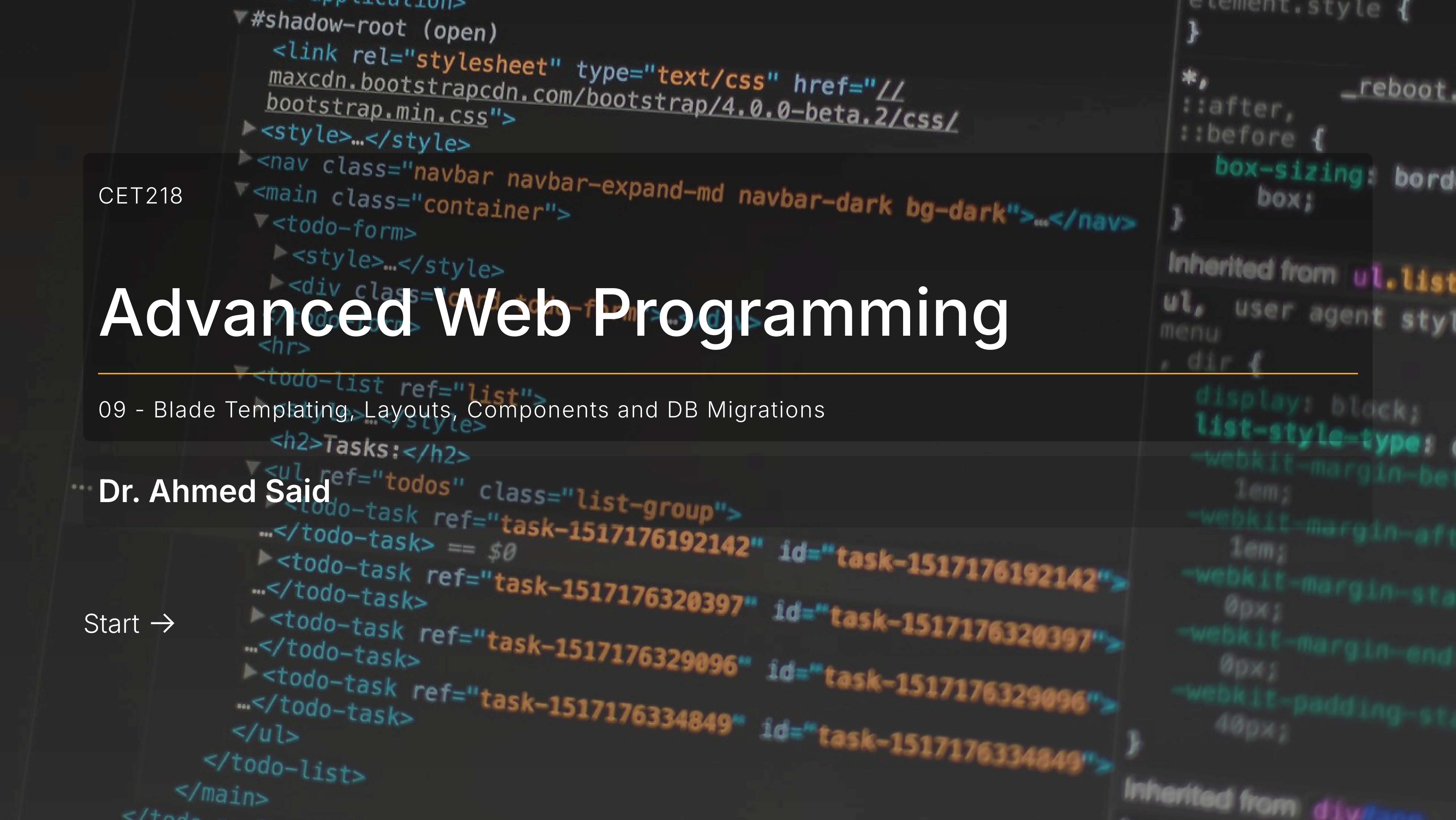

CET218

Advanced Web Programming

09 - Blade Templating, Layouts, Components and DB Migrations

Dr. Ahmed Said

Start →

Introduction to Blade Templating

- **Blade** is Laravel's powerful, simple templating engine.
- Allows you to write clean, readable templates with minimal PHP code.
- Blade templates are compiled into plain PHP and cached for performance.

Why Use Blade?

- **Separation of concerns:** Keeps logic out of your HTML.
- **Reusable components:** Layouts, includes, and components.
- **Easy syntax:** Shortcuts for control structures and data output.
- **Security:** Automatic escaping of output.

