // Unique constraint
    $table->string('modelNumber')->unique();

    // Not required
    $table->text('description')->nullable();

    // Default value
    $table->boolean('isActive')->default(true);

    // Foreign Key relation
    $table->foreignId('user_id')->constrained('users')->onDelete('casca
});

// down() --> What to delete on rollback
Schema::dropIfExists('products');
```

Example of an update migration

```php
// up() --> What to add on migrate
Schema::table('products', function (Blueprint $table) {
    $table->float('price', 5, 2);
});

// down() --> What to delete on rollback
Schema::table('products', function (Blueprint $table) {
    $table->dropColumn('price');
});
```

Eloquent Models

In Laravel, Eloquent Models are PHP classes that streamline interactions with your database tables, making data retrieval, manipulation, and organization much easier. These models reside in the **App/Models** directory.

Documentation

Models - Commands

```shell
# Creating a model :
$ php artisan make:model Product

# Some of the extra options via -flags :
# -m (migration), -f (factory), -s (seed)
$ php artisan make:model Product -mfs
```

Models - Mass assignment

Mass Assignment Protection in Laravel enhances security by enabling you to specify which attributes are allowed for mass assignment , preventing unauthorized or unintended modifications of sensitive data.

```php
// Use $guarded to list all attributes that can NOT be mass assigned
protected $guarded = []; // Empty means all attributes are guarded

// Use $fillable to list all attributes that can be mass assigned
protected $fillable = ['name', 'email', 'password'];
```

WARNING

Never use both at the same time in one specific model, pick one.

Models - Relations

Relations in Laravel define how different Eloquent models are related to each other, allowing you to establish connections and navigate between database tables with ease.

php
```php
// One to Many relationship (users with many orders)
public function orders()
{
    return $this->hasMany(Order::class);
}

// One to Many relationship (orders with one user)
public function user()
{
    return $this->belongsTo(User::class);
}


// One to One: User is the parent in the relationship
public function profile()
{
    return $this->hasOne(Profile::class);
}

// One to One: Profile is the child (contains user_id)
public function user()
{
    return $this->belongsTo(User::class);
}


// Many to Many relationship: Tags associated with multiple Products an
// 'product_tag' is the pivot table connecting 'product_id' and 'tag_id

// In Tag model...
public function products()
```

```php
{
    return $this->belongsToMany(Product::class);
}

// In Product model...
public function tags()
{
    return $this->belongsToMany(Tag::class);
}
```

The function name is important!

Why does this code not work in the **User** model?

```php
public function bestellingen()
{
    return $this->hasMany(Order::class);
}
```

- If you refer to the `Order` model inside the `User` model:
  - the function name should be `orders()` (plural name of the model you refer to)
  - the function assumes that the foreign key is `user_id` (singular name of the model it belongs to + `_id`)
  - the function assumes that the primary key is `id` (default primary key)
- Only if you follow these conventions, you can shorten the default code: `return $this->hasMany(Order::class, 'foreign_key', 'local_key');` to the shorter version: `return $this->hasMany(Order::class);`

Default models using `withDefault()`

You can add the `withDefault()` method in belongsTo and hasOne relationships which will return a new instance of the related model if the relationship is `null` (the foreign key is null).

```php
// Adding a default empty model to avoid conditional checks (Null Object patt
public function user()
{
    return $this->belongsTo(User::class)->withDefault();
}

```

```php
// Passing an array of data to provide a default model
public function user(): BelongsTo
{
    return $this->belongsTo(User::class)->withDefault([
        'name' => 'Guest Author',
        //...
    ]);
}
```

Models - Accessors & Mutators

Accessors in Laravel let you format or modify attribute values when you retrieve them. Mutators enable you to modify attribute values before they are saved to the database.

```php
// Method name is the attribute name in camelCase instead of snake_case
protected function firstName(): Attribute
{
    return Attribute::make(
        get: fn($value) => ucfirst($value),         // accessor
        set: fn($value) => strtolower($value),      // mutator
    );
}

// Add additional attributes that do not have a corresponding column in
// Make use of $attributes as a second argument to access all attribute
protected function userName(): Attribute
{
    return Attribute::make(
        get: fn($value, $attributes) => User::find($attributes['user_id
    );
}

protected function priceEuro(): Attribute
{
    return Attribute::make(
        get: fn($value, $attributes) =>  '€ ' . number_format($attribut
    );
}
```

Arrow functions & Named arguments

It's common to use arrow functions and named paramaters/arguments in accessors and mutators.

Arrow functions :

- `fn(arguments)` is a shorthand notation for `function(arguments)`
- `=>` is a shorthand for `return`
- E.g. `fn($value) => ucfirst($value)` is the same as `function($value) { return ucfirst($value); }`

Named arguments :

Named arguments allow passing arguments to a function based on the parameter name, rather than the parameter position.

```php
function greet($message, $name) {
  echo "$message, $name!";
  }

// Notice we are able to use a different order for the arguments
greet(name: "John", message: "Hello"); // Outputs: "Hello, John!"
```

Models - Scopes

Scope functions in Laravel provide a convenient way to encapsulate reusable query logic within Eloquent models, allowing for cleaner and more maintainable database queries.

```php
// Create a scope function in the Model
public function scopeMaxPrice($query, $price = 100)
{
    return $query->where('price', '<=', $price);
}

// Use the scope query anywhere with the Query Builder
$products = Product::maxPrice(20)->get();
```

Models - CRUD Operations

Eloquent makes it easy to execute CRUD operations on your models.

```php
// Create
$newUser = User::create(['name' => 'John Doe', 'email' => 'john@example

// Read
$users = User::all();
$user = User::find(1);
$activeUsers = User::where('status', 'active')->get();
$tags = Product::where('name', 'Some Product')->first()->tags; // Becau

// Update
$user = User::find(1);
$user->update(['name' => 'Updated Name']);
User::where('status', 'inactive')->update(['status' => 'active']);

// Delete
$user = User::find(1);
$user->delete();
User::where('status', 'inactive')->delete();
```

Query Builder

Laravel Query Builder simplifies database queries with a clean, expressive API. It provides an easy-to-understand way to build complex SQL queries without writing raw SQL statements.

Documentation

```php
// Selecting Data
$users = DB::table('users')
    ->select('name', 'email')
    ->where('name', 'like', "%vince%")
    ->orWhere('name', 'like', "%patrick%")
    ->where('active', true)
    ->orderBy('created_at', 'desc')
    ->get();

// Joining Tables and select specific Columns with conditions
```

```
$users = DB::table('users')
    ->join('orders', 'users.id', '=', 'orders.user_id')
    ->select('users.id as user_id', 'users.name as user_name', 'orders.
    ->where('orders.status', '=', 'completed')
    ->where('users.active', '=', true)
    ->orderBy('orders.created_at', 'desc')
    ->get();


// Aggregates
$totalOrders = DB::table('orders')->count();
$averagePrice = DB::table('products')->avg('price');

// Insert, Update, Delete
DB::table('users')->insert(['name' => 'John Doe', 'email' => 'john@exam
DB::table('users')->where('id', 1)->update(['name' => 'Updated Name']);
DB::table('users')->where('id', 1)->delete();
```

Eloquent ORM vs Query Builder

If you've looked at previous sections, you'll notice that the syntax for performing basic CRUD operations with Eloquent Models is very similar to that of the Query Builder. However, there is indeed a difference, and both have their use cases.

In short, Eloquent Models provide a simpler way to interact with data, especially when dealing with relationships. On the other hand, the Query Builder is often considered more performant in certain scenarios due to its direct manipulation of SQL queries and fine-grained control.

A common rule of thumb is to use Eloquent Models in most places to make your life easier and the Query Builder for larger amounts of data. In the end you as the developer decide what works best for you and your project.

php

```php
// Getting all tags for a product what a certain name (many-to-many relation:

// Using Eloquent ORM
$tags = Product::where('name', 'Some Product')->first()->tags;

// Using Query Builder
$tags = DB::table('products')
            ->where('name', 'Some Product')
            ->join('product_tags', 'products.id', '=', 'product_tags.product_
            ->join('tags', 'product_tags.tag_id', '=', 'tags.id')
            ->select('tags.*')
            ->get();
```

Route Model Binding

Route Model Binding is a feature in Laravel that saves you from writing boilerplate code to query the database.

Documentation

When you type-hint a model in an action, Laravel does the work for you:

- It takes the ID from the request.
- It searches the corresponding model in the database.
- It passes the model instance to your action.

php