Blade Template Files

- Blade templates use the `.blade.php` extension.
- Stored in the `resources/views` directory.

Example:

```text
resources/views/welcome.blade.php
```

- You can create subdirectories for better organization.

Example:

```text
resources/views/users/profile.blade.php
```

- Laravel automatically compiles Blade templates into PHP.
- No need to worry about caching; Laravel handles it for you.
- You can use Blade templates in any Laravel view.

Blade Syntax: Outputting Data

- Use double curly braces to echo data (escaped by default):
- escaped means that HTML tags will be converted to plain text.
- This prevents XSS attacks.

```blade
{{ $name }}
```

- To output unescaped data, use:

```blade
{!! $html !!}
```

- `!!` is used for raw HTML output (be cautious with this).

Blade Syntax: PHP Code

- Blade also supports PHP code within curly braces:

```blade
{{ $user->name }}
```

- You can use Blade's `@php` directive for complex logic:

```blade
@php
  $total = $item->price * $item->quantity;
@endphp
<p>Total: {{ $total }}</p>
```

Blade Syntax: Comments

- Blade comments are not included in the compiled HTML:

```blade
{{-- This is a comment --}}
```

- Regular PHP comments will be included in the compiled HTML:

```blade
<?php
  // This is a PHP comment
?>
```

- Use Blade comments to leave notes in your templates without affecting the output.
- Regular PHP comments will be visible in the compiled HTML, which may not be desirable.
- Use Blade comments for notes and explanations in your templates.

Blade Syntax: Control Structures

- Blade provides simple directives for common PHP structures.

If Statement:

```blade
@if ($user)
  Hello, {{ $user->name }}
@else
  Welcome, guest!
@endif
```

- `unless` directive is used to check if a condition is false.

  ```blade
  @unless ($user)
    Welcome, guest!
  @endunless
  ```

  - It is the opposite of the if directive.
  - It is useful for negating conditions.
  - It is a shorthand for if (!condition).

Blade Syntax: Loops

- Blade provides directives for loops.
- You can use `@foreach`, `@for`, and `@while` to iterate over data.
- These directives are similar to their PHP counterparts but have a cleaner syntax.

Loops:

```blade
@for ($i = 0; $i < 10; $i++)
  <li>{{ $i }}</li>
@endfor
```

```blade
@while ($count < 5)
  <li>{{ $count }}</li>
  @php $count++; @endphp
@endwhile
```

```blade
@foreach ($users as $user)
  <li>{{ $user->name }}</li>
@endforeach
```

Blade Syntax: More Foreach

- You can also access the key of the item in a foreach loop.

```blade
@foreach ($users as $key => $user)
  <li>{{ $key }}: {{ $user->name }}</li>
@endforeach
```

@forelse

- You can also use `@forelse` to handle empty collections.

  ```blade
  @forelse ($users as $user)`
    <li>{{ $user->name }}</li>
  @empty
      <li>No users found.</li>
  @endforelse
  ```

  - It is a shorthand for foreach with an else clause.
  - It is useful for displaying a message when the collection is empty.
  - It is a combination of foreach and if.

Blade Layouts

Blade Layouts: Extending Templates

- Use layouts to **avoid repeating code** (like headers/footers).
- Since most web applications maintain the same general layout across various pages, it’s convenient to define this layout as a single Blade view:
- Create a base layout file (e.g., `layouts/app.blade.php`), This file will contain the common structure of your pages.

Define a layout:

- use `@yield` to define sections that child can fill.

```blade
<!-- resources/views/layouts/app.blade.php -->
<html>
  <body>
    <header> </header>
    @yield('content')
    <footer> </footer>
  </body>
</html>
```

Using the layout:

- Create a child view that extends the layout:

```blade
@extends('layouts.app')

@section('content')
  <h1>Welcome</h1>
@endsection
```

Blade Layouts: Defining Layout

- You can define multiple sections in a layout.

```blade
<!-- resources/views/layouts/app.blade.php -->
<html>
    <head>
        <title>App Name - @yield('title')</title>
    </head>
    <body>
        @section('sidebar')
            This is the master sidebar.
        @show
        <div class="container">
            @yield('content')
        </div>
    </body>
</html>
```

- Take note of the `@section` and `@yield` directives.
- The `@section` directive, as the name implies, defines a section of content, while the `@yield` directive is used to display the contents of a given section.