```php
// The id of a product gets passed as the argument
// Laravel searches for the model based on this id because you type-hin
public function dumpProduct(Product $product)
    {
        var_dump($product) // This will show the entire product, not ju
    }
```

HTTP Client

Laravel simplifies outbound HTTP requests using Guzzle through its Http facade. This facade offers methods like `head` , `get` , `post` , `put` , `patch` , and `delete` for quick and easy communication with other web applications.

In our course we only use simple `get` requests (like the example below) but the facade is capable of a lot more.

Documentation

php
```php
// Import the HTTP facade
use Http

// Do a get request to the url
$response = Http::get('https://api.quotable.io/random');

// Get the json data in the response
$result = $response->json();

// Check if the request was succesful
if ($response->successful()) {

    // Get specific data from the json result
    $author = $result['author'];
    $quote = $result['content'];

} else {
    // Use the response status if request was not succesful
    $error = "ERROR {$response->status()}";
}
```

Storage

The Storage facade in Laravel provides a simplified interface for efficient file and storage management. From storing and retrieving files to directory operations, this tool offers a clean and consistent API.

Documentation

```php
// Store a file in the default disk (usually 'public')
Storage::put('file.txt', 'Hello, Laravel!');

// Get the contents of a file
$contents = Storage::get('file.txt');

// Delete a file
Storage::delete('file.txt');

// Check if a file exists
if (Storage::exists('file.txt')) {
    // File exists
}

// Create a directory
Storage::makeDirectory('images');

// Delete a directory
Storage::deleteDirectory('images');
```

Authentication

Laravel simplifies the implementation of authentication, providing robust, out-of-the-box solutions for verifying user credentials and protecting routes. It seamlessly integrates with various authentication services and offers a flexible approach to manage user access.

Documentation

TIP

As we are using the Jetstream starter kit in the course, a full authentication system is provided including views. You can customize these views as they are found, like all other views, in the **resources/views** directory.

You can make use of the **auth()** helper function or **Auth** facade where desired (controllers, middleware, blade views,...). These provide lots of functionality (see documentation), below are two simple yet very convenient examples.

```php
// Get the currently authenticated user
$user = auth()->user();             // or Auth::user();

// Get one attribute off the currently authenticated user (e.g. name, e
$name = auth()->user()->name;       // or Auth::user()->name;
```

The **@auth** and **@guest** directives can also be used to quickly determine if the current user is authenticated or is a guest:

```blade
@guest
    // only visible for guests and NOT for authenticated users (= not l
@endguest

@auth
    // only visible for authenticated users and NOT for guests (= logge
@endauth
```

Middleware

Middleware is a key tool for intercepting and handling HTTP requests and responses within your application. It provides a powerful way to apply various operations like authentication and logging, either globally or on a per-route basis.

Middleware are located in the **app/Http/Middleware directory**

Documentation

```shell
# Create your own middleware (called 'ActiveUser')
$ php artisan make:middleware ActiveUser
```

Below is an example of a simple custom middleware to check if the user is active (according to the database record).

```php
// Handle the request
public function handle(Request $request, Closure $next): Response
    {
        // Check if logged in user is active
        if (auth()->user()->active) {
            // Send the user to the requested route
            return $next($request);
        }
        // Log the user out
        auth('web')->logout();
        // Abort the request with a message
        return abort(403, 'Your account is not active. Please contact t
    }
```

You can register your custom middleware for easier usage in **app/Http/Kernel.php**.

```php
protected $middlewareAliases = [
    'auth' => \App\Http\Middleware\Authenticate::class,
    ...
    'active' => \App\Http\Middleware\ActiveUser::class,
];
```

Use your middleware on routes (for example in **routes/web.php**) which will cause the logic to be executed everytime a request is made to that route. Beware of the order you use when assigning middleware, this is also the order of execution.