Blade Layouts: Using Layouts

- Now that we have defined a layout for our application, let's define a child page that inherits the layout.
- When defining a child view, use the `@extends` Blade directive to specify which layout the child view should "inherit".
  - Views which extend a Blade layout may inject content into the layout's sections using `@section` directives.
  - Remember, the contents of these sections will be displayed in the layout using `@yield`:

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

@section('title', 'Page Title')

@section('sidebar')
    @parent
    <p>This is appended to the master sidebar.</p>
@endsection

@section('content')
    <p>This is my body content.</p>
@endsection
```

Blade Layouts: Using Layouts

- In previous example,
  - The sidebar section is utilizing the `@@parent` directive to append (rather than overwriting) content to the layout’s sidebar.
  - The `@@parent` directive will be replaced by the content of the layout when the view is rendered.
  - The `@extends` directive tells Blade to use the specified layout.
  - The `@section` directive defines a section of content that will be injected into the layout.
  - The `@endsection` directive ends the section.
  - The `@section('title', 'Page Title')` sets the title of the page.
- The `@yield` directive also accepts a **default value** as its second parameter. This value will be rendered if the section being yielded is undefined:

```blade
@yield('content', 'Default content')
```

Blade Layouts: Using Layouts

- Use `@show` to display the content and continue rendering the layout.

```blade
@section('content')
  <h1>Welcome</h1>
  @show
  <p>This is additional content.</p>
@endsection
```

- The `@stop` directive will stop rendering the layout and return to the view.

```blade
@section('content')
  <h1>Welcome</h1>
  @stop
  <p>This is additional content.</p>
```

- The `@overwrite` directive will overwrite the content of the section.

```blade
@section('content')
  <h1>Welcome</h1>
  @overwrite
  <p>This is additional content.</p>
```

Blade Stacks

- Blade allows you to push to named **stacks** which can be rendered somewhere else in another view or layout.
- This can be particularly useful for specifying any JavaScript libraries required by
- Use `@push` and `@stack` to manage stacks of content.

```blade
@push('scripts')
  <script src="app.js"></script>
@endpush
```

- You can render the stack in your layout:

```blade
<head>
    <!-- Head Contents -->
    @stack('scripts')
</head>
```

Blade Stacks, cont`d

- You can also use `@prepend` to add content to the beginning of a stack:

```blade
@push('scripts')
    This will be second...
@endpush

// Later...

@prepend('scripts')
    This will be first...
    <script src="jquery.js"></script>
@endprepend
```

- If you would like to `@push` content if a given boolean expression evaluates to true, you may use the `@pushIf` directive:

```blade
@pushIf($shouldPush, 'scripts')
    <script src="/example.js"></script>
@endPushIf
```

Blade Partials

- **Partials** are reusable pieces of Blade templates.
- They help you avoid code duplication.
- **Partials** are typically used for small sections of HTML that are reused across multiple views.
- Create a partial view file (e.g., `partials/header.blade.php`).

```blade
<!-- resources/views/partials/header.blade.php -->
<header>
  <h1>My Website</h1>
    <nav>
        <ul>
        <li><a href="/">Home</a></li>
        <li><a href="/about">About</a></li>
        </ul>
    </nav>
</header>
```

Blade Partials, cont`d

- Use `@include` to include a partial view in another view.

```blade
<!-- resources/views/welcome.blade.php -->
@extends('layouts.app')
@section('content')
  @include('partials.header')
  <h1>Welcome to My Website</h1>
@endsection
```

- Pass data to includes:

```blade
@include('partials.user', ['user' => $user])
```

Blade Components

Blade Components

- **Components** are reusable, self-contained pieces of UI.
- **Components** and **slots** provide similar benefits to sections, layouts, and includes; however, some may find the mental model of components and slots easier to understand.
- There are two approaches to writing components: class based Components and Anonymous Components.
- **Class based Components** are more powerful and flexible, while **Anonymous Components** are simpler and easier to use.
- To create a class based component, you may use the `make:component` Artisan command.

```bash
php artisan make:component Alert
```

- This will create a new component class in the `app/View/Components` directory and a Blade view in the

Blade Components: Define a component

```php
// app/View/Components/Alert.php
namespace App\View\Components;
use Illuminate\View\Component;
class Alert extends Component
{
    public function render()
    {
        return view('components.alert');
    }
}
```

```blade
<!-- /resources/views/components/alert.blade.php -->

<div class="alert alert-danger">
    {{ $slot }}
</div>
```

- The `slot` variable is used to pass content to the component.

Blade Components: Using a component

Use a component:

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

- The `x-` prefix is used to indicate that this is a Blade component.
- You can use the component like a regular HTML tag.

Blade Components, cont`d

- You can also pass data to components using attributes:
- You can define attributes in the component class.
- You can use them in the component view.

```blade
<!-- /resources/views/components/alert.blade.php -->
<span class="alert-title">{{ $title }}</span>

<div class="alert alert-{{ $type }}">
    {{ $slot }}
</div>
```

Blade Components, cont`d

```php
// app/View/Components/Alert.php
namespace App\View\Components;
use Illuminate\View\Component;
class Alert extends Component
{
    public $type;
    public $title;

    public function __construct($type = 'info', $title = 'Alert')
    {
        $this->type = $type;
        $this->title = $title;
    }

    public function render()
    {
        return view('components.alert');
    }
}
```

Blade Components, cont`d

- You can use the component like this:

```blade
<x-alert type="danger" title="Error">
    <strong>Whoops!</strong> Something went wrong!
</x-alert>
```

- Or

```blade
<x-alert>
    <x-slot:title>
        Server Error
    </x-slot>

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

Blade Components, cont`d

- **Anonymous components** are simpler and easier to use.
- You can also use the `@component` directive to create components without creating a class.

```blade
{--
@component('components.alert', ['type' => 'danger'])
    <strong>Whoops!</strong> Something went wrong!
@endcomponent
```

Blade Directives

- Blade provides many helpful directives:
  - `@if`, `@foreach`, `@for`, `@while`
  - `@include`, `@extends`, `@section`, `@yield`
  - `@csrf`, `@auth`, `@guest`, etc.
- `csrf` directive generates a CSRF token for forms.
- `auth` directive checks if the user is authenticated.

Custom Directives:

- You can define your own with `Blade::directive()` in a service provider.

Blade and Security

- By default, `{{ }}` escapes output to prevent XSS.
- Use `{!! !!}` only for trusted HTML.
- Always validate and sanitize user input.

Database Migrations & ORM

Database Migrations

- **Migrations** is the Laravel's way of managing database schema changes.
- Version control for your database schema.

Database Migrations

- **Migrations** is the Laravel's way of managing database schema changes.
- Version control for your database schema.

Why Use Migrations?

- **Team Collaboration:** Keep everyone's database schema synchronized.
- **Version Control:** Track schema changes alongside your code (Git).
- **Easy Setup:** Quickly set up the database schema for new developers or environments.
- **Rollback:** Easily revert schema changes if needed.
- **Database Agnostic (Mostly):** Write schema definitions once, run on different DB systems (MySQL, PgSQL, SQLite).

Migrations: Creating a Migration

- **Migrations** are PHP classes that define the structure of your database tables.
- Creates a new file in `database/migrations/`. The filename includes a timestamp.

Migrations: Creating a Migration

- **Migrations** are PHP classes that define the structure of your database tables.
- Creates a new file in `database/migrations/`. The filename includes a timestamp.
- **Creating a Migration**
  - Use the Artisan command:

```bash
# Create a migration to create the 'products' table
php artisan make:migration create_products_table

# Create a migration to add a 'price' column to 'products' table
php artisan make:migration add_price_to_products_table --table=products
```

Migrations: Defining Schema

Inside the migration file, use the `Schema` facade.

```php
<?php
use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

return new class extends Migration {
    /** Run the migrations. (Creates the 'products' table.) */
    public function up(): void {
        Schema::create('products', function (Blueprint $table) {
            $table->id(); // Auto-incrementing BigInt primary key 'id'
            $table->string('name'); // VARCHAR
            $table->text('description')->nullable(); // TEXT, allows NULL
            $table->decimal('price', 8, 2)->default(0.00); // DECIMAL(8, 2)
            $table->unsignedInteger('stock')->default(0); // Positive INT
            $table->boolean('is_active')->default(true); // TINYINT(1) or BOOLEAN
            $table->timestamps(); // Adds `created_at` and `updated_at` TIMESTAMP columns
        });
    }

    /* Reverse the migrations (Drops the 'products' table). */
    public function down(): void {
        Schema::dropIfExists('products');
```

Migrations: Defining Schema

Inside the migration file, use the `Schema` facade.

```php
<?php
use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

return new class extends Migration {
    /** Run the migrations. (Creates the 'products' table.) */
    public function up(): void {
        Schema::create('products', function (Blueprint $table) {
            $table->id(); // Auto-incrementing BigInt primary key 'id'
            $table->string('name'); // VARCHAR
            $table->text('description')->nullable(); // TEXT, allows NULL
            $table->decimal('price', 8, 2)->default(0.00); // DECIMAL(8, 2)
            $table->unsignedInteger('stock')->default(0); // Positive INT
            $table->boolean('is_active')->default(true); // TINYINT(1) or BOOLEAN
            $table->timestamps(); // Adds `created_at` and `updated_at` TIMESTAMP columns
        });
    }

    /* Reverse the migrations (Drops the 'products' table). */
    public function down(): void {
        Schema::dropIfExists('products');
```