```php
// When registered you can use middleware like this.
Route::get('/profile', function () {
    // ...
})->middleware(['auth', 'active']);

// Alternative way is to use it like this (also possible when it's not
Route::get('/profile', function () {
    // ...
})->middleware(Authenticate::class, Active::class);
```

Carbon (time)

Carbon is an easy-to-use PHP extension for date/time manipulation and formatting. It is installed by default in Laravel.

Documentation

```php
// Import Carbon
use Carbon\Carbon;

// Creating instances
$now = Carbon::now(); // Current date and time
$today = Carbon::today(); // Today's date at midnight
$custom = Carbon::create(2023, 11, 9, 14, 30, 0); // Custom date and ti

// Parsing (create instance from string)
$parsedDate = Carbon::parse('2023-11-09 14:30:00');

// Formatting
$now->format('Y-m-d H:i:s'); // 2023-11-09 15:45:00

// Manipulation
$nextWeek = $now->addWeek(); // Add one week
$yesterday = $now->subDay(); // Subtract one day

// Comparison
if ($now->gt($custom)) {
    echo 'The current time is greater than the custom time.';
}

// Localization
setlocale(LC_TIME, 'nl_NL'); // Set Dutch locale
$now->formatLocalized('%A %d %B %Y'); // "donderdag 09 november 2023"
```

Seeding (Optional)

Seeding automates the process of populating your database with initial data, making it useful for testing and database setup. Seeders can be found in the **database/seeders** directory.

Documentation

Seeding - Commands

```shell
# Create a seeder
$ php artisan make:seeder ProductSeeder
```

```

# Run the seeders following the order defined in database/seeders/Datab
$ php artisan db:seed

# Run a specific seeder
$ php artisan db:seed --class=ProductSeeder
```

Seeding - Syntax

Make use of the Query Builder and/or Factories in your seeder.

```php
public function run() {
    // Using a Factory to generate random data (using Faker library)
    Product::factory(10)->create();

    // Using a Factory to add fixed data
    Product::factory()->create([
        ['name' => 'necklace', 'price' => 19.99],
        ['name' => 'bracelet', 'price' => 12.99],
    ]);

    // Using the query builder to add fixed data
    DB::table('products')->insert(
        [
            ['name' => 'necklace', 'price' => 19.99],
            ['name' => 'bracelet', 'price' => 12.99],
        ]
    );
}
```

Define the order of the general seed command `php artisan db:seed` in `database/seeders/DatabaseSeeder` .

```php
public function run(): void
    {
        $this->call([
            UserSeeder::class,
            ProductSeeder::class,
            TagSeeder::class,
        ]);
    }
```

Factories (Optional)

Factories in Laravel simplify the creation of model instances with fake data (using the Faker Library), streamlining the testing and seeding processes. Factories can be found in the **database/factories** directory.

Documentation

```shell
$ php artisan make:factory ProductFactory
```

```php
public function definition() {
    return [
        'name' => $this->faker->text(20),
        'price' => $this->faker->numberBetween(10, 10000),
    ];
}
```

Utility functions

asset()

The asset() function in Laravel simplifies the generation of asset URLs, making it easy to reference stylesheets, JavaScript files, and images in your views.

```blade
{{-- Include a stylesheet using the asset() function --}}
<link rel="stylesheet" href="{{ asset('css/style.css') }}">
```

urlencode()

Laravel's urlencode() function is a helpful tool for encoding strings to ensure safe usage in URLs.

```php
// Create a URL with an encoded search query
$url = 'https://example.com/search?q=' . urlencode('Laravel Tutorial');
```

request()

The request() function in Laravel simplifies HTTP request interactions by providing convenient methods to retrieve query parameters, form inputs, and check current route patterns.

```php
// Retrieve the value of the 'q' query parameter from the current URL.
$searchQuery = request('q');

// Retrieve the value of the 'username' input field from a submitted fo
$username = request('username');

// Check if the current route matches the named route 'profile'.
if (request()->routeIs('profile')) { ... }

// Check if the current route matches the nested route 'admin/dashboard
if(request()->routeIs('admin.dashboard');

// Check if the current route falls under the 'admin' namespace.
if (request()->routeIs('admin.*')) { ... }

// Check if the current route matches 'home' or 'about'.
if (request()->routeIs(['home', 'about'])) { ... }
```

Last updated: 04/12/23, 4:40 pm