Migrations: Defining Schema

Inside the migration file, use the `Schema` facade.

```php
<?php
use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

return new class extends Migration {
    /** Run the migrations. (Creates the 'products' table.) */
    public function up(): void {
        Schema::create('products', function (Blueprint $table) {
            $table->id(); // Auto-incrementing BigInt primary key 'id'
            $table->string('name'); // VARCHAR
            $table->text('description')->nullable(); // TEXT, allows NULL
            $table->decimal('price', 8, 2)->default(0.00); // DECIMAL(8, 2)
            $table->unsignedInteger('stock')->default(0); // Positive INT
            $table->boolean('is_active')->default(true); // TINYINT(1) or BOOLEAN
            $table->timestamps(); // Adds `created_at` and `updated_at` TIMESTAMP columns
        });
    }

    /* Reverse the migrations (Drops the 'products' table). */
    public function down(): void {
        Schema::dropIfExists('products');
```

Migrations: Migration File Structure

- `up()` method: Defines the changes to apply (create table, add column).
- `down()` method: Defines how to reverse the changes made in `up()`.
- `Schema::create()`: Creates a new table.
- `Schema::table()`: Modifies an existing table.
- `Schema::renameTable()`: Renames a table.
- `Schema::dropIfExists()`: Drops a table if it exists.
- `Schema::dropColumn()`: Drops a column from a table.
- `Schema::renameColumn()`: Renames a column in a table.
- `Schema::enableForeignKeyConstraints()`: Enables foreign key constraints.

Migrations: Column Types

- `Blueprint $table`: Object used to define table columns and indexes.
  - See Laravel Docs for all column types.
  - Common column types:
    - `string()`: VARCHAR
    - `text()`: TEXT
    - `integer()`: INT
    - `bigInteger()`: BIGINT
    - `decimal()`: DECIMAL
    - `boolean()`: TINYINT(1)
    - `dateTime()`: DATETIME
    - `timestamps()`: Adds `created_at` and `updated_at` columns.

Migrations: Running Migrations

- Use Artisan commands to manage your schema.

Migrations: Running Migrations

- Use Artisan commands to manage your schema.

First: Configure Database Connection

Make sure your `.env` file has the correct database credentials (`DB_CONNECTION`, `DB_HOST`, `DB_DATABASE`, `DB_USERNAME`, `DB_PASSWORD`).

```dotenv
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=my_laravel_app # Make sure this database exists!
DB_USERNAME=root
DB_PASSWORD=your_password
```

Migrations: Artisan Migration Commands

```bash
# Run all pending migrations
php artisan migrate

# Rollback the last batch of migrations
php artisan migrate:rollback

# Rollback a specific number of batches
php artisan migrate:rollback --step=3

# Rollback all migrations
php artisan migrate:reset

# Rollback all migrations and run them again (useful for development)
php artisan migrate:refresh

# Drop all tables and run migrations again (faster, destructive!)
php artisan migrate:fresh

# Drop all tables, run migrations, and run seeders
php artisan migrate:fresh --seed

# Check the status of migrations
php artisan migrate:status
```

Migrations: Example

```php
// database/migrations/2023_10_01_000000_create_products_table.php
use Illuminate\Database\Migrations\Migration;
...
return new class extends Migration {
    public function up(): void {
        Schema::create('products', function (Blueprint $table) {
            $table->id();
            $table->string('name');
            $table->text('description')->nullable();
            $table->boolean('is_active')->default(true);
            $table->timestamps();
        });
    }

    public function down(): void {
        Schema::dropIfExists('products');
    }
};
```

Migrations: Example

```php
// another migration file
// database/migrations/2023_10_01_000001_add_price_to_products_table.php
use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;
return new class extends Migration {
    public function up(): void {
        Schema::table('products', function (Blueprint $table) {
            $table->decimal('price', 8, 2)->default(0.00)->after('description');
        });
    }

    public function down(): void {
        Schema::table('products', function (Blueprint $table) {
            $table->dropColumn('price');
        });
    }
};
```

- Run the migration

```bash
php artisan migrate
```

Thank You!